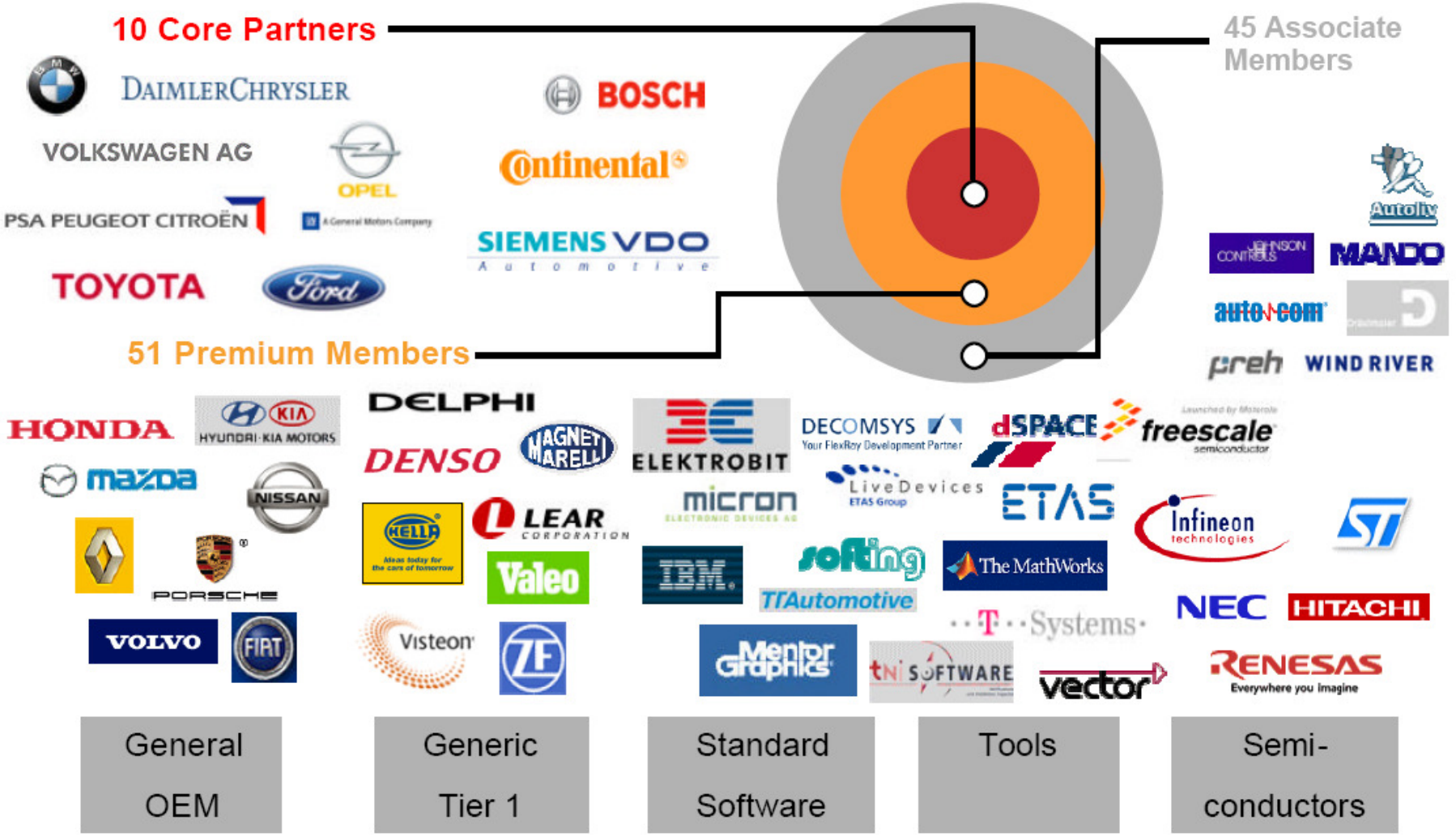
An introduction to AUTOSAR

AUTOSAR Consortium



AUTOSAR

What is AUTOSAR?

AUTOSAR – **AUT**omotive **O**pen **S**ystems **AR**chitecture

Middleware and system-level standard, jointly developed by automobile manufacturers, electronics and software suppliers and tool vendors.
More than 100 members

Motto: "*cooperate on standards, compete on implementations*"
Reality: current struggle between OEM and Tier1 suppliers

Target: facilitate portability, composability, integration of SW components over the lifetime of the vehicle

Standardization of Components and Interfaces

The software implementing the automotive functionality is encapsulated in software components. Standardization of the interfaces is central to support scalability and transferability of functions. Any standard-conformant implementation of a software component can be integrated with substantially reduced effort in a system.

The standardization could be developed incrementally towards:

- Level of abstraction
  - Functional aspects
  - Behavior and implementation aspects
- Level of decomposition
  - Low degree of decomposition of the functional domain
  - High degree of decomposition of the functional domain
- Level of architecture definition
  - Terminology
  - Standardized data-types
  - Partial description of interfaces (without semantics)
  - Complete description of interfaces (without semantics)
  - Complete description of interfaces (with semantics)
  - Partial definition of the functional domain
  - Complete definition of the functional domain

Functional domains

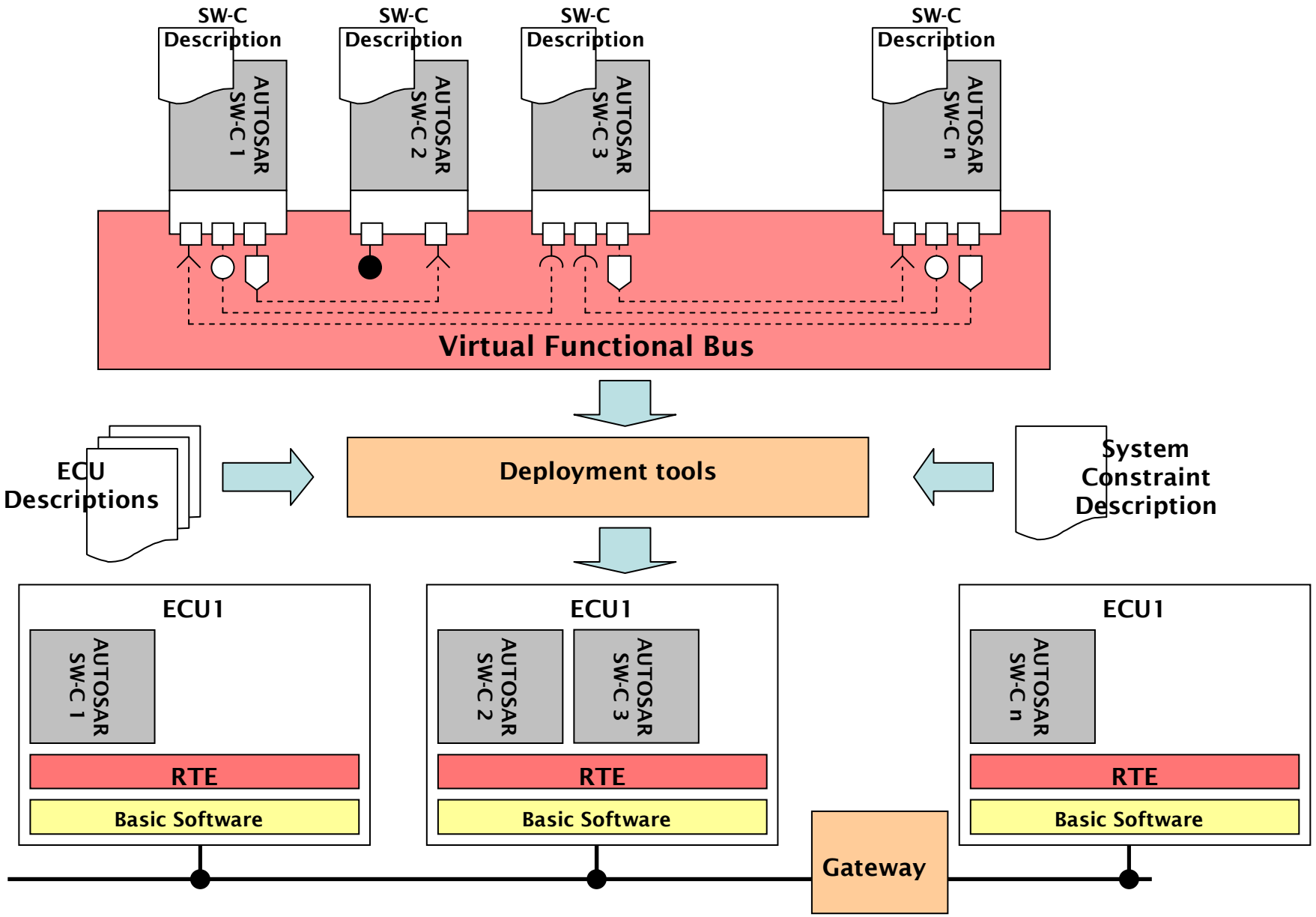
The specification of functional interfaces is divided into 6 domains:

- Body/Comfort
- Powertrain
- Chassis
- Safety
- Multimedia/Telematics
- Man-machine-interface

The domains could be differently handled due to intellectual property rights issues and decomposition levels.

In the first phase of AUTOSAR only in the domains body/comfort, chassis, and powertrain results can be expected. All others have lower priority in the first phase.

AUTOSAR Architecture



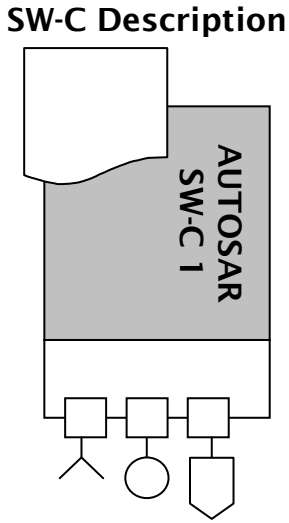
AUTOSAR Architecture

- **AUTOSAR SW-C**

  The AUTOSAR Software Components encapsulate an application which runs on the AUTOSAR infrastructure. The AUTOSAR SW-C have well-defined interfaces, which are described and standardized.

  – **SW-C Description**

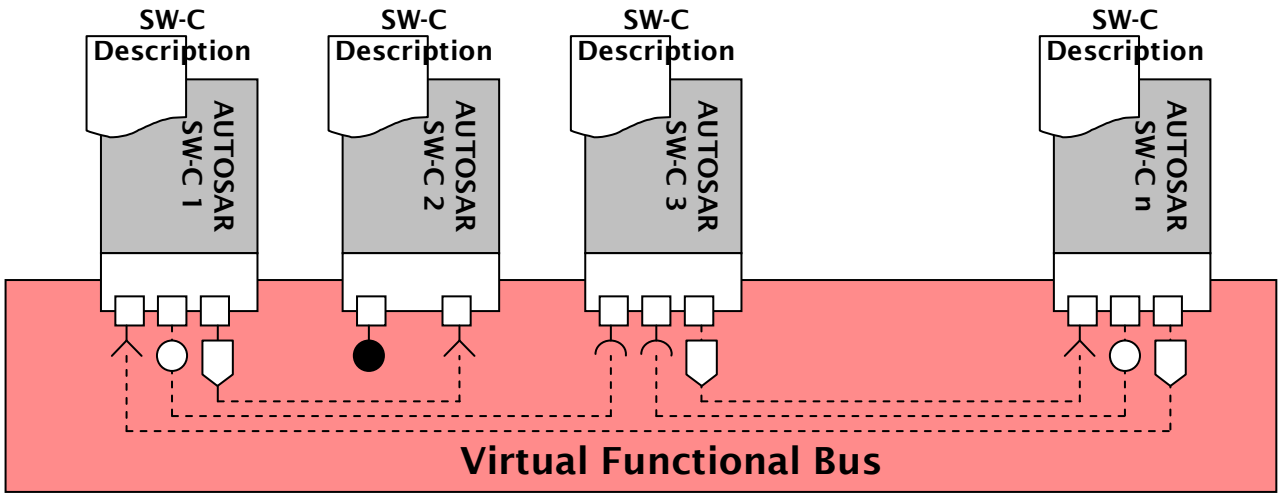
    For the interfaces as well as other aspects needed for the integration of the AUTOSAR Software Components, AUTOSAR provides a standard description format (SW-C Description).



AUTOSAR Architecture

- **Virtual Functional Bus (VFB)**

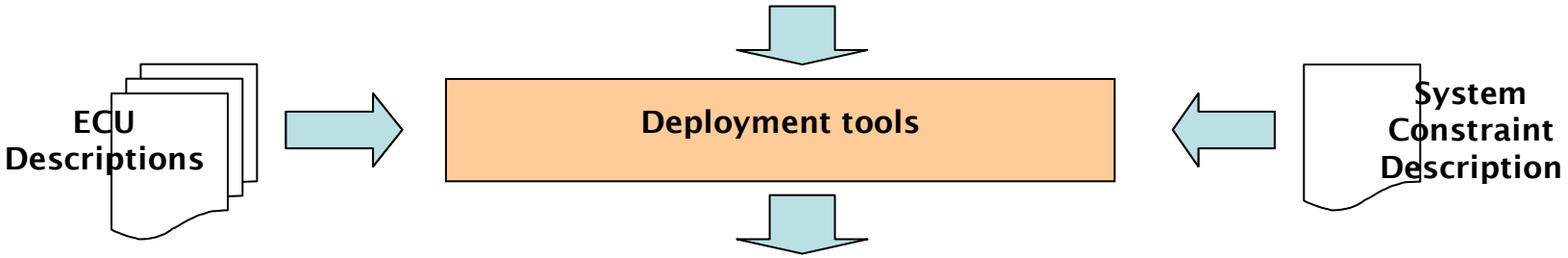
  The VFB is the sum of all communication mechanisms (and interfaces to the basic software) provided by AUTOSAR on an abstract (technology independent) level. When the connections for a concrete system are defined, the VFB allows a virtual integration in an early development phase.



AUTOSAR Architecture

- **System Constraint and ECU Descriptions**

  In order to integrate AUTOSAR SW-Components into a network of ECUs, AUTOSAR provides description formats for the system as well as for the resources and the configuration of the ECUs.

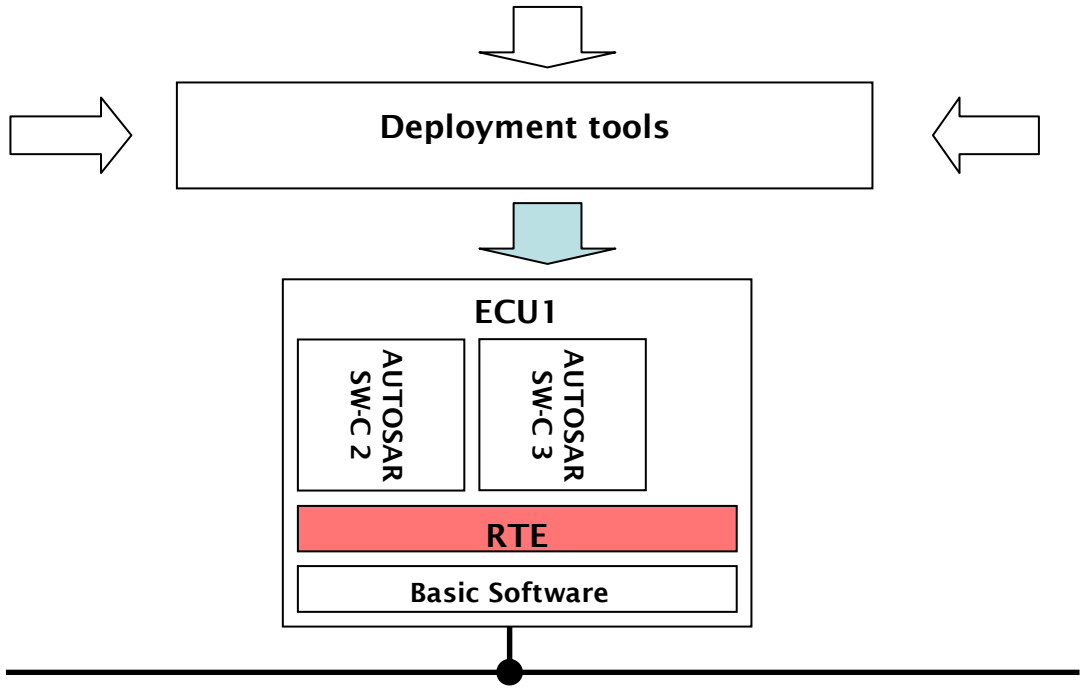


AUTOSAR Architecture: Mapping on ECUs

AUTOSAR defines the methodology and tool support to build a concrete system of ECUs. This includes the configuration and generation of the Runtime Environment (RTE) and the Basic Software (RTOS) on each ECU.

- **Runtime Environment (RTE)**

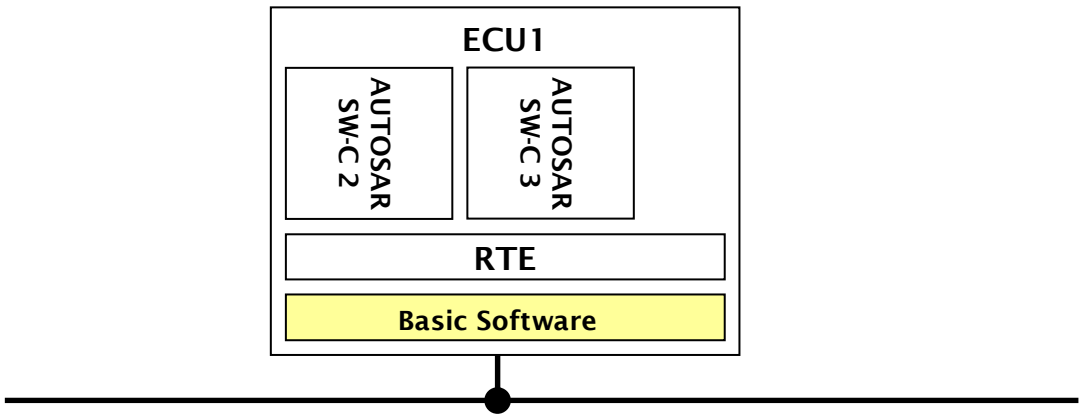
  From the viewpoint of the AUTOSAR Software Component, the RTE implements the VFB functionality on a specific ECU.



AUTOSAR Architecture

- **Basic Software**

  The Basic Software provides the infrastructure for execution on an ECU.

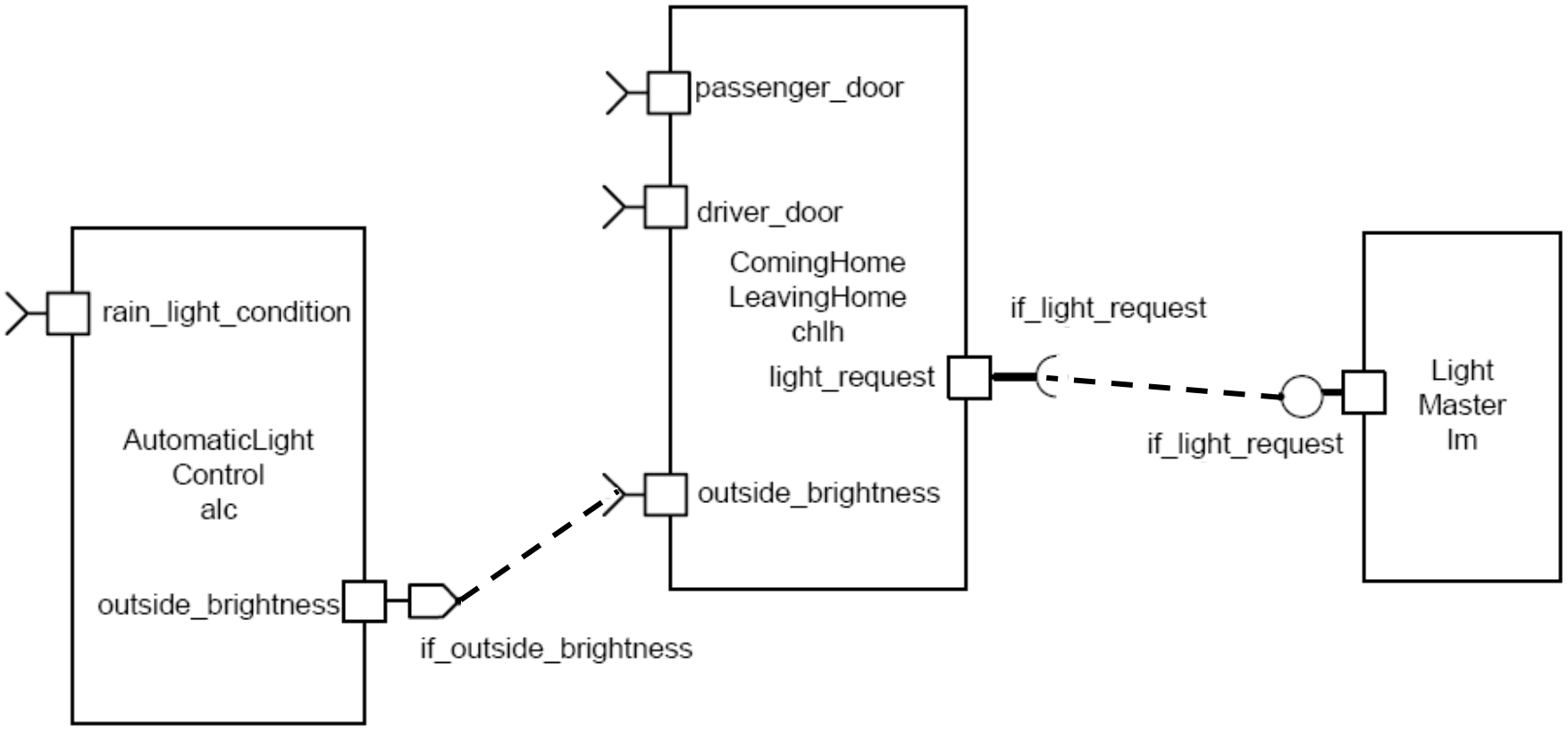


AUTOSAR Architecture

A fundamental concept of AUTOSAR is the separation between:

- **application** and
- **infrastructure**.

An application in AUTOSAR consists of Software Components interconnected by connectors



AUTOSAR Component

The generic “AUTOSAR Component” concept

- AUTOSAR Software Component
- Sensor/Actuator Software Component (special case).
- Composition
  - a logical interconnection of components packaged as a component. In contrast to the Atomic Software Components, the components inside a composition can be distributed over several ECUs.
- ECU Abstraction
- Complex Device Driver
- AUTOSAR Services.

AUTOSAR Component

Each AUTOSAR Software Component encapsulates part of the functionality of the application.

- AUTOSAR does not prescribe the granularity of Software Components. Depending on the requirements of the application domain an AUTOSAR Software Component might be a small, reusable piece of functionality (such as a filter) or a larger block encapsulating an entire sybsystem.

The AUTOSAR Software Component is an "Atomic Software Component"

Atomic means that the each instance of an AUTOSAR Software Component is statically assigned to one ECU.

AUTOSAR Components

Implementing an AUTOSAR Software Component

AUTOSAR does not prescribe HOW an AUTOSAR Software Component should be implemented

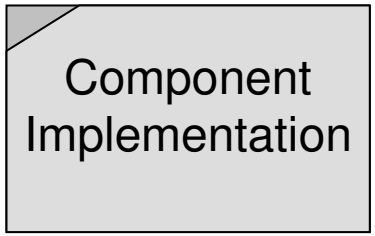
- a component may be handwritten or generated from a model

AUTOSAR Components

Shipping an AUTOSAR Software Component

A shipment of an AUTOSAR Software Component consists of

- a complete and formal ***Software Component Description*** which specifies how the infrastructure must be configured for the component, and
- an ***implementation*** of the component, which could be provided as "object code" or "source code".



AUTOSAR Components: Description

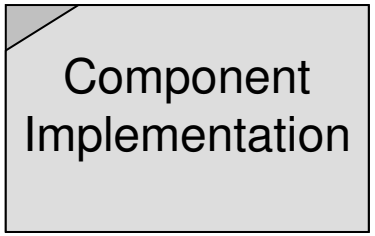
The AUTOSAR Software Component Description contains:

- the operations and data elements that the software component provides and requires
  - described using the PortInterface concept
- the requirements on the infrastructure,
- the resources needed by the software component (memory, CPU-time, etc.),
- information regarding the specific implementation of the software component.
- The structure and format of this description is called “software component template”.

AUTOSAR Components

A source code component implementation is independent from

- the type of microcontroller of the ECU and the type of ECU on which the component is mapped
  - The AUTOSAR infrastructure takes care of providing the software component with a standardized view on the ECU hardware
- the location of the other components with which it interacts. The component description defines the data or services that it provides or requires. The component doesn't know if they are provided from components on the same ECU or from components on a different ECU.
- the number of times a software component is instantiated in a system or within one ECU

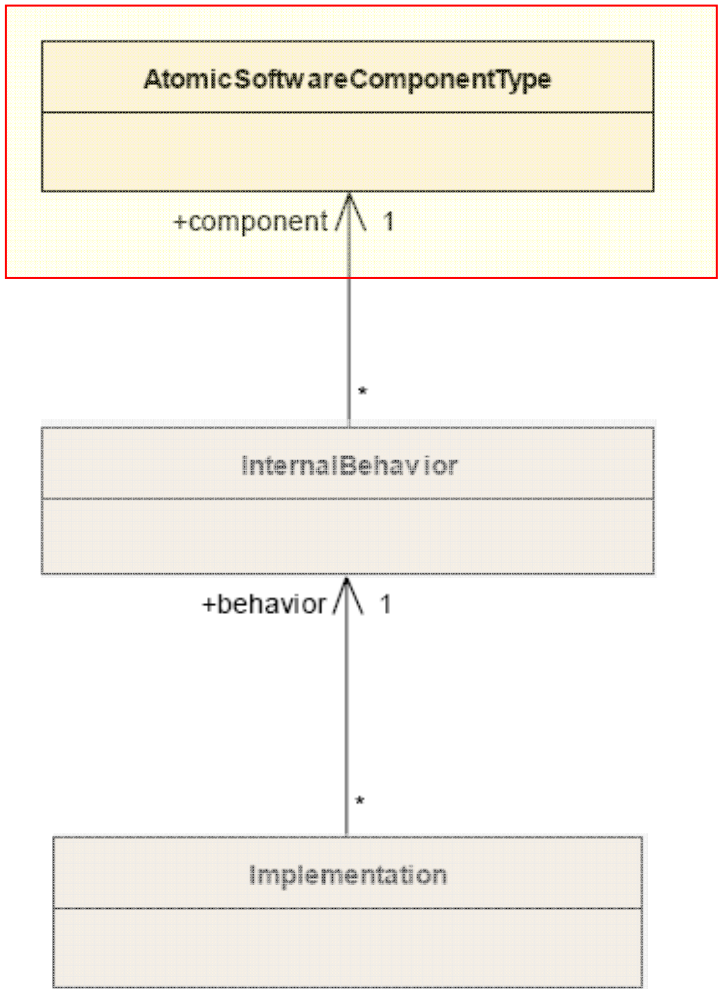


AUTOSAR Components: description levels

The highest (most abstract) description level is the Virtual Functional Bus.

Here components are described with the means of datatypes and interfaces, ports and connections between them, as well as hierarchical components. At this level, the fundamental communication properties of components and their communication relationships among each other are expressed.

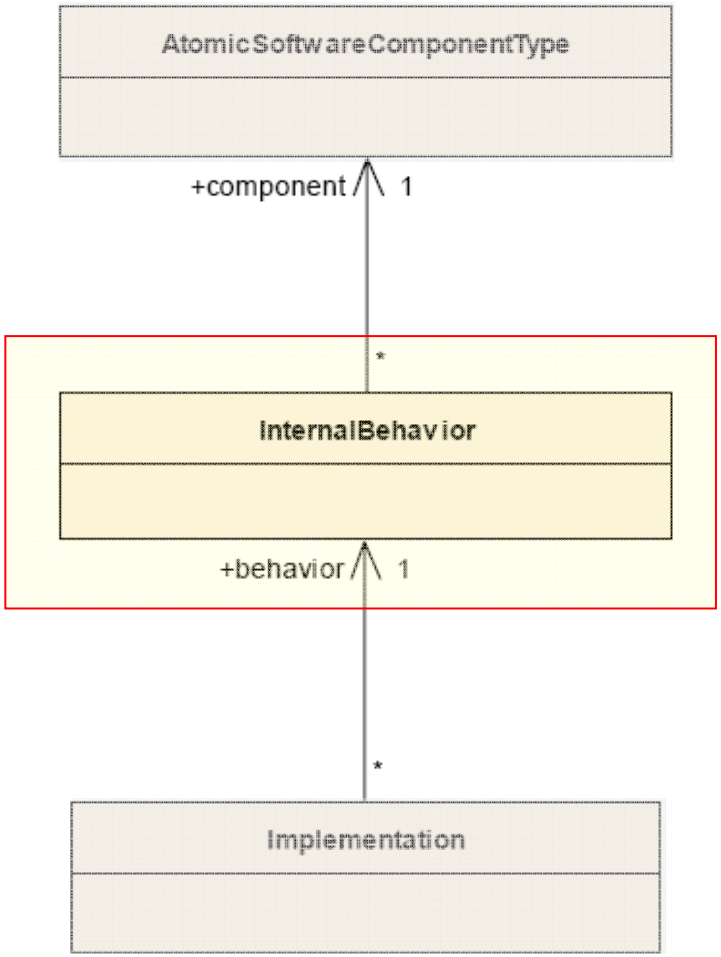
- **Software components**
- **Compositions**
- **Interfaces**



AUTOSAR Components

Description of components on RTE level: The middle level allows for behavior description of a given component. This behavior is expressed through RTE concepts, e.g. RTE events and in terms of schedulable units. For instance, for an operation defined in an interface on the VFB, the behavior specifies which of those units is activated as a consequence of the invocation of that operation.

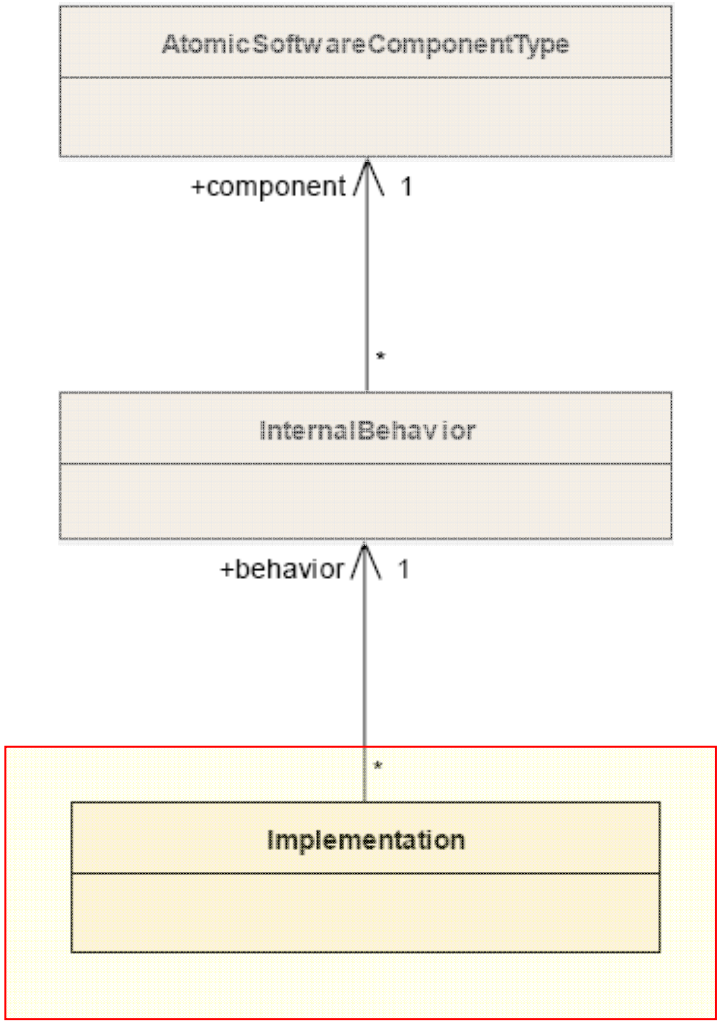
- **Runnables**
- **Events**
- **Interaction with the Run Time Environment**



AUTOSAR Components

Descriptions of components on implementation level: The lowest (most concrete) level of description specifies the implementation of a given behavior. More precisely, the schedulable units of such a behavior are mapped to code. The two layers above constrain the RTE API that a component is offered, the implementation now utilizes this API.

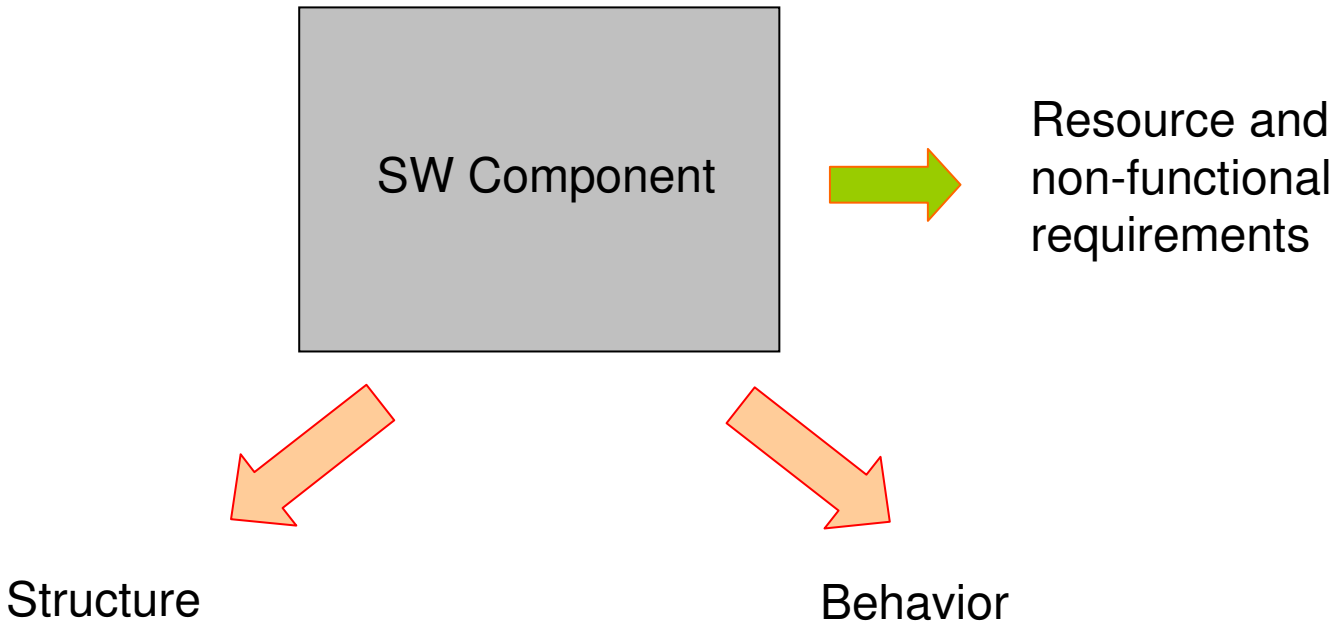
- **Component implementation**
- **Resource consumption of SW-Components**



Component-oriented design

What is a SW component?

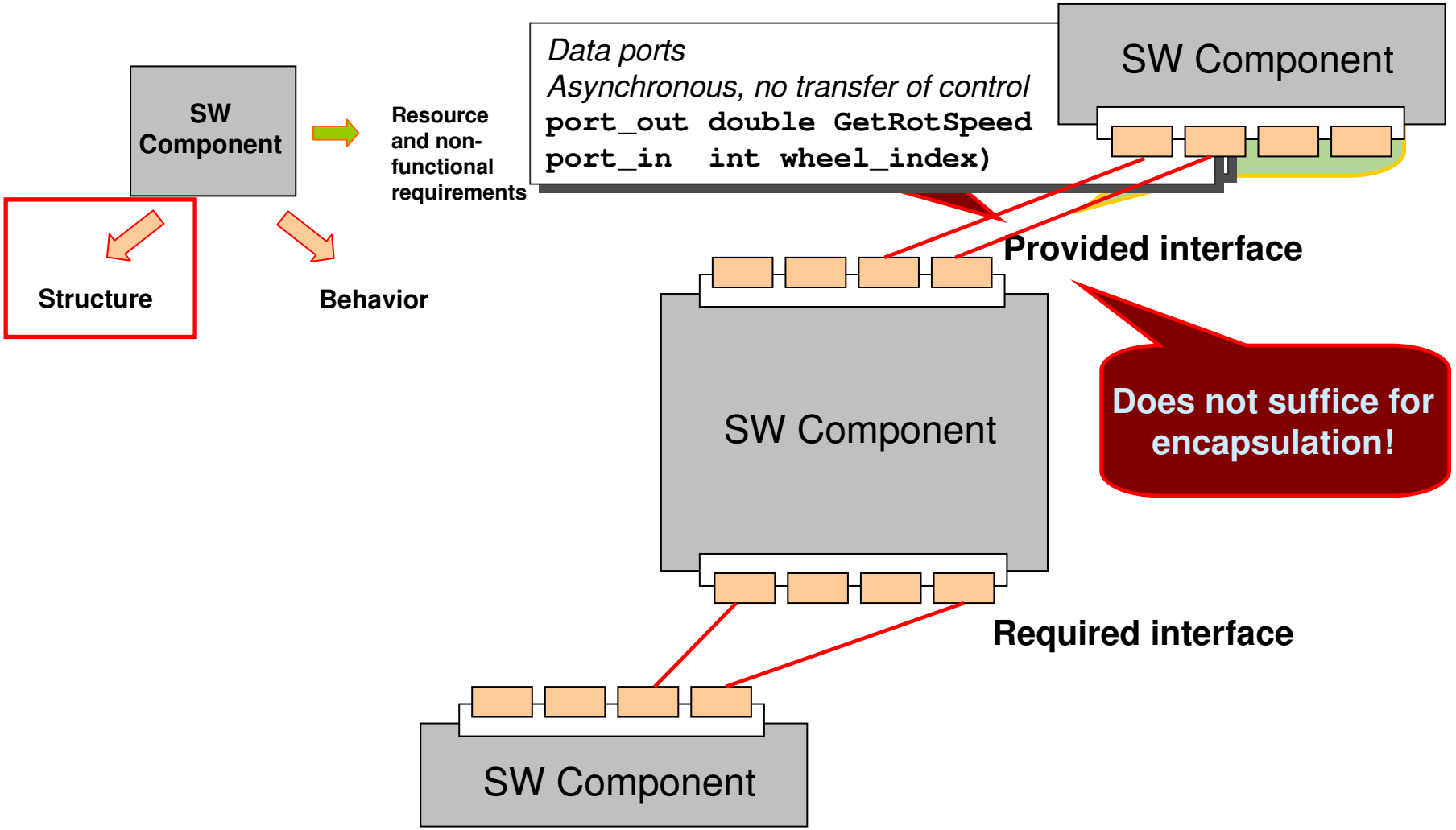
A reusable self-contained artefact implementing a function with given properties



Component-oriented design

Component structure

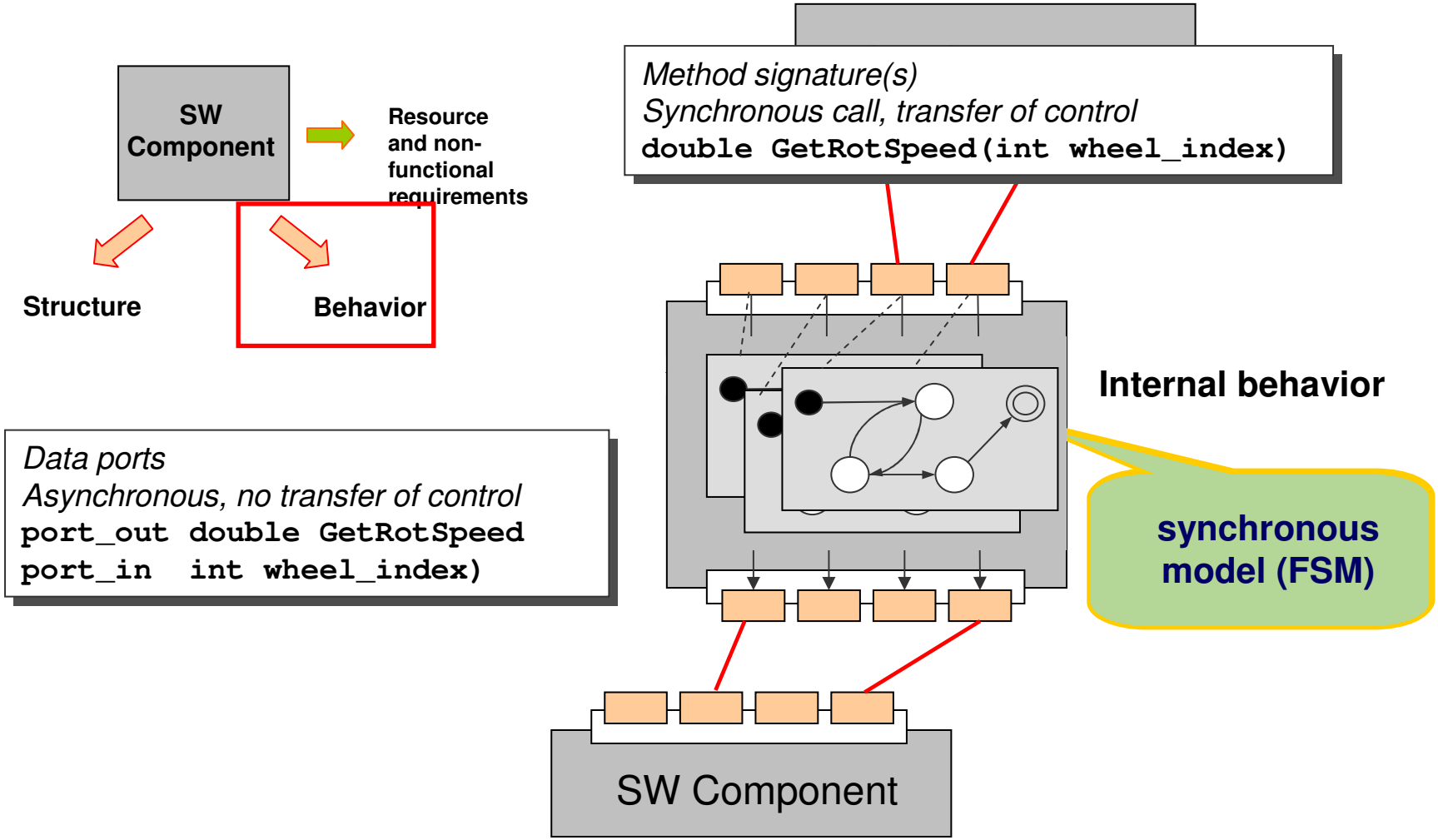
– Key concepts: information hiding and encapsulation



Component-oriented design

Component structure

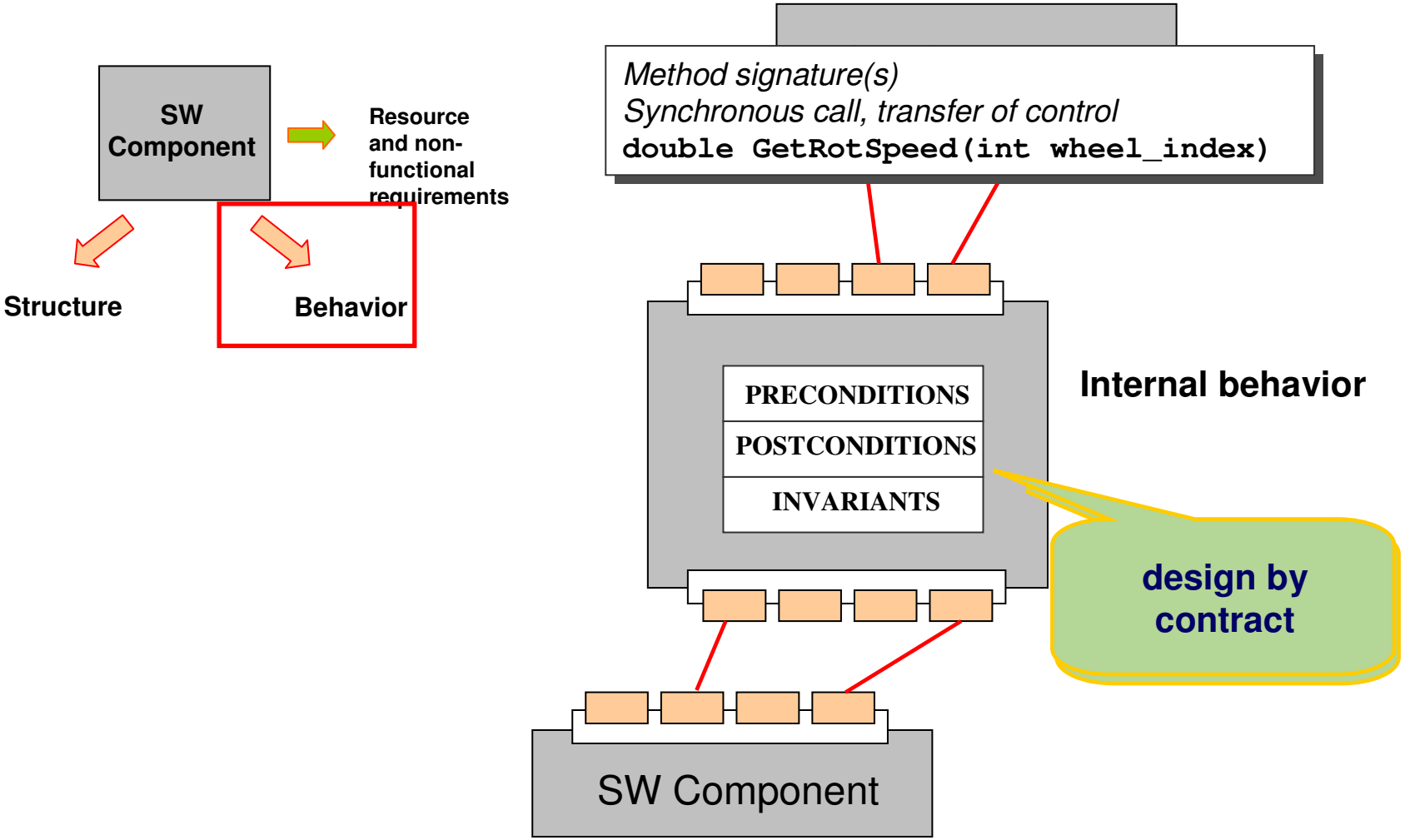
– Key concepts: information hiding and encapsulation



Component-oriented design

Component structure

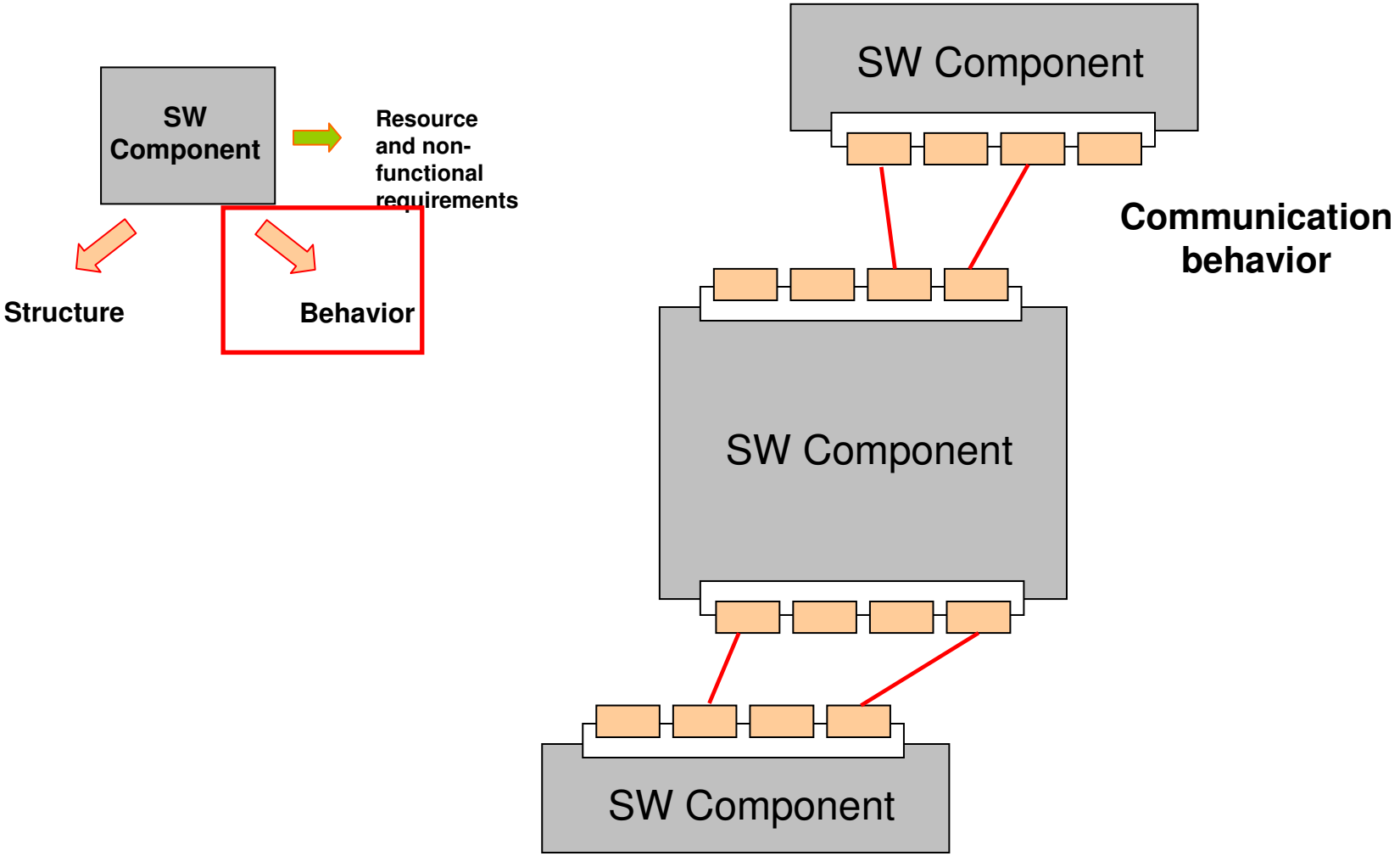
– Key concepts: information hiding and encapsulation



Component-oriented design

Component structure

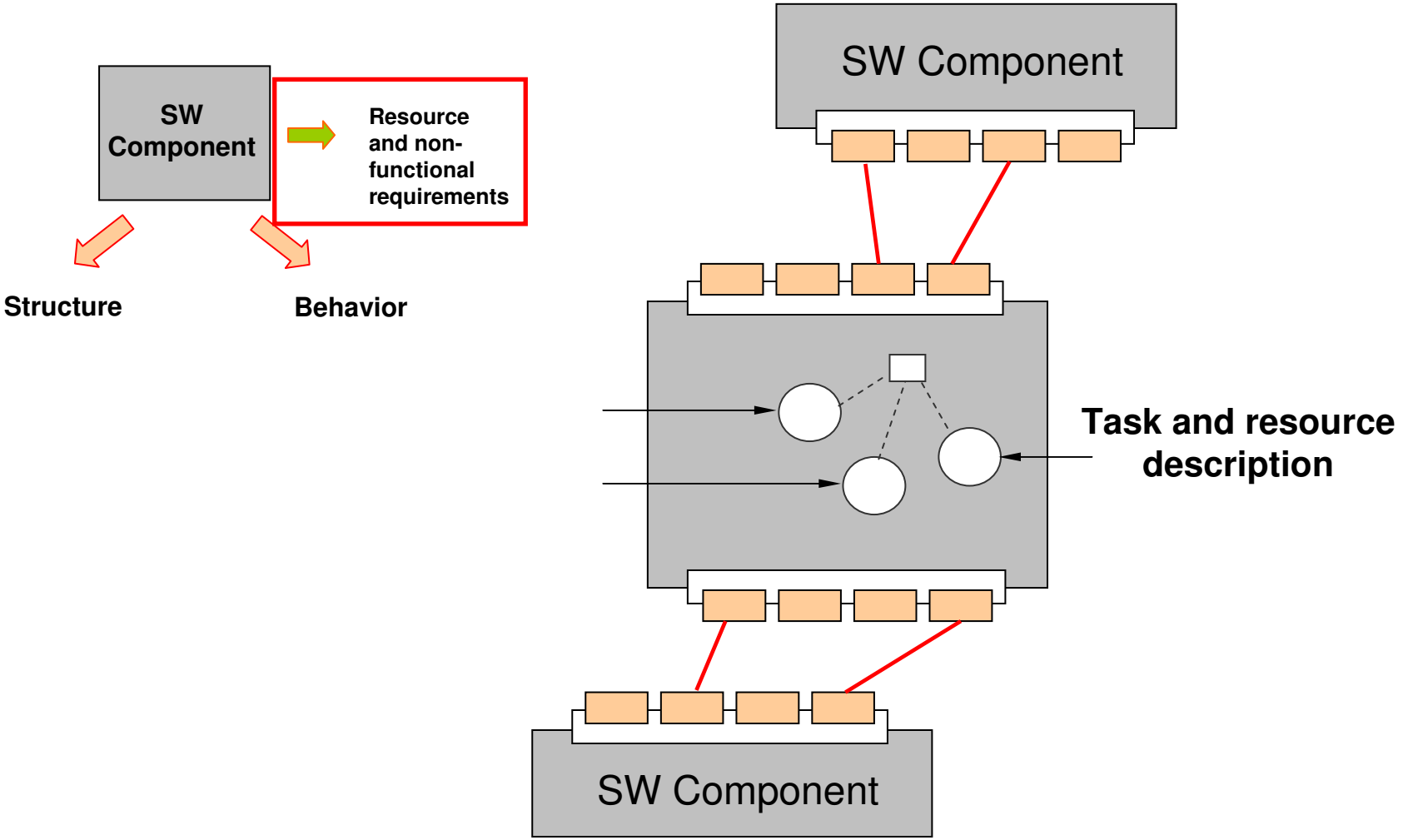
– Key concepts: information hiding and encapsulation



Component-oriented design

Component structure

– Key concepts: information hiding and encapsulation



AUTOSAR Components

Components dependencies are described in form of interfaces and ports, no internal, hidden dependencies may exist.

Therefore, components are in theory exchangeable as long as they implement the same logic and provide the same public communication interface to the remaining system.

Once a component is defined with the help of the software component template, a new component type has been defined. Such a component can be used an arbitrary number of times within the system as well as in different systems.

Components are developed against the virtual functional bus, an abstract communication channel without direct dependency on ECUs and communication busses and they must not directly call the operating system or the communication hardware.

- As a result, they are transferable and can be deployed to ECUs very late in the development process.

Components, Ports and Interfaces

A component has well-defined ports, through which the component can interact with other components.

A port always belongs to exactly one component and represents a point of interaction between a component and other components.

To define the services or data that are provided on or required by a port of a component, the AUTOSAR Interface concept is introduced.

The AUTOSAR Interface can be

- **Client-Server** Interface defining a set of operations that can be invoked
- **Sender-Receiver** Interface, for data-oriented communication

Components, Ports and Interfaces

A port can be

- PPort (provided interface)
- RPort (required interface)

When a PPort *provides* an interface, the component to which the port belongs

- provides an implementation of the operations defined in the Client-Server Interface
- generates the data described in a data-oriented Sender-Receiver Interface.

When an RPort of a component *requires* an AUTOSAR Interface, the component can

- invoke the operations when the interface is a Client-Server
- read the data elements described in the Sender-Receiver Interface.

Communication Patterns: summary

elementary communication patterns

- Client-Server
- Sender-Receiver

Interfaces specify

- what information sender receiver communication transports
- which services with which arguments can be called by client-server communication

The formal description of the interface is in the software component template, including also data types that can be used and interface compatibility.

The detailed behavior of a basic communication pattern is specified by attributes. With those attributes e.g. the length of data queues and the behavior of receivers (blocking, non-blocking, etc.) and senders (send cyclic, etc.) can be defined.

Client-server communication

The server is a provider and the client is a user of a service.

The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server waits for incoming communication requests from a client, performs the requested service and dispatches a response to the client's request.

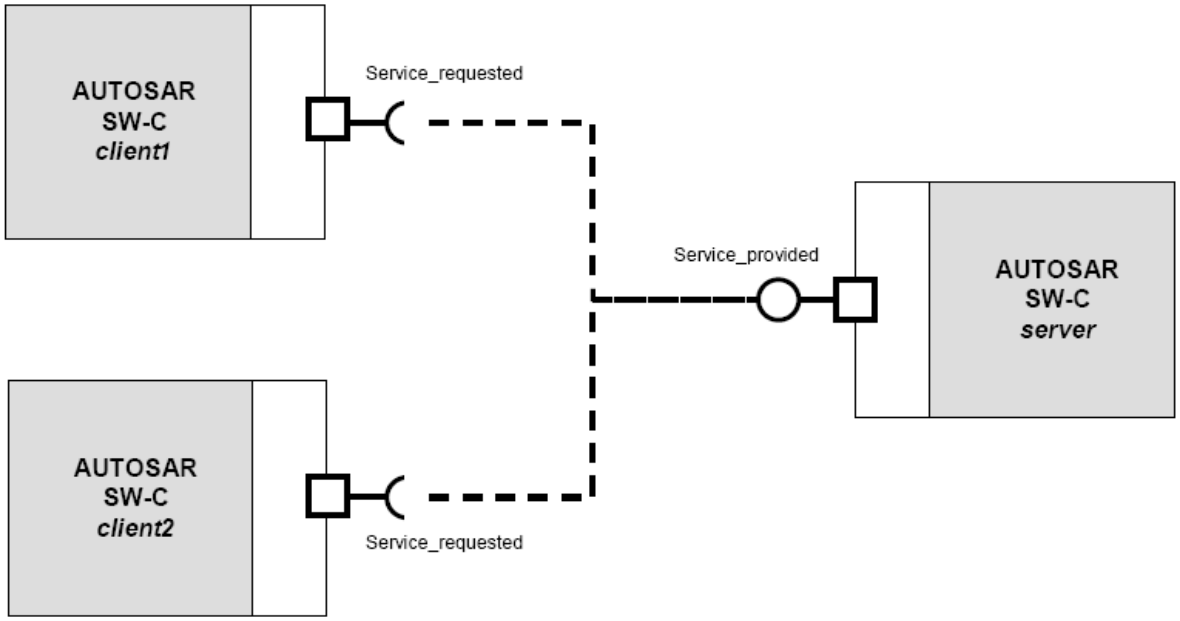
The direction of initiation is used to categorize whether an AUTOSAR Software Component is a client or a server. A single component can be both a client and a server depending on the software realization.

After the service request is initiated and until the response of the server is received The client can be

- blocked (synchronous communication)
- non-blocked (asynchronous communication).

Client-server communication: notation

An example of client-server communication in the composition of three software components and two connections in the VFB model view.



Sender-receiver communication

Model for the asynchronous distribution of information where a sender distributes information to one or several receivers.

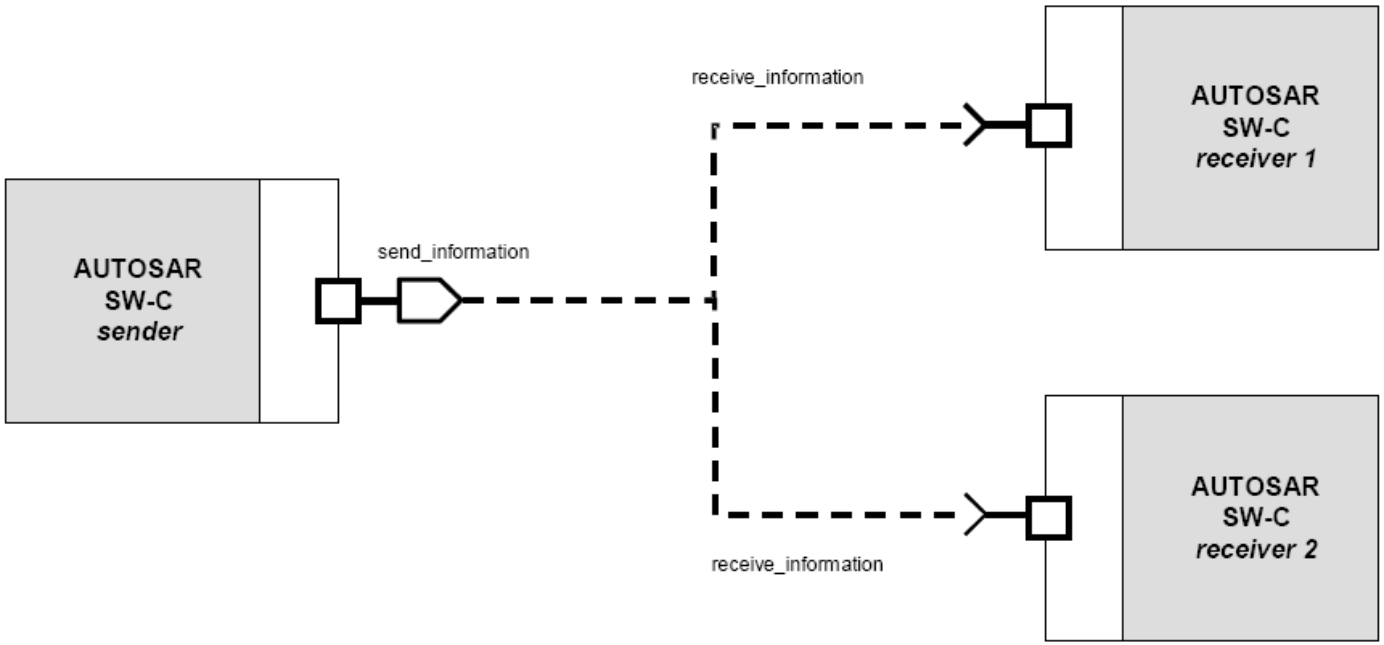
The sender is not blocked (asynchronous communication) and neither expects nor gets a response from the receivers (data or control flow), the sender just provides the information and the receivers decides autonomously when and how to use it

It is the responsibility of the communication infrastructure to distribute the information.

The sender does not know the identity or the number of receivers

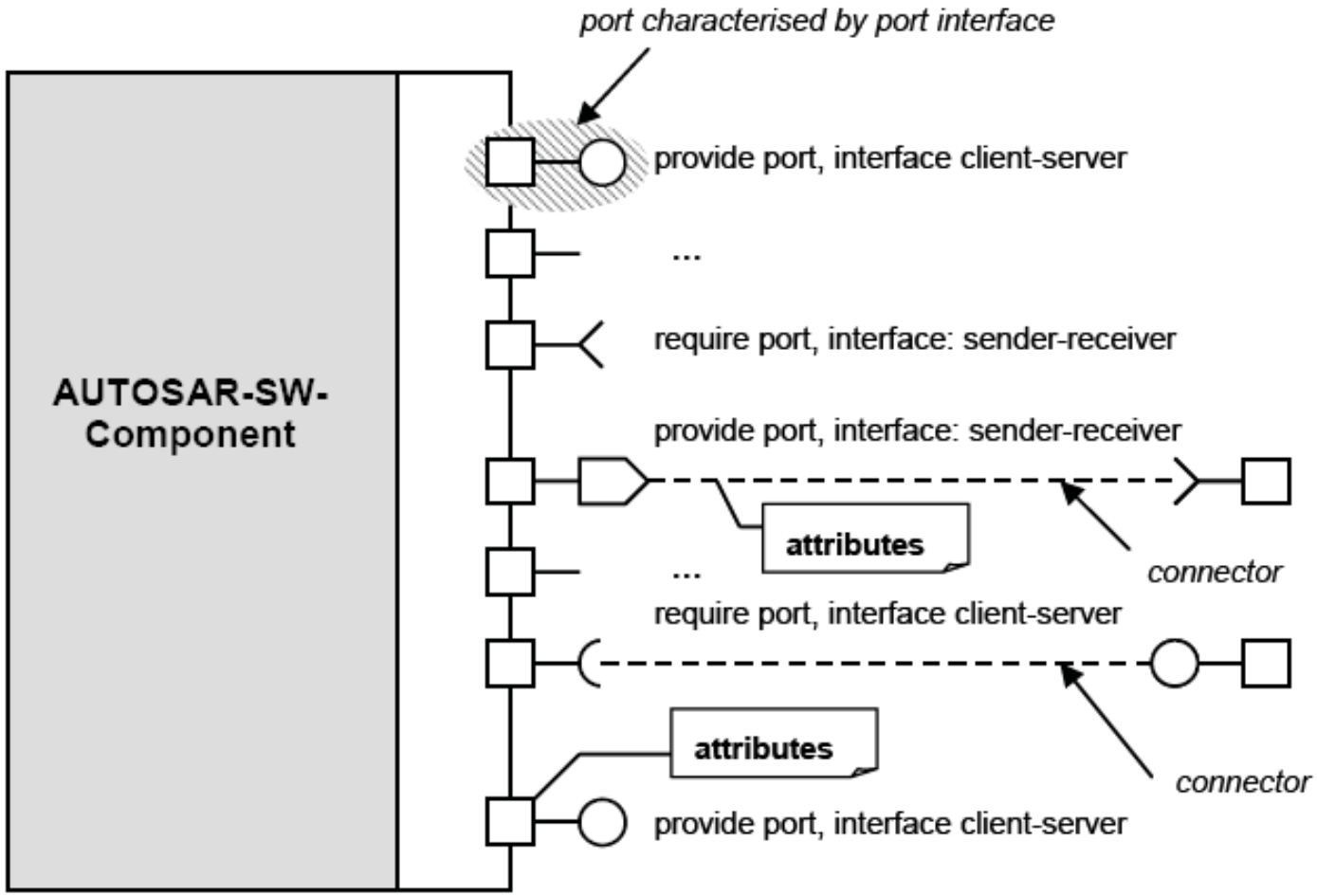
Sender-receiver communication

an example of how sender-receiver communication is modeled in AUTOSAR



AUTOSAR Components, Interfaces and ports

The visual representation of components, ports and interfaces in a component model



AUTOSAR Components: connections

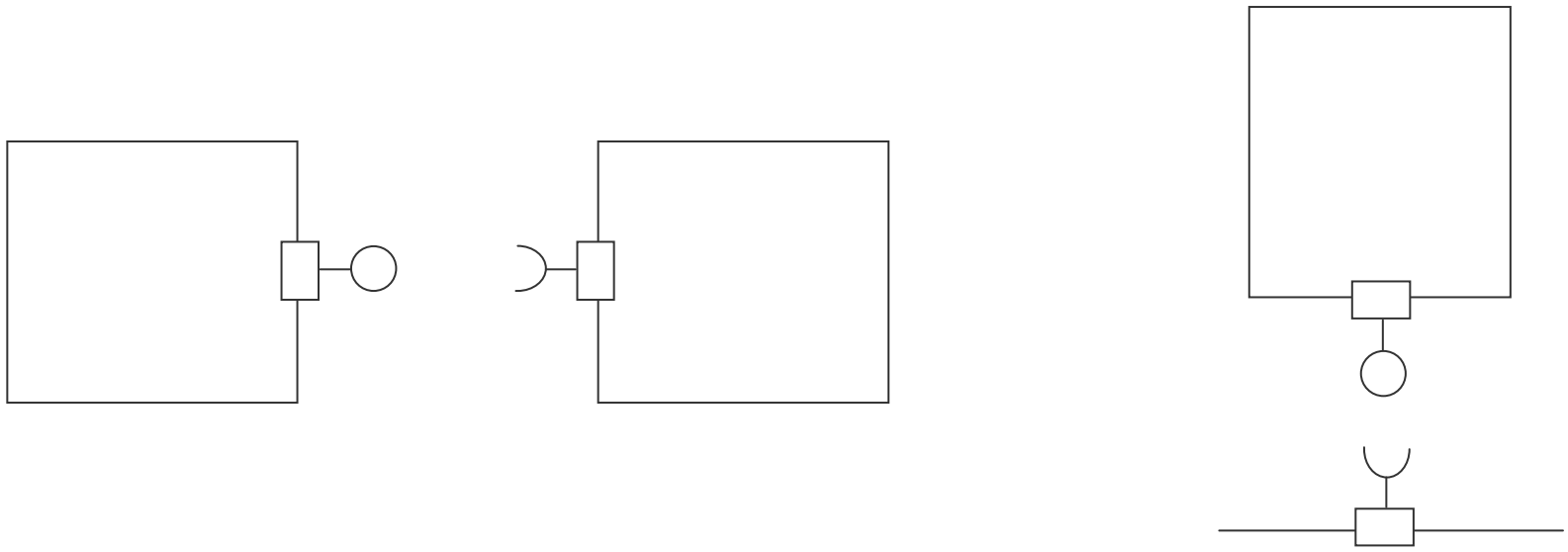
Two components are eventually connected by hooking up a p-port of one component to a compatible r-port of the other component.

- The model shows that it is allowed to define a component without any ports, which seems counterintuitive, this is only reasonable for components that are totally self-contained.

Whether ports are actually compatible is described by a “PortInterface”. As will be shown later, these port interfaces declare services or data elements that are required and provided by the respective ports.

AUTOSAR Components: connections

A port interface has a single attribute called “isService”. This flag indicates, whether service or data described in the interface is actually provided by AUTOSAR services instead of another AUTOSAR component.



AUTOSAR Components: communication behavior

AUTOSAR software components communicate via the Virtual Functional Bus. They need ways to express requirements and capabilities with respect to exchanging data, which is currently possible through two kinds of attributes:

Communication attributes, allow specifying parameters of the communication that affect the generation of the RTE or the actual communication taking place at runtime. An example for such an attribute is the aforementioned transfer time over a connector.

Application level attributes, allow describing properties of exchanged data that do *not* affect the RTE generation, but are indicate to the developer how data needs to be processed. An example for this kind of attribute is a flag, whether data is “filtered” or “raw”..

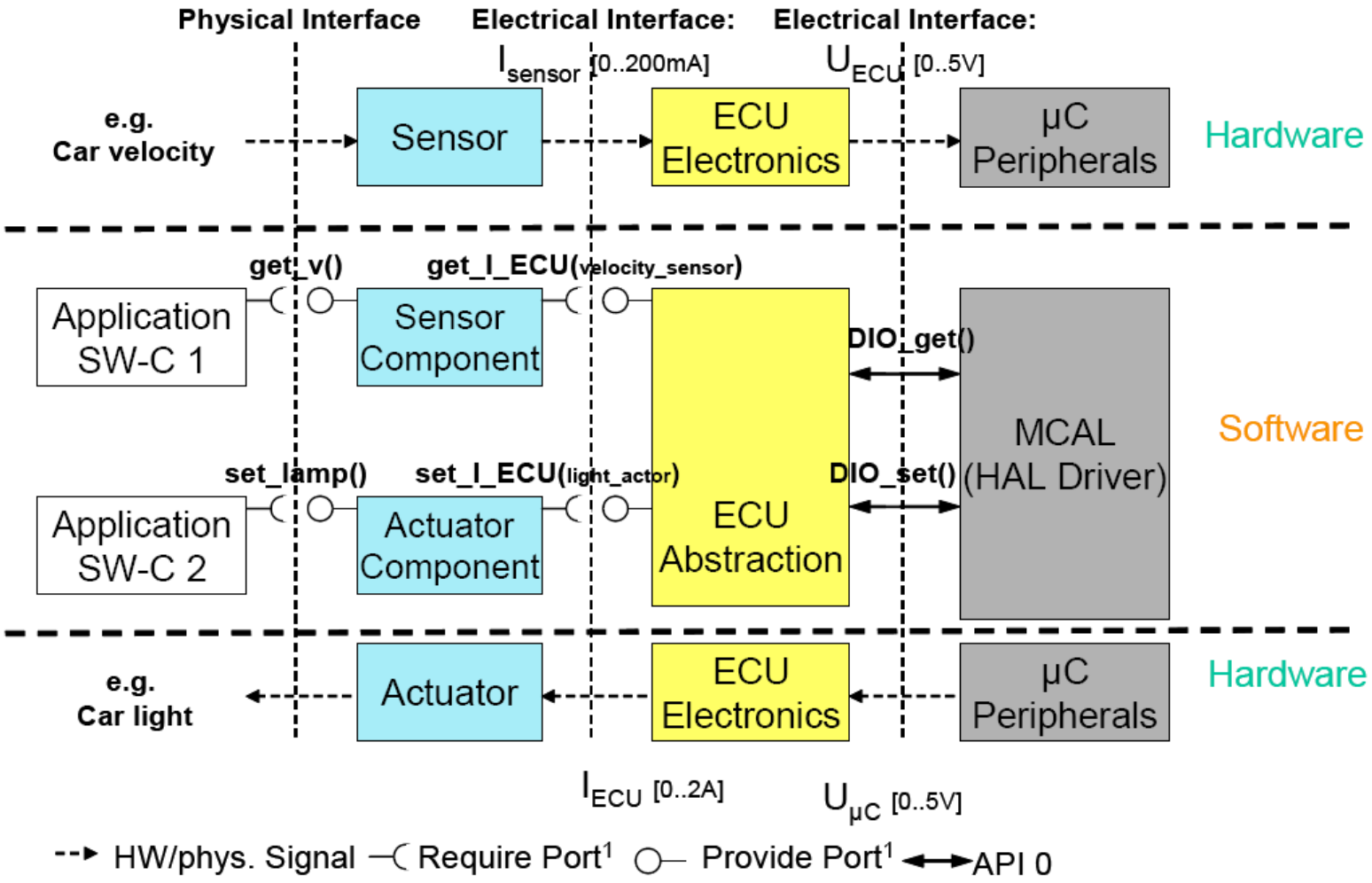
AUTOSAR Components: Sensor/Actuator Components

Sensor/Actuator Components are special AUTOSAR Software Components which encapsulate the dependencies of the application on specific sensors or actuators.

The AUTOSAR infrastructure takes care of hiding the specifics of the microcontroller (this is done in the MCAL, the microcontroller abstraction layer, which is part of the AUTOSAR infrastructure running on the ECU) and the ECU electronics (this is handled by the ECU-Abstraction which is also part of the AUTOSAR Basic Software).

AUTOSAR Components: Sensor/Actuator Components

Typical conversion process from physical signals to software signals (e.g. car velocity) and back (e.g. car light).



AUTOSAR Components: Sensor/Actuator Components

The AUTOSAR infrastructure does NOT hide the specifics of sensors and actuators.

The dependencies on a specific sensor and/or actuator are dealt with in "Sensor/Actuator Software Component", which is a special kind of Software Component. Such a component is independent of the ECU on which it is mapped but is dependent on a specific sensor and/or actuator for which it is designed.

- For example, a "Sensor Component" typically inputs a software representation of the electrical signal at an input-pin of the ECU (e.g. a current produced by the sensor) and outputs the physical quantity measured by the sensor (e.g. the car speed).

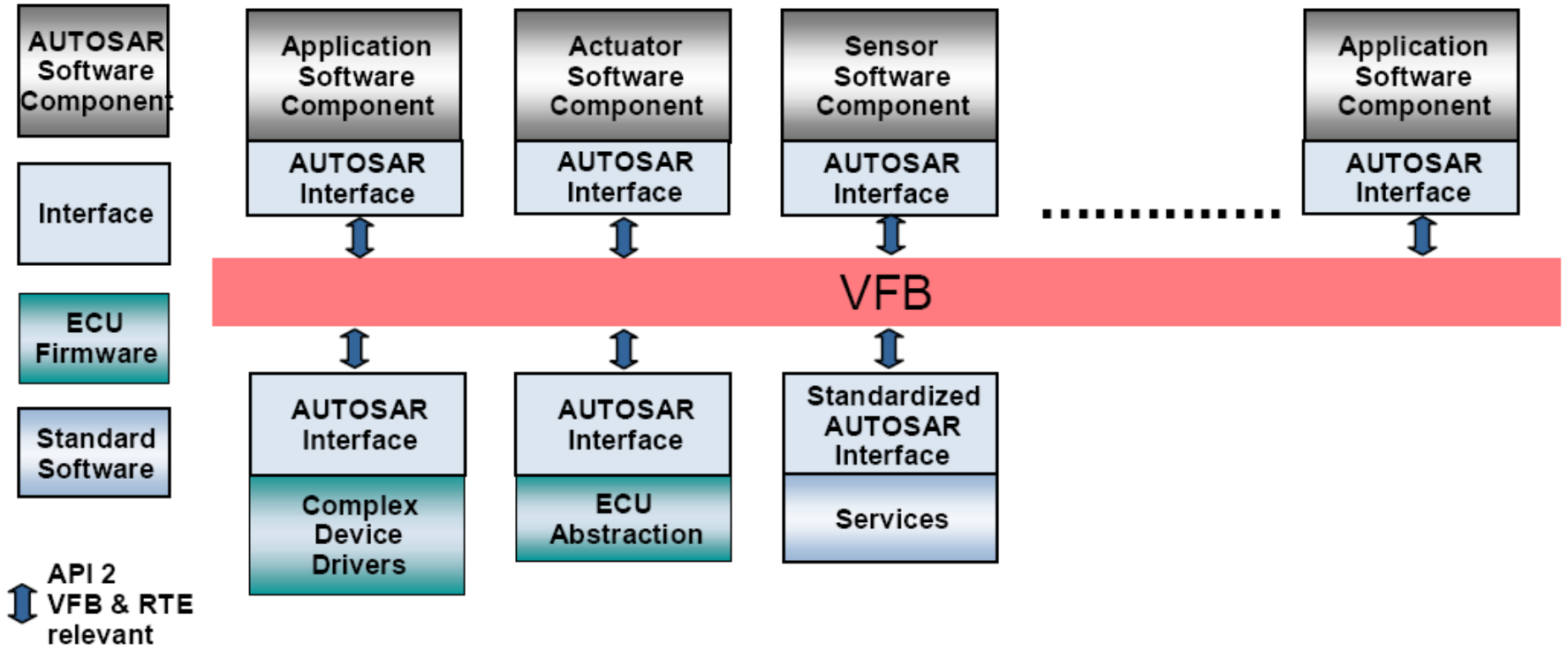
Typically, because of performance issues, such components will need to run on the ECU to which the sensor/actuator is physically connected.

The Virtual Function Bus (VFB)

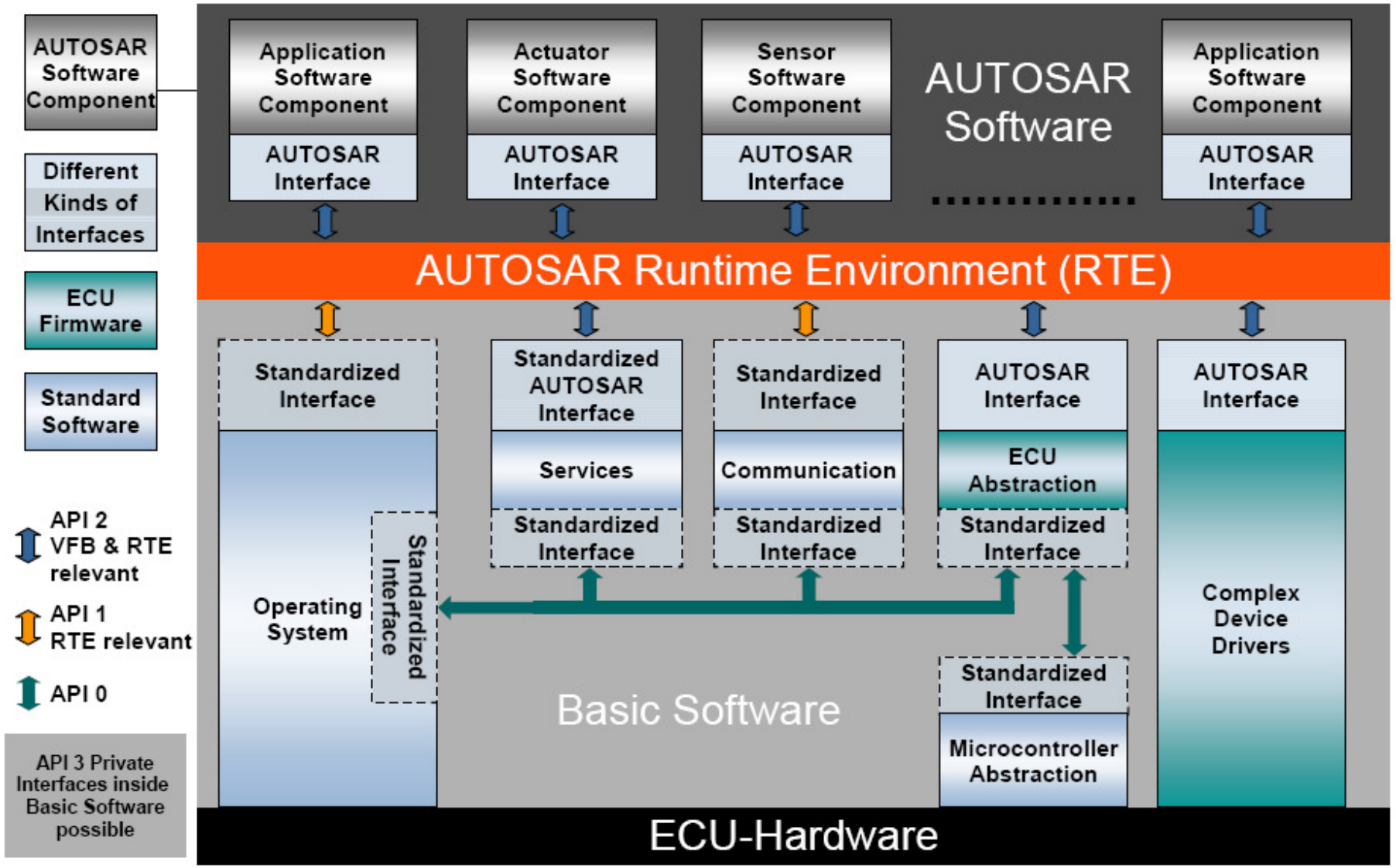
The virtual functional bus is the abstraction of the AUTOSAR Software Components interconnections of the entire vehicle. The communication between different software components and between software components and its environment (e.g. hardware driver, OS, services, etc.) can be specified independently of any underlying hardware (e.g. communication system). The functionality of the VFB is provided by communication patterns.

The Virtual Function Bus (VFB)

From the VFB view ports of AUTOSAR Software Components, Complex Device Drivers, the ECU Abstraction and AUTOSAR Services can be connected. Complex Device Drivers, the ECU Abstraction and AUTOSAR Services are part of the Basic Software.



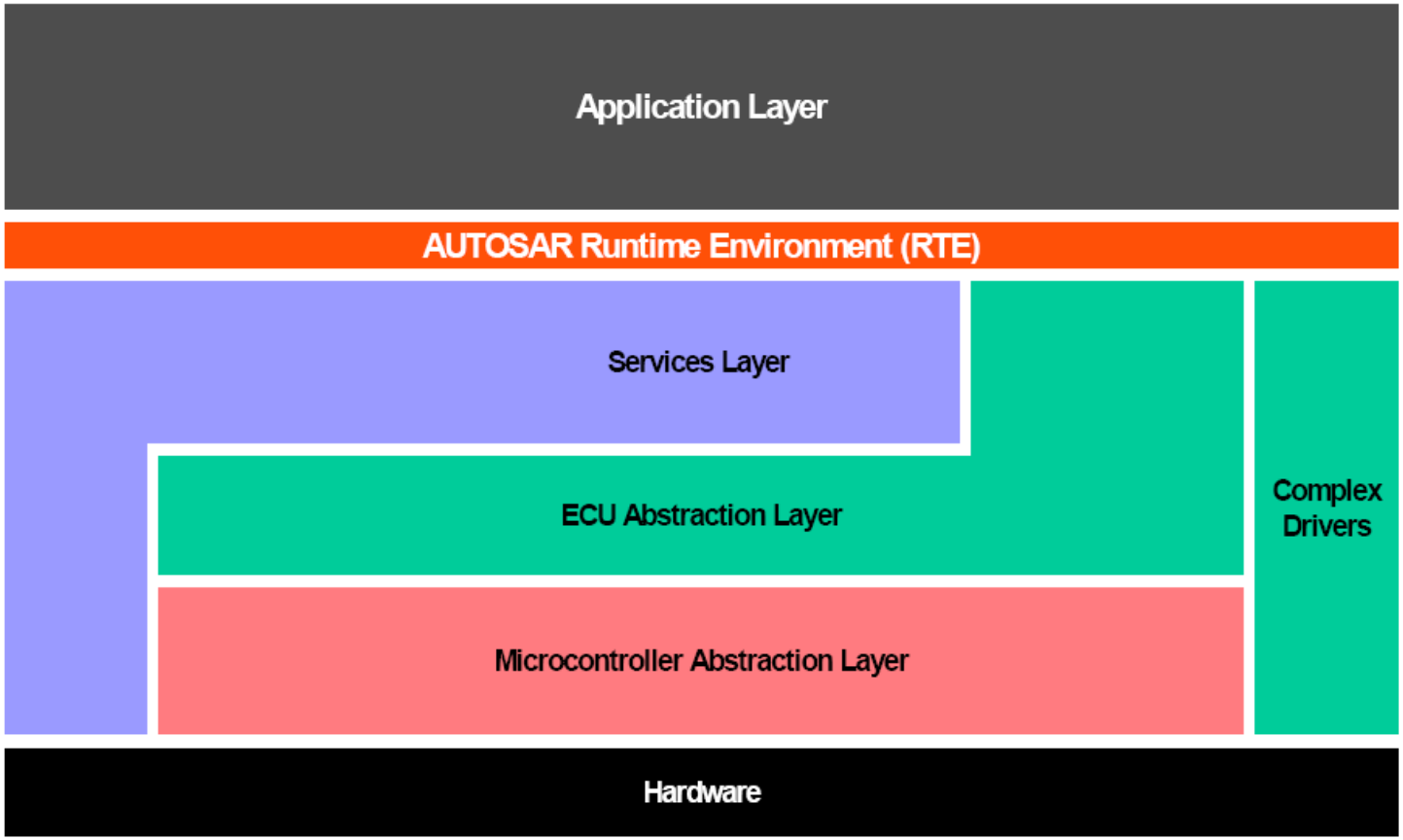
ECU SW Architecture



ECU Architecture - Layered Software Architecture

A layered architecture has been developed within AUTOSAR to enable a clear and structured interface definition and a well defined abstraction of the hardware.

The architecture is structured in 5 layers plus the Complex Drivers.



AUTOSAR Architecture Layers

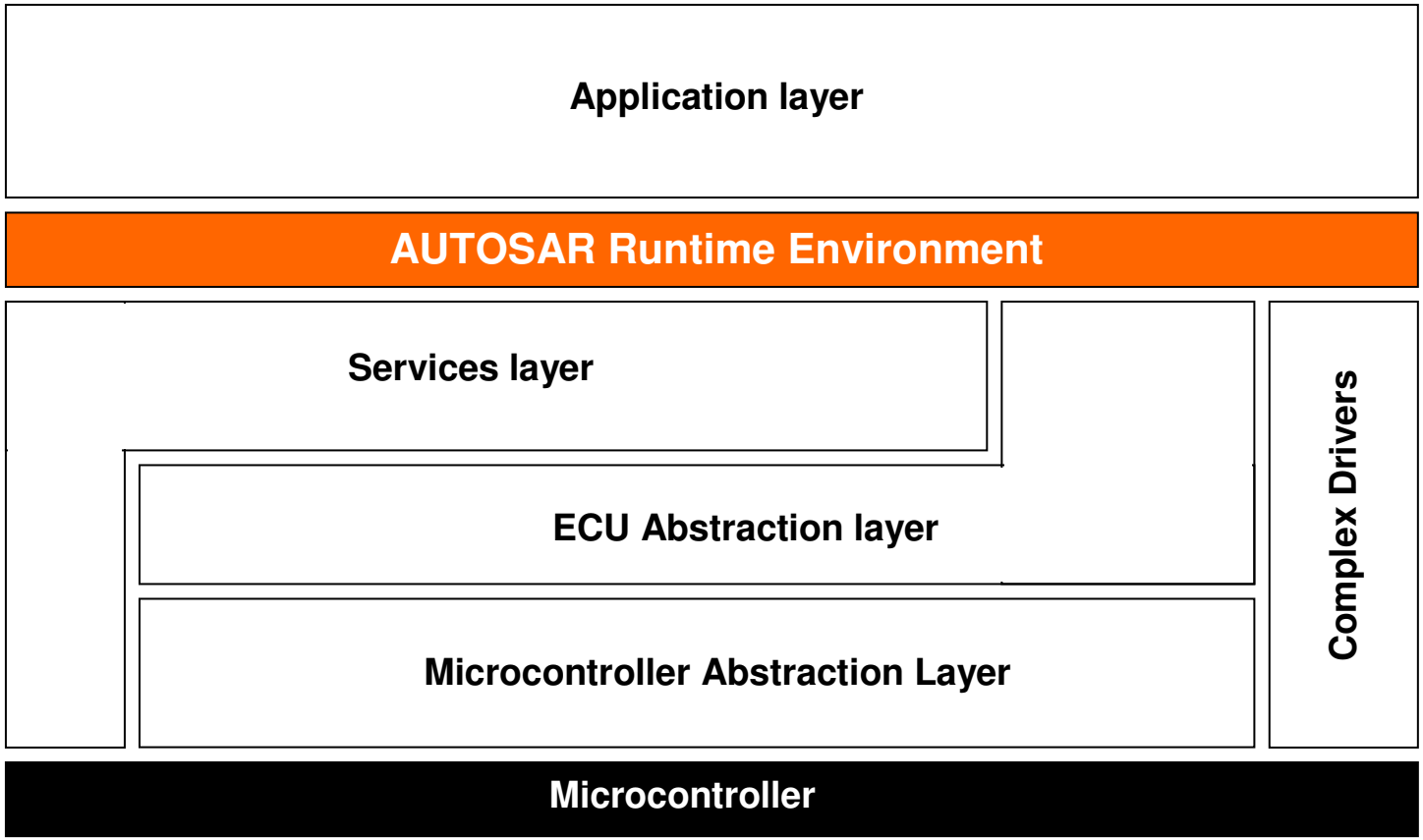
AUTOSAR Software

This layer consists of AUTOSAR Software Components that are mapped on the ECUs.

All interaction between AUTOSAR Software Components is routed through the AUTOSAR Runtime Environment. The AUTOSAR Interface specification assures the connectivity.

AUTOSAR Runtime Environment

The AUTOSAR Runtime Environment (RTE) acts as a system-level communication center for inter- and intra-ECU information exchange.



AUTOSAR Runtime Environment

The RTE is the runtime representation of the Virtual Function Bus *for a specific ECU*.

The RTE provides a communication abstraction to AUTOSAR Software Components providing the same interface and services for inter-ECU (using CAN, LIN, Flexray, MOST, etc.) or intra-ECU communication.

As the communication requirements of the software components are application dependent, the RTE needs to be tailored. It is therefore very likely, that the main parts of RTE will be generated and tailored to provide desired communication services while still being resource-efficient. Thus, the RTE will likely differ between one ECU and another.

- The RTE is typically tool-generated and statically configured

AUTOSAR Runtime Environment

The AUTOSAR Runtime Environment has the responsibility to provide a uniform environment to AUTOSAR Software Components to make the implementation of the software components independent from communication mechanisms and channels.

The RTE achieves this by mapping the communication relationships between components, that are specified in the different templates, to a specific intra-ECU communication mechanism, such as a function call, or an inter-ECU communication mechanism, such as a COM message which leads to CAN communication.

AUTOSAR Runtime Environment

Access to ports from a software component implementation

The implementation of an AUTOSAR Software Component is not allowed to use the communication layer, for example OSEK COM, directly.

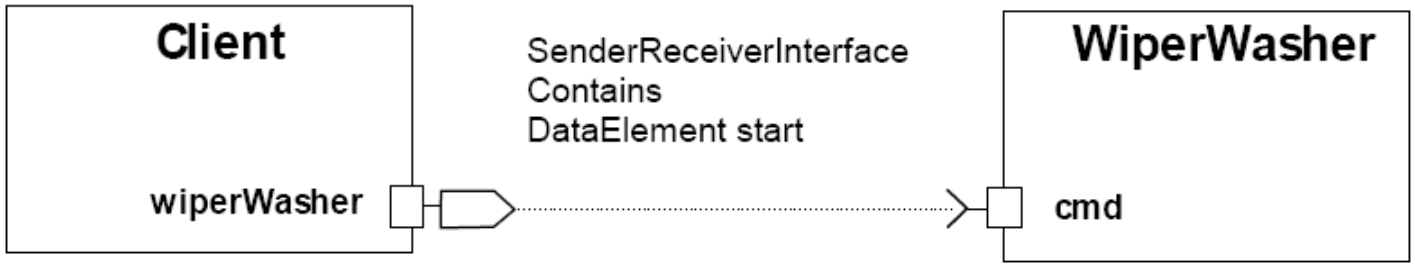
To communicate with other software components it uses ports and client-server communication or sender-receiver communication.

The RTE generator is responsible for creating the appropriate language-dependant APIs based on the definition of the interface of the component in the Software Component Template.

The API has to be the same independent from the mapping of the components, i.e. the component's code must not be changed when the mapping is changed.

The API names are derived from the XML files and conform to a naming convention.

AUTOSAR Runtime Environment



Implementation of Client Runnable run1:

```
Rte_Runnable_run1() {
        …
        Rte_Write_wiperWasher_start(…);
        …
}
```

Implementation of WiperWasher Runnable run1:

```
Rte_Runnable_run1() {
        …
        v = Rte_Read_cmd_start(…);
        …
}
```

AUTOSAR Runtime Environment

Implementation of connectors

The RTE generator is also responsible for generating code, which implements the connectors between the ports.

This generated code is dependant on the mapping of the software components to ECUs.

If the connector connects two components on the same ECU a local communication stub can be generated.

Otherwise, a stub that uses network communication must be generated.

Intra-ECU connector

```
Rte_Write_Client_wiperWasher_start(…) {
        modify variable
}
```

Inter-ECU connector

```
Rte_Write_Client_wiperWasher_start(…) {
        access COM
}
```

AUTOSAR Runtime Environment

The mapping from a connector to a communication stub must conserve the semantics of this connector independent from the used communication medium.

The communication stub is also responsible for parameter marshalling. This includes serializing complex data to a byte stream. But endian conversion (if any is necessary) is delegated to the communication module of the Basic Software.

Lifecycle management

The RTE is responsible for the lifecycle management of AUTOSAR Software Components. It has to invoke startup and shutdown functions of the software component.

AUTOSAR Runtime Environment

Access to Basic Software

An AUTOSAR Software Component is not allowed to access Basic Software directly.

Firstly, the access to services, to the ECU abstraction, or to Complex Device Drivers is abstracted via ports and AUTOSAR interfaces.

With respect to the component implementation, the RTE provides appropriately generated APIs for Basic Software access.

Multiple Instantiations of software components

The RTE shall support multiple instantiations of software components. The basic intention of multiple instantiation is to avoid code duplication if possible. Furthermore different private states of multiple instances shall be supported.

- but in AUTOSAR R2.0 this feature is cancelled !

AUTOSAR Architecture Layers

AUTOSAR Basic Software

Basic Software is the standardized software layer, which provides services to the SW Components. It does not fulfill any functional job and is situated below the AUTOSAR Runtime Environment. It contains

Standardized components

- **Services** including diagnostic protocols; NVRAM, flash and memory management.
- **Communication** the communication framework (e.g. CAN, LIN, FlexRay...), the I/O management, and the network management.

ECU specific components.

- Operating system
- Microcontroller abstraction
- Complex Device Drivers

AUTOSAR – Basic software: Operating system

AUTOSAR includes requirements for an AUTOSAR Operating System. The OS

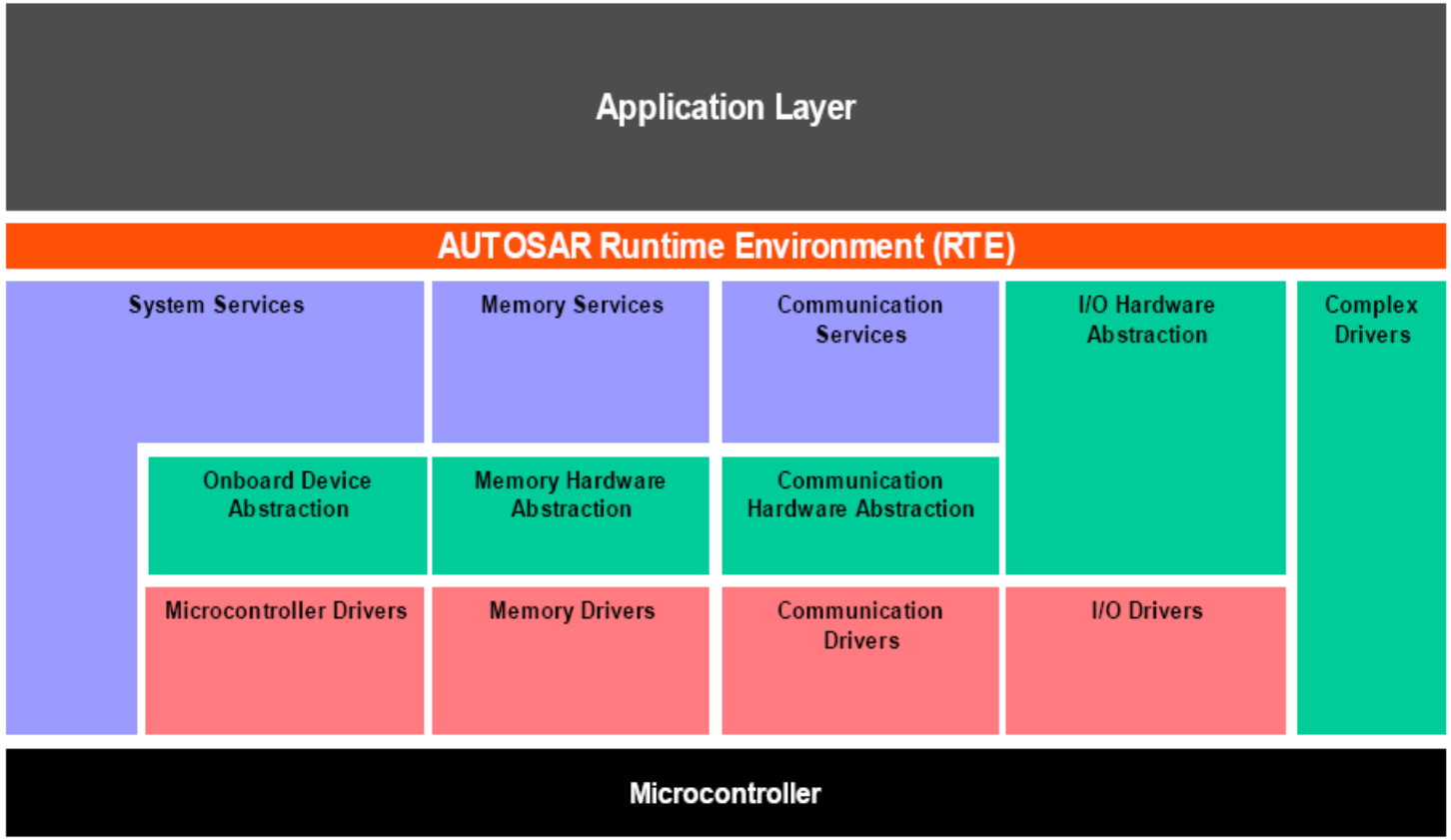
- is configured and scaled statically,
- is amenable to reasoning of real-time performance, provides a priority-based scheduling, provides protective functions at run-time, and
- is hostable on low-end controllers with and without external resources.

It is assumed that some domains (e.g. telematic/infotainment) will continue to use proprietary OSs. In these cases the interfaces to AUTOSAR components must still be AUTOSAR compliant (the proprietary OS must be abstracted to an AUTOSAR OS).

The standard OSEK OS (ISO 17356-3) is used as the basis for the AUTOSAR OS.

Basic Software structure

The layered architecture has been further refined in the area of Basic Software. Around 80 Basic Software modules have been defined (11 main blocks plus Complex Drivers).



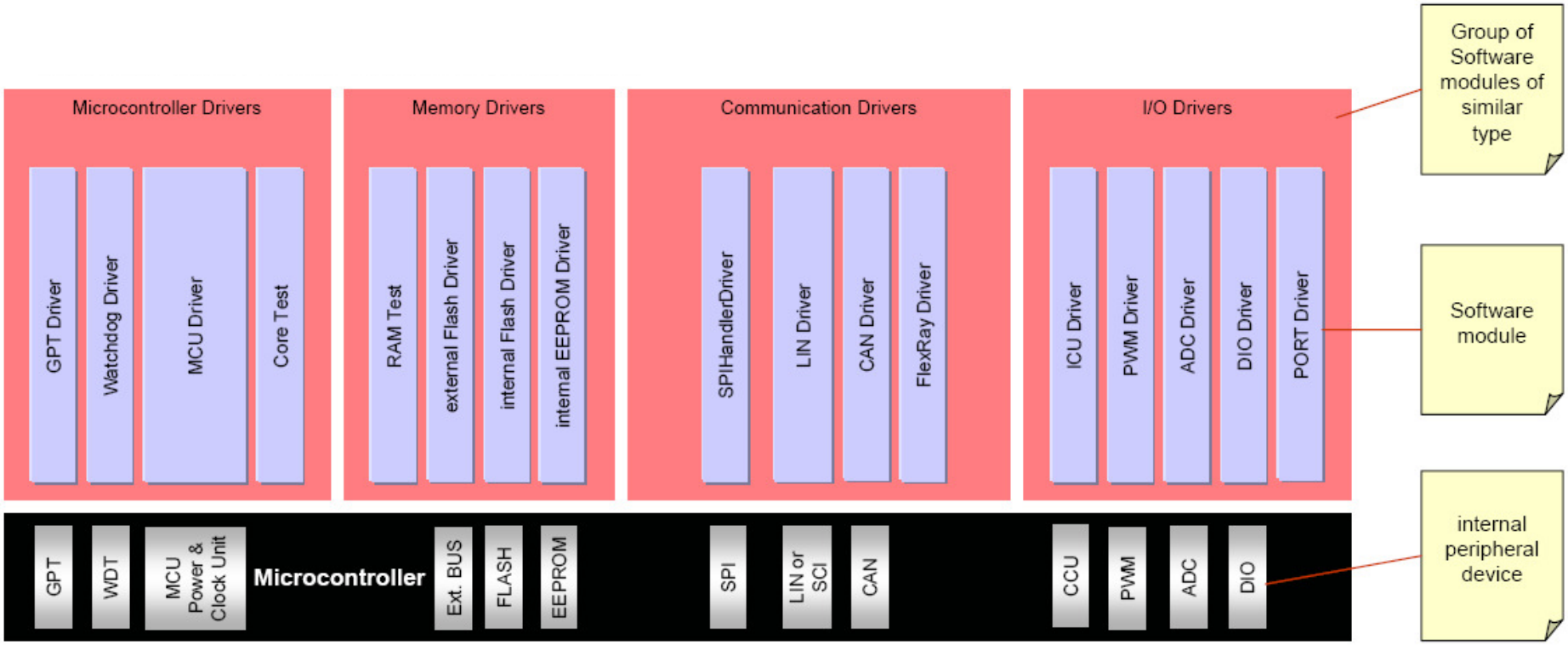
AUTOSAR – Basic software: microcontroller abstr.

Access to the microcontroller registers is routed through the Microcontroller Abstraction layer (MCAL).
MCAL is a hardware specific layer that ensures a standard interface to the Basic Software. It manages the microcontroller peripherals and provides the components of the Basic Software with microcontroller independent values. MCAL implements notification mechanisms to support the distribution of commands, responses and information to processes. It can include

- Digital I/O (DIO),
- Analog/Digital Converter (ADC),
- Pulse Width (De)Modulator (PWM, PWD),
- EEPROM (EEP),
- Flash (FLS),
- Capture Compare Unit (CCU),
- Watchdog Timer (WDT),
- Serial Peripheral Interface (SPI), and
- I2C Bus (IIC).

Microcontroller abstraction layer

The **Microcontroller Abstraction Layer** is the lowest layer of the Basic Software. It contains drivers, with direct access to the µC internal peripherals and memory mapped µC external devices.



Microcontroller abstraction layer

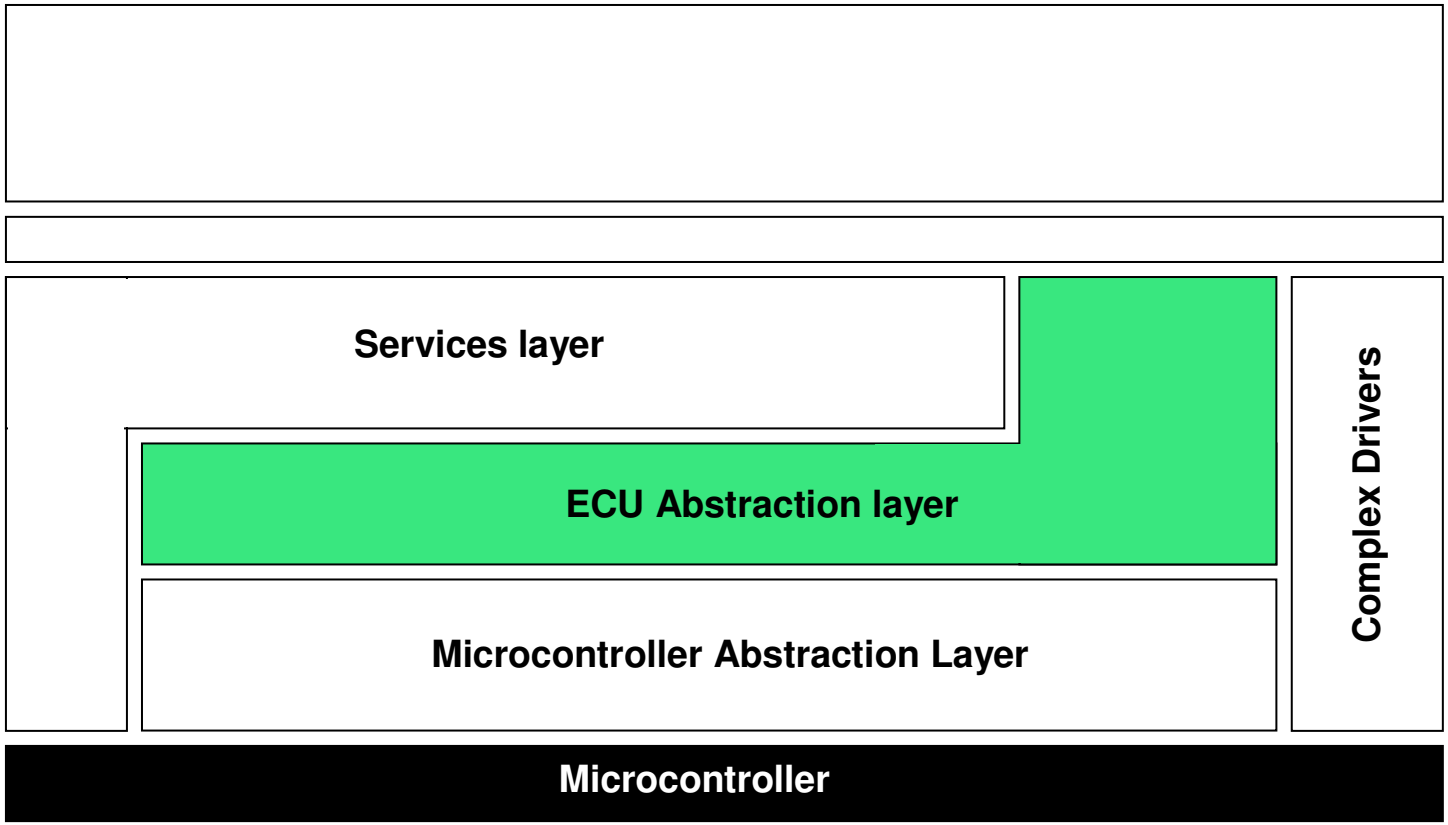
Microcontroller Abstraction Layer

The Microcontroller abstraction layer is subdivided into 4 parts:

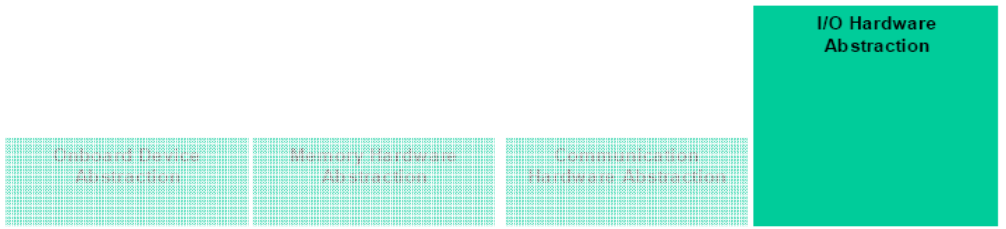
- I/O Drivers
  Drivers for analog and digital I/O (e.g. ADC, PWM, DIO).
- Communication Drivers
  Drivers for ECU onboard (e.g. SPI, I2C) and vehicle communication (e.g. CAN). OSI-Layer: Part of Data Link Layer.
- Memory Drivers
  Drivers for on-chip memory devices (e.g. internal Flash, internal EEPROM) and memory mapped external memory devices (e.g. external Flash).
- Microcontroller Drivers
  Drivers for internal peripherals (e.g. Watchdog, Clock Unit) and functions with direct µC access (e.g. RAM test, Core test).

ECU Abstraction layer

The ECU Abstraction Layer provides a software interface to the electrical values of any specific ECU in order to decouple higher-level software from all underlying hardware dependencies.



ECU Abstraction layer



I/O Hardware Abstraction:

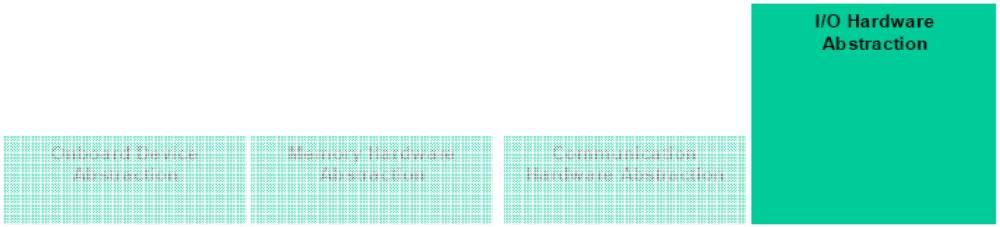
A group of modules which abstracts from the location of peripheral I/O devices (on-chip or on-board) and the ECU hardware layout (e.g. µC pin connections and signal level inversions).

The I/O hardware abstraction does not abstract from the sensors/actuators! I/O devices are accessed via an I/O signal interface.

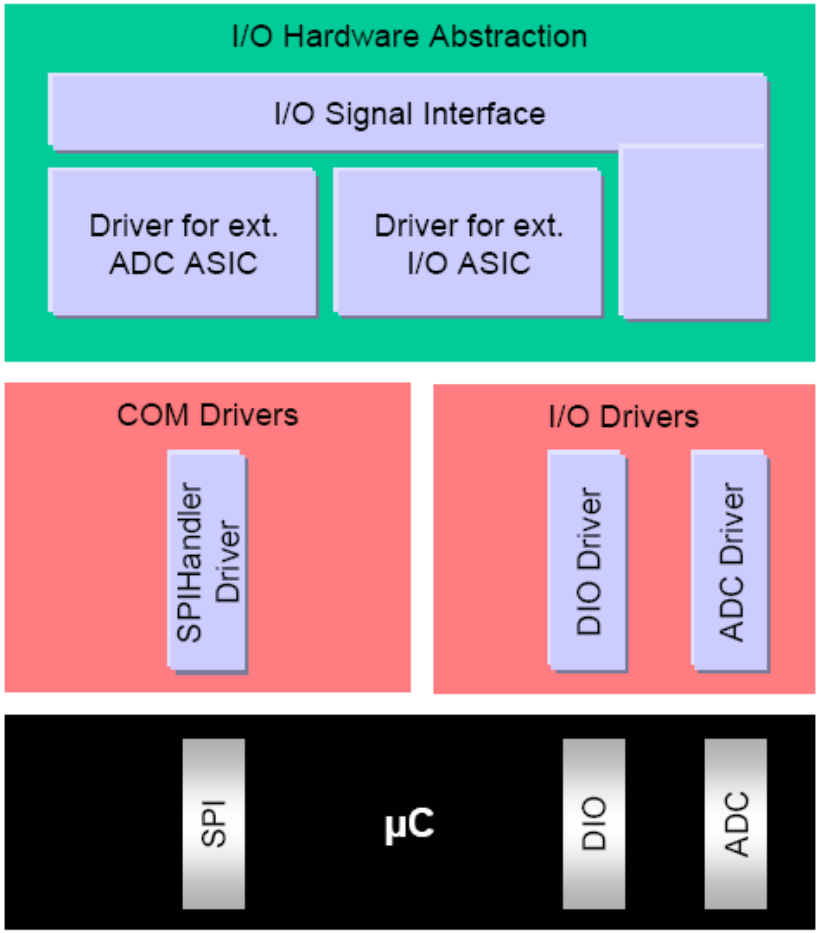
The task of this group of modules is

- to represent I/O signals as they are connected to the ECU hardware (e.g. current, voltage, frequency), and
- to hide ECU hardware and layout properties from higher software layers.

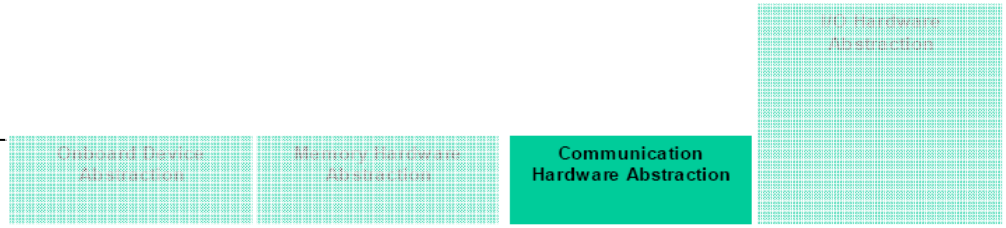
ECU Abstraction layer



An example:



ECU Abstraction layer



Communication HW abstraction

A group of modules which abstracts from the location of communication controllers and the ECU hardware layout. For all communication systems a specific communication hardware abstraction is required (e.g. for LIN, CAN, MOST, FlexRay).

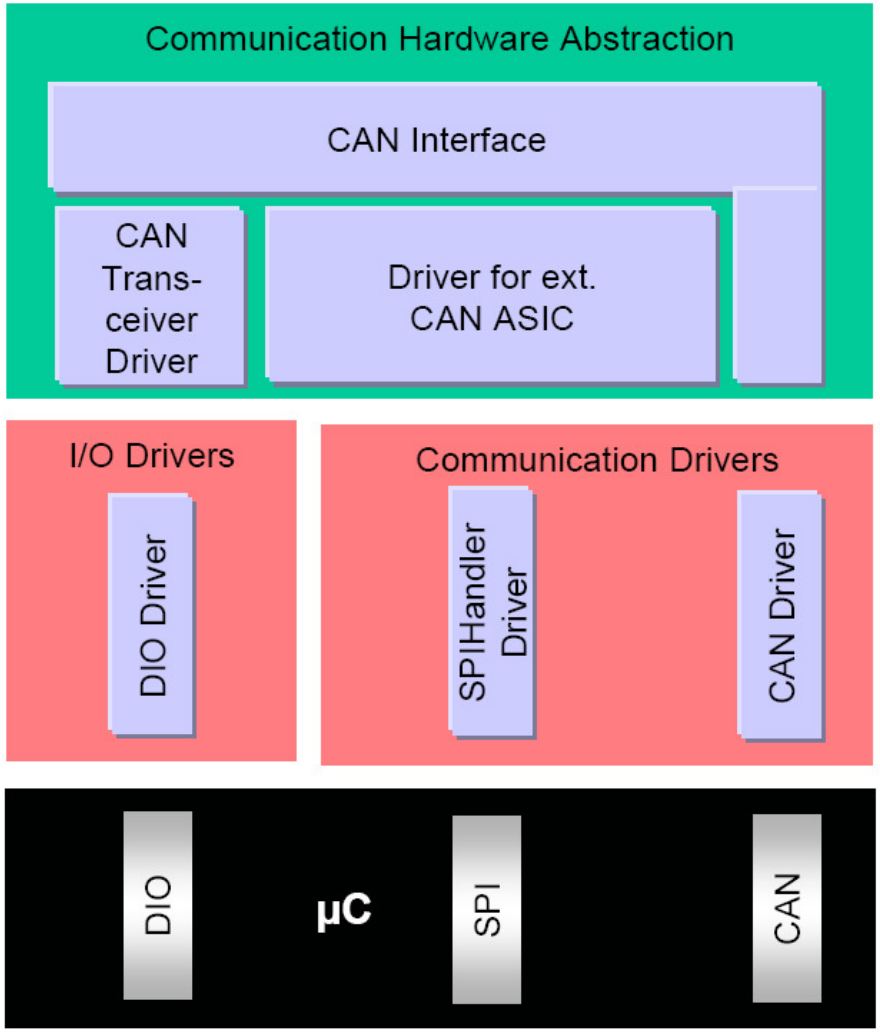
- Example: An ECU has a microcontroller with 2 internal CAN channels and an additional on-board ASIC with 4 CAN controllers. The CAN-ASIC is connected to the microcontroller via SPI.

The communication drivers are accessed via bus specific interfaces (e.g. CAN Interface). That means the access to the CAN controller should be regardless of whether it is located inside the microcontroller, externally to it, or whether it is connected via SPI.

The task of this group of modules is to provide equal mechanisms to access a bus channel regardless of its location (on-chip / on-board).

ECU Abstraction layer

An example:



ECU Abstraction layer

Memory HW Abstraction

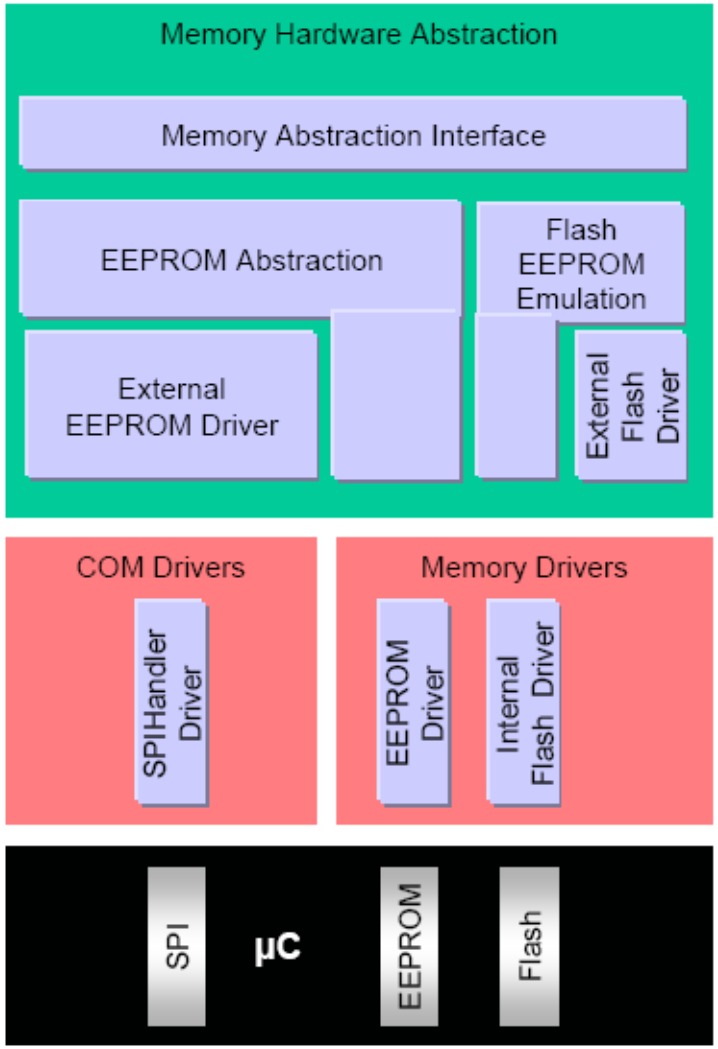
A group of modules which abstracts from the location of peripheral memory devices (on-chip or on-board) and the ECU hardware layout.

- Example: on-chip EEPROM and external EEPROM devices should be accessible via an equal mechanism.

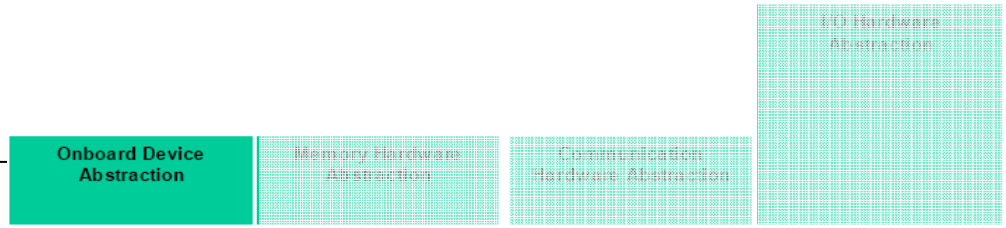
The memory drivers are accessed via memory specific interfaces (e.g. EEPROM Interface). The task of this group of modules is to provide equal mechanisms to access internal (on-chip) and external (onboard) memory devices.

ECU Abstraction layer

An example:



ECU Abstraction layer



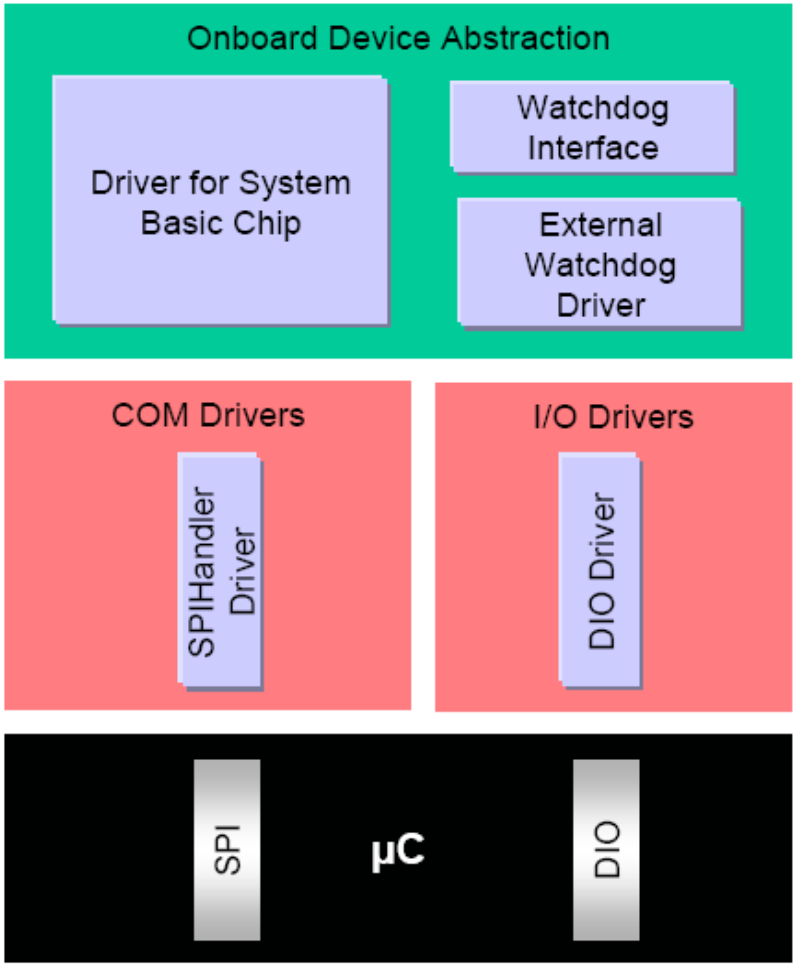
Onboard Device Abstraction

Contains drivers for ECU onboard devices which cannot be seen as sensors or actuators like system basic chip, external watchdog etc. Those drivers access the ECU onboard devices via the µC abstraction layer.

The task of this group of modules is to abstract from ECU specific onboard devices.

ECU Abstraction layer

An example:



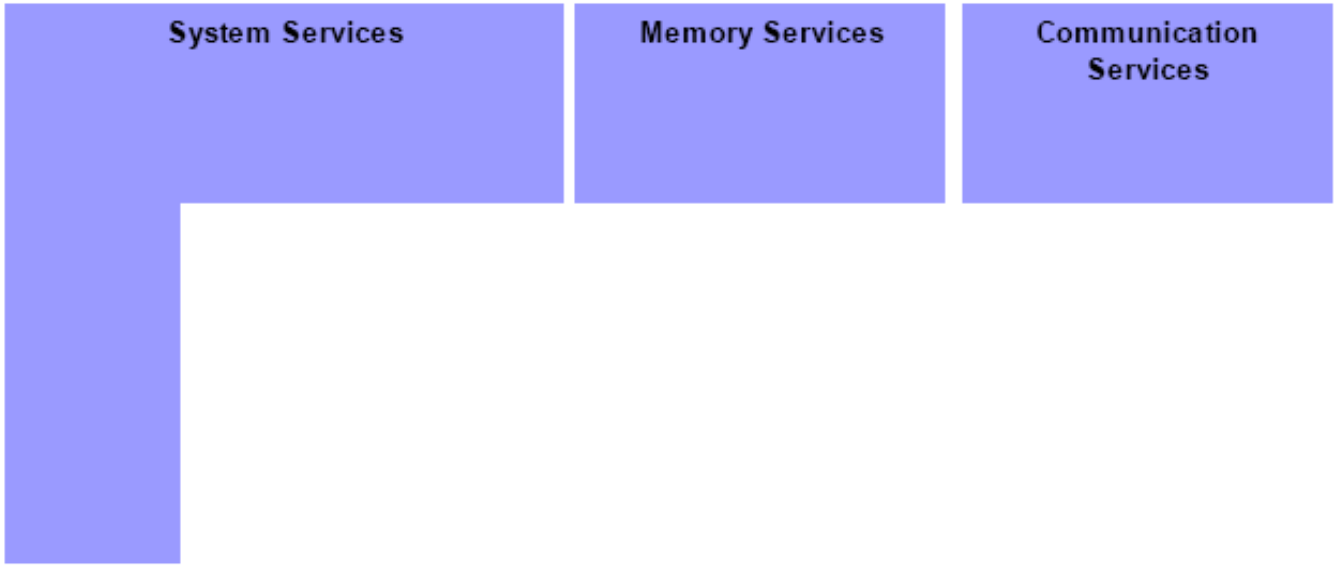
Service layer

The service layer consists out of 3 different parts:

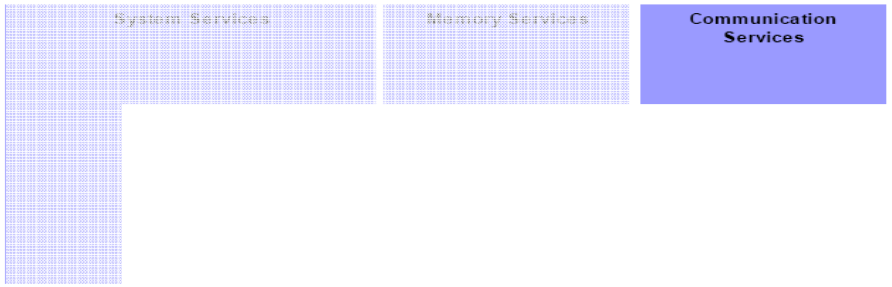
Communication Services

Memory Services

System Services



Service layer



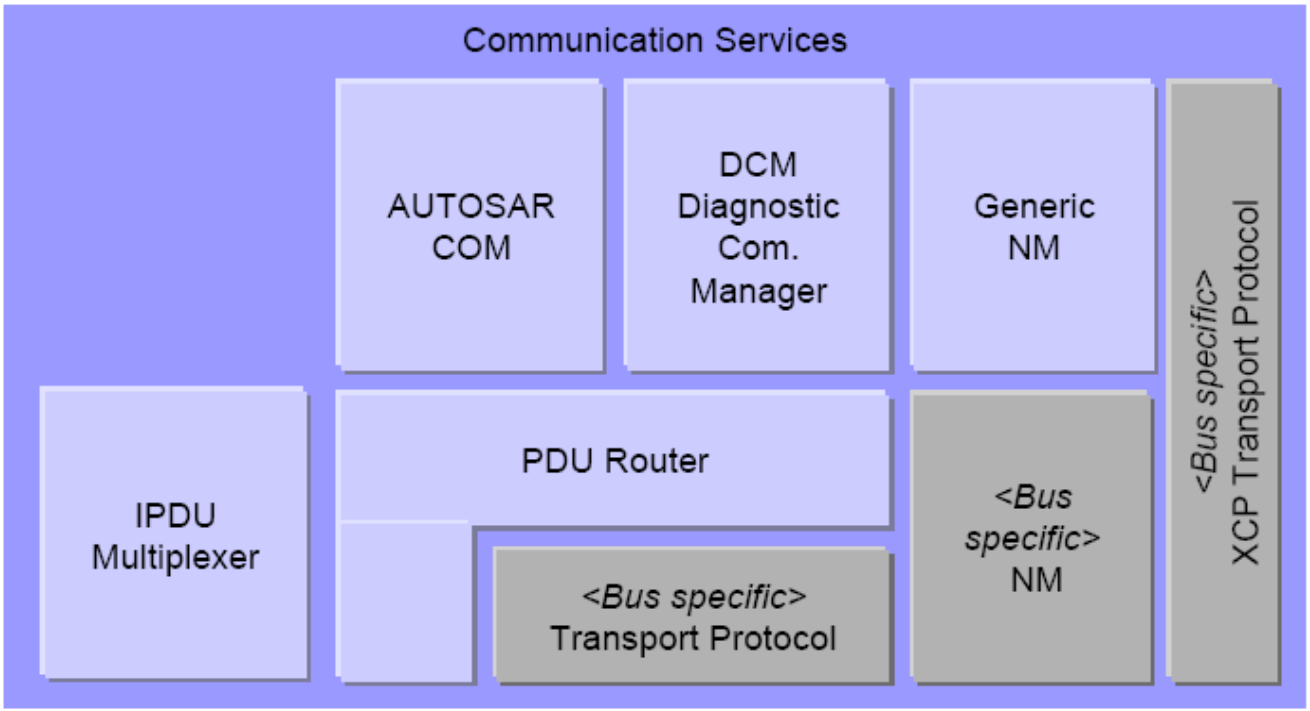
Communication Services

The communication services are a group of modules for vehicle network communication (CAN, LIN, FlexRay and MOST). They are interfacing with the communication drivers via the communication hardware abstraction. The task of this group of modules is

- to provide a uniform interface to the vehicle network for communication between different applications,
- to provide uniform services for network management,
- to provide a uniform interface to the vehicle network for diagnostic communication, and
- to hide protocol and message properties from the application.

Service layer

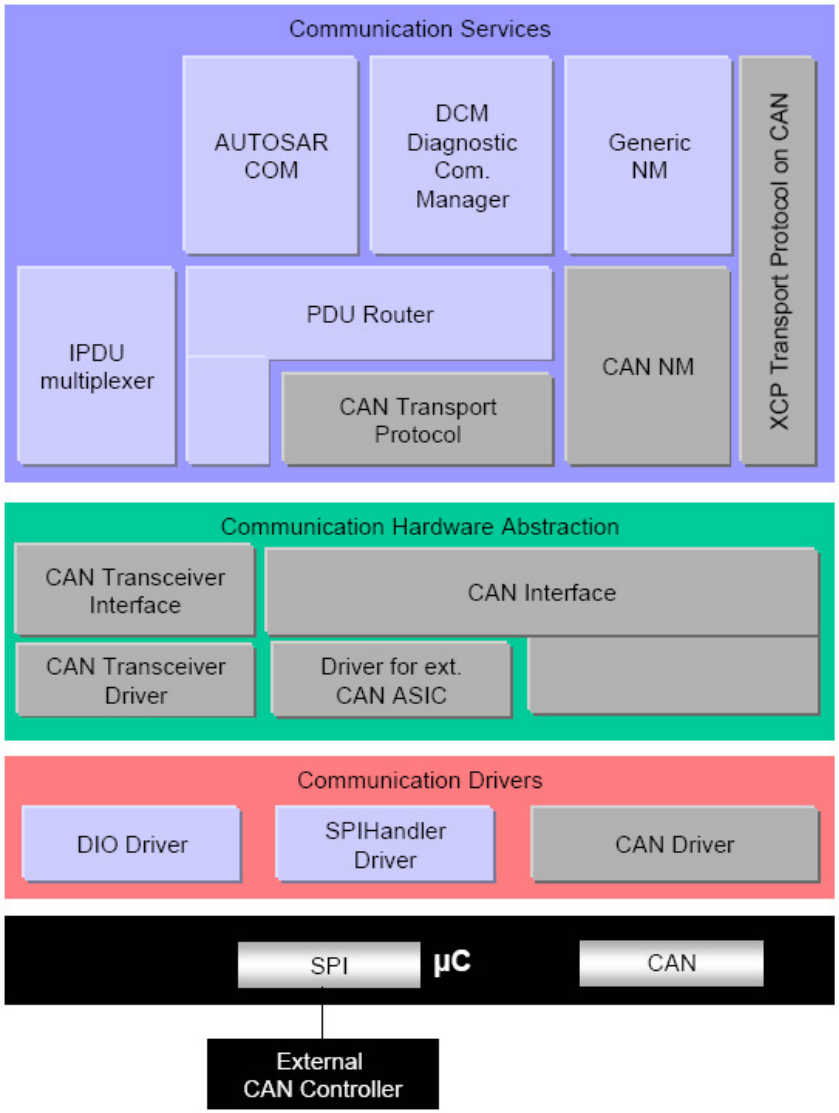
Generic structure



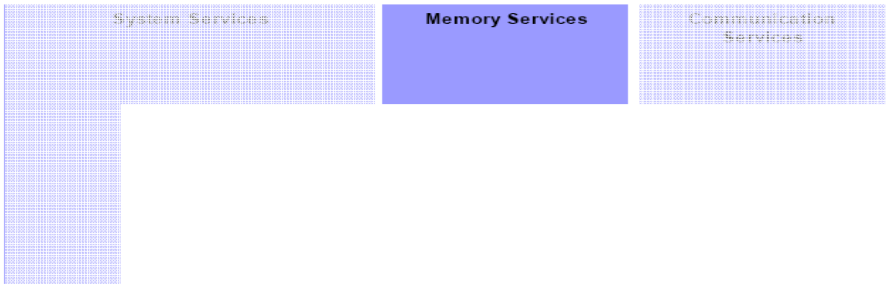
Color code: Bus specific modules are marked gray.

Service layer

An example (CAN bus):



Service layer



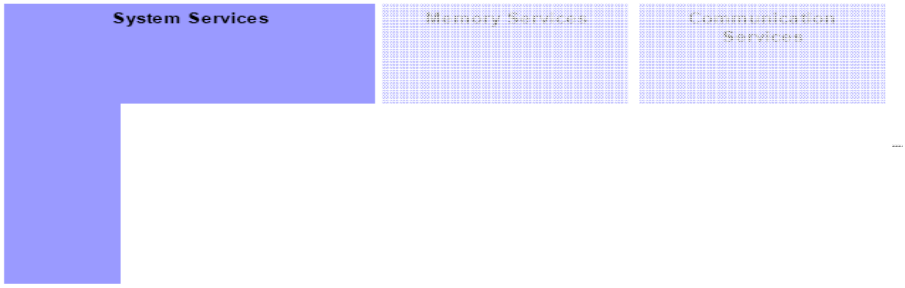
Memory Services

A group of modules responsible for the management of non volatile data (read/write from different memory drivers). The NVRAM manager provides a RAM mirror as data interface to the application for fast read access.

The task of this group of modules is

- to provide non volatile data to the application in a uniform way,
- to abstract from memory locations and properties, and
- to provide mechanisms for non volatile data management like saving, loading, checksum protection and verification, reliable storage etc.

Service layer



System Services

The system services are a group of modules and functions which can be used by modules of all layers.

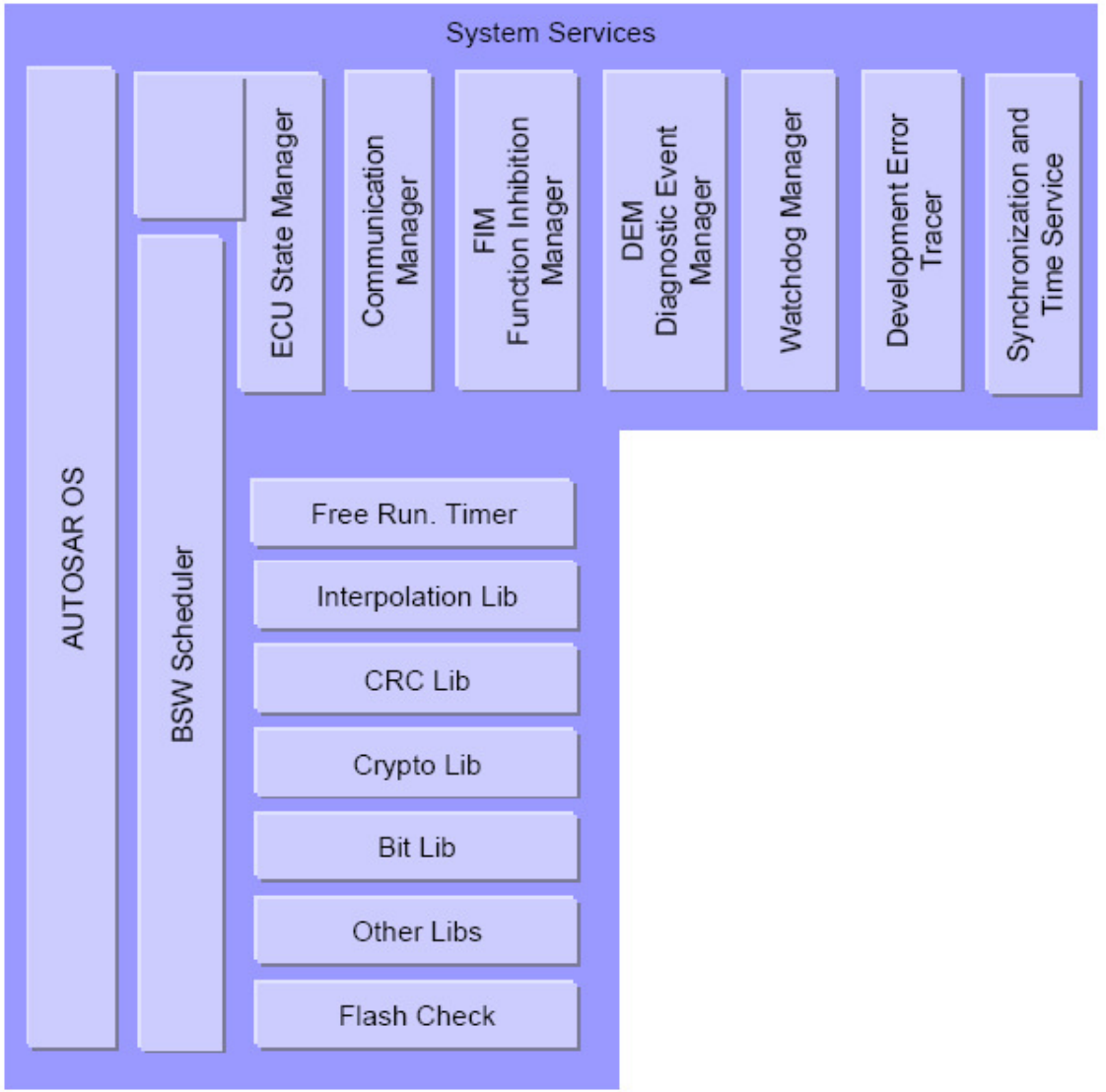
Examples are real-time operating system, error manager and library functions (like CRC, interpolation etc.).

Some of these services are µC dependent (like OS), ECU hardware and/or application dependent (like ECU state manager, DCM) or hardware and µC independent.

The task of this group of modules is to provide basic services for application and Basic Software modules.

Service layer

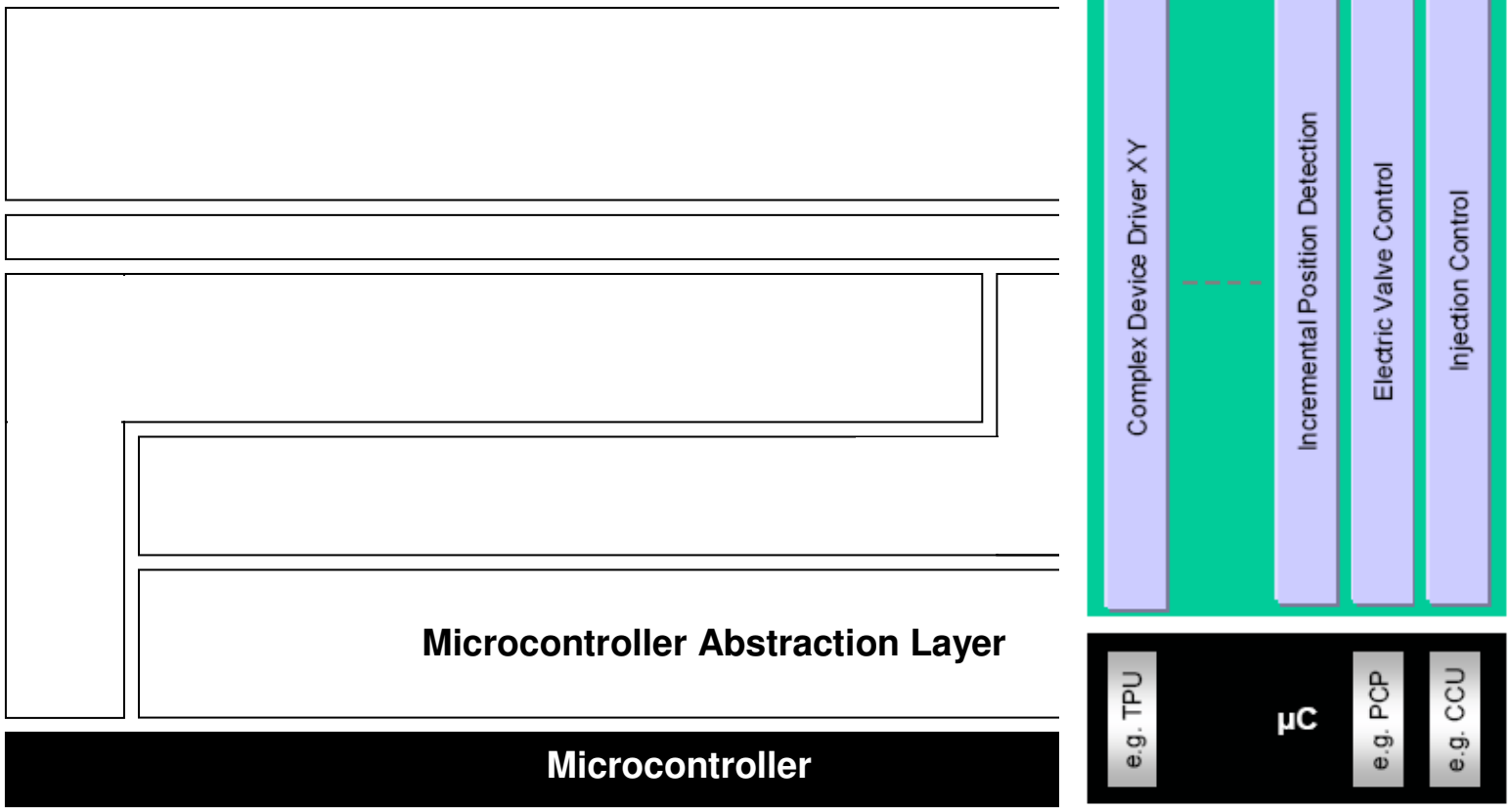
Structure



Complex Drivers

A **Complex Driver** implements complex sensor evaluation and actuator control with direct access to the µC using specific interrupts and/or complex µC peripherals (like PCP, TPU

Task: Fulfill the special functional and timing requirer complex sensors and actuators



Complex Drivers

The Complex Device Driver is a loosely coupled container, where specific software implementations can be placed. The only requirement to the software parts is that the interface to the AUTOSAR world has to be implemented according to the AUTOSAR port and interface specifications.

Complex Sensor and Actuator Control

The main task of the complex drivers is to implement complex sensor evaluation and actuator control with direct access to the µC using specific interrupts and/or complex µC peripherals (like PCP, TPU), e.g.

- injection control
- electric valve control
- incremental position detection

Complex Drivers

Non-Standardized Drivers

Further on the Complex Device Drivers will be used to implement drivers for hardware which is not supported by AUTOSAR.

If for example a new communication system will be introduced in general no AUTOSAR driver will be available controlling the communication controller. To enable the communication via this media, the driver will be implemented proprietarily inside the Complex Device Drivers. In case of a communication request via that media the communication services will call the Complex Device Driver instead of the communication hardware abstraction to communicate.

Another example where non-standard drivers are needed is to support ASICs that implement a non-standardized functionality.

Complex Drivers

Migration Mechanism

Last but not least the Complex Device Drivers are to some extend intended as a migration mechanism. Due to the fact that direct hardware access is possible within the Complex Device Drivers already existing applications can be defined as Complex Device Drivers. If interfaces for extensions are defined according to the AUTOSAR standards new extensions can be implemented according to the AUTOSAR standards, which will not force the OEM nor the supplier to reengineer all existing applications.

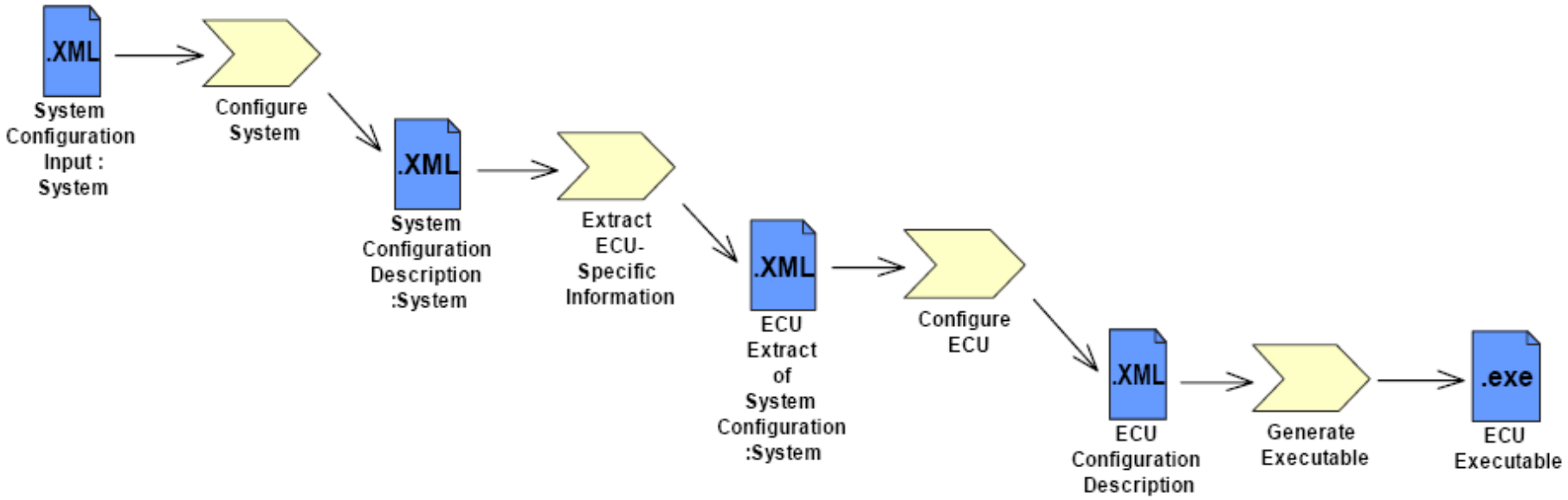
AUTOSAR Methodology

AUTOSAR requires a common technical approach for some steps of system development. This approach is called the “AUTOSAR Methodology”, which describes all major steps of the development of a system, from the system-level configuration to the generation of an ECU executable.

- The AUTOSAR Methodology is neither a complete process description nor a business model and “roles” and “responsibilities” are not defined.
- Furthermore, it does not prescribe a precise order in which activities should be carried out. The methodology is a mere work-product flow: it defines the dependencies of activities on work-products.

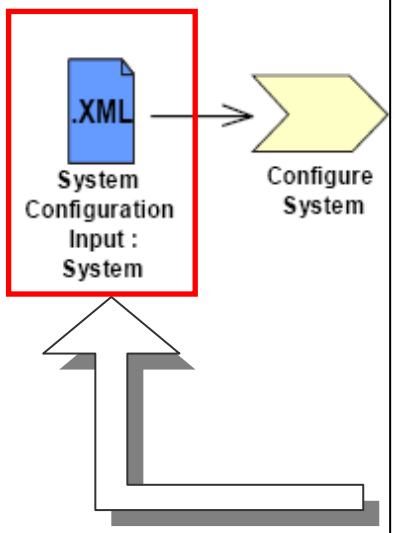
AUTOSAR Methodology

Design steps go from the system-level configuration to the generation of an ECU executable.



AUTOSAR Methodology

Design steps go from the system-level configuration to the generation of an ECU executable.



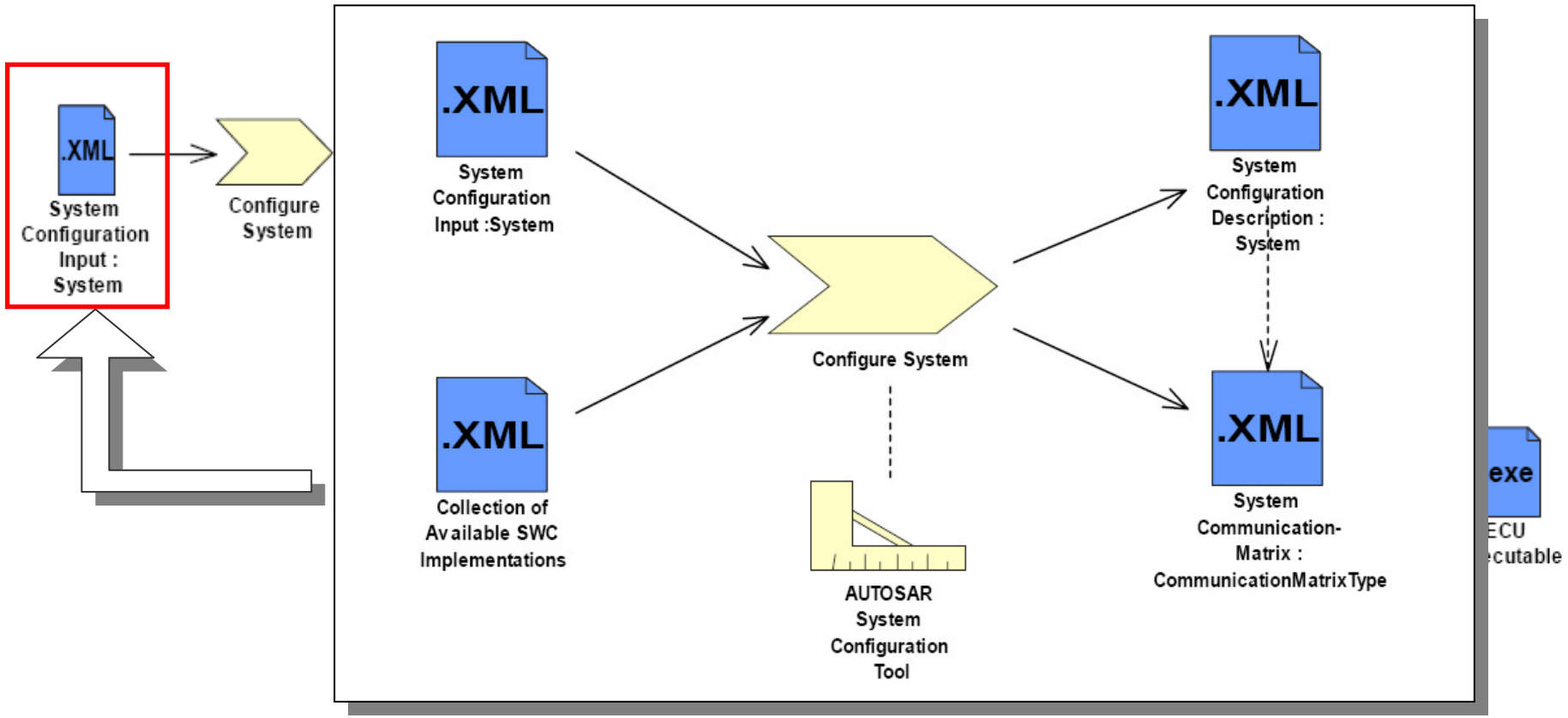
Firstly, the **`System Configuration Input`** must to be defined. The software components and the hardware are selected, and the overall system constraints identified. AUTOSAR provides a format for the formal description via the information exchange format and the use of the following templates.

- **Software Components**: each software component requires a description of the software API e.g. data types, ports, interfaces.
- **ECU Resources**: each ECU requires specifications regarding e.g. the processor unit, memory, peripherals, sensors and actuators.
- **System Constraints**: regarding the bus signals, topology and mapping of belonging together software components.

exe
ECU
ecutable

AUTOSAR Methodology

Design steps go from the system-level configuration to the generation of an ECU executable.



System Configuration

The System Configuration Input includes or references various constraints. These constraints can force or forbid certain components to be mapped to certain ECUs or requires certain implementations to be used for components.

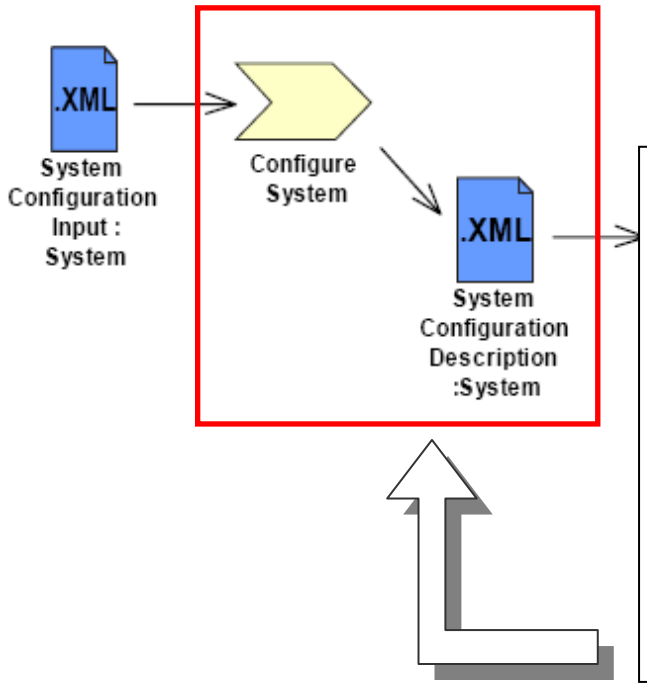
In addition, these constraints can contain resource estimations describing the net availability of resources on ECUs and thereby limiting the possible mappings.

The activity Configure System contains complex algorithms and/or engineering work. There is a strong need for experience in system architecture to map all the software components to the ECUs. The tool AUTOSAR System Configuration Tool supports the configuration.

An important output of the activity is the design of the System Communication-Matrix. This System Communication-Matrix completely describes the frames running on the networks described in the topology and the contents and timing of those frames.

AUTOSAR Software Process

Design steps go from the system-level configuration to the generation of an ECU executable.

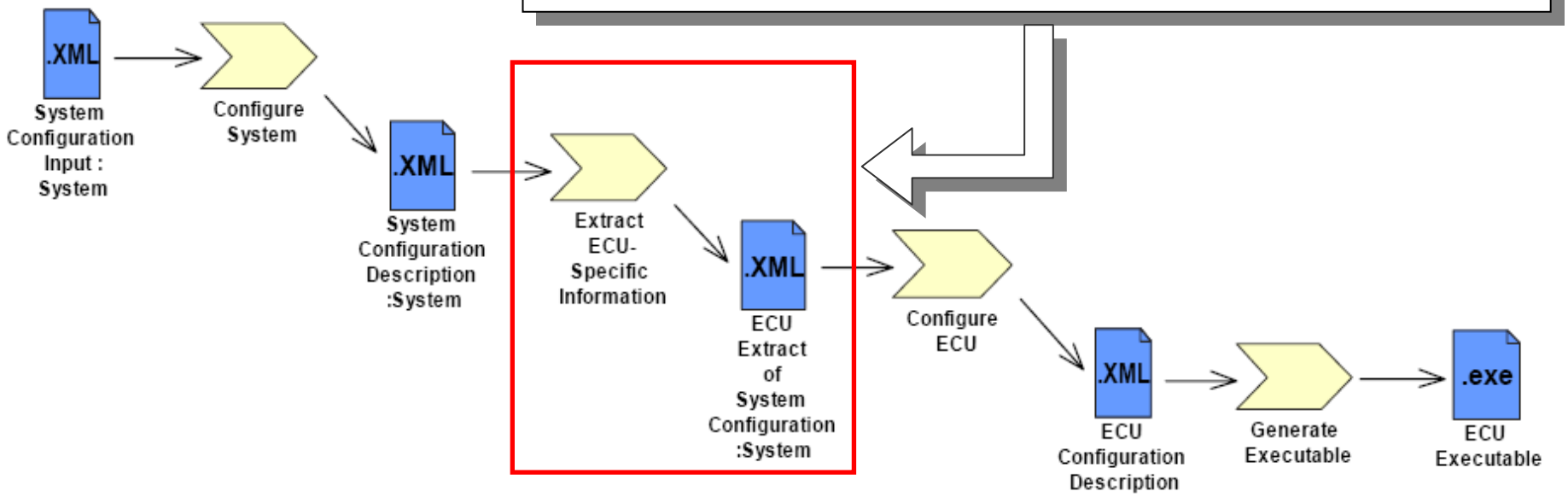


The Configure System activity maps the software components to the ECUs with resources and timing requirements. The output of this activity is the **`System Configuration Description`** including system information (e.g. bus mapping, topology) and the mapping of software components to ECUs.

AUTOSAR Software Process

Further steps have to be performed for each ECU in the system.

Extract ECU-Specific Information extracts the information from the System Configuration Description for a specific ECU. It becomes the **`ECU Extract of System Configuration`**.



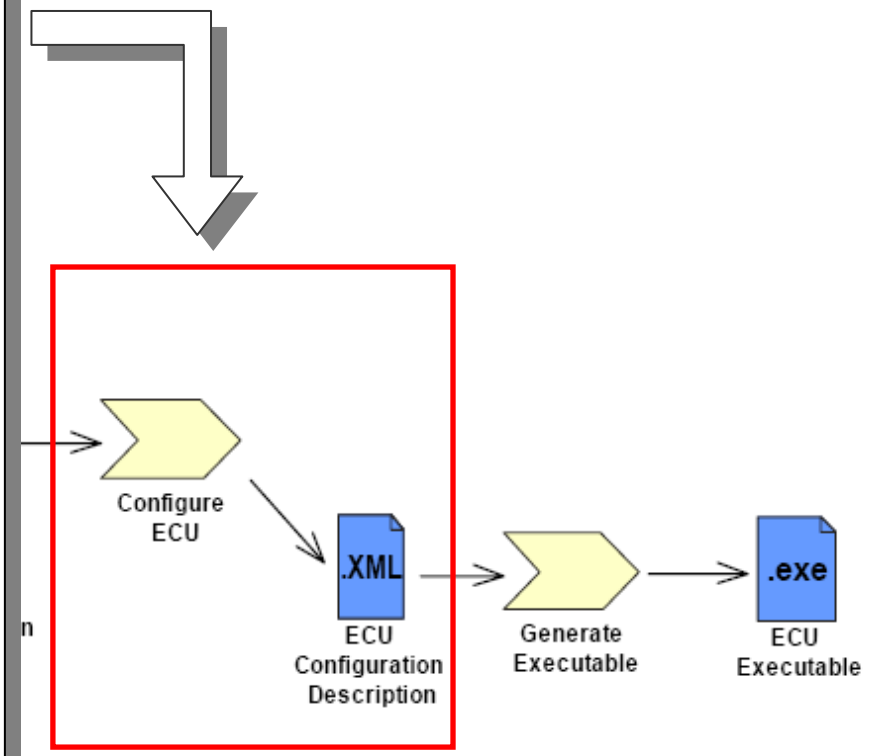
This is a one to one copy of all elements of the System Configuration Description that are appointed to this specific ECU.

This step can be completely automated.

AUTOSAR Software Process

Further steps have to be performed for each ECU in the system.

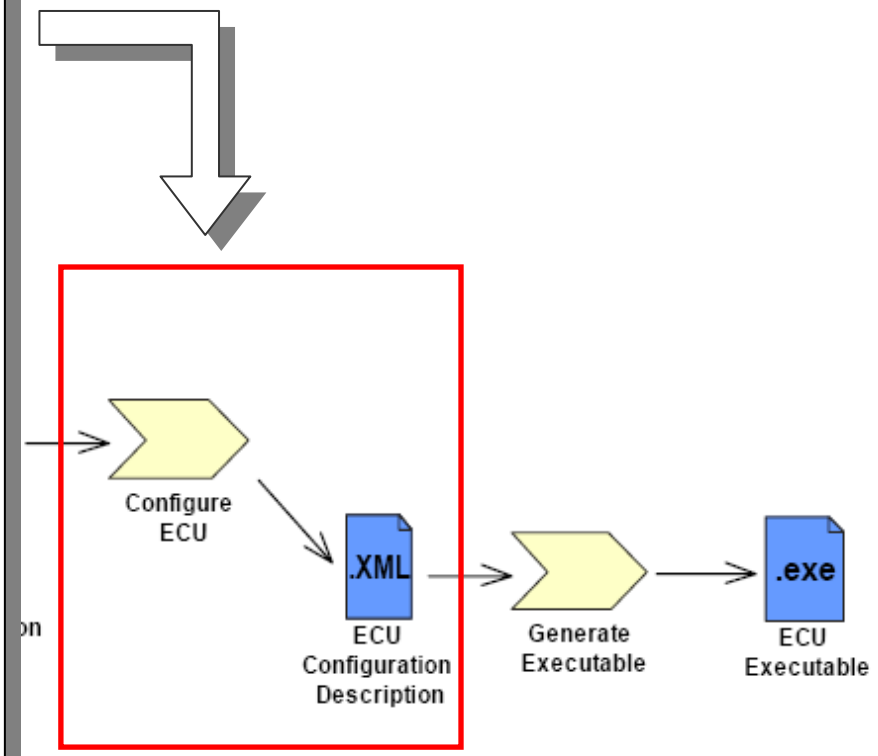
Configure ECU mainly deals with the configuration of the RTE and the Basic Software modules. It adds all implementation-related information, including task scheduling, required Basic Software modules, configuration of the Basic Software, assignment of runnable entities to tasks, etc. The result of the activity is included in the *ECU Configuration Description*, which collects ECU-specific information. The runnable software to this specific ECU can be built from this information.



AUTOSAR Software Process

Further steps have to be performed for each ECU in the system.

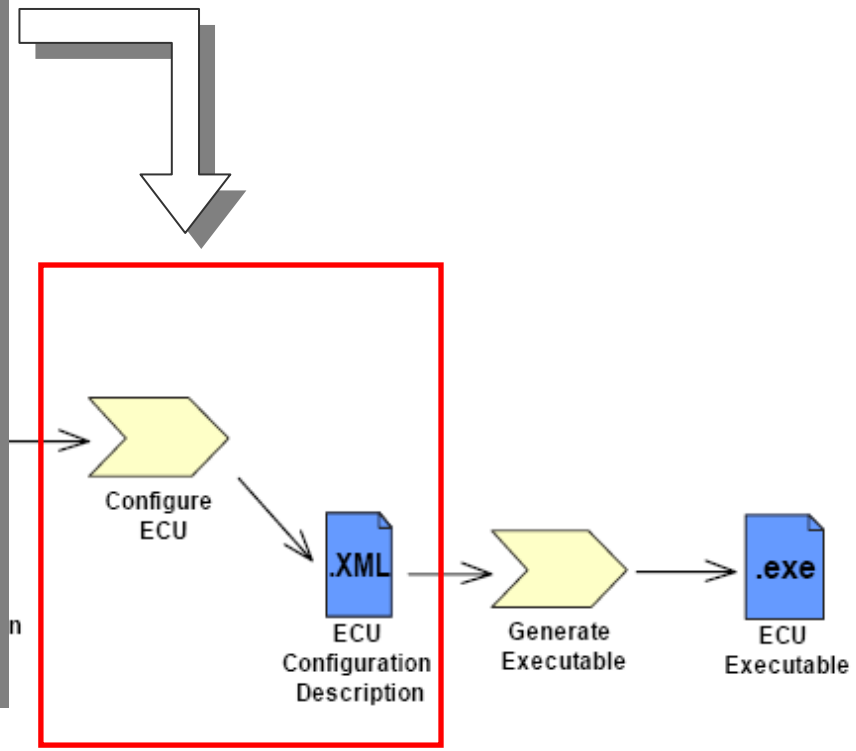
The configuration is based on the information extracted from the ECU Extract of System Configuration, Collection of Available SWC Implementations, and BSW Module Description.
The latter contains the Vendor Specific ECU Configuration Parameter Definition which defines all possible configuration parameters and their structure.
The BSW Module Description is assumed to consist of single descriptions delivered together with the appropriate used BSW module.



AUTOSAR Software Process

Further steps have to be performed for each ECU in the system.

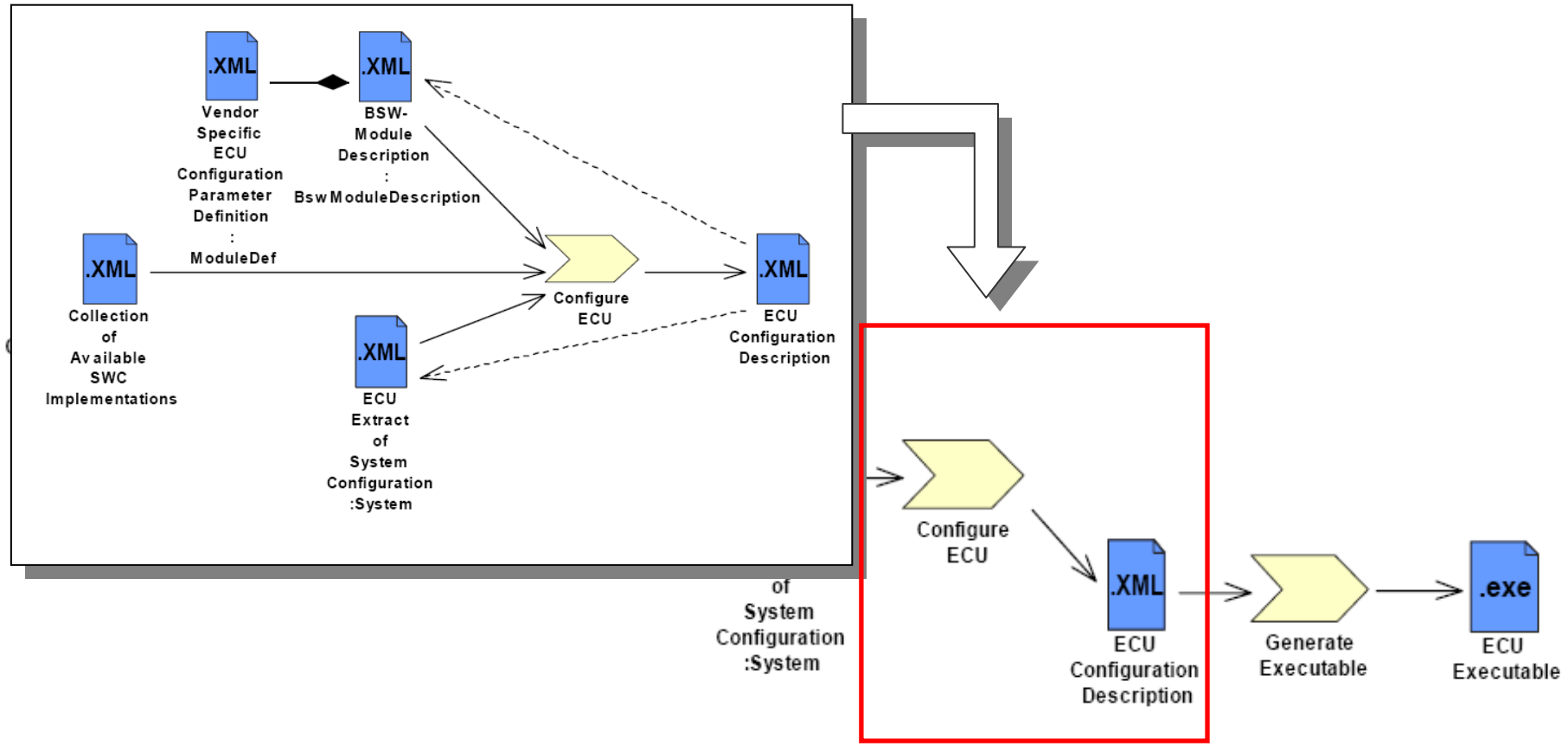
The detailed scheduling information or the configuration data for e.g. the communication module, the operating system, or AUTOSAR services have to be defined in this activity. Moreover at the latest here an implementation is selected for each Atomic Software Component.



In contrast to the extraction of ECU-specific information, the configuration activity is a non-trivial design step, which requires complex design algorithms and engineering knowledge.

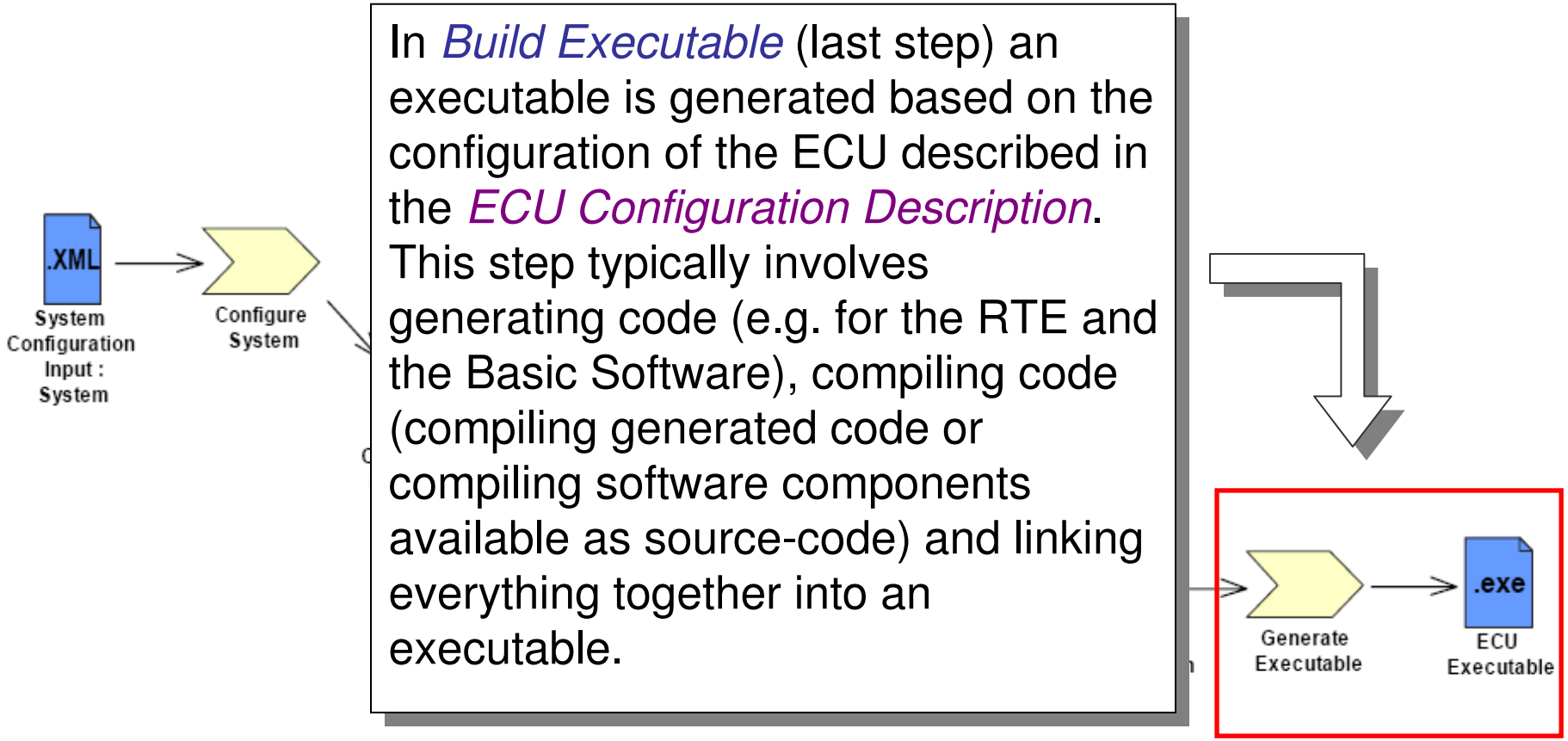
AUTOSAR Software Process

Further steps have to be performed for each ECU in the system.



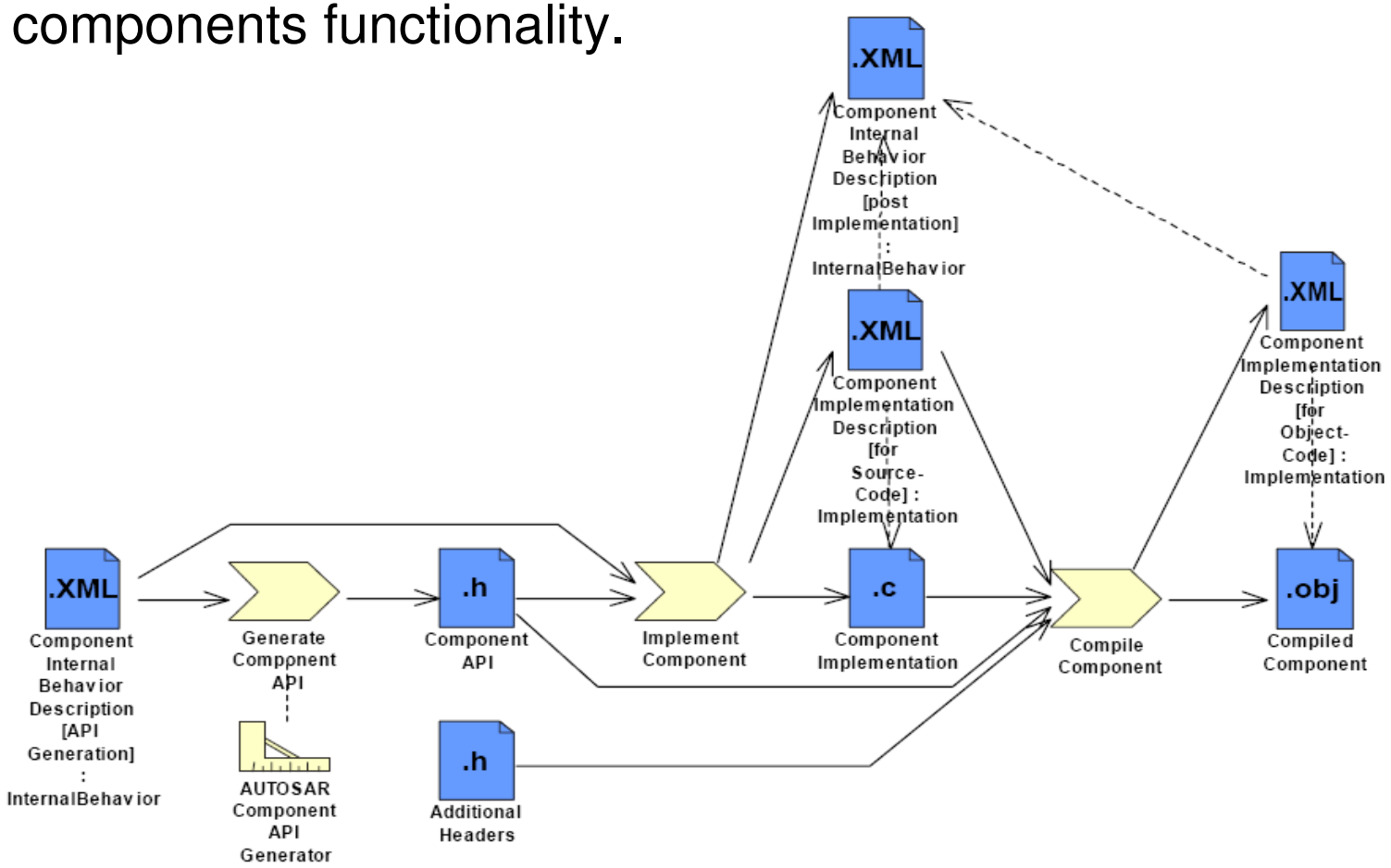
AUTOSAR Software Process

Further steps have to be performed for each ECU in the system.



AUTOSAR Software Process

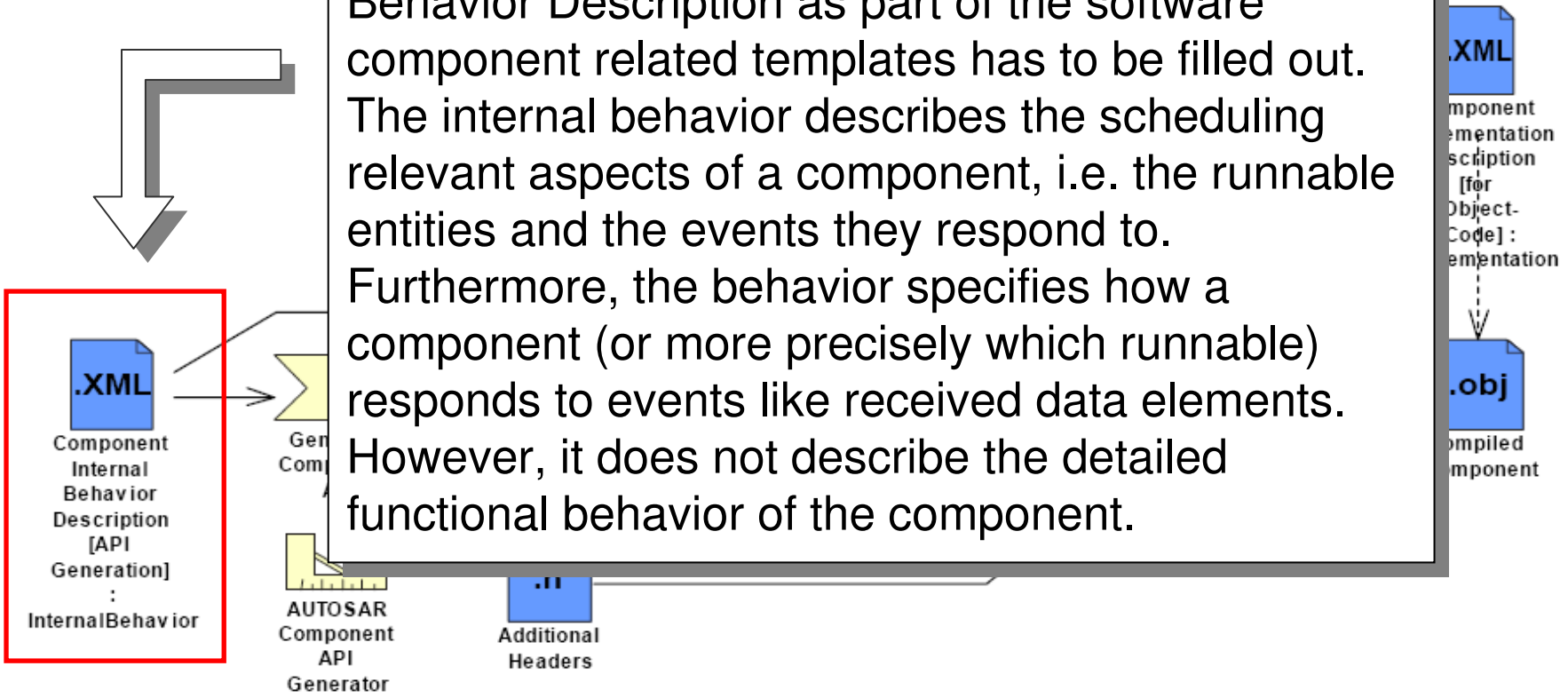
Parallel to these steps are several steps performed for every application software component (to be integrated later into the system), e.g. generating the components API, and implementing the components functionality.



AUTOSAR Software Process

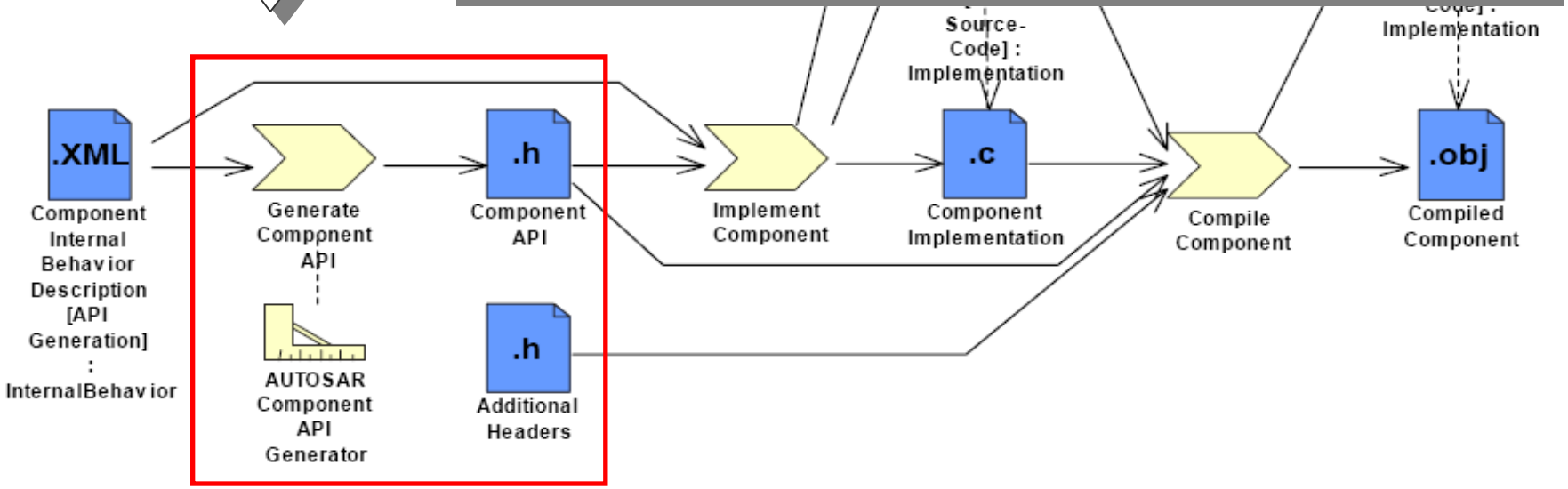
The initial work in this context starts with providing the necessary parts of the software component description.

That means at least the Component Internal Behavior Description as part of the software component related templates has to be filled out. The internal behavior describes the scheduling relevant aspects of a component, i.e. the runnable entities and the events they respond to. Furthermore, the behavior specifies how a component (or more precisely which runnable) responds to events like received data elements. However, it does not describe the detailed functional behavior of the component.



AUTOSAR Software Process

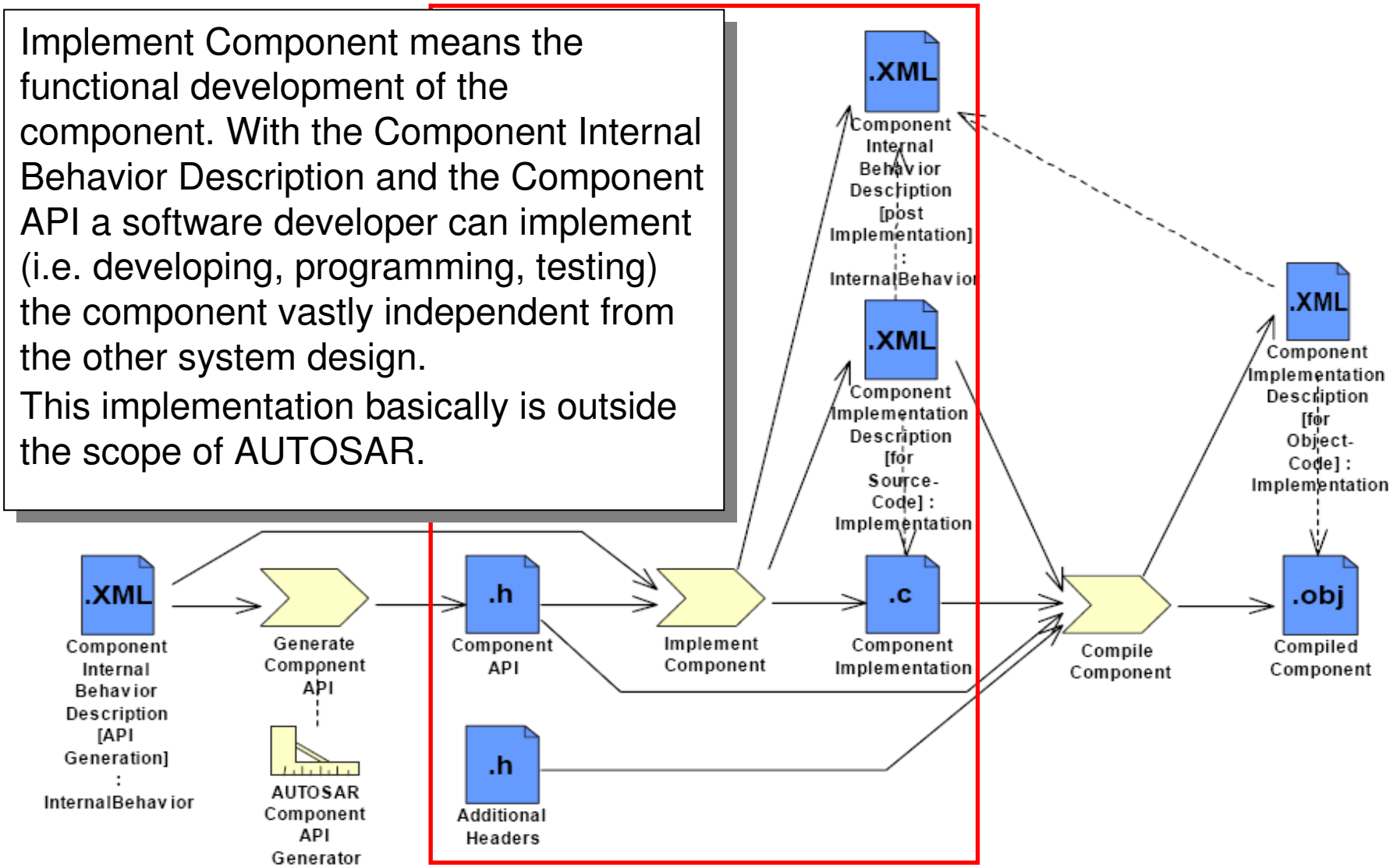
Afterwards Generate Component API has to be performed. This is a tool-based activity. The AUTOSAR Component API Generator reads the Component Internal Behavior Description of the appropriate software component and generates the Component API accordingly. The Component API contains all header declarations for the RTE communication. .



AUTOSAR Software Process

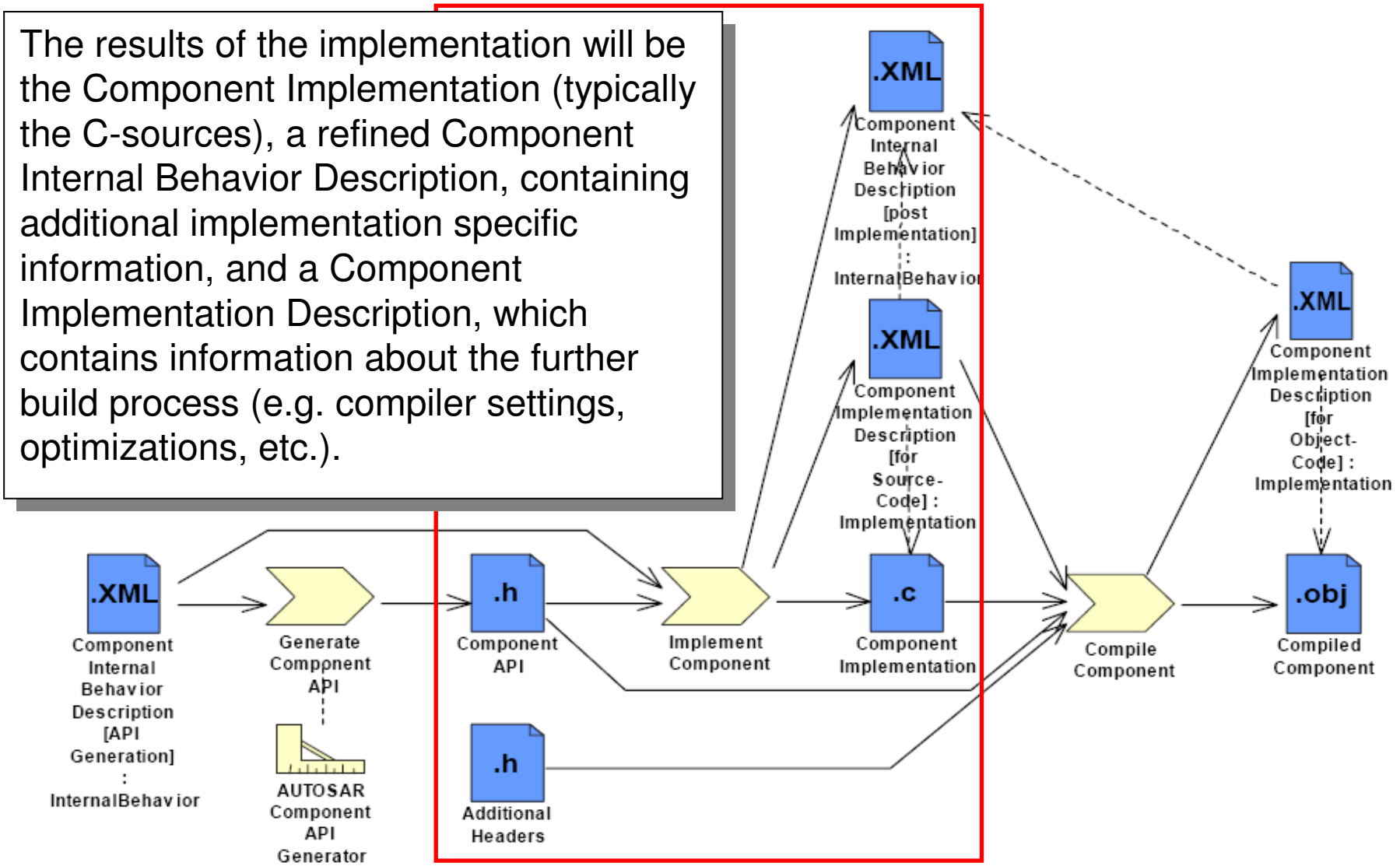
Implement Component means the functional development of the component. With the Component Internal Behavior Description and the Component API a software developer can implement (i.e. developing, programming, testing) the component vastly independent from the other system design.

This implementation basically is outside the scope of AUTOSAR.



AUTOSAR Software Process

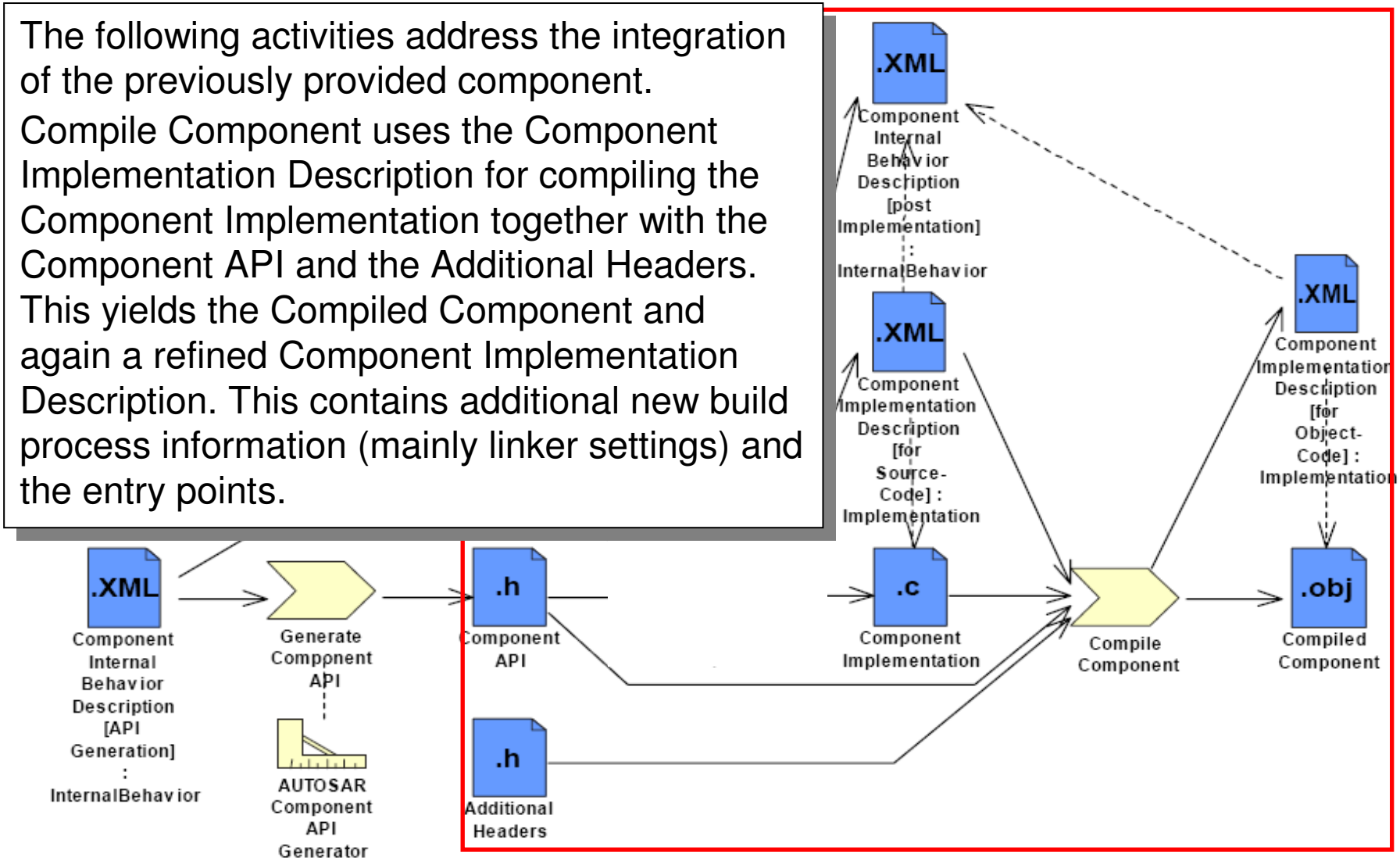
The results of the implementation will be the Component Implementation (typically the C-sources), a refined Component Internal Behavior Description, containing additional implementation specific information, and a Component Implementation Description, which contains information about the further build process (e.g. compiler settings, optimizations, etc.).



AUTOSAR Software Process

The following activities address the integration of the previously provided component.

Compile Component uses the Component Implementation Description for compiling the Component Implementation together with the Component API and the Additional Headers. This yields the Compiled Component and again a refined Component Implementation Description. This contains additional new build process information (mainly linker settings) and the entry points.



AUTOSAR

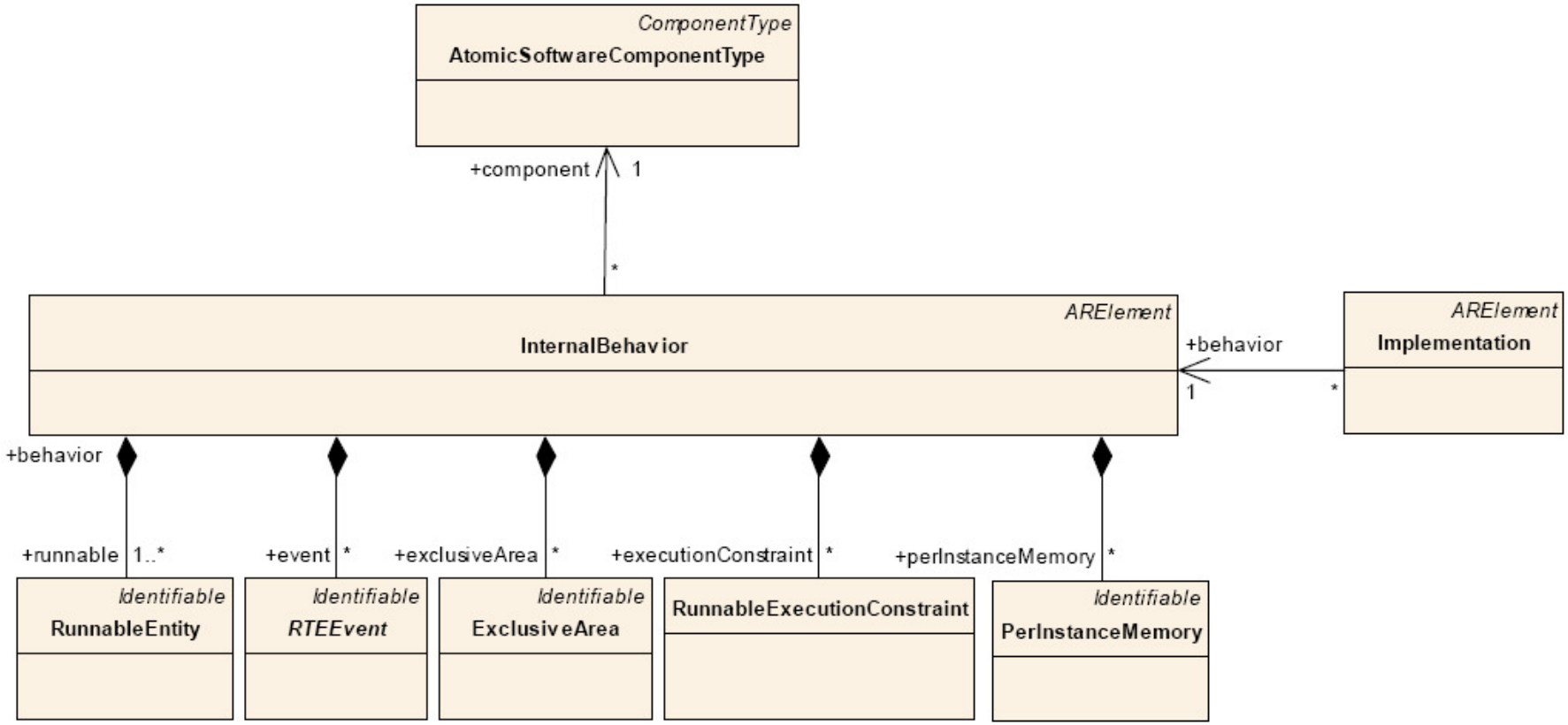
More on RTE, component integration and runnables

AUTOSAR Components: interface with the RTE

We now deal with component aspects that:

1. support the proper configuration of the RTE and the BSW: the software component description needs to provide detailed information on how the underlying software should behave with respect to the component (for example: what runnables of the software-component should be started when by the AUTOSAR OS),
2. describe the communication properties of a software-component,
3. serve as a basis for the description of the detailed resource requirements of software-components, and
4. provide a more detailed description of the timing behavior of atomic softwarecomponents.

AUTOSAR Components: interface with the RTE

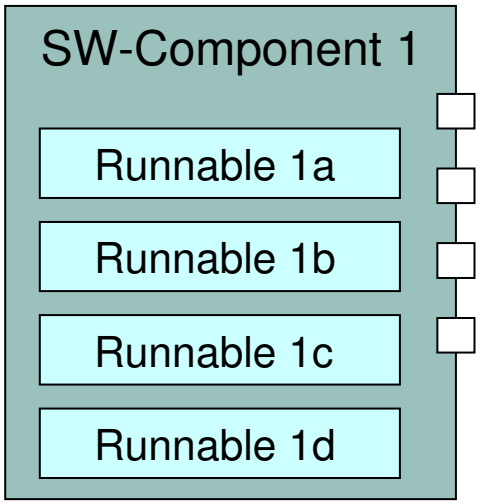


AUTOSAR Components: Runnable entities

Runnable Entities (also Runnable) are defined in the VFB specs.

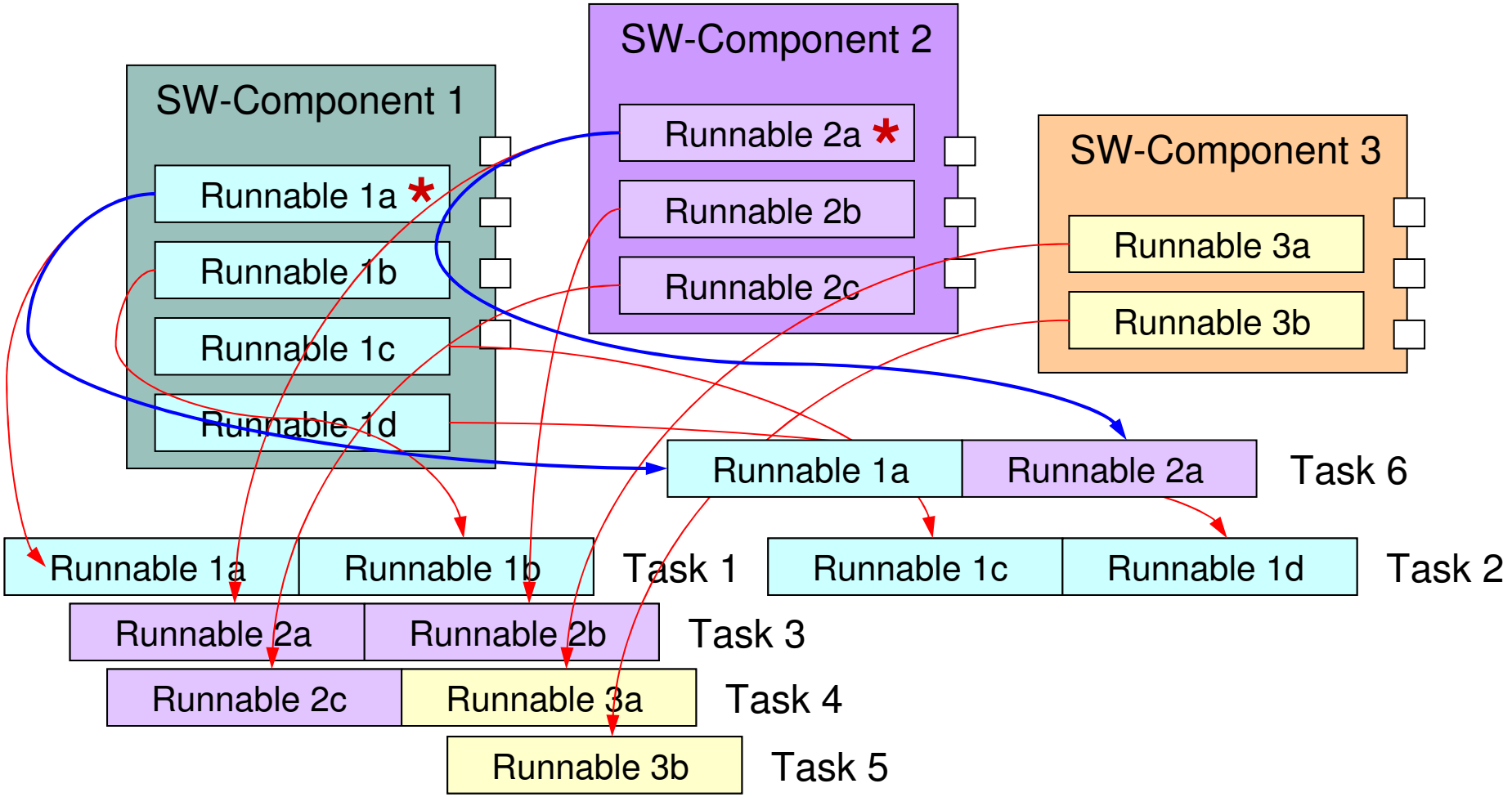
Runnable entities are the smallest code-fragments that are provided by the component and are (at least indirectly) a subject for scheduling by the operating system. An implementation of an atomic software-component has to provide an entry-point to code for each Runnable in its "InternalBehavior“.

- It is not possible for "CompositionType" to be referenced by "InternalBehavior". Only atomic software-components may have Runnables.

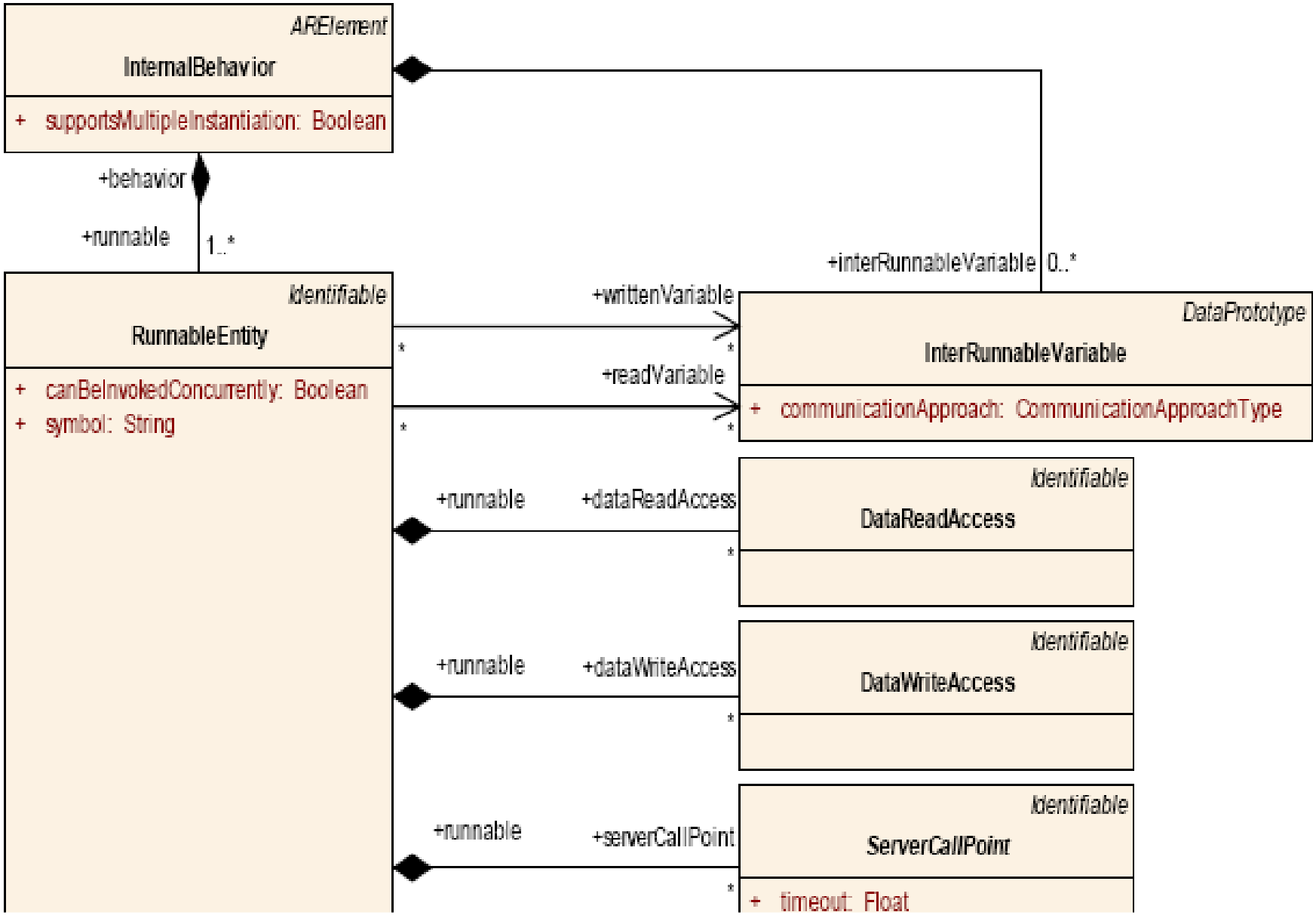


AUTOSAR Components: Runnable entities

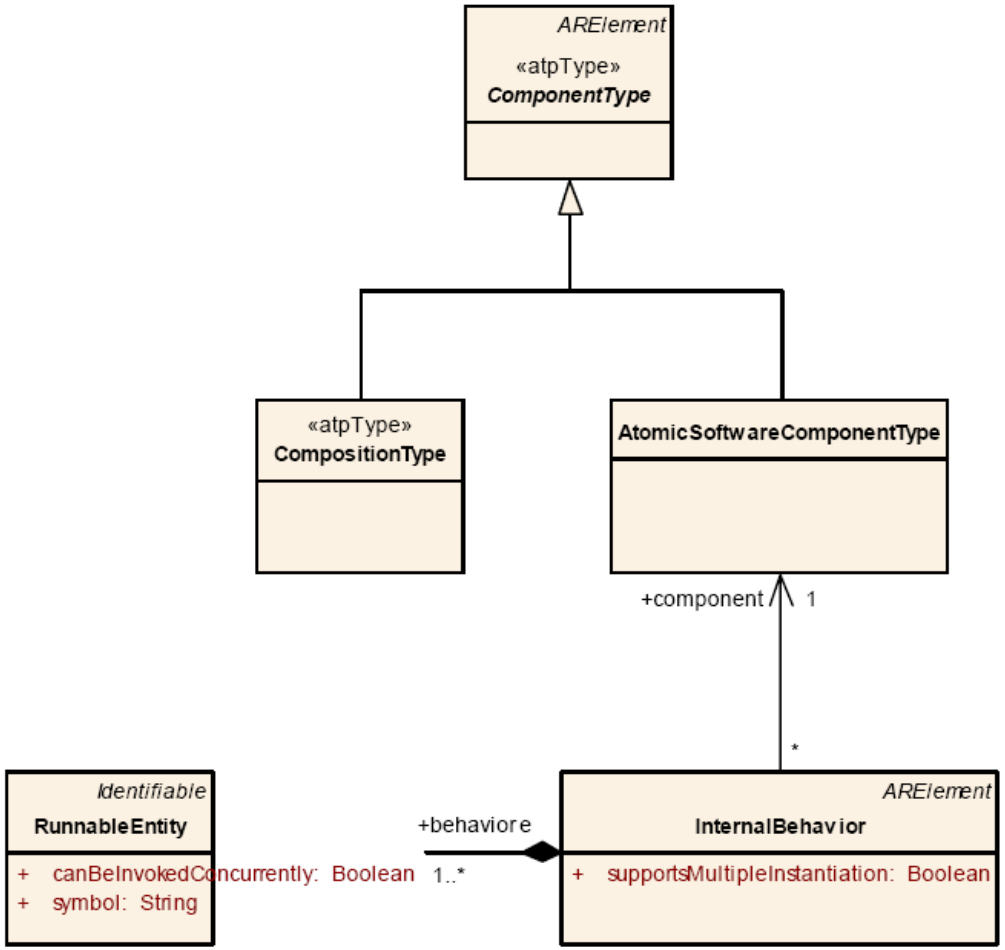
In most cases Runnables will not be scheduled individually but as parts of OS tasks.



AUTOSAR Components: Runnable entities



AUTOSAR Components: Runnable entities



AUTOSAR Components: Runnable entities

Class	**RunnableEntity**			
Package	AUTOSAR Templates::SWComponentTemplate::InternalBehavior			
Class Description	The runnable entities are the smallest code-fragments that are provided by the component and are executed in the RTE. Runnables are for instance set up to respond to data reception or operation invocation on a server.			
Base Class(es)	Identifiable			
Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
canBeInvokedConcurrently	Boolean	1	aggregation	Normally, this is FALSE. When this is TRUE, it is allowed that this runnable entity is invoked concurrently (even for one instance of the SW-C), which implies that it is the responsibility of the implementation of the runnable to take care of this form of concurrency.
dataReadAccess	DataReadAccess	*	aggregation	Runnable has read access to data element
dataReceivePoint	DataReceivePoint	*	aggregation	Data receive points of this runnable.
dataSendPoint	DataSendPoint	*	aggregation	The runnable has data send point.
dataWriteAccess	DataWriteAccess	*	aggregation	Runnable has write access to data element
insideExclusiveArea	RunnableEntityRunsInExclusiveArea	*	aggregation	The runnable entity runs inside the referenced exclusive area
readVariable	InterRunnableVariable	*	reference	Inter-runnable variables that are read by this Runnable.
serverCallPoint	ServerCallPoint	*	aggregation	The runnable has server call point.

Class	**RunnableEntity**			
symbol	String	1	aggregation	The symbol describing this runnable's entry point. This is considered the API of the runnable and is required during the RTE contract phase.
usesExclusiveArea	RunnableEntityCanEnterExclusiveArea	*	aggregation	This means that the runnable can enter/leave the referenced exclusive area through explicit API calls.
waitPoint	WaitPoint	*	aggregation	The runnable has wait point.
writtenVariable	InterRunnableVariable	*	reference	Inter-runnable variables that are written by Runnable.

AUTOSAR Components: interface with the RTE

In case “canBeInvokedConcurrently” is FALSE.

During run-time, each Runnable of each instance of an atomic software-component is (by being a member of an OS task) in one of three states:

- Suspended: the initial state, the Runnable is passive, can be started
- Enabled: the Runnable should run (because for example a message has been received on a port or a timing event occurs)
- Running: the Runnable is running within a running task.

The InternalBehavior describes for each Runnable, when a transition from Suspended to Enabled occurs using the concept of RTEEvent.

When a Runnable is "Enabled", the OS can decide to start running it. The delay between entering "Enabled" and moving into "Running" depends on the OS scheduling.

The transition from "Running" into "Suspended" depends on the Runnable: it occurs when the Runnable returns or terminates

AUTOSAR Components: interface with the RTE

In case the internal behavior defines a runnable as one that cannot be invoked concurrently, it is the responsibility of the RTE and the BSW to make sure that the runnable is never started concurrently. This implies that the implementation of the SW-Component does not need to worry about concurrency issues.

For example:

- The internal behavior of a component-type MyComponentType describes a Runnable R1, enabled when an operation on a clientserver p-port is invoked. The component specifies that the Runnable R1 cannot be invoked concurrently.
- The component MyComponentType is instantiated on an ECU.
- When a call of the operation is received, the corresponding instance of the Runnable R1 is enabled and the OS will start executing the Runnable in a task.
- If *another* call of the operation is received while the Runnable is "running", the OS must not run the Runnable again in a second task. Rather, the OS has to wait (and maybe queue the second incoming request) until the Runnable returns into "Suspended".

AUTOSAR Components: interface with the RTE

If “canBeInvokedConcurrently” is TRUE, the same runnable can run several times concurrently in different tasks. This implies that there is no single state associated to the runnable.

Note that the SW-Component description itself does not put any bounds on the number of concurrent invocations of the runnable that are allowed.

Allowing concurrent invocation of a runnable implies that the implementation of the SW-component needs to take care of this additional form of concurrency.

For example:

- The internal behavior of MyComponentType describes a Runnable R1, which should be enabled when an operation on a clientserver p-port of the component is invoked. The Runnable R1 can be invoked concurrently.
- The component MyComponentType is instantiated on an ECU.
- When a call of the operation is received, the corresponding instance of the Runnable R1 is enabled and the OS will start executing the Runnable in a task. If *another* call of the operation is received, it is allowed that the same runnable is started again in a different task.

AUTOSAR Components: interface with the RTE

A typical use-case of concurrent runnables are the AUTOSAR services. The AUTOSAR services will typically take care of concurrency internally: several software components can directly use the services in parallel. The ECU-integrator could then decide that the runnable implementing the AUTOSAR service runs directly in the context (in the task) of the software-component invoking the service. This is a very efficient, direct coupling between the client and the server: the connector between the client and the server is reduced to a local function-call.

AUTOSAR Components: interface with the RTE

supportsMultiple-Instantiation	canBeInvoked-Concurrently	Implication for an implementation of a runnable
FALSE	FALSE	This implies that the implementation of the runnable will never be invoked concurrently from several tasks. The implementation does not need care about reentrancy issues[16] and can typically use "static variables" to store state.
TRUE	FALSE	In case there are several instances of the same component on the local ECU, the implementation of the runnable can still be invoked concurrently from several tasks. However, there will be not concurrent invocations of the implementation with the same "instance handle". To ensure that this is safe, the implementation will typically use per-instance memory.
FALSE/TRUE	TRUE	In this case the runnable can be invoked concurrently from several tasks, even with the same instance handle.

AUTOSAR Components: interface with the RTE

Preemption

The basic execution model of the Runnables shown above does not fundamentally preclude the OS from preempting the execution of a Runnable to execute another runnable.

The "InternalBehavior" might however put additional constraints on the behavior of the OS, so that certain Runnables might never be preempted (for example to assure certain timing requirements when the Runnable needs to finish quickly after being enabled or for example to ensure that logical sequencing constraints are respected).

Reentrancy and “library functions”

Note that all code that is called by different Runnables (like e.g. library routines, etc.) must obviously be reentrant. A filter algorithm implemented in C, for example, is not allowed to store values from previous runs by means of static variables or variables with external binding.

AUTOSAR Components: RTE events

During execution, several run-time events will occur, such as the reception of a remote operation-invocation on a P-Port or a timeout on an R-Port that is not receiving the data-elements it expects. Describing an RTEEvent in the software-component template includes two aspects:

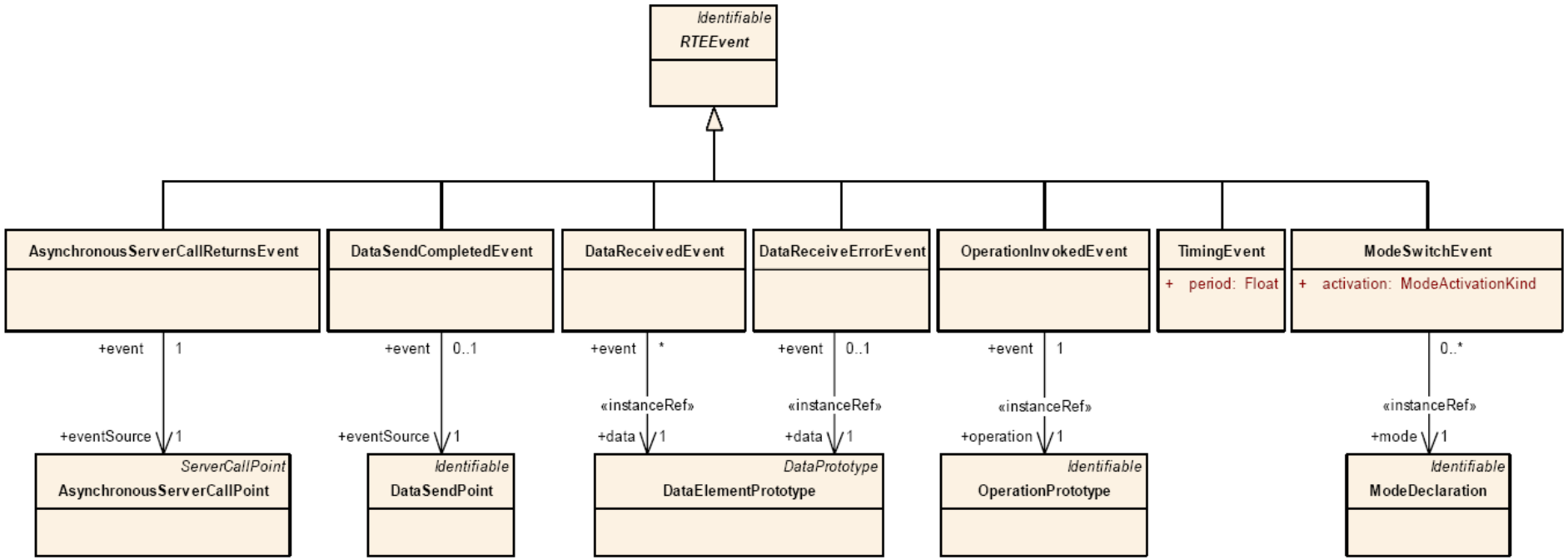
1. Defining an event
2. Defining how the RTE should deal with the event when it occurs

As described in the virtual functional bus specification, the implementation of a software-component can interact with the occurrence of such events in two ways:

- The RTE can be instructed to enable a specific runnable when the event occurs
- The RTE can provide "wait-points", that allow a runnable to block until an event in a set of events occurs

AUTOSAR Components: RTE events

The description of the internal behavior includes a description of all events that the internal behavior of the atomic software-component relies on. This "RTEEvent“ shows up as an "abstract" base-class in the meta-model: the exact attributes of the "RTEEvent" depend on the exact event that is described



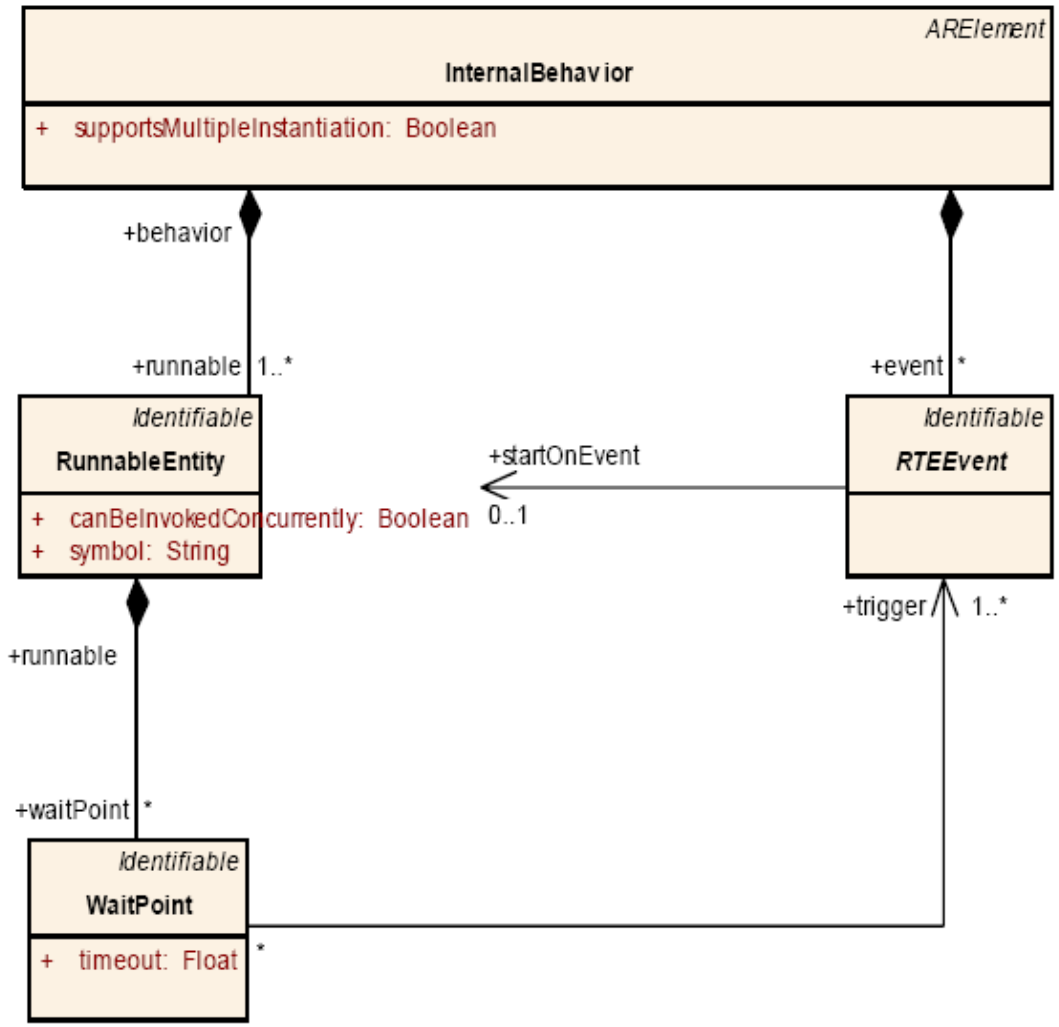
AUTOSAR Components: Response to events

In case the OS needs to start a Runnable when the corresponding event occurs, the "RTEEvent" can directly reference the Runnable that needs to be started. When the software-component description uses this feature, it is the responsibility of the OS to start the Runnable when the event occurs.

In case the Runnable wants to block and wait for events (which makes the runnable into a cat. 2 runnable), the description of the runnable may include the definition of a "wait-point". Such a "WaitPoint" contains a reference to all events that are waited for. The wait-point will block until one of the referenced events occurs.

A single "RunnableEntity" can **actually wait** only at a single "WaitPoint" for being scheduled. On the other hand, it is in general possible that a single event can be used to trigger "WaitPoints" in different "RunnableEntities"

AUTOSAR Components: RTE events



AUTOSAR Components: Communication attributes

The highest level of description of information exchanged between components in an AUTOSAR system is the “PortInterfaces”, as shown in earlier sections.

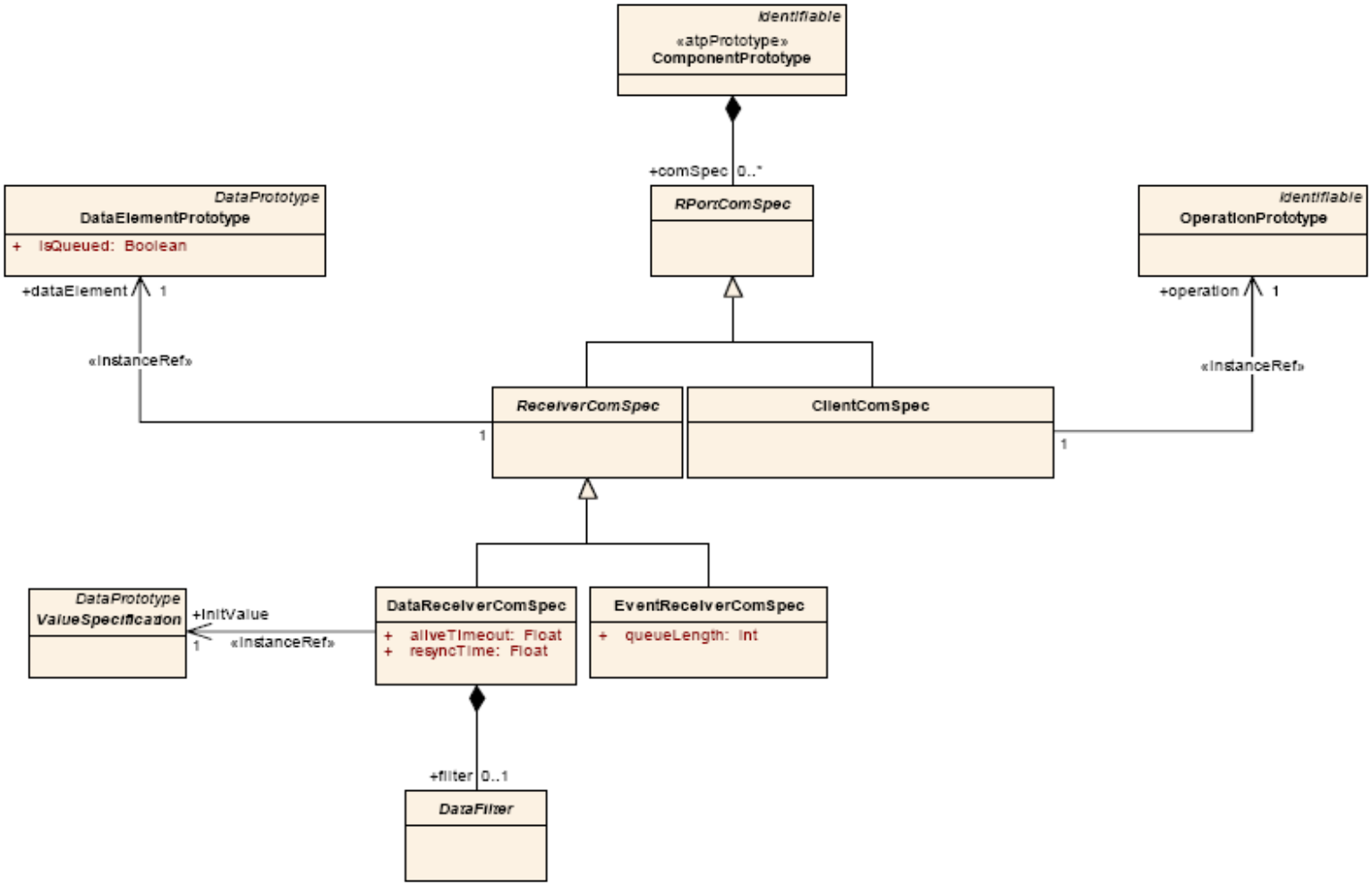
Such an interface however, only describes structure and does not include information about whether communication needs to be done reliably, or whether an init value exists in case the real data is not yet available.

This kind of information is known only within the particular scenario the interface is used and also frequently differs depending on whether an interface is required or provided. Therefore, most communication relevant attributes are related to the ports of a component.

The communication attributes are organized in “communication specification” (short: ComSpec) classes. The model distinguishes three basic classes depending on the role (R-, P-Port or connector) as detailed below.

AUTOSAR Components: Communication attributes

Model of the communication attributes for an R-Port.



AUTOSAR Components: Communication attributes

The ComSpec attributes are collected depending on the kind of data transmitted, which means they may differ depending on whether data elements are exchanged (sender-receiver), operations are called (client-server), or even depend on whether the data-elements represent data or events.

This is expressed in the inheritance tree of ComSpec classes. Each of these classes may then carry the specific attributes.

An R-Port may aggregate many ComSpec classes, possibly one for each interface element (data element or operation) the associated interface contains. The meaning of the attributes shown above is explained in the following class tables.

AUTOSAR Components: RPort attributes

Communication attributes specific to receiving data.

Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
aliveTimeout	Float	1	aggregation	Specify the amount of time (in seconds) after which the software component (via the RTE) needs to be notified if the corresponding data item have not been received according to the specified timing description.
filter	DataFilter	0..1	aggregation	
initValue	ValueSpecification	1	reference to instance	Initial value to be used in case the sending component is not yet initialized. If the sender also specifies an init value the receiver's value will be used.
resyncTime	Float	1	aggregation	Time allowed for resynchronization of data values after current data is lost, e.g. after an ECU reset.

Communication attributes specific to receiving events.

Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
queueLength	Int	1	aggregation	Length of queue for received events.

AUTOSAR Components: RPort attributes

Client-specific communication attributes.

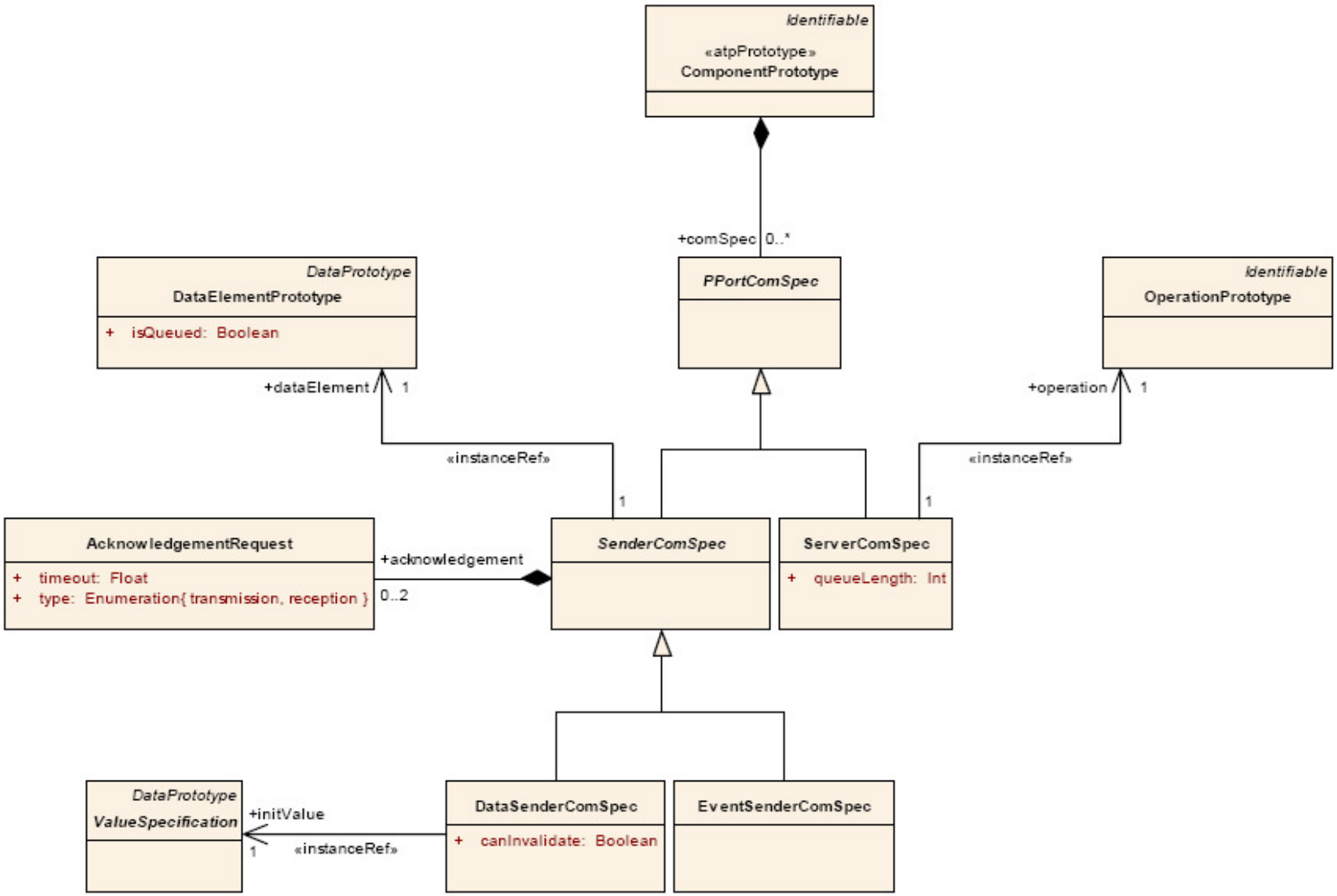
operation	OperationPrototype	1	reference to instance	Operation these attributes belong to.

Acknowledgement request attribute

Success/failure is reported via a SendPoint of a Runnable.

Attribute	***Datatype***	***Mul.***	***Link Type***	***Attribute Description***
timeout	Float	1	aggregation	Number of seconds before an error is reported or in case of allowed redundancy, the value is sent again.
type	Enumeration{ transmission, reception }	1	aggregation	Part of communication the acknowledgement is requested for. "transmission" refers reaching the receiving port, where "reception" refers to the value being actually passed to the reciving component code.

AUTOSAR Components: Pport attributes



AUTOSAR Components: PPort attributes

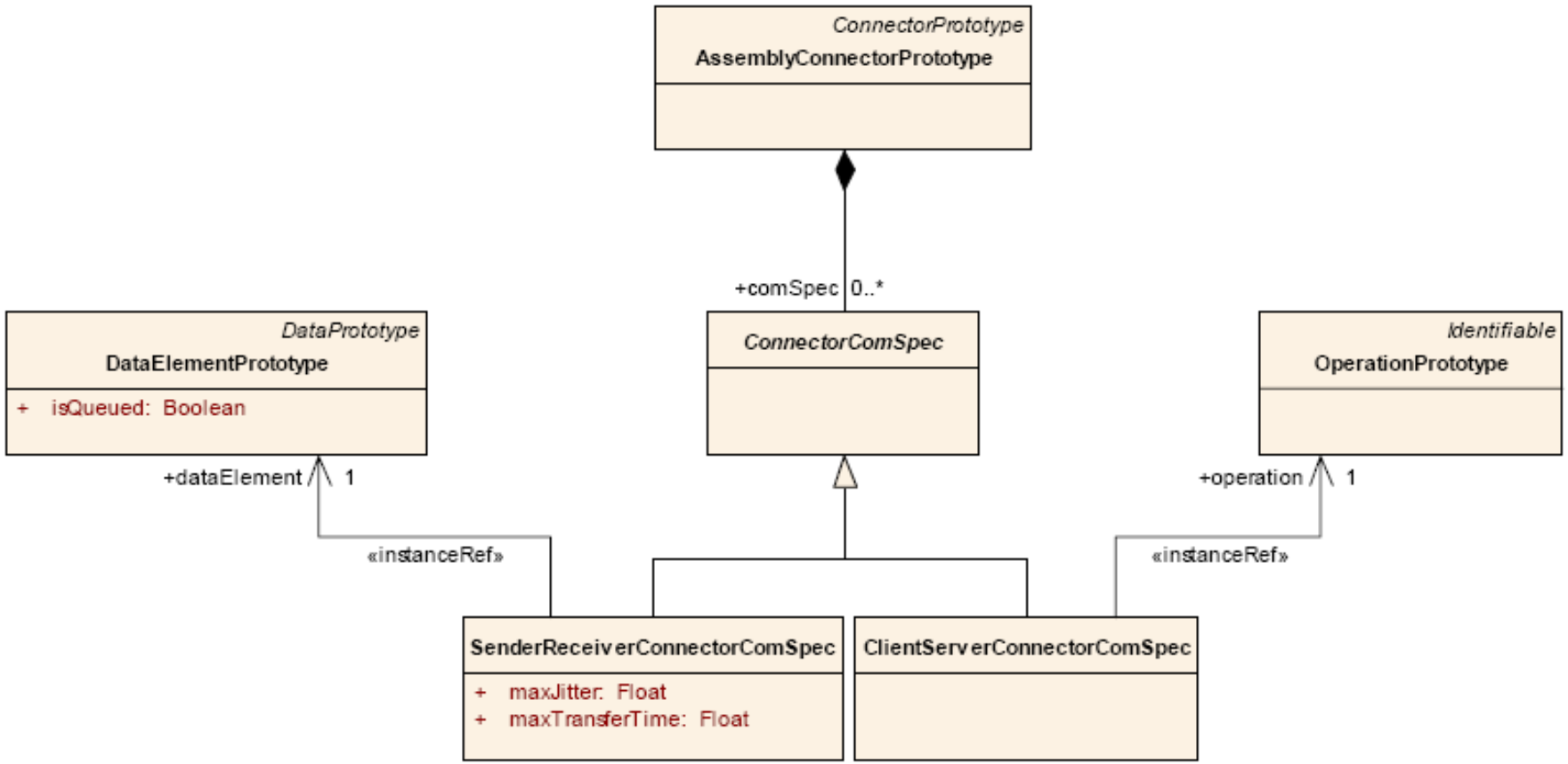
Attributes specific to distribution of data

Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
canInvalidate	Boolean	1	aggregation	Flag whether the component can actively invalidate data.
initValue	ValueSpecification	1	reference to instance	Init value to be sent if sender component is not yet fully initialized, but receiver needs data already.

Communication attributes for a server port

Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
operation	OperationPrototype	1	reference to instance	Operation these communication attributes apply to.
queueLength	Int	1	aggregation	Length of call queue on the server side. The queue is implemented by the RTE.

AUTOSAR Components: Connector attributes



AUTOSAR Components: Connector attributes

Communication attributes for connectors between sender and receiver ports

Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
dataElement	DataElementPrototype	1	reference to instance	Data element these attributes apply to.
maxJitter	Float	1	aggregation	Maximum allowed jitter as a measure for variance of transport time.
maxTransferTime	Float	1	aggregation	Maximum allowed time for transportation of data from sender to receiver in seconds.

Communication attributes for connectors between client and server ports

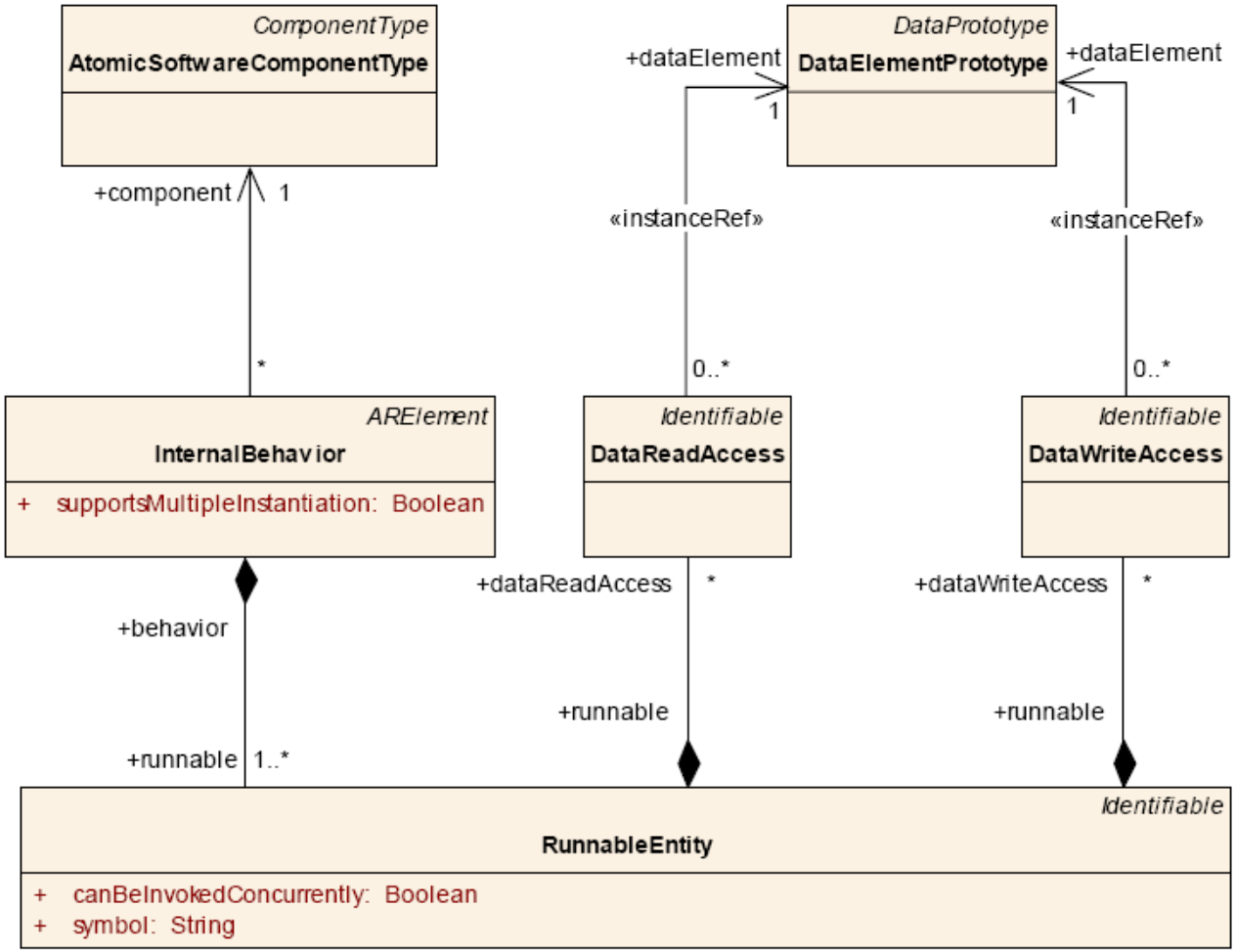
Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
operation	OperationPrototype	1	reference to instance	Operation these attributes apply to.

Runnables and communication

This section describes the sender-receiver communication relevant attributes of a component, which influence the behavior and API of the AUTOSAR RTE. Furthermore, the possible interaction patterns for application of the sender-receiver paradigm are explained, namely:

1. Data-access in a cat. 1 Runnable,
2. explicit sending,
3. the DataSendCompletedEvent: dealing with the success/failure of an explicit send, and
4. the DataReceivedEvent: responding to the reception of data

Runnables and communication



Runnables and communication

The "InternalBehavior" can specify that a Runnable needs read-access or write-access to the data-elements of an RPort or PPorts.

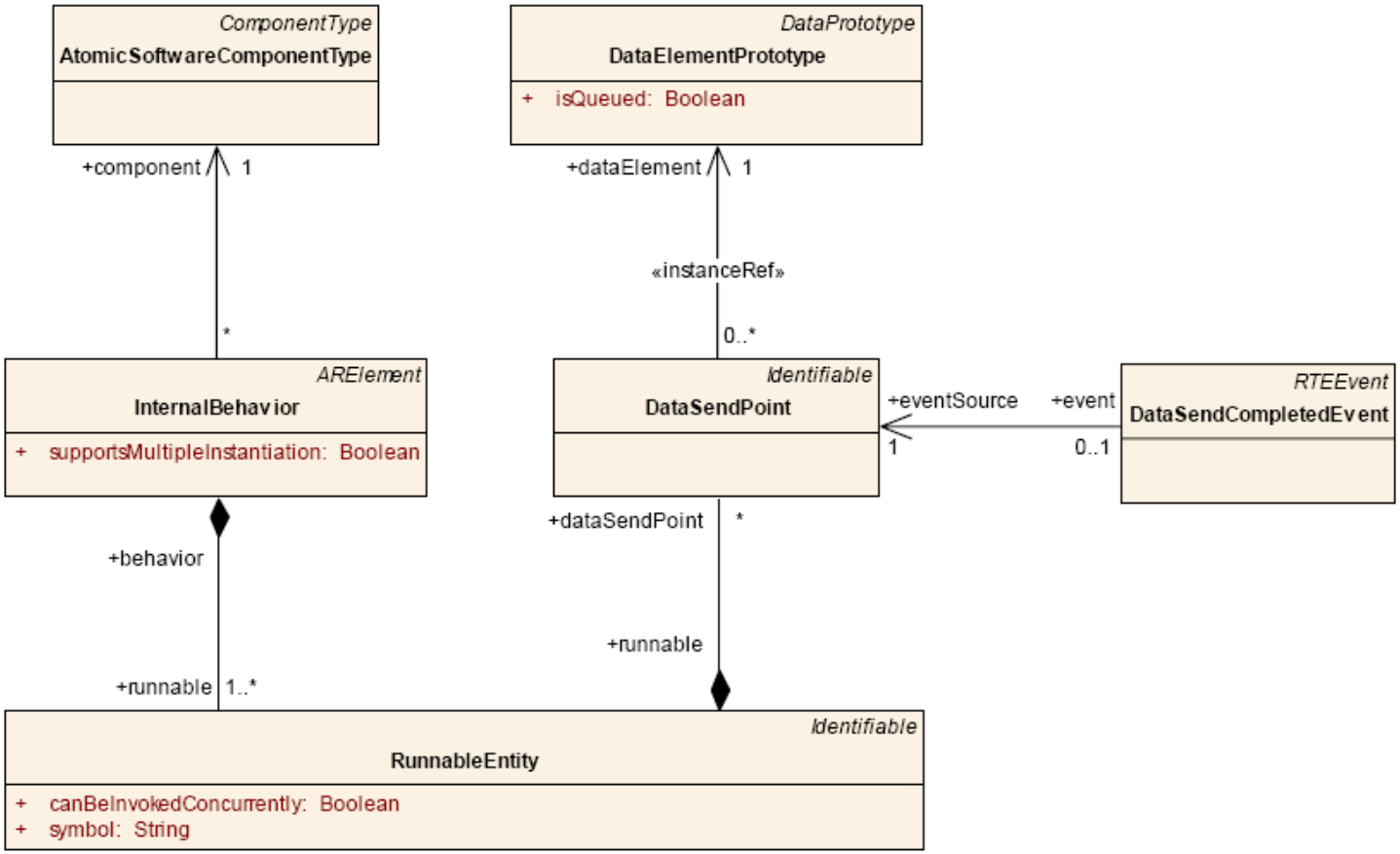
The presences of a DataReadAccess means that the runnable needs access to the DataElement in the rPort. The runnable will not modify the contents but only read the data and expects that the contents do NOT change.

The presences of a DataWriteAccess means that the runnable potentially modifies the dataElement in the pPort. The runnable must ensure that the data-element is in a consistent state when it returns. When using DataWriteAccess the new values of the data-element is only made available when the runnable returns (exits the "Running“ state).

Attribute	*Datatype*	*Mul.*	*Link Type*	*Attribute Description*
dataElement	DataElementPrototype	1	reference to instance	The data element that is going to be read by this runnable.

Runnables and communication

Explicit send and receive



Runnables and communication

The Runnable entity can also have "DataSendPoints" which reference an instance of a data element in the component's p-ports. The presence of a "DataSendPoint" means that this Runnable can explicitly "send" (an arbitrary number of times) new values of the specified data-elements of the p-port (as opposed to the "DataWriteAccess")

- In analogy to explicitly sending data it is also possible to define explicit polling for new available data through a "DataReceivePoint".

It would in general be possible to combine a "DataReceivePoint" with a "WaitPoint" in the scope of a particular "RunnableEntity". This would allow for a call to a blocking receive routine implemented by the RTE. The "timeout" attribute of meta-class "WaitPoint" can be used to specify the time until the blocking call expires.

Runnables and client-server communication

Invoking an operation

A "RunnableEntity" invokes an operation via an "RPortPrototype" of the enclosing "ComponentPrototype" typed by a particular "AtomicSoftwareComponentType". Note that the operation itself can be invoked either "synchronously" or "asynchronously".

In the majority of cases the operation will be invoked at a different "ComponentPrototype“ but in general it would be possible to invoke an operation on the same“ ComponentPrototype" as well.

The decision whether a specific operation is called synchronously or asynchronously needs to be specified in the formal description of the corresponding "AtomicSoftwareComponentType", namely in the context of an "InternalBehavior".

Runnables and client-server communication

Invoking an operation

In case of a synchronous operation invocation the particular "RunnableEntity" merely needs a "SynchronousServerCallPoint". The other case is a bit more complex because it is necessary to specify how to respond to a notification about the completion of the corresponding operation.

This is done using the generic “RTEEvent” mechanism: the notification about an asynchronously executed operation being complete is implemented as an "AsynchronousServerCallReturnsEvent".

Therefore, if an AsynchronousServerCallReturnsEvent is raised the RTE can either trigger the execution of a specific "RunnableEntity" **or** the "AtomicSoftwareComponentType“ can implement a "WaitPoint" that blocks the execution of the calling runnable until the "AsynchronousServerCallReturnsEvent" is recognized.

Runnables and communication

For example, let's consider the case of an asynchronous call to a remote operation where the RTE is supposed to trigger a specific "RunnableEntity" when the operation completes. The description of the corresponding AtomicSoftwareComponentType would typically contain the following elements:

1. The “AtomicSoftwareComponentType” contains an "RPortPrototype" '*myPort*' typed by a "PortInterface" that in turn contains the definition of an "Operation-Prototype" '*remoteOperation*'.
2. The "AtomicSoftwareComponentType's" "InternalBehavior" contains at least two "RunnableEntities": the "RunnableEntity" '*main*' is supposed to invoke the operation; the "RunnableEntity" '*callback*' is the one that should be called when the operation completes.
3. The description of the "RunnableEntity" '*main*' contains an “AsynchronousServerCallPoint” '*invokeMyOperation*' referencing the respective "OperationPrototype" in the "PortInterface" used to type the "PortPrototype" '*myPort*'. This implies that the "RunnableEntity" is allowed to invoke this operation asynchronously.
4. The description of the "AtomicSoftwareComponentType" includes an “AsynchronousServerCallReturnsEvent” '*myOperationReturns*' which references the previously defined "AsynchronousServerCallPoint" 'invokeMyOperation' out of "RunnableEntity" '*main*'.
5. The description of the “AsynchronousServerCallReturnsEvent” '*myOperation-Returns*' references the "RunnableEntity" '*callback*', indicating that the RTE should trigger the execution of this Runnable when '*myOperationReturns*' is raised.

Runnables and communication

Providing an implementation of an operation

A software-component can define an “OperationInvokedEvent” for each operation inside one of the server P-Ports. This way a Runnable may respond to such an invocation through the generic event handling mechanisms described above.

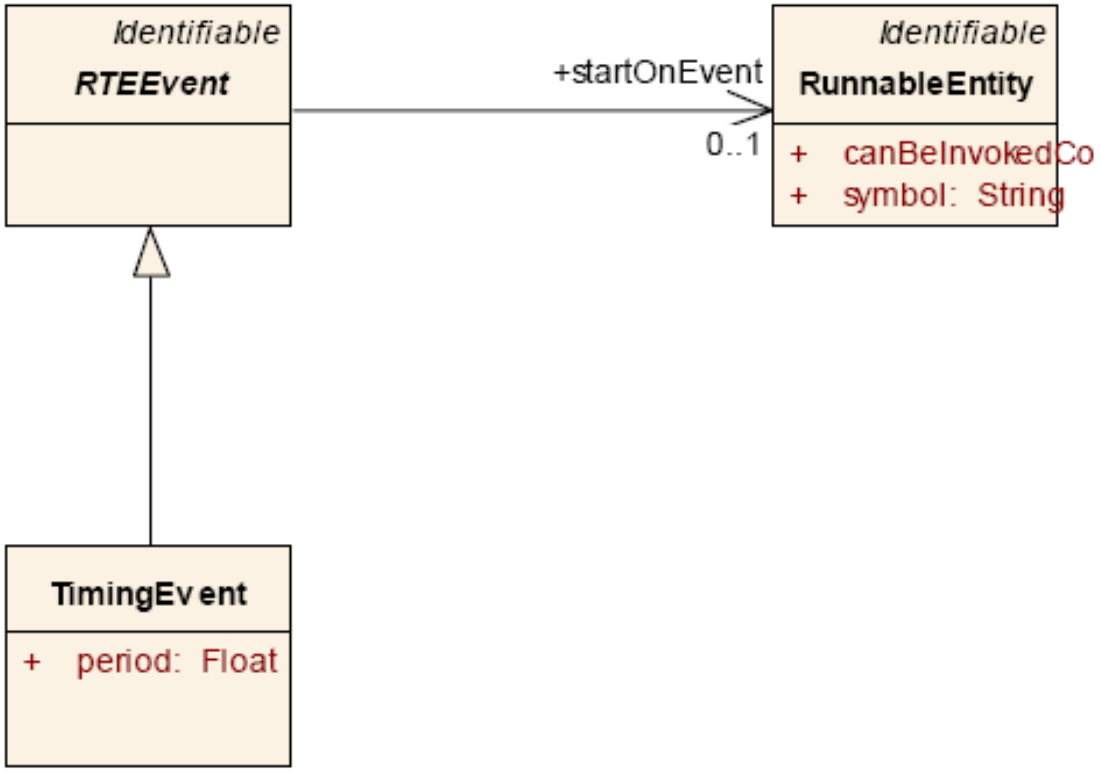
Activation of Runnables: time-driven activation

In many cases, Runnables do not need to be started by the AUTOSAR OS in response to events related to communication (e.g. the reception of a response to an asynchronous operation invocation) but to timing events. Many Runnables will need to run cyclically with a fixed rate.

The approach taken in the software-component description is to define so-called "TimingEvents" as special kinds of RTEEvents. So far, only one kind of timing event has been defined: a simple "TimingEvent", which has a period as attribute. When the internal behavior of an atomic software-component requires that the AUTOSAR OS executes certain Runnables periodically, the description will define a "TimingEvent" with the desired period.

This "TimingEvent" then contains a reference to the Runnable that needs to be executed with this period.

Runnables and communication



Runnable execution constraints

Execution order of Runnables of different software-components is affected by dataflow dependencies between the ports of the connected components.

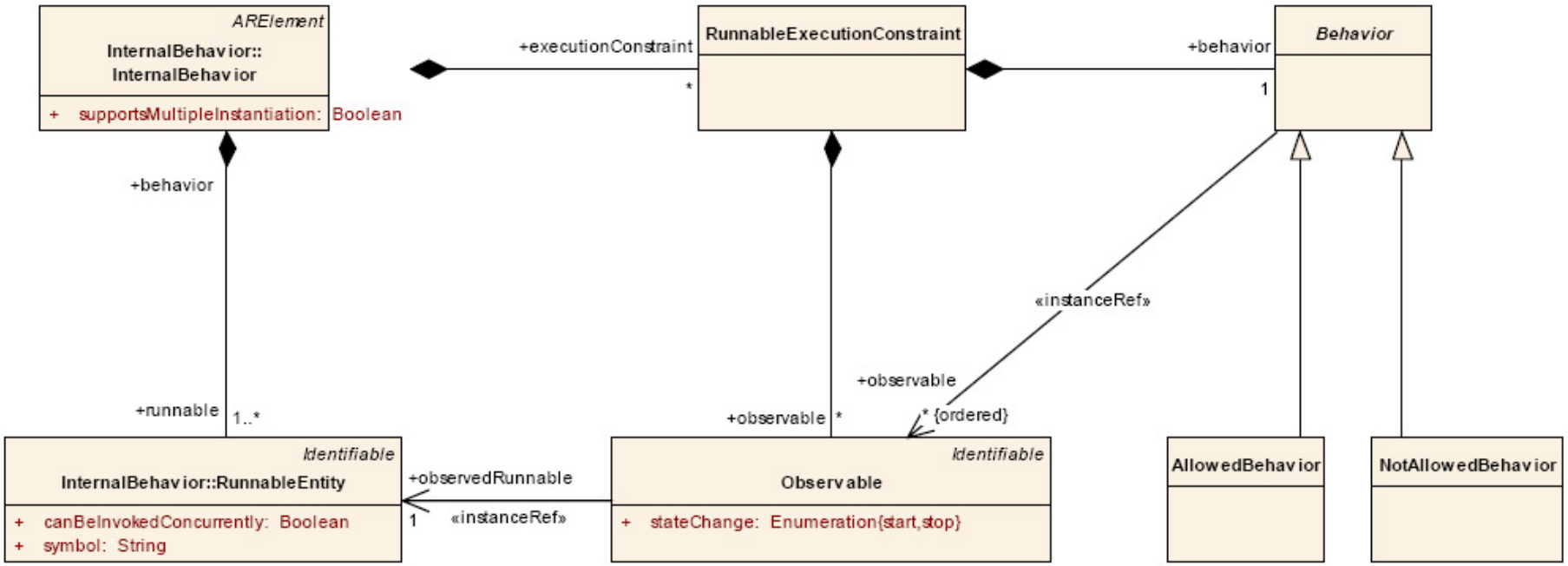
The execution order of Runnables according to data-flow dependencies can (in OSEK/CAN based systems) only be guaranteed if all affected Runnables are scheduled in one task.

In time-triggered systems (OSEKTime / FlexRay) the execution order can be guaranteed.

In addition to control-engineering driven data-flow dependencies there are additional criteria to define the execution-order of Runnables: for example some initialization Runnables must have finished executing before other runnables are allowed to start.

These dependencies can be described by “RunnableExecutionConstraints” which are aggregated to the “InternalBehavior”.

Runnables and communication



Runnables and communication

Example

The internal behavior of an atomic software-component describes three Runnables (“init”, “calculate1” and “calculate2”) and the following execution order constraints are given:

1.Runnable “init” has to terminate before the Runnable “calculate1” is allowed to start.

    1. Additionally it is not allowed that the “calculate2” starts before “calculate1” has terminated.

This can be specified by the following “RunnableExecutionConstraints”:

    1. Observables: calculate1.start, init.start, init.end
       AllowedBehavior: init.start -> init.end -> calculate1.start
    2. Observables:
       calculate2.start, calculate1.start, calculate1.end
       NotAllowedBehavior: calculate1.start -> calculate2.start

Runnables interaction within a component

RunnableEntities within a specific AtomicSoftwareComponentType typically need to communicate among each other. This implies that the RTE and/or the AUTOSAR OS need to provide synchronization mechanisms to the "RunnableEntities" such that safe (in the multi-threading sense) exchange of data is possible.

Several concepts for implementing communication among RunnableEntities can be identified.

There are various techniques to provide efficient interaction between "RunnableEntities“ within one "AtomicSoftwareComponentType".

Two possible approaches for formal specification of this kind of communication are:

- Specifying that several "RunnableEntities" belong in a specific "ExclusiveArea"
- Specifying the data exchanged between the "RunnableEntities"

Runnables interaction within a component

Communication among "RunnableEntities" can be implemented by means of "shared memory".

- RunnableEntities" within an "AtomicSoftwareComponentType“ are allocated to the same CPU.

Communication among RunnableEntities can then establish a data flow scheme (a very popular pattern in the application of control theory to automotive embedded systems). If global variables are used for establishing inter-RunnableEntity communication they acquire the semantics of so-called *state-messages*.

Nevertheless, directly sharing memory between RunnableEntities requires a serious problem to be solved: the guarantee of data consistency among communicating "RunnableEntities".

Runnables and communication

Note that this approach closely resembles the communication principle underlying "DataReadAccess" and "DataWriteAccess".

The counterpart to "DataReadAccess" and "DataWriteAccess" within the AUTOSAR meta-model is "DataReceivePoint" and "DataSendPoint". These allow for an immediate access to the underlying communication item. It should be possible to specify the same semantics even for communication among "RunnableEntities“ of the same "AtomicSoftwareComponentType".

The following paragraphs describe some common strategies that can be used to ensure the required data-consistency. We do not attempt to describe the pros or cons of these approaches.

Runnables and communication

Scheduling strategy

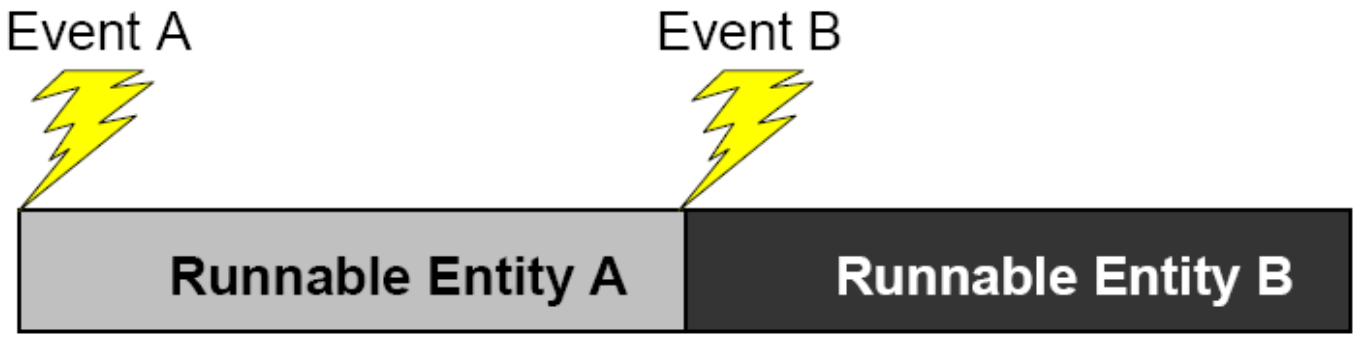
A first strategy for guaranteeing data consistency of concurrently accessed variables is based on a defined scheduling strategy:

Execute **a "RunnableEntity“ such that it can never be preempted by another "RunnableEntity“ that might modify the memory being read by the first one**.

As nearly all embedded AUTOSAR OS only allow for a single instance of each task running at the same time, this can be achieved by putting all "RunnableEntities" that interact by sharing memory into a single task.

The task must be instrumented (i.e. the execution of each "RunnableEntity" must be guarded by a flag or something similar) such that it is possible to suppress the execution of particular "RunnableEntities".

Runnables and communication



Runnables and communication

As a possible execution model, an (OSEK) event shall be associated with each "RunnableEntity" of a (OSEK ECC) task. Then it is easy to guard the execution of the "RunnableEntity" depending on the corresponding event.

In other words: the task is in *wait* state until one of the possible events occur in which case the task is *ready* for transition into state *running*. As a consequence of this transition, the "RunnableEntity" corresponding to the event is executed. This principle works even in case several events occur simultaneously.

"RunnableEntities" of a single task can obviously not be executed concurrently. A "RunnableEntity" can only be executed if other "RunnableEntities" of the same task have finished their execution. This could lead to a certain non-deterministic delay between the recognition of an event and the execution of the corresponding "RunnableEntity". It must be decided whether this delay can actually be tolerated by a particular application.

As suggested before, this concept requires the usage of a more advanced scheduling policy. In particular, the capabilities of OSEK ECC tasks (or similar, if a non-OSEK OS is used) are a prerequisite for the implementation of this concept.

Runnables and communication

Mutual exclusion with semaphores

Multi-threaded operating systems provide mutexes (mutual exclusion semaphores) that protect access to an exclusive resource that is used from within several tasks.

The RTE could use these OS-provided mutexes to make sure that the "RunnableEntites“ sharing a memory-space would never run concurrently. The RTE would make sure the task running the "RunnableEntity" has taken an appropriate mutex before accessing the memory shared between the "RunnableEntities".

Runnables and communication

Interrupt disabling

Another alternative would be the disabling of interrupts during the run-time of "RunnableEntities“ or at least for a period in time identical to the interval from the first to the last usage of a concurrently accessed variable in a "RunnableEntity". This approach could lead to seriously non-deterministic execution timing.

Runnables and communication

Priority ceiling

Priority ceiling allows for a non-blocking protection of shared resources. Provided that the priority scheme is static, the AUTOSAR OS is capable of temporarily raising the priority of a task that attempts to access a shared resource to the highest priority of all tasks that would ever attempt to access the resource.

By this means is technically impossible that a task in temporary possession of a resource is ever preempted by a task that attempts to access the resource as well.

Runnables and communication

Implicit communication by means of variable copies

Another alternative is the usage of copies of concurrently accessed variables with state message semantics.

This means in particular that for a concurrently used variable a copy is created on which a "RunnableEntity" entity can work without any danger of data inconsistency.

This concept requires additional code to write the value of the concurrently accessed variable to the copy before the "RunnableEntity" that accesses the variable is executed.

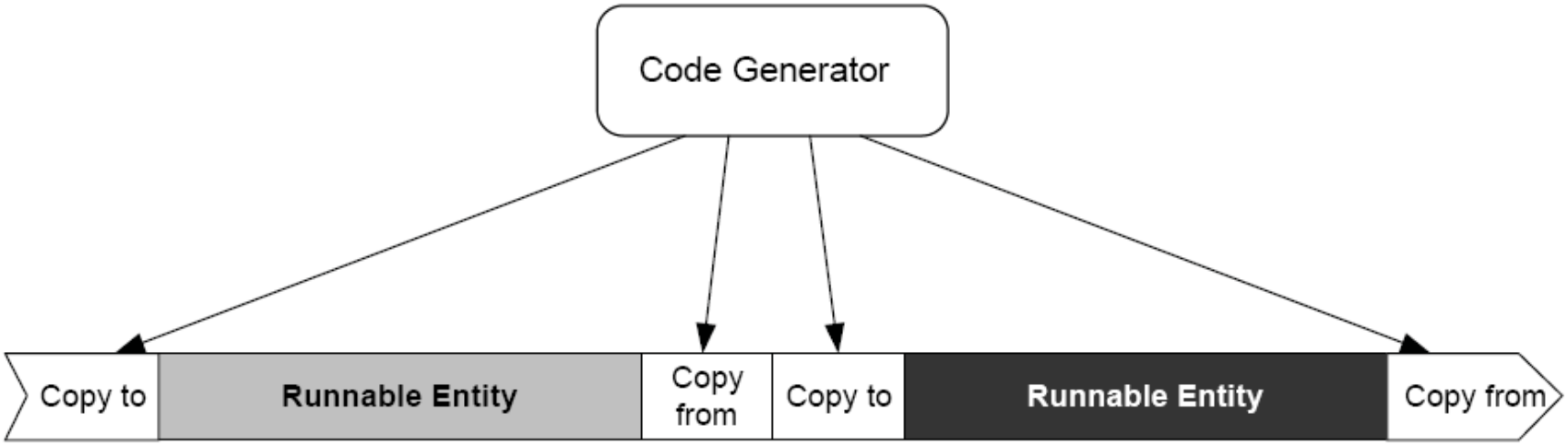
The value of the copy must be written back to the concurrently accessed variable after the "RunnableEntity" has been terminated.

Since it would be too expensive and errorprone to manually care about the copy routines it would be a good idea to leave the creation of the additional code to a suitable code generator.

Runnables and communication

It is possible to further optimize the process by reducing the additional code at the beginning and end of each task, for example, copy routines will only be inserted where appropriate, e.g. a copy routine for writing the value of a copy back to the concurrently accessed variable will only be inserted if the "RunnableEntity" has write access to the variable.

The copy routines have to make sure that the copy process is not interrupted in order to be capable of consistently copying the values from and to the shared variable. These periods, however, are supposed to be very short compared with the overall run-time of the "RunnableEntity".

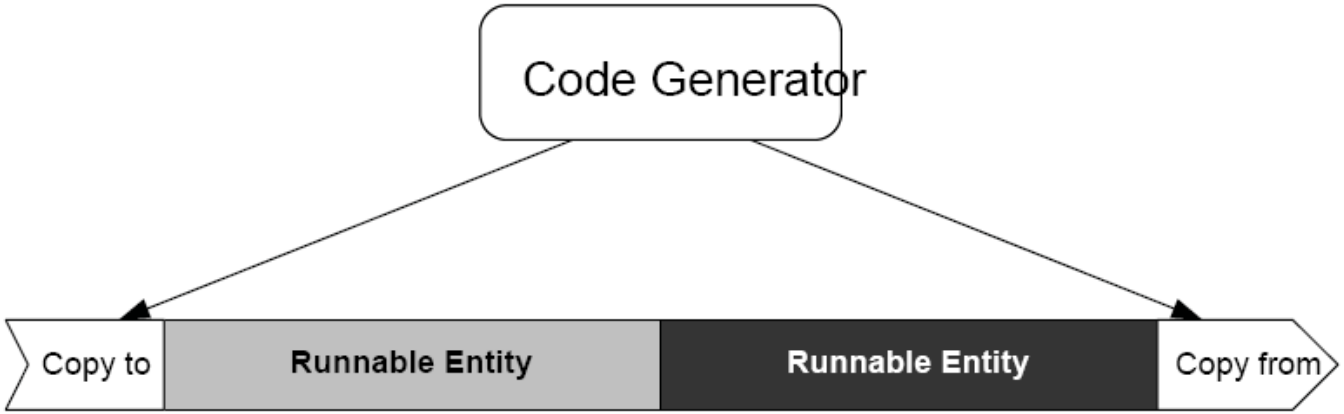


Runnables and communication

Further optimization criteria can be applied, for example: it would be perfectly safe to avoid the creation of copies for runnables that are scheduled in the task with the highest priority of all tasks that (via contained runnables) access a certain concurrently accessed variable.

In order to keep the application code free of any dependencies from the code generation, access to concurrently accessed variables will be guarded by macros that are later resolved by the code generator.

The presence of the guard macros directly supports the reuse on the level of source code. The reuse on the level of object code is only possible if the scheduling scenario (in terms of the assignment of "RunnableEntities" to priority levels) does not change. This concept can only be implemented properly with the aid of a code generator if the variables in question can be identified. In other words: the description of a software component has to expose all concurrently accessed variables to the outside world.



Runnables and communication

Description possibility 1: "ExclusiveArea“ (critical section)

This section describes how the concept of "ExclusiveAreas" can be used in the description of the "InternalBehavior" of an "AtomicSoftwareComponentType". These "ExclusiveAreas" do not imply a specific implementation (e.g. with mutual-exclusion semaphores).

They just specify a constraint on the scheduling policy and configuration of the RTE: If two or more “RunnableEntities” refer to the same “ExclusiveArea” only one of these "RunnableEntities" is allowed to be executed while being inside that “ExclusiveArea”. In other words: these "RunnableEntities" must not run concurrently (pre-empt each other) while executing inside the “ExclusiveArea”.

An attribute "executionOptimization" can provide hints for ECU configuration. The possible values are "executionTime" and "codeSize". The first hints to care for an efficient implementation in terms of execution time while the latter suggests focusing on code size.

Runnables and communication

There are in general two ways to use the "ExclusiveAreas".

In the first approach, the formal description specifies that certain "RunnableEntities“ always run inside an exclusive area.

For example, if the formal description specifies that both "RunnableEntity" 'r1' and "RunnableEntity" 'r2' run within "ExclusiveArea“ 's1', the RTE in collaboration with the scheduler must make sure that "RunnableEntities" 'r1' and 'r2' never run concurrently; the scheduler should never preempt 'r1' to run 'r2'.

This requirement could be implemented by several of the implementation strategies described above.

In the second approach, the runnable would explicitly make API-calls to the RTE within the implementation of the "RunnableEntity" to enter and leave a specific "ExclusiveArea".

This could, for example, be implemented by means of *priority ceiling*

Resource consumption

AUTOSAR SW-Components need to be mapped on ECUs at some point during the development. The mapping freedom is limited by the System Constraints and the available resources on each ECU.

The SW-Component description provides information about the needed resources concerning memory and execution time for each AtomicSoftwareComponentType. The hardware resources are going to be used by all software on that ECU, including OS, Basic SW, RTE, ECU abstraction, CCD, Services.

The resource consumption of the other software on an ECU (OS, RTE, Basic SW,...) is not covered by the AUTOSAR SW-Component template explicitly although the template might be used to capture the memory and execution time consumption of a specific configuration of the Basic SW.

- Some of these resources are highly dependent on the configuration actually mapped on the ECU. So an iterative resource description and estimation is needed to cover the RTE and Basic SW resource needs.

Resource consumption

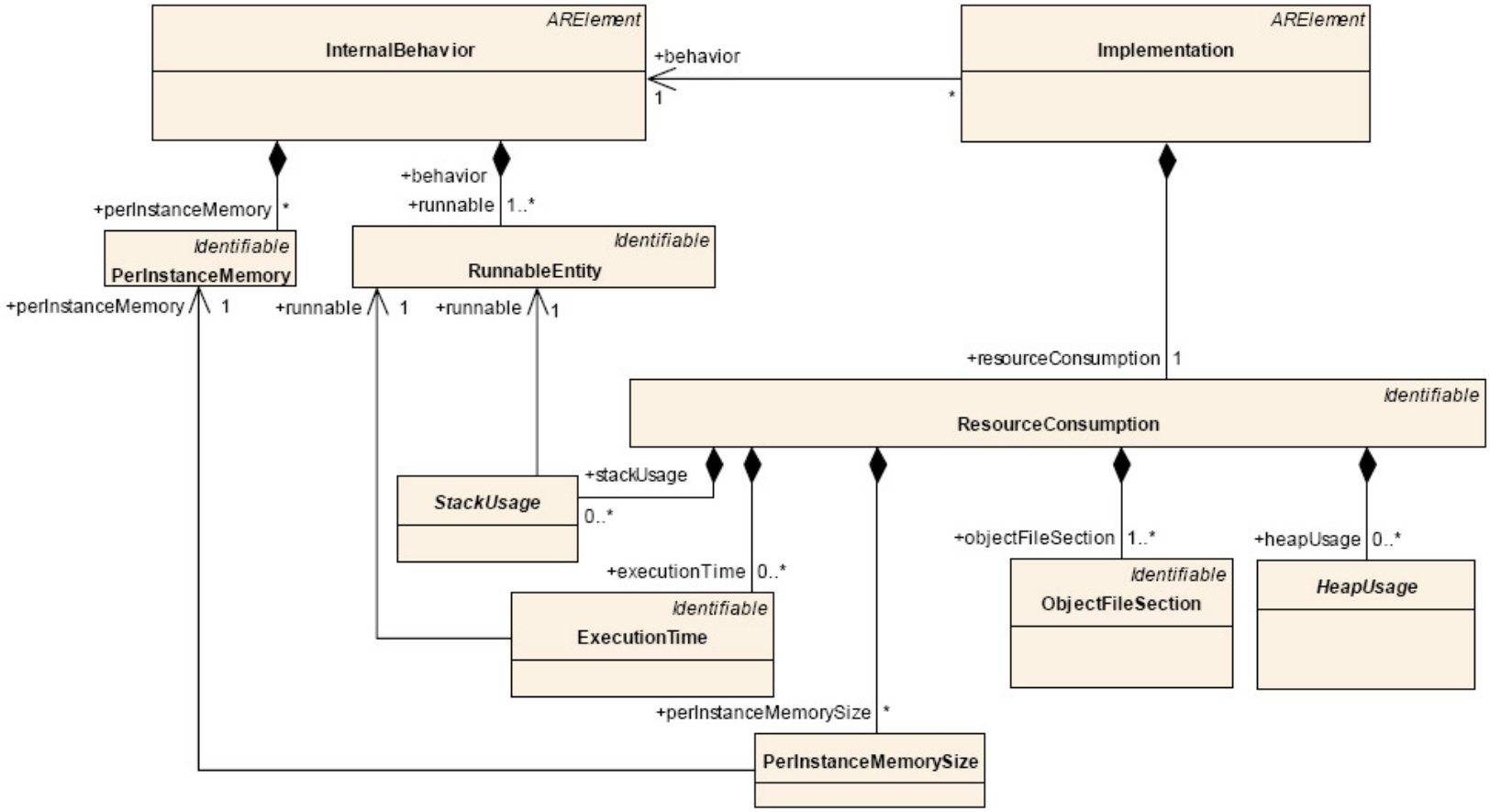
Resources can be divided into static and dynamic resources. Static resources can only be allocated by one entity and stay with this entity. If the required amount of resources is bigger than the available resources the mapping does not fit physically. ROM is an example of a spare resource where obviously only the amount of data can be stored that is provided by the storage capacity.

Dynamic resources are shared and therefore can be allocated dynamically to different control threads over time. Processing time is an example, where different "RunnableEntities" are given the processor for some time.

If some runnable entity uses more processing time than originally planned, it can lead to functional failure. Also some sections of RAM can be seen as dynamic resources (e.g. stack, heap which grow and shrink dynamically).

Resource consumption

The resource consumption is attached to an Implementation of an AtomicSoftwareComponentType. For each Implementation, there can be one ResourceConsumption description.



Resource consumption

All resources are described within the ResourceConsumption meta-class.

ExecutionTime and StackUsage are used to provide information on the implementation-specific resource usage of the runnables defined in the “InternalBehavior”.

“PerInstanceMemorySize” provides information regarding the actual size (in bytes) of the “PerInstanceMemory” defined in the “InternalBehavior”.

“ObjectFileSection” documents the resources needed to load the object-file containing the implementation on the ECU.

“HeapUsage” describes the dynamic memory usage of the component.

Resource consumption

Relation to the hardware description

In this section the relationships between the description methods of the ECU Resource template and the SW-Component resource needs are discussed. Only the memory description and the processing time description are covered.

Memory

The ECU resource template describes the total available memory due to the hardware characteristics, not the actual implementation technology. Therefore memory implementation names like EEPROM, FLASH or DRAM are not used in the description of an ECU.

The main criteria distinguishing memory is the volatile - non volatile category. First the attributes for volatile memory are discussed, then the additional attributes for non volatile memory will be introduced.

Resource consumption

Execution time

The description mechanism is defined how actual execution times for specific hardware can be provided.

The ECU Resource template description document introduces a different description mechanism which is based on some benchmarking technology.

The execution time is an ASSERTION: a statement about the duration of the execution of a piece of code in a given situation.
The execution time is NOT A REQUIREMENT on the software-component, on the hardware or on the scheduling policy.

Resource consumption

A description of the execution time of a runnable entity of an implementation of an atomic software-component should include:

- the nominal execution time (“0.000137 s”) or a range of times
- a description of the entire context in which the execution-time measurement or analysis has been made
- some indication of the quality of this measurement or estimation

The goal thereby is that the template finds a good compromise between flexibility and precision.

The description must be flexible enough so that the entire range between analytic results (“worst-case execution time”) and rough estimates can be described.

The description should be precise enough so that it is entirely clear what the relevance or meaning of the stated execution time is. This implies that a large amount of context information needs to be provided.

Resource consumption

The execution time can be described for a specific sequence of assembly instructions. It does not make sense to describe the execution time of a runnable provided as source-code.

In addition, the execution-time of such a sequence of assembly instructions depends on:

- the hardware-platform
- the hardware state
- the logical (software) context
- execution-time of external pieces of code called from the runnable

These dependencies are discussed in detail in the following.

Resource consumption

Dependency of the execution time on hardware

The execution-time depends both on the CPU-hardware and on certain parts of the peripheral hardware:

- The execution time depends on a complete description of the processor, including:
- kind of processor (e.g. „PPC603“)
- the internal Processor frequency („100 MHz“)
- amount of processor cache
- configuration of CPU (e.g. power-mode)

Aspects of the periphery that need to be described include:

- external bus-speed

Resource consumption

MMU (memory management unit)

- configuration of the MMU (data-cache, code-cache, write-back,...)
- external cache
- memory (kind of RAM, RAM speed)

In addition, when other devices (I/O) are directly accessed „as memory“, the speed of those devices has a potentially large influence.

On top of this, the ECU might provide several ways to store the code and data that needs to be executed. This might also have a large influence on the execution time.

For example:

- execution of assembly instructions stored in RAM vs. execution out of ROM may have different execution times
- when caching is present, the relative physical location of data accessed in memory influences the execution time

Resource consumption

Dependency on hardware state

In addition to the static configuration of the hardware and location of the code and data on this hardware, the dynamically changing state of the hardware might have a large influence on the execution time of a piece of code : some examples of this hardware state are:

- which parts of the code are available in the execution-cache and what parts will need to be read from external RAM
- what part of the data is stored in data-cache versus must be fetched from RAM
- potentially, the state of the processor pipeline

Despite the potential importance of this initial hardware-state when caching is present, it is almost impossible and definitely impractical to describe this hardware state.

Therefore it is important and clear that AUTOSAR does not provide explicit attributes for this purpose.

Resource consumption

Dependency on logical context

This logical context includes:

- the input parameters with which the runnable is called
- the logical “state” of the component to which the runnable belongs (or more precisely: the contents of all the memory that is used by the runnable)

While a description of the input-parameters is relatively straight-forward to specify, it might be very hard to describe the entire logical state that the runnable depends on.

In addition, in certain cases, one wants to provide a specific (e.g. measured or simulated) execution time for a very specific logical context; whereas in other cases, one wants to describe a “worst-case execution time” over all valid logical contexts or over a subset of logical contexts.

Resource consumption

Dependency on external code

Things get very complex when the piece of code whose execution time is described makes calls into (“jumps into”) external libraries.

To deal with this problem, we could take one of the following approaches:

- Do not support this case at all
- Support a description of the execution time for a very specific version (again at object-code level) of the libraries. The exact versions would be described together with the execution time.
- Conceptually, it might be possible to explicitly describe the dependency on the execution-times of the library. This description would include:
  - the execution time of the code provided by the component itself
  - a specification of which external library-calls are made

Option 3 is deemed impractical and is not supported.

Option 2 however is important as many software-components might depend on very simple but very common external libraries (a math-library that provides floating point capability in software).

Resource consumption

Description-model for the execution time

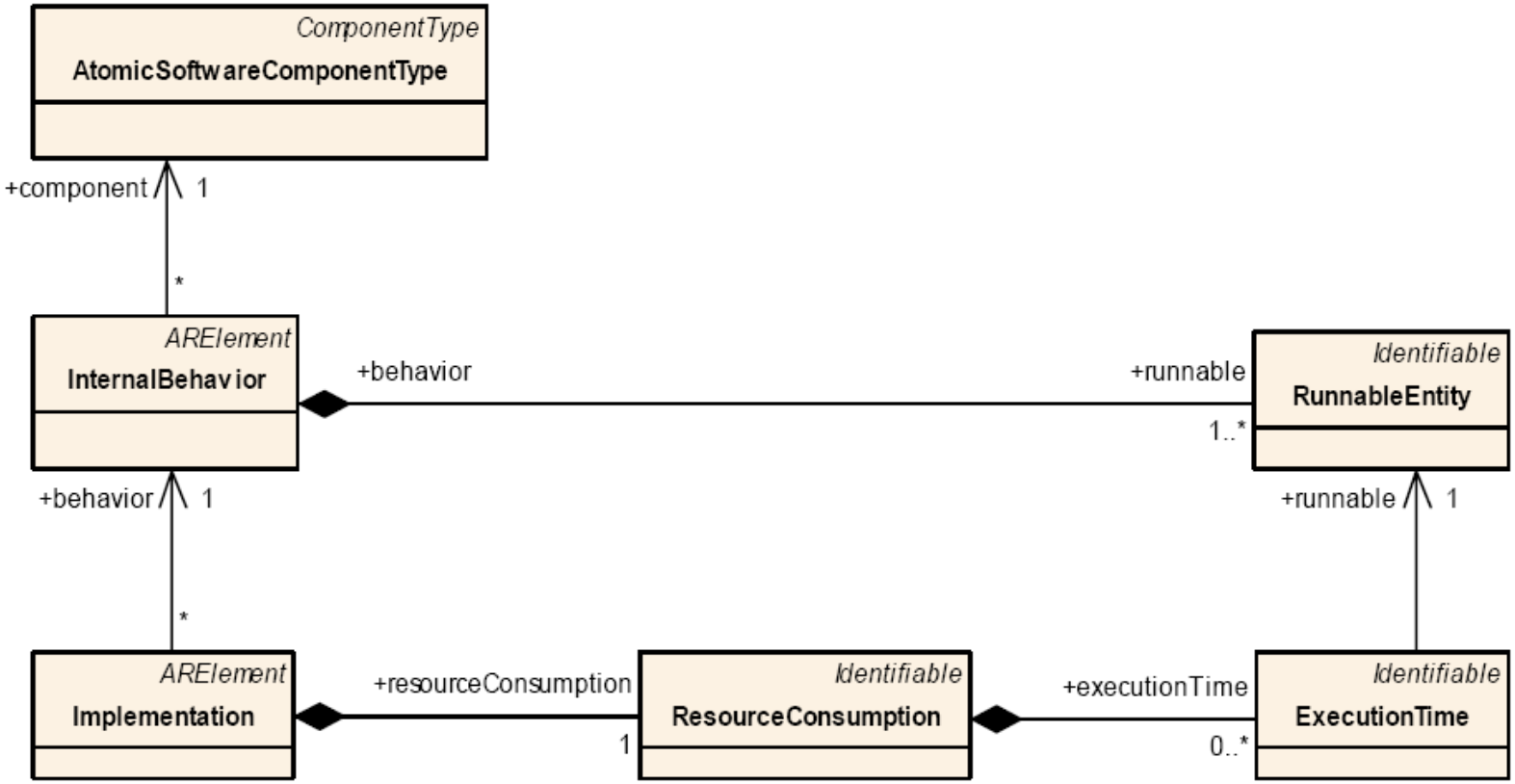
The description of the implementation of a component references the description of the internal behavior of the atomic software-component that is implemented The description of the internal behavior describes all the "runnable entities" of the component.

Each description of such a runnable entity (of a specific implementation) can include an arbitrary number of execution-time descriptions. Thereby this execution time description may also depend on code or data variant of the implementation.

It is expected that many runnable entities will not have execution-time descriptions. For runnable-entities that do have execution-time descriptions, the componentimplementor could provide several execution-time descriptions: for example one per specific ECU on which the implementation can run and on which the time was measured or estimated.

Resource consumption

How the execution time is part of the overall description of the implementation of a component.



Resource consumption