Design and applications of non-intrusive interceptors

Citation for published version (APA):

Document status and date:
Published: 25/10/2018

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: 10. Dec. 2019
Design and application of non-intrusive interceptors

Qing Cai
September 2018
Design and applications of non-intrusive interceptors

Qing Cai

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

The design described in this report has been carried out in accordance with the TU/e Code of Scientific Conduct.
Abstract
At ASML, there is an increasing interest in dynamically extending the production software for development purposes, for example, hardware simulation, tracing and protocol observing. The extended software exist as libraries that can be dynamically linked and will not be shipped to customers. One of the solution direction is interceptors. This report describes the investigation of the interceptor technique within the ASML context. The result shows it is a promising solution.

Keywords
PDEng, TU Eindhoven, ASML, Interceptor, Testing, Simulation, Plugins, Dynamic Linking

Preferred reference
Qing Cai, Design and applications of non-intrusive interceptors: Eindhoven University of Technology, Stan Ackermans Institute, September 2018. (2018/047)
Foreword

Within ASML there is a great interest in testing our TWINSCAN software, without adding any test specific code to it (i.e. being non-intrusive). A generic pattern has been defined to intercept and observe data flowing through the parameters of an arbitrary function. However the applicability of this pattern still needed to be determined. That was the main goal of this assignment.

Qing Cai did a great job in verifying that the pattern can be applied over different operating systems (Solaris/Linux/VxWorks). He has also shown that all client and server side interactions can be intercepted. Even an integration within the ASML development has been done, to show that it can be used in practice. All of his experiments have been archived and documented in detail such that ASML developers can easily learn from or take over his results.

Qing has quickly learned the ASML way of software development, which is quite an achievement as it has a high learning curve. As more stakeholders were interested in his results he included their requirements where possible.

Will Denissen (Project Mentor)
September 17, 2018
Acknowledgements

This project is supported by many people. Without them, the project would have been impossible.

To begin with, I would like to thank Theo Baan (Senior Software Designer, ASML). We were in the same office and he sat just behind me. When I had trouble with ASML tooling such as ClearCase or devbench, he always kindly helped me with his years of experience and expertise.

Second, I would like to thank Nontas Rontogiannis (Software Designer, ASML). Nontas is my senior OOTI fellow. He is very interested in my project and shows great enthusiasm about technology. He is also the man-to-go when I have OOTI related questions because of his approachable and friendly personality.

I also would like to thank Jinfeng Huang (Senior Software Architect, ASML). Jinfeng is a stakeholder of my project and he expressed great interests. He helped me to integrate the interceptor into a production component. Without his help, it would not be possible to discover the integration issues and solutions. He also helped me understand Dutch company culture and gave me useful tips about work and life in the Netherlands.

My project would not have succeeded without the help from my ASML supervisor Will Denissen (Senior Software Architect, ASML). He oriented me with the concept of interceptor and hardware probe. Whenever there is deviation from the correct solution path, he would point it out. He influenced me a lot about SUT isolation and testability in software engineering. He also gave many valuable suggestions to this report.

Last but not least, I would thank my university supervisors: Alexander Serebrenik and Tom Verhoeff. Alexander gave me a lot of good advices on how to kick off the project. Tom gave me valuable advices on carrying out the project as well as feedback on reports.

Qing Cai
Veldhoven – September 10, 2018
Executive Summary

To alter or augment the behavior of TWINSCAN software, it can be done by changing production code, but in some situations, it can also be done without. By intercepting function calls and events passed through software components, production library functions can be modified to some extent without touching production code. The intercepting libraries are called interceptors. This is interesting and can be used for many purposes, including debugging, tracing and extending functionality. This project is intended to investigate the applicability of interceptors within the ASML context.

Three applications of interceptors are discovered in the stakeholder interview phase, which are hardware probe, non-intrusive logger and protocol checker. Requirements were gathered and then refined. In the feasibility analysis phase, the technical part was closely investigated, including the support for various operating systems and the cyclic reference issue. The performance degradation, information leak and mitigation are also discussed.

The design phase covers the problemsimmerged when generating and deploying interceptors. It is decided to modify ddgen in order to generate interceptors. The building and deploying are in line with standard ASML guideline.

The implementation was conducted on a production software. In this phase, ddgen is modified to generate correct class stubs and interceptors. However, an unexpected case of cyclic dependency between the modified class stub and legacy code was discovered. A solution was found together with a domain expert. A future guideline for all ASML software should be in place to comply with interceptor requirements.

As a result, the project shows that interceptors provide a great way to decouple non-essential development software from production software. Three applications, which are hardware probe, non-intrusive logger and protocol checker, can very well fit into the interceptor paradigm. The performance degradation is negligible with early symbol binding technique. However, the introduction of interceptors may cause minor compatibility issues, and expose certain amount of function names to customers. It is recommended to apply interceptors on a few more TWINSCAN software, before pushing to all software components.
Preface

This report documents the graduation project of Qing Cai, conducted at ASML from January 2018 to October 2018. The project is required for him to graduate from the Software Technology (ST) Postgraduate Doctorate in Engineering (PDEng) program of the Eindhoven University of Technology (TU/e). In order to graduate, each trainee must complete three team projects, and, in the end, a personal graduation project. Qing’s personal graduation project is sponsored by ASML Netherlands, a company that builds photo-lithography machines.

The graduation project mainly has two goals. The first goal is to evaluate the trainee’s development as a software designer, with regards to the ST PDEng standard. The second goal, is to investigate the applicability of non-intrusive interceptors within the ASML context.

This report gives a full research on the interceptor technique, presents the problems it solves, and describes a design and implementation. At last, the results and evaluation of the proposed solution are given.

This report is mostly intended for readers with a technical background.

Qing Cai
Veldhoven – September 10, 2018
Table of Contents

Foreword .................................................................................................i
Acknowledgements ..................................................................................iii
Executive Summary ....................................................................................v
Preface ......................................................................................................vi
Table of Contents .......................................................................................vii
List of Figures ...........................................................................................xi
List of Tables .............................................................................................xiii

1. Introduction .........................................................................................1
   1.1 ASML ..............................................................................................1
   1.2 The interceptor project .....................................................................2

2. Stakeholder Analysis ....................................................................... 3
   2.1 Simulation competence ...................................................................3
   2.2 Protocol architects ..........................................................................3
     2.2.1 Protocol inference ....................................................................4
     2.2.2 Protocol checking .....................................................................4
   2.3 University supervisor .......................................................................4
   2.4 PDEng trainee ................................................................................4

3. Problem Analysis ........................................................................... 7
   3.1 Non-intrusive data flow interception .............................................7
     3.1.1 Network packet interception ....................................................7
     3.1.2 Library function interception ...................................................8
     3.1.3 Conclusion .............................................................................8
   3.2 Client/server object architecture .................................................9
     3.2.1 Client interactions ..................................................................10
     3.2.2 Server interactions ..................................................................13
     3.2.3 Sequence of events ..................................................................15
   3.3 Interception use cases ............................................................... 16
     3.3.1 Hardware probe .....................................................................16
     3.3.2 Protocol checker .....................................................................16
     3.3.3 Non-intrusive logger .................................................................16
   3.4 Design Opportunities ...............................................................17

4. Feasibility Analysis .......................................................................19
   4.1 Introduction ...................................................................................19
   4.2 Library pre-loading .......................................................................19
     4.2.1 Library pre-loading on Unix systems ........................................19
4.2.2. Library pre-loading on VxWorks ........................................20
4.3 Located the _next function ............................................20
  4.3.1. Locating the _next function on Unix systems ...............20
  4.3.2. Locating the _next function on VxWorks .................21
4.4 Resolving cyclic symbol dependencies ................................21
  4.4.1. Splitting manually .............................................21
  4.4.2. Changing ddgen for automatic splitting ....................22
4.5 Interception .............................................................22
  4.5.1. Client side interception .......................................23
  4.5.2. Server side interception .....................................25
  4.5.3. Conclusion ......................................................26
4.6 Performance concerns ..................................................26
4.7 Security concerns regarding symbol exposure .......................27

5. System Requirements ......................................................29
  5.1 Requirement Gathering Process ....................................29
  5.2 Functional Requirements ............................................29
  5.3 Non-functional Requirements .......................................30
  5.4 Risks ..................................................................30

6. System Architecture .......................................................33
  6.1 Introduction ................................................................33
  6.2 Logical views by applications ......................................33
    6.2.1. Hardware probe ...............................................33
    6.2.2. Protocol checker ...............................................33
    6.2.3. Logger ...............................................................34
  6.3 Single versus multi process deployment .........................34
    6.3.1. Single process deployment ..................................34
    6.3.2. Multi process deployment ..................................35

7. System Design ...............................................................36
  7.1 Introduction ................................................................36
  7.2 Interceptor generation ...............................................36
  7.3 Templates of generated interceptors ...............................37
    7.3.1. Hardware probe ...............................................37
    7.3.2. Protocol checker ...............................................38
    7.3.3. Logger ...............................................................38
    7.3.4. Design decision ...............................................38
  7.4 Building and deploying interceptors ...............................39
  7.5 Running with interceptors ..........................................39

8. Implementation ..............................................................41
  8.1 Introduction ................................................................41
  8.2 Code generation ........................................................41

9. Verification and Validation ...............................................43
9.1  Introduction........................................................................................................43
9.2  Verification of interceptor..................................................................................43
9.3  Validation of interceptor...................................................................................44

10.  Conclusions and Recommendations...........................................................45
  10.1  Results............................................................................................................45
        10.1.1. Concept soundness ..............................................................................45
        10.1.2. Applicability of interceptors.................................................................45
        10.1.3. Impact on existing codebase.................................................................46
  10.2  Future work and recommendations.............................................................46

11.  Project Management..........................................................................................47
  11.1  Introduction....................................................................................................47
  11.2  The management process..............................................................................47
        11.2.1. Initiating and planning.........................................................................47
        11.2.2. Executing and documentation...............................................................47
        11.2.3. Monitoring and controlling.................................................................47
        11.2.4. Closing..................................................................................................48
  11.3  PSG meetings..................................................................................................48

12.  Project Retrospective.........................................................................................49
  12.1  Technical reflection.......................................................................................49
  12.2  Design opportunities revisited....................................................................49

Glossary .....................................................................................................................51

Appendix: _next function on VxWorks.................................................................53

Bibliography .............................................................................................................54
    References ............................................................................................................54

About the Authors ...................................................................................................55
List of Figures

Figure 1 ASML EUV Machine ................................................................. 1
Figure 2 Where ASML machine fits in the process .................................. 1
Figure 3 Data flow in real and simulation mode ..................................... 7
Figure 4 Network packet reconstruction .............................................. 8
Figure 5 Communication between objects ............................................ 10
Figure 6 Client side object handling ..................................................... 10
Figure 7 Trigger Sequence .................................................................... 11
Figure 8 Blocking function sequence .................................................. 12
Figure 9 Non-blocking call sequence ................................................... 12
Figure 10 Event handling ..................................................................... 13
Figure 11 Server side object handling .................................................. 13
Figure 12 Synchronous function sequence .......................................... 14
Figure 13 Asynchronous function sequence ......................................... 14
Figure 14 Event raising sequence ......................................................... 15
Figure 15 Sequence of events of a synchronous call .............................. 15
Figure 16 Symbol table illustrating locating the _next function .............. 20
Figure 17 Call relation between _class.c and _rq.c .............................. 21
Figure 18 Trigger interceptor ............................................................... 23
Figure 19 Blocking function interceptor .............................................. 23
Figure 20 Callback function ............................................................... 24
Figure 21 Event function .................................................................... 24
Figure 22 Intercepted synchronous function ....................................... 25
Figure 23 Intercepted asynchronous function ..................................... 25
Figure 24 Intercepted event function ................................................... 26
Figure 25 Library splitting imposes information leaking risk .................. 28
Figure 26 Hardware probe architecture .............................................. 33
Figure 27 Single process model ............................................................ 34
Figure 28 Multi process model ............................................................. 35
Figure 29 Verification workflow .......................................................... 43
List of Tables

Table 1 Dependencies direction between Y_class.so and Y_rq.so .................15
Table 2 Functional requirements for general interceptors .........................29
Table 3 Functional requirements for logger........................................30
Table 4 Non-functional requirements for general interceptors ..................30
Table 5 Interceptor generation map..................................................36
Table 6 Comparison of interceptor templates.....................................38
1. Introduction

Abstract – This chapter introduces the company and lays the context for this project.

1.1 ASML

ASML is one of the world’s leading manufacturers of chip-making equipment. Founded in 1984, the Veldhoven-headquartered company was once a joint venture of Philips and Advanced Semiconductor Materials International (ASMI), under the name ASM Lithography. When the company became independent in 1988, the abbreviation ASML became the official company name. ASML is the largest photolithography machine provider in the world, amounting to more than 85 percent market share.

Figure 1 ASML EUV Machine

The main products of ASML are lithography machines. Figure 1 illustrates a flag-ship TWINSCAN EUV machine. Lithography is a key step in the semiconductor manufacturing process, as can be seen in Figure 2. Lithography machines are essentially project systems, using laser light to lay out the transistors. The light is projected using predefined patterns onto a wafer covered by a thin film of light-sensitive material called photoresist. When the unexposed parts are etched away, the pattern is revealed on the wafer.

Figure 2 Where ASML machine fits in the process
1.2 The interceptor project

The TWINSCAN product software is designed to control the hardware. The product software is crucial to let the machine operate at nanometer accuracy. The accumulated line of code is commonly quoted as 45 million\(^1\). Because of its large codebase, testing and simulation are very important to ensure quality. As not all machine configurations are available or too expensive to be used for software testing, the hardware needs to be simulated, such that the software can still be tested (a.k.a. software-in-the-loop test). This hardware simulator needs somehow to be connected to the software in order to bypass the hardware. This must be done in such a way that the HW/SW interface (i.e. actuator and sensor data) is as stable as possible and the evolution of the software is decoupled from the evolution of the hardware simulator as much as possible. The control flow (i.e., the function calling order) within production software may not be altered (a.k.a. non-intrusive) for testing purposes. This has led to the non-intrusive interceptors project.

Three applications of interception are discovered from my investigation:

1. **Hardware probe**: observe and inject data flow from/to a given function
2. **Protocol checker**: check the protocol compliance (i.e. pre/post-conditions on the objects member functions)
3. **Logger**: log function call details into a persistent data store

Hardware probe has been partially explored by my supervisor Will Denissen. Protocol checker is a project owned by Jinfeng Huang and requires interceptors. Logger is fully owned by me. More applications can be foreseen, because of the generic property of interceptors. Though these applications may have different characteristics, they all need interceptors to add additional functionalities.

In this project, a full research and assessment of the interceptor technique are conducted.

\(^1\) Connecting formal and legacy systems without a single line of code, Gemma Church
2. Stakeholder Analysis

Abstract – This chapter identifies the main stakeholders of the project. Two groups are involved in general: ASML and Eindhoven University of Technology (TU/e). For each concerned party in their groups, the representative stakeholders are listed with their role, responsibility, acceptance criteria, and involvement.

2.1 Simulation competence

<table>
<thead>
<tr>
<th>Representative</th>
<th>Will Denissen (project supervisor)</th>
</tr>
</thead>
</table>
| Role                   | 1. research and transfer test and isolation technology to ASML  
                        | 2. define the project and make sure the project goals are met on time |
| Responsibility         | 1. provide high-level guidance on the project and contacts in the company  
                        | 2. monitor, evaluate, assess and provide regular feedbacks on the progress and deliverables  
                        | 3. provide relevant domain knowledge and references  
                        | 4. evaluate and assess the functional and non-functional aspects of the system  
                        | 5. provide knowledge on hardware probe  
                        | 6. review project report |
| Acceptance criteria    | 1. explore the solution space and propose various strategies and techniques to non-intrusively intercept the behavior of existing system  
                        | 2. investigate the feasibility of the interceptor concepts on both Linux and VxWorks  
                        | 3. provide a complete description on the experiments having done  
                        | 4. design and implement an code generator for ASML’s development environment  
                        | 5. deliver code and report in time |
| Involvement            | continuous communication via daily meetings on ad-hoc basis; monthly PSG meeting with university supervisor and trainee |

2.2 Protocol architects

<table>
<thead>
<tr>
<th>Representatives</th>
<th>Nontas Rontogiannis, Sven Weber and Ivo ter Horst</th>
</tr>
</thead>
</table>
| Role                   | 1. users of the interceptor  
                        | 2. transform legacy components to their formal counterparts  
                        | 3. mature preliminary formal components |
| Responsibility         | 1. provide relevant domain knowledge  
                        | 2. provide needs and requirements regarding the project |
| Acceptance criteria    | interceptor fulfilling their requirements |
| Involvement            | meeting, demo are scheduled upon trainee’s request |
### 2.2.1. Protocol inference

<table>
<thead>
<tr>
<th>Representative</th>
<th>Ramon Schiffelers, Nan Yang, and Kousar Aslam</th>
</tr>
</thead>
</table>
| Role           | 1. user of logger interceptor  
2. construct and validate formal components from existing software behaviors |
| Responsibility | 1. provide relevant domain knowledge  
2. provide their requirements regarding the project |
| Acceptance criteria | interceptor meeting their requirements |
| Involvement    | Meetings and demos are scheduled upon trainee’s request |

### 2.2.2. Protocol checking

<table>
<thead>
<tr>
<th>Representative</th>
<th>Jinfeng Huang</th>
</tr>
</thead>
</table>
| Role           | 1. users of the interceptor  
2. understand the existing software  
3. refactor existing software |
| Responsibility | 1. provide relevant domain knowledge  
2. provide their requirements regarding the project |
| Acceptance criteria | interceptor meeting their requirements |
| Involvement    | meetings and demos are scheduled upon trainee’s request |

### 2.3 University supervisor

<table>
<thead>
<tr>
<th>Representative</th>
<th>Alexander Serebrenik / Tom Verhoeff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role</td>
<td>guard the educational interests of the university and the trainee</td>
</tr>
</tbody>
</table>
| Responsibility | 1. provide relevant domain knowledge, references and contacts  
2. monitor, evaluate, assess and provide regular feedback on the project progress and deliverables  
3. trigger reasoning and discussions on major decisions  
4. review project report |
| Acceptance criteria | 1. deliver reports on time  
2. design, implementation, project management and documentation meeting the level of PDEng project |
| Involvement    | weekly meeting on Monday morning, and monthly PSG meeting with company supervisor and trainee |

### 2.4 PDEng trainee

<table>
<thead>
<tr>
<th>Representative</th>
<th>Qing Cai</th>
</tr>
</thead>
<tbody>
<tr>
<td>Role</td>
<td>software designer, project manager</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Responsibility</td>
<td>1. solicit requirements from stakeholders and scope the project</td>
</tr>
<tr>
<td></td>
<td>2. investigate the solution space</td>
</tr>
<tr>
<td></td>
<td>3. design, implement, integrate and test interceptors</td>
</tr>
<tr>
<td>Acceptance criteria</td>
<td>1. timely deliverables</td>
</tr>
<tr>
<td></td>
<td>2. quality content meeting the expected PDEng level</td>
</tr>
<tr>
<td></td>
<td>3. interceptor meeting the requirements from stakeholders</td>
</tr>
</tbody>
</table>
3. Problem Analysis

Abstract – This chapter gives related information on the current context of ASML software, and discovers opportunities to design the interceptor.

3.1 Non-intrusive data flow interception

As control flow cannot be altered (see 1.2), we can only influence the data flow within the software. Data flow is basically the data that is flowing in or out through the function parameters. Data flow interception provides a way to let the hardware simulator observe or inject data into the data flow.

![Diagram: Data flow in real and simulation mode]

In a software-in-the-loop test, the traditional way to replace hardware is to create a separate stub hardware controller that feeds pre-defined sensor data to the higher level controller, as shown in Figure 3. This involves changing production code, making simulation mode different than real mode, which is not ideally preferred.

In order to observe and inject data flow to the system under test (SUT) without changing production code, two solution directions are foreseen: network packet interception and library function interception.

3.1.1. Network packet interception

Network packet interception operates on IP packets (raw data) flowing in the local network. The network uses the four layer TCP/IP model. This can be seen in Figure 4.
The Internet layer is used to transport IP packets from the originating host across network boundaries. Data is encapsulated in IP packets, which can be intercepted within the operating system (Linux/Solaris). Even if the operating system does not support intercepting IP packets, a network sniffer can be introduced to monitor and modify network data.

Although intercepting network packets offers many possibilities for data observation and processing, there are several drawbacks:

1. If a data packet handed from upper layers are encrypted, then it is impossible to read the data.
2. Data from the application layer can be spliced into many IP diagrams. In order to observe and modify the data, the transport layer and application layer packets have to be reconstructed as well, as depicted in the above diagram. The underlying implementation can be very complex.
3. If data is not transmitted via network interface, then there is no way to observe and interfere.

This method is quite complex, error-prone and has a rather high limitation.

### 3.1.2. Library function interception

Library function interception is often used to alter or augment existing library functions. It naturally gives a lot of useful logical information such as function name, parameters and return value.

In this mechanism, a library double is loaded in front of the original library. When a function in the library double is called, it can further call the original function. In this way, the library double extends or alters the original library behavior. The library double is the interceptor. This mechanism can be implemented on most operating systems that supports dynamically linked libraries, including Windows, Linux and VxWorks.

Library doubling also can be used to factor out certain features from the production code into separate libraries. Such feature externalization especially works for development features such as simulation, debugging, profiling and tracing.

### 3.1.3. Conclusion

As described above, library function interception has lower complexity and can intercept all data flowing through functions within libraries. For network packet interception, one critical drawback is its inability to intercept non-network data flow. The following table gives a comparison of the two interception method.
<table>
<thead>
<tr>
<th></th>
<th>Network interception</th>
<th>Library interception</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complexity</strong></td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Ease to get logical information such as function names</strong></td>
<td>Difficult</td>
<td>Easy</td>
</tr>
<tr>
<td><strong>Potential overhead</strong></td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Ability to intercept TCP/IP data</strong></td>
<td>Yes</td>
<td>Yes (send/receive functions should be exposed in a library)</td>
</tr>
<tr>
<td><strong>Ability to intercept non network data</strong></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Flexibility of deployment</strong></td>
<td>Flexible (can be deployed in hosts or routers)</td>
<td>Not flexible (must be installed on the hosts)</td>
</tr>
<tr>
<td><strong>OS support</strong></td>
<td>All (in a unified way)</td>
<td>Many (different way of interception in different OS, support Windows, Linux etc.)</td>
</tr>
<tr>
<td><strong>Language support</strong></td>
<td>All (in a unified way)</td>
<td>Not all (supports C, C++ at least)</td>
</tr>
</tbody>
</table>

Overall, given the ASML context, library interception is clearly more suitable as the method to intercept data flow.

### 3.2 Client/server object architecture

ASML has an inhouse solution to support cross-language, cross-process object oriented communication mechanism. There are two core inputs in this mechanism:

- **Interface definition**: defines functions that are provided in the interface, in the form of *ddf* (Data Definition File)
- **Class definition**: defines a class that implements one or many interfaces, in the form of *acf* (ASML Class File)

An object is initialized from a class definition and runs in a process. Once a server object is initialized, it needs to be **bound to a name** such that client objects can connect to it. The interactions between the client and server objects consist of calling the server object’s member functions and listen to its events.

A client object and a server object can reside in either in the same process or in a different process. If in the same process, then the client object calls server object directly within its process. If not, the requests will be sent to the remote server object **over the network**. This feature is called **local/remote transparency** and is automatically done by generated supporting libraries.

For example, in Figure 5, object C is an object living in a VxWorks process (a.k.a. task), while objects A and B are living in a Linux process. If object A or B needs to call a method provided in object C, a remote procedure call is automatically made. However, if object A needs to call a method in object B, then there is no inter-process communication.

Another benefit of using a client/server architecture is language and OS transparency. When using the client and server object abstraction, it does not matter which language or which operating system is actually crossed. At ASML, C and C++ are used prevalently as programming language in production codebase. TWINSCAN software runs on Linux, Solaris and VxWorks operating systems. Interceptors should consider such environments.
For simplicity, the next section describes the client server communication in more detail for the C language on Linux only.

### 3.2.1. Client interactions

In the client/server object architecture, a client object is the part that requests a server object (a.k.a. servants) to do something. The proxy object pattern is used to access remote servers. A proxy object represents the servant in the client process context. It is responsible for request marshalling, response unmarshalling and network transmission.

An example of client proxy communication is illustrated in Figure 6. The client object C, whose behavior is implemented in a shared library cli.so, communicates to the proxy of object A, whose behavior is implemented in a shared library prx.so. From a testing point of view, the **business logic** is contained in cli.so and needs to be tested. The proxy code prx.so is considered as glue code, as its purpose is to make inter process communication (IPC) possible.
At ASML, a *ddf* describes an interface of a proxy library, which can be generated by *ddgen*. A sample *ddf* looks like the following:

```c
interface X:Fun {
    trigger trg(in  double p1;);
    fcn nonblocking function sfun(in  double p1; inout double p2);
    function sfun(in  double p1; inout double p2; out double p3);
    event evt(out double p1;);
}
```

This *ddf* defines an interface called X:Fun. The interface defines four kinds of functions:

1. a trigger function *trg*,
2. an asynchronous function with a callback *sfun*,
3. a synchronous function *afun* and
4. an event *evt*.

Each parameter has a direction. The direction *in* means the data flows from client to server object. The direction *inout* means the data flows from client to server object and back. The type *out* the data flows from server to client object.

*ddgen* generates a client side proxy code `Xmet.c` from `Xddf`, which can be compiled and linked into a shared library (`libX_met.so`).

The four different kinds of proxy functions are discussed below.

**Trigger**

![Figure 7 Trigger Sequence](image)

A trigger is a client function that informs a server about something, without expecting any returns. Triggers are not guaranteed to be handled by the server. This is illustrated in Figure 7.
**Blocking function call**

A blocking function actively waits for results from a server. For example in Figure 8, a client calls $f$ from a server. While waiting for the result, the client is blocked.

**Asynchronous call with callback**

An asynchronous call with callback (a.k.a. function callback notification) does not expect results right away. After sending the request to the server, the process continues to run. When a reply is received from the server, a callback function will be called to handle the result. This is shown is Figure 9. (Note: unlike a message sequence diagram the vertical dotted bar does not depict a thread of control but just the location in which the callback is implemented.)
Event

A client can subscribe and unsubscribe to server events. When a server object raises an event, all subscribed client objects will receive the event and their callbacks are called subsequently. This is illustrated in Figure 10.

3.2.2. Server interactions

A server object is the part that listens to requests from client objects and handles the requests accordingly.

In the ASML context, a server process hosts one or many server objects. A server object is instantiated from an ASML class. acf describes the interfaces of a remote object implemented in a class, which can be compiled and linked into a skeleton shared library. The skeleton library handles the client requests and turns it into a member function call of the actual server object. As an example, Y.acf looks like the following:

```plaintext
class Y;
  implements X:Fun;
  standardconstructor;
  asynchronous aFun;
```

In the above example, a class that implements X:Fun interface is defined. This class has a standard constructor, and has an asynchronous function reply for aFun only. All other functions defined in the X:Fun interface will be synchronous by default.
ddgen generates for a given Y.acf a server side skeleton with Y_class.c and a header file Y_rq.h. Users are required to provide their business logic in the Y_rq.c file. The server side skeleton Y_class.so waits for the requests and calls the corresponding handler in Y_rq.c. This is illustrated in Figure 11. Note that it is very common to compile Y_class.c and Y_rq.c into one library.

Different types of server handling are discussed in the coming parts.

**Synchronous handling**

![Synchronous function sequence](image)

**Figure 12 Synchronous function sequence**

In the synchronous handling mode, the server process is blocked while handling the request. When the user library function returns, a reply is automatically sent by server side stub Y_class.so. This can be seen in Figure 12.

**Asynchronous handling**

![Asynchronous function sequence](image)

**Figure 13 Asynchronous function sequence**

In the asynchronous handling mode, a server is free to handle other server object requests before returning the corresponding reply. This process can be seen in Figure 13.
Event raising

![Event raising sequence](image)

Figure 14 Event raising sequence

The sequence diagram of event raising is depicted in Figure 14. The user-provided $Y_{rq}.so$ raises events to indicate server state change.

The library dependency relation between $Y_{class}.so$ and $Y_{rq}.so$ is shown in Table 1. Because of the way how $ddgen$ generates $Y_{class}.c$, there is a cyclic dependency between the two libraries for asynchronous function replies and event raising. This will cause issues in VxWorks, where such dependency is not allowed, as discussed in 0.

Table 1 Dependencies direction between $Y_{class}.so$ and $Y_{rq}.so$

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous function</td>
<td>$Y_{class}.so \rightarrow Y_{rq}.so$</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>$Y_{class}.so \leftrightarrow Y_{rq}.so$</td>
</tr>
<tr>
<td>Event raising</td>
<td>$Y_{class}.so \leftrightarrow Y_{rq}.so$</td>
</tr>
</tbody>
</table>

3.2.3. Sequence of events

Given the separate description of client and server, this section shows how a full client/server interaction behaves step by step. Take the simplest example: blocking function from a client and synchronous process from a server. This is illustrated in Figure 15.

![Sequence of events of a synchronous call](image)

Figure 15 Sequence of events of a synchronous call

1. The client calls the function $f$ in the proxy ($prx.so$). The call is a local function call.
2. The client proxy checks whether the object is located in the same process. If so, the proxy will call the function directly. Otherwise it packs the parameters
into a message and makes a system call to send the message. Packing the parameters is called marshalling.

3. The client's local operating system sends the message from the client machine to the server machine.

4. The operating system on the server machine passes the incoming packets to the server skeleton (`Y_class.so`).

5. The server skeleton unpacks the parameters from the message. Unpacking the parameters is called unmarshalling.

6. Finally, the server skeleton calls the server object member function. The reply traces the same steps in the reverse direction.

3.3 Interception use cases

In the next three sections, the different use cases of the interceptors are described.

3.3.1. Hardware probe

In order to accomplish software-in-the-loop testing at an early stage, a hardware simulator can be used to observe actuator data directed to the hardware and inject sensor data coming from the hardware. The hardware simulator and production software should be maximally decoupled, allowing them to evolve independently. However, current solution of hardware simulator is replacing the entire server binary with a stub, which requires all class interfaces to be implemented. This makes the simulator difficult to maintain in the long run.

Hardware simulators also enable injecting possible erroneous hardware behavior and verifying that the system behaves according to the recovery specifications.

At the hardware/software interface (HSI), all data goes through function parameters. Therefore by intercepting HSI library functions, then we can observe and inject data within the system under test. This use case of interception is called a hardware probe.

3.3.2. Protocol checker

At ASML, there is more and more software being developed using model based engineering for supervisory control. The core concept is an interface protocol. An interface protocol defines a set of states, transitions and actions. ASD is the tool that translates a protocol to the supervisory part of business logic code.

When a new protocol is developed to replace the manually-written code, it should be tested, especially the transition part. A conceptual solution is online validation, where the validator runs on top of the correct system and checks whether there is a violation between the protocol and the system. If there is, the issue should be reported and fixed.

Changing production code is not wanted, as it pollutes the production code in many places. This use case of interception is called protocol checker.

3.3.3. Non-intrusive logger

Logging and tracing are very important for debugging and profiling. It’s also a key step in process mining and model inference. The traditional way of doing so requires putting lots of tracing statements in the production code. This is inefficient and prone to errors. For example, if one tracing statement is missing, it might lead to a misleading result. An automatic way of logging is very desirable in this case.

The logger must not alter the correct machine behavior. Since there may be a large number of function calls to log, it is desirable to disable loggers for production environment. Turning on and off the logger should be made effortless.

This use case of interception is called non-intrusive logger.
3.4 Design Opportunities

Based on the aforementioned analysis, it is seen all three use cases can be solved by interceptors.

The design should cover the following aspects of interceptors:

- Interceptors generation
- Default interceptor template
- Building and deploying
- Running with interceptors

Chapter 7 discusses these topics in detail. Before starting to design, it is necessary to investigate the interceptor technique, and see whether it is possible to fit the technique within ASML’s context. The next chapter gives a full feasibility analysis on interceptors.
4. Feasibility Analysis

Abstract – In this chapter, feasibility of interceptors in ASML context is evaluated. Such analysis supports better project plan and lowers the risk of wrong choice of solution direction.

4.1 Introduction

At ASML, there are three operating systems being used: Solaris, Linux and VxWorks. Among these, Solaris and Linux are both derived from Unix family and share much in similar. VxWorks is a very different system with a focus on real-time performance. An important difference between these systems is around the sharing of libraries. Unix family systems uses a dynamic library loader that supports shared libraries (suffixed with .so). On VxWorks, objects and archives (suffixed with .o or .a) are used to ensure high performance and reliability. For simplicity purpose, we call both the shared library as well as the object archive libraries, from this point on.

An interceptor is a library that intercepts other library. In order to see how the interception can be done, a closer look at normal library function call is needed.

Without interception, calling a library function requires the following steps:

1. Load the library into memory and link the library’s required symbols to the provided symbol
2. Call the function based on the address in the symbol table

With interception, calling a library function requires the following steps:

1. **Load** the interceptor library into memory before the original library. As a result each function defined in the interceptor library will be called instead of original function.
2. Locate the original functions (a.k.a. the _next function) and call it.

The first step is called library pre-loading, while the second step is named locating the _next function. The next two sections cover how it works for shared objects .so and object archives .a.

4.2 Library pre-loading

Library pre-loading is the process of loading an interceptor library in front of the actual one, causing the library double to have a higher execution priority. An interceptor library implements a subset of the functions defined in the original library. An interceptor library can further call the original one, thus creating a wrapper at library load time.

4.2.1. Library pre-loading on Unix systems

On Linux, a dynamic library loader *ld.so* loads the required libraries at load time (just before main starts). An optional environmental variable *LD_PRELOAD* indicates the libraries that will be loaded before any other shared libraries, including C runtime library *libc.so*. The following bash script preloads *libTATC_rq.so* when running *TATC_srv*.

```bash
#!/bin/sh
LD_PRELOAD=libTATC_rq_ntcp.so TATC_srv
```
In this example, \textit{TATC\_srv} is dependent on \textit{libTATC\_rq.so}. \textit{libTATC\_rq\_ntcp.so} implements a subsets of the functions implemented in \textit{libTATC\_rq.so}. Preloading allows functions in this library to be altered or enhanced.

4.2.2. Library pre-loading on VxWorks

In VxWorks, the loading process is done manually using the object loader \textit{ld}. At ASML, software running on VxWorks is built as kernel modules running inside the OS kernel, ensuring maximum performance and flexibility. All kernel modules share the same \textit{global symbol table}. The global symbol table maps data/function names to data/function addresses. If there are multiple symbols with the same function name, then required symbols will be resolved to the address of the last loaded one. As a result, to achieve pre-loading, the interceptor libraries should be actually loaded after the original library.

For example, the following loading sequence pre-loads the interceptors before the original ones. Note that the interceptors are loaded after the original ones.

\begin{verbatim}
ld < bin/vw/libTATC_class_pr_vwppc.a
ld < bin/vw/libTATC_class_pr_ntcp_vwppc.a
ld < bin/vw/libTATC_rq_vwppc.a
ld < bin/vw/libTATC_rq_ntcp_vwppc.a
\end{verbatim}

4.3 Locating the \textit{\_next} function

From an library interceptor, it must be possible to call the original (a.k.a. \textit{\_next}) function. This is because the original function is the next function with the same name in the symbol table, as shown in Figure 16.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{symbol_table.png}
\caption{Symbol table illustrating locating the \_next function}
\end{figure}

In the above symbol table, symbol resolution order is from top to bottom. The function \textit{trigger\_motor} in the second row comes from the interceptor library. But how to execute the \textit{NEXT} function, which is located in the fifth row?

4.3.1. Locating the \textit{\_next} function on Unix systems

On Unix systems, the \textit{dlsym} function in \textit{libdl.so}(dlfcn.h) can be used to find the \textit{\_next} function, as the following code shows:

\begin{verbatim}
void f() {
    typedef typeof(f) T;
    static T* _next = NULL;
    if (!_next) {
        _next = (T*)dlsym(RTLD_NEXT, "f");
    }
    _next();
}
\end{verbatim}

In this code snippet, \textit{\_next} is defined as a function pointer to the original function. A static function pointer caches the result. When function \textit{f} is executed for the first time, \textit{\_next} will be initialized \textit{only once}. 

\begin{verbatim}
void f() {
    typedef typeof(f) T;
    static T* _next = NULL;
    if (!_next) {
        _next = (T*)dlsym(RTLD_NEXT, "f");
    }
    _next();
}
\end{verbatim}
4.3.2. Locating the _next function on VxWorks

VxWorks kernel module only uses object (.o) or object archives (.a). With the symEach function and pointer to the symbol table sysSymTbl makes creating a _next function possible by iterating over the symbol table.

The source code is attached in the appendix.

4.4 Resolving cyclic symbol dependencies

As described in 3.2.2., to intercept event raising and asynchronous replies on VxWorks, the cyclic dependency between _class.c and _rq.c must be broken. As can be seen in Figure 17, _class.c and _rq.c are interdependent, and they are compiled to _class.o and _rq.o respectively. In Unix systems, the dynamic loader ld.so handles the cyclic symbol dependency without a problem, because it delays the symbol resolution until all are loaded. On VxWorks however, all required symbols of a library must have been loaded prior to its resolution. This causes unresolved symbol issues on VxWorks. In order to solve the problem, _class.o should be split into two libraries, namely _class_pr.o and _class_rq.o, eliminating the external cyclic dependency.

The following sections shed light on how the split is done manually, and then, automatically.

4.4.1. Splitting manually

To remove the cyclic dependencies, it is important to understand where the dependencies happen. This requires a deeper analysis at the code level.

When a request is processed by the skeleton code, it is unmarshalled and the actual handler in _rq.c is called. In synchronous reply scenario, the server skeleton immediately packs the function results (the return value and its out parameters) into a reply and sends it back to the client. A sample synchronous function from _rq.c can be:
However, in the asynchronous reply scenario, some other function will send the reply. A sample asynchronous reply can be seen below. \texttt{Z\_rp\_func} is the reply function defined in \texttt{Z\_class.c}.

```c
void Some\_other\_function(\ldots) {
    // business logic
    // set <out> data
    Z\_rp\_func(<out>\ldots);
}
```

Asynchronous replies always return void, and use a \texttt{ddgen} generated reply function to send back the results.

For event raising, the situation is similar to asynchronous reply. Dedicated event raising functions are generated by \texttt{ddgen} in \texttt{Z\_class.c}. The following is an example.

```c
void Some\_other\_function(<in>\ldots, <out>\ldots) {
    // business logic
    // can raise event multiple times
    Z\_raise\_event\_func(<out>\ldots);
}
```

To conclude, it is sufficient to move asynchronous reply and event raising functions to a new library, supposedly called \_class\_pr, meaning a providing class library. In order to differentiate the original \_class, the new one is renamed to \_class\_rq.so, meaning a requiring class library. The resulting dependency relation is depicted in Figure 17.

A proof of concept has been given that this actually works within ASML development environment. The work consisted of splitting \_class.c and change the \texttt{makefile} accordingly. For TATC component, this is validated both on a devbench and a testbench.

### 4.4.2. Changing \texttt{ddgen} for automatic splitting

Since splitting the generated \_class.c file has been proven to work. Therefore, generating a modified \_class.c should be possible too, by changing the code generator \texttt{ddgen}.

A proof of concept has been given that this actually works within ASML development environment. The source code of \texttt{ddgen} is versioned in ASML Bitbucket. It is free to be forked. More information of changing \texttt{ddgen} can be found in section 8.2.

### 4.5 Interception

Now that we have proven that all cyclic dependencies can be removed and both proxy and skeleton code is packaged into a library, we know that the preconditions for library interception at the client side as well as at the server side are fulfilled. But does it work in the ASML development environment? For all kinds of client/server interactions?
With the interception technique described earlier, client side and server side interceptors are analyzed in this section. Sequence diagrams are used to describe the interception behavior.

### 4.5.1. Client side interception

Based on the client side interactions discussed in 3.2.1, the interceptor library `prx1.so` can be loaded in front of the client sided proxy library `prx.so`.

For a **trigger function**, the executed code in each library is depicted in Figure 18. When the client calls the trigger function `X_trg`, the `X_trg` function within the interceptor library `prx1` is actually called, who in its turn will call its `_next` function being the `X_trg` function in the original library `prx.so`.

![Figure 18 Trigger interceptor](image)

**Figure 18 Trigger interceptor**

Similarly, the interception behavior of a **blocking function** is depicted in Figure 19.

![Figure 19 Blocking function interceptor](image)

**Figure 19 Blocking function interceptor**
The interception behavior for **function with callback notification** is slightly different, as can be seen in Figure 20. In the function \( f_{\text{fcn}@\text{prx1.so}} \), a wrapped callback \( cb1 \) and wrapped context \( ctx1 \) are created and passed on to \( f_{\text{fcn}@\text{prx.so}} \). The \( ctx1 \) contains the pointer to the original context \( ctx \) and the original callback function \( cb \). When the wrapped callback \( cb1 \) is called, the original \( ctx \) is extracted from the wrapped context \( ctx1 \) and passed to the original callback \( cb \). NOTE: This only works because the context is passed on **without any interpretation or modification**. In this way, any callback can be intercepted and relayed.

![Figure 20 Callback function](image)

**Event handling** are similar as callback functions. When an event is raised at the server side, an event handler callback is invoked on the client side. In order to intercept the

![Figure 21 Event function](image)
callback, in the intercepted event subscription function `e_subscribe@prx1.so`, a new wrapper callback `cb1` is registered. When an event is raised, `cb1` gets called and it will call the original `cb` with the original context `ctx`. This process can be seen in Figure 21.

### 4.5.2. Server side interception

Based on the server side interactions discussed in 3.2.1., the interceptor library `Y_rq1.so` can be put in front of the original library `Y_rq.so`.

The sequence diagram for intercepted synchronous replies at the server side is depicted in Figure 22.

**Figure 22 Intercepted synchronous function**

Intercepting asynchronous reply functions, as described in 3.2.2., requires `Y_class.so` to be split into `Y_class_rq.so` and `Y_class_pr.so`, and 2 interceptor libraries `Y_r1.so` and `Y_class_pr1.so`. The corresponding sequence diagram is shown in Figure 23. `Y_rq1.so` intercepts the functions to `Y_rq.so`, while `Y_class_pr1.so` intercepts the functions to `Y_class_pr.so` (i.e. the asynchronous reply function `Y_rp_X_af`).

**Figure 23 Intercepted asynchronous function**
The sequence diagram for intercepting **event raising function** at the server side is shown in Figure 24. \( Y_{rq1}.so \) intercepts the event raising function \( Y_{rp\_raise\_X\_evt} \) provided in \( Y\_class\_pr.so \).

**intercepted event evt(\text{\textless}out{\textgreater}\ldots)**

![Sequence Diagram](image)

**Figure 24 Intercepted event function**

### 4.5.3. Conclusion

A proof of concept has been given that all client and all server side interactions interception actually work within the ASML development environment for both Linux and VxWorks. The code is versioned in ASML Bitbucket\(^2\).

### 4.6 Performance concerns

Performance and deterministic execution is important in ASML software, especially on VxWorks. In terms of performance, CPU use, memory use and network can all cause performance degradation. Regarding the interceptor, the overhead mainly happens when locating the address of the \(_next\) functions.

On VxWorks, having multiple functions with same the name is allowed for the system symbol table. Resolving the first symbol is \( O(1) \) complexity by using a hash table. But for other symbols with the same name, the system symbol table must be iterated in a linear fashion. The worst case happens when all the intercepted functions are located near the end of the table. In a typical ASML environment, the size of a symbol table can be as large as ten thousand. On Linux each library has its own dynamic symbol table each with its own hash table, so there is less of a performance penalty.

By using a **static** function pointer to point to \(_next\) function, the lookup process is reduced to once per function, significantly alleviated the overhead problem. Since a real time system can't afford the uncertainty of such expensive lookup, **all** the static function pointer **initializations** should be processed at **library load time**. By using the special function attributes **constructor** and **destructor** of the compiler and the ELF ( Executable and Linkable Format ) support at runtime, all static pointers can be initialized at once, once the interceptor library is loaded. This technique is called **early symbol binding**.

---

static void at_lib_load (void) __attribute__((constructor));
static void at_lib_unload(void) __attribute__((destructor));

static void at_lib_load(void)
{
    /* This is run once when library is loaded */
}

static void at_lib_unload(void)
{
    /* This is run once when library is unloaded */
}

4.7 Security concerns regarding symbol exposure

ELF shared libraries expose certain information to be linked correctly. For example the symbol table. This may leak certain intellectual property to the customers, as a function name could indicate an algorithm, a process method or a patent.

Each ELF shared library contains two symbol tables: the .symtab table and the .dynsym table (or dynamic symbol table). The .symtab table contains all local and global symbol names for the static linker. The .dynsym contains the dynamic symbol table for the dynamic linker, so that it can be properly loaded and linked at run-time.

The strip command takes out the .symtab table, but cannot take out the .dynsym table. The following takes out the local and global symbols in library libTATC_rq.so.

$ strip --strip-all --discard-all libTATC_rq.so

The dynamic symbols are still exposed, as can be seen in the following code, using the TATC component as example.

In order to intercept all function interactions, one server library has to be split into multiple libraries as illustrated in Figure 25. By splitting the libraries, more previously internal function symbols handled by the static linker (.symtab) and could have been stripped, now need to become visible for the dynamic loader (.dynsym). This may pose certain risks on confidential information.
Figure 25 Library splitting imposes information leaking risk
5. System Requirements

Abstract – This chapter describes the requirements solicited from the stakeholders.

5.1 Requirement Gathering Process

The project has several groups of stakeholders as discussed in Chapter 3. By conducting interviews with the stakeholders, their requirements and concerns are clarified and then analyzed. Some of them can be overlapping, and some can be contradicting. Requirements are therefore broken into smaller, independent pieces and are assigned with priority based on the influence of the stakeholder and the nature of the problem. The goal is to fulfill high priority requirements first and as many low priority ones as possible.

Two sets of requirements are identified: functional and non-functional requirements. Each requirement is assigned with a priority, as defined by S. Brander:

1. MUST - absolute requirement of the specification.
2. SHOULD - there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
3. OPTIONAL - the requirement is truly optional.

5.2 Functional Requirements

Functional requirements define the functions of a system. Table 2 shows a list of functional requirements of general interceptors.

Table 2 Functional requirements for general interceptors

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR1</td>
<td>The interceptor function shall be able to call the original function.</td>
<td>MUST</td>
</tr>
<tr>
<td>FR2</td>
<td>Users should be able to add pre/post processing to an interceptor functions.</td>
<td>MUST</td>
</tr>
<tr>
<td>FR3</td>
<td>The interceptor shall intercept all server interactions, which are, (a)synchronous handling, and event raising.</td>
<td>MUST</td>
</tr>
<tr>
<td>FR4</td>
<td>The interceptor shall intercept all client interactions, which are, (non-)blocking functions, event (un)subscription and event broadcasting.</td>
<td>MUST</td>
</tr>
<tr>
<td>FR5</td>
<td>ddgen shall generate interceptors for acf/ddf interfaces.</td>
<td>SHOULD</td>
</tr>
<tr>
<td>FR6</td>
<td>The interceptor shall be able to intercept arbitrary library functions including non-ddf functions.</td>
<td>SHOULD</td>
</tr>
</tbody>
</table>

The functional requirements for logger are listed in Table 3.
Table 3 Functional requirements for logger

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>FR20</td>
<td>The interceptor shall provide a default logger.</td>
<td>MUST</td>
</tr>
<tr>
<td>FR21</td>
<td>The default logger shall be able to log ddf name, interface name, parameter, return value and timestamp.</td>
<td>SHOULD</td>
</tr>
</tbody>
</table>

5.3 Non-functional Requirements

Non-functional requirements are quality attributes, which describe how the system should work. Table 4 shows a list of non-functional requirements gathered from stakeholders.

Table 4 Non-functional requirements for general interceptors

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR1</td>
<td>The interceptor shall work on Linux and Solaris.</td>
<td>MUST</td>
</tr>
<tr>
<td>NFR2</td>
<td>The interceptor shall work on testbench with VxWorks controllers.</td>
<td>MUST</td>
</tr>
<tr>
<td>NFR3</td>
<td>The interceptor shall be integrated into ASML build environment.</td>
<td>OPTIONAL</td>
</tr>
<tr>
<td>NFR4</td>
<td>The interceptor shall be able to be enabled and disabled individually.</td>
<td>OPTIONAL</td>
</tr>
<tr>
<td>NFR5</td>
<td>One interceptor library shall intercept one library.</td>
<td>SHOULD</td>
</tr>
</tbody>
</table>

NFR5 describes the modularity property of interceptors. It is desired to minimize the functionality provided by an interceptor. One interceptor intercepts only one, not many libraries. This makes interceptors easier to maintain and can be switched on and off independently.

<table>
<thead>
<tr>
<th>ID</th>
<th>Requirements</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFR11</td>
<td>The default logger must be completed under 1ms on VxWorks.</td>
<td>SHOULD</td>
</tr>
<tr>
<td>NFR12</td>
<td>The default logger shall not change system behavior.</td>
<td>SHOULD</td>
</tr>
</tbody>
</table>

Since the interceptors are supposed to be used inside ASML development and testing environment only, therefore security issues such as preventing third-parties using interceptors are not investigated. The security rules applied to ASML software also applies to interceptors.

5.4 Risks

In this section, the risks of applying interceptions are discussed. Two risks are identified.

1. The duration within the interceptor can be too large and such that it influences the system behavior (from all three stakeholders)
The added overhead of pure interceptor without any pre/post processing is negligible. However, if a time-consuming algorithm is added to the interceptor, then there's an uncertainty of changing the behavior of the system (i.e. timeouts going off or the work does not fit anymore in the assigned time-slot). To mitigate the risk, all the users must assure their code runs within the given timeslot.

2. The logged data cannot be so large such that it causes buffer overflow or protocol timeouts (protocol architects)

On VxWorks, the available memory space is very limited. If the logs are created faster than being consumed, then there's a possibility of buffer overflow causing data misses. This is not acceptable for protocol logging. To mitigate this risk, at least a mechanism to signal buffer overflow is necessary. It's also helpful to find the maximum logging throughput.
6. System Architecture

Abstract – The interceptor project is not an independent system. As a result, it is difficult to apply some architectural view patterns such as 4+1 view model. Also because of the large software system at ASML, the system architecture in this chapter is described in a simplified way.

6.1 Introduction

When we talk about system architecture, we usually talk about the conceptual model that defines structure, behavior and views of a system. Once the conceptual model is defined, all software components should comply with it. This makes it difficult to change architecture in the future. As a result, an architectural decision usually needs to go through thorough reviews.

The interceptor is essentially a plugin-in mechanism, applicable to each software in the system. Once it is put into practice and many interceptors are developed, changes will be difficult. Thus a reliable architecture is needed.

This chapter describes the architecture in different views, inspired by 4+1 view models and some other architectural patterns.

6.2 Logical views by applications

The logical view is concerned with the functionality that the system provides. Three applications are discovered and discussed separately.

6.2.1. Hardware probe

A hardware probe is an interceptor that observes or injects (typically actuator or sensor) data passing through it. To observe data coming from the hardware, the probe sends the data to a dataport (a publish/subscribe communication mechanism called DDS). To inject data, the probe receives data from the dataport and returns it to the calling function.

Probes are always installed at the server side, as displayed in Figure 26. The naming convention for probes is adding a _probe suffix to the component name.

![Figure 26 Hardware probe architecture](image)

6.2.2. Protocol checker
A protocol checker has two parts. One is a protocol specific observer. The other part is a generic protocol checker runtime library (e.g., PC.so). The protocol observer is the interceptor that contains the state transition table and calls generic functions from the PC.so runtime. The protocol checker runtime library contains common code such as, signaling a protocol violation, logging context information to do some offline diagnostics, or stepping through the state transition table. Once a violation of the state machine is detected, the violation will be signaled to users either through 1) inline raising of an event to the application, or 2) logged to some offline store.

6.2.3. Logger
The TWINSCAN software has a dedicated logging and tracing facility library. It is available on both Linux and VxWorks, and uses ring-buffers and other techniques to ensure real time performance.

The proposed logger design has two parts as well. One is the logger interceptor, and the other is a logging facility library which communicates to a remote logging process (which does the storage on the file system). The logger interceptor can be installed on both client side and server side. Although, direct logging to a file can be done within the host process, it is recommended to use a separate logger process.

6.3 Single versus multi process deployment
Interceptors are always loaded within the host process, and share the same memory space with it. However, the external part that interceptors communicate with, for example, logger and protocol checker, can be deployed into a different process. If the external part shares the same process as interceptors, we call it single-process deployment. Otherwise, we call it a multi-process deployment.

6.3.1. Single process deployment
In the single process deployment, the interceptor Z_ntcp.so are loaded along with its runtime library ntcp_runtime.so in the same process.

![Diagram of single process model]

Figure 27 Single process model

Figure 27 shows the libraries loaded in a server process. The advantage of this model is that the communication overhead between interceptor and runtime library runtime.so is very low. The downside is that a badly implemented runtime.so can corrupt the process, cause memory leaks, thus making the system unreliable.

For the protocol checker and hardware probe the single process deployment is preferred as the runtime library can be kept small, and persistent storage is not needed.
6.3.2. Multi process deployment

In the **multi process deployment**, the **runtime support** is implemented as an independent process. The interceptor functions call functions within the proxy library `runtime_proxy.so`. The proxy library in turn communicates with the run-time process. The deployment is depicted in Figure 28.

If the output of multiple interceptors needs to be **collected in a central place**, or the states of interceptors need to be **persistent** even when the host processes are terminated or restarted, then **multi process deployment** is the preferred choice.

![Multi process model](image)

**Figure 28 Multi process model**

All three applications can benefit from **multi process deployment**. In case of **HW simulator**, the **actuator data** from several hardware probes need to be collected in one process (the HW simulator). At the same time, **sensor data** needs to be distributed back to some probes in order to **inject** the simulated data into the TWINSCAN application. For **loggers**, the proxy library and the remote process already exist as a TWINSCAN facility. For the **protocol checker**, multiple protocol observers can exists for different server processes, and the remote process deployment can be used to check the overall protocol compliance.
7. System Design

Abstract – In this chapter, the system design is described based on system architecture.

7.1 Introduction

In the previous chapters, it is proved that library pre-loading technique can be used in ASML context, considering all client/server interactions, all operating systems, and C/C++. This chapter describes the design of the interceptors.

Unlike traditional standalone programs, where all requirements end up in one product, different kind of interceptors are needed for different applications. Therefore, designing interceptors is more about how to keep the creation of its content flexible within the library pre-loading technique.

Based on the previous analysis, the design of the interceptor comes down to four problems:

1. How to generate interceptors?
2. What is the template for generated interceptors?
3. How to build and deploy interceptor libraries?
4. How to run TWINSCAN system with interceptors?

There are several alternatives to solve each problem. The goal in this chapter is to find the best suitable one.

7.2 Interceptor generation

It is unarguable that the interceptor source code should be automatically generated, since there are simply too many possible interceptors and they share similar templates. But what should be the input to the generator and what should be generated as output?

Two sources of input for interceptors exist, one is acf/ddf interfaces, and the other is library header files. Library header files are also always generated from acf/ddf interfaces.

For acf/ddf interfaces, interceptors should be generated as shown in Table 5. Interceptor library must be generated on a one-to-one basis, meaning that one interceptor library intercepts one library. This ensures linking correctly on VxWorks. The naming convention is to add a suffix to the original library. For instance, _class_pr_ntcp.c is the interceptor for _class_pr.c. By changing code generator ddgen, this can be done relatively easily.

Table 5 Interceptor generation map

<table>
<thead>
<tr>
<th>Source file</th>
<th>Linux library</th>
<th>Interceptor library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client side proxy</td>
<td>_met.c</td>
<td>_met.so</td>
</tr>
<tr>
<td>Server side business logic</td>
<td>_rq.c</td>
<td>_rq.so</td>
</tr>
<tr>
<td>Server side async and event handling</td>
<td>_class_pr.c</td>
<td>_class_pr.so</td>
</tr>
</tbody>
</table>

For non acf/ddf defined library interfaces, header file can be used to generate a the interceptors. For example, a sample header file for a library is as following:
In order to generate interceptors for this header file, a customized new code generator is needed. The code generator reads the function declarations and creates a corresponding interceptor implementation. The naming convention for the source file is the same. For example, algorithms.c is built as libalgo.so, and the interceptor should be named libalgo_ntcp.so.

As a **design decision**, it is proposed that source files of interceptors be generated by `ddgen` along with _met.c, _rq.h and _class_pr.c. If users want to compile certain interceptor libraries, relevant `makefile` should be updated. Otherwise, the generated interceptor source files are not compiled. For non acf/ddf interceptors, a Python script can be written to generate interceptor source file from a library header.

### 7.3 Templates of generated interceptors

`ddgen` generates interceptor source files. The content of generated interceptors is discussed in this section.

The content of interceptors is subject to applications. Let’s take the following function as example.

```c
T f_sync(this, <in>.., <out>..) {
    // do some work
    return result;
}
```

`f_sync` is a synchronous reply function provided in a server side implementation. Depending on its application, the **interceptor function body** is different. The following sections discuss three different interceptor function bodies based on this function.

#### 7.3.1. Hardware probe

In case of the **hardware probe** application, the interceptor function is similar to the following code snippet.

```c
T f_sync(this, <in>.., <out>..) {
    _SETNEXT(f);
    send(<in>..);
    receive_and_update(<in>..);
    T res = _next(this, <in>.., <out>..);
    send(res, <in>.., <out>..);
    receive_and_update(res, <in>.., <out>..);
    return res;
}
```

Depending on whether it is observing or injecting data, this code can be slightly different. The macro `SETNEXT` sets the _next pointer. `send` and `receive_and_update` are functions from the DDS runtime library (e.g. libDDS.so). Before calling _next, the input data can be observed and modified. After calling _next, the input and output can be observed and modified. Typically only a few functions need to be intercepted.

Because it is depending on another library, it is difficult to test early, which is not ideal.
7.3.2. Protocol checker

For the protocol checker application, the situation is a bit more complicated. An inter-
ceptor for protocol checker is as the following code snippet.

```c
T f_sync(this, <in>.., <out>..) {
    SETNEXT(f);
    PC_start(<ctx>, "f_sync", cond<in>..);
    T res = _next(this, <in>.., <out>..);
    PC_end(<ctx>, "f_sync", cond<res, <out>..>);
    return res;
}
```

`PC_start` and `PC_end` are functions from the protocol checker runtime library (e.g. `libPC.so`). Typically the constructor/destructor and all the object member functions
need to be intercepted. A complicating factor in this case is that the state machine of
the protocol checker, might depend on `condition`, which is a generic boolean condition
of the input or output variables.

7.3.3. Logger

The logger application is the simplest application and has almost always the same code.
A logger interceptor function is as following.

```c
T f_sync(this, <in>.., <out>..) {
    SETNEXT(f);
    LOGBEG(<in>..);
    T res = _next(this, <in>.., <out>..);
    LOGEND(res, <out>..);
    return res;
}
```

`LOGBEG` is a macro that logs the inputs of the function call. `LOGEND` logs the output
and the result of the function call when it returns.

7.3.4. Design decision

There is a proposed general guideline: the generated interceptor should compile and
build without any modification. This ensures early testing. A comparison of the three
templates are listed in Table 6.

<table>
<thead>
<tr>
<th></th>
<th>Number of intercepted functions in an interceptor source</th>
<th>External dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware probe</td>
<td>A few</td>
<td>Yes (to data manipulator such as libDDS)</td>
</tr>
<tr>
<td>Protocol checker</td>
<td>Almost all acf/ddf functions</td>
<td>Yes (to libPC)</td>
</tr>
<tr>
<td>Logger</td>
<td>A few</td>
<td>No</td>
</tr>
</tbody>
</table>

As can be seen in the previous analysis, different templates exist for different applications. Of the three templates, only logger template has no external dependency and can be built out of the box. It is preferred to use logger as the default template according to the guideline. As logger and hardware probe template only intercept a part of functions, changing logger template to probe does not require much effort. However, pro-
tocol checker intercepts almost all functions, and changing all logger functions to proto-

To conclude, the design decision for interceptor template is as follows:
1. Logger and hardware probe use the same logger template, as described in 7.3.3.
2. Protocol checker use a separate template as described in 7.3.2.
3. ddgen has two interceptor generation modes

7.4 Building and deploying interceptors
Building and deploying interceptors is no different from building other libraries. With the interceptor source files generated by ddgen, making a few changes in makefiles is enough to build interceptors. When interceptor libraries are built, they can be deployed on different test environments, such as devbench and testbench.

Within the ASML software development process, software patches (called devpatches) can be made, which are incremental changes that update existing software system. By putting the interceptor in a devpatch, the interceptors and their configuration can be installed on a test machine.

Design decision: use existing ASML software development tools to build and deploy interceptors.

7.5 Running with interceptors
.smjob files are used to load a software component on a given platform. For Linux/So-

laris, simple file system modifications are sufficient to get the interceptors applied at the proper locations. For VxWorks, .smjob files must be changed, to get the interceptors loaded in the correct order in the correct processes.
8. Implementation

Abstract – In this chapter, the implementation details of a protocol checker is described. The interceptor is located in a production component. The lessons learned within this section also apply for other two applications as well.

8.1 Introduction

The implementation descriptions follow the four steps mentioned in the design, which is generating the interceptors from acf/ddf, changing the null-interceptor, building and deploying the interceptors, and then running the TWINSCAN software with the interceptors.

8.2 Code generation

Based on the previous chapter, a new ddgen generator needs to be implemented with the ability to generate modified class skeletons (to avoid cyclic dependencies) and different interceptors for each application.

ddgen is an ASML internal tool to generate communication libraries (proxies and skeletons) to hide the (possibly inter process) communication between the client and (possibly remote) server object.
9. Verification and Validation

Abstract – This chapter gives information on the verification on the specifications.

9.1 Introduction

Verification and validation are independent procedures that are used together for checking that a product, service, or system meets requirements and specifications and that it fulfills its intended purpose. According to the PMBOK Guide and Standard\(^3\), a standard adopted by IEEE, they are defined as follows in its 4th edition:

- **Verification.** The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation.
- **Validation.** The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with verification.

The next two sections describe them separately.

9.2 Verification of interceptor

The interceptors can be verified against the requirements and ASML imposed conditions. The verification workflow can be depicted in Figure 29.

![Figure 29 Verification workflow](image)

The first step is to generate interceptors. This mainly concerns *ddgen*. The *ddgen* is modified for experimental purpose without thorough testing, thus the quality cannot be guaranteed. It is tested on one experimental component TATC, as well as one major production component.

The second step is to modify the interceptors for different applications. This varies from application to application. The default generated interceptor is a null interceptor. If users want to log parameters or return result, it can be done at this step.

---

\(^3\) PMBOK: The Project Management Body of Knowledge is a set of standard terminology and guidelines (a body of knowledge) for project management.
The third step is to **change configurations for building and deployment**, which includes *makefiles* and *.smjob* files.

**During testing on a devbench**, several early issues can be found. Most functional requirements can be verified on devbench, for example,

- Trace and log data (FR21)
- Function types that can be intercepted (FR3, FR4)
- Configuration of tracing and bootstrapping (FR20)

The last step is **testing on a testbench**. As a testbench has a test environment that is the closest to the production environment. **Functional requirements** should all be satisfied, for example, being able to call the original functions (FR1) on VxWorks. Most **non-function requirements** such as performance, can only be verified on a testbench because it is more time-realistic. Some requirements can be difficult to verify. For example, the requirement NFR12 requires that the addition of interceptors does not change system behavior. This cannot be guaranteed easily with one-time testing. However, this can be done by analyzing where the overhead comes from, and focusing on testing that locality of codes. Apart from that, automated tests should also be made to verify the interceptors to increase confidence level.

The internal verification also incorporated certain level of code review, mainly on *ddgen* and interceptor functions. This helps reduce moments of surprise.

### 9.3 Validation of interceptor

The validation of interceptors involves stakeholders. The validation is done in the form of demos, code reviews and pair programing with certain stakeholders. During such sessions, problems and caveats are discussed. This helped the stakeholders validate that it is what they want. Two main stakeholders, Will and Jinfeng, confirm that the interceptor project, gave usable results, that can be applied in some form within the TWINSCAN software.
10. Conclusions and Recommendations

Abstract – In this chapter, the results of the project are reviewed and conclusion is made.

10.1 Results

The experiments showed the scope of interception:
1. Interception on both client side and server side are possible
2. All client(ddf)/server(acf) interactions can be intercepted
3. Both shared (.so) and non-shared objects (.o/.a) can be intercepted

Integration within the legacy driver component proved:
1. Interceptors can be applied on legacy TWINSCAN code
2. Interceptors can fit within the ASML software development process
3. Interceptors is deployable on devbench/testbench test environments
4. That legacy code may contain extra cyclic dependencies that need to be broken
5. That the delay introduced by interceptors can be kept low by using early symbol lookup technique.

The following sections further discuss three major concerns when applying interceptors.

10.1.1 Concept soundness

The concept of library interception is basically a dynamic function extension mechanism applied in the ASML context. By intercepting an library, functions provided in that library can be altered or extended. This provides a simple and powerful way to add features such as profiling and tracing, protocol checking, attaching hardware simulators, without modifying any production code.

The performance degradation is negligible and can be further reduced (and predictable) with early symbol resolution.

Security wise, introducing interceptors involves splitting software into more libraries. This however, exposes more information to the outside world, namely the dynamic symbol table within each library. Customers may gain more knowledge on how the machine works.

Conclusion: Overall, the benefits of library interception outweigh the drawbacks.

10.1.2 Applicability of interceptors

Although interception can be applied in many cases, it might require changing some production code to remove cyclic dependencies. Removing cyclic dependencies is anyhow a good idea, irrespective of whether you want to apply interception or not.

Although the interceptors are tested only in one production component, it is expected they work also on other production components. That is because all ddf/acf expressible interactions are already tested in the test component TATC.

The identified problem of cyclic dependencies becoming external, arising from splitting of proxy/skeleton, and production libraries can be solved in a generic way, by modifying ddgen to split the skeleton code _class.c into q requiring part _class_rq.c and a providing part _class_pr.c.
10.1.3. Impact on existing codebase

The introduction of interceptors has several impacts on the existing codebase.

The **first** impact is on **glue code** (the proxy and skeleton libraries). The new *ddgen* should be updated to generate a new split skeleton (_class_pr.[ch] and _class_rq.[ch]) and separate backends for each application specific interceptor. It must preserve backward compatibility, such that if users do not want to intercept their object, nothing needs to be changed.

The **second** impact is on **production code** (i.e. _rq.c). All cyclic dependencies between glue and production code must be removed.

The **third** impact is on **some tests**. Since some production/glue code is changed, corresponding tests may also need to be updated.

10.2 Future work and recommendations

Due to the limited time span of the assignment, still some more directions can be explored, such as:
- generate non-ddf interceptors from library header files;
- generate client side interceptors;
- validate the interceptors on other production components.

With that being said, the applicability of interception has been proven to be very large, and the future work should not be difficult.

Interceptors are a great way to dynamically add development features for testing, debugging and prototyping certain tools. However, it also comes with the price of some amount of information exposure and some impact on legacy codebase.

**General recommendation:** interceptors can be applied on production components once the above considerations are taken into account.
11. Project Management

Abstract – In this chapter, how the project is managed is discussed.

11.1 Introduction
Project management is the practice of initiating, planning, executing, controlling, and closing the work of a team to achieve specific goals and meet specific success criteria at the specified time. A good management is critical to a successful project. This chapter describes how I managed the project.

11.2 The management process
Project management process usually can be grouped by stages of development. Different stages are identified and conducted.

11.2.1. Initiating and planning
The project proposal was already formulated by ASML at the start of the project. With the help of the university supervisor and the company supervisor, an early brainstorm chart was made.

After the initiation stage, the project is planned roughly to
- requirement gathering and analysis phase
- feasibility analysis phase
- design and integration phase
- verification phase.

Due to the highly exploratory nature of the project, the plan is rather high level, and an iterative way of planning and working is selected as project management methodology. The purpose of planning is to keep the project within the time bound. Project scope is determined in this stage.

Due to the exploratory attribute of the project, the plan is rather high granularity, and an iterative way of planning and working is selected as project management methodology. The purpose of planning is to keep the project within the time bound. Project scope is determined in this stage.

The planning is done using a Gantt chart.

11.2.2. Executing and documentation
The execution phase ensures the project plan is executed accordingly. The phase extended throughout the project, from stakeholder analysis to verification. The main deliverable are the proof of concept experiments. Their definition, results and needed codebase changes.

For documentation in 4 days per week is reserved for the experiments, and one day per week to document them. This ensures that insights gained from these experiments can be communicated to the stakeholders and define new experiments to explore new directions.

11.2.3. Monitoring and controlling
Monitoring and controlling consists of those processes performed to observe project execution so that potential problems can be identified in a timely manner and corrective action can be taken. This is facilitated by the project steering group (PSG).
Every Friday afternoon, a weekly report is sent to both supervisors. Keeping supervisors aware of the progress and issues is important. It also serves as a track record of progress and decision making. In addition, it helps myself to think, not only to do.

11.2.4. Closing
In the closing phase, the results are presented to relevant stakeholders. An executive summary and recommendations are provided. The retrospective of the project is also made of what went right and what went wrong, and how to prevent that in the future.

11.3 PSG meetings
PSG (Progress Steering Group) is a group of people navigating the project to the correct direction, which consists of three parties: university supervisor, company supervisor and the trainee. Monthly meetings are organized to discuss the progress, obstacles and plans. In case of any progress issues, the group is able to come up with a solution in a timely manner.

In total, seven PSG meetings were organized. During these meetings, some omitted topics are pointed out, such as define, execute, and document per experiment, or take security concerns into account.
12. Project Retrospective

Abstract – In this chapter, the reflection of the project is presented. This provides some ideas for future projects.

12.1 Technical reflection

The past nine months in ASML have been a valuable experience to me, not only the technical part but also the analytical/organizational part. Organizing meetings, presenting ideas and aligning between different disciplines were exercised many times. It very well complemented my PDEng training as a software designer.

The proof-of-concept project covered the whole design process, from gathering requirements, feasibility analysis, design and integration, to verification. Most of the things went smoothly, with some challenges and difficulties.

The first challenge was to embrace and understand the general ASML tooling. For example, the ClearCase toolset was sometimes confusing. Understanding streams, branches, baselines is important when testing in different environments.

The second challenge is to understand the software architecture and the components within it. ASML has many very helpful element product specifications and element design specifications. I needed to study the current system, to know which facilities were already provided to build upon. For example, when designing the logger I found that a tracing facility already satisfied each requirement. As a result, there is no need to reinvent the wheel, which saves tons of work.

The last challenge was integrating library interception into a production component. Creating a modified version of ddgen paved the way for this, which is challenging yet interesting. Even managed to generate the needed interceptors. Legacy code was found to have cyclic dependencies, hindering the application of library interception. With the help of domain expert, a workaround could be found such that library interception could be applied.

Fortunately, with the help of my supervisor and colleagues, the challenges and difficulties are timely tackled.

From an organizational point of view, schedule meetings or reserve resources weeks ahead. In a large organization, like ASML, people, rooms, and testbenches are very difficult to claim.

12.2 Design opportunities revisited

The design opportunities are covered in section 3.4. In the design, the focus was put on four design criteria: realization, genericity, elegance and documentation.

The realization criterion regards the certainty that the artefact can actually be realized. It was assured by conducting a feasibility analysis, considering all client-server interactions and various platform support.

Genericity criterion regards the extent to which the designed artefact can be re-used. It is expressed in that the interceptors can be applied to all ddf interfaces and non-ddf interfaces. It can be assumed that all ASML production software can use interceptors.

Elegance criterion is determined by inspecting the architecture of the artefact. Interceptors sit nicely between client side logic and proxy, server side skeleton and server logic. They exhibit great symmetry. Interceptors also couple to all interactions of both client side and server side, showing completeness. The logger application reuses the
existing tracing facility, demonstrating the beauty of self-consistency inside the system.

**Documentation criterion** is determined by creating deliverables that are correct, complete and readable. This is addressed by following the report template.
# Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interceptor</strong></td>
<td>A library, of which functions override functions from another library</td>
</tr>
<tr>
<td><strong>Devbench</strong></td>
<td>Testing platform used with software-in-the-loop simulation within the company</td>
</tr>
<tr>
<td><strong>Testbench</strong></td>
<td>Testing platform used with hardware-in-the-loop simulation within the company</td>
</tr>
<tr>
<td><strong>VxWorks</strong></td>
<td>A real-time operating system (RTOS) developed as proprietary software by Wind River Systems</td>
</tr>
<tr>
<td><strong>Acf</strong></td>
<td>ASML class file, defines a class</td>
</tr>
<tr>
<td><strong>Ddf</strong></td>
<td>Data definition file, defines interfaces</td>
</tr>
<tr>
<td><strong>Ddgen</strong></td>
<td>A code generator that generates data communication libraries for different languages.</td>
</tr>
<tr>
<td><strong>SUT</strong></td>
<td>System under test</td>
</tr>
<tr>
<td><strong>ELF</strong></td>
<td>(Executable and Linkable Format)</td>
</tr>
</tbody>
</table>
Appendix: _next function on VxWorks

```c
#ifdef VXWORKS
static BOOL iterator(
    char *name, /* symbol/entry name */
    int val,   /* symbol/entry value */
    SYM_TYPE type,  /* symbol/entry type */
    int arg,    /* arbitrary user-supplied arg */
    UINT16 group /* symbol/entry group number */
) {
    static BOOL firstFound = FALSE;
    BOOL result = TRUE; /* continue iteration */
    char *pName; 

    pName = (char *)&arg;
    if ((pName != NULL) && (name != NULL))
        if (!strcmp(name, pName))
            if (firstFound) {
                /* symbol found => stop iteration! */
                result = FALSE;
                firstFound = FALSE;
            }
            else {
                firstFound = TRUE;
            }
        }
    return (result);
}

void *funcNext(
    const char * name /* symbol to search for */
) {
    SYMBOL *sym;
    sym = symEach(sysSymTbl,(FUNCPTR)iterator, (int)name);
    if (sym != NULL) {
        return sym->value;
    }
    else {
        fprintf(stderr, "NOT FOUND\n");
    }
    return NULL;
}
#define NEXT(fnam) typedef typeof(fnam) T; static T* _next =
    if (!_next) _next = (T*)funcNext(__func__);#endif
```
Bibliography

References


VXWORKS® 7 PROGRAMMER'S GUIDE EDITION 17

VXWORKS® 7 KERNEL SHELL USER'S GUIDE
About the Authors

Qing Cai received his master degrees in Embedded System Engineering from ESIGELEC in Rouen, France and Computer System Architecture from University of Shanghai for Science and Technology in China through a double degree program. He interned at PSA Peugeot Citroen for the project Personalized Navigation System built upon TomTom’s map SDK.

Apart from embedded software, Qing is also interested in developing map and location based software. He published an iOS app called Traces, which helps users remember where they have been in the past. An accompanying website, based on React, was developed and open-sourced on Github. He gained tremendous happiness in programming.