

UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione  
Corso di Laurea Magistrale in Computer Engineering



# Security modeling and automatic code generation in AUTOSAR

Supervisors:

Prof. Cinzia Bernardeschi

Prof. Gianluca Dini

Candidate:

Gabriele Del Vigna

May 2016

*To my family*

## **Abstract**

Nowadays, due to the increasing diffusion of software in automotive, security is becoming increasingly important and should be taken into account from the early stages of software development. The AUTomotive Open System ARchitecture (AUTOSAR) standard, an open industry standard for automotive software architecture, covers many aspects of software modeling and development in automotive, security aspects included.

In this thesis, an extension of security modeling concepts available in AUTOSAR is proposed. The proposed extension gives to the developers the possibility to add security requirements (confidentiality and/or integrity) to a communication links at functional level. They are made available as attributes and can be used to annotate the high level system specification.

Then, we have developed a tool which can be used to automate some steps that the developers have to follow in order to use specific AUTOSAR security services. Our tool automatically add the required security elements in the AUTOSAR XML (ARXML) file (which is the main file format used in AUTOSAR to describe a system). The security elements are added within new software components or within the existing components, based on the specifications provided by the developers. The security requirements are then fulfilled by using the services provided by the AUTOSAR standard.

The tool has been applied to an AUTOSAR use case, namely, the front light management system.

# Contents

<b>List of Figures</b>	<b>3</b>
<b>1 Introduction</b>	<b>5</b>
<b>2 Introduction to the AUTOSAR standard</b>	<b>8</b>
2.1 AUTOSAR architecture . . . . .	9
2.1.1 Application Layer . . . . .	10
2.1.2 RTE layer . . . . .	12
2.1.3 Basic Software Layer . . . . .	12
2.2 AUTOSAR tools . . . . .	13
<b>3 AUTOSAR meta-model</b>	<b>14</b>
3.1 UML concepts . . . . .	14
3.1.1 Class representation . . . . .	14
3.1.2 Generalization . . . . .	15
3.1.3 Association . . . . .	16
3.1.4 Composition . . . . .	16
3.2 Software components . . . . .	17
3.2.1 Ports . . . . .	18
3.2.2 Interfaces . . . . .	19
3.2.3 Internal behavior . . . . .	21
3.3 AUTOSAR services . . . . .	23
<b>4 Safety and security in AUTOSAR</b>	<b>25</b>
4.1 Safety mechanisms . . . . .	25

---

4.1.1	End-to-End protection mechanisms . . . . .	26
4.2	Security mechanisms . . . . .	32
4.2.1	Crypto Service Manager . . . . .	33
<b>5</b>	<b>Rational Rhapsody modeling tool</b>	<b>36</b>
5.1	Model a simple system . . . . .	36
5.2	Internal behavior and runnable entities . . . . .	42
5.2.1	Implicit and explicit data reception and transmission . . . . .	45
5.3	Add End-to-End protection . . . . .	45
5.4	Add security (by using Crypto Service Manager) . . . . .	48
5.5	Generate ARXML . . . . .	54
<b>6</b>	<b>Security level and automatic generation of security elements</b>	<b>55</b>
6.1	Security level specification and elements generation . . . . .	57
6.2	Example usage . . . . .	58
6.3	ARXML and Python . . . . .	61
6.3.1	Python script . . . . .	61
6.4	ARXML code . . . . .	64
<b>7</b>	<b>Application to an AUTOSAR use case</b>	<b>68</b>
7.1	Use case description . . . . .	68
7.2	Application to the use case . . . . .	71
<b>8</b>	<b>Conclusions</b>	<b>74</b>
	<b>Acknowledgments</b>	<b>75</b>
	<b>Acronyms</b>	<b>76</b>
	<b>Bibliography</b>	<b>78</b>

# List of Figures

2.1	AUTOSAR layer overview [1] . . . . .	10
2.2	Graphical representation of software components [2] . . . . .	11
3.1	A class representation in UML . . . . .	15
3.2	Generalization representation in UML . . . . .	16
3.3	Association relation in UML . . . . .	16
3.4	Composition relation in UML . . . . .	17
3.5	Software components meta-model . . . . .	17
3.6	Components, ports and interfaces meta-model . . . . .	18
3.7	Sender-receiver interface meta-model . . . . .	20
3.8	Client-server interface meta-model . . . . .	21
3.9	Internal behavior meta-model . . . . .	22
3.10	Service dependency meta-model . . . . .	23
4.1	Example of faults mitigated by E2E protection . . . . .	27
4.2	Control fields added to data by the E2E protection . . . . .	28
4.3	AUTOSAR meta-model for the E2E protection . . . . .	30
4.4	E2E Protection wrapper . . . . .	31
4.5	AUTOSAR layered view with CSM . . . . .	34
5.1	New project creation in Rhapsody . . . . .	37
5.2	Create a new component in Rhapsody . . . . .	38
5.3	Create a port's interface in Rhapsody . . . . .	38
5.4	Left panel view in Rhapsody . . . . .	39
5.5	Software component prototype in Rhapsody . . . . .	40

## LIST OF FIGURES

---

5.6	Select a prototype type in Rhapsody . . . . .	41
5.7	Software composition model in Rhapsody . . . . .	41
5.8	Software composition model in Rhapsody (final model) . . . . .	42
5.9	Specify the period attribute of the RTE TimingEvent . . . . .	43
5.10	Runnable and RTE events in Rhapsody . . . . .	44
5.11	Specify an E2E protection profile in Rhapsody . . . . .	47
5.12	Data element to which the E2E protection has to be applied . . . . .	48
5.13	Final view for the E2E protection in Rhapsody . . . . .	49
5.14	System overview, before the specification of the CSM . . . . .	50
5.15	Set interface tags for a service interface in Rhapsody . . . . .	51
5.16	Assign a SwcServiceDependency to a port in Rhapsody . . . . .	52
5.17	Left panel view after the specification of service dependency . . . . .	53
6.1	System used as a starting point . . . . .	58
6.2	Specify a security tag . . . . .	60
6.3	Required elements added by means of new components . . . . .	60
6.4	Required elements added to the existing components . . . . .	61
7.1	Front lights system overview . . . . .	69
7.2	Communication focused view of the front light system . . . . .	70
7.3	Initial FLM system . . . . .	71
7.4	Final FLM system . . . . .	72

# Chapter 1

## Introduction

As stated in [3], modern motor vehicles contain an increasing number of computers in the form of Electronic Control Unit (ECU), which control numerous vehicle functions (such as steering, braking, acceleration, lights and so on). ECUs are interconnected by common wired networks: the most common network is the Controller Area Network (CAN) bus. Furthermore, many of these components also have wireless capability: like keyless entry, diagnostic, entertainment systems and so on. This increased connectivity leads to an increasing number of potential cyber security threats.

In [4], Stephen Checkoway et al. demonstrate that remote exploitation is feasible via a broad range of attack vectors (such as CD players, Bluetooth etc.). Many such systems are connected to the CAN bus, either to directly interface with other automotive systems (e.g., to support certain hands-free features, or to display messages on the console) or simply to support a common maintenance path for updating all ECU firmware. Thus, a compromised CD player, for example, can offer an effective vector for attacking other automotive components.

For all these reasons, security in automotive is becoming increasingly important and should be taken into account from the early stages of software development. In this work, we focus on the AUTOSAR standard, which is the de-facto standard in automotive industry. AUTOSAR defines a service called Crypto Service Manager (CSM)[5], which provides a set of security



functionalities (such as Hash function, MAC and so on) that can be used by software components of a system to add the required security functions. The steps that the developers have to follow in order to specify an AUTOSAR security mechanism, are quite complex and error prone. Furthermore, in the early development stages, the developers may need something less specific to indicate that a security mechanism is required. AUTOSAR does not provide any means to specify this kind of high level security requirement.

In this work, we proposed an approach to cover these lacks, which allows the developers to specify a security level (confidentiality and/or integrity) for system's communication links. Then we have developed a tool which automatically add security elements to the system, by following the specification assigned to the system by the developers.

This work was conducted as part of the European project SAFURE[6] (Safety And Security By Design For Interconnected Mixed-Critical Cyber-Physical Systems). The project targets the design of cyber-physical systems by implementing a methodology that ensures safety and security "by construction". The goals of the SAFURE project are the following:

- to implement a holistic approach to safety and security of embedded dependable systems, preventing and detecting potential attacks;
- to empower designers and developers with analysis methods, development tools and execution capabilities that jointly consider security and safety;
- to set the ground for the development of SAFURE-compliant mixed-critical embedded products.

Chapter 2 provides an overview of the AUTOSAR standard and briefly describes its architecture. Chapter 3 describes the main aspects of the AUTOSAR meta-model. Chapter 4 describes the safety and security mechanisms provided by AUTOSAR. Chapter 5 describes how to model an AUTOSAR compliant system by using the IBM Rational Rhapsody tool. Chapter 6 describes the tool we have developed, which can be used to automate some steps during the modeling phase of an AUTOSAR system. Chapter 7

shows the application of the proposed approach and the developed tool to an AUTOSAR use case. Chapter 8 recaps what we have done and briefly describes what can be subject of future work.

## Chapter 2

# Introduction to the AUTOSAR standard

In a modern car there are 70 or more ECUs[7] which interact with each other via a complex wired network. ECUs are made by different manufacturers and each ECU has a software embedded in it, which provides a specific functionality. For these reasons, software of an ECU could not work on a different manufacturers' ECU; this means that software is not portable. The main problems can be summaries as follow:

- increasing number of ECUs
- increasing number and complexity of the embedded software
- software is not portable

As a consequence, in the process of cars' software development, the complexity of software management and the final cost increases, as well as time to market. To deal with all these problems AUTomotive Open System ARchitecture (AUTOSAR) was founded.

AUTOSAR standard [8] is an open industry standard for automotive software architecture, founded in 2003 and developed by a partnership of different automotive Original Equipment Manufacturers (OEMs), suppliers and tool vendors. Some AUTOSAR partners[9] are BMW, Bosch, Ford, General

Motor, Toyota and others. The AUTOSAR has been formed with the goals of create an open and standardized software architecture for automotive ECUs; it provides a set of specifications that describes the software architecture, application interfaces and a methodology for the development process. The main AUTOSAR goals can be summarized as follow:

- fulfillment of future vehicle requirements, such as, availability and safety, software upgrades/updates and maintainability;
- increased scalability and flexibility to integrate and transfer functions;
- higher penetration of "Commercial off the Shelf" software and hardware components across product lines;
- improved containment of product and process complexity and risk;
- cost optimization of scalable systems;

### **2.1 AUTOSAR architecture**

AUTOSAR uses a three-layered architecture (shown in figure 2.1):

- Application layer
- Runtime Environment (RTE) layer
- Basic Software (BSW) layer

The application layer contains the Software Components (SWCs): piece of software which provides specific functionality. RTE layer is the middleware layer, and it provides a communication abstraction for software components. BSW provides basic services and basic software modules to software components.

An AUTOSAR software component cannot directly access BSW modules [10]; software components can communicate between each other and with BSWs only through the RTE. Due to RTE abstraction layer, software components can be developed independently of underlying hardware, which

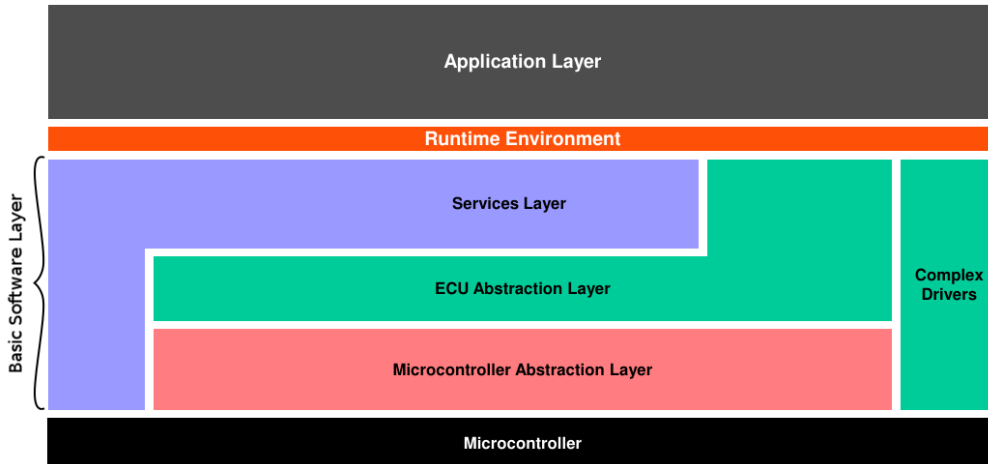


Figure 2.1: AUTOSAR layer overview [1]

means that they have the transferability and reusability property. In other words, software which provides a specific function, can be reused in vehicles which have ECUs made by different tool vendors.

The following subsections, provides a brief description of the three layers of the AUTOSAR standard.

### 2.1.1 Application Layer

The application layer is the highest layer of the AUTOSAR architecture. It contains all the software components (application, sensor and actuator software components) which provides specific functionality. Due to the RTE abstraction layer, these software components can be implemented independently of the underlying hardware. Software components encapsulate the implementation of their functionality and behavior and they expose well-defined connection points, called *PortPrototypes*, to the outside world. The graphical appearance of a software component is shown in Figure 2.2. SWCs may only interact by means of their *PortPrototypes*.

The main communication paradigms between software components are client/server, for operation-based communication, and sender/receiver, for data-based communication:

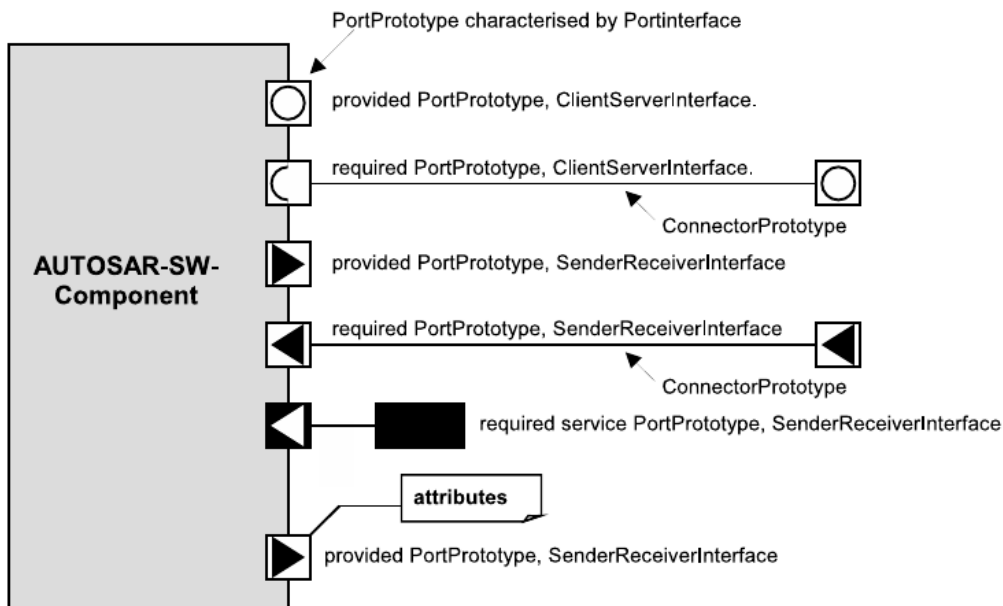


Figure 2.2: Graphical representation of software components [2]

- Sender-receiver communication involves the transmission and reception of signals consisting of atomic data elements that are sent by one component and received by one or more components. A sender-receiver interface can contain multiple data elements. Sender-receiver communication is one-way: any reply sent by the receiver is sent as a separate sender-receiver communication.
- Client-server communication involves, the client which is the requirer (or user) of a service and the server that provides the service. The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server, in the form of the RTE, waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client's request. The invocation of a server is performed by the RTE itself when a request is made by a client. The invocation occurs synchronously with respect to the RTE (typically via a function call) however the client's invocation can be either synchronous (wait for server to complete) or asynchronous with respect to the server.

### 2.1.2 RTE layer

The RTE [10] provides the infrastructure services that enable communication to occur between AUTOSAR software components as well as acting as the means by which software components access basic software modules including the Operating System (OS) and communication service. As already stated, the RTE make software components independent from the mapping to a specific ECU.

### 2.1.3 Basic Software Layer

The BSW layer is divided in three sub-layer (as shown in Figure 2.1):

- The Services Layer is the highest BSW layer. It provides basic services for applications, RTE and BSW modules. The Services Layer offers:
  - operating system functionality;
  - vehicle network communication and management services;
  - memory services (Non-Volatile Random Access Memory (NVRAM) management);
  - diagnostic services (memory errors, fault treatment and so on);
  - ECU state management, mode management;
  - logical and temporal program flow monitoring (Watchdog Manager)
  - cryptographic services (Crypto Service Manager)
- The ECU Abstraction Layer interfaces the drivers of the Microcontroller Abstraction Layer. It also contains drivers for external devices. It offers an Application Programming Interface (API) for access to peripherals and devices regardless of their location (internal/external micro-controller) and their connection to the micro-controller (port pins, type of interface). The ECU Abstraction Layer make higher software layers independent of ECU hardware layout.

- The Microcontroller Abstraction Layer is the lowest software layer of the BSW layer. It contains internal drivers, which are software modules with direct access to the micro-controller and internal peripherals. The Microcontroller Abstraction Layer makes higher software layers independent of micro-controller.
- The Complex Drivers Layer spans from the hardware to the RTE. It provides the possibility to integrate special purpose functionality, e.g. drivers for devices:
  - which are not specified within AUTOSAR;
  - with very high timing constrains;
  - for migration purposes;
  - ...

## 2.2 AUTOSAR tools

AUTOSAR tool refers to all tools that support the tasks of creation, modification and interpretation of AUTOSAR models. In the development process of an AUTOSAR compliant system, many tools (coming from different vendors) may be involved. The data exchanged between these different tools need to agree on a common understanding about the wording and the semantics. AUTOSAR formally defines the structure and semantics of data by means of Unified Modeling Language (UML) class diagrams. In addition, AUTOSAR has chosen eXtensible Markup Language (XML) as a language for exchange of data between different AUTOSAR tools [11]. Therefore, an AUTOSAR system can be described in AUTOSAR XML (ARXML).



# Chapter 3

## AUTOSAR meta-model

The AUTOSAR meta-model is an UML representation of the AUTOSAR templates. UML class diagrams are used to describe the attributes and the operations of each class, and the interrelationships between the various classes. In the following sections, the main UML diagrams of the AUTOSAR meta-model are shown. In section 3.1 an overview of the main UML concepts are provided. Section 3.2 describes the software components meta-model (and other related concepts like ports, interface, etc.). Section 3.3 provides a description of the AUTOSAR service meta-model.

### 3.1 UML concepts

UML is a modeling language used in software development, that is intended to provide a standard way to visualize the design of a system. UML has many types of diagrams (class diagram, components diagram, sequence diagram, etc.), but we focus on the class diagram, which is the main type of diagram used within the AUTOSAR standard.

#### 3.1.1 Class representation

In UML a class is represented as shown in Figure 3.1. In the figure, *Class1* is the name of the class. *attribute1* and *attribute2* are the attributes of the class; for every attributes, in addition to the attribute's name, it is possible

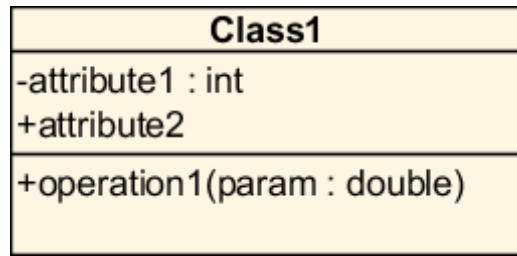


Figure 3.1: A class representation in UML

to specify a variety of information, such as the type (int, double, etc.), the multiplicity, a default value, visibility and so on. *Operation1* is a function provided by the class; for every operation it is possible to specify a set of parameters.

The visibility is specified in the same way for attributes and operations. The used symbols and their meaning are shown in Table 3.1.

Visibility	Description
+	public
-	private
#	protected
~	package

Table 3.1: Attributes visibility in UML

### 3.1.2 Generalization

Generalization is a relation between two classes and it is graphically represented as shown in Figure 3.2: the class *Animal* is a generalization of the class *Owl* and *Cobra* and it contains characteristics that are common to both classes; every animal has a weight, an average life span and a family. *Owl* and *Cobra* are specializations of the class *Animal*: they have some characteristics that not all animals have (i.e. not all animals have wings).

Note that in the figure, the name of the class *Animal* is written in italic, because it is an abstract class. Abstract classes cannot be instantiated (i.e. an animal must belong to some species, it cannot be just an "animal").

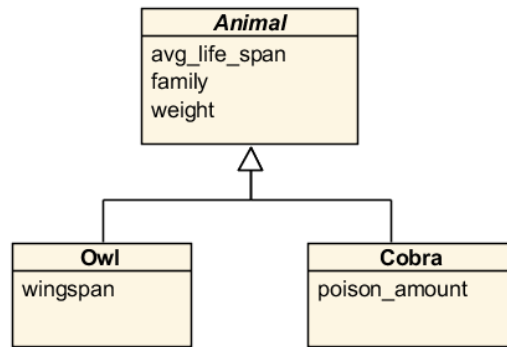


Figure 3.2: Generalization representation in UML

### 3.1.3 Association

An association is a link between two or more classes and it is graphically represented with a line (if it is a bi-directional association) or with an arrow (if it is a uni-directional association). In AUTOSAR, the uni-directional associations (as the one shown in Figure 3.3) are more common than the bi-directional associations. The association shown in Figure 3.3, means that

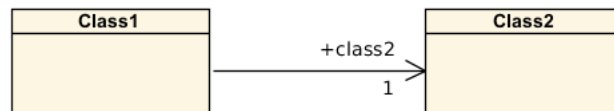


Figure 3.3: Association relation in UML

the class *Class1* has exactly one attribute named *class2*, which is a reference to *Class2*. In practice this can be seen as a sort of pointer: every object of type *Class1*, has an attribute *class2* which point to an object of type *Class2*.

### 3.1.4 Composition

Composition is used to represent a containment relation and it is graphically represented as shown in Figure 3.4. Figure 3.4 can be read as follow: *Car* contains exactly one *engine*. *engine* is the name of the attribute within the class *Car*, and it is of type *Engine*.

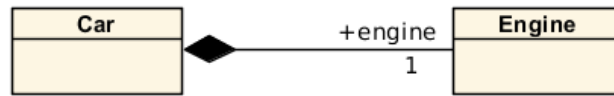


Figure 3.4: Composition relation in UML

## 3.2 Software components

Software Component (SWC) are one of the most important architectural elements of the AUTOSAR meta-model. They represent piece of software which provide specific functionalities and they provide and/or require interfaces and are connected to each other to fulfill architectural responsibilities[12].

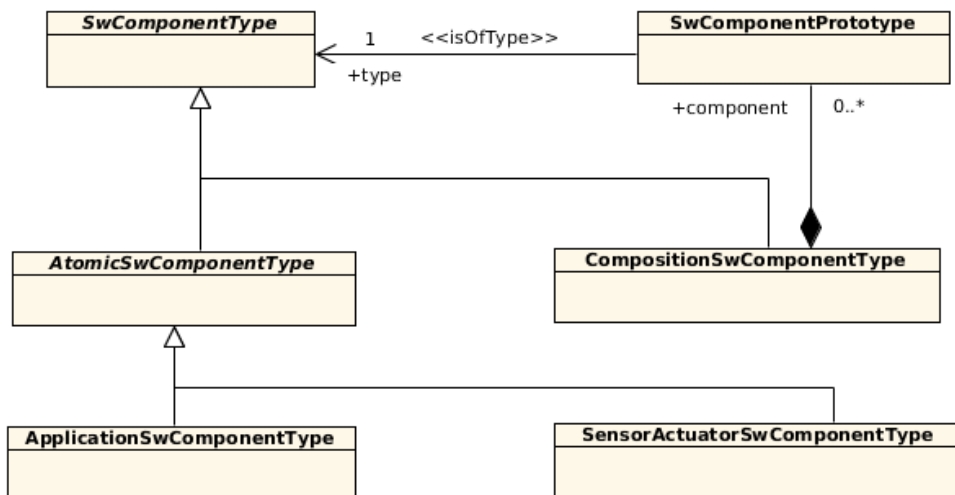


Figure 3.5: Software components meta-model

Figure 3.5 shows the meta-model for SWCs:

- *ApplicationSwComponentType* and *SensorActuatorSwComponentType* represent, respectively, the application software and the software running on sensor/actuator (there are other types of *SwComponentType* which are not shown in figure, because they are not interesting for our purposes);

- *CompositionSwComponentType* is used to aggregate two or more *SwComponentType* (everyone of which is typed by a *SwComponentType*). This can be useful to represents a system composed by two or more software components and to see how they are connected between each other;
- *SwComponentPrototype* represents an instance of a *SwComponentType* (*ApplicationSwComponentType*, *SensorActuatorSwComponentType* or other not shown in figure) within a *CompositionSwComponentType*. Every *SwComponentPrototype* is typed by one *SwComponentType*.

### 3.2.1 Ports

Software components can communicate between each other, only by means of their *PortPrototype*. Every port is typed by one interface. Two ports can be connected only if they are typed by the same interface.

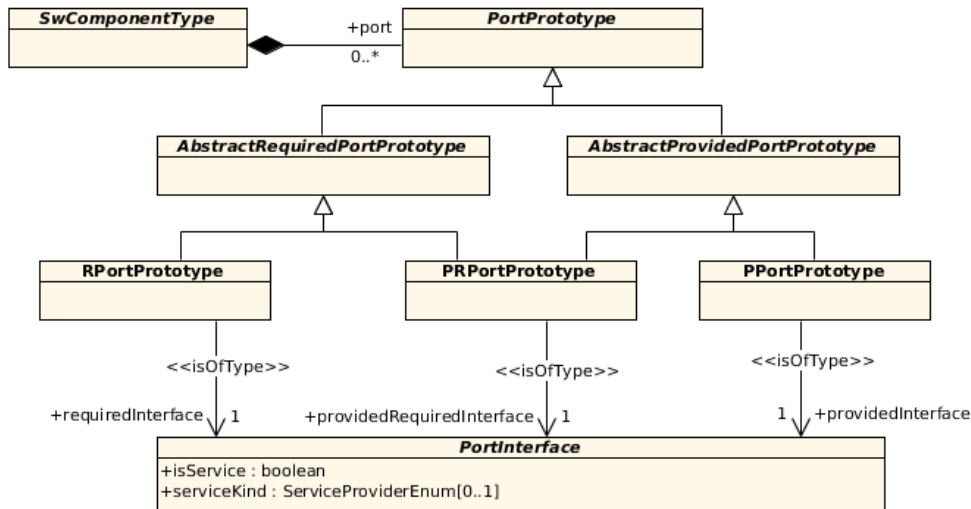


Figure 3.6: Components, ports and interfaces meta-model

Figure 3.6 shows the meta-model for ports and their relations with software components:

- every *SwComponentType* (*ApplicationSwComponentType*, *SensorActuatorSwComponentType*, etc.) can have 0 or more ports;

- a port can be:
  - a provide-port (*PortPrototype*): a port which provides services or data;
  - a require-port (*RPortPrototype*): a port which requires services or data;
  - a provide-require-port (*PRPortPrototype*): a port which can provide and requires services or data;
- as already stated, every port can be typed by one and only one interface.

### 3.2.2 Interfaces

A *PortInterface* is used to specify what kind of information is exchanged between two *PortPrototypes*. *PortInterface* represents also a "compatibility" between two ports, because the communication between two ports is possible only if they have the same interface. In this section we focus on two kinds of interfaces:

- sender-receiver interface;
- client-server interface.

#### Sender-receiver interface

Sender-receiver interface (*SenderReceiverInterface* in the AUTOSAR meta-model) can be used for the specification of the typically asynchronous communication pattern, where a sender provides data that are required by one or more receivers. A *PortPrototype* typed by a *SenderReceiverInterface* may be connected to establish a 1:n (i.e. one sender, multiple receivers) communication relationship. *SenderReceiverInterface* is also used to specify the data elements sent and received over the ports which are typed by that interface.

The meta-model for *SenderReceiverInterface* is shown in Figure 3.7:

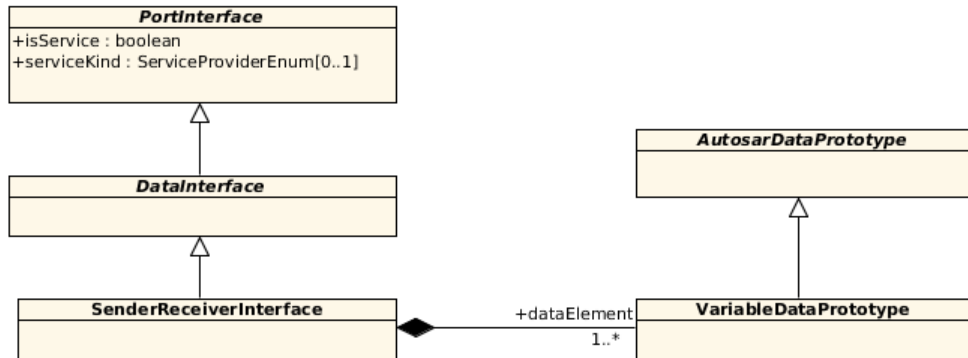


Figure 3.7: Sender-receiver interface meta-model

- *PortInterface* is the base class from which *SenderReceiverInterface* derives. The public attributes *isService* and *serviceKind* are used to specify if the interface is related to an AUTOSAR service or not (this aspect is better explained in section 3.3);
- *SenderReceiverInterface* is the class used to specify that the communication is of type sender-receiver. This class contains one or more *dataElement* of type *VariableDataPrototype*, which represents the piece of information transmitted among *PortPrototypes* typed by a *SenderReceiverInterface*.

### Client-server interface

A client-server interface (*ClientServerInterface* in the AUTOSAR meta-model) is used to define a client-server communication, where a client may initiate the execution of an operation by a server which supports that operation. The server executes the operation and, when completed, it provides the client with the result (synchronous operation call) or else the client checks for the completion of the operation by itself (asynchronous operation call). A client shall not be connected to multiple servers.

The meta-model for *ClientServerInterface* is shown in Figure 3.8:

- a *ClientServerInterface* defines a collection of client-server operations (*ClientServerOperation* in figure);

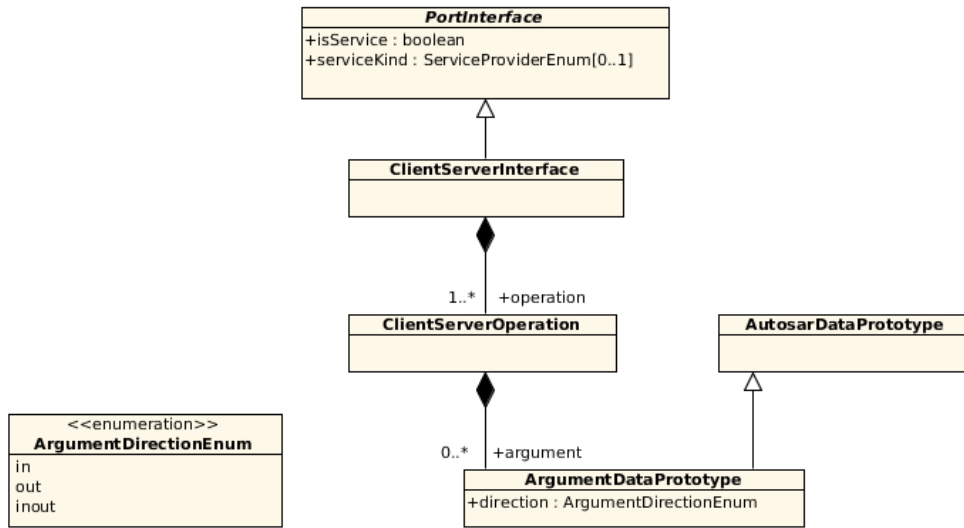


Figure 3.8: Client-server interface meta-model

- a *ClientServerOperation* consists of  $0..*$  *ArgumentDataPrototype* (the parameters for the operation). Every parameter has a direction, which can be:
  - *in*: the parameter is passed to the operation;
  - *inout*: the parameter is passed to, and returned from the operation;
  - *out*: the parameter is returned from the operation.

### 3.2.3 Internal behavior

Internal behavior (*SwcInternalBehavior* in the AUTOSAR meta-model) provides means for formally defining the behavior of a software component. In other words, it describes the relevant aspects of the software-component with respect to the Runtime Environment (RTE), i.e. the runnable entities (which are the the smallest code-fragments that are provided by a software component) and the RTE events they respond to. The meta-model for internal behavior is shown in Figure 3.9:

- *SwcInternalBehavior* can contain one or more *RunnableEntity*;



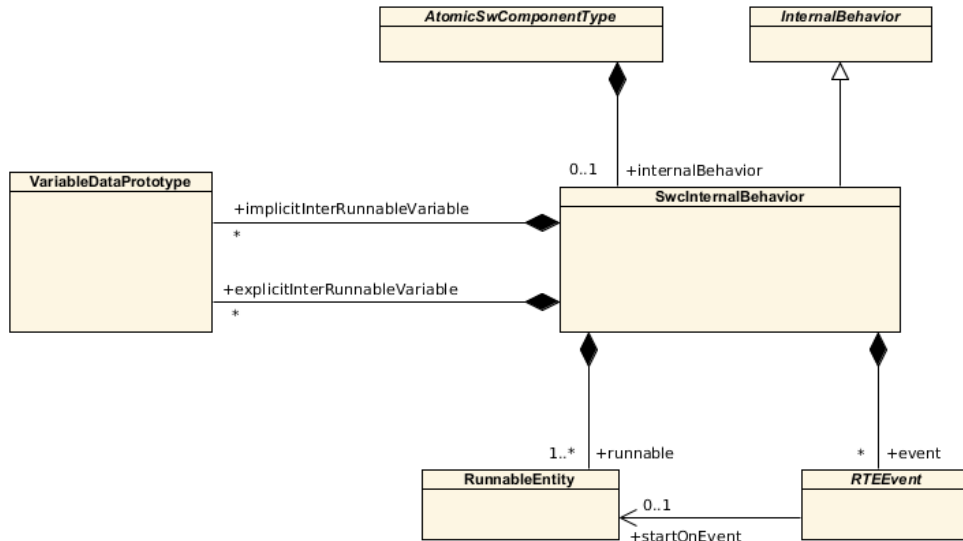


Figure 3.9: Internal behavior meta-model

- *implicitInterRunnableVariables* and *explicitInterRunnableVariables* are variables which can be written and read by runnable entities belonging to the same software component; the difference between the two is that:
  - *implicitInterRunnableVariable* is used to avoid concurrent access to a variable, by creating copies of it (one for every runnable entity that want to access the variable);
  - *explicitInterRunnableVariable* is used to block potential concurrent accesses to a variable; it is used to get data consistency;
- *RTEEvents* are used to trigger the execution of runnable entities; it is important to note (as stated in Section 2.2.4 of [10]) that all activities within an AUTOSAR application is initiated by the triggering of runnable entities by the RTE as a result of RTE events. Relation between a runnable entity and an *RTEEvent* is specified by means of *startOnEvent* reference (as shown in figure); this means that when an *RTEEvent* occurs, it is the responsibility of the RTE to trigger the execution of the corresponding runnable entity.

One of the most common RTE events is the *TimingEvent*: a periodic

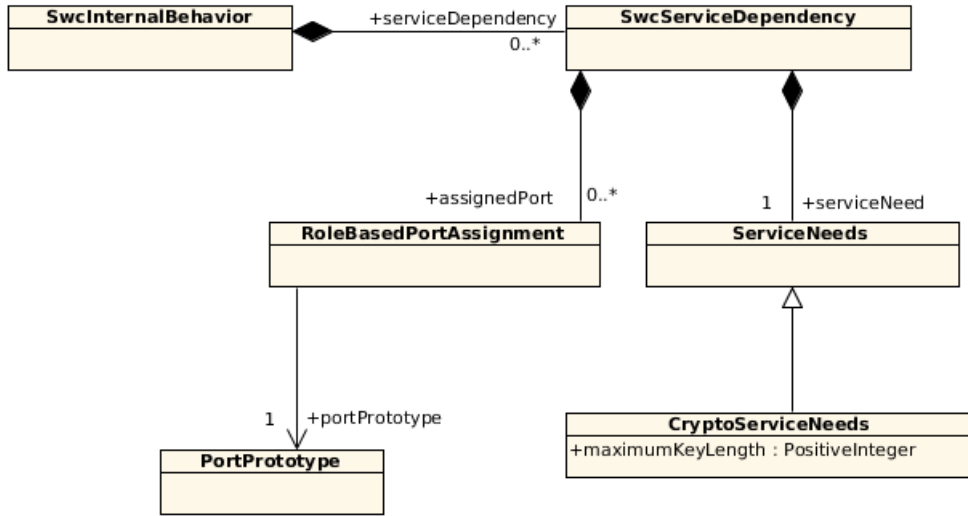


Figure 3.10: Service dependency meta-model

event whose period can be specified (in seconds) in the *period* attribute of the *TimingEvent* class. The complete list of events is written in section 4.2.2.4 of [10].

### 3.3 AUTOSAR services

AUTOSAR Services can be seen as an hybrid concept between Basic Software Modules and Software components. Software components that requires AUTOSAR services use standardized AUTOSAR interfaces to communicate with them.

The dependency of a software component from an AUTOSAR service is modeled by adding provided or required ports (hereinafter referred to as 'service ports') to the software component. The interface for these ports needs to be one of the standardized service interfaces defined in the AUTOSAR documentation, and the attribute *isService* of the interface must be set to *true*. Furthermore, the internal behavior of the software component shall contain a *SwcServiceDependency*, which is used to add more information about the required service.

The meta-model of the *SwcServiceDependency* is shown in Figure 3.10:

- *SwcServiceDepencency* is used to associate ports, port groups and (in special cases) data defined for a software component to a given *ServiceNeeds* element;
- *ServiceNeeds* are used to provide detailed information about what a software component expects from the AUTOSAR service. For instance, *CryptoServiceNeeds* can be used to specify a maximum length for the key used in cryptographic services;
- *RoleBasedPortAssignment* is used to specify an assignment of a role to a particular service port of a software component. With this assignment, the role of the service port can be mapped to a specific *ServiceNeeds* element. The attribute *portPrototype* of this class, is used to refer a service port of the software component.

There are many other classes derived from *ServiceNeeds*, but they are not shown in figure 3.10 to avoid unnecessary complexity; the other classes can be seen in Figure 7.36 of [2].

# Chapter 4

## Safety and security in AUTOSAR

The AUTOSAR standard provides a number of mechanisms which can be used by the software developers to build safe and secure software. In the following sections an overview of these mechanisms is provided (with a special focus on the End-to-End (E2E) protection mechanisms and the Crypto Service Manager (CSM), which are more for our purposes).

### 4.1 Safety mechanisms

AUTOSAR supports the development of safety-related systems by offering safety measures and mechanisms. Those mechanisms assist with the prevention, detection and mitigation of hardware and software faults to ensure freedom from interference between software components. However AUTOSAR is not a complete safe solution. The use of AUTOSAR does not imply ISO26262[13] compliance (ISO26262 is an international standard for functional safety of automotive equipment). It is still possible to build unsafe systems using the AUTOSAR safety measures and mechanisms. This should be taken into account by the software developers.

Safety mechanisms provided by AUTOSAR are the following[14]:

- Memory partitioning: it provides protection by means of restricting

access to memory. Memory partitioning means that OS-Applications reside in different memory areas (partitions) that are protected from each other. In particular, code executing in one partition cannot modify memory of a different partition;

- Timing monitoring: timing protection and monitoring can be described as monitoring of the following properties: monitoring that tasks are dispatched at the specified time, meet their execution time budgets, and do not monopolize OS resources. To guarantee that safety-related functions will respect their timing constraints, tasks monopolizing the Central Processing Unit (CPU) (such as heavy CPU load, many interrupt requests) shall be detected and handled;
- Logical supervision: it is a technique for checking the correct execution of software and focuses on control flow errors. Control flow errors cause a divergence from the valid program sequence during the error-free execution of the application. An incorrect control flow occurs if one or more program instructions are processed either in the incorrect sequence or are not even processed at all;
- E2E protection: in a distributed system, the exchange of data between a sender and the receiver(s) can affect functional safety (if safety depends on the integrity of such data). Therefore, such data shall be transmitted using mechanisms to protect it against the effects of faults within the communication link, and this can be done by means of E2E protection mechanisms.

#### 4.1.1 End-to-End protection mechanisms

The concept of E2E protection[15] assumes that safety-related data exchange shall be protected at runtime against the effects of faults within the communication link. Examples for such faults (as shown in Figure 4.1) are random hardware faults, interference, and systematic faults within the software. By using E2E communication protection mechanisms, the faults in the communication link can be detected and handled at runtime. The algorithms

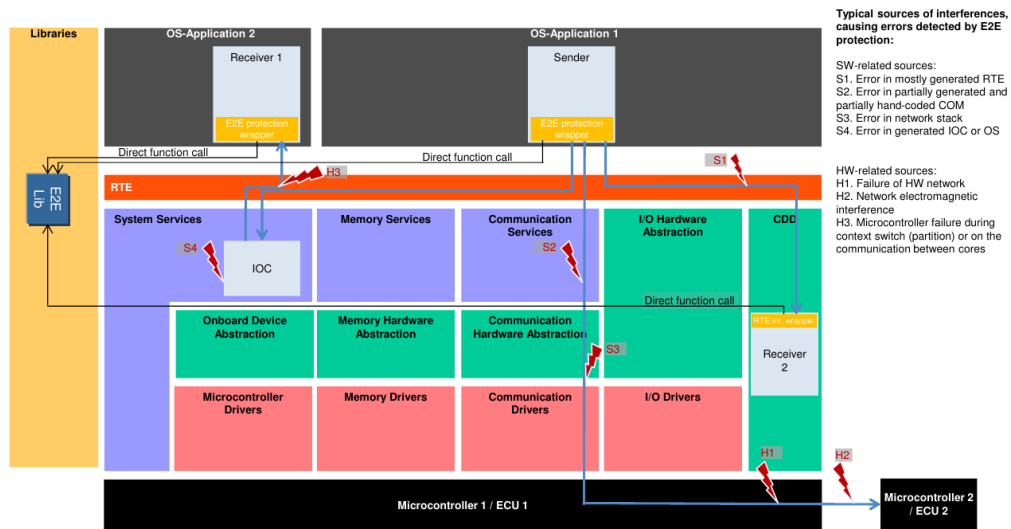


Figure 4.1: Example of faults mitigated by E2E protection

of protection mechanisms are implemented in the E2E Library. The E2E protection has the following characteristics:

1. it can be used to protect safety-related data elements to be sent over the Runtime Environment (RTE) by attaching control data;
2. it can be used to verify safety-related data elements received from the RTE using those control data;
3. it indicates that received safety-related data elements are faulty, which then has to be handled by the receiver software component.

To provide the appropriate solution addressing flexibility and standardization, AUTOSAR specifies a set of E2E profiles that implement an appropriate combination of E2E protection mechanisms.

It is important to note that the E2E protection is for data elements and a data element (and the corresponding signal group) is either completely E2E-protected, or it is not protected. It is not possible to protect only a part of it.

An appropriate usage of the E2E library alone is not sufficient to achieve a safe E2E communication according to ASIL D requirements (Automotive

Safety Integrity Level (ASIL) is a risk classification scheme used in automotive industry and defined by the ISO 26262 standard[13]; the possible ASIL values are A, B, C and D, where D is the highest). Solely the user is responsible to demonstrate that the selected profile provides sufficient error detection capabilities for the considered network.

E2E protection works as follows:

- on sender side: it adds control fields like Cyclic Redundancy Check (CRC) or counter to the transmitted data (as shown in Figure 4.2);
- on receiver side: it evaluates the control fields of the received data (e.g. it computes the CRC on the received data and then it compares the computed CRC with the received CRC in the control field).

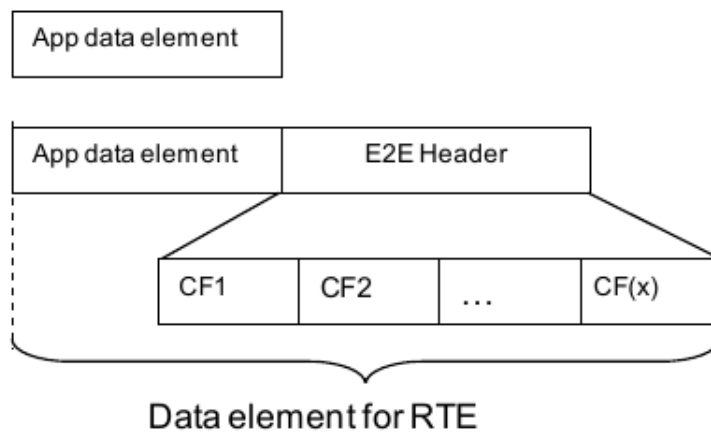


Figure 4.2: Control fields added to data by the E2E protection

### Overview of E2E Profiles

The E2E profiles provide a consistent set of data protection mechanisms, designed to protect against the faults considered in the fault model. Each E2E profile provides an alternative way to protect the data, by means of different algorithms and it defines a subset of the following protection mechanisms:

1. A CRC;

2. A Sequence Counter incremented at every transmission request; the value is checked at receiver side for correct incrementation;
3. An Alive Counter incremented at every transmission request; the value is checked at the receiver side if it changes at all, but correct incrementation is not checked;
4. A specific ID for every port's data element sent over a port or a specific ID for every Interaction layer Protocol Data Unit (I-PDU) group;
5. Timeout detection:
  - Receiver communication timeout;
  - Sender acknowledgement timeout.

The E2E profiles can be used for both inter and intra Electronic Control Unit (ECU) communication. The E2E profiles are optimized for communication over the following buses: CAN, FlexRay and can be used also for Local Interconnect Network (LIN). Depending on the system, the user selects which E2E profile is to be used from those provided by E2E library.

### **E2E meta-model**

To avoid unnecessary complexity, some attributes and stereotypes are not shown in the meta-model of Figure 4.3. The complete meta-model can be found in section 4.7 of [2].

In Figure 4.3:

- *EndToEndProtectionSet* can contain multiple *EndToEndProtection*;
- *EndToEndProtection* is the basic class used to specify the desired characteristics for one end-to-end protection and it contains:
  - one *EndToEndDescription*, which is used to specify an end-to-end profile by means of its attribute *category*;



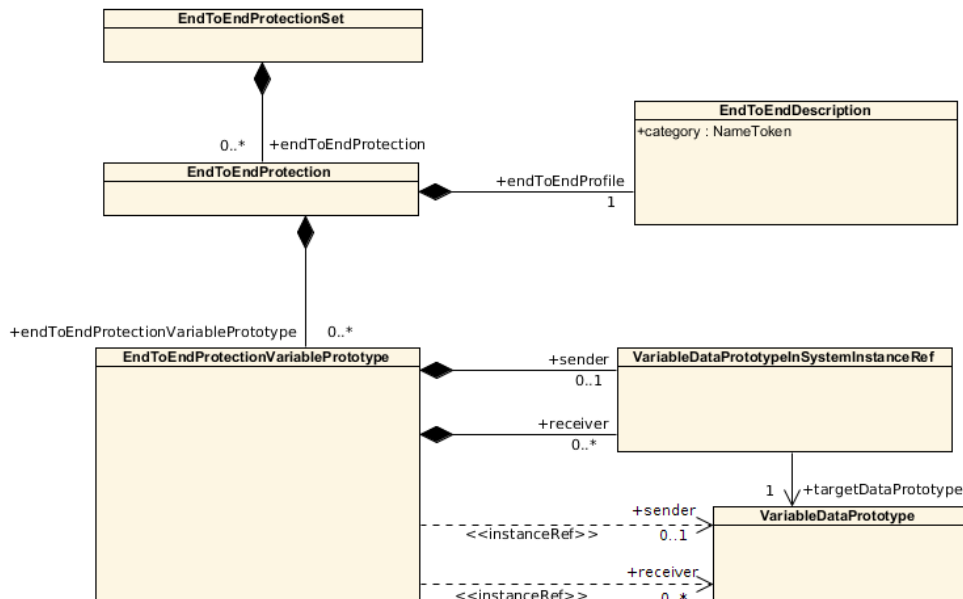


Figure 4.3: AUTOSAR meta-model for the E2E protection

- zero or more *EndToEndProtectionVariablePrototype*, which is used to specify which data has to be protected and if the protection has to be applied by the sender (it adds the additional control fields to the data element) and/or by the receivers (they check the received content, e.g. CRCs comparison).

The attribute *category* of the *EndToEndDescription* class can have one of the following values:

- NONE
- PROFILE\_01
- PROFILE\_02

These values are standardized, however, if needed, it is possible to define non-standardized values provided that they do not create name clashes with future standardized values. This can be achieved by using, for example, a company-specific prefix or suffix to the value of category.

### Usage of E2E protection (E2E protection wrapper)

One possible usage of the E2E library is by means of an E2E wrapper (section 12.1 of [15]). In this approach, every safety-related software component has its own additional sub-layer (which is a .h/.c file pair) called E2E Protection Wrapper, which is responsible for correct invocation of E2E library and the RTE. The functions provided by the E2E Protection Wrapper, act as a wrapper over the write and read functions, and they are provided to software components.

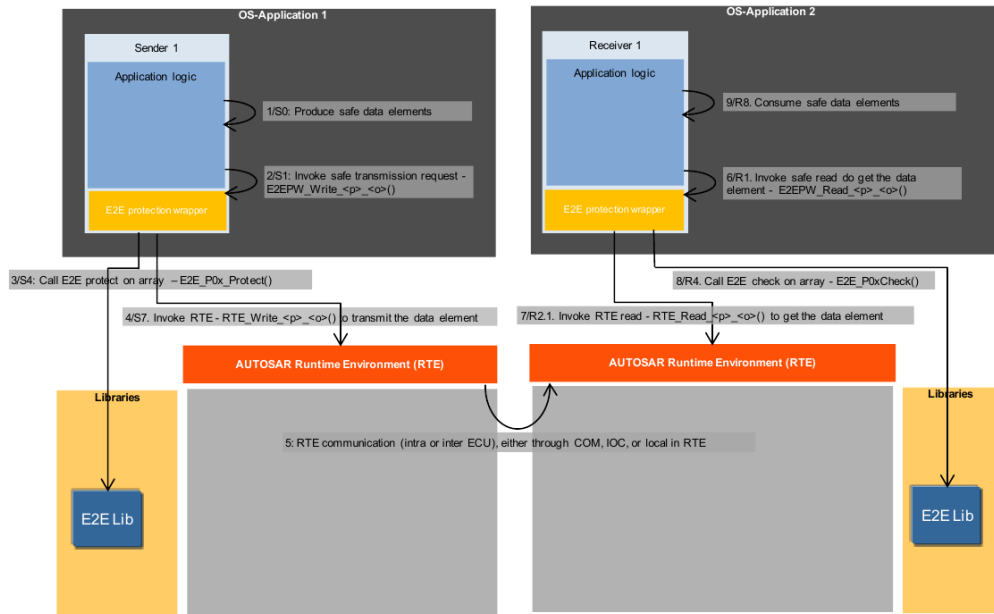


Figure 4.4: E2E Protection wrapper

The overall flow of usage of E2E library and E2E Protection Wrapper from software components is shown in Figure 4.4:

1. the sender application produces safety critical data;
2. the sender application invokes a function provided by the E2E protection wrapper;
3. the E2E protection wrapper invokes the protection routine provided by

the E2E library, to protect the data received from the sender application. These routines are specified in section 8.3 of [15];

4. Once that data has been updated with the needed information (CRC, etc.), the E2E protection wrapper invokes the write function of the RTE to transmit the data;
5. RTE deals with data transmission (intra or inter ECU);
6. on the receiver side, when the RTE receives the data, it wakes up the receiver, which invokes the function provided by the E2E protection wrapper to read the data;
7. the E2E protection wrapper invokes the read function of the RTE to get the received data;
8. the E2E protection wrapper invokes the check routine provided by the the E2E library to check if the data has been received correctly or if it is corrupted;
9. the receiver application consumes the data;

Without the E2E protection wrapper, the correct usage of the E2E library would fall on the developers of the sender/receiver application.

## 4.2 Security mechanisms

AUTOSAR provides different methodologies which can be used by the developers to develop secure software. The main methodologies are the following:

- Secure On-board Communication (SecOC) [16]: the purpose of the SecOC module is to provide an AUTOSAR Basic Software (BSW) to transmit secured data between two or more peers exchanging information over an automotive embedded network;
- Crypto Abstraction Library (CAL) [17]: the AUTOSAR library CAL provides other BSW modules and application software components

with cryptographic functionalities. As the CAL is a library, it is not related to a special layer of the AUTOSAR Layered Software Architecture. CAL has been introduced for using cryptographic functionalities directly by bypassing the RTE [18].

- CSM: CSM is an AUTOSAR service which provides cryptographic functionalities to other software modules, based on a software library or based on an hardware module.

The AUTOSAR CSM and CAL specifications define the same cryptographic functionalities, which cover the following areas[18]:

- Hash calculation
- Generation and verification of message authentication codes
- Random number generation
- Encryption and decryption using symmetrical algorithms
- Encryption and decryption using asymmetrical algorithms
- Generation and verification of digital signature
- Key management operations

The existence of both CSM and CAL, which provide the same (or similar) functionalities is for historical reasons, and as already stated, CAL is a library whereas CSM is a service.

### 4.2.1 Crypto Service Manager

As already stated, the CSM[5] is an AUTOSAR service, and so it is part of the AUTOSAR service layer, as shown in Figure 4.5 (the AUTOSAR service meta-model is described in Section 3.3). The CSM provides an abstraction layer, which offers a standardized interface to higher software layers to access to cryptographic functionalities. Different software modules can require

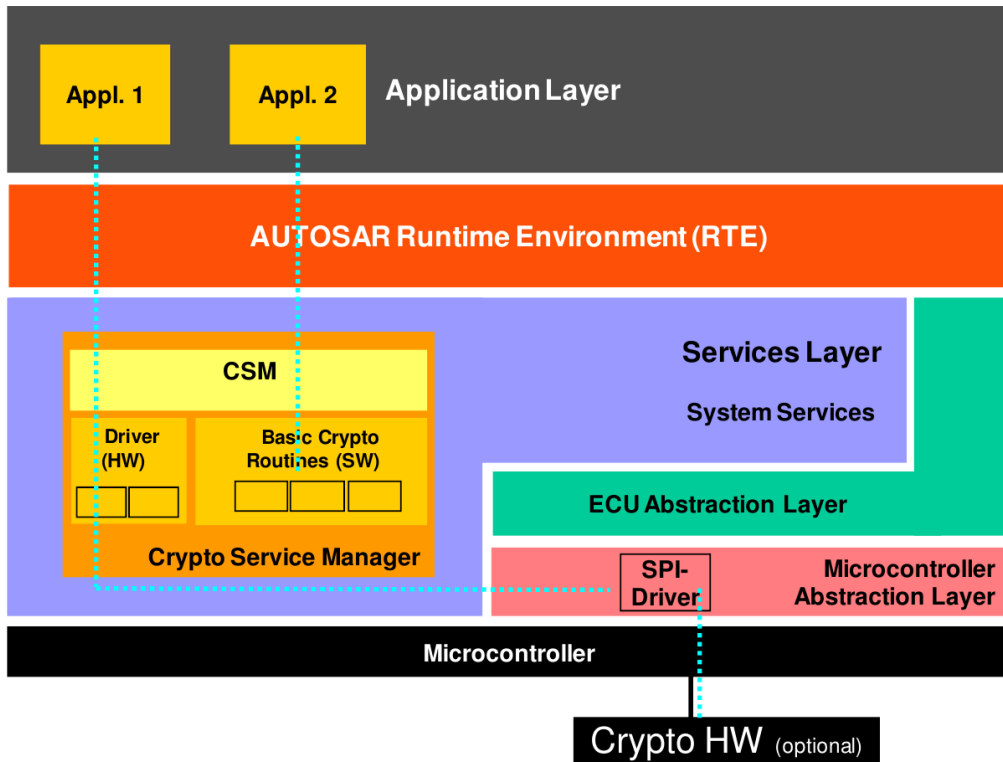


Figure 4.5: AUTOSAR layered view with CSM

different security functionalities. For this reason the CSM service can be configured individually by each software module. This configuration comprises as well the selection of synchronous or asynchronous processing of the CSM services. It also controls the concurrent, multiple and synchronous/asynchronous access of one or multiple clients to one or more services (i.e. it performs buffering, queuing, arbitration, multiplexing).

The services offered by the CSM can be used locally only[18]: it is not possible to access to those services directly from a different ECU. If this is needed, it is up to the provider to specify, implement and provide some proxy for access to CSM. If the CSM is used remotely (via a proxy), it must be taken into account that this raises security implications: any communication between ECUs is done via not protected communication buses (e.g. CAN). This means, that unencrypted data, not yet signed data, would be transmitted and might become stolen or manipulated.

CSM services use cryptographic algorithms that are implemented using a software library or cryptographic hardware modules - both are out of scope and not specified by AUTOSAR.

Note that there is no user management in place, which prevents non-authorized access to any of CSM's services. This means, that if any access protection is needed, it must be implemented by the application; access protection is not target of the CSM.

# Chapter 5

## Rational Rhapsody modeling tool

This chapter provides an overview of IBM Rational Rhapsody (hereinafter referred to as Rhapsody), which is a modeling tool developed by IBM company which can be used to model an AUTOSAR system. It is not free but a 30 days trial is available for download on [19].

Section 5.1 describes the main modeling concept. Section 5.2 describes how to specify the internal behavior, the runnable entities and the RTE events of a software component. Section 5.3 and 5.4 focus respectively on the E2E and CSM modeling within a given system. Section 5.5 describes how to generate the ARXML code of the modeled system.

### 5.1 Model a simple system

This section describes how to create a simple system, based on AUTOSAR, in Rhapsody. The described system is composed by two application software components (one sender and one receiver) that exchange data between each other by means of sender/receiver communication paradigm provided by AUTOSAR.

To create this system in Rhapsody the sequence of steps to follow are the following:

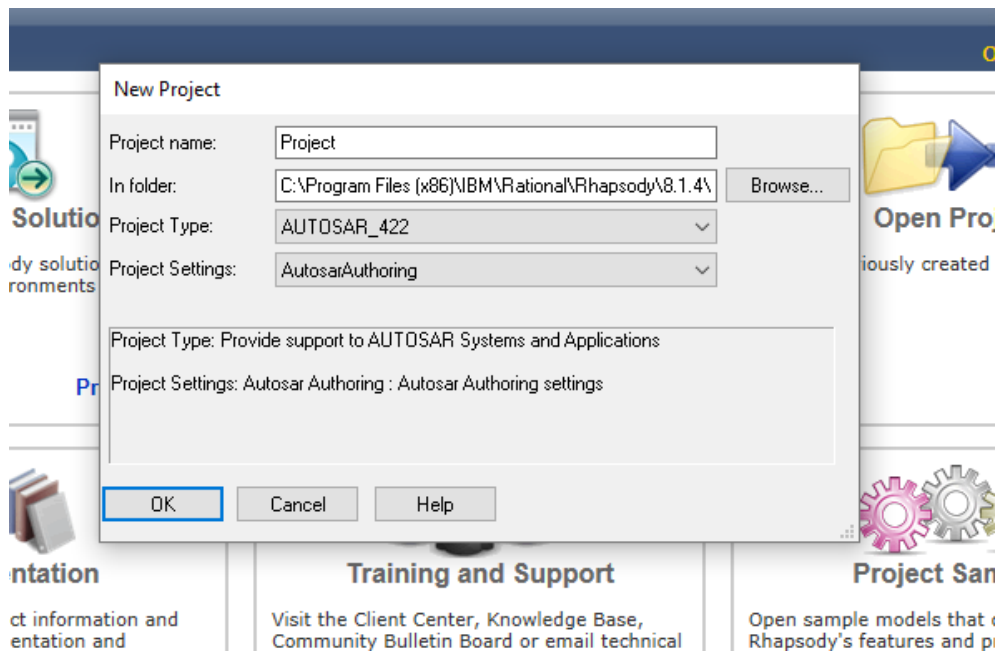


Figure 5.1: New project creation in Rhapsody

1. In the starting window of Rhapsody, it is possible to start a new AUTOSAR project. In the *Project Type* drop down menu select the desired AUTOSAR release as shown in Figure 5.1.
2. Right click on the *Default* folder in the left panel and click on *Add New* → *AUTOSAR components* → *ApplicationSwComponentType* as shown in Figure 5.2 and then insert a name for the component.
3. Software components in AUTOSAR communicate by means of their ports. To add a new port on the created component, right click on it in the left panel and click on *Add New* → *AUTOSAR\_42* → *dataSender-Port* and then insert a name for the port.
4. Every port must have an interface (as described in Section 3.2.2). To add an interface to a port right click on the desired port in the left panel and click on *Features*. Then in the *Contract* drop down menu select *<New>* as shown in Figure 5.3; in the interface window that appear, insert a name for the interface and then click *Ok*.



## 5.1. MODEL A SIMPLE SYSTEM

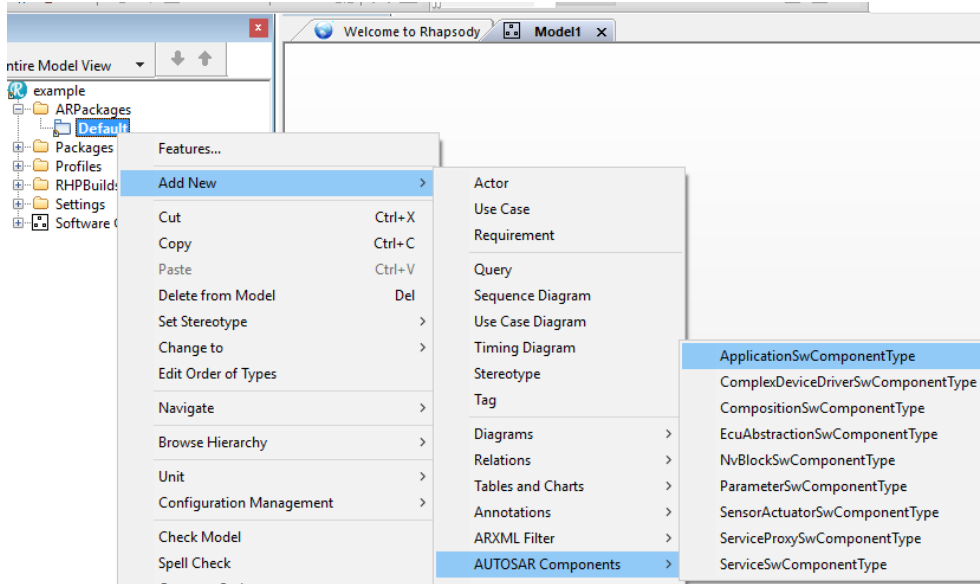


Figure 5.2: Create a new component in Rhapsody

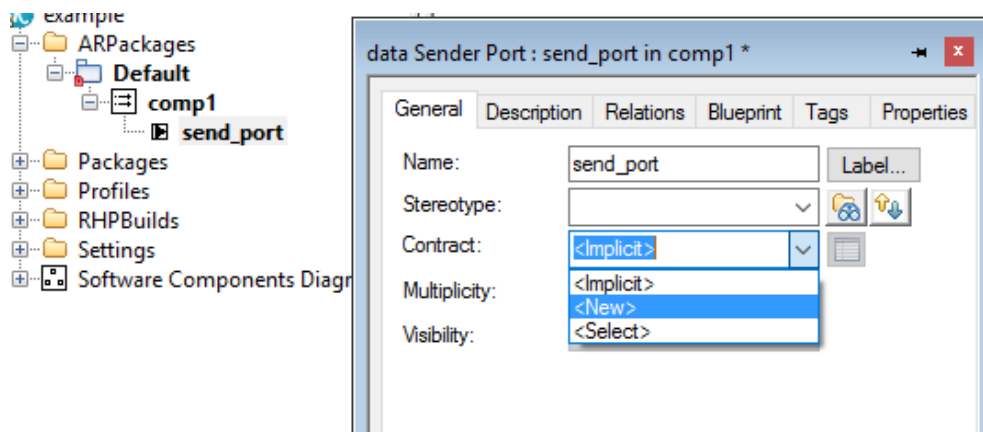


Figure 5.3: Create a port's interface in Rhapsody

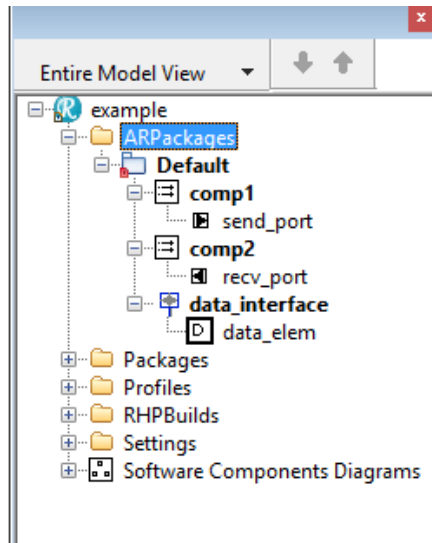


Figure 5.4: Left panel view in Rhapsody

5. For sender/receiver interface, is necessary to specify a data element (the piece of information transmitted among the ports typed by that interface). To specify a data element for an interface, right click on the interface in the left panel and click on *Add New* → *AUTOSAR\_42* → *dataElement*. In the window that appears, it is possible to insert a name for the *dataElement* and a type.
6. Repeat steps 2 and 3 to create another application software component. This time, to add an interface to the port, there's no need to create a new interface, because to be able to communicate between each other, two ports must have the same interface. So, we assign to this port the interface created in step 4: right click on the desired port in the left panel and click on *Features*, then in the *Contract* drop down menu select *<Select>* and choose the interface created in step 4.
7. At this point, the left panel should be similar to the one shown in Figure 5.4.
8. To specify the connection between components, we need to create a *CompositionSwComponentType*; to create it, right click on the *Default*

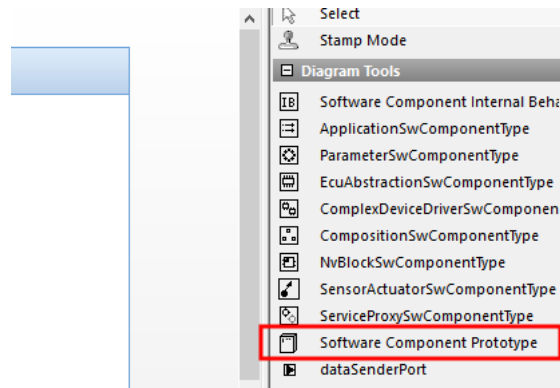


Figure 5.5: Software component prototype in Rhapsody

folder in the left panel and click on *Add New* → *AUTOSAR components* → *CompositionSwComponentType* and insert a name for it. The following steps can be completed like before (by adding components from the left panel) or can be completed "graphically" by simply drag and drop (we follow this way, because it is faster than the other way).

9. Drag and drop the *CompositionSwComponentType* created in the previous step, in the *Model* tab.
10. Using the right panel, add two *SwComponentPrototypes* (as shown in Figure 5.5). *SwComponentPrototypes* are going to be an instance of the software components created in the previous steps.
11. To specify the type for a *SwComponentPrototypes*, right click on it and then click on *Features*. In the *Features* window, in the *Type* drop down menu, select a type as shown in Figure 5.6.
12. Repeat the step 11 for the other *SwComponentPrototypes*. Now the model should look like the one shown in Figure 5.7.
13. To connect the two components, click on the *AssemblySwConnector* in the right panel, then click the ports that have to be connected.
14. Final result is shown in Figure 5.8.

## 5.1. MODEL A SIMPLE SYSTEM

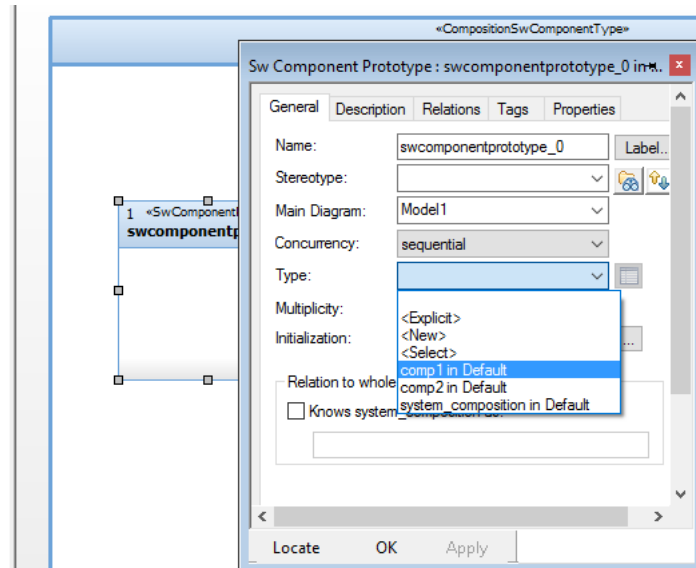


Figure 5.6: Select a prototype type in Rhapsody

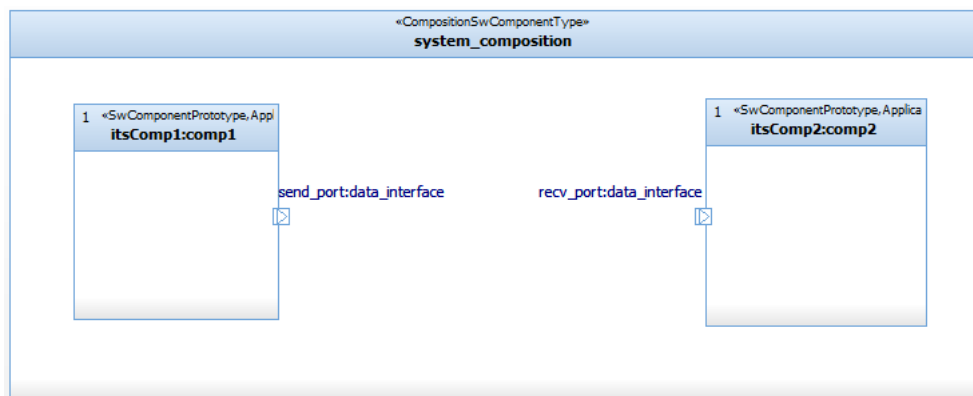


Figure 5.7: Software composition model in Rhapsody

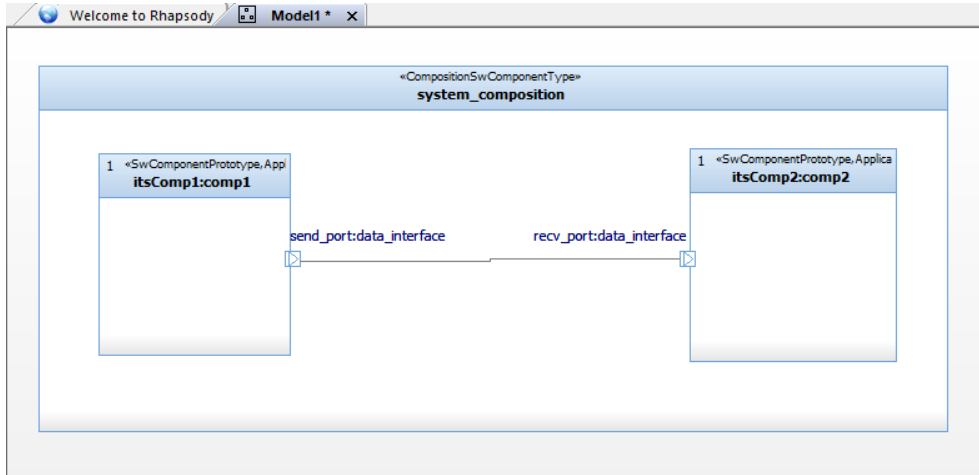


Figure 5.8: Software composition model in Rhapsody (final model)

## 5.2 Internal behavior and runnable entities

This section describes how the internal behavior and the runnable entities of a software component can be specified in Rhapsody, and it specifies also the relations between a runnable entity, the software component's ports and RTE events.

Take the system created in Section 5.1 as a starting point, the following steps refer to the sender component (*comp1*):

1. The first step consists in adding the internal behavior element, which acts as a container for runnable entities and RTE events: right click on the sender component in the left panel and then click on *Add New* → *AUTOSAR\_42* → *SwcInternalBehavior*.
2. To add a runnable entity inside the behavior element, right click on *SwcInternalBehavior* element created in the previous step and then click on *Add New* → *AUTOSAR\_42* → *RunnableEntity*.
3. In AUTOSAR all runnable entities are activated by the RTE as a result of an RTE events (as stated in Section 4.2.2.3 of [10]); so, for every runnable entity we have to specify the triggering RTE event. Suppose that the sender component is activated periodically (i.e. once per

second). This kind of RTE event is called *TimingEvent* and can be created by right clicking on the *SwcInternalBehavior* element and then by clicking on *Add New* → *AUTOSAR\_42* → *TimingEvent*. To specify the period of the event, right click on the *TimingEvent* element and then click on *Features*. In the *Tags* tab there is the *period* attribute in which we can specify the period in second (as shown in Figure 5.9).

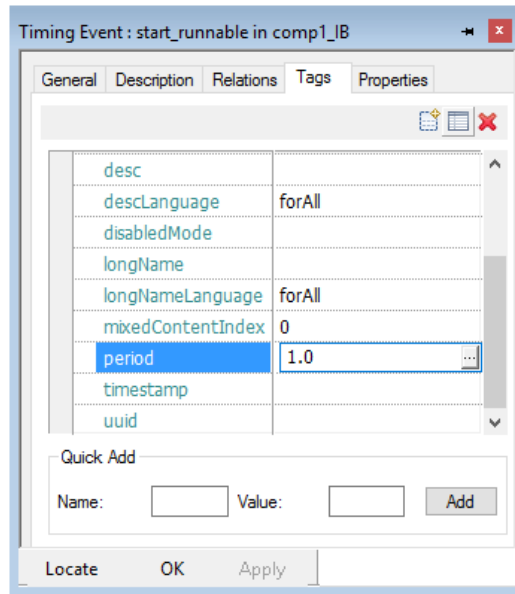


Figure 5.9: Specify the period attribute of the RTE TimingEvent

4. To specify that a runnable entity has to be triggered by an RTE event, right click on the RTE event element (which, in our case, is the *TimingEvent* created in the previous step) and then click on *Add New* → *AUTOSAR\_42* → *I\_startOnEvent*, which is used to reference the runnable entity that has to be started when this event occurs. In the drop down menu that appears, select the runnable entity which has to be triggered.
5. To be able to send data, the runnable entity needs to access to the sender's port and to the data element defined in the port's interface. For this purpose, we need to add a *dataWriteAccess* element to the

## 5.2. INTERNAL BEHAVIOR AND RUNNABLE ENTITIES

runnable entity: right click on the runnable entity and then click on *Add New* → AUTOSAR\_42 → *dataWriteAccess*. Now, to specify the data element to which the runnable entity needs to access, right click on the *dataWriteAccess* and then click on *Add New* → AUTOSAR\_42 → *accessedVariable*. Finally, to reference the data element, right click on the *accessedVariable* element and then click on *Add New* → AUTOSAR\_42 → *I\_localVariable*. In the drop down menu select the name of the data element to be accessed, which in our example is *data\_elem* belonging to the *data\_interface*.

At this point, the left panel in Rhapsody should be similar to the one shown in Figure 5.10. Note the reference of the *dataWriteAccess* (*datawriteaccess\_0* in figure) to the data element *data\_elem*, which is the data element of the sender-receiver interface named *data\_interface*.

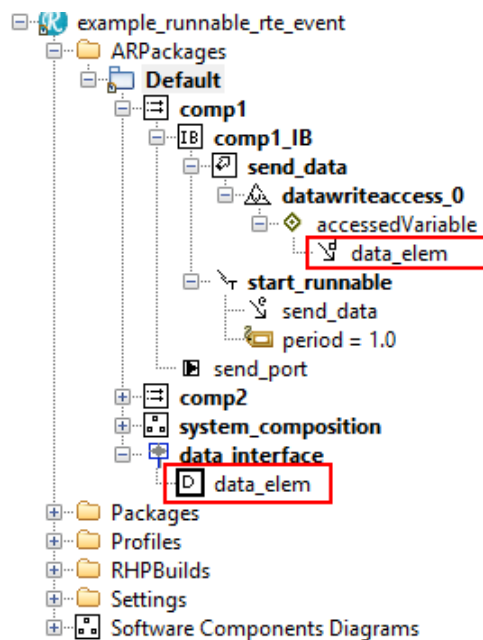


Figure 5.10: Runnable and RTE events in Rhapsody

Repeat all the previous steps on the receiver side (*comp2*). The differences are the following:

1. The RTE event which trigger the runnable entity of the receiver, in this case, could be the *DataReceivedEvent*: this event trigger the referenced runnable entity when the referenced data element is received.
2. This time the runnable entity needs a read access to the receiver's port, not a write access. So the element to add to the runnable entity is the *dataReadAccess* element.

### 5.2.1 Implicit and explicit data reception and transmission

The RTE supports 'explicit' and 'implicit' data reception and transmission:

- **Implicit** data access transmission means that a runnable does not actively initiate the reception or transmission of data. Instead, the required data is received automatically when the runnable starts and is made available for other runnables at the earliest when it terminates.

The *dataWriteAccess* fall in this category.

- **Explicit** data reception and transmission means that a runnable employs an explicit API call to send or receive certain data elements. Depending on the category of the runnable and on the configuration of the according ports, these API calls can be either blocking or non-blocking.

More information about implicit and explicit data reception and transmission can be found in Section 4.3.1.5 of [10].

## 5.3 Add End-to-End protection

This section describes how the End-to-End (E2E) protection mechanism can be specified in Rhapsody, in order to protect the data exchanged between two or more software components, against the effect of software or hardware faults (more information about E2E protection mechanism can be found in Section 4.1.1).



Starting from the system created in Section 5.1, to add E2E protection the steps to follow are:

1. Right click on *Default* folder in the left panel and then click on *Add New* → *AUTOSAR\_42* → *EndToEndProtectionSet*.
2. To add an *EndToEndProtection*, right click on the *EndToEndProtectionSet* element created in the previous step and then click on *Add New* → *AUTOSAR\_42* → *EndToEndProtection*. *EndToEndProtection* acts as a container for configuration of the E2E protection: it is used to specifies an E2E protection profile and to specifies on which ports the E2E protection mechanism has to be applied.
3. To specify an E2E protection profile, right click on *EndToEndProtection* element created in the previous step, then click on *Add New* → *AUTOSAR\_42* → *endToEndProfile*; right click on it and then click on *Features*. In the *Tags* tab, click on the *category* attribute and insert a profile as shown in Figure 5.11. The possible values (defined in Section 4.7 of [2]) are the following:
  - NONE
  - PROFILE\_01
  - PROFILE\_02

The value *NONE* specifies that the E2E protection wrapper (described in Section 4.1.1) works as pass-through. Everyone of the other profiles has a different E2E protection setting: for instance, algorithm used to compute CRC in *PROFILE\_01* is not the same as the one used in *PROFILE\_02*.

4. Now we have to define to which *VariableDataPrototypes*, in the roles of one sender and one or more receivers, this *EndToEndprotection* applies. "In the role of a sender" means that the sender applies the E2E protection mechanism by attaching control information (CRC, counter etc.) to the data. "In the role of one or more receivers" means that the

### 5.3. ADD END-TO-END PROTECTION

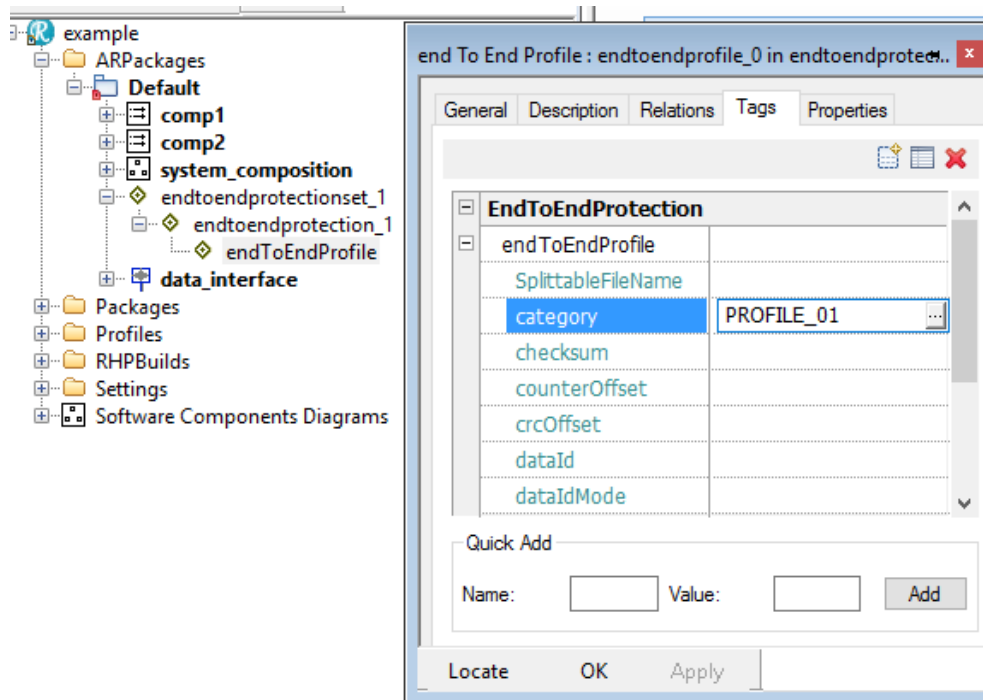


Figure 5.11: Specify an E2E protection profile in Rhapsody

receiver/receivers of the data applies the E2E protection by verifying the control information attached to the data. To define to which *VariableDataPrototypes* the E2E protection has to be applied, we have to add an *EndToEndProtectionVariablePrototype* element: right click on the *endToEndProfile* created in the previous step and then click on *Add New* → *AUTOSAR\_42* → *EndToEndProtectionVariablePrototype*.

5. To specify a sender for the *EndToEndProtectionVariablePrototype*, right click on *EndToEndProtectionVariablePrototype* element created in the previous step, then click on *Add New* → *AUTOSAR\_42* → *sender*. This *sender* has to have a reference to the data element to be protected; right click on the *sender* element, then click on *Add New* → *AUTOSAR\_42* → *targetDataPrototype* (in Rhapsody the name of this element is *I\_targetDataPrototype*). In the drop down menu select the data element of the interface created in our example as shown in Figure 5.12.

## 5.4. ADD SECURITY (BY USING CRYPTO SERVICE MANAGER)

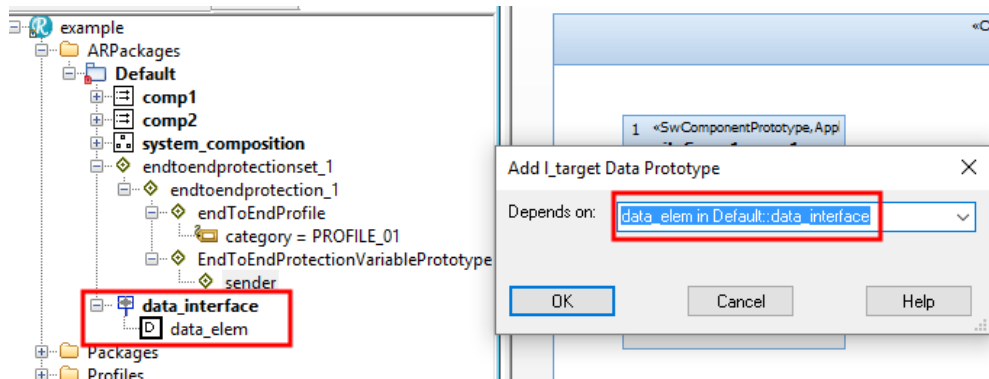


Figure 5.12: Data element to which the E2E protection has to be applied

6. Repeat the previous step, but this time, instead of *sender* select *receiver*.
7. At this point the left panel of Rhapsody should be similar to the one shown in Figure 5.13.

## 5.4 Add security (by using Crypto Service Manager)

This section describes how the Crypto Service Manager (CSM) can be specified in Rhapsody, in order to protect the communication between two or more software components against malicious alteration of messages (CSM provides not only this kind of protection, but it provides a variety of protection mechanisms; more information about CSM can be found in Section 4.2.1).

Starting from the system created in Section 5.1 (and shown in Figure 5.14), suppose that we don't want only data integrity to be guaranteed (which can be achieved by using the E2E protection), but we want also data origin authentication: the receiver should be able to verify that the received data comes from a trusted entity. For this purpose, the E2E protection is not sufficient, because it can only guarantees data integrity. To achieve both (integrity and authenticity) we can make use of a service provided by the

## 5.4. ADD SECURITY (BY USING CRYPTO SERVICE MANAGER)

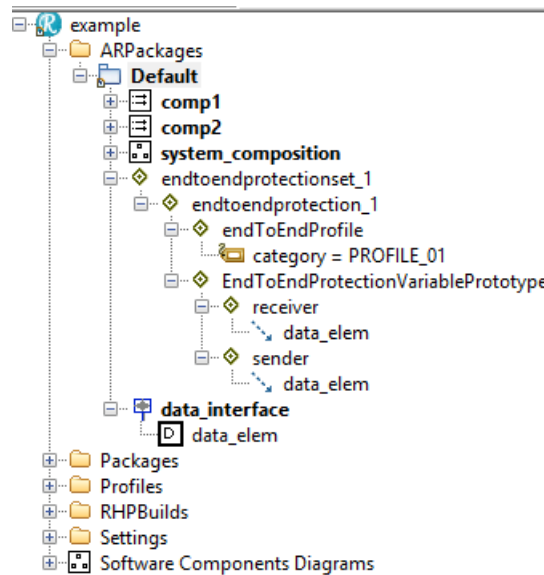


Figure 5.13: Final view for the E2E protection in Rhapsody

CSM. CSM offers a variety of services related to security aspects and for our example we focus on Message Authentication Code (MAC) service (for more information about CSM, see Section 4.2.1).

Before continue, it is important to remember that a *SwComponentPrototype* represents an instance of a *SwComponentType* within a composition (*CompositionSwComponentType*). In our example, *itsComp1* and *itsComp2* are *SwComponentPrototypes*. *itsComp1* is an instance of *comp1* (the sender component), and *itsComp2* is an instance of *comp2* (the receiver component). It is also important to remember that the CSM can be used locally only (in this context locally means "on the same ECU"). So, if the sender and the receiver reside on different ECUs, CSM has to be placed on both of them.

Furthermore, the following assumptions are made:

- The sender and the receiver share a secret key. This key will be used for MAC computation. The sender component, by means of the operations provided by the CSM, computes the MAC on the data he wants to send, he attaches the MAC to the data and then he sends the message (data + MAC) to the receiver. Note that, as specified in Section 8.1.7 of [5],

## 5.4. ADD SECURITY (BY USING CRYPTO SERVICE MANAGER)

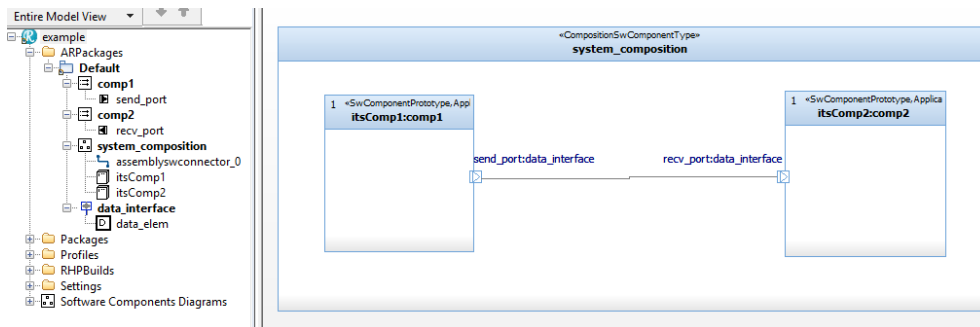


Figure 5.14: System overview, before the specification of the CSM

it is the CSM that shall check if the provided buffer is large enough to hold the result of the MAC computation. The receiver, after the reception of the message, verifies the MAC by means of the operations provided by the CSM.

- The communication between sender and the CSM, and also between the receiver and the CSM, is synchronous.

Starting from the sender (*comp1*), to specify that he needs to use the MAC service provided by the CSM, the steps to follow are listed below:

1. The CSM is an AUTOSAR service and it uses a client-server communication paradigm: the client of the service requires a service (invoke an operation) and the CSM performs the required operation and returns the result to the client of the service. So, we need to add a client port to the sender: right click on the sender in the left panel (*comp1* in Figure 5.14) and then click on *Add New* → *AUTOSAR\_42* → *clientPort*, and insert a name for it.
2. Create the port interface: right click on the port created in the previous step and then click on *Features*. In the *Contract* drop down menu select *<New>*. The name of the interface cannot be chosen at will, and must be exactly *CsmMacGenerate*, because in AUTOSAR, every service interface has standardized name (specified in the AUTOSAR documentation).

## 5.4. ADD SECURITY (BY USING CRYPTO SERVICE MANAGER)

Furthermore, in the *Tags* tab of the interface features window, check the *isService* box (to specify that this interface is used to interact with an AUTOSAR service) and select *cryptoServiceManager* in the *serviceKind* drop down menu (as shown in Figure 5.15).

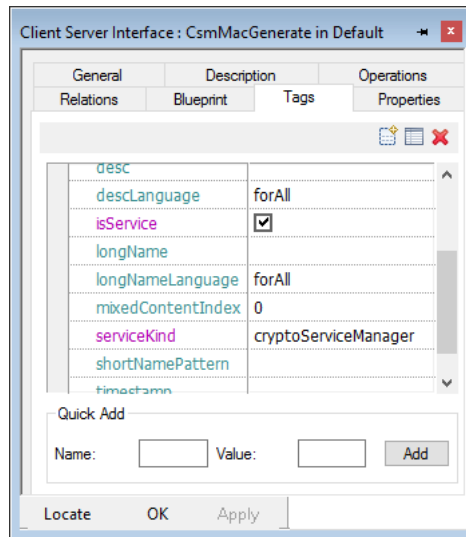


Figure 5.15: Set interface tags for a service interface in Rhapsody

3. Add an internal behavior element to the sender component: right click on the sender in the left panel and then click on *Add New* → *AUTOSAR\_42* → *SwcInternalBehavior*.
4. Right click on the internal behavior element of the sender, created in the previous step, and then click on *Add New* → *AUTOSAR\_42* → *SwcServiceDependency* (which is used to associate ports to a given service).
5. Within a *SwcServiceDependency* we need to add one and only one class derived from *ServiceNeeds*. In our case we add a *cryptoServiceNeeds* by clicking on *SwcServiceDependency* element created in the previous step and then clicking on *Add New* → *AUTOSAR\_42* → *cryptoServiceNeeds*. In the *Features* window of the *cryptoServiceNeeds* (in the *Tags*

#### 5.4. ADD SECURITY (BY USING CRYPTO SERVICE MANAGER)

tab) there is the attribute *maximumKeyLength*, in which it is possible to specify the maximum length of a cryptographic key (in bit).

6. We also have to add a *RoleBasedPortAssignment* to the *SwcServiceDependency* element: right click on *SwcServiceDependency* created in step 4 and then click on *Add New* → *AUTOSAR\_42* → *assigned-Port*. Add a reference to the service port by clicking on the *assigned-Port* element and then by clicking on *Add New* → *AUTOSAR\_42* → *I\_portPrototype*. In the drop down menu that appears, select the service port of the software components as shown in Figure 5.16.

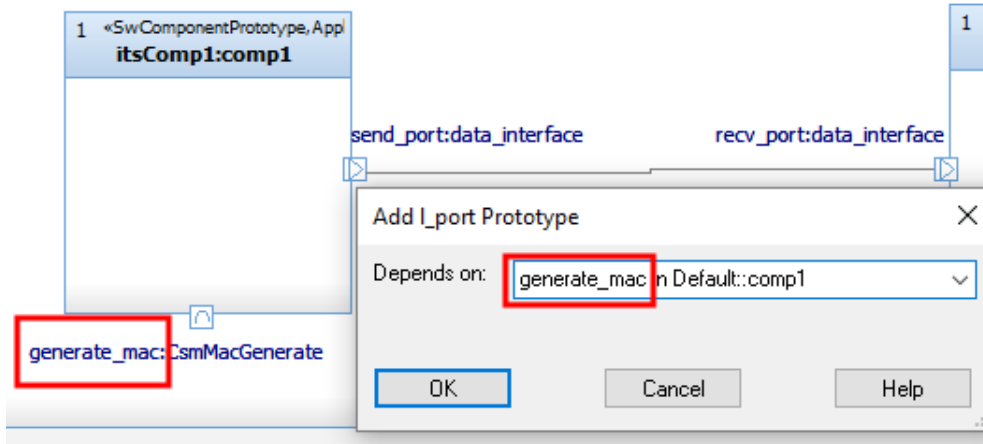


Figure 5.16: Assign a SwcServiceDependency to a port in Rhapsody

Repeat all the previous steps for the receiver (*comp2* in Figure 5.14). The only differences are in the name of the interface and obviously in the assigned port:

- the name of the interface in this case must be *CsmMacVerify*: because the receiver of the data does not generate MAC, but when he receives the data, he needs to verify it;
- the assigned port will be the port which is typed by the *CsmMacVerify* interface.

## 5.4. ADD SECURITY (BY USING CRYPTO SERVICE MANAGER)

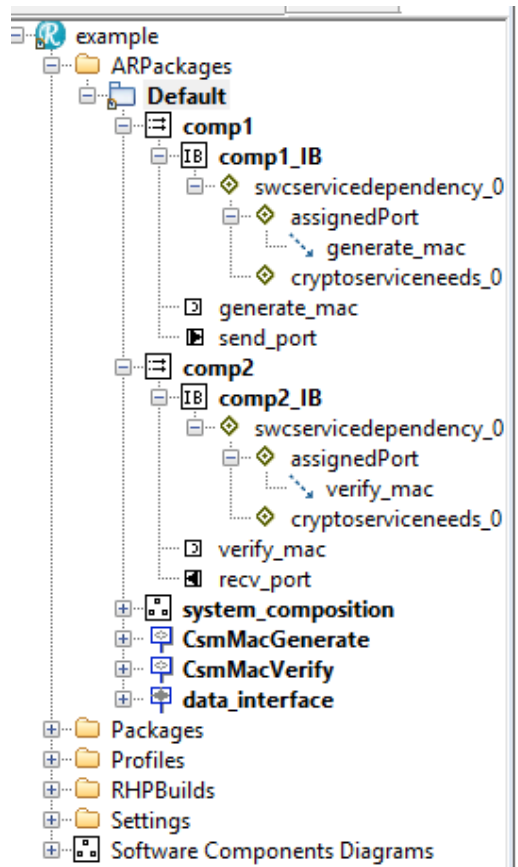


Figure 5.17: Left panel view after the specification of service dependency

At this point the left panel in Rhapsody should be similar to the one shown in Figure 5.17. How the services are used by a software component, depends on the software component's runnable entities and on their implementation.

To specify the needs to use other kind of security mechanisms, such as: hash function, symmetric or asymmetric cryptography, etc. the steps to follow are the same. The main difference is in the names of the interfaces (all the standardized names of the interfaces can be found in the CSM documentation [5]).



## 5.5 Generate ARXML

As already stated in Section 2.2, the standardized format for exchanging data between different AUTOSAR compliant tools is AUTOSAR XML (ARXML). Also Rhapsody is able to generate ARXML.

To generate ARXML in Rhapsody:

- click on the *Tools* menu and then click on *AUTOSAR → Export AUTOSAR XML Document*;
- click on the *Browse* button to specify a name for the generated ARXML file.
- click on the *Export* button to start the ARXML generation process.

If no errors occurs, the ARXML description of the model will be generated.

# Chapter 6

## Security level and automatic generation of security elements

As described in Chapters 3 and 5, to specify that a software component needs to use an AUTOSAR service, the procedure that system developers have to follow is a bit tricky and error prone: they have to know the services meta-model and the features of the service they need to use. The AUTOSAR documentation provides all these information, but is not easy to find what you are looking for in more than 200 documents (in the AUTOSAR release 4.2.2), some of which are more than one thousand pages long.

With this in mind, we proposed an approach for the specification of security levels and for the automatic generation of the AUTOSAR elements which are required to use the security services provided by AUTOSAR by means of the Crypto Service Manager (CSM). The security levels that we have defined are the following:

- integrity;
- confidentiality;
- both;

They apply to communication links (or better, to the ports involved in the communication), and can be specified by the developers by the insertion of

a specific security tag in the description field (*desc*, in AUTOSAR specifications) of the ports of a software component. At the end, the security levels must be written in the ARXML file, which is the file format used in AUTOSAR to describes a system. For this purpose, the developers have two possible choices:

1. they can insert the security tag by using an AUTOSAR compliant tool like Rhapsody, and then they can export the system as ARXML file;
2. they can insert the security tag directly in the ARXML file which represents the system.

By parsing the ARXML file, the Python[20] script that we have developed, automatically generates the required elements based on the specified security level.

The security requirements are fulfilled by using the services provided by the CSM (which is part of the AUTOSAR service layer). In order to use these services, a component has to have what we called "elements", which in practice are client ports, interfaces, and so on. The script allows the developers to choose if these elements have to be added directly within the component which requires a specific security level, or if they have to be added within a new component (which acts as a filter) that performs the following actions:

1. it takes the data to protect as input;
2. it applies the required security level to the received data;
3. it sends out the protected data.

The modified ARXML file can be imported in Rhapsody; all the elements added by the script are visible in the graphical representation of the system.

In order to use Python, Windows users can install Liclipse; an Integrated Development Environment (IDE) based on Eclipse which provides the plugin for Python development. For Linux users, Python is usually available in the official repository of the Linux distribution.

## 6.1 Security level specification and elements generation

In practice, the security levels are expressed as a tag in the form of a pair [name; value] within the description field of the two ports involved in the communication and they can assume the following values:

- *SecurityNeeds=INTEG*, which stands for "integrity": in a communication between two entities (A and B), if A sends a message to B, B is able to verify that the received message was not altered by an external entity;
- *SecurityNeeds=CONF*, which stands for "confidentiality": in a communication between two entities (A and B), a third (non-authorized) entity (C), is not able to understand the content of the message exchanged between A and B;
- *SecurityNeeds=BOTH*, which means that both (integrity and confidentiality) are required.

In addition, the developers can specify if the security has to be added by means of a new component or if it has to be added on the existing component. For this purpose, the tag to add in the description field of the port is the following:

- *NewComponent=TRUE*: the script adds a new component which provides the required security elements. This component acts as a filter: it takes the output data of one component, it applies the required security service to that data, and then it sends out the protected data;
- *NewComponent=FALSE*: the required security elements are added directly within the component which requires the security service.

Setting *NewComponent=TRUE* can be useful when the developers cannot modify an existing component (i.e. a legacy component). If the *NewComponent* tag is not found, the script assumes *NewComponent=FALSE* as default value: no new component is added.

The values of the *SecurityNeeds* tag has to be the same for the ports involved in the communication; instead, the value of the *NewComponent* tag can be different.

## 6.2 Example usage

Take the simple system modeled in Section 5.1 as a starting point (reported in Figure 6.1 for convenience), in which there is one sender (*comp1*) and one receiver (*comp2*) which exchange data between each other. Suppose that we

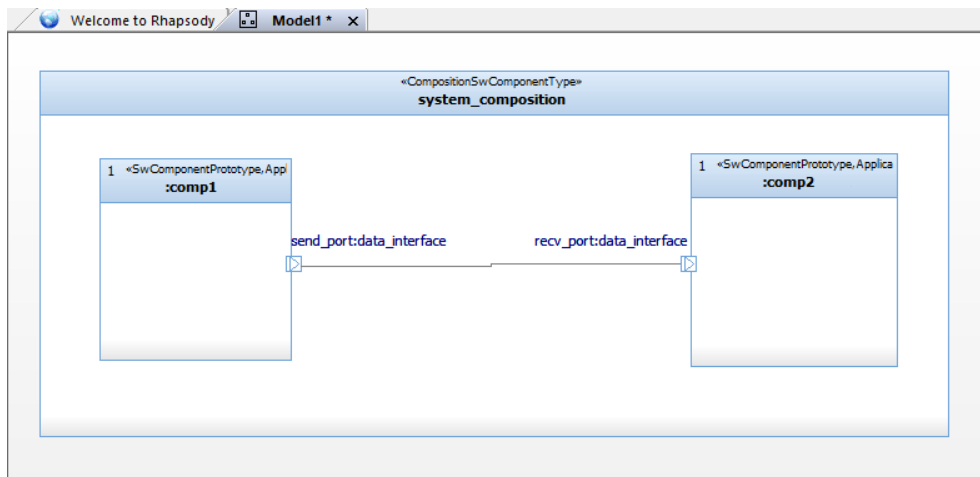


Figure 6.1: System used as a starting point

are not interested in confidentiality, instead we want the receiver to be able to verify the integrity of the messages received from the sender. For this purpose the CSM offers the Message Authentication Code (MAC) service, which guarantees integrity and also authenticity of the received data (Section 8.1.7 of [5]), under the condition that the key used for MAC computation is not compromised by an external entity. Under the assumption of a synchronous communication paradigm between the component and the AUTOSAR service, the steps that the developers have to follow, in order to specify that a software component wants to use the MAC service, are described in Section 5.4 and they are briefly recap here for convenience:

1. the software component that wants to use the MAC service needs to have a client port;
2. the interface of the client's port has to have exactly one of the following names: *CsmMacGenerate* if the software component is a sender, *CsmMacVerify* if it is a receiver. All the standardized names of all the interfaces provided by the CSM are written in [5];
3. the *isService* attribute of the interface must be set to *true*;
4. the value of the *serviceKind* attribute of the interface must be *cryptoServiceManager*;
5. the internal behavior element of the software component must contain a *SwcServiceDependency*, which in turn must contain exactly one *cryptoServiceNeeds* and one or more *RoleBasedPortAssignment*;
6. the *RoleBasedPortAssignment* must have a reference to the service port of the software component (the port defined in the step 1).

The script we have developed, automates this process. The developers only need to insert the tags previously described:

1. *SecurityNeeds=INTEG*, *SecurityNeeds=CONF* or *SecurityNeeds=BOTH*;
2. *NewComponent=TRUE* or *NewComponent=FALSE* (if no specified, *NewComponent=FALSE* is assumed as default);

The aforementioned tags have to be inserted in the description field of the ports involved in the communication. In Rhapsody this can be done by right clicking on the port element of a software component and then by clicking on *Features*. In the *Tags* tab of the *Features* window, there is the *desc* field in which the developers can insert the desired tags. An example is shown in Figure 6.2. Rhapsody also allows the user to define a new couple [tag; value] within an element, which seems a most reasonable solution than inserting the tags in the description field. But we decided to use the description field, since it is part of the AUTOSAR standard and then it is exported in ARXML file.

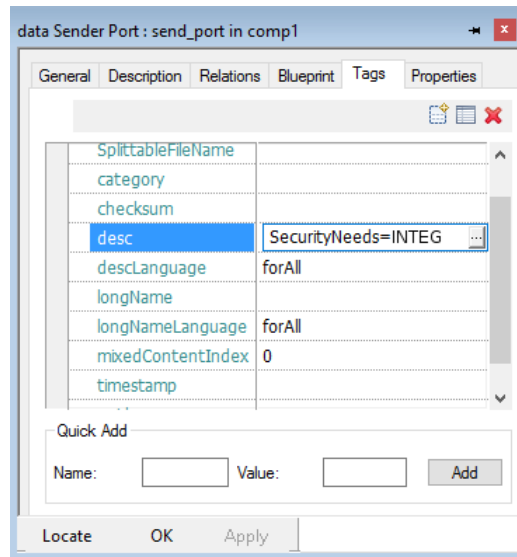


Figure 6.2: Specify a security tag

If we specify *SecurityNeeds=INTEG* and *NewComponent=TRUE* for both components, by executing the script on the ARXML representing the modeled system, the resulting system is shown in Figure 6.3. The script added

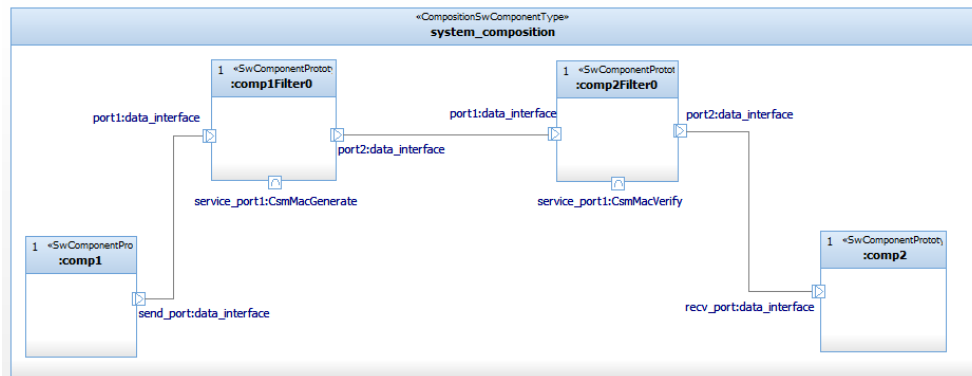


Figure 6.3: Required elements added by means of new components

the two components (*comp1Filter0* and *comp2Filter0*):

- on the sender side, the filter takes the data produced by the sender, it computes the MAC value on the data, and then it sends out the data+MAC;

- on the receiver side, the filter reads the data+MAC, it verifies the MAC value and it sends the data to the receiver component.

If we specify *SecurityNeeds=INTEG* and *NewComponent=FALSE* for both components, the resulting system is the one shown in Figure 6.4. This

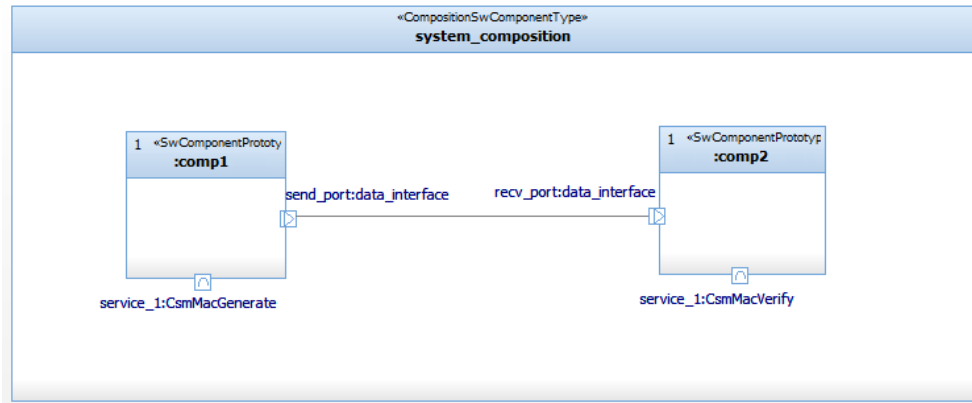


Figure 6.4: Required elements added to the existing components

time, no new components have been added to the system, because we specify *NewComponent=FALSE* for both components. All the required elements have been added to the existing components: the service ports, the interfaces needed to use the MAC service and other elements not visible in the figure.

## 6.3 ARXML and Python

This section provides a brief description of the python script and shows an excerpt of the ARXML file before and after the script execution. The ARXML file refers to the system used also in Section 6.2.

### 6.3.1 Python script

The input parameters of the script are the name of the ARXML file and, optionally, a name for the output file. If no name is specified for the output file, the script use a default name for it.

For every connections, the script performs the following steps:



1. it checks if the ports involved in the communication have a security tags specified in their description field;
2. if no security tag was found, or if the two ports have two different security tags, the script moves on the next connection to analyze;
3. if the two ports have the same security tag, the script first looks for the *NewComponent* tags and based on their value, it builds 0, 1 or 2 new components;
4. it adds the following elements in the components created in the previous step, or in the existing components, based on the value of the *NewComponent* tag:
  - (a) based on the security tag it creates the needed ports;
  - (b) it creates the service interface;
  - (c) it creates the internal behavior element;
  - (d) it adds the *SwcServiceDependency*, *RoleBasedPortAssignment* and all the other required elements to the internal behavior;
  - (e) it creates the connection between all the involved components;
  - (f) if needed, it deletes the original connection;

At the end, the script saves the changes in the specified ARXML file. Below, the pseudo-code of the *add\_security* function is shown.

```
#Analyze and add the required security levels to the system described in the  
#ARXML file and saves the changes in a new ARXML file  
#function's paramentes:  
# file_name: the name of the ARXML file which describes the system  
# new_file_name: the name of the file where the changes have to be saved  
#returned value:  
# -1 in case of errors  
# 0 otherwise  
def add_security(file_name, new_file_name=""):  
    #Variables initialization
```

```

...

#Checks if file exists
if ( os.path. isfile (file_name) == False):
    return -1

if (new_file_name == ''):
    new_file_name = "new_" + file_name.split(".arxml")[-2]

#For every connections (ASSEMBLY-SW-CONNECTOR) checks if a
# security level is specified , and applies it.
#Note: root is the root of the tree
for conn in root.iter(namespace+"ASSEMBLY-SW-CONNECTOR"):
    #Get the ports involved in the communication (the provider port and
    #the required port)
    ...

    #Get the two components to which the ports belong
    comp1 = get_component(root, provider_name)
    comp2 = get_component(root, requester_name)
    ...

    #Checks the tags within the ports. "add_filter1" is equal to
    #TRUE if "provider_port" of "comp1" has the tag
    #"NewComponent=TRUE". The same reasoning applies to
    #"add_filter2"
    [add_filter1, tag1] = get_tags(comp1, provider_port)
    [add_filter2, tag2] = get_tags(comp2, requester_port)

    #If the two tags are the same and they are not empty...
    if (tag1 == tag2) and (tag1 != ""):

        if (add_filter1 == FALSE):
            #Add service elements to component1 (comp1)

```

```

...
else:
    #Build a new component (filter1) which act as a filter
    #for comp1
    ...

if (add_filter2 == "FALSE"):
    #Add service elements to component2 (comp2)
    ...
else:
    #Build a new component (filter2) which act as a filter
    #for comp2
    ...

    #Update connections
    ...

    #save the changes to 'new_file_name' and return
    update_tree(root, new_file_name)
return 0

```

## 6.4 ARXML code

In this section, an excerpt of the ARXML code of the system modeled in Section 6.2 is shown: the code refers to the *comp1* of Figure 6.1. Below, the ARXML which describes the *comp1* is shown.

```

...
<SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
  <SHORT-NAME>comp1</SHORT-NAME>
  <PORTS>
    <P-PORT-PROTOTYPE>
      <SHORT-NAME>send_port</SHORT-NAME>
      <PROVIDED-INTERFACE-TREF DEST="SENDER-RECEIVER-
        INTERFACE">/Default/data_interface</PROVIDED-INTERFACE-

```

```

    TREF>
  </P-PORT-PROTOTYPE>
</PORTS>
</SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
...

```

After the specification of the security tag *SecurityNeeds=INTEG*, the ARXML of *comp1* looks like the one shown below. The differences are highlighted in green.

```

...
<SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
  <SHORT-NAME>comp1</SHORT-NAME>
  <PORTS>
  <P-PORT-PROTOTYPE>
    <DESC>
    <L-2 L="FOR-ALL">SecurityNeeds=INTEG</L-2>
    </DESC>
    <PROVIDED-INTERFACE-TREF DEST="SENDER-RECEIVER-
      INTERFACE">/Default/data_interface</PROVIDED-INTERFACE-
      TREF>
  </P-PORT-PROTOTYPE>
</PORTS>
</SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
...

```

Note that there isn't the *NewComponent* tag in the description field (*<DESC>*); so, after the script execution, the elements that are needed to use the services provided by the CSM, are written inside the block which represents the *comp1*. The resulting ARXML is the following (the differences are highlighted in different colors):

```

...
<SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
  <SHORT-NAME>comp1</SHORT-NAME>
  <PORTS>
  <P-PORT-PROTOTYPE>
    <SHORT-NAME>send_port</SHORT-NAME>
    <DESC>
    <L-2 L="FOR-ALL">SecurityNeeds=INTEG</L-2>

```

```

</DESC>
<PROVIDED-INTERFACE-TREF DEST="SENDER-RECEIVER-
INTERFACE">/Default/data_interface</PROVIDED-INTERFACE-
TREF>
<R-PORT-PROTOTYPE>
  <SHORT-NAME>service_1</SHORT-NAME>
  <REQUIRED-INTERFACE-TREF DEST="CLIENT-SERVER-
INTERFACE">/Default/CsmMacGenerate</REQUIRED-INTERFACE-
TREF>
</R-PORT-PROTOTYPE>
</PORTS>
<INTERNAL-BEHAVIORS>
<SWC-INTERNAL-BEHAVIOR>
  <SHORT-NAME>comp1_IB</SHORT-NAME>
  <SERVICE-DEPENDENCYS>
  <SWC-SERVICE-DEPENDENCY>
    <SHORT-NAME>comp1_servive_dep</SHORT-NAME>
    <ASSIGNED-PORTS>
    <ROLE-BASED-PORT-ASSIGNMENT>
      <PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">/
      Default/comp1/service_1</PORT-PROTOTYPE-REF>
    </ROLE-BASED-PORT-ASSIGNMENT>
    </ASSIGNED-PORTS>
    <SERVICE-NEEDS>
    <CRYPTO-SERVICE-NEEDS>
      <SHORT-NAME>cryptoserviceneeds</SHORT-NAME>
    </CRYPTO-SERVICE-NEEDS>
    </SERVICE-NEEDS>
  </SWC-SERVICE-DEPENDENCY>
</SERVICE-DEPENDENCYS>
</SWC-INTERNAL-BEHAVIOR>
</INTERNAL-BEHAVIORS>
</SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
...

```

The blue part contains the service port, which has a reference to the *CsmMac-Generate* interface. The green part contains the internal behavior elements, the service dependency and so on.

In the other component (*comp2*) the changes are similar to that of *comp1*;

the main noticeable differences is the service interface, which is *CsmMacVerify*.

The ARXML structure, obviously reflects the service meta-model described in Section 3.3 and shown in Figure 3.10.

# Chapter 7

## Application to an AUTOSAR use case

This chapter, shows a possible usage of the python script, by applying it to an AUTOSAR use case. Section 7.1 provides a description of the use case. Section 7.2 shows the systems before and after the execution of the script.

### 7.1 Use case description

The AUTOSAR document [21], describes the low beam function of the front light system of a car. In the following we focus on the security aspects.

The system overview is shown in Figure 7.1. The general purpose of the low beam system is to illuminate the roadway in the dark. The low beams are turned on by the user through the Light Switch, if the Ignition Key is ON. System status (malfunctions included) shall be reported to the driver through the Human Machine Interface (HMI). The functioning rules for the Front Light Management (FLM) are the following:

1. Detection of a low beam request:
  - The FLM shall evaluate the Ignition Key position;
  - The FLM shall read the Light Switch position;
2. Evaluate the low beam light request:

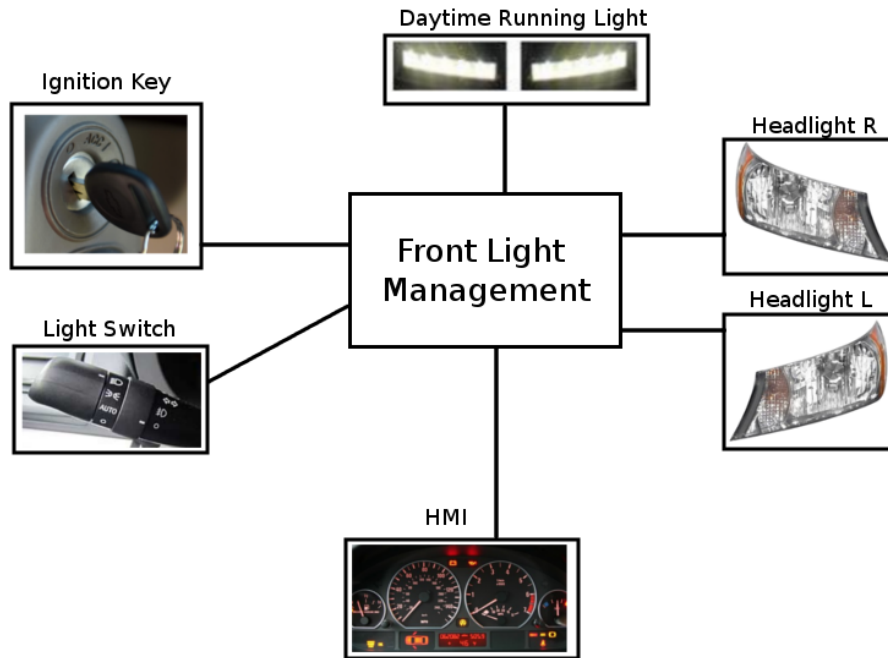


Figure 7.1: Front lights system overview

- The FLM shall evaluate the Light Switch status;
  - The FLM shall create a switch event ON if the Light Switch status changes from OFF to ON;
  - The FLM shall create a switch event OFF if the Light Switch status changes from ON to OFF;
3. Control the low beam lights:
- The FLM shall activate the low beam lights if the Ignition Key position is ON and a Light Switch event ON is detected;
  - The FLM shall deactivate the low beam lights if the Ignition Key position is OFF, or a Light Switch event OFF is detected;
4. Monitoring the low beam lights function:
- The FLM shall supervise the low beam lights;



## 7.1. USE CASE DESCRIPTION

System element	Communication mean
Light Switch	Digital Input Output (DIO)
Ignition Key	Controller Area Network (CAN)
HMI	CAN
Head Lights (left and right)	Pulse Width Modulation (PWM)
Daytime Running Lights (left and right)	PWM

Table 7.1: Communication means between FLM and other elements

- The FLM shall display the low beam lights status and report malfunctions to the user by means of the HMI;

### 5. Activation of the daytime running lights:

- The FLM shall activate the daytime running lights in case of a low beam lights malfunctioning.

Figure 7.2 and table 7.1 shows how the involved entities communicate with the FLM.

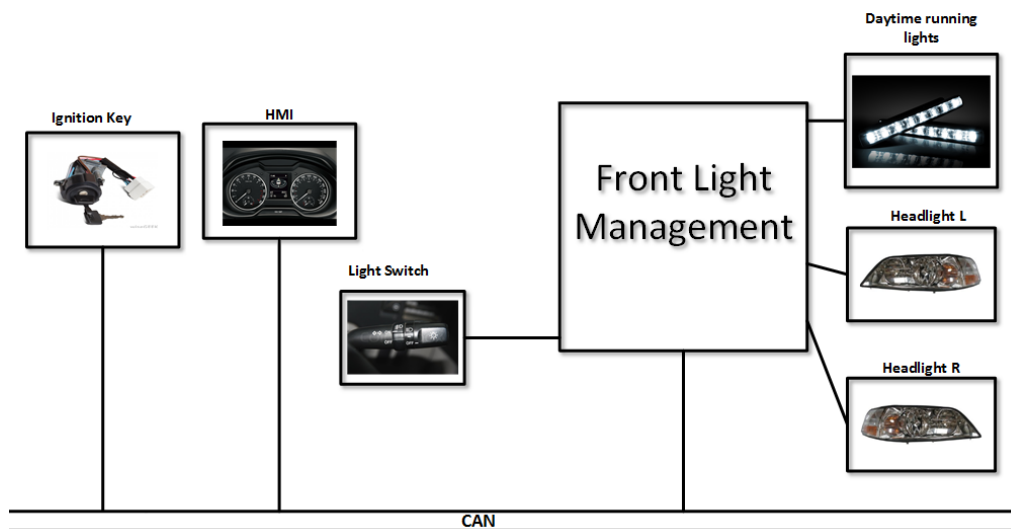




Figure 7.2: Communication focused view of the front light system

## 7.2 Application to the use case

As stated in [4], since many ECUs are connected to the CAN bus, a compromised ECU can send altered messages to every ECU connected to the CAN. For this reason, we apply security on the CAN bus communications.

The initial system is shown in Figure 7.3. Note that  represents a sender-receiver port, which can be used for both sending and receiving data, and  represents a client port, which is used here for communication with the services provided by the CSM.

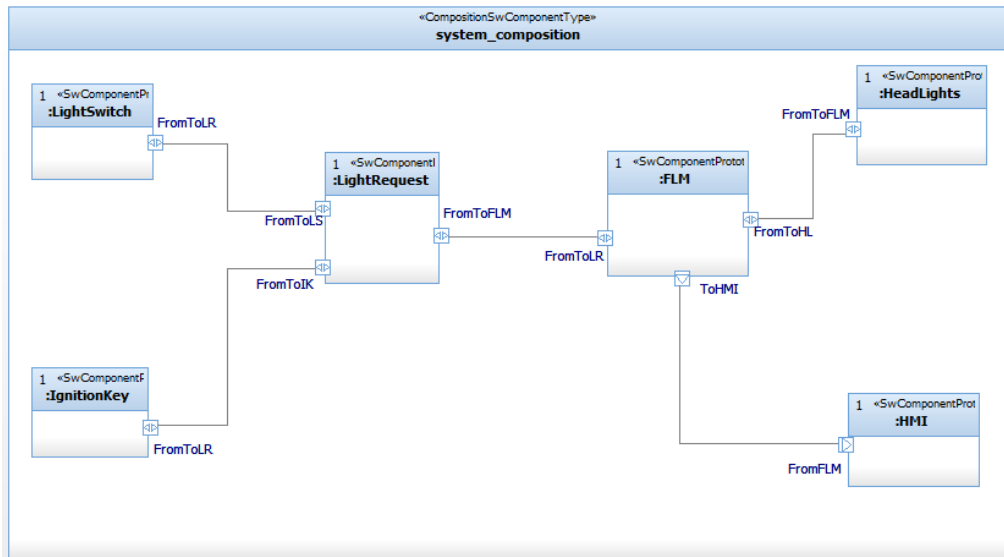


Figure 7.3: Initial FLM system

We assume the following:

- *LightRequest* and *FLM* reside on the same ECU;
- *IgnitionKey* and *HMI* share a secret key with *FLM*;
- *IgnitionKey* and *HMI* cannot be modified by the developers;

We also assume that communications within the same ECU are secure. So, we apply the following tags to the components' ports:

- *FromToLR* of *IgnitionKey*: *SecurityNeeds=INTEG* and *NewComponent=TRUE*;
- *FromToIK* of *LightRequest*: *SecurityNeeds=INTEG*;
- *ToHMI* of *FLM*: *SecurityNeeds=INTEG*;
- *FromFLM* of *HMI*: *SecurityNeeds=INTEG* and *NewComponent=TRUE*.

After the script execution, the resulting system is shown in Figure 7.4. The

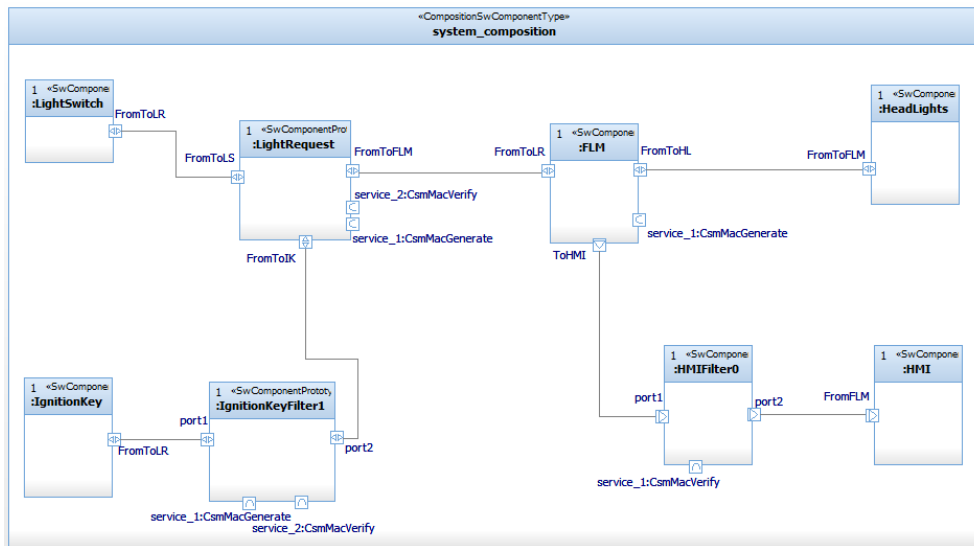


Figure 7.4: Final FLM system

script, based on the specified tags, has made the following changes to the system:

- it adds *IgnitionKeyFilter1*, which is connected to *IgnitionKey* and to *LightRequest*;
- it adds the required service ports to *LightRequest* and *FLM*;
- it adds *HMIFilter0*, which is connected to *FLM* and to *HMI*.

Both, *IgnitionKeyFilter1* and *LightRequest*, have two service ports, because the communication is bi-directional:

## 7.2. APPLICATION TO THE USE CASE

---

- when *IgnitionKey* sends data to *LightRequest*, *IgnitionKeyFilter1* (on the behalf of *IgnitionKey*) has to compute the MAC value on the data, so it needs to use the *CsmMacGenerate* service;
- when *IgnitionKey* receives data from *LightRequest*, *IgnitionKeyFilter1* (on the behalf of *IgnitionKey*) has to verify the MAC value of the received data, so it needs to use the *CsmMacVerify* service.

# Chapter 8

## Conclusions

In this work, we analyzed the AUTOSAR standard, which is quite verbose, but it is the de-facto standard for embedded software development in automotive industry. In order to develop an AUTOSAR compliant system, the developers must have a good knowledge of the standard and often this is not enough.

The approach we have proposed can be used by the developers to specify high level security requirements already in the early stages of the system design following a "security by design" methodology. A tool has been developed to ease the work of system developers and to avoid oversight caused by the complexity of the AUTOSAR standard. We focused on the security aspects of communications, but our tool can be further improved, for example, to take into account also the specification of the End-to-End (E2E) protection mechanism.

There are other aspects of the AUTOSAR standard that can be subject of future work; for example, safety requirements in AUTOSAR are expressed in natural language and this can lead to ambiguity and misunderstandings. Also the compliance between the final system and the requirements may be difficult to verify. For this reason, the specification of a formal way to define the requirements may be useful.

# Acknowledgments

I would like to thank Prof. Marco Di Natale from Scuola Superiore Sant'Anna, for the useful discussions and suggestions about the AUTOSAR standard, and Stefania Botta from Magneti Marelli for her valuable advice on the AUTOSAR Crypto Service Manager.

I thank my supervisors, Prof. Cinzia Bernardeschi and Prof. Gianluca Dini, for their support and guidance during the development of this thesis.

# Acronyms

**API** Application Programming Interface. 12, 45

**ARXML** AUTOSAR XML. 1, 13, 36, 54, 56, 59–62, 64, 65, 67

**ASIL** Automotive Safety Integrity Level. 27, 28

**AUTOSAR** AUTomotive Open System ARchitecture. 1, 3, 5–10, 12–14, 16, 17, 19–25, 27, 32, 33, 35–37, 42, 50, 51, 54–56, 58, 59, 68, 74, 75

**BSW** Basic Software. 9, 12, 13, 32

**CAL** Crypto Abstraction Library. 32, 33

**CAN** Controller Area Network. 5, 29, 34, 70, 71

**CPU** Central Processing Unit. 26

**CRC** Cyclic Redundancy Check. 28, 30, 46

**CSM** Crypto Service Manager. 5, 25, 33–36, 48–50, 53, 55, 56, 58, 59, 65, 71

**DIO** Digital Input Output. 70

**E2E** End-to-End. 25–29, 31, 32, 36, 45–48, 74

**ECU** Electronic Control Unit. 5, 8–10, 12, 29, 32, 34, 49, 71

**FLM** Front Light Management. 68–70

**HMI** Human Machine Interface. 68, 70

**I-PDU** Interaction layer Protocol Data Unit. 29

**IDE** Integrated Development Environment. 56

**LIN** Local Interconnect Network. 29

**MAC** Message Authentication Code. 49, 50, 52, 58–61, 73

**NVRAM** Non-Volatile Random Access Memory. 12

**OEM** Original Equipment Manufacturer. 8

**OS** Operating System. 12, 26

**PWM** Pulse Width Modulation. 70

**RTE** Runtime Environment. 9–13, 21, 22, 27, 31–33, 36, 42, 43, 45

**SecOC** Secure On-board Communication. 32

**SWC** Software Component. 9, 10, 17

**UML** Unified Modeling Language. 13, 14

**XML** eXtensible Markup Language. 13



# Bibliography

- [1] AUTOSAR: Layered Software Architecture. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR\\_EXP\\_LayeredSoftwareArchitecture.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf).
- [2] AUTOSAR Software Component Template, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/templates/standard/AUTOSAR\\_TPS\\_SoftwareComponentTemplate.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/templates/standard/AUTOSAR_TPS_SoftwareComponentTemplate.pdf).
- [3] Federal Bureau of Investigation - Internet Crime Complaint Center. Motor vehicles increasingly vulnerable to remote exploits. <http://www.ic3.gov/media/2016/160317.aspx>, 2016.
- [4] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces, 2011. USENIX Security.
- [5] Specification of Crypto Service Manager, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/standard/AUTOSAR\\_SWS\\_CryptoServiceManager.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/standard/AUTOSAR_SWS_CryptoServiceManager.pdf).
- [6] Safure project. <https://safure.eu/>.
- [7] Robert N. Charette. This car runs on code. <http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code>, 2009.

- [8] AUTOSAR standard. <http://www.autosar.org>.
- [9] AUTOSAR core partners. <http://www.autosar.org/partners/current-partners/core-partners/>.
- [10] Specification of RTE, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/standard/AUTOSAR\\_SWS\\_RTE.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf).
- [11] Interoperability of AUTOSAR Tools, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/tools/auxiliary/AUTOSAR\\_TR\\_InteroperabilityOfAutosarTools.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/tools/auxiliary/AUTOSAR_TR_InteroperabilityOfAutosarTools.pdf).
- [12] Glossary, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR\\_TR\\_Glossary.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR_TR_Glossary.pdf).
- [13] International Organization for Standardization. <http://www.iso.org>.
- [14] Overview of Functional Safety Measures in AUTOSAR, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR\\_EXP\\_FunctionalSafetyMeasures.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR_EXP_FunctionalSafetyMeasures.pdf).
- [15] AUTOSAR E2E Protection library, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/standard/AUTOSAR\\_SWS\\_E2ELibrary.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/standard/AUTOSAR_SWS_E2ELibrary.pdf).
- [16] Requirements on Module Secure Onboard Communication. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR\\_SRS\\_SecureOnboardCommunication.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR_SRS_SecureOnboardCommunication.pdf).
- [17] Specification of Crypto Abstraction Library, release 4.2.2. <http://www.autosar.org/fileadmin/files/releases/4-2/>

software-architecture/safety-and-security/standard/AUTOSAR\_SWS\_CryptoAbstractionLibrary.pdf.

- [18] Utilization of Crypto Services, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR\\_EXP\\_UtilizationOfCryptoServices.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR_EXP_UtilizationOfCryptoServices.pdf).
- [19] IBM Rational Rhapsody software. <http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/>.
- [20] Python programming language. <https://www.python.org/>.
- [21] Safety Use Case Example, release 4.2.2. [http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR\\_EXP\\_SafetyUseCase.pdf](http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/safety-and-security/auxiliary/AUTOSAR_EXP_SafetyUseCase.pdf).