

**AUTOSAR I/O GUI:  
Eclipse-Based Visualization and Test Access  
to a Configurable Automotive Driver  
Framework**

**Study Thesis in Computer Sciences**

by

***Wanja Hofer***

born 04/29/1983 at Ludwigshafen/Rhein

Department of Computer Sciences 4  
Friedrich-Alexander University, Erlangen-Nuremberg

Advisors:       Dipl.-Ing. (FH) Felix Fastnacht, Elektrobit Automotive  
                  Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat  
                  Dr.-Ing. Jürgen Kleinöder  
                  Dipl.-Inf. Fabian Scheler

Start of work:   05/30/2006

End of work:    01/30/2007



Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, \_\_\_\_\_

## **Kurzzusammenfassung**

Die Simulation von AUTOSAR-basierten eingebetteten Systemen auf PC-Basis ist nützlich sowohl für Software-Entwickler, die AUTOSAR-Anwendungen schreiben, als auch für die Entwickler der AUTOSAR-Implementierung selbst.

Die vorliegende Studienarbeit untersucht die Möglichkeiten, ausgewählte Module der AUTOSAR-Treiberschicht zu simulieren und wie die zugehörige Pseudo-Hardware graphisch dargestellt werden kann; dabei wird die Konfigurierbarkeit des AUTOSAR-Frameworks berücksichtigt. Außerdem wird das Design eines Script-Zugangs analysiert und entworfen, um extensives, automatisiertes Testen der höheren AUTOSAR-Schichten, inklusive der Applikation, zu ermöglichen.

Das entwickelte Programm – AUTOSAR I/O GUI – ist durch verschiedene Mittel des Architekturentwurfs so generisch wie möglich gehalten, um eine einfache Erweiterbarkeit der prototypenhaften Arbeit zu gewährleisten.

## **Abstract**

The simulation of an AUTOSAR-based embedded system on PC basis is very useful to developers of AUTOSAR applications and to those developing the AUTOSAR implementation itself.

The present study thesis investigates the feasibility to simulate selected modules of AUTOSAR's driver layer and how to visualize the attached pseudo devices, bearing the configurability of the AUTOSAR framework in mind. It also analyzes and designs a script access possibility to the framework in order to allow extensive automated testing of the upper AUTOSAR layers, including the application.

The resulting framework program—AUTOSAR I/O GUI—is kept as generic as possible through different means of architecture design to provide for easy extensibility of the prototype work.



---

---

# Contents

---

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Overview . . . . .	1
1.2	Structure of This Thesis . . . . .	2
1.3	Goals of the Project . . . . .	2
1.4	Related Work . . . . .	3
1.5	Requirements . . . . .	4
1.6	Basics . . . . .	6
1.6.1	About AUTOSAR . . . . .	6
1.6.2	About Eclipse . . . . .	7
1.6.3	About <i>tresos</i> ECU . . . . .	8
1.6.4	About Python . . . . .	9
1.7	Timeline . . . . .	10
<b>2</b>	<b>Analysis of Selected AUTOSAR Modules</b>	<b>13</b>
2.1	Introduction to AUTOSAR's Microcontroller Abstraction Layer	13
2.1.1	The AUTOSAR Modules . . . . .	14
2.1.2	Storage of the Simulation States . . . . .	15
2.1.3	Encountered Problems in the AUTOSAR Specification	17
2.2	DIO Analysis . . . . .	18
2.2.1	The I/O Driver Class . . . . .	18
2.2.2	DIO Terminology . . . . .	19
2.2.3	DIO Example Use Case . . . . .	19
2.2.4	DIO Simulation Interfaces . . . . .	20
2.3	CAN Analysis . . . . .	21

## CONTENTS

---

2.3.1	The Communication Driver Class . . . . .	21
2.3.2	CAN Simulation Scenarios . . . . .	22
2.3.3	CAN Example Use Case . . . . .	23
2.3.4	CAN Simulation Interfaces . . . . .	24
2.4	EEPROM Analysis . . . . .	25
2.4.1	The Memory Driver Class . . . . .	25
2.4.2	EEPROM Example Use Case . . . . .	26
2.4.3	EEPROM Simulation Interfaces . . . . .	27
2.5	DET Analysis . . . . .	28
2.5.1	Introduction . . . . .	28
2.5.2	DET Simulation Interfaces . . . . .	29
2.6	Summary . . . . .	30
<b>3</b>	<b>Analysis of the Program Environment</b>	<b>31</b>
3.1	Programming Language and Platform . . . . .	31
3.2	GUI Toolkits . . . . .	32
3.2.1	AWT and Swing . . . . .	32
3.2.2	SWT and JFace . . . . .	33
3.2.3	GEF and Draw2D . . . . .	34
3.2.4	Conclusion . . . . .	35
3.3	Tests and Scripting Access . . . . .	35
3.3.1	Example Use Case: Scripting Access . . . . .	36
3.3.2	Domain-Specific Languages . . . . .	37
3.3.3	Embedding Scripting Languages . . . . .	38
3.3.4	Jython . . . . .	39
3.4	Communication AUTOSAR I/O GUI-Simulation . . . . .	40
3.4.1	Java Native Interface . . . . .	40
3.4.2	Shared Memory or Memory-Mapped File . . . . .	41
3.4.3	CORBA . . . . .	41
3.4.4	Sockets . . . . .	42
3.4.5	Conclusion . . . . .	42
3.5	Summary . . . . .	43
<b>4</b>	<b>Design and Implementation</b>	<b>45</b>
4.1	Program Architecture . . . . .	45
4.1.1	The Central State Database . . . . .	46
4.1.2	The Communication Thread . . . . .	48
4.1.3	Micro Architecture of Model, EditPart, and Figure . . . . .	48
4.1.4	Threads and Thread Synchronization . . . . .	50
4.1.5	Effects of the Currently Active Configuration . . . . .	52
4.2	Reusing Design Patterns . . . . .	53



4.2.1	Model–View–Controller . . . . .	53
4.2.2	Command . . . . .	55
4.2.3	Singleton . . . . .	55
4.2.4	Memento . . . . .	56
4.2.5	Observer . . . . .	56
4.3	Recording of Test Scripts . . . . .	56
4.3.1	Short Introduction to Aspect-Oriented Programming . . . . .	57
4.3.2	AUTOSAR I/O GUI’s <code>RecordingAspect</code> . . . . .	57
4.4	Genericity in AUTOSAR I/O GUI . . . . .	58
4.4.1	Providing Abstract Base Classes for <code>Model</code> , <code>EditPart</code> , and <code>Figure</code> . . . . .	58
4.4.2	Saving and Restoring . . . . .	59
4.4.3	Implementation of the Script Recording Feature . . . . .	60
4.5	Definition of the Message Protocol Between AUTOSAR I/O GUI and the Simulation . . . . .	61
4.5.1	Message Format . . . . .	61
4.5.2	Simulation Proxy . . . . .	63
4.6	User Interface Design . . . . .	63
4.6.1	Goals of Human Factors . . . . .	63
4.6.2	Considering Multiple Scenarios . . . . .	64
4.6.3	User Analysis . . . . .	64
4.6.4	Consistency . . . . .	65
4.6.5	Icons and Redundancy . . . . .	65
4.6.6	Display Layout . . . . .	66
4.7	Summary . . . . .	67
<b>5</b>	<b>Summary and Prospects</b> . . . . .	<b>69</b>
5.1	Summary of the Results . . . . .	69
5.2	Future Work . . . . .	70
5.2.1	Displaying Internal Information . . . . .	70
5.2.2	Connecting Several Instances of AUTOSAR I/O GUI to One Simulation . . . . .	70
5.2.3	Modifying the MCAL on the Target to Communicate With AUTOSAR I/O GUI . . . . .	71
5.2.4	Using AUTOSAR I/O GUI for Software Component Tracing . . . . .	71
<b>A</b>	<b>Features</b> . . . . .	<b>I</b>
A.1	Installing AUTOSAR I/O GUI . . . . .	I
A.2	Invoking AUTOSAR I/O GUI . . . . .	II
A.3	AUTOSAR I/O GUI’s User Interface . . . . .	II

## CONTENTS

---

A.3.1	The GUI . . . . .	II
A.3.2	Keyboard Access . . . . .	III
A.4	Populating the Displayed Diagram . . . . .	III
A.5	Saving the Populated Diagram . . . . .	IV
A.6	Managing the Connection to the Simulation . . . . .	V
A.7	Replaying Test Access Scripts . . . . .	V
A.8	Recording Test Access Scripts . . . . .	V
<b>B</b>	<b>How to Introduce a New Control</b>	<b>VII</b>
B.1	Defining Custom Model, EditPart, and Figure . . . . .	VII
B.1.1	Defining the Model Class . . . . .	VII
B.1.2	Defining the EditPart Class and Adding It to the Factory . . . . .	VIII
B.1.3	Defining the Figure Class . . . . .	VIII
B.2	Adapting the Internal Configuration . . . . .	IX
B.2.1	Adapting the Configuration Class . . . . .	IX
B.2.2	Adapting the Parsing Step . . . . .	IX
B.3	Extending the Model Factory . . . . .	IX
B.4	Extending the Palette . . . . .	IX
B.5	Upgrading the Central State Database and the Access Interfaces	X
B.6	Adapting the AUTOSAR Core Host Data Interface and the Communication Thread . . . . .	X
B.7	Extending the RecordingAspect Pointcut Expression . . . . .	XI
B.8	Miscellaneous Hints . . . . .	XI
<b>C</b>	<b>Definition of Exchanged Messages</b>	<b>XIII</b>
C.1	DIO Messages . . . . .	XIII
C.2	CAN Messages . . . . .	XIII
C.3	EEPROM Messages . . . . .	XV
C.4	DET Messages . . . . .	XV
C.5	Service Messages . . . . .	XV
	<b>Bibliography</b>	<b>XIX</b>
	<b>List of Figures</b>	<b>XXV</b>
	<b>Glossary</b>	<b>XXVII</b>

# Introduction

---

## 1.1 Overview

---

Elektrobit Automotive<sup>1</sup> is one of the premium members of the AUTOSAR industry initiative and therefore involved in the process of standardization of embedded systems components. The implementation of the developed standards is one of the current main points of focus, also driven by the high demand of customers seeking to deploy their systems on the proposed AUTOSAR platform.

Since the simulation of a target embedded system on a host PC is of very high interest to the customers for testing purposes, this study thesis and another diploma thesis at Elektrobit Automotive investigate the possibilities and feasibility of implementing both the operating system and parts of the hardware and driver layer using a regular PC operating system like Microsoft Windows or Linux. The necessary work is divided up into two parts consisting of the simulation itself (developed in the other diploma thesis) and of the visualization and a test access possibility developed in this thesis.

The document prototypically explores selected driver modules in order to design an architecture suited to fit the additional requirements as defined by Elektrobit Automotive.

---

<sup>1</sup>Formerly 3SOFT.

## 1.2 Structure of This Thesis

---

This introductory chapter presents the reader with the aims of the project as well as with the environment and the resulting outer requirements. It also looks for related work and introduces the knowledge about the utilized tools that is necessary so that the reader can thoroughly follow the rest of the text.

The first part of the analysis ([Chapter 2](#)) introduces AUTOSAR’s driver layer and closely investigates it in order to select the prototype modules. These are further examined, use cases are developed, and interfaces for the design are determined.

The second analytical chapter ([Chapter 3](#)) looks for environmental conditions and conditions resulting from the requirements in order to be able to narrow down the possibilities for feasible design decisions.

These decisions are made and justified in [Chapter 4](#) on the design and implementation of the resulting program. Interesting and challenging parts of the architecture are presented as well as deployed software design patterns and genericity mechanisms. Decisions affecting the user interface design are also justified at that place. Moreover, the interface to the actual simulation is defined there—in agreement between the two interconnected pieces of work.

[Chapter 5](#) sums up the results of the thesis and gives an outlook on connected work in the future and possible developments.

[Appendix A](#) gives an overview of the features of the resulting program and can be considered a small user guide. Crucial information for the party extending the prototype in the future is given in form of a step-by-step explanation in [Appendix B](#). The concrete messages that are exchanged with the simulation are listed in [Appendix C](#) for reference purposes.

The document is concluded with the references listed in a [Bibliography](#), a [List of Figures](#), and a [Glossary](#) explaining the used acronyms and selected terms that are heavily used is available at the end of this document.

## 1.3 Goals of the Project

---

The program to be designed and implemented in this thesis was baptized “AUTOSAR I/O GUI”<sup>2</sup> (hence the title of the thesis) because it is supposed to provide a graphical interface to the user enabling him to perceive and tweak the input and output interface of AUTOSAR’s MCAL (microcontroller

---

<sup>2</sup>The implementation therefore often uses the abbreviation AIOGUI as a prefix for class names.

abstraction layer) and providing an access point for automated testing.

The project’s target audience are both software engineers developing modules of the upper AUTOSAR layers and software developers building applications using the whole AUTOSAR framework. Both classes of users benefit from an MCAL simulation on PC basis in order to enable rapid development and testing without the need to deploy their programs on the intended target. Since currently there is support for only few platforms (i.e. MCAL implementations are on-hand), the simulation is of high interest to those customers wishing to begin developing applications before the target implementation is available.

The three identified fields of application to keep in mind are

1. tests of basic applications using the AUTOSAR framework,
2. functional tests of newly introduced upper layer AUTOSAR modules,
3. visualization and input manipulation in an educational context like training courses, for example.

## 1.4 Related Work

---

Due to the fact that the AUTOSAR standard is still very new, no alternative to AUTOSAR I/O GUI is on the market (yet). Still there is one program on-hand containing resembling functionality at first glance.

“OSEK Simulation GUI” was developed by Elektrobit Automotive’s own Thomas Seydel in his diploma thesis [1]. It interchanges data with a Win32 OSEK implementation which is an operating system without any drivers (unlike AUTOSAR).

In order to interact with the GUI the application needs to be modified to include specific header files and execute specific calls to the GUI. This is a fundamental difference to the goal of AUTOSAR I/O GUI and the actual simulation where the aim is to leave the application and the upper AUTOSAR layers untouched.

It also implements arbitrary controls that are determined independently—without being oriented at a driver interface like AUTOSAR I/O GUI is supposed to be.

## 1.5 Requirements

---

Beginning with the abstract project goals (see [Section 1.3](#)) a comprehensive requirements list for AUTOSAR I/O to meet was compiled by Elektrobit Automotive and the author; many parts of the thesis refer to these entries. In the following specification the semantics of the key words as specified by RFC 2119 by the Internet Engineering Task Force [2] are used. Requirements 1 to 10 are mandatory to meet, 11 through 14 are optional.

1. It shall be possible to use widely available PC systems as simulation hosts.  
*Rationale:* PC systems dominate the market of workstations that are used for developing purposes by the customers.
2. The simulation and AUTOSAR I/O GUI shall be decoupled and clearly separated.  
*Rationale:* It shall be possible to run the simulation without executing AUTOSAR I/O GUI in parallel. It shall also be possible to maintain both programs separately from each other.
3. There shall exist the possibility to access the simulation in an automated kind of way by the use of a scripting language. This test access method shall be available concurrently to the GUI visualization.  
*Rationale:* An automated access method is needed for comprehensive and reproducible software tests.
4. The scope of the simulation and the attached AUTOSAR I/O GUI shall be a functional one only, that is timing aspects are of no particular concern.  
*Rationale:* The timing behavior can normally only be measured on the target itself and is not relevant to tests in the earlier state of simulation.
5. AUTOSAR I/O GUI shall be easily extensible, enabling the developer to program custom kinds of visualization.  
*Rationale:* The author or another Elektrobit Automotive party will continue to develop the prototype to make it suitable for delivery to customers.
6. AUTOSAR I/O GUI shall visualize the specific kinds of output data in an appropriate kind of way.  
*Rationale:* Every type of output data needs different representation according to its nature. For example, a single bit value may be well

represented by a color whereas this is not a good choice for a 16 bit value.

7. AUTOSAR I/O GUI shall enable its user to specify and modify the different kinds of input data in an appropriate kind of way.

*Rationale:* Every type of input data needs different representation according to its nature. For example, a single bit value may be well represented by a check box to set its value whereas this is not a good choice for a 16 bit value.

8. The driver configuration data shall be retrieved from the *tresos* data base.

*Rationale:* AUTOSAR I/O GUI needs to adapt to the different scenarios emerging from the configuration possibilities in the *tresos* GUI configuration program (see [Section 1.6.3](#)). It shall offer only the subset of features made available in the selected configuration.

9. The user shall be enabled to save and restore modifications he performed in AUTOSAR I/O GUI.

*Rationale:* Once a layout (e.g. for demonstration purposes) is established, it shall be possible to restore that layout without going through the same arranging steps again.

10. The thesis shall investigate and implement only selected AUTOSAR drivers.

*Rationale:* AUTOSAR I/O GUI at this stage shall only be a framework program to eventually integrate all of the drivers. The implemented drivers are a proof of concept.

11. The documentation should be written in English language.

*Rationale:* Both developers and users of AUTOSAR I/O GUI are resident all over the world.

12. AUTOSAR I/O GUI should be independent of the underlying operating system (i.e. at least support Microsoft Windows and Linux).

*Rationale:* Users of AUTOSAR I/O GUI should not be forced to use a specific operating system.

13. Performance should be an issue of concern.

*Rationale:* There should not be a big overhead running AUTOSAR I/O GUI.

14. The recording of user interaction with the GUI in order to generate a test script should be possible.

*Rationale:* The user should be assisted in writing test scripts.

Since this requirements outline is rather abstract in respect to the concrete design of AUTOSAR I/O GUI, many decisions—also concerning the environment—are to be taken independently. These decisions are documented in parts of the analytical Chapters 2 and 3 and in Chapter 4 on the design and implementation.

## 1.6 Basics

---

This section gives a short survey of topics of particular importance to AUTOSAR I/O GUI and pitches on the relevant issues. It is supposed to be a short reference that is backreferenced by several of the following chapters. The reader is also given several directions for further reading.

### 1.6.1 About AUTOSAR

AUTOSAR is an open standards organization created to provide an open source architecture to serve as an infrastructure for the management of functions within applications and standard software modules [3]. Its name is an acronym for “Automotive Open System Architecture” and its members include several automotive manufacturers and suppliers, including Elektrobit Automotive.

Influences from the very successful OSEK standard can particularly be seen in the AUTOSAR OS standard which is based on the OSEK OS standard specifying a static real time operating system [4] configured at compile time. The AUTOSAR framework is also supposed to be highly configurable and therefore customizable which results in a basic condition for AUTOSAR I/O GUI.

The declared goal of the AUTOSAR initiative is to establish an open industry standard for automotive electrics/electronics architectures. The need for a standard derives from the increasing complexity of the deployed architectures and its software resulting from the growth in functional scope in the automobile. Thus the main goal is the ability to develop software components independently from the actual lower infrastructure—including the possibly distributed ECUs and their communication channels or the possibility of a target simulated on a PC host.



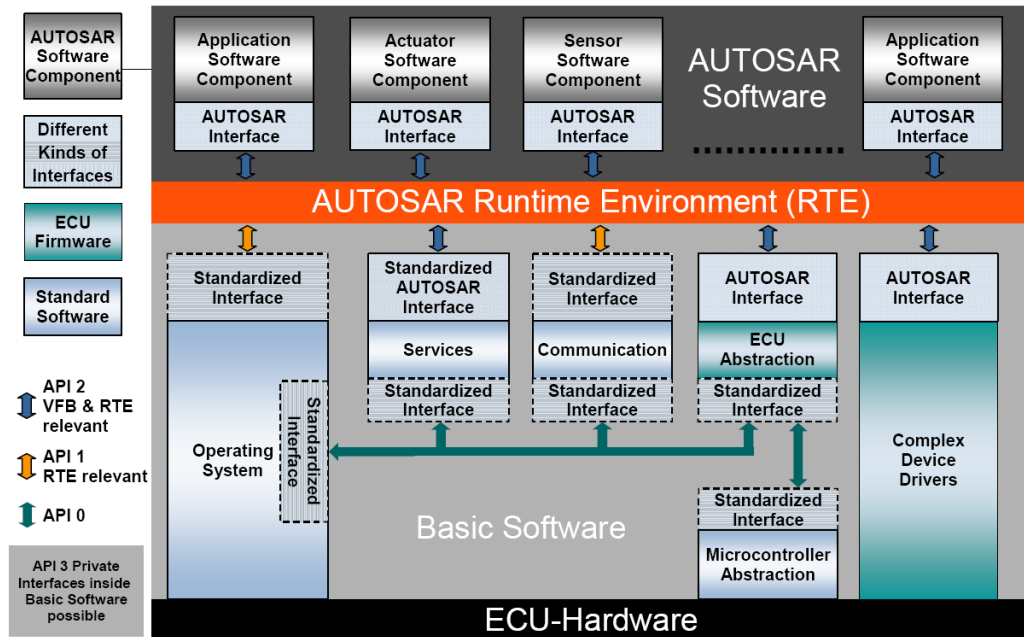


Figure 1.1: AUTOSAR software architecture [5].

The architecture proposed by the AUTOSAR standard contains several layers (see [Figure 1.1](#)) including the software application itself—consisting of several AUTOSAR Software Components that are mapped to different ECUs. These components communicate through abstract ports which are routed through the Runtime Environment (RTE) internally using inter-ECU communication channels or intra-ECU communication mechanisms appropriately. The Basic Software (BSW) block provides services to the components including communication, an operating system, and an abstraction layer to the underlying microcontroller or simulated microcontroller.

Besides that, AUTOSAR also elaborates a methodology describing the steps in the development process from the configuration of the system to the generation of ECU executables; it also investigates the feasibility to automate some of the steps using an appropriate tool chain.

For a more detailed introduction to the technical aspects of AUTOSAR, please consider the “Technical Overview” document [5].

## 1.6.2 About Eclipse

The developers of the Eclipse platform say it is an IDE for anything, and for nothing in particular [6]. Due to its highly complex plug-in architecture it is extremely generic and therefore versatile [7]. Besides the possibility to write a

plug-in for the Eclipse platform itself it is possible to integrate it into an RCP (rich client platform) that is based on Eclipse. *tresos* GUI (see [Section 1.6.3](#)) is such an Eclipse-based RCP and therefore suited to seamlessly integrate Eclipse-based plug-ins.

### On Eclipse’s Plug-in Mechanism.

Plug-ins in the Eclipse platform may be related by either a dependency or an extension relationship. The former merely indicates that the dependent plug-in makes use of the prerequisite plug-in while the latter defines a more sophisticated coupling of the so-called extender plug-in and host plug-in. Host plug-ins define extension points that are to be addressed by the extender plug-in’s extension, thereby effectively adding some processing element to it. All of this is done in a configuration file named `plugin.xml` located in the plug-in’s base directory.

AUTOSAR I/O GUI makes use of this mechanism by contributing extensions to the workbench’s menubar and toolbar.

For a more thorough overview of Eclipse’s plug-in mechanism, see the article “Notes on the Eclipse Plug-in Architecture” [8].

### On Libraries Provided by Eclipse.

Eclipse also provides GUI libraries that are naturally of great concern when developing a GUI program like AUTOSAR I/O GUI.

The details of the toolkits named Standard Widget Toolkit (SWT) and JFace are discussed in [Section 3.2.2](#) in the context of the decision for a toolkit and in comparison to other toolkits.

## 1.6.3 About *tresos* ECU

*tresos* ECU is the product name of Elektrobit Automotive’s family of integrated products to develop embedded automotive software. It is a framework for the different modules and provides a central storage facility for the configuration data in order to avoid consistency problems.

The biggest part of *tresos* ECU are its actual software components. These include:

- AUTOSAR Suite: Elektrobit Automotive’s implementation of the AUTOSAR Standard Core consisting of the service, hardware abstraction, and microcontroller abstraction layer modules

- AUTOSAR OS: Elektrobit Automotive’s implementation of an AUTOSAR compliant operating system based on OSEK
- Other modules not (yet) in the context of AUTOSAR like OSEKtime and cryptography plug-ins

*tresos* GUI is the application enabling the user to maintain the configurations in a graphical environment without the need to edit numerous XML files by hand. After the configuration of the whole project has been performed, it can be automatically verified for consistency between the independent but interacting modules. If that step is passed, the proper code is generated and prepared for compilation. After successful compilation the resulting code can be linked to the application code to form the system image to be loaded on the target microcontroller unit.

The GUI is an open platform that allows arbitrary configuration masks to be integrated—and also plug-ins with non-configuration functionality like AUTOSAR I/O GUI.

#### 1.6.4 About Python

Python<sup>3</sup> is an easy to learn, powerful programming language [9], so its author Guido van Rossum claims. It is typed dynamically and provides for automatic memory management, thereby making available an ideal environment for the test programmer wishing to deploy his test scripts rapidly.

It is also possible to program in an object-oriented kind of way in Python, thus enabling the user to manage the complexity of even very comprehensive test scripts. Moreover, the abstraction and inheritance feature provided by object-oriented languages allows the writing of very abstract test scripts hiding the concrete function calls (e.g. in AUTOSAR I/O GUI by defining a `Car` class containing functions like `startEngine ()` or `brake ()`). Also the known object-oriented programming techniques like polymorphism and encapsulation can be exploited.

Other main advantages of Python over other scripting languages include the following [12]:

- Python is open source software. There are no restrictions to using it in own software products (like AUTOSAR I/O GUI) and still there is

---

<sup>3</sup>The author named the language in admiration to Monty Python [9], a group of British comedians. They themselves chose the name of the snake merely because it sounded funny to them [10]. To the readers not speaking English natively: The “th” in Python is pronounced as a voiceless dental non-sibilant fricative like in the word “bath” and not as a voiceless alveolar plosive like in “water” [11] which is a common mistake.

a huge community providing support (e.g. to the users developing test scripts for AUTOSAR I/O GUI).

- Python is portable. Python programs compile to a kind of portable bytecode that runs in the same way on each supported platform. That means that libraries developed on a foreign platform (and widely available on the internet) can be used in the test scripts.
- Python is powerful. It includes comprehensive built-in operations (e.g. for string manipulations) and an abundant amount of libraries for various purposes is offered to its users.

## 1.7 Timeline

---

The following schedule represents a mixture of both planned activities determined in the very first phase of the work and activities that were added during the evolution of this thesis because they followed only from the results of previous activities (e.g. getting acquainted with Eclipse and GEF after deciding to pick those frameworks when determining the GUI visualization)—it was both an intention plan in advance and is now a summary with hindsight.

### 1. Orientation (2 weeks)

- Getting acquainted with AUTOSAR, its architecture, and its driver classes
- Selecting AUTOSAR drivers to be further investigated and implemented in the prototype (see [Chapter 2](#))
- Researching possibilities to visualize certain kinds of data produced or needed by the drivers
- Looking for GUI toolkits and scripting possibilities and evaluating them
- Discussing the interface between AUTOSAR I/O GUI and the simulator (the AUTOSAR Core Host Data Interface)

### 2. Analysis (2 weeks)

- Definition of the AUTOSAR Core Host Data Interface (see [Section 2.1](#), [Section 3.4](#), and [Section 4.5](#))
- Determination of the GUI visualization (see [Section 3.2](#))

- Determination of the test access (see [Section 3.3](#))
  - Development of application use cases (see [Sections 2.2](#) through [2.5](#) and [Section 3.3](#))
  - Getting acquainted with Eclipse and GEF in particular
3. Design (3 weeks)
- Development of the system architecture (see [Section 4.1](#))
  - Considering constraints given by the frameworks in use
  - Design of the user interface (see [Section 4.6](#))
4. Implementation (5 weeks)
- Implementation of the prototype according to the previously specified design
  - Integration of the prototype into *tresos* GUI
5. Documentation (4 weeks)
- Writing a short feature overview of the AUTOSAR I/O GUI program (see [Appendix A](#))
  - Writing documentation for Elektrobit Automotive on how to extend AUTOSAR I/O GUI (see [Appendix B](#))
  - Finish writing this thesis



# Analysis of Selected AUTOSAR Modules

---

The following two analytical chapters lay the groundwork for the program design in the subsequent [Chapter 4](#).

This chapter introduces AUTOSAR's Microcontroller Abstraction Layer and how it is relevant to AUTOSAR I/O GUI, and exposes some of the problems of the AUTOSAR specification documents faced ([Section 2.1](#)). It then dives deep into the analysis of selected AUTOSAR modules ([Sections 2.2 through 2.5](#)), seeking for use cases and interfaces in AUTOSAR I/O GUI to be used for the design and implementation described in [Chapter 4](#).

[Section 2.6](#) rounds up the chapter by giving a short summary of the results.

## 2.1 Introduction to AUTOSAR's Microcontroller Abstraction Layer

---

The microcontroller abstraction layer (MCAL) of the AUTOSAR standard corresponds to a hardware abstraction layer; it therefore defines an interface which can be used by the upper layers of the architecture to access the microcontroller and its devices (see [Figure 2.1](#)).

The simulation is supposed to provide an implementation of the MCAL on PC basis (see also [Requirement 1](#) in [Section 1.5](#)) simulating the devices with the help of AUTOSAR I/O GUI. AUTOSAR I/O GUI can then ask its

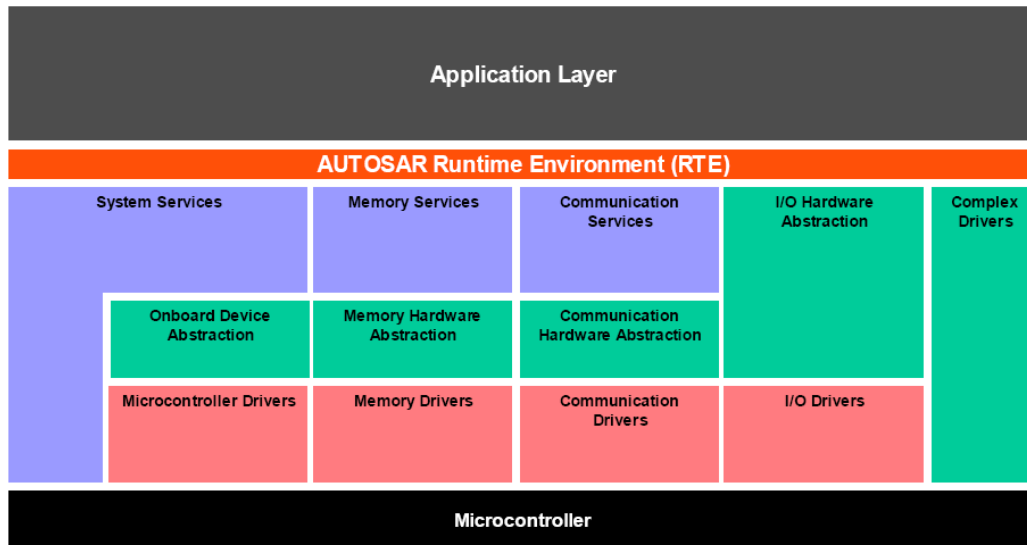


Figure 2.1: AUTOSAR driver classes [13].

user to supply the needed data (input direction) and show data provided by the application (output direction).

### 2.1.1 The AUTOSAR Modules

The AUTOSAR architecture divides the core drivers in the MCAL into four classes (see also [Figure 2.1](#)):

1. I/O (input/output) drivers
2. Communication drivers
3. Memory drivers
4. Microcontroller drivers

This thesis is supposed to represent a prototype implementation, therefore only few drivers are picked to be analyzed, and only some of them are actually implemented (see also [Requirement 10](#) in [Section 1.5](#)). The analysis and implementation of all defined modules is beyond the scope of this thesis and possibly the content of a student trainee job by the author at Elektrobit Automotive.

Besides the driver classes that are further analyzed in the following sections, AUTOSAR also specifies a microcontroller driver class—those drivers specific and internal to a microcontroller—including a general purpose timer



driver, a microcontroller unit driver and a watchdog driver. Since none of those is suited to be visualized (input or output), this driver class is not further considered from now on.

The following Sections 2.2 through 2.4 discuss the selected drivers and the reasons for picking them out of their driver classes. The Development Error Tracer (DET) discussed in Section 2.5 has an exceptional position since it is not a real driver but of interest to the user of AUTOSAR I/O GUI anyway. All of the sections provide background knowledge where necessary for the discussion, and give an example use case to clarify the scenario using AUTOSAR I/O GUI.

The most important part of each section, however, is the section analyzing which information needs to be exchanged through the AUTOSAR Core Host Data Interface. The AUTOSAR Core Host Data Interface is the interface between the simulation and the GUI; its environment is depicted in Figure 2.2. The way that the two communicating parties are connected is discussed in Section 3.4 of the next chapter, and the concrete interface messages are defined in Section 4.5 and Appendix C—based on the analysis of the interfaces in the next sections. By agreeing on the interface between the simulation and AUTOSAR I/O GUI in advance, both of them can be developed independently and a step towards decoupling is made by design (see Requirement 2 in Section 1.5).

The AUTOSAR MCAL specification documents mentioned in each analysis part contain the concrete interfaces for the driver module (in this case the module in the simulation) to implement. They also provide Service IDs for each of the functions in the interface that AUTOSAR I/O GUI makes use of and extends by custom defined ones (see Section 4.5).

### 2.1.2 Storage of the Simulation States

Decisions concerning the design architecture are generally taken only after the analysis parts (Chapters 2 and 3), but there is one condition that has to be determined in advance because it is needed for the analysis: where to store the simulated states.

There are three possibilities that can be distinguished:

1. Simulation only: both input and output states are stored on simulation side. This way, AUTOSAR I/O GUI submits input changes whenever they occur, and has to poll for output state changes.
2. GUI only: both input and output states are stored on GUI side. This way, the simulation submits output changes whenever they occur, and

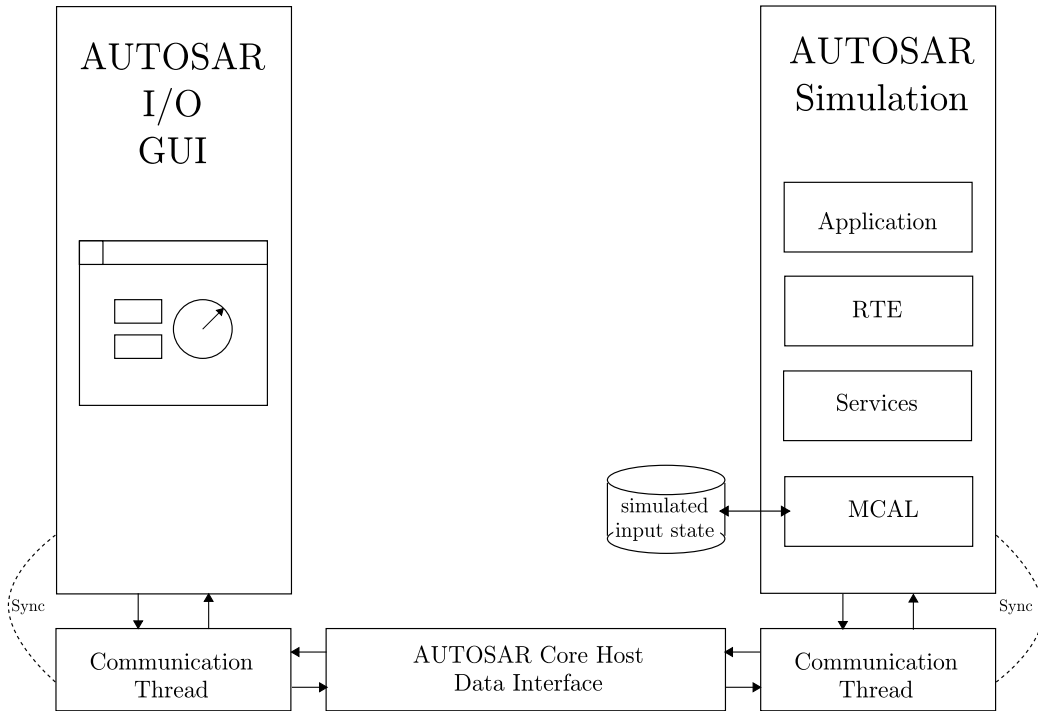


Figure 2.2: The AUTOSAR Core Host Data Interface.

has to poll for input state changes (everytime the application requests it).

3. Simulation and GUI: the two parties inform each other everytime a state change occurs so that the other part can store the updated value and inform a connected entity, if applicable.

The author and the simulation party agreed upon the third approach— notifying each other asynchronously at every state change. Since these changes happen less frequently than requests for them, this is the most economical solution expected to burden the AUTOSAR Core Host Data Interface the least.

This way, the MCAL of the simulation can resort to a local simulated state storage (see also [Figure 2.2](#)) whenever it is supposed to provide an input value; the storage is kept up-to-date through the AUTOSAR Core Host Data Interface.

### 2.1.3 Encountered Problems in the AUTOSAR Specification

Because of the AUTOSAR standard not yet being completely mature, there are some stumbling blocks encountered during the analysis of the AUTOSAR documents on the way to a program design. Three of those are documented here in order to give an example.

#### Confusing Mistakes in the Specification.

Slips encountered in the AUTOSAR documents are unfortunately not very rare and especially harmful in the period of orientation in the AUTOSAR world. Sentences containing mistakes like “A [DIO channel] group consists of consecutive channels from a single DioChannel.” [14] can easily be read over by the experienced audience but is very confusing for the AUTOSAR novice.

#### Addressing of DIO Channel Groups.

Another issue—regarding the specification contents—concerns the addressing of a DIO channel group (see [Section 2.2](#) for an introduction to the module and the terminology). The DIO driver specification document [15] lists all functions concerning a channel group with a parameter of type pointer to a `Dio_ChannelGroupType` structure. Furthermore, `Dio_ChannelGroupType` is defined to include three values determining the DIO port of the channel group and its offset and mask applied to it to get the included channels.

However, the document standardizing the ECU configuration parameters [14] (the DIO configuration being among them) defines a DIO channel group through two references to DIO channels named `DioGroupChannelHigh` and `DioGroupChannelLow`. AUTOSAR I/O GUI needs the pieces of information from the two parts linked though and therefore has to implement a special routine to compute the offset and mask in order to identify the channel group on driver side.

Newer versions of the ECU configuration parameter document now recognize this issue and contain the additional remark that the two references “can be used to calculate the implementation parameters `DIO_PORT_OFFSET` and `DIO_PORT_MASK`”. That could have been solved in a better way though by a consistent addressing scheme throughout the whole AUTOSAR framework.

### Mapping Between DIO and PORT Configuration.

Contrary to the module name, the configuration of the PORT module has no notion of a port (see [14]), only single `PortPins` are defined. Since the DIO driver *does* make use of whole ports, a mapping of `PortPins` to DIO ports needs to be made available. Additionally, the position of the `PortPin` (corresponding to a DIO channel) inside the DIO port needs to be known for a complete mapping.

The resulting table hence contains three columns: the `PortPinID`, the DIO port it belongs to, and the channel position inside that DIO port. An appropriate table has to be supplied for each supported target architecture.

## 2.2 DIO Analysis

---

### 2.2.1 The I/O Driver Class

The I/O driver class includes all drivers responsible for both analog and digital I/O; AUTOSAR specifically differentiates between

1. PORT driver: initializes the whole port structure of the microcontroller and assigns its ports and pins to the different specific drivers.
2. DIO (digital I/O) driver: provides read and write access to the general purpose I/O ports, based on single digital channels. It also provides an abstraction of several different channels by offering the concept of channel groups or whole ports.
3. ADC (analog-to-digital conversion) driver: controls the ADC unit of the microcontroller, providing single value result access as well as stream based access.
4. PWM (pulse width modulation) driver: controls the PWM unit of the microcontroller which generates pulses with adjustable pulse width.
5. ICU (input capture unit) driver: provides services for time measurement and edge timestamping.

As all of the listed drivers base on the PORT driver, this driver *has to be* considered in the scope of this work. However, as it does not really provide any further functionality beyond the initialization step, another driver is to be selected. This thesis focuses on DIO because of its basic importance

for many applications. Furthermore, its digital nature allows for a natural mapping in the resulting GUI.

### The PORT Module.

AUTOSAR's PORT module is the management facility for the I/O pins of the microcontroller unit. Besides the initialization, which has to take place prior to any use of the pins by other I/O driver modules, the module provides services concerning the direction of the pins (input or output) [16]. However, this functionality is to be neglected in this thesis (in agreement with Elektrobit Automotive) because of the fundamental complexity the possibility of runtime direction change introduces. Furthermore, that feature is not used very often and thus will not be missed in a prototype work.

Nevertheless, AUTOSAR I/O GUI makes use of the data provided during the PORT initialization step. It needs the information, for example, to determine if a specific DIO channel is an input or output channel. The DIO module is introduced and further investigated in the following sections.

## 2.2.2 DIO Terminology

During the analysis the reader is confronted permanently with three basic terms that are defined in AUTOSAR [17] as follows.

**Channel:** A channel is the digital unit assigned to exactly one pin, carrying one of the two possible signal levels high or low.

**Port:** A port is a group of 8, 16 or 32 channels and can thus be represented by a byte, a word, or a double word, respectively.

**Channel Group:** A channel group is a selection of distinct though adjoining channels of a single port. It is defined using the port number, a bitmask and an offset that is applied to it—or its lowest and highest channel (see [Section 2.1.3](#) for the DIO channel group addressing issue).

## 2.2.3 DIO Example Use Case

**Use Case Name:** DIO Input Output Test

**Summary:** The user modifies the value of a DIO input channel X while observing a DIO output channel Y.

**Rationale:** This basic test case corresponds to a very simplified scenario where the user both emulates a hardware device providing input to the microcontroller and the application, and another device one capturing and processing the output.

**Preconditions:** The functionality of the application running in the simulation consists of writing the value of DIO input channel X to DIO output channel Y whenever a level change of DIO input channel X occurs.

**Triggers:** The user modifies the value of DIO input channel X by manipulating the corresponding GUI element.

**Basic Course of Events:**

1. The user manipulates the GUI element corresponding to DIO input channel X.
2. If the input value did not change, nothing happens. If it *did* change, steps 3 through 6 are performed.
3. The simulation is notified of the change of the value of DIO input channel X.
4. When the polling cycle of the application is triggered, it reads in the new value and performs the necessary computations resulting in the alteration of DIO output channel Y of the simulation.
5. The GUI is notified of the change of the value of DIO output channel Y.
6. The GUI performs the change of state of the GUI element corresponding to DIO output channel Y.

**Postconditions:** The state of the GUI element corresponding to DIO output channel Y complies with the state of the GUI element corresponding to DIO input channel X.

## 2.2.4 DIO Simulation Interfaces

According to the AUTOSAR standard [15], the simulation has to implement seven functions, among them six to both read and write channels, ports, and

channel groups, respectively. The last function serves to get more information about the module (module ID and vendor ID) which is not relevant to AUTOSAR I/O GUI though because it does not care whether the application asks the simulation for the IDs.

### **Interface on Simulation Side.**

So which information is the GUI interested in?

It is certainly concerned about when a change of state in the appearance of any write call happens. Hence each of the write calls (namely `Dio_WriteChannel ()`, `Dio_WritePort ()` and `Dio_WriteChannelGroup ()`) will be added a hook that informs the GUI of the call and its parameters.

The read calls are of no interest to the GUI—the simulation itself has to keep track of the levels of the different input pins.

### **Interface on GUI Side.**

In which cases does the GUI have to send information to the simulation?

Only when a change of state on GUI side happens, communicating to the simulation is necessary. This includes changes of levels on input channels, ports, or channel groups. A message containing the well defined Service ID and the appropriate parameters (see [Section C.1](#)) is then sent to the simulation which stores the new value and begins to communicate it to subsequent read calls on that channel, port, or channel group.

## **2.3 CAN Analysis**

---

### **2.3.1 The Communication Driver Class**

Communication drivers include drivers providing access to specific kinds of communication media. The AUTOSAR communication driver class contains

1. FlexRay driver: is responsible for FlexRay communication controllers and the handling of its buffers, and maps abstract functional operations to sequences of hardware accesses.
2. CAN (controller area network) driver: provides access to CAN controllers and the connected buses, and initiates transmissions and callback functions for notifying receive events.

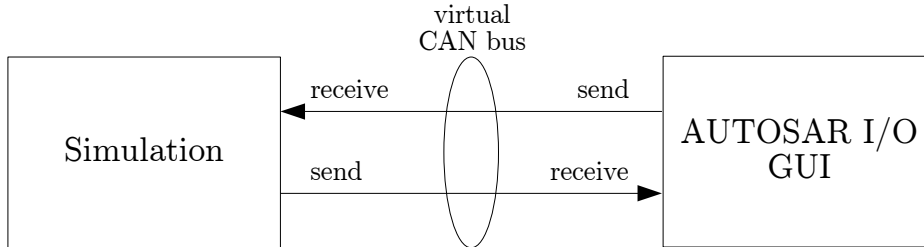


Figure 2.3: CAN simulation scenario 1: Independent communication partners.

3. LIN (local interconnect network) driver: controls the LIN communication unit of the microcontroller, and performs initialization and the actual communication.
4. SPI (serial peripheral interface) handler/driver: provides services for accessing devices connected via SPI buses.

Still being the most important bus system in the automotive field, CAN is the technology of choice for further analysis and possible later prototypical implementation in this thesis.

For an introduction to the CAN bus protocol and its surroundings, consider [18] or [19].

### 2.3.2 CAN Simulation Scenarios

Basically there are two different scenarios possible concerning the behavior of AUTOSAR I/O GUI in a CAN module simulation.

The first one would see the GUI as an independent communication partner to the simulation on a virtual dedicated bus between the GUI and the simulation (see Figure 2.3). In this case a message sent from the GUI is then received by the simulation and vice versa.

A second possibility is the mirroring of the simulation in AUTOSAR I/O GUI (see Figure 2.4). This way each message received by the simulation is also displayed in the GUI, and a message sent through the GUI is actually sent through the simulation, thereby emulating a sending application. This scenario does not specify the implementation of the virtual CAN bus—this is part of the simulation and therefore another thesis. It would be possible to



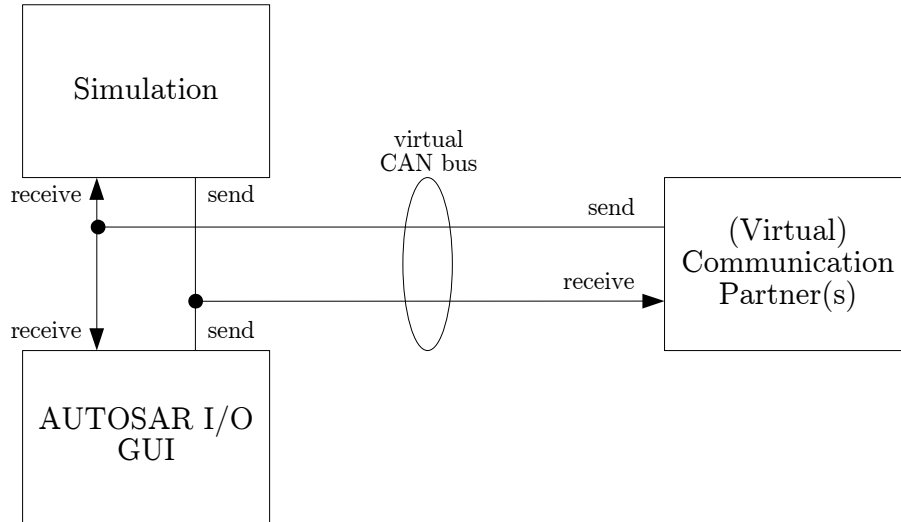


Figure 2.4: CAN simulation scenario 2: Mirroring the simulation.

use either a CAN card attached at the host PC and to redirect the messages to a real CAN bus, or to even simulate the bus itself and to provide the possibility to attach several simulation programs to the simulated bus. Work in this direction already exists, see for example [20].

This thesis uses the first scenario in its considerations because it is more relevant to testing the application or upper AUTOSAR layers. The CAN test stimuli correspond to messages sent to the simulation through the GUI and the test output can be measured by either receiving another message sent by the simulation or by other output means (e.g. using the DIO module).

### 2.3.3 CAN Example Use Case

**Use Case Name:** CAN Send Receive Test

**Summary:** The user manually generates a CAN message which is sent on the CAN bus while waiting for a CAN reply message. In this scenario, the user (through AUTOSAR I/O GUI) acts as a communication partner to the simulated application (see also [Section 2.3.2](#)).

**Rationale:** A test of the application's behavior resulting from (virtual) CAN bus activity is conducted while also registering its output on the bus.

**Preconditions:** The functionality of the application residing on another host connected to the CAN bus consists of echoing the received data bytes on the same CAN bus.

**Triggers:** The user generates a CAN message by entering the appropriate values in the corresponding GUI element.

**Basic Course of Events:**

1. The user manipulates the subelement of the GUI responsible for sending a message to the CAN bus, choosing an arbitrary sequence of bytes as payload data.
2. The simulation is notified of the generated message and simulates the receiving of the message to the application.
3. The application logic receives the message, performs its computations and itself generates a message echoing the payload data.
4. The simulation is notified of the sending of the new CAN message.
5. The simulation itself notifies the GUI of the sending of the new CAN message.
6. The GUI adds an entry in the GUI element corresponding to the listening part of the CAN driver, logging the time of reception and the contents of the message.

**Postconditions:** The GUI element corresponding to the listening part of the CAN driver lists an entry containing the same payload as the CAN message sent by the user earlier.

### 2.3.4 CAN Simulation Interfaces

The specification of the AUTOSAR CAN driver [21] propagates different services

- for initializing the CAN hardware unit or a single controller,

- for setting a controller’s mode (like uninitialized, started, stopped, or sleep mode) and its interrupt enable logic,
- for sending a message on the bus and a call-back function to indicate that a message has been received.

If AUTOSAR I/O GUI acts as an independent virtual CAN controller on a virtual CAN bus (see [Section 2.3.2](#)), the initialization and mode transitions of the simulation’s CAN controllers do not affect it. The GUI implements its own hardware that does not need any initialization or modes in its simplest implementation.

### Interface on Simulation Side.

Given the case the application tries to send a CAN message through the use of the simulation’s CAN driver, the GUI has to receive the message, too—besides the real communication partners depending on the implementation of the simulation’s CAN driver. Thus a hook has to be introduced into `Can_Write ()`.

### Interface on GUI Side.

When the user sends a CAN message to the simulation and its running application, the simulation has to be informed about that event. Its CAN driver can then store the message in an internal buffer and deliver it either the next time that `Can_MainFunction_Read ()` is called (in a cyclic task by the CAN interface module in the case of polling mode) or issue a `CanIf_RxIndication ()` (to inform the CAN interface module in the case of interrupt mode). To find out more about the CAN interface module, please consider its AUTOSAR specification document [22].

The analysis of the interfaces directly affects the decisions for a CAN message design (see [Section C.2](#)).

## 2.4 EEPROM Analysis

---

### 2.4.1 The Memory Driver Class

The AUTOSAR memory driver class is responsible for handling the different kinds of memory devices. It includes

1. EEPROM driver: provides data block oriented services for reading, writing, and erasing an EEPROM. It also enables the possibility to compare an EEPROM data block to one located in RAM.
2. Flash driver: controls read, write and erase accesses to flash memory. Furthermore, it enables setting and resetting the write and erase protection if supported.
3. RAM test: provides a functional test of RAM cells internal to the microcontroller—during initialization and shut down, manually triggered or cyclically.

The interfaces to the EEPROM and flash drivers are rather similar—this thesis arbitrarily chooses EEPROM to concentrate on in its further investigation.

## 2.4.2 EEPROM Example Use Case

**Use Case Name:** EEPROM Read Write Test

**Summary:** The user directly modifies the value of an EEPROM cell X while observing the EEPROM memory image, concentrating on an EEPROM cell Y.

**Rationale:** The user hereby either simulates another application running on the same target and working on the same memory area or he acts as a fault injector to the memory image. That way the robustness of the application can be tested, and the state of the EEPROM memory image can be monitored.

**Preconditions:** The functionality of the application running in the simulation consists of monitoring the value of EEPROM cell X using a polling mechanism, mapping its value to EEPROM cell Y in case cell X changes.

**Triggers:** The user directly modifies the value of EEPROM cell X by entering its new value in the corresponding GUI element.

**Basic Course of Events:**

1. The user manipulates the value of EEPROM cell X directly by processing the corresponding GUI element.

2. If the input value did not change, nothing happens. If it *did* change, steps 3 through 6 are performed.
3. The simulation is notified of the change of the value of EEPROM cell X.
4. When the application logic polls EEPROM cell X for a change the next time, it performs the necessary computations resulting in the alteration of the value stored in EEPROM cell Y.
5. The GUI is notified of the change of the value stored in EEPROM cell Y.
6. The GUI performs the change of state of the GUI element corresponding to EEPROM cell Y.

**Postconditions:** The state of the GUI element corresponding to EEPROM cell Y complies with the state of the GUI element corresponding to EEPROM cell X.

### 2.4.3 EEPROM Simulation Interfaces

AUTOSAR defines an interface for the EEPROM module [23]

- that offers services to initialize and set the mode of the unit,
- a read, write, erase, and compare function on single memory cells that will trigger a new asynchronous job,
- functions to check and modify the status of memory jobs,
- and a method to get information about the module and vendor ID;
- additionally, two callback functions to indicate the successful end or erroneous end of a job may be provided, depending on the configuration of the module.

#### Interface on Simulation Side.

The EEPROM module on GUI side will only be enabled after the simulated EEPROM is initialized. This information in form of an `Eep_Init ()` call has to be forwarded to the GUI.

The GUI has to be notified by the EEPROM module on simulation side whenever a modification of state of the EEPROM is triggered. That includes

calls to `Eep_Write ()` and `Eep_Erase ()`. However, this information can not be directly used due to the asynchronous processing of the commands inside the EEPROM driver which is to be simulated. The simulation rather has to inform the GUI about the memory cells effectively written or erased, respectively—that is why messages with custom Service IDs are used in order not to mix up the semantics (see [Section C.3](#)).

The EEPROM mode and the scheduled EEPROM jobs are internal to the driver and of no further interest to the user and hence to AUTOSAR I/O GUI.

### Interface on GUI Side.

The user shall be able to directly modify the contents of an EEPROM cell (in order to simulate erroneous behavior or an application running concurrently). Thus the simulation has to be kept informed about those cell value changes; the given information has to include EEPROM address, length and data parameters.

## 2.5 DET Analysis

---

### 2.5.1 Introduction

The Development Error Tracer (DET) is a configurable AUTOSAR module that is not a driver but—if enabled—performs checks at runtime and collects the reported errors. The functionality behind that error reporting API is not defined in its AUTOSAR specification [24], but it proposes:

- Setting a debugger breakpoint
- Counting reported errors
- Logging calls and passed parameters in a dedicated RAM buffer
- Sending reported errors via a communication interface to an external logger

In the case of the simulation, the latter two propositions are rather useless because a much more powerful facility is available: it could be *AUTOSAR I/O GUI* that includes a graphical element tracing the calls to the DET presenting the occurred errors to the user and counting them. The GUI could also

offer the symbolic (human readable) values additionally to the numerical ones delivered to the DET API, for example `DIO_DRIVER` and `Dio_WritePort ()` instead of Module ID 120 and Service ID 3.

### 2.5.2 DET Simulation Interfaces

If enabled during the configuration step of the AUTOSAR core, the DET module offers the following interface [24]:

1. `void Det_Init (void);`
2. `void Det_ReportError (uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId);`
3. `void Det_Start (void);`

#### Interface on Simulation Side.

The GUI has to be notified of each of the offered function calls. Only after the consecutive call of `Det_Init ()` and `Det_Start ()`, the representation on GUI side will start working (as does the AUTOSAR module itself according to the specification). After that, each reported error using `Det_ReportError ()` has to be forwarded to the GUI in order to be displayed.

The error codes and their symbolic names are defined in the AUTOSAR specification of the module producing the error, so information in order to transcribe between those two forms has to be obtained there.

#### Interface on GUI Side.

The DET representation on GUI side is merely a passive one; the communication flow is only oneway from the simulation to the GUI. Hence the interface on GUI side is empty, impacting the message design definition (see [Section C.4](#)).

## 2.6 Summary

---

The chapter introduced the AUTOSAR Core Host Data Interface and shortly analyzed the AUTOSAR module architecture, then heavily concentrating on a selection consisting of

- the **Digital I/O driver**,
- the **CAN driver**,
- the **EEPROM driver**,
- and the **Development Error Tracer** module.

Each of these modules was introduced through an example use case and an explanation of its terminology where necessary. The analytical part determined the information exchange between the two entities of AUTOSAR I/O GUI and the actual simulation. The results of these sections lead directly to the message design decisions found in [Section 4.5](#) and the concrete messages listed in [Appendix C](#).



# Analysis of the Program Environment

---

The following chapter is the second part of the analysis, investigating the environment of AUTOSAR I/O GUI and its possibilities while considering the given outer requirements (see [Section 1.5](#)). This way decisions in respect to the close environment of the GUI—used programming language and platform ([Section 3.1](#)), GUI toolkit ([Section 3.2](#)), scripting access ([Section 3.3](#)), and communication mechanism to the simulation ([Section 3.4](#))—are reached after comprehensive justification.

[Section 3.5](#) concisely summarizes the decisions made.

## 3.1 Programming Language and Platform

---

An optional requirement on this work is the platform neutrality of the implementation (see [Requirement 12](#) in [Section 1.5](#)). Especially accessing GUI elements is a very platform specific task since every operating system offers a different interface with differing functionality.

There exist GUI toolkits (see [Section 3.2](#)) that overcome that restriction by offering a uniform API to the programmer while implementing it on different platforms. Normally this is done in a way so that programs making use of that toolkit behave similarly on each (possibly heterogenous) system.

But also the operating system interface differs from system to system and has to be taken account of. For example, the path separator token is “\” in Windows and “/” in Unix-like operating systems, to mention only a very

tiny but annoying difference.

The Java platform and the corresponding Java programming language solve these issues by implementing a Java Virtual Machine (JVM) on several platforms making use of operating system specific functions while offering a common interface to the applications running above. That way a program written in Java can run everywhere a JVM implementation is available. In particular the implementations on Windows and Linux have been proved to be very mature and reliable.

But this is not the only reason why AUTOSAR I/O GUI relies on Java both as its programming language and platform of choice: During the development of this thesis it became clear that the new Elektrobit Automotive *tresos* GUI program (see [Section 1.6.3](#)) would be based on the Eclipse technology. Since then it was an important objective to integrate AUTOSAR I/O GUI with it—which demands for an Eclipse-based solution. The whole Eclipse platform is based on Java and the interfaces it offers, too (see [Section 1.6.2](#)), so this was the final argument in favor of the Java technology. The integration of AUTOSAR I/O GUI is in the form of an Eclipse plugin that extends the functionality of the Rich Client Platform (RCP) that constitutes *tresos* GUI (see also [Section 1.6.2](#)).

## 3.2 GUI Toolkits

---

Since originally there were no restrictions concerning which GUI toolkit to choose, this is another point to analyze and finally make a decision for. The only restriction kept in mind was the operating system independence (see [Requirement 12](#) in [Section 1.5](#)) that is not given with every toolkit.

The following sections introduce selected GUI toolkits. Since the decision was reached to use Java as a platform for AUTOSAR I/O GUI (see [Section 3.1](#)), only Java based toolkits are discussed. Because of the vast amount of GUI toolkits available, only the ones leading to the finally used Graphical Editing Framework (GEF) are examined while justifying that decision.

### 3.2.1 AWT and Swing

Sun's Java Development Kit always included a GUI toolkit named AWT (Abstract Window Toolkit). The components used when programming a user interface with AWT are platform independent, AWT uses the functionality of the target platform to draw the components [25]. That is why the same component appears differently on different platforms. The big drawback is



Figure 3.1: A typical Swing dialog [26].

that only the intersection of the GUI functionalities of all supported platforms is offered in AWT.

The Swing library was hence designed to avoid AWT’s handicaps. Its unique selling point is its lightweight architecture—all of the Swing components are implemented in pure Java without any native code [25], thus effectively emulating all the widgets which inevitably results in a performance drawback. The look and feel of Swing programs is therefore the same on all platforms (see [Figure 3.1](#)), it can be defined choosing from a collection of different sets (buzzword “pluggable look and feel”). Swing also gets a lot more powerful than AWT by defining new complex components like trees or tables.

Swing’s propagated advantage over AWT is also its biggest disadvantage: Swing programs do look the same on all platforms, but they do not integrate very well into any of them. This is a dilemma situation, but the ergonomist’s position in the debate is quite clear: It is far more important to integrate an interface into the environment the operator is used to (because he chose it as his personal working environment) than to make the interface look exactly the same in all possible environments. Only a few users are forced to concurrently use the same GUI on different platforms.

Both AWT and Swing had a significant influence on later developed GUI toolkits like SWT (see [Section 3.2.2](#)) or GEF’s Draw2D (see [Section 3.2.3](#)). The propagated use of different kinds of layout managers can still be found there as well as AWT’s event handling architecture using event sources and event listeners.

### 3.2.2 SWT and JFace

Eclipse’s Standard Widget Toolkit (SWT) combines two advantages where other GUI libraries possess at most one of them. It provides both a common operating system independent API while it is still implemented in a tightly integrated way with the underlying native window system [27]. That means that SWT uses native widgets wherever possible and only falls back



Figure 3.2: A typical SWT dialog on the Windows platform [28].

to emulating them in the case the window system lacks that kind of widget.

The advantages of that approach are obvious: The programming model is kept consistent independent from the target platform but the user still will not notice any difference to an implementation using the native widgets directly [6] (see Figure 3.2). Therefore he is kept in his consistent and known environment and does not have to adapt to a new look and feel. This is a very important issue in software usability because it not only affects the visual part of the program but also its behavior in the surrounding environment; for example, the support for native drag and drop is essential for a tight integration of the target program. A successful integration has a positive impact on all of the five software usability criteria identified by usability expert Jakob Nielsen: learnability, efficiency, memorability, low error rate, and user satisfaction [29]. That is why some people speak of SWT as highly as being the breakthrough for Java on the desktop [6].

Eclipse also provides JFace, which is a UI toolkit layered on top of SWT. It provides complex classes for the handling of many common UI programming tasks [27] like, for example, message dialogs to interact with the user.

### 3.2.3 GEF and Draw2D

The Eclipse platform further simplifies the development of AUTOSAR I/O GUI by providing GEF, the Graphical Editing Framework. GEF makes it possible to develop feature rich graphical editors [30] representing an arbitrary underlying model. It therefore provides common functions like drag and drop, copy and paste, or actions invoked from menus or toolbars. The visualization is programmed using the specially developed Draw2D framework which itself is based on SWT (see Section 3.2.2).

AUTOSAR I/O GUI is not supposed to be an editor in the known way.

Its editing capabilities are limited to

- introducing offered GUI elements into the working space,
- resizing and arranging them in the desired way,
- tweaking their behavioral properties,
- and editing the elements capable of receiving input.

Nevertheless, this is enough editing to be able to benefit from making use of GEF because the framework assists those tasks enormously. Moreover, there is no alternative to GEF that integrates into Eclipse as smoothly [31].

Furthermore, by using GEF’s abridged version of the Model–View–Controller architecture (see [Section 4.2.1](#) for a description of the traditional design pattern and a discussion of the differences to the version that GEF implements), AUTOSAR I/O GUI obtains a big amount of flexibility. The independence of the model and the view in the architecture makes it easy to introduce new views to the existing models, thereby complying with [Requirement 5](#) (see [Section 1.5](#)).

### 3.2.4 Conclusion

AUTOSAR I/O GUI is based on the toolkits offered by Eclipse and its sub-project GEF. This decision is based solely on the discussion above and it is rather coincidental that it integrates perfectly with the new *tresos* GUI program by Elektrobit Automotive which is also based on Eclipse and therefore SWT and JFace.

Only after the decision to use the Eclipse framework, the corresponding part of the subtitle of this thesis (“Eclipse-based”) was added.

## 3.3 Tests and Scripting Access

---

A fundamental feature that AUTOSAR I/O GUI is supposed to offer is the ability to run automated tests through it (see [Requirement 3](#) in [Section 1.5](#)). This way the testing can not only be performed interactively using the GUI elements but also in an automated way in the background, possibly in a batch manner. Therefore another goal is the development of a customly built or the deployment of an already existing scripting language.

This section outlines the deliberations on the way to a decision concerning the scripting environment used in AUTOSAR I/O GUI, beginning with an exemplary use case.

```
print channel level of channel ENGINE_RUNNING;
set channel level of channel START_ENGINE to HIGH;
wait for 1 second;

set new_level to the channel level of channel ENGINE_RUNNING;
print new_level;

if (new_level is not HIGH) then
    print "Test failed.";
else
    print "Test succeeded.";
end if
```

Figure 3.3: Pseudo code test script for use case “Scripting Access Test”.

### 3.3.1 Example Use Case: Scripting Access

**Use Case Name:** Scripting Access Test

**Summary:** The user performs an automated scripted test without needing to modify any GUI controls.

**Rationale:** This scenario corresponds to a user wanting to test an application running in the simulation using an exactly defined stimuli pattern without interacting with the GUI during the test.

**Preconditions:** The simulation and the GUI are up and running. The application running in the simulation responds to a change on a hypothetical DIO input channel `START_ENGINE` to `HIGH` by setting the level of DIO output channel `ENGINE_RUNNING` to `HIGH`.

**Triggers:** The user instructs the GUI to load and execute a previously written script, namely the one depicted in [Figure 3.3](#) (denoted in a pseudo code format).

**Basic Course of Events:** AUTOSAR I/O GUI dynamically performs the actions given in the script file without further user interaction.

**Postconditions:** The scripted commands executed successfully, eventually indicating the correct or incorrect functionality of the application getting tested.

### 3.3.2 Domain-Specific Languages

The scripts written by the user in order to automate part of AUTOSAR I/O GUI's behavior are composed in what is called a domain-specific language (DSL). A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [32]. It is supposed to reflect high-level concepts in a direct manner so that the “sentences” become synonymous to the intentions of its author. Normally this can be seen by a significant smaller amount of coded lines using a DSL for a given problem than using a general purpose language. Note that the syntax of the pseudo code script shown in the use case (Figure 3.3) might depict part of a DSL for AUTOSAR I/O GUI.

Developing a completely new DSL from scratch bears the advantage that the developer has no restrictions concerning the syntax—the DSL gains an arbitrary amount of expressiveness and the semantic distance between the problem and the program can be reduced to a minimum. It can also be adapted to the needs of the testing individual, thereby resulting in high performance due to the elimination of overhead by features that are not used.

On the other hand, a customly defined language will always lack one feature or another that is needed in contexts the original programmer did not think of. An efficient implementation is also difficult to achieve if you are not an expert on compiler techniques. The development generally is a costly task although there are instructions and toolkits facilitating the process available<sup>1</sup>. That is why the decision to implement an own DSL with a personally developed compiler, interpreter, or source-to-source transformer has to be considered carefully.

The alternative to this approach is to use what is called a language extension pattern [34] by relying on an existing language and extending it to offer the domain-specific functionality. The resulting DSL syntax is surely constrained by the syntax of the host language, but this is not necessarily a disadvantage: If the host language is a popular one, developers using the DSL are already familiar with it and do not need much education on the spe-

---

<sup>1</sup>One approach is the exploitation of C++ templates by metaprogramming. Other ones use specially developed toolkits; for an overview see [33].

cialties of the DSL. Moreover, the popular general purpose languages come with an extensive integrated development environment (IDE) support for the programmer which are not available for a completely custom DSL.

These arguments lead to the decision that an existing programming language will be used as a basis for the scripting access in AUTOSAR I/O GUI.

### 3.3.3 Embedding Scripting Languages

The decision to rely on a host language to embed in leads to the search for a suitable one. There are several factors influencing the feasibility of an investigated language:

- The language syntax should be close to a DSL in order to allow intuitive and rapid development.
- It should be possible to integrate the implementation of the language into the Java platform in order to be able to seamlessly communicate with AUTOSAR I/O GUI.
- The implementation should not be a prototype research project but under active development in order to guarantee timely bug fixes if discovered.
- The licence of the implementation should allow redistribution of the program using it (AUTOSAR I/O GUI in this case).

[35] contains a list of programming languages for the Java Virtual Machine. Since it comprises more than 200 different entries, further filtering includes only the scripting language implementations (since only those bear a natural syntax close to a DSL) and excludes pure research projects where development is stalled. The resulting intersection set still contains more than a dozen scripting language implementations like Jacl, Jython, Rhino (implementing JavaScript), JRuby, or JudoScript (with JavaScript-like syntax).

David Kearns wrote two articles for JavaWorld about scripting languages embedding in Java ([36], [37]), comparing them in terms of performance and integration amongst others. Since performance might be an issue of concern in some test cases (see also [Requirement 13](#) in [Section 1.5](#)) and Jython was consistently the fastest (in the first comparison) or among the fastest (in the second one), Jython is further investigated for suitability for AUTOSAR I/O GUI. Note that native implementations (e.g. in Java) have a significantly lower execution time but more time is needed for development of the script



because its syntax is not very close to a DSL—that is not acceptable in the context of AUTOSAR I/O GUI.

### 3.3.4 Jython

Jython<sup>2</sup> is an implementation of the Python scripting language for the Java Virtual Machine [38]. Its integration into Java is rather seamless, enabling the programmer to exchange information between the two platforms. For example, the user can utilize Java classes in his Python scripts just by importing them like he would classes written in Python themselves. He can also access concrete Java instances through the same interface. Moreover, the Python language uses exceptions to indicate erroneous conditions like Java does and therefore integrates very well in a Java environment.

Its syntax is very close to natural language and it is typed dynamically, thereby unburdening the programmer. The example script introduced in Section 3.3.1 rewritten in Python would look like the one depicted in Figure 3.4. Note that it is extremely similar to the DSL pseudo code depicted in Figure 3.3 and therefore very easy to understand and almost as concise as the DSL variant.

The DSL character of the script is achieved by offering a limited interface to AUTOSAR I/O GUI (in this case the `gui` object). This interface is well-defined both in terms of access rights (see Section 4.1.1) as well as in naming of the methods to be close to natural language in order to reflect domain specificity.

Jython’s license allows redistribution in commercial products [39], it disclaims any warranties, however—as is the case with almost every open source license. Support for Jython can be found in the form of a dedicated book (“Jython Essentials”, [40]) and a special IDE plug-in for Eclipse (JyDT, [41]). Furthermore, there is at least one other project at Elektrobit Automotive that relies on Jython, so synergy effects may be achieved. A minor drawback is that there are some extension modules implemented in CPython (the original Python implementation in C) that are still missing in Jython to be implemented [42]. The functionality relevant to AUTOSAR I/O GUI is completely available, however.

The Python language is popular among programmers and therefore the scripting interface to AUTOSAR I/O GUI will impose no difficulties for many of them. To find out more about the features of the Python language and how they are relevant to AUTOSAR I/O GUI, please consider Section 1.6.4.

---

<sup>2</sup>Formerly JPython.

```
import time

print gui.getDIOOutputChannelLevel ("ENGINE_RUNNING")
gui.setInputChannelLevel ("START_ENGINE", HIGH)
time.sleep (1)

new_level = gui.getDIOInputChannelLevel ("ENGINE_RUNNING")
print new_level

if (new_level != HIGH):
    print "Test failed."
else:
    print "Test succeeded."
```

Figure 3.4: Python test script for use case “Scripting Access Test”.

## 3.4 Communication AUTOSAR I/O GUI–Simulation

---

The analysis of the AUTOSAR modules ([Chapter 2](#)) among other things determined the interfaces and therefore the data that needs to be communicated between AUTOSAR I/O GUI and the simulation.

However, the requirements leave it open how the two modules exchange the data needed—although they demand for platform independence ([Requirement 12](#) in [Section 1.5](#)), the decoupling of the two parties ([Requirement 2](#)), and the consideration of performance issues ([Requirement 13](#)) and thereby provide criteria to evaluate the different possibilities.

This section therefore discusses the choice of a basic communication infrastructure to use for the AUTOSAR Core Host Data Interface. A limiting factor in the discussion is the fact that the simulation part will be written in C as is the rest of the AUTOSAR framework while the GUI part uses Java for the reasons stated above (see [Section 3.1](#)).

### 3.4.1 Java Native Interface

A first approach to the problem is the use of a direct (function) call interface between the simulation and the GUI. The only obstacle is the interface between the native platform (in this case C based) and the Java platform which can be addressed by using JNI.

Sun’s Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the Java Virtual Machine into native applications [43]. It allows Java code to call or be called by native applications [44] and would therefore be suited for the examined scenario and provide for a pretty good performance compared to the following alternatives (although approximately three times slower than a real method invocation [45]).

The big drawback in using a direct call interface is that the two modules will inherently be tightly coupled. They would share the same address space (running the risk of jeopardizing each other), and the simulation and the GUI would not be clearly separated, thereby effectively violating **Requirement 2** (see **Section 1.5**). Moreover, part of the platform independence would get lost since the native part of AUTOSAR I/O GUI would have to be adapted and compiled for each platform to support. These disadvantages outweigh the possible performance advantage.

### 3.4.2 Shared Memory or Memory-Mapped File

Another possibility for exchanging data between the two inhomogeneous programs is the use of a shared memory area or a common file mapped into the address spaces of both modules. Thereby changes to parts of the shared memory committed by one of the programs will immediately be visible to the other one.

This approach is very suitable where larger amounts of data need to be exchanged, in our scenario particularly the simulation of the EEPROM module. A problem is the implementation of asynchronous events exchanged between the two parts. These are not supported by the concept of shared memory itself but need to be performed using other mechanisms like signals which are mostly operating system specific and therefore are not considered further. Moreover, a synchronisation mechanism would have to be used—this is a platform-specific concern as well.

### 3.4.3 CORBA

OMG’s Common Object Request Broker Architecture (CORBA) addresses exactly the mentioned inhomogeneity. It allows a CORBA-based program on almost any operating system, programming language, and network, to interoperate with another CORBA-based program [46]. CORBA works by defining a common interface definition language (IDL) and a common protocol between the communication ORBs (General Inter-ORB Protocol GIOP).

CORBA can easily solve the problems faced but will result in a significant overhead due to the power it inheres by offering many additional services and features (e.g. network transparency) that are not made use of in our context.

### 3.4.4 Sockets

Sockets are platform independent as well as capable of remote communication. They offer a basic but clean and generic interface that is open for custom protocols.

The possibility of remote communication leads to a whole new dimension of possibilities of integrating different platforms: Running the simulation on a Windows machine while watching and manipulating the corresponding GUI running in a Linux environment is now possible. Especially during the phase where the simulator is not yet ported to other platforms than Windows this is a huge advantage and even extends [Requirement 12](#) (see [Section 1.5](#)) in a way through this introduction of new flexibility.

Moreover, it would be possible to connect several GUIs to just one simulation by using multicast IP addresses (when UDP is used as a transport protocol). In some scenarios (like a distributed training environment) this might be of good use, too.

Since software on microcontrollers that are more powerful will implement a TCP/IP stack, AUTOSAR I/O GUI could even be connected to a real target instead of a simulation, provided that an alternate microcontroller abstraction layer is deployed on the target. This way applications running on the target platform can be tested the same way as applications running on the simulation host.

### 3.4.5 Conclusion

Sockets offer platform independence and the remote benefit while not introducing a big overhead like CORBA does. That is why the interface between the simulation and AUTOSAR I/O GUI relies on sockets.

The concrete protocol consisting of the messages that are exchanged between the two entities is defined in [Section 4.5](#) and [Appendix C](#).

## 3.5 Summary

---

This chapter investigated the environment of the AUTOSAR I/O GUI program to determine the general framework it will rely on.

It identified **Java** as the technology of choice for AUTOSAR I/O GUI and **Eclipse** as the platform it will be based on. The decision to use an additional Eclipse plug-in, **GEF**, was reached in addition to the one of making use of **SWT** as a GUI toolkit.

Further considerations were in respect to the way of accessing AUTOSAR I/O GUI in an automated way for tests and the concrete communication mechanism to use to commune with the simulation. **Jython** will be the module of choice to provide the GUI with a scripting access and the communication interface with the simulation will be based on **sockets**.



# Design and Implementation

---

This chapter develops the concrete design of AUTOSAR I/O GUI, based on the information and environmental decisions gathered in the preceding analytical Chapters 2 and 3.

The first [Section 4.1](#) concentrates on the big picture by giving an overview of the program architecture and detailing its main elements of design as well as some pieces of subtleness. The following [Section 4.2](#) treats the deployment of well-defined design patterns in AUTOSAR I/O GUI, focusing on the Model–View–Controller pattern that is essential for the micro architecture. A special part ([Section 4.3](#)) is dedicated to the optional feature of recording test scripts and the way it is implemented; [Section 4.4](#) presents how AUTOSAR I/O GUI is kept generic in order to be easily extensible. [Section 4.5](#) defines the message format to be used for communication to the actual simulation and therefore builds the foundation for the assessment of the messages for the implemented AUTOSAR modules (see [Appendix C](#)). The chapter is concluded with [Section 4.6](#) which shortly introduces basic human factors principles and how they are considered in the implementation of AUTOSAR I/O GUI.

A short summary of the discussed topics is given in [Section 4.7](#).

## 4.1 Program Architecture

---

The central part of AUTOSAR I/O GUI is its database storing the states of all implemented drivers—the program’s macro architecture evolves around it according to [Figure 4.1](#). There are three entities accessing the data on GUI

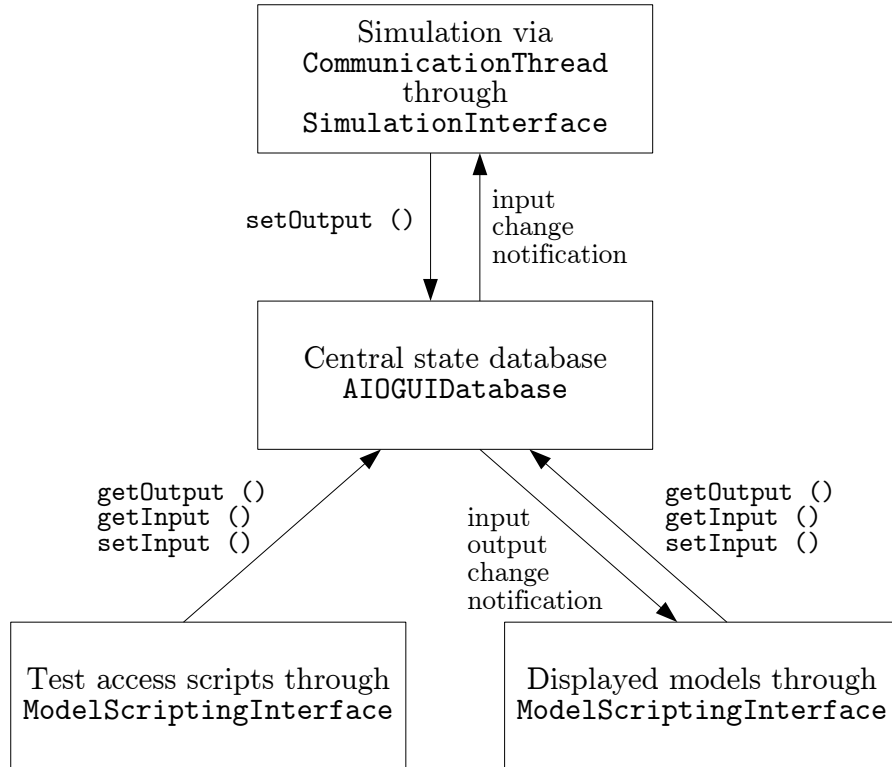


Figure 4.1: AUTOSAR I/O GUI's macro architecture.

side: the simulation propagating its output changes through the communication thread, the currently displayed models in the layout reading their corresponding state and writing it if applicable (if they have input capabilities), and scripted tests running concurrently.

[Section 4.1.1](#) describes the database and its interfaces in a more thorough manner and [Section 4.1.2](#) picks up the communication thread. The displayed models themselves are part of a micro architecture which is explained in [Section 4.1.3](#).

### 4.1.1 The Central State Database

All of the states of AIOGUI's different drivers are saved centrally using the class `AIOGUIDatabase`.



### Processing State Changes in General.

Every time a substate of the database changes—no matter if triggered by the simulation, a script, or the user, and no matter if input or output state—the models currently displayed and affected by the change have to be updated as well. Special care has to be taken if there are more than only the obvious models that must be considered: for example, the alteration of a DIO channel not only affects the models displaying that channel but also the models displaying DIO channel groups and DIO ports *containing* that channel.

### Using Access Interfaces.

Since the database is accessed from three different entities with differing access rights (see [Figure 4.1](#)), only well defined interfaces to the database are given to them, namely:

1. `ModelScriptingInterface` contains the functions the models displayed in the editor should be able to call. This includes functions to manipulate input states (if the user alters the corresponding controls) and to get input and output states from the database. The models are *not* allowed to set output states, so these functions are excluded from this interface. In contrast to the scripts (see below), the models get notified of a change of state in the database.
2. Since the running scripts should simulate user interaction with the controls, they share the same `ModelScriptingInterface` to the database. The names of the interface functions are to be chosen carefully because they constitute the DSL character of the scripting access (see [Section 3.3.4](#)). All of the functions are declared to throw a `ConfigurationException` if the specified AUTOSAR entity does not exist or is configured wrong (which may happen when the script was developed for a different—incompatible—configuration, see also [Section 4.1.5](#)). This interface is a polling interface; that is, it is not notified about state changes in the database (see connecting arrows in [Figure 4.1](#))—that is the nature of a script.
3. `SimulationInterface` is the interface to the communication thread which represents the simulation. It includes only functions that correspond to the ones defined in AUTOSAR, setting output states. Input state changes can only happen on GUI side.

### Propagating Input State Changes.

When either the user or a script (simulating a user) manipulates an input control, that state change leads to two reactions. First the new input state is stored in the central state database in order to satisfy following requests to get the input state (e.g. resulting from a scripted command). Secondly, the state change is propagated to the simulation so it can notify the application as specified by AUTOSAR. This is done through the communication thread, further described in [Section 4.1.2](#).

### Incorporating Specialized Databases.

Module extensions to AUTOSAR I/O GUI are encouraged to define specialized databases for their data if the amount of complexity asks for it. This subdatabase is then referenced by the main `AIOGUIDatabase`.

The DIO module, for instance, defines a specialized `DIODatabase` that keeps the main database from fiddling about single bits (the DIO channels) and their mapping to DIO ports and DIO channel groups.

## 4.1.2 The Communication Thread

The thread responsible for the communication with the simulation is implemented in `CommunicationThread`. It takes care of marshalling and demarshalling the messages sent to and received from the simulation, respectively, by implementing the communication protocol defined by the AUTOSAR Core Host Data Interface (see [Section 2.1.1](#), [Section 4.5](#), and [Appendix C](#)). Should this interface ever be changed, only the `CommunicationThread` class will need adaptation.

## 4.1.3 Micro Architecture of Model, EditPart, and Figure

*One* displayed control in AUTOSAR I/O GUI's layout is represented by a triad of three objects—a `Model` object, an `EditPart` object, and a `Figure` object. This design is a version of the popular Model–View–Controller design pattern, abridged by GEF—please consider [Section 4.2.1](#) for a detailed discussion of the differences.

There are two kinds of controls that need to be distinguished: one restricted to only displaying output, and the other one that can also receive input from the user. The latter contains all the functionality of the first one but in addition possesses more methods and links between the parts of the micro architecture.

Note that functionality common to all controls is implemented in abstract base classes (see [Section 4.4.1](#)).

### The Triad of an Output Control.

#### The `OutputModel`

- stores some kind of information to identify itself to the database (typically an unambiguous AUTOSAR configuration name),
- implements an interface to the control (e.g. `setLevel ()`) firing a property change with the new value to inform the `EditPart` (managing of the listener(s) is implemented in the super class, see [Section 4.4.1](#)),
- takes care of the property descriptors stored in the field `_descriptors` and the implementation of the `getPropertyValue ()` and `setPropertyValue ()` methods to customize the information displayed in the properties view,
- and supplies the implementation of the abstract base methods to provide for restoring that control (see [Section 4.4.2](#)).

#### The corresponding `OutputEditPart`

- instantiates the `Figure` in `createFigure ()` (which is called by the framework upon a user's drag action),
- and implements `refreshVisuals ()` which first calls the super implementation to update the bounds (location and size in the layout) and then updates the `Figure` (through its defined interface) according to the state of the `Model`. Note that `refreshVisuals ()` is called upon a property change resulting from a change of the model or the bounds (through direct editing of the layout or the corresponding properties in the properties view). The handling of these property changes (and thus the implementing of the `PropertyChangeListener` interface) is done in the abstract super class (see [Section 4.4.1](#)).

#### Finally, the attached `OutputFigure`

- provides the concrete visual implementation in its constructor (aided by the constructor of the abstract base class, see [Section 4.4.1](#)),
- provides an interface to the control (e.g. `setLevel ()`, notice the similarity to the `OutputModel` interface),
- and manages its visual state according to the accesses to that interface.

### The Triad of an Input Control.

The objects pertaining to an input control provide further functionality to reflect the data flow from the `InputFigure` (that is manipulated by the user) to the `Model` and therefore to the database.

The `InputModel` additionally has to extend its implementation of the control interface (e.g. `setLevel ()`) in a way so that it is possible to alter the state in the central database due to an editing request by the `InputEditPart`. Such an alteration leads to a message sent to the simulation through the AUTOSAR Core Host Data Interface; the simulation then has to store the updated state to satisfy further requests from the upper layers, including the application.

New functionality to the `InputEditPart` is

- to register itself as a change listener to the `InputFigure`,
- and to override the `propertyChange ()` method. In the case the property event comes from the `InputFigure`, it edits the model accordingly. Otherwise the method call is delegated to its super implementation (see [Section 4.4.1](#)).

Furthermore, an `InputFigure`

- listens to user interactions and triggers a property change event for the listening `EditPart`,
- but nevertheless supplies a method to change the visual state of the input widget (e.g. `setLevel ()`) and therefore having output functionality also. This is necessary because the state needs to be modified when either a control belonging to the same AUTOSAR entity is modified (e.g. the same DIO input channel) or when user interaction is simulated through a running script.

#### 4.1.4 Threads and Thread Synchronization

AUTOSAR I/O GUI uses more than one thread of execution running in parallel in order to implement quasi-simultaneous activity. The following threads may be running concurrently at one point in time:

- One user interface thread that is dedicated to communicating with the user in order to keep the response time short. If other threads try to make changes to the user interface directly, they get an invalid thread access exception. That is why they have to use `Display`'s

`asyncExec ()` or `syncExec ()` and provide runnables to do the job. The cases that follow a certain pattern can be covered using `UiThread-ExecutionAspect` which gives advice to the calls specified by an appropriate pointcut and runs them in the user interface thread (see [Section 4.3.1](#) for an introduction to aspect-oriented programming and the main use of it in AUTOSAR I/O GUI).

- One communication thread that is created upon user request to establish a connection (see [Section 4.1.2](#)). The thread runs in parallel, dispatching the incoming messages into function calls on the central database altering the state and also displayed models affected by the change.
- One or more scripting threads that are executing the behavior defined in one or more script files. These threads may also operate on the state database and call the state manipulation functions.

### Thread Synchronization.

Since a thread may be interrupted in the middle of its control flow and another one may be scheduled to execute instead, possible hazards have to be discovered and appropriate synchronization strategies have to be thought of.

In AUTOSAR I/O GUI there are two scenarios that can lead to an inconsistent program state or behavior due to thread synchronisation problems:

1. Two or more threads (e.g. the user interface thread and a running scripting thread) try to send a message to the simulation through the socket interface simultaneously, thereby effectively corrupting the byte stream received by the simulation.
2. Two or more threads (e.g. the user interface thread and a running scripting thread) try to alter a substate of the central database simultaneously, leading to inconsistencies between the different models displaying the state and also the internal state.

The first problem can be addressed by sourcing out the actual sending of data to the simulation into a dedicated method and keep that method `synchronized` so that only one thread can access it at a time. The implementation therefore includes a method `sendMessage ()` in the `CommunicationThread` class that takes a byte array as its argument and sends it through the socket interface in a synchronized kind of way.

The second hazard applies only to those methods of the database that change *input* states (methods that alter *output* states can only be called by the simulation and therefore only by a single thread—the communication thread). The single entry point for scripting threads are the methods of the state database that are revealed to them through the `ModelScriptingInterface`. The actual state change resulting from user interaction with the user interface thread also occurs in those methods after traversing the triad of `Model`, `EditPart`, and `Figure`. Therefore declaring those methods `synchronized` avoids concurrent state alteration.

### 4.1.5 Effects of the Currently Active Configuration

Since AUTOSAR I/O GUI is embedded in the *tresos* GUI program, it can be supplied the AUTOSAR configuration that was defined for the ECU that the software is being developed for. The adaption of AUTOSAR I/O GUI to the currently active configuration is an essential part of it ((see also [Requirement 8](#) in [Section 1.5](#))).

That configuration, encapsulated in an object of type `AIOGUIConfiguration`, has an impact on different parts of AUTOSAR I/O GUI:

- The most obvious effect for the user of AUTOSAR I/O GUI is the adaption of the palette of offered controls. Only the controls available according to the configuration are offered.
- Consistency checks are performed in the central database to verify calls to it. That includes general checks if the requested control is available in the currently active configuration and more specific ones like, for example, to check that the DIO channel that is set by the simulation is really an *output* channel.

This way three types of consistency are being checked:

- The consistency between the currently active configuration in AUTOSAR I/O GUI and the one active in the simulation.
- The consistency between the currently active configuration in AUTOSAR I/O and a loaded diagram (that was possibly created and stored using a different—incompatible—configuration).
- The consistency between the currently active configuration in AUTOSAR I/O and a running script (that was possibly created using a different—incompatible—configuration).

- The configuration instance also bears information about how to translate between the symbolic names used by the operator and the numerical IDs used internally in AUTOSAR. It also knows the mapping between the name of a control and its type (e.g. DIO output channel).
- The database needs the information from the configuration to hold its state correctly. E.g. the DIO subpart of the database needs to know which DIO channels belong to which DIO port or channel groups in order to handle state manipulation functions in a correct manner.

## 4.2 Reusing Design Patterns

---

AUTOSAR I/O GUI makes use of several well described object-oriented design patterns at different occasions. Some of them are encouraged to use by the ambient frameworks (Eclipse and GEF), other ones are used freely to achieve a good and reusable design which is the main purpose of using documented design patterns [47].

This section shortly introduces selected patterns and explains why and where they are deployed in AUTOSAR I/O GUI.

### 4.2.1 Model–View–Controller

In [Section 3.2.3](#), GEF was elected to be the framework of choice for AUTOSAR I/O GUI. GEF forces its user to think of the deployed objects in a way similar to the Model–View–Controller (MVC) design pattern. The resulting micro architecture (see [Section 4.1.3](#)) is not the same as it would look like according to the description of the original MVC pattern, however.

This section uncovers those differences between the two notions and explains the design decisions made in AUTOSAR I/O GUI in that respect.

#### **The Original Model–View–Controller Pattern.**

The original MVC pattern was developed to build user interfaces for Smalltalk-80 [47] and defines a triad of classes. The *model* object is the actual object bearing a state that is to be displayed and manipulated. The *view* is the visual representation of the model object’s state whereas the *controller* object bridges interaction from the user.

If the model’s state changes (either through its controller or an external trigger like the simulation in the case of AUTOSAR I/O GUI), it notifies its registered views of the change (through the use of the Observer design

pattern, see [Section 4.2.5](#)). The views can then update their displayed state according to the model state retrieved through a reference to the model stored in an instance variable [48].

The user does not interact with the model directly but through a controller object. The controller object then decides which changes to apply to the model. Depending on the actual controller implementation handed to the user, these changes can turn out differently (this part of the MVC pattern is also called Strategy pattern).

The flexibility of deploying the MVC pattern results from the ability to hand out different controller objects and to display different views (even at the same time) for a single model object.

### The Model–View–Controller Pattern as Used in GEF.

GEF abridges the original MVC notion by handling the controller (called `EditPart` in its terminology) as a bridge between the model and the view (`Figure` in GEF jargon) [49]. The user interacts directly with the `Figure` which then notifies its `EditPart` to change the model. The `Figure` herein also contains some of the functionality of a classic controller—it is the target of the user interaction by providing editable controls like checkboxes or buttons. Changes on the model side, however, are also propagated to its `EditPart` first which then updates the `Figure`. Each `EditPart` holds references to its attached model and `Figure` and is registered to be notified in case either one changes.

On behalf of the framework, the `EditParts` are instantiated on demand (after dragging a model on the display by selecting it in the attached palette) and pooled for performances purposes [31].

### Conclusion.

Because of the restrictions imposed by the design of the GEF framework it is not possible to develop a clean design for AUTOSAR I/O GUI using the original MVC notion.

An ideal design would use a single model object that accesses the central state database and populate the display with the different views (only displaying a subset of the model and therefore the database) according to the user's drag behavior. The appropriate controller object would also be paired to each of the views.

However, GEF allows only for *models* to be instantiated as the result of a drag action. The framework then instantiates the controller which eventually instantiates the view itself.



That is why two views for the same entity (e.g. a specific DIO channel) also possess two different controller and model objects. The two model objects then access the same central state database, however.

### 4.2.2 Command

Eclipse encourages the use of `Commands` in order to implement changes to a file (in this case through the AUTOSAR I/O GUI layout editor). Changes to be committed are encapsulated in a `Command` object containing the code necessary to perform the changes as well as to revoke them. The object is then passed to a `CommandStack` which executes the command while registering it internally. This way Eclipse manages to support undo and redo operations (which is one of the main advantages of the use of the Command design pattern [47]) and the ability to detect whether an edited file is “dirty” (contains changes) in order to determine the state of the “Save” action (disabled or enabled).

There are different kinds of `Commands` in AUTOSAR I/O GUI, all dedicated to changing the layout—`CreateCommand` and `DeleteCommand` to add or delete a new control, respectively, and `SetConstraintCommand` to change the location or the size of a control.

Whenever a visual element is restored (by undoing a delete operation or redoing a create operation) it is given the state matching the one of the central state database (even if it has changed when the element was not visible) and *not* the one it had when it disappeared from the visible area.

### 4.2.3 Singleton

According to Gamma et al. the Singleton design pattern’s intent is to ensure that a class only has one instance, and to provide a global point of access to it [47].

The global database containing the states of all simulated AUTOSAR drivers is per se a single object—of type `AIOGUIDatabase`. It has to be accessible from the main editor instance which initializes its structure as well as from the displayed models, the scripting interface and the communication thread (representing the simulation).

The `AIOGUIDatabase` class implements the Singleton design pattern by providing a static `getDefault()` method which instantiates a new `AIOGUIDatabase` singleton object or returns a reference to an already existing one.

`Recorder`, implementing AUTOSAR I/O GUI’s script recording feature (see Section 4.3), is also a Singleton object and therefore implements the same design pattern.

### 4.2.4 Memento

The use of the Memento design pattern is also supported by Eclipse by providing classes like `XMLMemento`. The Memento pattern is commonly deployed to externalize an object's internal state (storing it) and restoring to this state at a later point in time [47].

When saving the layout of an AUTOSAR I/O GUI session (through the actions “Save” or “Save As...”), it creates a Memento object which is written to a file using XML syntax (see also [Requirement 9](#) in [Section 1.5](#)).

More detailed information about AUTOSAR I/O GUI's save-and-restore mechanism making use of the Memento pattern can be found in [Section 4.4.2](#).

### 4.2.5 Observer

The Observer pattern is supposed to be used in order to notify and update dependents of an object whenever the state of that object changes [47]. In the context of the Model–View–Controller architecture deployed in AUTOSAR I/O GUI (see [Section 4.2.1](#)) this is particularly useful to notify the bridging `EditPart` whenever the associated `Figure` or `Model` changes.

AUTOSAR I/O GUI's `EditParts` follow that pattern by registering themselves as `PropertyChangeListeners` at the associated `Figure` and `Model`. When the user manipulates the `Figure`, the `EditPart` is notified and takes the appropriate actions (probably editing the `Model`). When in contrast the `Model` changes (by access through the simulation, a running script, or a concurrently displayed control operating on the same part of the database), the `EditPart` is also notified and can update its `Figure` if necessary.

## 4.3 Recording of Test Scripts

---

An optional requirement to the work is the possibility to record user interaction with AUTOSAR I/O GUI and generate the frame for a test script complying with that interaction (see [Requirement 14](#) in [Section 1.5](#)).

The design of AUTOSAR I/O GUI copes with that requirement by introducing a singleton `Recorder` object that is the contact point for all recording activity. It stores the information whether recording is currently enabled or not and whether the time passed between the manipulation of the GUI elements shall be recorded also or not. Both of these settings can be adjusted by the user through the corresponding menubar and toolbar items (see [Appendix A](#) for an overview of AUTOSAR I/O GUI's features and GUI).

Each time that the state of an input control in the central database is altered (through the actions of the user or a concurrently running test script), the database records an appropriate script text line representing that action using the `Recorder`. The `Recorder` manages the inclusion of a script header and the time measurement resulting in a corresponding script text line (e.g. `time.sleep (0.5)`) if enabled.

The source code that records the script line after a database state alteration is included in a programming entity called an aspect.

### 4.3.1 Short Introduction to Aspect-Oriented Programming

Aspect-oriented programming (AOP) is an extension to traditional programming paradigms. It modularizes existing code further by extracting so-called cross-cutting concerns into a separate code entity called an aspect [50]. Cross-cutting concerns are those that are effective on many different parts of the source code and therefore distributed all over it.

The recording concern of AUTOSAR I/O GUI is in fact a modified form of the classical tracing concern that captures function calls. Since tracing is *the* most popular example for deploying AOP, the definition of an aspect for the recording feature of AUTOSAR I/O GUI can help integrating the functionality without polluting the database source code.

An aspect is defined through one or more pieces of advice it gives. An advice operates after, before, or instead of other parts of the source code, depending on its type—after advice, before advice, or around advice, respectively. The target parts of the source code are determined by a so-called pointcut expression that can contain wildcard characters for the name, return type, and parameters of a function and many other modifiers to distinguish the context of the control flow, for example.

### 4.3.2 AUTOSAR I/O GUI's RecordingAspect

AUTOSAR I/O GUI uses AspectJ as an implementation of AOP extending the Java programming language (see e.g. [51] or [52] for an introduction and a thorough description of AspectJ) and therefore possesses an additional dependency on the plug-in of the AspectJ Runtime (see also [Section A.1](#)).

The concrete aspect defined in AspectJ is called `RecordingAspect`. Its advice code generically constructs the script line to record by exploiting methods offered by the `thisJoinPoint` object available at runtime. This object represents the point of the code where the aspect takes effect. It can there-

fore provide the function name, return type, and parameters of the affected method.

The method calls that are to be recorded are determined by the pointcut named `toBeRecorded`. This is the single point that is to be adapted when a new control is introduced to AUTOSAR I/O GUI (see [Section B.7](#)) and therefore provides for another part of AUTOSAR I/O GUI's genericity (see [Section 4.4.3](#)).

## 4.4 Genericity in AUTOSAR I/O GUI

---

In order to suit the requirement of easy extensibility (see [Requirement 5](#) in [Section 1.5](#)), AUTOSAR I/O GUI's architecture is kept as generic as possible. The most interesting points of that aspect of the program design are described in this section.

### 4.4.1 Providing Abstract Base Classes for Model, EditPart, and Figure

The functionality common to all GUI elements is extracted into common base classes in each part of GEF's Model–View–Controller triad (see also [Section 4.2.1](#)). This way most of AUTOSAR I/O GUI can work with references to only a base class type in a very generic kind of way.

#### The Model Base Class.

The provided class `AIOGUIModel` implements the following responsibilities:

- It manages the listening `EditPart` by providing `addListener ()` and `removeListener ()` functions.
- It stores the bounds of the corresponding GUI element (x position, y position, width, and height) and the methods to get and set them with appropriate notification of the `EditPart`.
- It implements `IPropertySource`, thereby effectively enabling Eclipse's properties view depending on the GUI element selection. It only provides support for the viewing and editing of the element's bounds; subclasses have to extend the functionality if desired (see also [Section 4.1.3](#)).

- It declares abstract methods to be overridden by concrete implementations in order to support generic saving and restoring (see [Section 4.4.2](#)).

#### The EditPart Base Class.

`AIUGUIEditPart` is the abstract `EditPart` base class provided by AUTOSAR I/O GUI, implementing the following functionality:

- It implements GEF's entry points `activate ()` and `deactivate ()` by starting or stopping to listen to the connected `Model`, accordingly. These functions are called by the framework on creation or deletion of a corresponding GUI element, respectively.
- It implements `PropertyChangeListener`, refreshing the visuals of the connected `Figure` whenever a change event comes in from the `Model`, the changed bounds of the figure, or the properties view.
- It is responsible for placing the figure according to its bounds when `refreshVisuals ()` is called.

#### The Figure Base Class.

The abstract base class to the figure part of the architecture, `AIUOGUIFigure`, defines a common look in its constructor and is responsible for managing the listening `EditPart` (if it bears input capabilities) by providing `addListener ()` and `removeListener ()` methods.

### 4.4.2 Saving and Restoring

The save-and-restore mechanism in AUTOSAR I/O GUI is completely generic and therefore fully operable even when new GUI elements are introduced—without adaptation.

#### Saving the Diagram.

The layout and the configuration of the different GUI controls is saved by iterating over all the model displayed on the current diagram and putting different attributes into an `XMLMemento` object (see also [Section 4.2.4](#)). An XML element `aioguimodel` is created for each model, saving its type in an attribute by getting the canonical name of the subclass the currently investigated model belongs to. After that, specific settings to be saved are retrieved from the model subclass which has to implement `getSpecificSettingsNames ()`

```
<?xml version="1.0" encoding="UTF-8"?>
<aiogui>
<aioguimodel analog="true" height="239"
  portname="/Dio/Dio/LED_PORT"
  type="de.dreisoft.aiogui.model.DIOOutputPortModel"
  width="77" x="619" y="296"/>
<aioguimodel channelname="/Dio/Dio/INPUT_PORT/INPUT_CHANNEL_0"
  height="74" level="true"
  type="de.dreisoft.aiogui.model.DIOInputChannelModel"
  width="173" x="650" y="35"/>
</aiogui>
```

Figure 4.2: Example AIOGUI XML file containing the data of a DIO output port and a DIO input channel.

and `getSpecificSettings ()` (see also [Section 4.4.1](#)). These key–value pairs are saved as attributes to the element. Finally, the bounds of the element (included in every kind of element) are also added as attributes.

An example XML file layout resulting from a save operation can be found in [Figure 4.2](#).

### Restoring a Diagram.

Restoring the layout of a diagram from the contents of a file is implemented by creating an `XMLMemento` object from the file content (see also [Section 4.2.4](#)). The children of the root model object (the diagram) are then iterated over. First, the type attribute is retrieved and a new model of that type is instantiated by the means of reflection (Java methods `Class.forName ()` and `Class.newInstance ()`). Then the bounds of the newly created model are set. In the following step the specific settings are set by exploiting `getSpecificSettingsNames ()` and `setSpecificSettings ()` *declared* in the abstract base class `AIOGUIModel` (see also [Section 4.4.1](#)) and *defined* in the concrete subclass, providing the necessary key–value pairs. Eventually `finalizeInitialization ()` is called to allow the newly created model to finalize the process if necessary.

### 4.4.3 Implementation of the Script Recording Feature

Support for AUTOSAR I/O GUI’s script recording feature is given through the use of aspect-oriented programming (see [Section 4.3](#)). The deployed

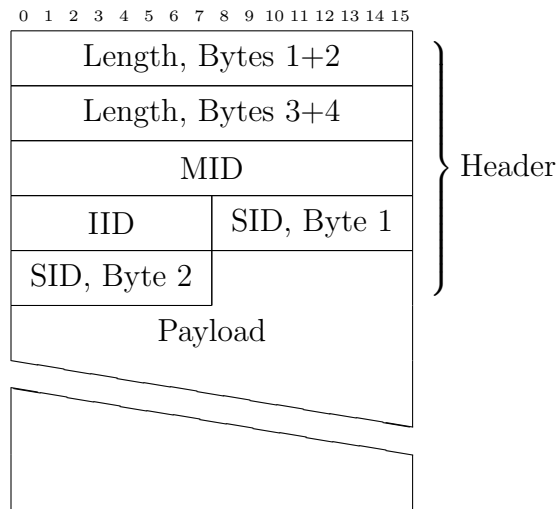


Figure 4.3: General message format for the communication AUTOSAR I/O GUI-simulation.

`RecordingAspect` is highly generic in the way that it constructs the script text line that is to be recorded dynamically at runtime. No specific code is necessary, the advice code fits all affected methods.

## 4.5 Definition of the Message Protocol Between AUTOSAR I/O GUI and the Simulation

The communication between AUTOSAR I/O GUI and the actual simulation relies on sockets as argued in [Section 3.4](#). Both sides send and receive messages depending on the occurrence of different events, namely distinct function calls. The protocol consisting of the exchanged messages and the triggering events is defined in this section and [Appendix C](#)—based on the information gathered in the analysis parts of [Sections 2.2](#) through [2.5](#).

### 4.5.1 Message Format

The general format of all messages exchanged by the simulation and the GUI is defined as follows. [Figure 4.3](#) gives an overview.

The message header starts with a length field that indicates the length of the whole message (including the header itself) in bytes. It was deliberately

chosen to be four bytes wide in order to support future API calls possibly transferring whole memory regions. The advantage in prepending the length information is the possibility of the communication module of either communicating part to receive the whole message first before forwarding it to the module that interprets its contents. Otherwise, the parsing of the Service ID (AUTOSAR's identification of an API function) would have to take place in the communication module already in order to determine the length of the message because the payload depends on the indicated function. That design would destroy the possibility to introduce a complete separation of concerns between the modules.

Every communication message, sent in either of the both directions, next has to append the Module ID (MID) of the calling or called module (type `uint16`) which is globally defined in [53]. The Instance ID (IID, type `uint8`) represents the driver instance, starting at 0, if applicable (e.g. there will always be only a single DIO driver, thus the DIO Instance ID will always be 0).

After that, a Service ID (SID) will be transmitted in order to be able to identify the called function. If the service is oriented at a function defined in AUTOSAR, the Service ID will be the one found in the AUTOSAR specification corresponding to the investigated module; in AUTOSAR, it is defined in the range of 0 to 255 (type `uint8`<sup>1</sup>). If in contrast the service can not be mapped to the AUTOSAR API, the Service ID will be an arbitrarily chosen one in the range from 256 to 65535 in order to distinguish from the Service IDs defined by AUTOSAR. Hence the Service ID field is of type `uint16`.

Eventually, the actual payload data is appended, containing, for example, the parameters of an AUTOSAR API call. If the parameter list is `void`, there is no payload data—the communication partner will just be notified of the function call itself. If a parameter is of a configurable AUTOSAR data type (e.g. `uint8`, `uint16` or `uint32`), the communication is always performed using the biggest and thus compatible one (in the case of the example: `uint32`). This way, the interface supports all kinds of configuration sets while only providing for a slight overhead in some cases.

The concrete messages that are sent and received in the current implementation and the chosen data types are listed in [Appendix C](#) for reference purposes.

---

<sup>1</sup>The definitions of the AUTOSAR data type ranges can be found in the DET specification [24].



### **Error Detection.**

Since the actual communication is performed through TCP based sockets, and TCP is a reliable transport protocol, no further error detection or correction mechanisms have to be implemented.

### **Byte Order.**

Since the Java Virtual Machine uses the big endian format to represent its data, and big endian is also the network byte order, no conversion on the side of AUTOSAR I/O GUI is needed. The simulation, however, written in C and possibly running on a little endian machine, must take that problem into account and use macros like `htonl ()` wherever needed.

## **4.5.2 Simulation Proxy**

The advantage of an early definition of the message protocol is the possibility to deploy a proxy for the simulation. This is especially interesting in the phase when the simulation is still under development but the GUI is ready to be tested. A “simulation of the simulation” in the case of AUTOSAR I/O GUI is a simple TCP/IP server dispatching the incoming messages and reacting by sending other ones back to the GUI, simulating a specific application behavior.

## **4.6 User Interface Design**

---

AUTOSAR I/O GUI was developed bearing human factors principles in mind. A really good introduction to this topic is the book “An Introduction to Human Factors Engineering” [29] which gathers information on that subject from many competent sources. It gives a good outline on the goals and methods of ergonomics in general and it also includes a chapter especially focusing on human–computer interaction.

This section picks out the relevant points and explains how they are realized in the user interface of AUTOSAR I/O GUI.

### **4.6.1 Goals of Human Factors**

Human factors engineering is supposed to serve four goals regarding the human–system interaction:

1. It shall reduce the occurrence and the negative effects of errors.

2. It shall increase the productivity of the human–machine system.
3. It shall enhance the safety of the system (not relevant in the context of this thesis).
4. It shall enhance the comfort of the operator using the system.

So the major principle is to consider the human operator when designing a system—preferably very early in the design process in order avoid expensive redesign steps. All of the following refined principles serve at least one of the four original goals.

### 4.6.2 Considering Multiple Scenarios

When designing a product not only the human actually using the running system has to be kept in mind. Also the people being in touch with the product during other lifecycle stages like the manufacturing or disposal have to be considered.

In the case of AUTOSAR I/O GUI, this primarily affects the people maintaining the program by changing or extending it at a later point in time. These people are helped by designing a well defined architecture (see [Section 4.1](#)) and documenting in the code as well as in a separate document as done in this thesis. Extension of the GUI is facilitated by providing a step-by-step documentation on how to introduce a new control (see [Appendix B](#)). The use of an object-oriented programming language (Java in this case) also helps programmers to find their way more quickly in foreign code compared to the use of a procedural programming language since object orientation encourages modular programming using classes and their instances.

### 4.6.3 User Analysis

One of the basic human factors commandments is sometimes postulated as “Know thy user.”. It is essential to realize that the people using a system are inherently different from the ones designing them and hence may face other difficulties. This is prevented by explicitly thinking about the potential user group and their abilities and keeping them in mind during the design.

The typical user of AUTOSAR I/O GUI will not be an average person using his personal computer but a highly specialized software developer configuring and testing his embedded system and application. He is also very likely to have a university degree in computer science or a similar study. This is why AUTOSAR I/O GUI can rely on metaphors common in engineering

circles like an LED for a digital channel displaying one of two possible values. The same consideration holds for the user maintaining or extending AUTOSAR I/O GUI—he will also be a highly skilled software engineer.

#### 4.6.4 Consistency

A basic but very important principle in human factors design is to design an interface in a consistent manner. That includes outer consistency with its environment and inner consistency inside the system itself. A properly consistent interface leads to faster usage because of the consideration of previous top-down knowledge in locating items (outer consistency) and reduced visual search time (outer and inner consistency).

AUTOSAR I/O GUI considers both aspects. It embeds in the environment of the operating system window manager by displaying native widgets the operator is already used to. This is done using SWT as a GUI toolkit as described in [Section 3.2.2](#) and through other measures like using system wide color definitions (e.g. the color of the title bar) for some GUI elements, for example.

The GUI also embeds into the Eclipse platform. Since the *tresos* GUI program (see [Section 1.6.3](#)) is also based on Eclipse, the user is already familiar with the environment from the configuration step of the AUTOSAR modules if he was not already before by using Eclipse as an integrated development environment, for example.

The inner consistency is provided by carefully designing the visual elements of the GUI and their behavior. Providing an abstract base implementation for the displayed **Figure** as described in [Section 4.4.1](#) is one step in ensuring a common visual appearance of all GUI elements.

#### 4.6.5 Icons and Redundancy

Since the human perception is susceptible to failure especially with cluttered displays containing only similar information, measures to antagonize errors help to increase usability.

For example, the overly excessive use of icons in current state of the art programs with a graphical user interface is only partly justified—pictograms *can* lead to increased usability, but only when the risks are taken care of. These include the discriminability of similar symbols and particularly the issue of correct interpretation of a symbol. If an icon is always interpreted correctly it has the big advantage that it is language independent and uses a different code (visual and not verbal) and hence different mental resources, effectively unburdening the human mind.

AUTOSAR I/O GUI makes use of icons only in addition to regular text labels and thus incorporates the advantages while eliminating the disadvantages.

### 4.6.6 Display Layout

A lot of research results are available concerning the layout of elements displayed on a screen. These include proper grouping of elements according to their relatedness, positioning of linked control and display next to each other following the principle of stimulus–response compatibility, and others.

By offering an editor-like interface in which the user can both choose *which* elements to display and *where* to display them, AUTOSAR I/O GUI leaves this task completely to the user and his personal preferences. It also allows the user to save the (probably sophisticated) layout in order to restore it whenever needed.

## 4.7 Summary

---

The chapter gave an overview of AUTOSAR I/O GUI's design as well as detailed information on selected interesting pieces of the architecture.

AUTOSAR I/O GUI's macro architecture focuses on a **central state database** and its accessing entities using separate interfaces. The micro architecture for each offered control consists of a triad of objects of types `Model`, `EditPart`, and `Figure`, roughly corresponding to the three objects described in the **Model–View–Controller design pattern**. This design pattern and several others are deployed in AUTOSAR I/O GUI which was described in a separate section. AUTOSAR I/O GUI's test script recording feature is modularized and kept highly generic using **aspect-oriented programming** techniques.

Another part of the chapter was dedicated to the extensibility of AUTOSAR I/O GUI. It described the implemented mechanisms to increase **genericity** (and therefore to allow for easy extension) by providing abstract classes and a save and restore system applicable to any kind of control. The definition of a **common message format** to use in the socket connection to the simulation (agreed upon in the analytical [Chapter 3](#)) was recorded in order to allow future module extensions to adhere to it.

The last section of the chapter was devoted to the discussion of basic **ergonomics principles**. It showed the importance of a consistent, redundant and user-oriented user interface and which measures are taken in AUTOSAR I/O GUI to achieve such an interface design.



---

## Summary and Prospects

---

This concluding chapter presents both a final summary of the results of the whole thesis (Section 5.1) and considerations about possible extensions to the developed prototype (Section 5.2).

### 5.1 Summary of the Results

---

The thesis at hand presented the steps on the way to an extensible architecture for a GUI program controlling and visualizing an ECU simulation.

The AUTOSAR driver modules DIO, CAN, EEPROM, and DET were selected for the prototype work, their environment was investigated using use cases, and their specification documents were analyzed for information relevant to the visualization and manipulation of the corresponding simulated devices. This includes the definition of the appropriate part of the AUTOSAR Core Host Data Interface between the simulation and the GUI program.

The environment of the whole AUTOSAR I/O GUI program was examined carefully to provide for a seamless integration and easy extensibility; Java and Eclipse were selected as the programming platforms, GEF as the graphical framework of choice, and sockets for the communication between the simulation and the GUI part.

The challenging feature requirement of an automated test access to the resulting AUTOSAR I/O GUI platform was deliberated thoroughly, reaching the decision to integrate the Java implementation of the Python scripting language—Jython—into the program.

These results from the analysis step in mind as additional requirements, an architecture design for AUTOSAR I/O GUI was developed providing a high level of genericity in order to allow for easy and flexible extensibility. This goal was reached through the provision of abstract classes implementing common useful functionality, through the design of a save-and-restore mechanism that does not require adaptation on the introduction of a new control, and a generically formulated advice for the script recording feature.

AUTOSAR I/O GUI's design also features different well-described software design patterns for enhanced code reusability and aspect-oriented programming techniques where appropriate.

Finally, the visual design of AUTOSAR I/O GUI was oriented at several basic human factors principles in order to achieve a better-than-average usability.

## 5.2 Future Work

---

This thesis and its connected implementation only provide the framework for future implementations to integrate by assisting the party extending AUTOSAR I/O GUI in several ways. Since merely few actual drivers were investigated and implemented, the remaining modules still need to go through those steps.

Nevertheless, AUTOSAR I/O GUI's generic architecture allows for a nearly unlimited range of controls to be displayed. The following sections shortly outline other ideas on how to extend AUTOSAR I/O GUI for other uses.

### 5.2.1 Displaying Internal Information

Besides the state of the driver modules defined in AUTOSAR, one could imagine additional information to be given concerning, for example, the internal state of the AUTOSAR operating system. Such information could be simple access violation messages or more sophisticated ones using custom (configurable) hooks in the operating system.

### 5.2.2 Connecting Several Instances of AUTOSAR I/O GUI to One Simulation

As outlined in [Section 3.4.4](#), the simulation could be adapted to accommodate more than a single AUTOSAR I/O GUI instance. This can be achieved by



using multicast UDP/IP sockets or by handling more than one TCP connection in parallel. The associated work would be mainly on the simulation (e.g. thinking of proper synchronization strategies for the requests from multiple GUIs), though—AUTOSAR I/O GUI would need little or no modification.

An example application of such an implementation would be a training environment where every participating party can interact with the simulated application through the instance of AUTOSAR I/O GUI running on their own PC.

### 5.2.3 Modifying the MCAL on the Target to Communicate With AUTOSAR I/O GUI

As mentioned in [Section 3.4.4](#), it would be possible to test an application and the upper AUTOSAR layers not only when simulated on a PC host but also when running on the target platform. Therefore the target's MCAL needs to be adapted in a way so that it uses a TCP/IP stack to communicate with an instance of AUTOSAR I/O GUI. The modified MCAL then does not operate on the real devices (e.g. DIO pins) but uses AUTOSAR I/O GUI as a communication point for all input and output activity.

### 5.2.4 Using AUTOSAR I/O GUI for Software Component Tracing

Elektrobit Automotive has already developed a concept for manual and automated testing of AUTOSAR Software Components; most of the following information is based on a draft document by Daniel Kerk [\[54\]](#).

#### Software Component Unit Tests.

AUTOSAR applications consist of one or more AUTOSAR Software Components which encapsulate certain functionality. They communicate with their environment via dedicated Require Ports—for input—and Provide Ports—for output (see [Section 1.6.1](#) for references for further reading).

Because of the encapsulation and the well-defined interface, Software Components can be perfectly unit tested. The proposed framework separates between a test trace kernel linked with the AUTOSAR software and the actual Software Components, and a host application communicating and controlling it. The requirements for that host application fit AUTOSAR I/O GUI perfectly: It shall be platform independent, on a PC basis, integrated with *tresos* GUI in Eclipse, and support automation through test scripts.

AUTOSAR I/O GUI can easily be extended to provide the necessary controls for controlling the RTE, stimulating of signals, and rendering results of run unit tests.

### **Virtual Functional Bus Tracing.**

Besides the black-box unit testing, the Software Component developer can rely on a feature called Virtual Functional Bus (VFB) Tracing offered by the RTE. If VFB Tracing is enabled, hook functions are called at specific RTE internal events and certain locations in the RTE code. According to the proposed framework, the test trace kernel shall implement those hook functions and then notify the host application accordingly.

This functionality can also be delegated to AUTOSAR I/O GUI when properly extended.





---

# Features

---

This chapter shortly describes the features currently implemented in AUTOSAR I/O GUI and serves as a small user guide.

## A.1 Installing AUTOSAR I/O GUI

---

Since AUTOSAR I/O GUI is implemented as an Eclipse-based plug-in, it is deployed and integrated into *tresos* GUI by copying of the AIOGUI jar file to the `plugins/` directory of the Rich Client Platform that constitutes *tresos* GUI.

Additionally, it poses some dependency demands on its environment:

- GEF plug-in needs to be available (mandatory)
- Draw2D plug-in needs to be available (mandatory)
- AspectJ Runtime plug-in needs to be available (mandatory, see [Section 4.3.2](#))
- Properties view plug-in needs to be available (if properties need to be displayed)
- *tresos* error log view needs to be available

AUTOSAR I/O GUI needs all of the mentioned plug-ins for full functionality, thus their availability has to be ensured.

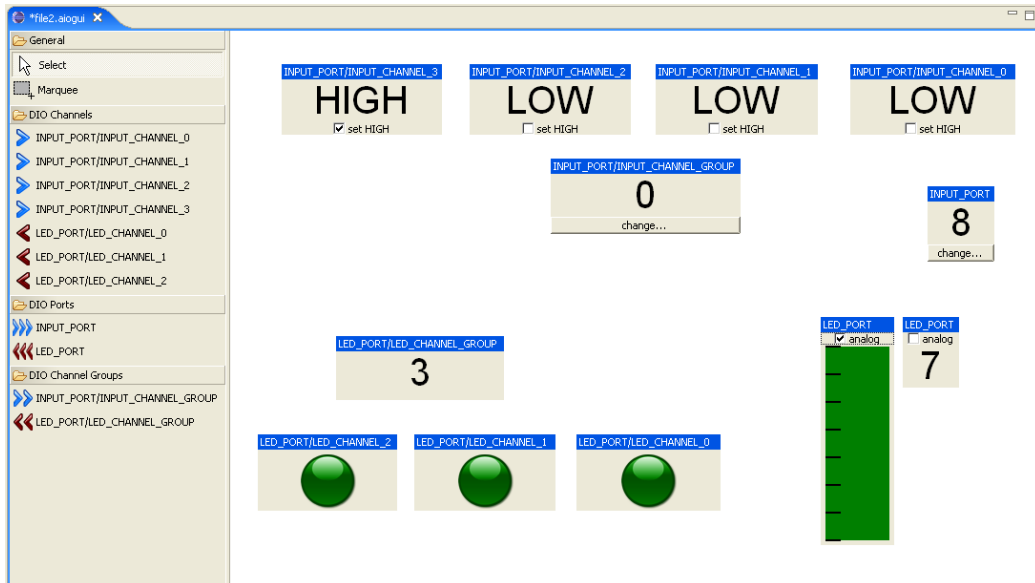


Figure A.1: An example AUTOSAR I/O GUI layout showing all DIO controls; input controls on the top, output controls on the bottom.

## A.2 Invoking AUTOSAR I/O GUI

---

Since the surrounding *tresos* GUI program is not yet fully implemented, the contact point for AUTOSAR I/O GUI is still missing. It is supposed to integrate in the project browser displaying all available configurations of one or more ECUs and to enable the user to start a new AUTOSAR I/O GUI layout with the selected configuration or open an existing one. In either case, the palette on the left of the editor is populated only with the controls available depending on the configuration. If an existing layout is opened, the consistency between the displayed elements and the supplied configuration is checked additionally.

## A.3 AUTOSAR I/O GUI's User Interface

---

### A.3.1 The GUI

AUTOSAR I/O GUI's editor window is shown in [Figure A.1](#). The editor's menubar and toolbar are depicted in [Figure A.2](#) and [Figure A.3](#), respectively.



Figure A.2: AUTOSAR I/O GUI's menubar.



Figure A.3: AUTOSAR I/O GUI's toolbar.

The properties view (see [Figure A.4](#)) shows important properties of the currently selected control—both visual (the bounds) and control-specific ones. The visual properties can be directly modified in the properties view—even if multiple controls are selected via the marquee tool. This is very useful to align various controls to a common x or y coordinate.

### A.3.2 Keyboard Access

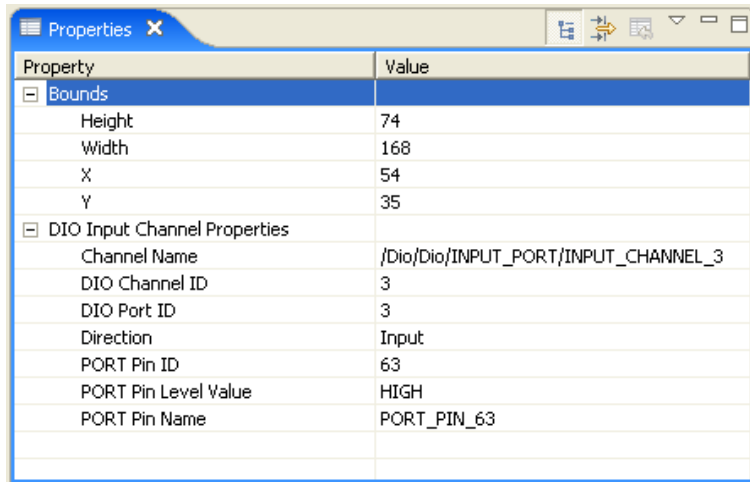
AUTOSAR I/O GUI's layout can also partly be modified using the keyboard.

If an AUTOSAR I/O GUI diagram is active, the cursor keys can be used to select a displayed control. When a control is selected, it can be clipped for dragging by pressing the period (“.”) key. Successive cursor key movements drag the control as desired; a final enter key or return key press commits the change, an escape key press aborts it.

## A.4 Populating the Displayed Diagram

After having selected a configuration for AUTOSAR I/O GUI, the user can modify the layout by:

- Populating it with new controls from the palette toolbar. The current implementation offers all available DIO channels, port, channel groups, and a DET error counter.



Property	Value
<b>Bounds</b>	
Height	74
Width	168
X	54
Y	35
<b>DIO Input Channel Properties</b>	
Channel Name	/Dio/Dio/INPUT_PORT/INPUT_CHANNEL_3
DIO Channel ID	3
DIO Port ID	3
Direction	Input
PORT Pin ID	63
PORT Pin Level Value	HIGH
PORT Pin Name	PORT_PIN_63

Figure A.4: AUTOSAR I/O GUI's properties view.

- Modifying the size and the location of single controls. This can be done by dragging the control directly or by modifying the values through the properties view (see [Section A.3.1](#)).
- Deleting controls from the diagram.
- Undoing or redoing one of the above changes through the corresponding toolbar buttons, menu items (“Edit – Undo” and “Edit – Redo”), or context menu items.

The layout can even be modified at runtime when the connection to the simulation is already established—the controls that are newly introduced are instantiated with the current state automatically.

One control can be displayed multiple times simultaneously—AUTOSAR I/O GUI cares for the synchronisation of them.

## A.5 Saving the Populated Diagram

---

The current layout, including the input states of the input controls, can be saved to a file. This is done using “File – Save” or “File – Save As...” or the corresponding toolbar items.

The stored layout can then be reopened, provided that the given configuration is not incompatible with the information defined in the layout (e.g. different DIO channel direction for a given DIO channel name).



## A.6 Managing the Connection to the Simulation

---

When the user wants AUTOSAR I/O GUI to become active, the connection to the simulation has to be established. This is done through the corresponding menu item (“AUTOSAR I/O GUI – Connect To Simulation”) or toolbar button. The user is then asked for the hostname and the port that the simulation is currently listening to. A successful connection is indicated in the status line.

If the simulation needs to be stopped, the menu item “AUTOSAR I/O GUI – Disconnect From Simulation” or the corresponding toolbar button can be used. The current connection status in the status line is then updated.

## A.7 Replaying Test Access Scripts

---

AUTOSAR I/O GUI features an automated test access interface enabling the user to replay actions defined in a script file. The syntax to be used is the one of the Python script language and all of the common built-in Python modules like `time` can be used.

The interface to AUTOSAR I/O GUI is exported in the object named `gui`, the available methods are those listed in the `ModelScriptingInterface`. For an example script file see [Figure 3.4](#) in [Section 3.3.4](#).

Available test scripts can be executed using the menu item “AUTOSAR I/O GUI – Run Script” or the corresponding toolbar item. It is then executed in the background, reading the current state and modifying it if applicable. The user can then watch the GUI controls changing and concurrently intervene himself. If AUTOSAR I/O GUI is currently connected to the simulation, all scripted modifications of input controls are also sent to the simulation, of course.

It is even possible to execute several scripts in parallel. Accesses to the internal state and the communication to the simulation are synchronized internally, so no conflicts will arise.

## A.8 Recording Test Access Scripts

---

AUTOSAR I/O GUI can assist the user in writing his test scripts by recording his manual interaction with the GUI. The start of the recording

session is triggered through the menu item “AUTOSAR I/O GUI – Record Script” or the appropriate toolbar button. The user is then asked for a filename to save the recorded script sequence to and asked again before overwriting an existing file. Beginning with that point in time, all the user interactions resulting in communication with the simulation (input state changes) are recorded to the given script file using the appropriate syntax. The recording is only stopped when the user selects the menu item “AUTOSAR I/O GUI – Stop Recording Script” or presses the corresponding toolbar button.

The user of AUTOSAR I/O GUI can choose whether he wants to record the time that passes between the manipulation of two GUI controls also or not. This is determined by the setting of the menu item “AUTOSAR I/O GUI – Time Recording” and the appropriate toolbar item. If this feature is enabled, the recorded script file bears `time.sleep ()` calls between input state manipulation calls.

If one or more scripts are running concurrently to the recording and manual interaction with AUTOSAR I/O GUI, *all* of this activity combined is recorded.

---

# How to Introduce a New Control

---

This chapter explains in detail which steps are to be taken when a new control is being developed in AUTOSAR I/O GUI. It is primarily thought as a “how to” for developers extending AUTOSAR I/O GUI but also serves as a developer-oriented overview of program internals.

## B.1 Defining Custom Model, EditPart, and Figure

---

The actual task when introducing a custom control is the definition of the three parts of the Model–View–Controller architecture (see also [Section 4.2.1](#)). As comprehensively described in [Section 4.4.1](#), abstract base classes for all three of them are provided, implementing the functionality that is common to all controls.

So only specific functions have to be introduced in the derived classes, some suggestions follow (see also [Section 4.1.3](#) for the main responsibilities of an output or input control).

### B.1.1 Defining the Model Class

You may want to redefine the field `_descriptors` in order to add additional information displayed in the properties view when an instance of the new con-

control is selected. This includes assigning new validators if necessary and an implementation of `getPropertyValue ()` and `setPropertyValue ()`, calling the super implementation eventually.

Every subclass of `AIOGUIModel` also needs to implement four functions as explained in [Section 4.4.2](#) in order to provide the generic save-and-restore mechanism with the data it needs.

### B.1.2 Defining the `EditPart` Class and Adding It to the Factory

The specific `EditPart` has to provide the connection between the `Model` and the `Figure` by implementing:

1. `createFigure ()`: Returns the `Figure` initialized with the state of the underlying `Model`.
2. `refreshVisuals ()`: Is called by the framework to synchronize the displayed `Figure` with its `Model`. The super implementation should be called first and `repaint ()` should be used in the end when necessary.
3. `propertyChange ()`: If the control is an *input* control, this method should contain a condition whether the event was issued by the `Figure` or the `Model`. `Model` changes should be processed by calling the super implementation, `Figure` changes should be processed in the current class by adjusting the model appropriately.

The newly defined `EditPart` also needs to be added to the `AIOGUIEditPartFactory` which the framework uses in order to instantiate the `EditPart` corresponding to a given `Model`. Therefore another branch needs to be introduced in `createEditPart ()`.

### B.1.3 Defining the `Figure` Class

The `Figure` is the concrete visual implementation and therefore includes the actual `Draw2D` graphical elements. It is also supposed to provide methods to react to a change of state of the associated `Model` in a visual manner.

If the control is an *input* control, the appropriate `Draw2D` elements capable to receive input have to be deployed, and occurring manipulation events have to be forwarded to the `EditPart` through a notification.

## B.2 Adapting the Internal Configuration

---

The external configuration provided by the *tresos* GUI application is analyzed internally in AUTOSAR I/O GUI and only the relevant pieces of information are stored in the internal configuration.

### B.2.1 Adapting the Configuration Class

The class `AIIOGUIConfiguration` holds the internal configuration state. It is therefore to be extended in a way to provide all the configuration details needed lateron. This includes introducing internal fields to store the actual information as well as introducing methods in order to retrieve it.

### B.2.2 Adapting the Parsing Step

The step performing the transition from the external to the internal configuration also needs adaptation. It is supposed to retrieve the relevant information items and store them in the adapted internal configuration class instance in `initializeConfiguration ()`.

## B.3 Extending the Model Factory

---

AUTOSAR I/O GUI's `AIIOGUIModelFactory` serves the purpose of instantiating a new `Model` using its name and the configuration provided in the constructor. Because the constructor of the `Model` depends on its actual subtype, a new branch has to be introduced to `getNewObject ()` for each kind of control. It shall return a new instance of the control `Model` class using its constructor and implementation specific information extracted from the configuration and the model name.

If this mechanism is too sophisticated for a new type of control, a custom (more simple) factory may be provided and used instead of `AIIOGUIModelFactory` when defining the palette entries (see [Section B.4](#)).

## B.4 Extending the Palette

---

The AUTOSAR I/O GUI editor has a palette attached to its left side that accommodates all the available controls or subcontrols. The items offered in

that place depend on the chosen configuration (e.g. if DIO is not enabled in the chosen configuration, no DIO channels will be offered to introduce in the editor).

The method to extend in order to introduce a new control to the palette is `getPaletteRoot ()` in `AIOGUIEditor`. The internal configuration is found in the field `_config` and can be used to define palette entries dependent on the configuration (e.g. if enabled at all or other configuration details).

## B.5 Upgrading the Central State Database and the Access Interfaces

---

The central state database and its associated interfaces for the scripting, model, and simulation part will also need extension since this is the part where the communication with the actual simulation takes place. Therefore the interfaces to introduce in `ModelScriptingInterface` and `SimulationInterface` have to be thought of—which is a module-specific task. Next the newly introduced methods have to be implemented in `AIOGUIDatabase` and new fields that save the state of the new module probably have to be introduced.

Be sure to choose the function names to be introduced to `ModelScriptingInterface` wisely—this is the interface that all scripted activity is based on and therefore the one that constitutes the DSL character of the test access (see also [Section 3.3.4](#)).

Moreover, it may be necessary to extend the pointcut in `UiThreadExecutionAspect` in such a way that database state alterations resulting in a change of displayed `Figures` get executed in the user interface thread (see [Section 4.1.4](#) for more about that difficulty).

## B.6 Adapting the AUTOSAR Core Host Data Interface and the Communication Thread

---

The next change concerns the interface between the GUI and the simulation, the AUTOSAR Core Host Data Interface. It will need a thorough extension implemented on both sides of the interface. On the side of the GUI, the processing of simulation requests and the sending of GUI requests happens in the `CommunicationThread` class which is to be adapted accordingly.

## B.7 Extending the RecordingAspect Pointcut Expression

---

In order for the test script recording feature (see [Section 4.3](#)) to work correctly and completely, the `RecordingAspect` has to be minimally adapted. Since the advice code itself is completely generic (see [Section 4.4.3](#)), only the pointcut expression `toBeRecorded` determining the database calls to be recorded needs extension.

This is only necessary if the introduced control offers the ability to modify database input states, though; if the control is of read-only nature, no recording is possible.

## B.8 Miscellaneous Hints

---

1. Make use of the interface `AIOGUIConstants`. This way problems occurring because of not matching string constants (e.g. resulting from a typing error) are excluded by design. Moreover, the definition of a central contact point allows quick and global change of appearance settings like the color of a title bar, for example.  
Be sure to use the prefixes already defined in the interface (e.g. `PROPERTY_PREFIX_FIGURE`) in order to make the program work correctly.
2. Take advantage of the `AIOGUIHelper` class. This is a central class containing useful static methods which are needed at several points in the code.





---

# Definition of Exchanged Messages

---

This chapter lists the concrete messages exchanged between AUTOSAR I/O GUI and the simulation in the current implementation. It is intended for reference purposes—see [Section 4.5](#) for the general message protocol.

## C.1 DIO Messages

---

The definition of the messages exchanged between the DIO simulation module and the corresponding GUI part is based on the analysis of the DIO AUTOSAR module performed in [Section 2.2](#); it can be found in [Figure C.1](#).

The messages sent to the simulation correspond to a notification of an input facility resulting from input activity on GUI side (through the user or a running script). These notifications are custom functions and therefore have a custom Service ID.

## C.2 CAN Messages

---

An overview of the messages exchanged between the CAN modules of the simulation and the GUI can be found in [Figure C.2](#). The attendant argumentation preceding the concrete message design definition is found in [Section 2.3](#).

APPENDIX C. DEFINITION OF EXCHANGED MESSAGES

---

MID	IID	SID	Function	Parameters
<i>Direction Simulation To GUI</i>				
120	0	1	Dio_WriteChannel ()	uint32 ChannelId, uint32 Level
120	0	3	Dio_WritePort ()	uint32 PortId, uint32 Level
120	0	5	Dio_WriteChannelGroup ()	uint32 PortId, uint8 Offset, uint32 Mask, uint32 Level
<i>Direction GUI To Simulation</i>				
120	0	256	Dio_ChannelChanged ()	uint32 ChannelId, uint32 Level
120	0	257	Dio_PortChanged ()	uint32 PortId, uint32 Level
120	0	258	Dio_ChannelGroupChanged ()	uint32 PortId, uint8 Offset, uint32 Mask, uint32 Level

Figure C.1: DIO module messages.

MID	IID	SID	Function	Parameters
<i>Direction Simulation To GUI</i>				
80	0–255	6	Can_Write ()	uint32 TransmitHandle, uint32 MessageId, uint8 Length, uint8 [Length] Data
<i>Direction GUI To Simulation</i>				
80	0–255	256	Can_NewMessage ()	uint32 TransmitHandle, uint32 MessageId, uint8 Length, uint8 [Length] Data

Figure C.2: CAN module messages.

### C.3 EEPROM Messages

---

Exchanged messages between the EEPROM module of the simulation and the GUI include the ones listed in [Figure C.3](#). The corresponding analysis part is [Section 2.4](#)—among other things it justifies the decision to use custom Service IDs for the write and read services although the services are very much oriented at the ones defined by AUTOSAR.

### C.4 DET Messages

---

The messages that can be sent from the DET simulation module to the GUI are listed in [Figure C.4](#). The communication protocol of this module is oneway only as argued in [Section 2.5](#).

### C.5 Service Messages

---

Service messages are those internal to the pair of simulation and AUTOSAR I/O GUI. They are used to communicate states and data other than the ones from the simulated AUTOSAR modules, therefore they bear

APPENDIX C. DEFINITION OF EXCHANGED MESSAGES

---

MID	IID	SID	Function	Parameters
<i>Direction Simulation To GUI</i>				
90	0-255	0	Eep_Init ()	—
90	0-255	256	Eep_WriteCells ()	uint32 EepromAddress, uint32 Length, uint32 [Length] Data
90	0-255	257	Eep_EraseCells ()	uint32 EepromAddress, uint32 Length
<i>Direction GUI To Simulation</i>				
90	0-255	258	Eep_CellsChanged ()	uint32 EepromAddress, uint32 Length, uint32 [Length] Data

Figure C.3: EEPROM module messages.

MID	IID	SID	Function	Parameters
<i>Direction Simulation To GUI</i>				
15	0	0	Det_Init ()	—
15	0	1	Det_ReportError ()	uint16 ModuleId, uint8 InstanceId, uint8 ApiId, uint8 ErrorId
15	0	2	Det_Start ()	—

Figure C.4: DET module messages.

## APPENDIX C. DEFINITION OF EXCHANGED MESSAGES

---

MID	IID	SID	Function	Parameters
<i>Direction GUI To Simulation</i>				
1313	0	0	Disconnect ()	—

Figure C.5: Service messages.

an artificial Module ID of 1313. See [Figure C.5](#) for an overview.

The disconnect message signals to the simulation that the user wants to disconnect and the socket connection will be closed shortly. This way the simulation can distinguish a disconnection on purpose from one resulting from a network error.



---

---

# Bibliography

---

- [1] Thomas Seydel, 2002. *Entwurf und Implementierung einer grafischen Oberfläche für einen Betriebssystemsimulator*. Diploma Thesis, Friedrich-Alexander University, Erlangen-Nuremberg
- [2] S. Bradner, 1997. *Key words for use in RFCs to Indicate Requirement Levels*. <http://www.ietf.org/rfc/rfc2119.txt>
- [3] AUTOSAR GbR, 2006. *Automotive Open System Architecture—Official Website*, <http://www.autosar.org>
- [4] Matthias Homann, 2005. *OSEK – Betriebssystem-Standard für Automotive und Embedded Systems*. 2., überarbeitete Auflage, Bonn
- [5] AUTOSAR GbR, 2006. *Technical Overview*. Version 2.0.0. [http://www.autosar.org/download/AUTOSAR\\_TechnicalOverview.pdf](http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf)
- [6] Dr. Berthold Daum, 2005. *Java-Entwicklung mit Eclipse 3.1*. 3., überarbeitete Auflage, Heidelberg
- [7] Sherry Shavor, Jim D’Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy, 2004. *Eclipse – Anwendungen und Plug-Ins mit Java entwickeln*. 1. Auflage, München
- [8] Azad Bolour, 2003. *Notes on the Eclipse Plug-in Architecture*. [http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin\\_architecture.html](http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html)

## BIBLIOGRAPHY

---

- [9] Guido van Rossum, 2006. *Python Tutorial*. Release 2.4.3. <http://docs.python.org/tut/tut.html>
- [10] Wikipedia, The Free Encyclopedia, 2006. *Monty Python*. Date of last revision: 21 September 2006 08:43 UTC. [http://en.wikipedia.org/w/index.php?title=Monty\\_Python&oldid=76948977](http://en.wikipedia.org/w/index.php?title=Monty_Python&oldid=76948977)
- [11] Merriam-Webster Online Dictionary, 2006. *Definition of python*. <http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=python>
- [12] Mark Lutz, David Ascher, 2000. *Einführung in Python*. 1. Auflage, Köln
- [13] AUTOSAR GbR, 2006. *Layered Software Architecture*. Version 2.0.0. [http://www.autosar.org/download/AUTOSAR\\_LayeredSoftwareArchitecture.pdf](http://www.autosar.org/download/AUTOSAR_LayeredSoftwareArchitecture.pdf)
- [14] AUTOSAR GbR, 2006. *ECU Configuration Parameters*. Version 0.08.
- [15] AUTOSAR GbR, 2006. *Specification of DIO Driver*. Version 2.0.0. [http://www.autosar.org/download/AUTOSAR\\_SWS\\_DIO\\_Driver.pdf](http://www.autosar.org/download/AUTOSAR_SWS_DIO_Driver.pdf)
- [16] AUTOSAR GbR, 2006. *Specification of PORT Driver*. Version 2.0.1. [http://www.autosar.org/download/AUTOSAR\\_SWS\\_Port\\_Driver.pdf](http://www.autosar.org/download/AUTOSAR_SWS_Port_Driver.pdf)
- [17] AUTOSAR GbR, 2006. *Requirements on DIO Driver*. Version 2.0.1. [http://www.autosar.org/download/AUTOSAR\\_SRS\\_DIO\\_Driver.pdf](http://www.autosar.org/download/AUTOSAR_SRS_DIO_Driver.pdf)
- [18] Wolfhard Lawrenz (publisher), 2000. *CAN – Controller Area Network – Grundlagen und Praxis*. 4., überarbeitete Auflage, Heidelberg
- [19] Konrad Etschberger (publisher), 2000. *Controller-Area-Network – Grundlagen, Protokolle, Bausteine, Anwendungen*. München, Wien
- [20] Thomas Betz, 2002. *Entwurf und Implementierung einer CAN Restbus-Simulation unter Windows NT*. Diploma Thesis, Georg-Simon-Ohm University of Applied Sciences, Nuremberg
- [21] AUTOSAR GbR, 2006. *Specification of CAN Driver*. Version 2.0.1. [http://www.autosar.org/download/AUTOSAR\\_SWS\\_CAN\\_Driver.pdf](http://www.autosar.org/download/AUTOSAR_SWS_CAN_Driver.pdf)
- [22] AUTOSAR GbR, 2006. *Specification of CAN Interface*. Version 2.0.0. [http://www.autosar.org/download/AUTOSAR\\_SWS\\_CAN\\_Interface.pdf](http://www.autosar.org/download/AUTOSAR_SWS_CAN_Interface.pdf)



- [23] AUTOSAR GbR, 2006. *Specification of EEPROM Driver*. Version 2.0.0. [http://www.autosar.org/download/AUTOSAR\\_SWS\\_EEPROM\\_Driver.pdf](http://www.autosar.org/download/AUTOSAR_SWS_EEPROM_Driver.pdf)
- [24] AUTOSAR GbR, 2006. *Specification of Development Error Tracer*. Version 2.0.1. [http://www.autosar.org/download/AUTOSAR\\_SWS\\_DET.pdf](http://www.autosar.org/download/AUTOSAR_SWS_DET.pdf)
- [25] Regionales Rechenzentrum für Niedersachsen / Universität Hannover (publisher), 2002. *Java 2 – Grundlagen und Einführung*. 2., veränderte Auflage, Bonn
- [26] Sun Microsystems, Inc., 2006. *Learning Swing by Example*. <http://java.sun.com/docs/books/tutorial/uiswing/learn/example2.html>
- [27] Object Technology International, Inc., 2003. *Eclipse Platform Technical Overview*. <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>
- [28] Eclipse Developers, 2006. *SWT: The Standard Widget Toolkit—Official Website*. <http://www.eclipse.org/swt/>
- [29] Christopher D. Wickens, Sallie E. Gordon, Yili Liu, 1998. *An Introduction to Human Factors Engineering*. New York
- [30] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, Philippe Vanderheyden, 2004. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. First Edition. <http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf>
- [31] Dr. Berthold Daum, 2005. *Rich-Client-Entwicklung mit Eclipse 3.1 – Anwendungen entwickeln mit der Rich Client Platform*. 1. Auflage, Heidelberg
- [32] Arie van Deursen, Paul Klint, Joost Visser, 2000. *Domain-Specific Languages: An Annotated Bibliography*. ACM SIGPLAN Notices, 35(6):26–36. <http://homepages.cwi.nl/~arie/papers/dslbib/>
- [33] Marjan Mernik, Jan Heering, Anthony M. Sloane, 2005. *When and How to Develop Domain-Specific Languages*. ACM Computing Surveys, 37(4):316-344. <http://ftp.cwi.nl/CWIreports/SEN/SEN-E0517.pdf>

## BIBLIOGRAPHY

---

- [34] Diomidis Spinellis, 2001. *Notable design patterns for domain-specific languages*. Journal of Systems and Software, 56(1):91-99. <http://www.dmst.aueb.gr/dds/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.pdf>
- [35] Robert Tolksdorf. *Programming Languages for the Java Virtual Machine*. <http://www.robert-tolksdorf.de/vmlanguages>
- [36] David Kearns, 2002. *Java scripting languages: Which is right for you?*. JavaWorld. <http://www.javaworld.com/javaworld/jw-04-2002/jw-0405-scripts.html>
- [37] David Kearns, 2005. *Choosing a Java scripting language: Round two*. JavaWorld. [http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-scripting\\_p.html](http://www.javaworld.com/javaworld/jw-03-2005/jw-0314-scripting_p.html)
- [38] Jython Developers, 2006. *Jython User Guide*. <http://www.jython.org/Project/userguide.html>
- [39] Jython developers. *Jython Software License*. <http://www.jython.org/Project/license.html>
- [40] Samuele Pedroni, Noel Rappin, 2002. *Jython Essentials*. First Edition, Sebastopol
- [41] Red Robin Software, 2006. *JyDT Project Homepage*. <http://www.redrobinssoftware.net/jydt/index.html>
- [42] Andreas Jung, 2002. *Schlange im Kaffee*. Linux-Magazin 01/2002. <http://www.linux-magazin.de/Artikel/ausgabe/2002/01/python/python.html>
- [43] Sun Microsystems, Inc., 2004. *Java Native Interface*. <http://java.sun.com/j2se/1.5.0/docs/guide/jni/>
- [44] Adrian Ofterdinger, 2005. *Java Native Interface ab J2SE 1.4*. [http://www.sigs.de/publications/js/2005/05/ofterdinger\\_JS\\_05\\_05.pdf](http://www.sigs.de/publications/js/2005/05/ofterdinger_JS_05_05.pdf)
- [45] Sheng Liang, 1999. *The Java Native Interface Programmer's Guide and Specification*. <http://java.sun.com/docs/books/jni/download/jni.pdf>
- [46] Object Management Group, Inc., 2006. *Introduction To OMG Specifications*. <http://www.omg.org/gettingstarted/specintro.htm#CORBA>

- [47] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995. *Design Patterns – Elements Of Reusable Object-Oriented Software*. First Edition, Amsterdam
- [48] Steve Burbeck, 1992. *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*. <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>
- [49] Randy Hudson, 2006. *Create an Eclipse-based application using the Graphical Editing Framework*. <http://www-128.ibm.com/developerworks/opensource/library/os-gef/>
- [50] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, Mehmet Akşit (editors), 2004. *Aspect-Oriented Software Development*. First Printing, Boston
- [51] Joseph D. Gradecki, Nicholas Lesiecki, 2003. *Mastering AspectJ – Aspect-Oriented Programming in Java*. Indianapolis
- [52] Ramnivas Laddad, 2003. *AspectJ in Action – Practical Aspect-Oriented Programming*. Greenwich
- [53] AUTOSAR GbR, 2006. *List of Basic Software Modules*. Version 1.0.0. [http://autosar.org/download/AUTOSAR\\_BasicSoftwareModules.pdf](http://autosar.org/download/AUTOSAR_BasicSoftwareModules.pdf)
- [54] Daniel Kerk, 2006. *Software Component Trace and Test Framework*. Version 0.10. Elektrobit Automotive draft document



---

---

# List of Figures

---

1.1	AUTOSAR software architecture [5]. . . . .	7
2.1	AUTOSAR driver classes [13]. . . . .	14
2.2	The AUTOSAR Core Host Data Interface. . . . .	16
2.3	CAN simulation scenario 1: Independent communication partners. . . . .	22
2.4	CAN simulation scenario 2: Mirroring the simulation. . . . .	23
3.1	A typical Swing dialog [26]. . . . .	33
3.2	A typical SWT dialog on the Windows platform [28]. . . . .	34
3.3	Pseudo code test script for use case “Scripting Access Test”. . . . .	36
3.4	Python test script for use case “Scripting Access Test”. . . . .	40
4.1	AUTOSAR I/O GUI’s macro architecture. . . . .	46
4.2	Example AIOGUI XML file containing the data of a DIO output port and a DIO input channel. . . . .	60
4.3	General message format for the communication AUTOSAR I/O GUI–simulation. . . . .	61
A.1	An example AUTOSAR I/O GUI layout showing all DIO controls; input controls on the top, output controls on the bottom. . . . .	II
A.2	AUTOSAR I/O GUI’s menubar. . . . .	III
A.3	AUTOSAR I/O GUI’s toolbar. . . . .	III
A.4	AUTOSAR I/O GUI’s properties view. . . . .	IV
C.1	DIO module messages. . . . .	XIV

*LIST OF FIGURES*

---

C.2 CAN module messages. . . . .	XV
C.3 EEPROM module messages. . . . .	XVI
C.4 DET module messages. . . . .	XVI
C.5 Service messages. . . . .	XVII

---

---

# Glossary

---

<b>AIOGUI</b>	Short name for AUTOSAR I/O GUI.
<b>AOP</b>	Aspect-Oriented Programming: An extension to traditional programming paradigms, modularizing cross-cutting concerns ( <a href="#">Section 4.3.1</a> ).
<b>API</b>	Application Programming Interface: A well-defined interface to a specific module or program.
<b>AUTOSAR</b>	Automotive Open System Architecture: An organization working on an open standard as well as the name for the standard itself (see <a href="#">Section 1.6.1</a> ).
<b>AUTOSAR I/O GUI</b>	The title of this thesis and the name of the program to be developed.
<b>AUTOSAR SC</b>	AUTOSAR Standard Core: The AUTOSAR modules in the service layer, hardware abstraction layer, and microcontroller abstraction layer (see <a href="#">Section 2.1</a> ).
<b>AWT</b>	Abstract Window Toolkit: The original GUI toolkit for the Java platform, developed by Sun (see <a href="#">Section 3.2.1</a> ).
<b>BSW</b>	Basic Software: The AUTOSAR module layers beneath the RTE (see <a href="#">Section 2.1</a> ).

<b>CAN</b>	Controller Area Network: A commonly used bus system in the automotive area (see <a href="#">Section 2.3</a> ).
<b>CORBA</b>	Common Object Request Broker Architecture: A middleware architecture with numerous services (see <a href="#">Section 3.4.3</a> ).
<b>DET</b>	Development Error Tracer: The AUTOSAR module tracking errors occurring in the development phase (see <a href="#">Section 2.5</a> ).
<b>DIO</b>	Digital I/O: The AUTOSAR module providing access to digital signals (see <a href="#">Section 2.2</a> ).
<b>DSL</b>	Domain-Specific Language: A language defined to describe problems specific to an area of application (see <a href="#">Section 3.3.2</a> ).
<b>Eclipse</b>	An extensible framework with a dynamic plugin architecture (see <a href="#">Section 1.6.2</a> ).
<b>ECU</b>	Embedded Control Unit.
<b>EEPROM</b>	Electrically Erasable Programmable Read-Only Memory: A specific kind of memory and the name of the managing AUTOSAR module (see <a href="#">Section 2.4</a> ).
<b>Flash</b>	A memory storage technology and the name of the corresponding AUTOSAR module.
<b>FlexRay</b>	A new field bus technology finding its way into current automobiles.
<b>GEF</b>	Graphical Editing Framework: A plug-in for Eclipse simplifying the development of graphical editors (see <a href="#">Section 3.2.3</a> ).
<b>GUI</b>	Graphical User Interface: The visual screen elements of a program that the user sees and can manipulate.
<b>I/O</b>	Input/Output.
<b>IDE</b>	Integrated Development Environment: A program integrating the different tasks encountered during development.



<b>IID</b>	Instance ID: The AUTOSAR identification of a module instance.
<b>JNI</b>	Java Native Interface: A Java library to call or be called from machine native code (see <a href="#">Section 3.4.1</a> ).
<b>JVM</b>	Java Virtual Machine: A program implementing an own instruction set based on the Java programming language.
<b>LIN</b>	Local Interconnect Network: An inexpensive field bus technology deployed especially in the automotive comfort sector.
<b>MCAL</b>	Microcontroller Abstraction Layer: The AUTOSAR layer abstracting from the concrete microcontroller type.
<b>MID</b>	Module ID: The AUTOSAR identification of a module type.
<b>MVC</b>	Model–View–Controller: A popular software design pattern (see <a href="#">Section 4.2.1</a> ).
<b>OSEK</b>	“Offene Systeme und deren Schnittstellen fuer die Elektronik im Kraftfahrzeug”, translating into “open systems and the corresponding interfaces for automotive electronics”: An industry standard concentrating on a static embedded operating system among other things.
<b>OSEKtime</b>	A real-time embedded operating system standard.
<b>PORT</b>	The AUTOSAR driver responsible for initializing the I/O pins (see <a href="#">Section 2.2.1</a> ).
<b>RCP</b>	Rich Client Platform: An application allowing dynamic extension through plug-ins (see <a href="#">Section 1.6.2</a> ).
<b>RFC</b>	Request For Comments: A document form common in the Internet society, often resulting in a standard.

<b>RTE</b>	Runtime Environment: The AUTOSAR abstraction layer providing abstract ports for applications to communicate.
<b>SID</b>	Service ID: The AUTOSAR identification of a function pertaining to a specific module.
<b>SWT</b>	Standard Widget Toolkit: A GUI toolkit using operating system native widgets (see <a href="#">Section 3.2.2</a> ).
<b>TCP/IP</b>	Transmission Control Protocol/Internet Protocol: A reliable protocol that most Internet protocols are based on.
<b><i>tresos</i> ECU</b>	Elektrobit Automotive's configuration framework for the embedded automotive area (see <a href="#">Section 1.6.3</a> ).
<b><i>tresos</i> GUI</b>	The application part of <i>tresos</i> ECU enabling the user to configure modules in a graphical manner.
<b>VFB</b>	Virtual Functional Bus: The abstraction of the interconnections between AUTOSAR Software Components.
<b>XML</b>	Extended Markup Language: A generic data format stored in human readable form.