

<b>Document Title</b>	Applying ASCET to AUTOSAR
<b>Document Owner</b>	AUTOSAR
<b>Document Responsibility</b>	AUTOSAR
<b>Document Identification No</b>	227
<b>Document Classification</b>	Auxiliary

<b>Document Version</b>	1.0.4
<b>Document Status</b>	Final
<b>Part of Release</b>	3.2
<b>Revision</b>	1

<b>Document Change History</b>			
<b>Date</b>	<b>Version</b>	<b>Changed by</b>	<b>Change Description</b>
23.11.2011	1.0.4	AUTOSAR Administration	Legal disclaimer revised
23.06.2008	1.0.3	AUTOSAR Administration	Legal disclaimer revised
31.10.2007	1.0.2	AUTOSAR Administration	<ul style="list-style-type: none"> <li>• Document meta information extended</li> <li>• Small layout adaptations made</li> </ul>
24.01.2007	1.0.1	AUTOSAR Administration	<ul style="list-style-type: none"> <li>• “Advice for users” revised</li> <li>• “Revision Information” added</li> <li>• Legal disclaimer revised</li> </ul>
04.12.2006	1.0.0	AUTOSAR Administration	Initial release

## **Disclaimer**

This specification and the material contained in it, as released by AUTOSAR is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the specification.

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

This specification may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only.

For any other purpose, no part of the specification may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

## **Advice for users**

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

## Table of Contents

1	Preliminary Remarks .....	5
2	Introduction to AUTOSAR concepts .....	6
3	ASCET Use-Cases for AUTOSAR .....	10
3.1	Integration Tool Use-Case .....	11
3.2	Additional Programmer Use-Case .....	12
4	Migration of Legacy ASCET Projects to AUTOSAR Software Components.....	14
4.1	Methodology .....	14
4.2	Implementation of the ASCET message concept. ....	14
4.3	Scenario for ASCET Projects and Modules .....	15
4.4	Scenario for ASCET Classes.....	16
5	ASCET & AUTOSAR Overview.....	18
5.1	Major ASCET Concepts for AUTOSAR .....	18
5.2	Clustering.....	20
6	Creating ASCET Projects in Dedicated AUTOSAR Style.....	22
6.1	Example System.....	22
6.2	Atomic Software Component Types .....	22
6.2.1	DataTypes .....	24
6.2.2	PrimitiveTypeWithSemantics .....	26
6.2.3	Summary of “DataType”entities .....	28
6.2.4	Interfaces.....	28
6.2.5	PortPrototypes .....	31
6.2.5.1	SenderReceiverInteface.....	31
6.2.5.2	ClientServerInterface .....	31
6.2.6	AtomicSoftwareComponentType Entities .....	32
6.2.7	Internal Behavior.....	32
6.2.7.1	SingleInstanceTypes with an arbitrary number of (R- and P-) PortPrototypes realizing SenderReceiverInterfaces with one DataElementPrototype .....	33
6.2.7.2	MultipleInstanceTypes with an arbitrary number of (R- and P-) PortPrototypes realizing SenderReceiverInterfaces with one DataElementPrototype .....	35
6.2.7.3	MultipleInstanceType with one PPortPortotype realizing a ClientServerInterface with OperationPrototypes having one OUT ArgumentPrototype.....	39
6.3	Creating SoftwareComponentPrototype Entities in ASCET .....	41
6.3.1	Wrapper Runnables.....	41
6.3.2	Port-Type-Converters .....	43
7	The ECU Composition.....	46
8	Summary .....	50

9 References ..... 51

## 1 Preliminary Remarks

This document is split into two parts. The first part (chapters 2, 3 and 4) addresses ASCET users who want to migrate existing ASCET projects to AUTOSAR platforms. To describe the basic principles, the wording is chosen in line with the virtual functional bus description which, however, blurs the terms of the software component template. Especially, the type/prototype concept is blurred.

The second part (chapters 5,6, and 7) describes how ASCET can be used to build AUTOSAR software components from scratch. It is shown how ASCET concepts can be combined to produce flexible and efficient AUTOSAR systems. Here the wording is strict in the AUTOSAR software component template sense.

This document is not a roadmap of features for ASCET products. It just shows how ASCET concepts can be used in an AUTOSAR software component modeling context.

## 2 Introduction to AUTOSAR concepts

The basic idea of AUTOSAR is to decouple the control-engineering functionality (represented by so-called application software components) from the basic software (represented by standardized but configurable basic software modules). This idea has two major implications.

1. The control-engineering functionality is developed independently from an underlying ECU or even an E/E-Architecture. The ensemble of application software components constitute the Virtual Functional Bus description. Mapping of application software components to ECUs is done at later stages in the design.
2. An AUTOSAR ECU will run a lot of standardized basic software modules surplus a runtime environment RTE. Mapped application software components use this RTE to communicate with other application software components via this RTE where it does not matter whether the other application software component is located on the same ECU or on another ECU in the vehicle network. In the first case, RTE will route the data to be exchanged using the ECUs memory while in the latter case the, AUTOSAR COM stack will be invoked.

The AUTOSAR idea is depicted in Figure 1 where the VFB with its application software components and their connectors is shown on the top, the mapping process using the ECU and network description in the middle while ECU oriented view is shown in the bottom. The leftmost ECU shows a more detailed view of the mapped application software components, the RTE and other basic software modules like the AUTOSAR COM stack. This detailed view is also known as ECU-software architecture and is neither to be mismatched with an application software component architecture or an Electric/Electronic- (E/E) Architecture.

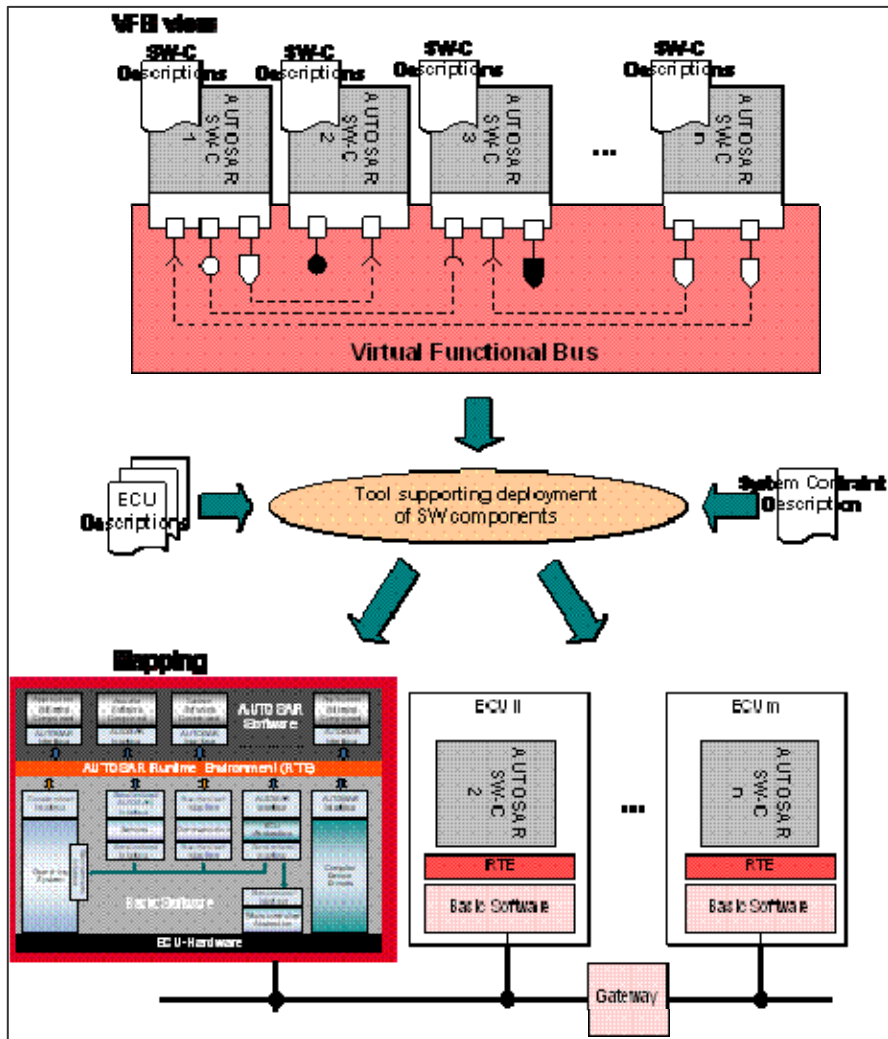


Figure 1: The AUTOSAR Idea

The virtual functional bus describes the outer-view of an atomic-software-component. Besides this outer-view, there is also an inner-view. This inner-view is called internal behavior and describes how the component's implementation interacts with its own ports. Furthermore, this view is mandatory to understand how the component should be scheduled on an ECU. The internal behavior is composed of so-called runnable entities, or runnables for short. These runnables are sequences of instructions that can be started by the run-time-environment (RTE). A "runnable entity" runs in the context of a "task". The task provides the common resources to the "runnable entities" such as a context and stack-space. Typically the operating-system scheduler has the responsibility to decide during run-time when which "task" can run on the CPU (or multiple CPUs) of the ECU.

The internal behaviour with its runnables is associated to an software component, which interact with other software-components just by ports. To perform their tasks, the runnables have to read the data from ports and write data to ports. The interaction mechanisms are listed below:

- The runnable can interact with ports by simply reading or writing variables (this is called DataReadAccess, resp. DataWriteAccess)
- The RTE can start certain runnables when new data arrives on a port or a certain time has elapsed

- The runnables might want to explicitly send data, receive data or invoke Operations

"DataAccess" means that the runnable entity does not need to invoke operations on the RTE but is rather given the locations where it can read and/or write the information.

This pattern implies:

1. it is the responsibility of the RTE to make sure that the information needed by the runnable entity is present and available while the runnable entity is running
2. the communication mechanism between the RTE and the runnable entity can be made very efficient.

This simple interaction pattern is appropriate for runnable entities that are simple but deterministic in behavior. A runnable can request `DataReadAccess` to a `DataElement` available on an RPort of the component or `DataWriteAccess` to a `DataElement` provided over a PPort of the component.

The presence of a `DataReadAccess` means that the runnable needs access to the value of the specified `DataElement` during the entire time that the runnable runs. The runnable will not modify the contents of the data but only read the information. The runnable expects that the contents of this data does NOT change during the execution of the runnable. The presence of a `DataWriteAccess` means that the runnable will potentially modify the specified `DataElement`. The runnable has free access to the data-element while it is running. The runnable has the responsibility to make sure that the data-element is in a consistent state when the runnable returns. The `DataElement` will only be sent after the runnable terminates. Note that this mechanism makes the most sense for the case that the Sender/Receiver communication mechanism is used for data-transfer. The semantics are not clear in case the Sender/Receiver communication is used to transfer "events". The RTE software specification defines the API in the C- programming-language constructs that are used to realize this data-access pattern.

At a first glance, the internal behavior concept of AUTOSAR is quite similar to ASCET. Runnables match with processes in that they are allocated to tasks, and that a runnables have data-access points to data-elements on ports which is similar to reading from a receive-message and writing to a send-message in a sequence-call of a process. Furthermore, a module groups processes just like the internal behavior the runnables.

From the virtual functional bus view, the differences predominate. Data-Elements can be grouped in interfaces in AUTOSAR while messages a merely scalar entities. Then, within an ASCET project, messages are matched by name, i.e. data on a send-message will only appear at a receive-message when it has the same name, while AUTOSAR requires an explicit connection between ports.

Table 1 gives a first overview of how AUTOSAR concepts are realized by ASCET concepts. This table is refined later on an proper software-component-template terms are used.

<b>AUTOSAR Concept</b>	<b>AUTOSAR Description</b>	<b>ASCET Concept</b>
DataElement	Data in SenderReceiver Interface	Message
SenderReceiver Interface	Clusters signals	Implicitly given



PPort	Access Point in component where data is provided	Send-Message at SendReceive Interface
RPort	Access Point in component where data is required	Receive Message at SenderReceiver Interface
Runnable	Behavioral Code	Process
Atomic Software Component	Tentative Instance of an atomic software component.	Module

**Table 1: First-Glance matching of AUTOSAR and ASCET concepts**

### 3 ASCET Use-Cases for AUTOSAR

ASCET does not directly distinguish between application software components and basic software modules, but in the use-cases *integration tool* and *additional programmer*. In both use-cases, the ASCET project is the mean to generate code.

An ASCET project clusters a number of modules, which communicate via messages. Within a project, messages have to have a unique name. They are exported by one module (e.g. as send-message) and imported by another module (e.g. as receive-message). Control-engineering functionality is modeled using so-called processes, which are associated to modules. Processes access the messages of the module. An ASCET project also provides the means to group processes into tasks and assign trigger conditions to the task, e.g. cyclic or interrupt driven activation.

The ASCET programming model is given by the ERCOSEK operating system. Therefore, the process task association can be used to generate the code of a whole ECU out of an ASCET project. Of course, this requires that all basic software modules are described in ASCET using e.g. C-Code modules. This use-case of ASCET is known as integration tool.

However, the ASCET project can be used differently. In this case, a project will be composed of one or several modules and no task allocation of processes is done. If one now applies code-generation to that project, C-Code will be generated for the processes of the modules as well as all messages. This code can then be integrated in an arbitrary ECU-configuration tool. Allocation of messages to memory as well as the allocation of processes to tasks has to be done in that ECU-configuration tool on C-Code level.

All send-messages of a module in a project which have no corresponding receive-message (in the same or in other modules of the project), will become visible to the project outside as well as all receive-messages which have no send-message within the project. Thus, the list of all non-matched messages establishes the interface of a project with other software components on an ECU.

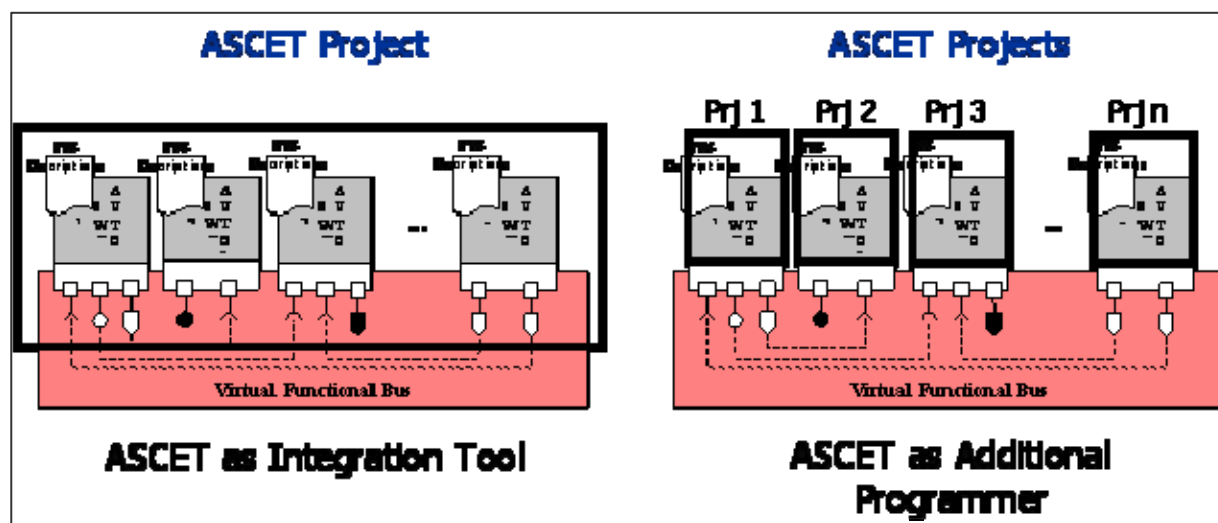


Figure 2: Different ASCET Use-Cases to represent a VFB

As shown in Figure 2, the integration tool and the additional programmer use-case of ASCET can be used to model application software components on the virtual functional bus level.

### 3.1 Integration Tool Use-Case

In the integration tool use-case, loosely spoken, the ASCET project holds the part of the virtual functional bus which is mapped to a dedicated ECU. An ASCET module represents a software-component, ports are represented by messages and the connectors are realized by compatible ASCET messages have identical names and data-types. Using this approach,

- Software components in the project are mapped to one ECU
- Assembly connectors can be generated identifying corresponding messages within an ASCET project (a send-message corresponds to a receive-message when they have the same name). All identified connections will be represented as connectors in a composition. The resulting composition is then embedded in an ECU composition which is the prerequisite to generate RTE-code.
- The runnable-task mapping can be done in the OS-configurator
- 

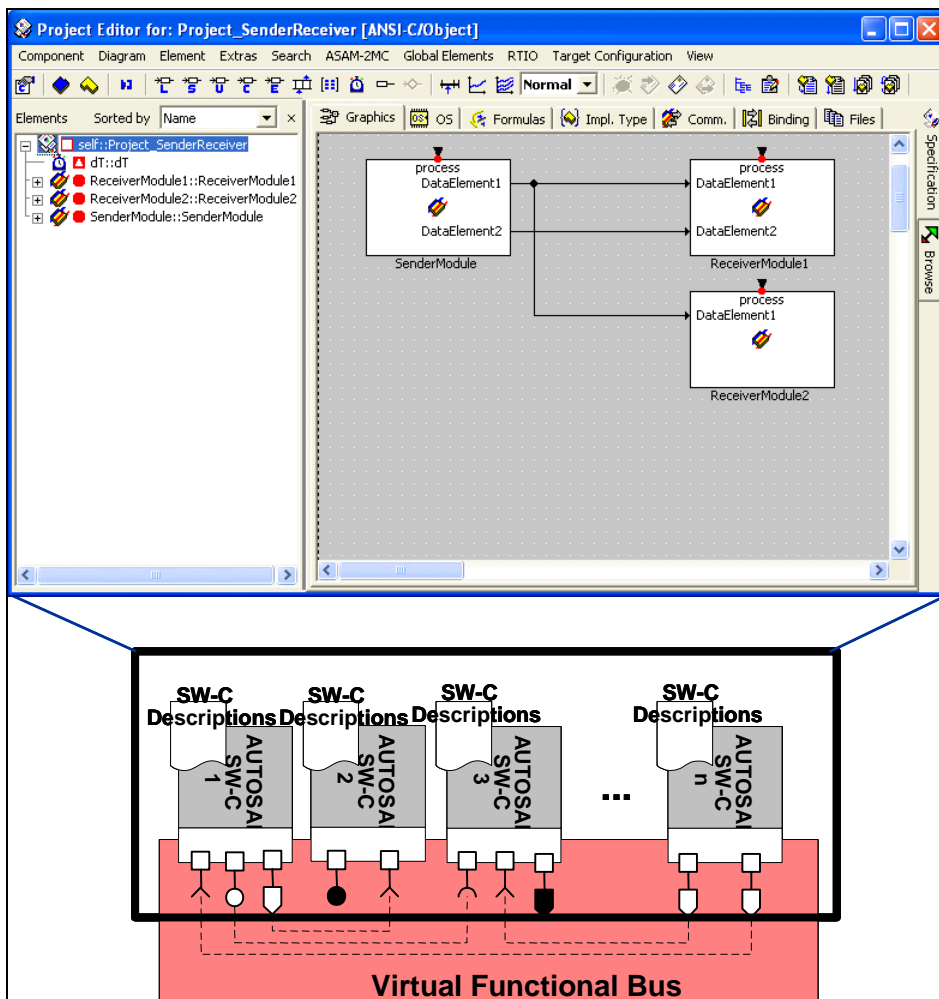


Figure 3: VFB representation in ASCET as full-programmer

This approach limits the AUTOSAR basic idea to one ECU. As written above, this use-case implies, that a module is seen as a representation of a software component and the module's messages represent the software components ports (**Figure 3**). Processes of modules represent runnables and are allocated to task (**Figure 4**). This approach will result in relatively small software-components which are well understood by the function-developer but tend to expose too many details to the system integrator.

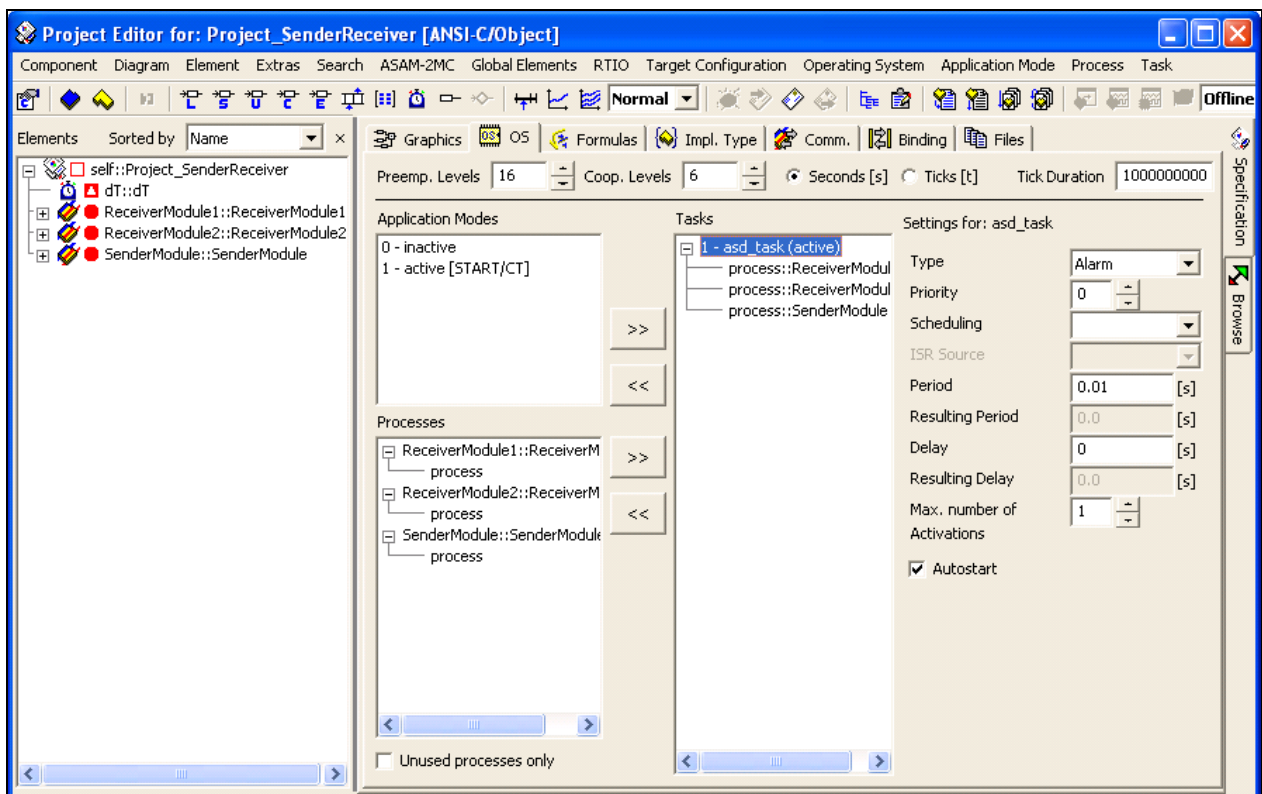


Figure 4: Using the OS-Editor of an ASCET project in the Full-Programmer Use-Case

### 3.2 Additional Programmer Use-Case

An alternative approach is to use ASCET in the additional programmer use-case. As described above, one module is clustered in one project and the project is treated as ONE atomic software component. This approach is conceptually shown in Figure 5. Messages become ports of the atomic software component. The ports are connected explicitly in an AUTOSAR authoring tool. The connections are shown as dashed lines in the upper part of Figure 5. All subsequent steps like system- and ECU-configuration will afterwards be done in dedicated AUTOSAR configuration tools. ECU-configuration includes RTE-generation.

From an ASCET perspective, this means that different ASCET projects can be connected to a virtual functional bus. As a result, two limitations of the integration tool use-case can be overcome:

1. It is possible to design atomic software components independently of ECUs.

2. Connecting of messages with different names is possible  
 But there are still some limitations:

1. The atomic software component can only instantiated once.
2. Limited support of client-server communication

How these limitations will affect the design of AUTOSAR systems has to be identified, e.g. in the migration of legacy projects.

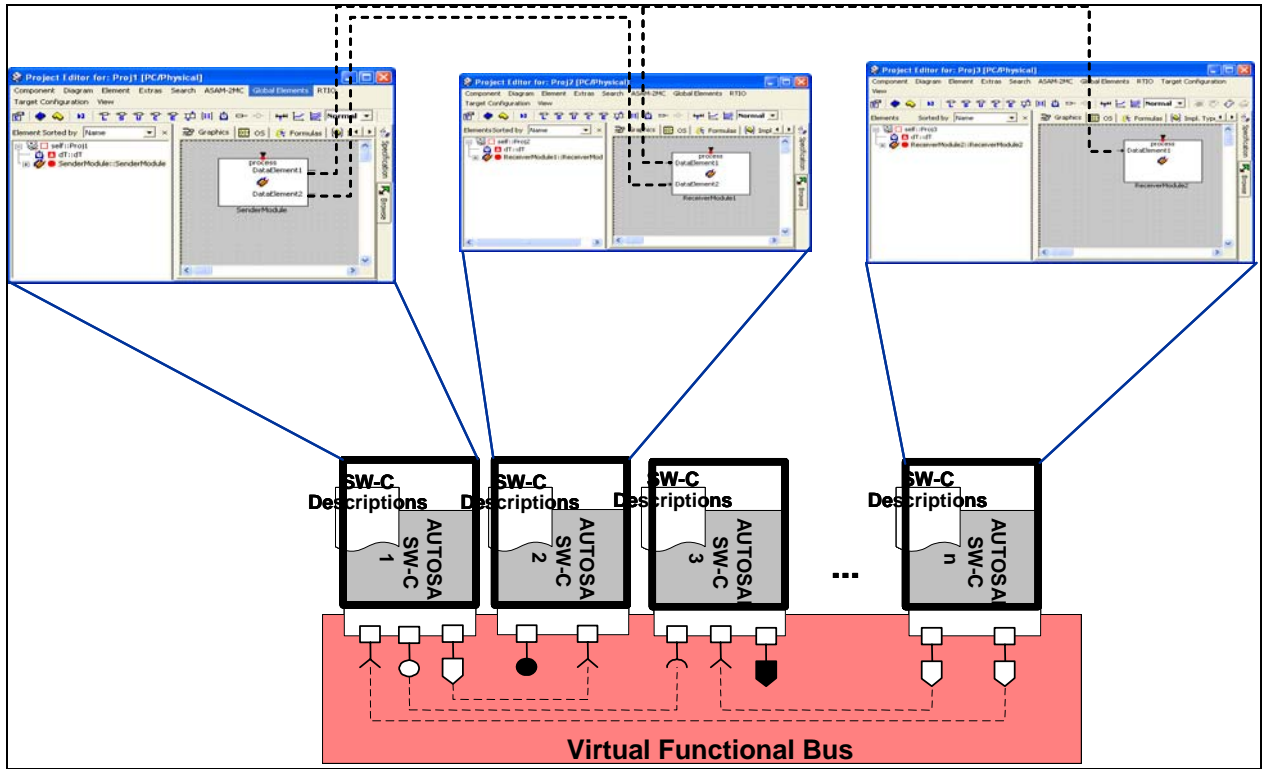


Figure 5: VFB-Representation in ASCET as Additional Programmer

## 4 Migration of Legacy ASCET Projects to AUTOSAR Software Components.

### 4.1 Methodology

To transform existing ASCET projects to AUTOSAR it is proposed to use the additional programmer use-case. The ASCET project is considered as one atomic software component with internal behavior. The result of an AUTOSAR component generation step are the XML-descriptions for the atomic-software-component type, the internal behavior and the implementation. Furthermore, header- and C-Code files, realizing the implementation, will be generated by the ASCET production code-generator ASCET-SE.

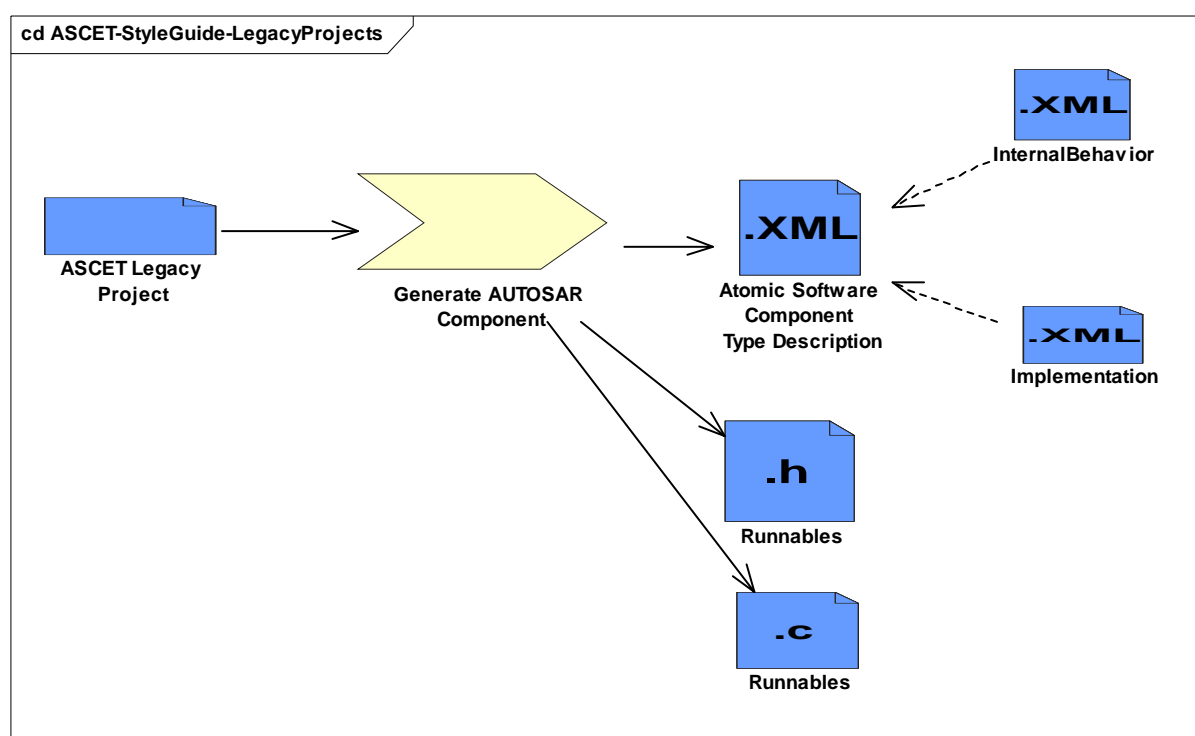


Figure 6: Legacy ASCET Project Methodology

### 4.2 Implementation of the ASCET message concept.

ASCET modeling elements like messages and instances of classes have a scope. This scope is either local, imported or exported. Since messages are used for communication purposes, their scope is either imported or exported w.r.t a certain module or project. All messages are defined ("owned") in a certain module by setting its scope to export. Other modules in a project who want to have access to these message declare a message with the same name but set its scope to import. Since in embedded software development sender-receiver communication is used to realize 1:n communication relationships, it is wise to set the scope of a send-message to export and the scope a receive message to import. This is exactly the default scope of the ASCET messages. In a production-code generator (ASCET-SE) in ANSI-C

option exported messages are defined in module context as variable and hence will later allocate static RAM in an ECU. Wherever necessary, to ensure thread-safeness w.r.t. the task-schedule, a variable duplicate will be allocated either in static RAM or dynamic RAM, i.e. the stack.

### 4.3 Scenario for ASCET Projects and Modules

As written above, an ASCET project can be used to represent an atomic software component in AUTOSAR. Existing ASCET projects can thus easily be represented as single instance atomic software components if all C-Code classes and modules having direct hardware-access are removed.

All modules and instances of classes used in the project will become part of the internal behavior in the atomic software component. However, the signals to be exchanged with other software-components have to be specified. To achieve data-exchange on project level, one can use project global messages. Since in ASCET-SE exported variables are defined on module-level, it is wise to have dedicated module(s) for send- and receive-messages which are intended to be exported beyond project borders<sup>1</sup>. An “input-module” provides all receive-messages with scope export while an “output-module” will provide all send-messages also with scope export. These dedicated modules will be added to the already existing (or ordinary) modules in the project. If in one of the ordinary modules, an exported send-message shall be sent via an AUTOSAR port beyond project borders, this particular send-message has to be set to imported while in the output-module a send-message with the same name but scope exported will be created. The case of an input-message is more straight-forward, because the scope of the receive-message is already set to import, and only an exported receive-message has to be created in the “input-module”, i.e. the receiving module itself need not to be touched. Messages which are not intended to be used beyond project borders, i.e. not being part of an interface, are matched by name within the project and hence are not visible to the outside. Using this approach, matching the messages of the input- and output modules with the messages in the original modules by name creates something similar to delegation connectors.

All receive-messages of the output-module will become PPorts and all send-messages of the input module will become RPort prototypes, both realizing a sender/receiver interface with one data-element.

Figure 7 shows this approach for a simple ABS system. The original modules are

- Brake\_Slip\_Control\_wc,
- Brake\_Deceleration\_Control\_wc, and
- ControlCoordinator\_wc.

The modules with the receive-port-icon, here WheelSpeeds and VehicleSpeed, are the input-modules. The module with the send-port-icon, here ValveRequest, is the output-module. The messages

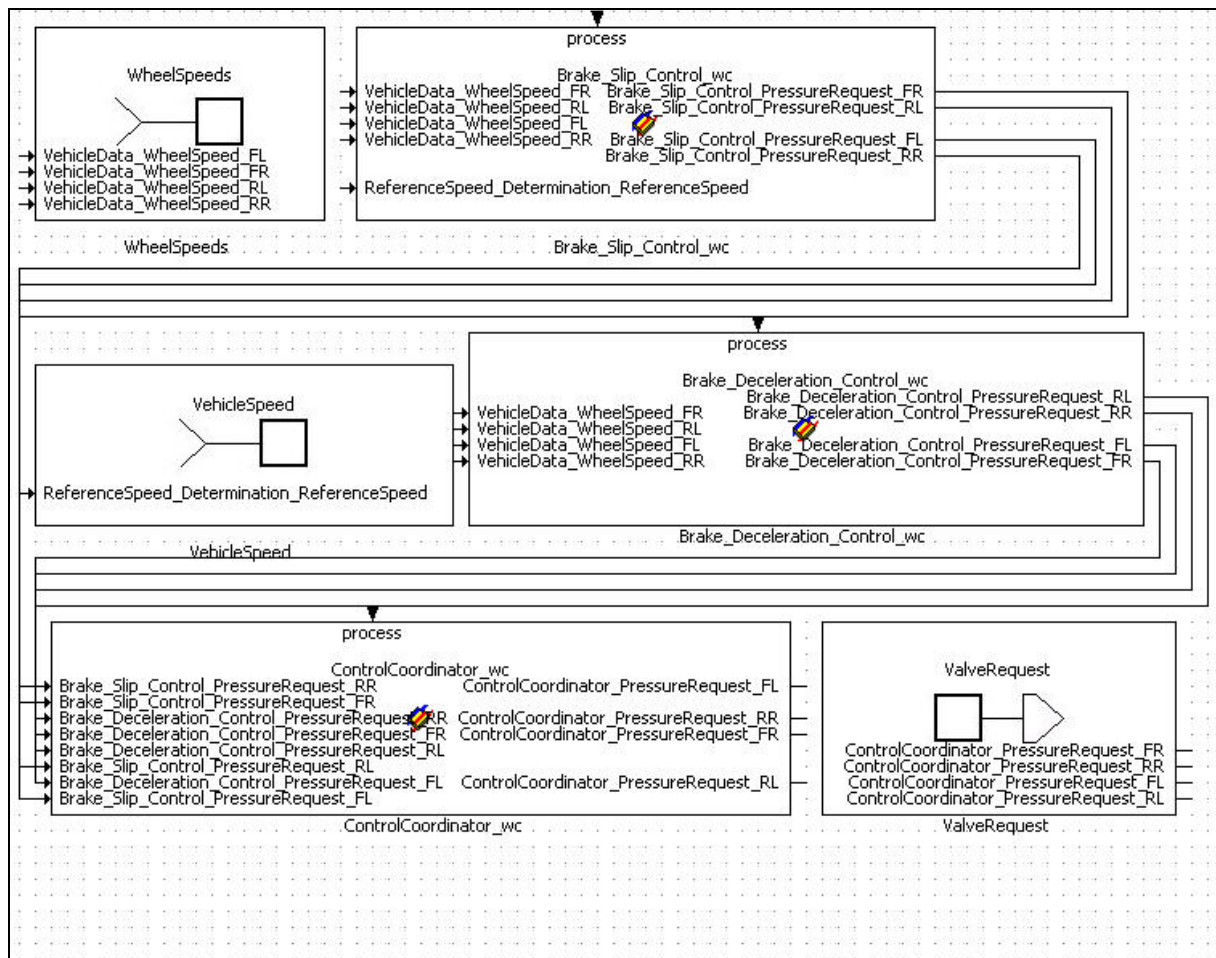
- ControlCoordinator\_PressureRequest\_FL
- ControlCoordinator\_PressureRequest\_FR
- ControlCoordinator\_PressureRequest\_RL
- ControlCoordinator\_PressureRequest\_RR

are set to scope import in the ControlCoordinator\_wc module. The corresponding export messages are now defined in the ValveRequest output-module.

---

<sup>1</sup> To simulate port-grouping, one might use an input-module for every port of an software component using sender-receiver communication.





**Figure 7: ASCET Project with dedicated Input- and Output Modules**

The messages, while being accessed in a process, assume an implicit behavior. The messages of the input-module will obtain “stable” messages by using implicit `data_read_access` and implicit `data_write_access` resulting in a `RTE_IRead` macro for receive-messages and a `RTE_IWrite` macro for send-messages. Using ASCET-SE in ANSI-C modus (without optimizations), a process will always use a local variable to realize implicit behavior. A non-optimized AUTOSAR version of ASCET-SE will use this local variable to put the return value of `RTE_IRead` in, while `RTE_IWrite` will take the value from the local variable which holds the result of all calculations at the end of the process.

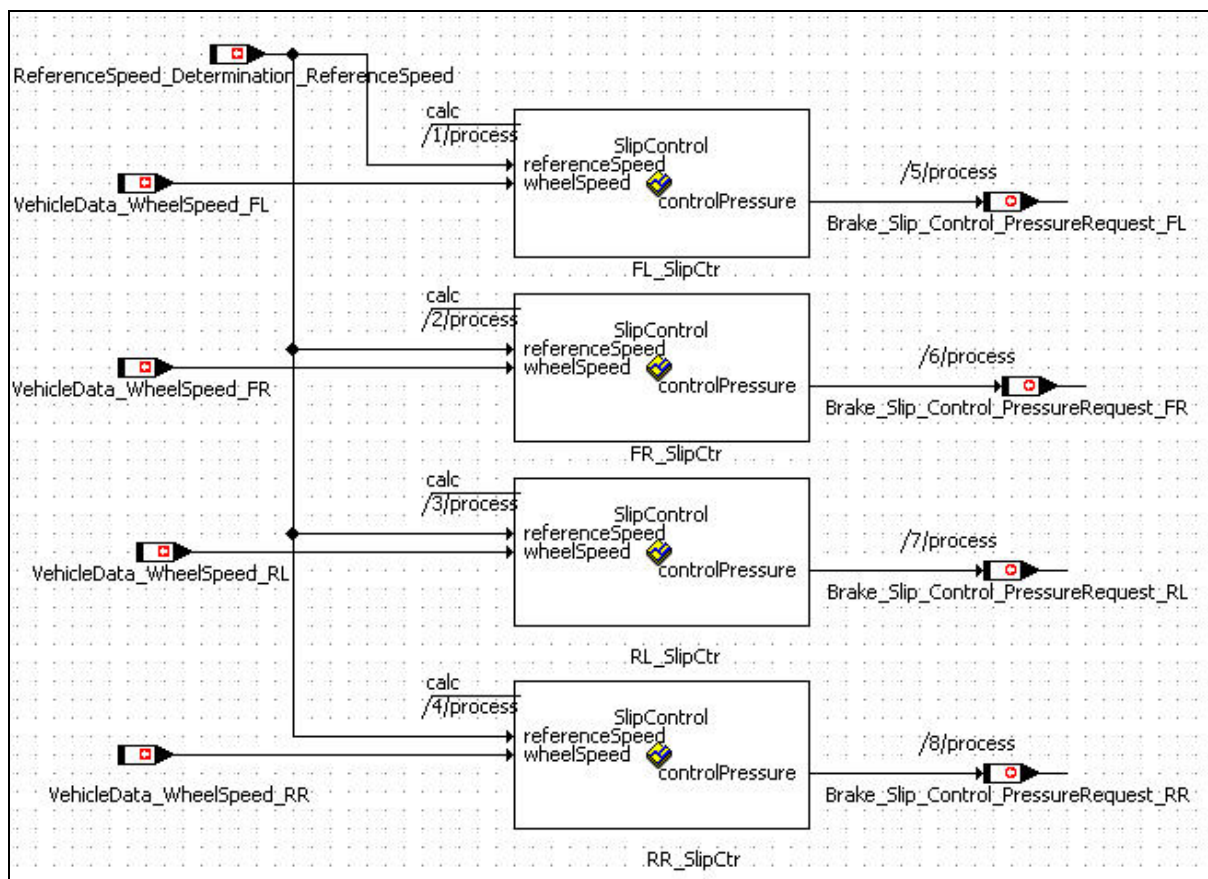
If explicit `data_read_access` and `data_write_access` or `data_send_points` and `data_receive_points` are used, the structure of the ASCET project might be changed.

#### 4.4 Scenario for ASCET Classes

On the one hand, an ASCET class implements a software component where services are provided as methods on one PPort. On the other hand, methods of an ASCET class can be seen as runnable which is started by a client-call from another software



component<sup>2</sup>. If a class shall be migrated to AUTOSAR, it has to be wrapped by a module. The methods of the class can be mapped to processes. For each argument in the method a receive message has to be created while for each return value a send message has to be created. This approach is shown in Figure 8. The module Brake\_Slip\_Control\_wc clusters four instances of the class SlipControl.



**Figure 8: Wrapping four instances of the SlipControl Class in the Brake\_Slip\_Control\_wc Module**

The resulting module has then to be wrapped in a project as written in the previous section. As a result, the wrapped module treats the class as single instance atomic software component with sender/receive interface. Since sender/receiver interfaces in ports are realized by ASCET messages, the same kind of code generation using RTE\_IRead and RTE\_IWrite will be applied as written in section 4.3.

<sup>2</sup> thus reacting to an OperationInvokedEvent

## 5 ASCET & AUTOSAR Overview

### 5.1 Major ASCET Concepts for AUTOSAR





ASCET uses an object based real-time paradigm to construct embedded automotive control software. The main building block are classes for the functional design and modules for the real-time design. To constitute an ECU, modules are aggregated in projects.

A class aggregates methods having arguments and return values. Classes provide an internal state by means of variables and can aggregate instances of other classes. Referencing other classes is possible but might lead to real-time runtime problems and is this not considered in the AUTOSAR modeling guide. Inheritance is not supported. There are two flavors of classes, “simple” classes and finite state machine classes.

For embedded real-time modeling, there are modules which provide messages as means to transfer data between modules. Modules use instances of classes. Modules themselves cannot be instantiated multiple times! Modules provide so-called processes which read and write to messages and call methods of a class.

Projects provide an OS-configuration editor where tasks can be defined. The processes are allocated to tasks. The messages of the modules are connected by name-matching on the ECU-global project level. ASCET models are transferred to executable code by applying code-generation to projects which will generate the code for the modules and classes.

The execution paradigm of ASCET project is: A task calls a process. The process reads from receive-messages and calls the methods of the class-instances by passing message data to arguments. Then, the processes call other messages of the instances receiving return values which will then be written to send-messages. Then the next process in the task is called. Data integrity is ensured by copying mechanisms of ASCET messages. Below, the icons of the major elements are shown.

-  Class Icon
-  Finite-State-Machine Class Icon
-  Module Icon
-  Project Icon

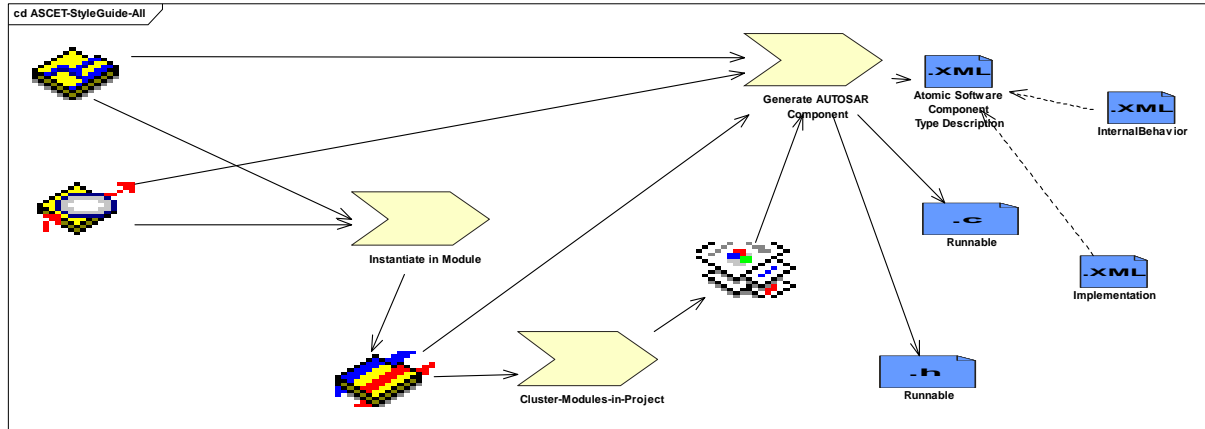
The table below shows how the major design elements map to AUTOSAR concepts:

<b>AUTOSAR Concept</b>	<b>AUTOSAR Description</b>	<b>ASCET Concept</b>
DataType	Type of a DataPrototype	User-Defined Type, Built-In Type
DataElementPrototype	DataPrototype in SenderReceiver Interface	Implicitely realized by message
OperationPrototype	Methods in Client-Server Interface	Method of a class
ArgumentPrototype OUT	DataPrototype in OperationPrototype	Return value of a method
ArgumentPrototype IN	DataPrototype in OperationPrototype	Argument of a method

SenderReceiver Interface	Clusters signals	Implicitly given
ClientServer Interface	Provides or Requires Method calls	Method (of a class)
PPort-Prototype	Access Point in component where data or services are provided	Send-Message at SendReceive Interface, Implicitly given at Client/Server Interface
RPortPrototype	Access Point in component where data or services are required	Receive Message at SenderReceiver Interface, MethodCall in Wrapper-Module of Proxy-Module
Runnable	Behavioral Code	Process in SenderReceiver Interface, Method in Client/Server Interface
InterRunnableVariable	Reentrant buffer for communication between different runnables of one component (proto-) type.	SendReceiveMessage
Atomic Software Component Type	Least software part to be mapped on an ECU	Class (Multiple Instance), Module, Project (Single Instance)
(Atomic) Component Prototype	Tentative Instance of an atomic software component.	Wrapped Instance of Class in Module.

**Table 2: Matching of ASCET and AUTOSAR Concepts**

Whereas ASCET employs an object based real-time modeling paradigm, AUTOSAR employs a different object based real-time paradigm. The overlapping concepts can be used to generate AUTOSAR software component types, the internal behavior and the implementation. While in AUTOSAR the association is done quite late in the methodology, ASCET will generate all three descriptions with correct links. Depending on the chosen granularity of an ASCET model, atomic software components including referenced internal behavior and implementation can be sensibly generated from classes, modules and projects as shown in Figure 9. However, all of the elements realize different subsets of the software component template. A combination the elements realize a bigger subset. This combination is called clustering and explained below.



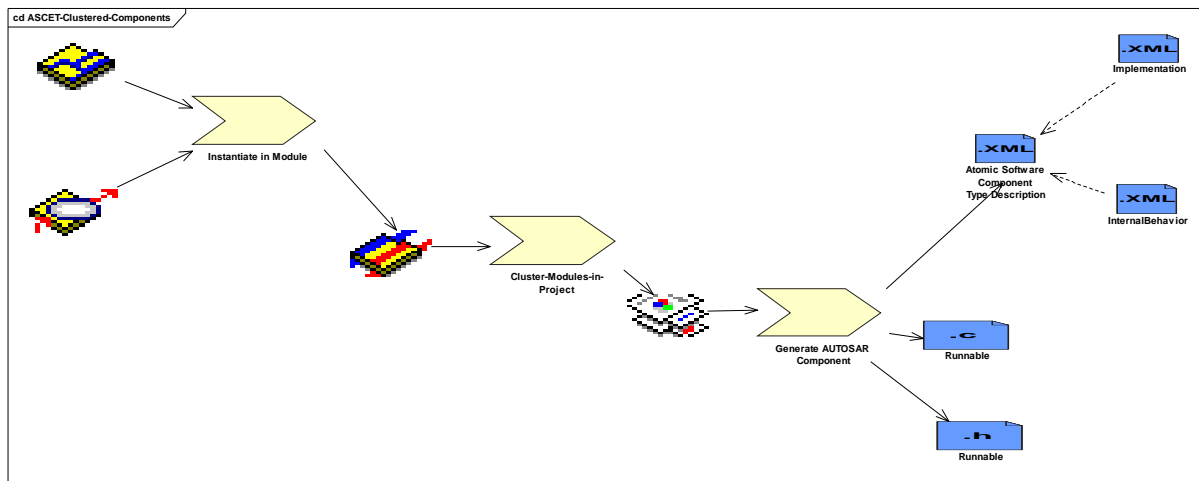
**Figure 9: Generating Component Types, Internal Behavior, and Implementation Files**

## 5.2 Clustering

The ASCET concepts Project, Module, and Class can be used in several ways to compose executable systems. As written above, after more or less modifications, all projects can be transferred to atomic software components. However, the AUTOSAR concepts like interface and component types, data-, port- and component prototypes as well as compositions and runnables can be used directly in ASCET. Depending on a clustering step, atomic software components can be tailored to match with the interfaces given by the OEM. This tailoring (or clustering) has the following advantages over compositions:

- All multiple instantiations are resolved in the cluster, thus the RTE can be configured much leaner.
- The number of component- and connector prototypes is reduced significantly for the system-integrator, thus contributing to the complexity management.
- Clear distinction between function developer and system integrator. Especially, the type/prototype concept which has implementation advantages on an ECU can be hidden from the system-integrator.
- For the function-developer, prototype naming is likely to be more straightforward, i.e. prototype names like `SeatheatControlObject` can be directly be named to `SeatheatControlDriver` and `SeatheatControlPassenger` within a cluster.

However, this approach requires a strict use of the ASCET class concept. Projects, modules and messages become a mere composition feature. It leverages the advantages of both the additional programmer- and the integration tool use-case of ASCET.



**Figure 10: Clustering Approach to create efficient atomic software components**

Figure 10 shows the clustering approach in SPEM notation. The ASCET-model workproducts are represented by using the ASCET icons.

## 6 Creating ASCET Projects in Dedicated AUTOSAR Style

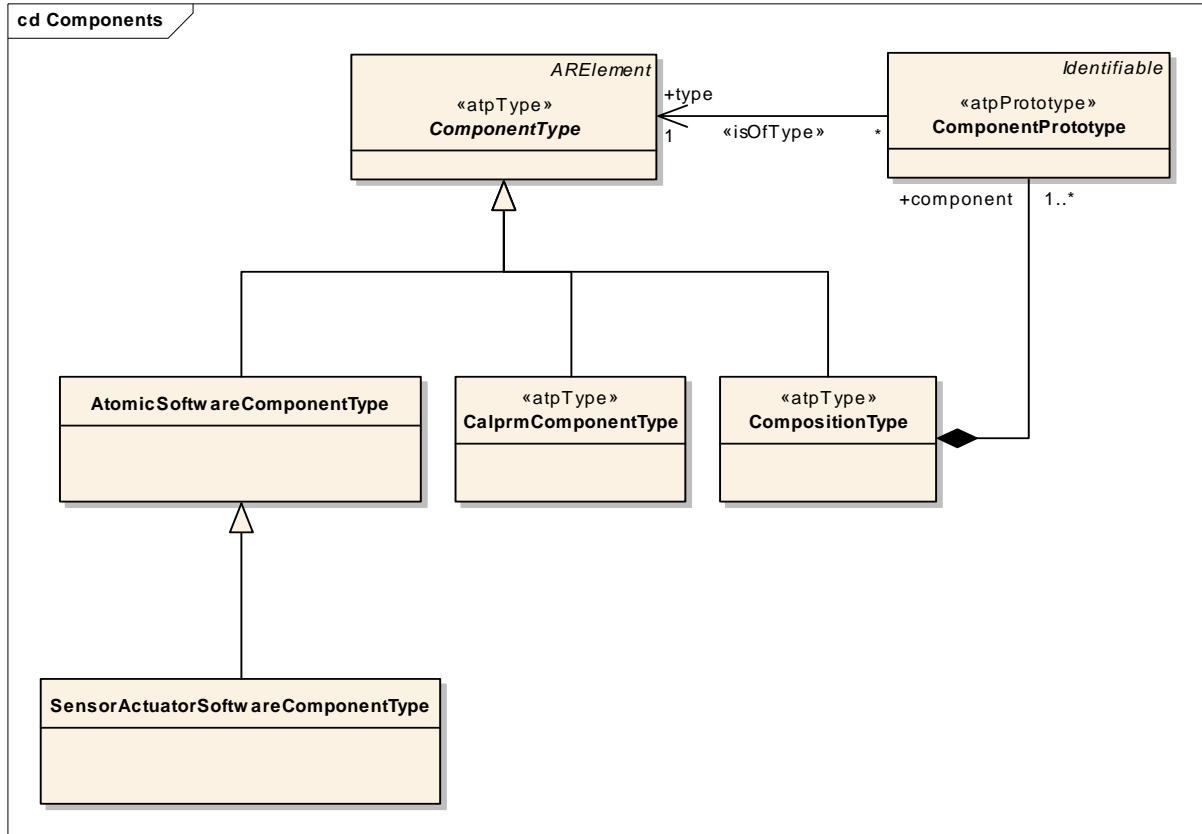
The software-component-template version 2.01 is used as reference.

### 6.1 Example System

A simple seat heating function is used to show the concepts. The seat heating function is symmetrical for driver and passenger. A switch with the stages Off, Stage1, and Stage2 determines the user-request. A controller will arbitrate the user-request with the actual clamp status, the vehicle-voltage and the availability of the seat heating equipment. The seat heating equipment is driven by a PWM-Signal of the IoHwAbstraction-Layer and evaluates the diagnostics status. The actual stage of the seat heating function is indicated by two individual LED.

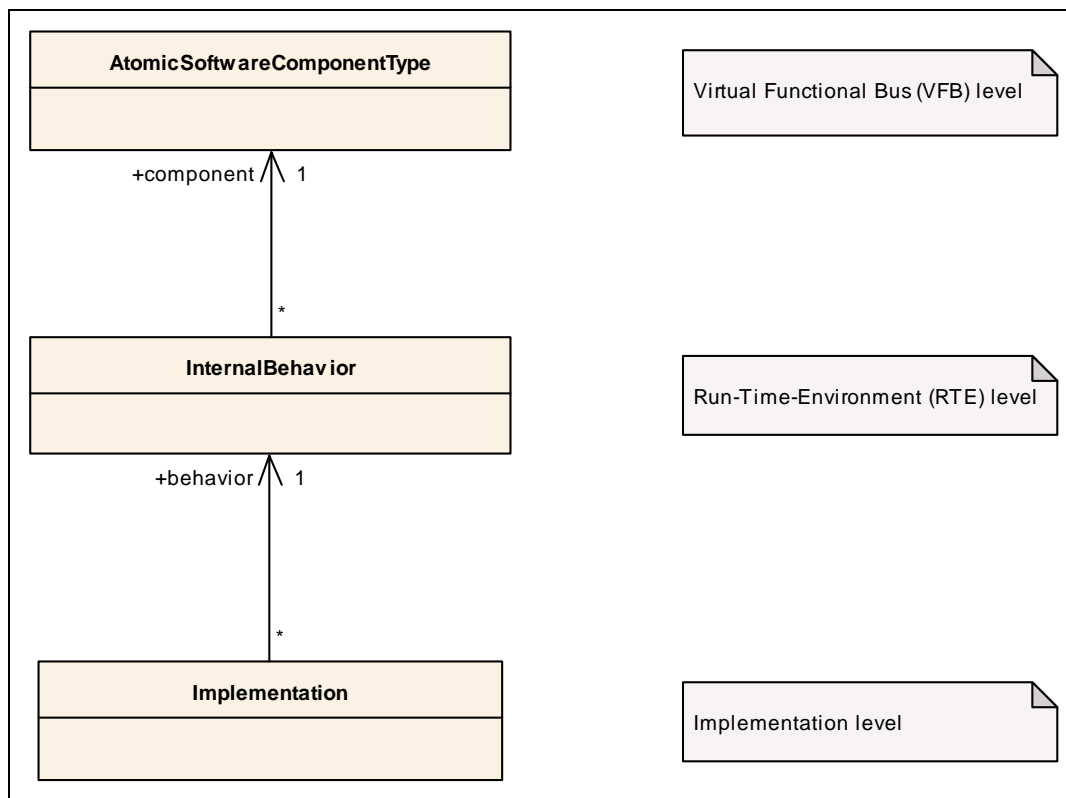
### 6.2 Atomic Software Component Types

In AUTOSAR, application software is organized in independent units which hide their internal behavior and communicate with other software components through ports realizing interfaces. Loosely spoken these entities are the software components. A closer look on the software component template, reveals that there are no software components but `ComponentType` entities and `ComponentPrototype` entities. `ComponentPrototype` entities describe the use of `ComponentType` entities in certain roles and will become instances on the running ECU. But this type/prototype concept is also used to construct an `AtomicSoftwareComponentType` entity, which is a special version of a `ComponentType`. An `AtomicSoftwareComponentType` is made up of a whole number of nested prototypes. Nesting means that a `DataElementPrototype` in a `SenderReceiver Interface` has a `DataType` and can be used in more than one `Port-Prototype` of an `AtomicSoftwareComponentType`. On virtual functional bus level but also on one ECU there might be several `ComponentPrototype` entities of the same `AtomicSoftwareComponentType`. Another specialization of the `ComponentType` is the `CompositionType` which aggregates other `ComponentPrototype` entities which can be of all `ComponentType` entities, e.g. `AtomicSoftwareComponentType` or `CompositionType`.



**Figure 11: ComponentType Entities**

The AUTOSAR VFB-Level is made up of a composition of ComponentPrototype entities. The InternalBehavior can be assigned to AtomicSoftwareComponentType entities. It is composed of several RunnableEntities. The AUTOSAR RTE-Level is made up of set of runnables per ECU which are associated to AtomicSoftwareComponentType entities mapped to that ECU. Last but not least, the runnables are implemented either by source- or object-code.



**Figure 12: Referencing Relationship between Implmentation, Internal Behavior and an Atomic Software Component Type**

This document focuses on the development and generation of AtomicSoftwareComponentType entities on virtual functional bus (VFB), run-time environment (RTE), and implementation level. ASCET can thus be used as authoring- and behavioral modeling tool for a subset of AtomicSoftwareComponentType entities.

### 6.2.1 DataTypes

ASCET provides built-in and user-defined data-types. Built-in data-types are

- cont (Continuous) ,
- sint (Signed Integer),
- uint (Unsigned Integer), and
- log (Logical).

Another Built-in data-type in ASCET are arrays and matrices, but these types cannot be used to type messages. User-defined types are

- enumerations

and classes. Of the user-defined types, only enumerations can be used to type messages. User-defined types have an explicit name and are maintained in the ASCET repository.



Enumeration types are handled differently from the above ASCET approach but also differently from AUTOSAR. The enumerators are assigned to unsigned integer values in a dedicated editor. Below there are examples of user-defined data-types. They are exclusively enumerations. The built-in data-types are explained in the context of atomic software components.

Value	Label
0	T_15
1	T_15R
2	T_15X
3	T_15C
4	IgnitionOn
5	IgnitionOff

**Figure 13: Clamp Data-Type**

Value	Label
0	NoValidInformationAvailable
1	ShortToPowerSupply
2	ShortToGround
3	OpenLoad
4	OverTemperature
5	DiagnosticOK

**Figure 14: DiagnosticsClass**

Value	Label
0	stageoff
1	stage1
2	stage2

**Figure 15: SeatHeatStages**

Enumeration types are handled differently from the above ASCET approach but also differently from AUTOSAR. The enumerators are assigned to unsigned integer values in a dedicated editor. Below there are examples of user-defined data-types. They are exclusively enumerations. The built-in data-types are explained in the context of atomic software components.

Value	Label
0	T_15
1	T_15R
2	T_15X
3	T_15C
4	IgnitionOn
5	IgnitionOff

**Figure 16: Clamp Data-Type**

Value	Label
0	NoValidInformationAvailable
1	ShortToPowerSupply
2	ShortToGround
3	OpenLoad
4	OverTemperature
5	DiagnosticOK

**Figure 17: DiagnosticsClass**

Value	Label
0	stageoff
1	stage1
2	stage2

**Figure 18: SeatHeatStages**

### 6.2.2 PrimitiveTypeWithSemantics

PrimitiveTypeWithSemantics entities: For fixpoint-arithmetics, AUTOSAR allows to assign CompuMethod entities to integer values. CompuMethod entities determine how an integer value has to be interpreted. In case of internal-to-phys an integer value determines a number of CompuScale entities with an upper and a lower limit as well as a scale. The inverse way, i.e. specifying the physical values and then derive the internal values, can also be used in AUTOSAR. Simple CompuMethod entities can be defined in ASCET by using implementations. Applying an implementation formula to the built-in datatype cont results in a different interface type. Thus, the interface of

an implemented atomic software component changes too. Take for example the vehicle voltage. If a message is of type Cont, an SenderReceiverInterface of type Put-Cont will be used. Now, an implementation formula with name VehicleVoltage8Bit will be used which maps the range of the Cont data-type to an 8-bit integer ranging from 0 to 31,875 Volt in 256 steps of 0.125 Volt.

Meta-type	Range	Base Type
CHAR-TYPE	Encoding 'UTF-16'	uint16
STRING-TYPE	Declaration, n is defined maximum length including zero terminator	uint8[n]
STRING-TYPE	Function parameter	uint8*
INTEGER-TYPE	[-128,127]	sint8
INTEGER-TYPE	[-32768,32767]	sint16
INTEGER-TYPE	[-2147483648,2147483647]	sint32
INTEGER-TYPE	[0,255]	uint8
INTEGER-TYPE	[0,65535]	uint16
INTEGER-TYPE	[0,4294967295]	uint32
OPAQUE-TYPE	Bit length 1..8	uint8
OPAQUE-TYPE	Bit length 9..16	uint16
OPAQUE-TYPE	Bit length 17..32	uint32
REAL-TYPE	Encoding single	float32
REAL-TYPE	Encoding double	float64
BOOLEAN-TYPE	N/A	boolean

**Table 3: Mapping of Software-Component-Template Types to RTE-Base-Types**

However, this physical range is not considered by the RTE. In the vehicle voltage example, the model-type cont will result in an INTEGER-Type with lower limit 0 and upper limit 255. Using Table 3, the resulting RTE type will be uint8. To ensure that only semantically compatible interfaces are connected, the implementation of a built-in ASCET datatype will be reflected in the DataType definition of a SenderReceiver-Interface.

### 6.2.3 Summary of “DataType” entities

To summarize, only the scalar built-in datatypes like Cont, Udisc, Sdisc, Log and user-defined enumeration types can be transferred via messages and can thus be used as DataPrototype entities. If these messages are implemented using an implementation formula, the implementation name will be reflected in the DataPrototype entity. Complex built-in ASCET data-types like arrays and matrices and user-defined types like classes cannot be transmitted over messages and thus not be used to type DateElementPrototype entities.

### 6.2.4 Interfaces

AUTOSAR distinguishes between sender-receiver and client-server communication. To achieve this, ComponentType entities employ PortPrototype entities which are typed by an InterfaceType. PortPrototype entities are either provided (PPortPrototype) or required (RPortPrototype).

- **SenderReceiverInterface:** It is derived from PortInterface and has several DataElementPrototype entities. DataElementPrototype entities are DataType entities which can either be of primitive type or complex type. A SoftwareComponentType with an RPortPrototype being typed by a SenderReceiverInterface can be connected as a ComponentPrototype to an PPortPrototype (of an other ComponentPrototype) being typed by a compatible SenderReceiverInterface. A SenderReceiverInterface is compatible if the interface of a the RPort has a smaller or equal number of DataElementPrototypes than the interface in the PPort. However, this rules holds only for assembly connectors and not for delegation connectors.
- **ClientServerInterface:** It is derived from PortInterface and aggregates a number of OperationPrototype entities. OperationPrototype entities aggregate several ArgumentPrototype entities being typed by DataType entities. ArgumentPrototype entities have either an IN, an INOUT or an OUT direction.

The software component template defines several DataType entities. These DataType entities type DataProtoType entities which are used for communication between software components and not for modeling the behavior of control algorithms. Of particular interest in ASCET are the IntegerType, the BooleanType, the RealType, and the OpaqueType. CompositType entities like the ArrayType and the RecordType have no equivalence in ASCET for communication, but can be used for behavioral modeling of RunnableEntities internally. Last but not least, AUTOSAR takes over from MSR-SW the concept the PrimitiveTypeWithSemantic. This concept plays an important role in behavioral modeling with ASCET.

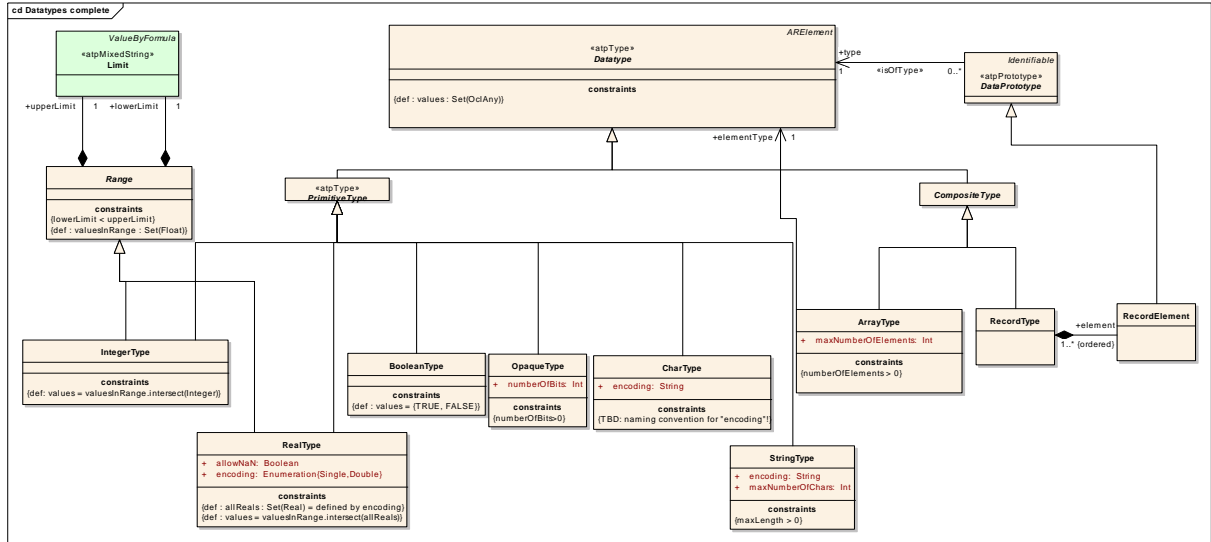


Figure 19: Overview of AUTOSAR Datatypes for application software components

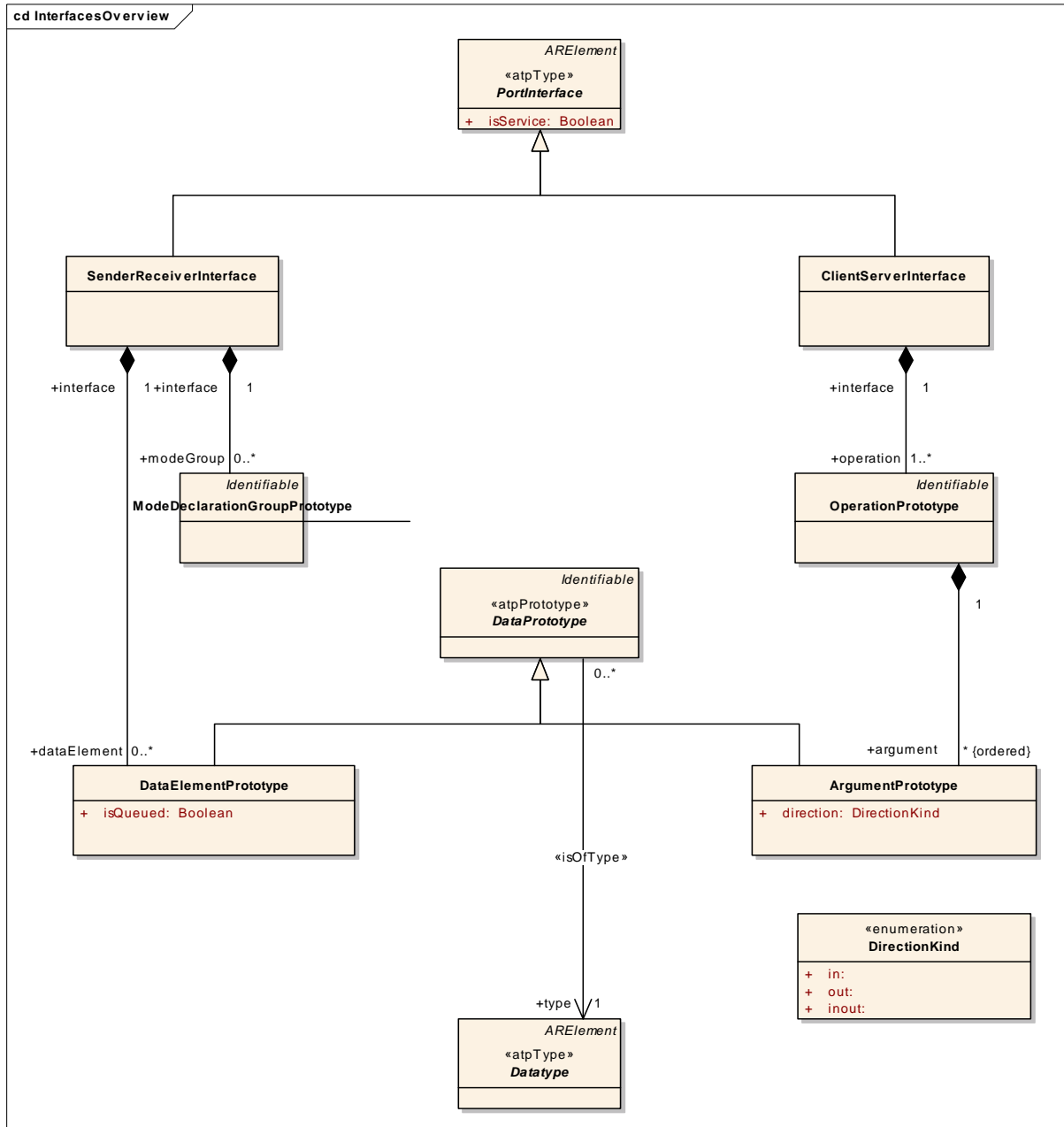


Figure 20: PortInterface entities

The AUTOSAR name of a sender/receiver interface given by concatenating the word *Put* with the name of the *DataType* entity of the *DataElementPrototype* entity. For the built-in datatypes, there are the following *SenderReceiverInterface* entity-names defined:

- PutCont
- PutUdisc
- PutSdisc
- PutLog

In the case of a user-defined type, one might obtain

- PutClamp
- PutSeatHeatStages
- PutOnOff
- PutDefectNotDefect

In case of an implemented built-in datatype, the *SenderReceiverInterface* entity-name is given by the concatenation of *Put* with the implementationtype followed by the implementation formula name. In the above example with the vehicle voltage, one will obtain:

- Putuint8VehicleVoltage8bit

This naming convention ensures that the *CompuMethod* is reflected in the interface name. Since on virtual functional bus level, only *PortPrototype* entities with the same *Interface* can be connected, this convention ensures semantic compatibility.

## 6.2.5 PortPrototypes

### 6.2.5.1 SenderReceiverInterface

In a sender-receiver Communication, ASCET does not distinguish between *DataElementPrototypes* and *PortPrototypes* and hence not between *Interfacetype* and *DataType*. In ASCET, a send-message represents a *PPortPrototype* with a generic *InterfaceType* having exactly one *DataElementPrototype* of AUTOSAR type *IntegerType* (which can be *sint*, *unit*, or (user-defined) enumeration in ASCET), *BooleanType* (log in ASCET) or of *RealType* with enumeration *double* (Cont in ASCET). Send-messages in ASCET represent *PPortPrototype* entities with *SenderReceiverInterface* entities and Receive-messages *RPortPrototype* entities with *SenderReceiverInterface* entities respectively! If a message is typed by a built-in data-type, this means that the interface-name is given by concatenating *Put* with the type-name of the user-defined data-type of the *DataElementPrototype* name as seen in the AUTOSAR repository. The *PortPrototype* name of an *AtomicSoftwareComponentType* is thus given by the ASCET message name.

### 6.2.5.2 ClientServerInterface

A *ClientServerInterface* entity is realized in ASCET as an aggregation of methods. As special case of an *OperationPrototype* entity in AUTOSAR, an ASCET method will have just one *OUT-ArgumentPrototype* entity per method. ASCET methods may have an arbitrary number of arguments, which represent *IN-ArgumentPrototype* entities in *OperationPrototype* entities of *ClientServerInterface* entities. Arguments can be of

user-defined type enumeration and built-in data-types. INOUT-ArgumentPrototype entities are not supported by ASCET methods.

A (non finite state machine) class represents a (single) PPortPrototype entity realizing a ClientServerInterface entity in ASCET. The PPortPrototype entity name is given implicitly by concatenating *PPort* to the name of the ASCET class.

### 6.2.6 AtomicSoftwareComponentType Entities

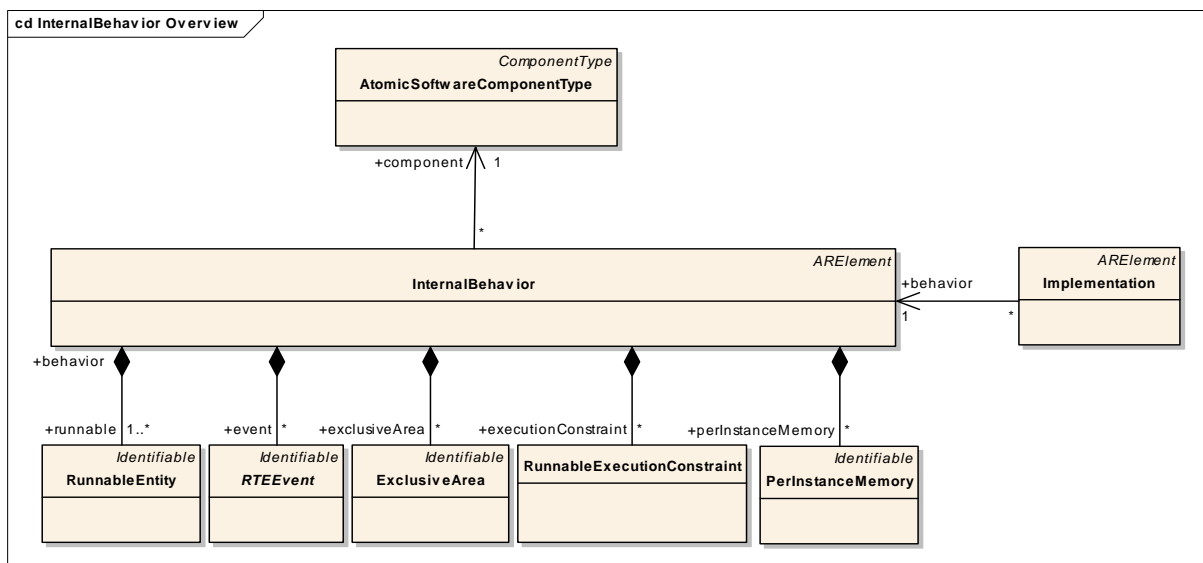
In AUTOSAR an AtomicSoftwareComponentType has one to many PortPrototype entities in P- and R-Direction which are typed by either SenderReceiverInterface- or ClientServerInterface-entities. An arbitrary number of ComponentPrototype entities can be derived from an AtomicSoftwareComponentType.

ASCET supports sensibly three flavors of an AtomicSoftwareComponentType:

1. SingleInstanceType entities with an arbitrary number of (R- and P-) PortPrototype entities realizing SenderReceiverInterfaces with one DataElementPrototype
2. MultipleInstanceType entities with an arbitrary number of (R- and P-) PortPrototype entities realizing SenderReceiverInterfaces with one DataElementPrototype.
3. MultipleInstanceType entities with one PPortPrototype realizing a Client-ServerInterface with OperationPrototype entities having one OUT Argument-Prototype.

### 6.2.7 Internal Behavior

While AUTOSAR distinguishes between an AtomicSoftwareComponentType and the InternalBehavior, ASCET has an integrated approach describing both the AtomicSoftwareComponentType and the InternalBehavior by realizing the instance-ref of DataPrototype access of the runnables. This means that the XML file entries corresponding to the meta-model elements as shown in Figure 21 and Figure 22 are generated from the ASCET project-, class-, and module-description.



**Figure 21: InternalBehavior and RunnableEntities**



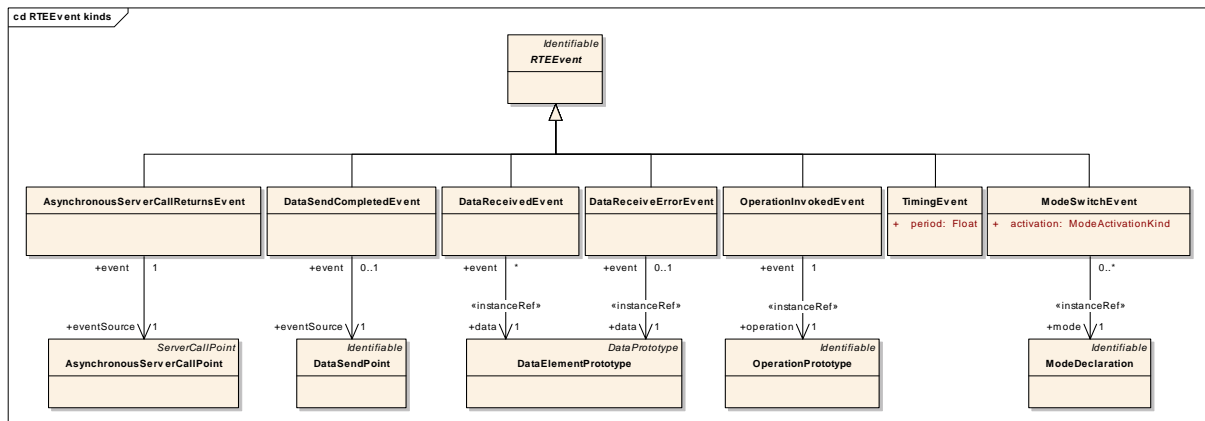


Figure 22: RTE-Events of RunnableEntities

**6.2.7.1 SingleInstanceTypes with an arbitrary number of (R- and P-) PortPrototypes realizing SenderReceiverInterfaces with one DataElementPrototype**

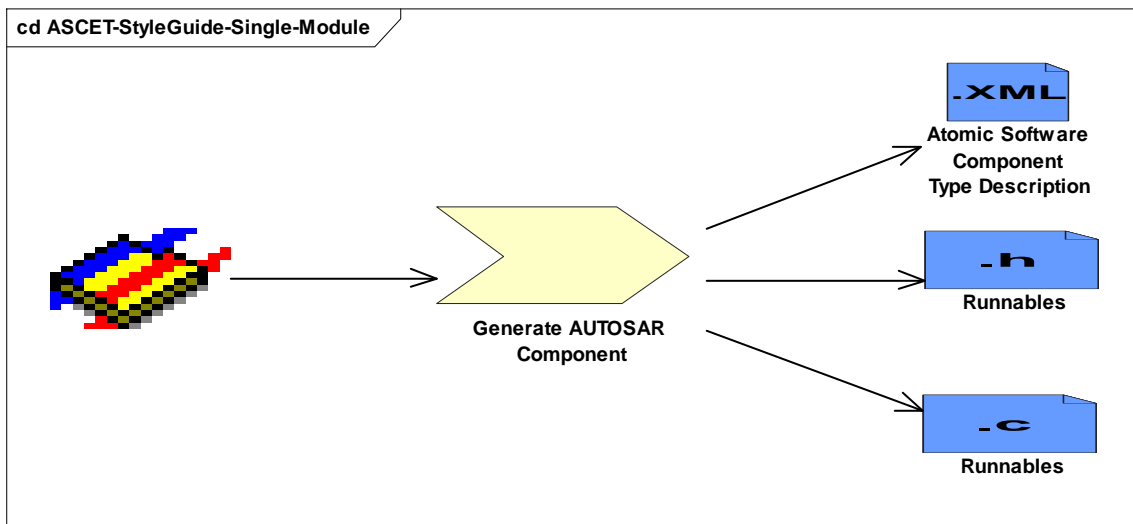


Figure 23: ASCET Module as Single Instance AtomicSoftwareComponentType

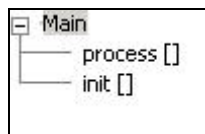
ASCET Modules provide processes, which will be translated to void-void-C-functions, and messages. Messages are interrupt-save inter-process-communication means. In case of a tentative interruption of process by a process of a task with higher priority, the messages will be translated in a double-buffered variables. Messages can only carry scalar data-types (Built-In and User-Defined, see above). Modules can only instantiated once.

Messages of a module correspond to the PortPrototype entities of an AtomicSoftwareComponent. SendMessages are PPortPrototype entities realizing a “SenderReceiverInterface”, ReceiveMessages are RPortPrototype entities realizing a SenderReceiverInterface. For the SenderReceiverInterface entities the constraints defined in section 6.2.4 apply.



**Figure 24: Interface of a Single Instance Component with S/R-Interfaces**

The InternalBehavior entity will have the same name as the module name. Processes are realizing RunnableEntity elements with implicit data access semantics. Every time a process reads from a receive-message, it realizes a DataReadAccess. The same holds for a DataWriteAccess which is established every time a process writes to a send-message. While the implicit data access is finally implemented in the RTE, the ASCET code-generation ensures that the messages are correctly mapped to RTE\_IRead and RTE\_IWrite calls at the appropriate points in the generated code. However, ASCET processes do not determine their RTEEvent. This is part of the ECU configuration in ASCET and can be used only in the full-programmer use-case. Therefore, adding an RTEEvent to a RunnableEntity has to be done in a dedicated separate authoring tool or ECU-configuration tool with this ability. However, only those RTE events can be added which sensibly support the implicit data-access semantics. Therefore, it is recommended to use the TimingEvent.



**Figure 25: Internal Behavior of a Single Instance Component**

Using the methodology as depicted in Figure 23, the send-messages will be translated to RTE\_IWrite calls while the receive messages will be translated to RTE\_IRead calls. The original message properties w.r.t. interrupt safeness will be used in the clustering use-case.

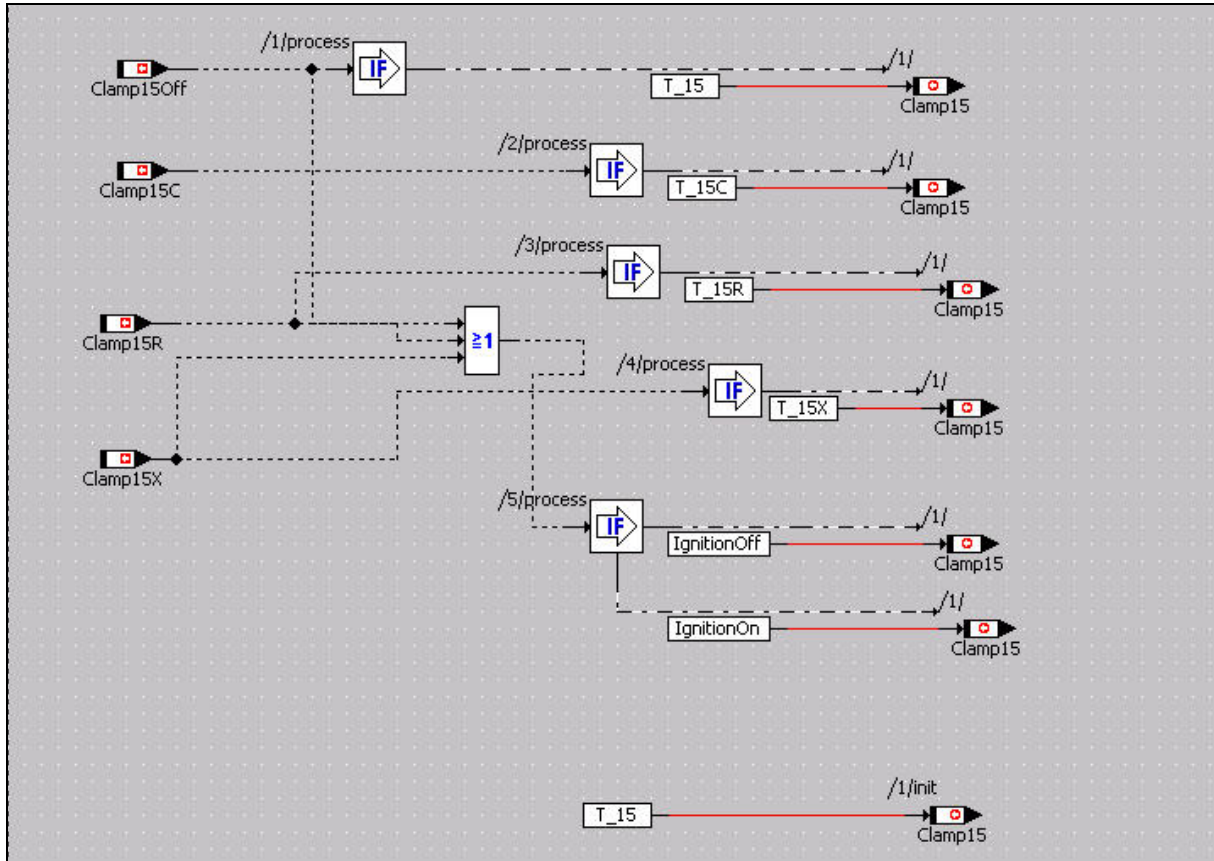
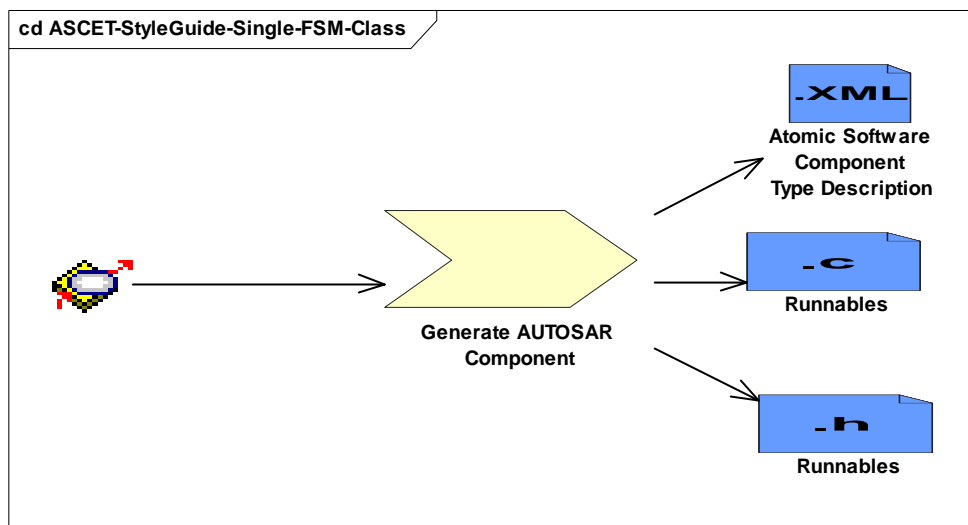


Figure 26: Graphical View of Internal Behavior

**6.2.7.2 MultipleInstanceTypes with an arbitrary number of (R- and P-) PortPrototypes realizing SenderReceiverInterfaces with one DataElementPrototype**

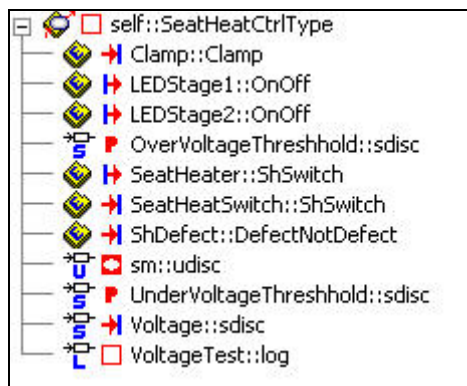


A finite-state-machine-class has dedicated input- and output-ports. These ports have the same features and restriction w.r.t. types like messages. A finite-state-machine class is based on the class concept in ASCET can be instantiated several times.

The internal behavior is given by so-called trigger functions, or triggers for short, which can be best compared to AUTOSAR runnables because they are realized by void functions with instance handle. Conceptually, these trigger functions evaluate the transition conditions of an automaton depending on the current state, perform transitions if applicable (including actions), and execute exit-, state-, and entry-statement. Since a trigger function is intended to execute a finite-state-machine, there should only be one trigger-function per finite-state-machine class.

To support transitions, conditions and actions can be modeled as internal methods of the finite-state-machine<sup>3</sup> and are not visible from the outside.

In the seat heating functionality example a finite-state-machine takes over the central control of the heating equipment of one seat. The input-ports are Clamp (of user-defined enumeration Clamp), SeatHeatSwitch (of user-defined enumeration ShSwitch), Voltage (of built-in type Sdisc), and ShDefect (of user-defined enumeration DefectNotDefect). Output-ports are LEDStage1 and LEDStage2, both of user-defined enumeration OnOff (and thus showing the ASCET realization of the PortPrototype concept of AUTOSAR), and SeatHeater, also being of user-defined enumeration ShSwitch (and showing the DataType employment concept in P- and RPortPrototypes of an AtomicSoftwareComponentType). The finite-state-machine ports are shown in Figure 27.



**Figure 27: Interfaces of the AtomicSoftwareComponentType SeatHeatCtrlType**

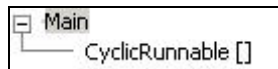
<sup>3</sup> From the ASCET point of view, a trigger function in a finite-state-machine class can be seen as external method.

Trigger functions can have arguments. In this case, an operation-invoked event will be generated.

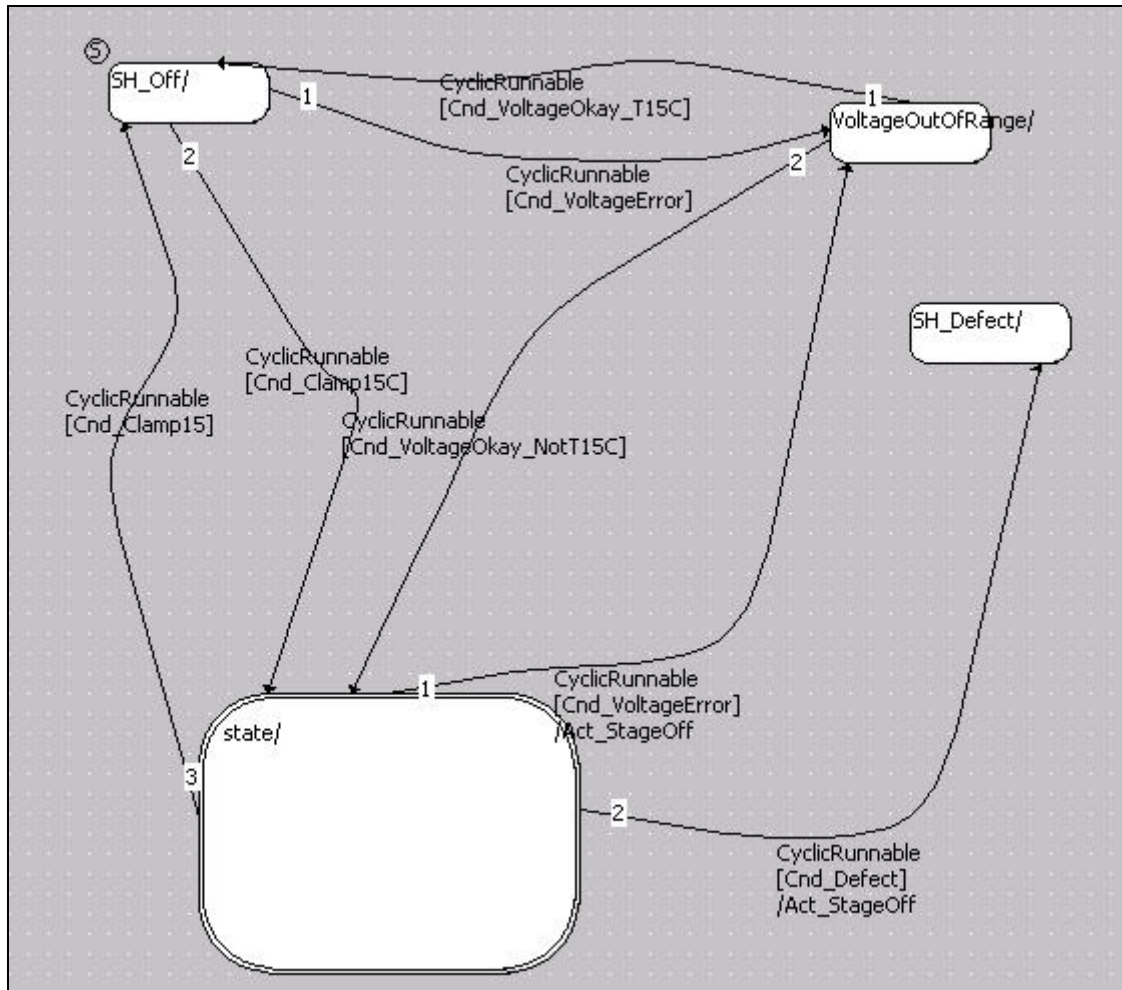
The InternalBehavior of the seat-heating system is modeled using one trigger function of name CyclicRunnable and is shown in Figure 28. In this example, the DataReadAccess (as well as DataWriteAccess) is implemented in the local methods of the finite-state-machine class which are evaluated in the trigger function realizing the RunnableEntity.

However, the access to the finite-state-machine ports can also directly done in the actions and conditions of the transitions of the automaton.

As with processes, no RTEEvent will be attached to the trigger function. This has to be done in an AUTOSAR authoring tool where the TimingEvent is the natural RTEEvent for trigger functions without arguments.



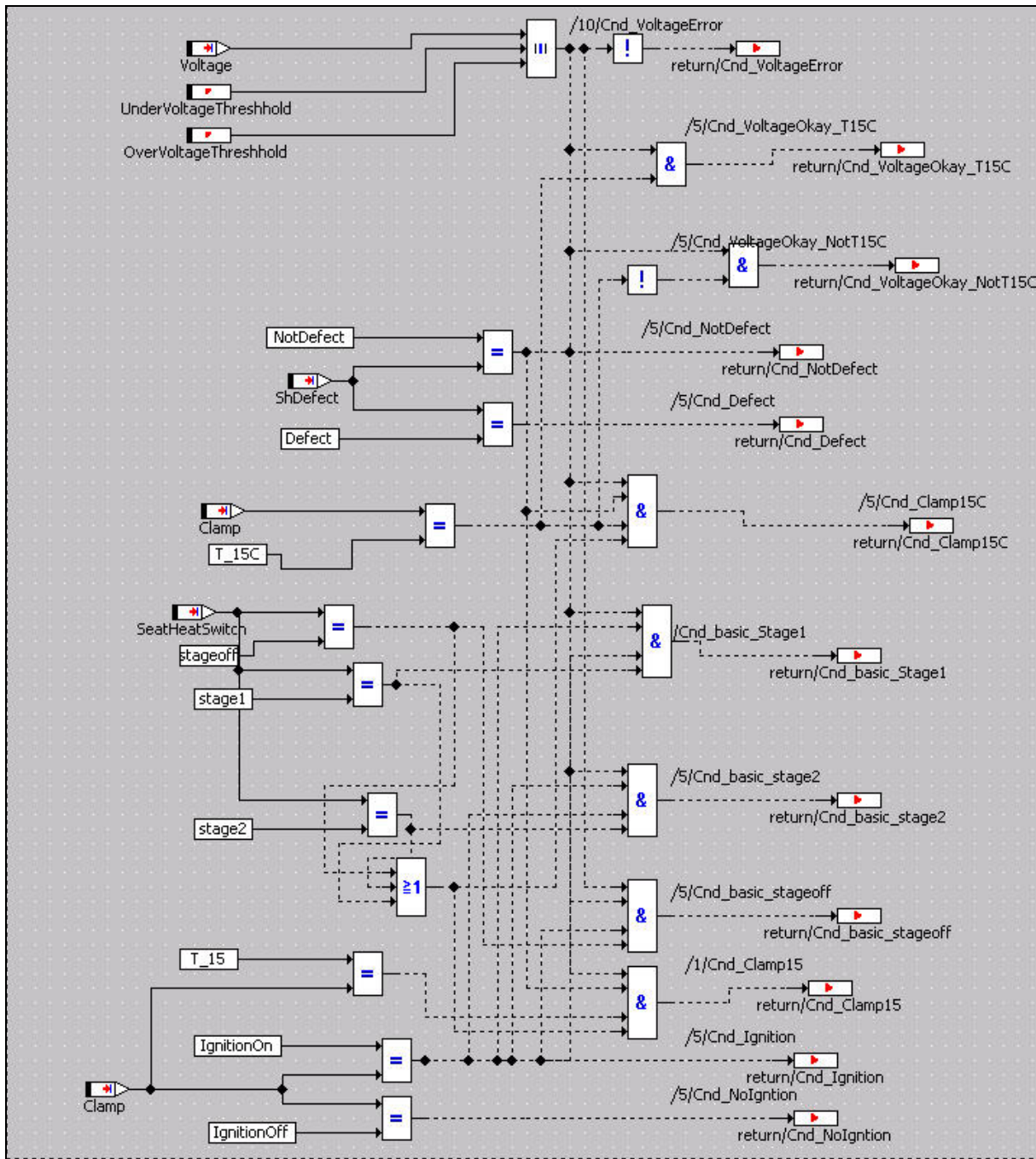
**Figure 28: Trigger Function as Cyclic Runnable Using Sender-Receiver Semantics**



**Figure 29: Graphical "Implementation" of the Trigger Function CyclicRunnable**

Figure 29 shows how the CyclicRunnable triggers the transition by invoking internal condition and action methods. DataRead- and DataWriteAccess of the condition methods is shown in Figure 30 and Figure 31 respectively.





**Figure 30: DataReadAccess of Condition Methods**

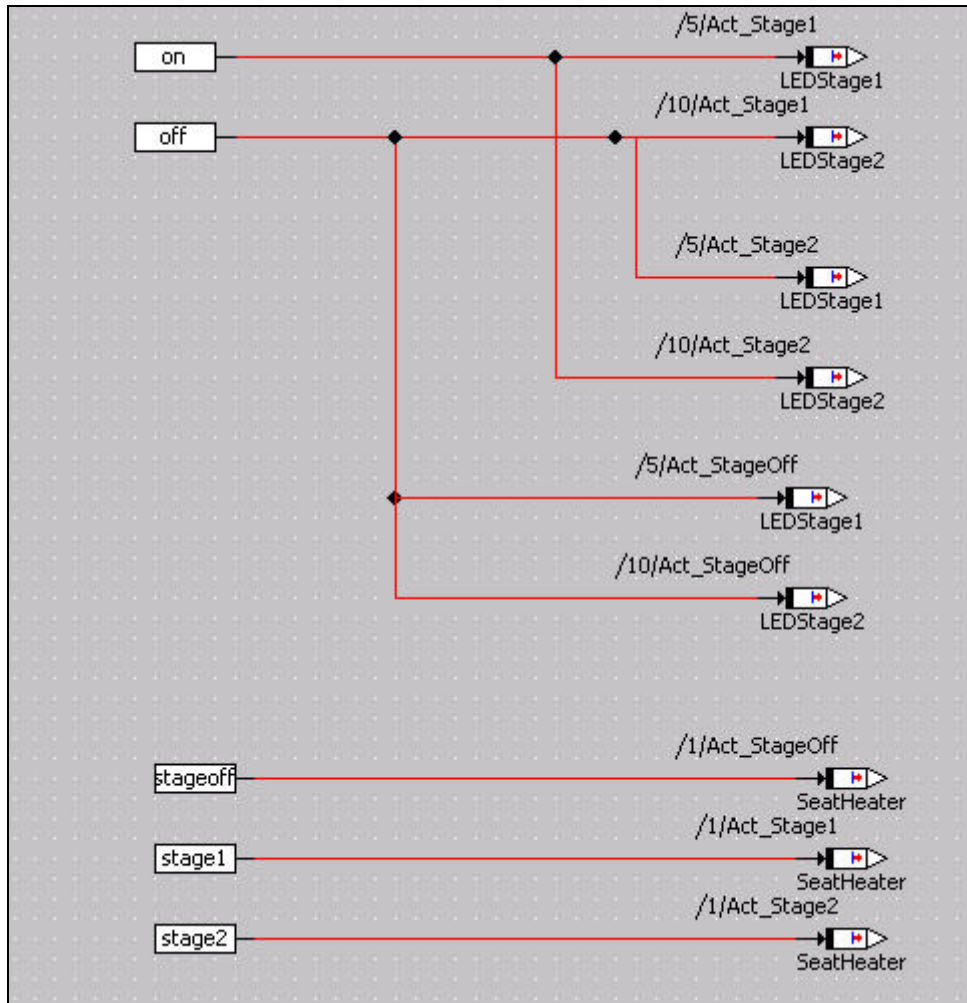
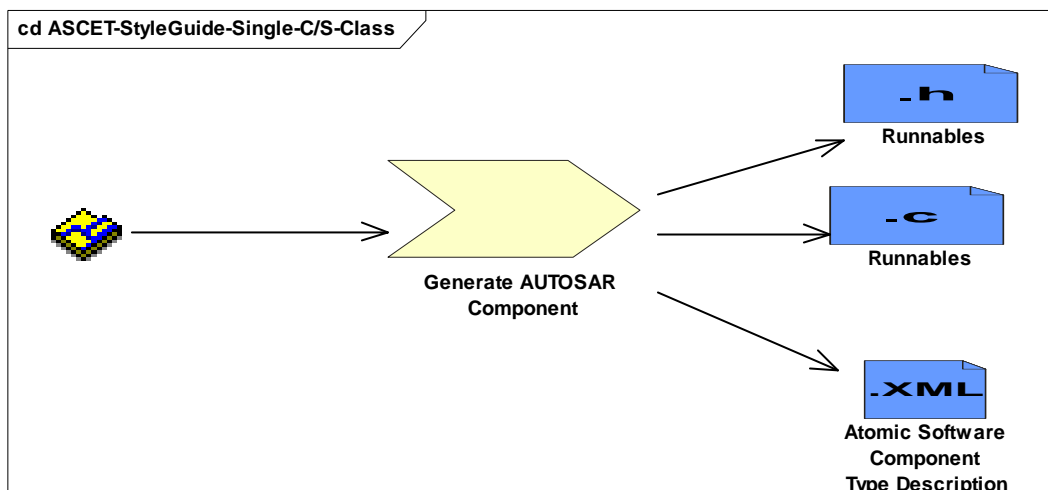


Figure 31: DataWriteAccess of Action Methods

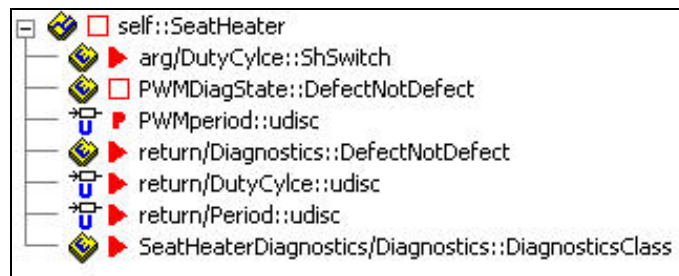
**6.2.7.3 MultipleInstanceType with one PPortPortotype realizing a ClientServerInterface with OperationPrototypes having one OUT ArgumentPrototype.**



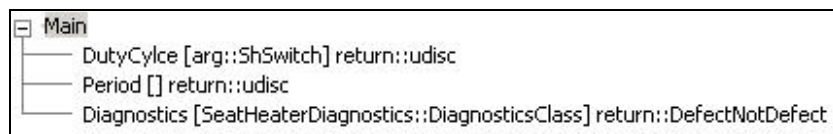
As written in the PortPrototype section 6.2.5, using of ClientServerInterface entities is supported by ASCET with a lot of limitations by using classes. Like their counterparts finite-state-machine-classes, “generic” classes can be instantiated multiple times on an ECU, thus having multiple “prototypes” in an AUTOSAR virtual functional bus description. The interface of a class is given one exclusive PPortPrototype realizing a ClientServerInterface whose OperationPrototype entities can have an arbitrary number of IN ArgumentPrototypes and at most one OUT ArgumentPrototype which is realized as method return value.

This means that the name of the exclusive PPortPrototype name is given implicitly by concatenating *PPort* to the name of the ASCET class. Figure 32 shows that the ArgumentPrototype entities of the OperationPrototype entities, as depicted in Figure 33, are typed either by used-defined enumeration (e.g. ShSwitch) or built-in types (e.g. duty-cycle).

In the seat heating function example, the SoftwareComponentType SeatHeater is an ActuatorSoftwareComponentType providing the translation between atomic software components and the ECU abstraction layer. In particular, it realizes the translation between AUTOSAR types and SignalClasses. The duty-cycle is determined by the chosen stage of the seat heat controller and indicated by the actual value of the enumeration ShSwitch.



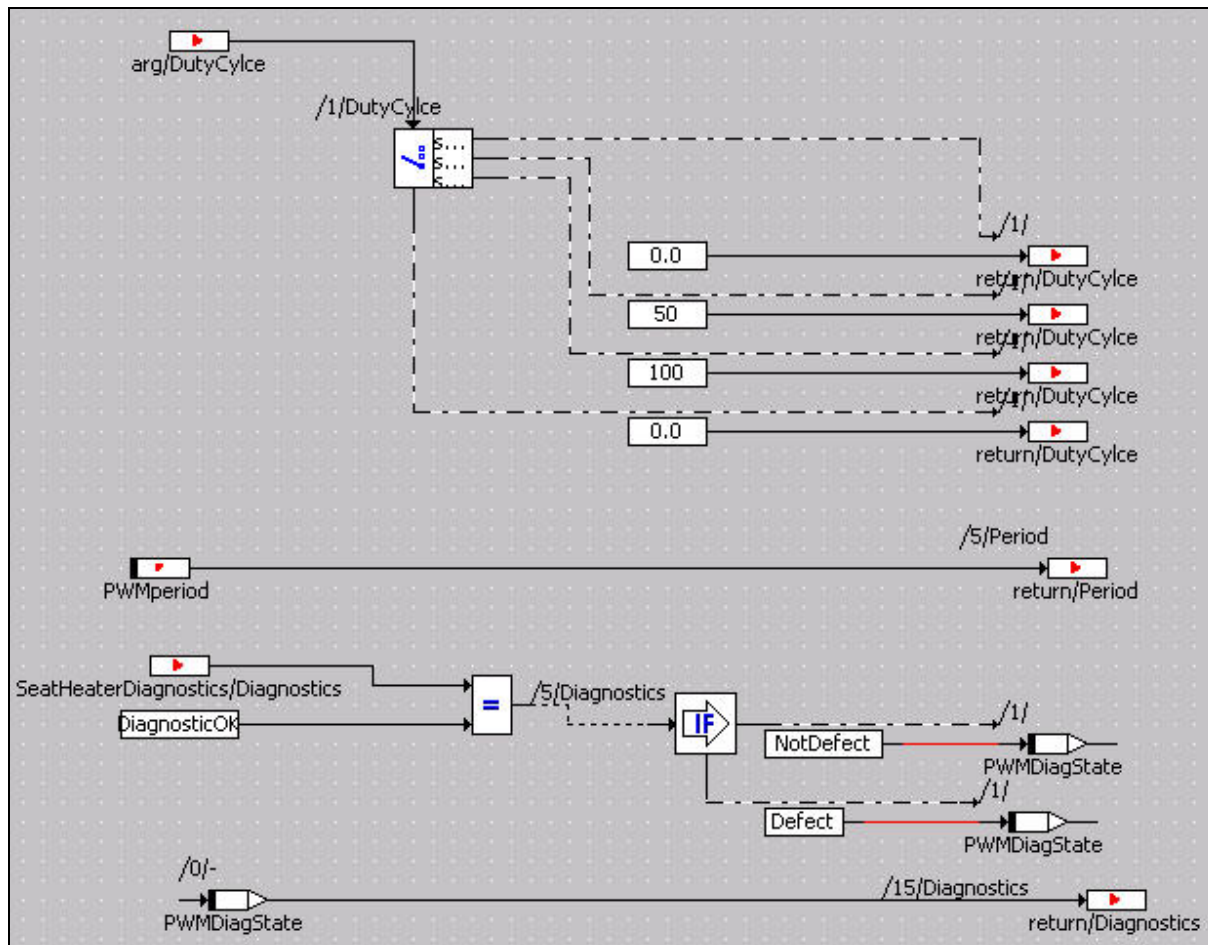
**Figure 32: Interface of the SeatHeater ComponentType**



**Figure 33: OperationPrototypes of the SeatHeat ComponentType**

Figure 33 shows the OperationPrototypes of the ClientServerInterface. These OperationPrototypes become RunnableEntity elements of the internal behavior with an OperationInvokedEvent attached to them.



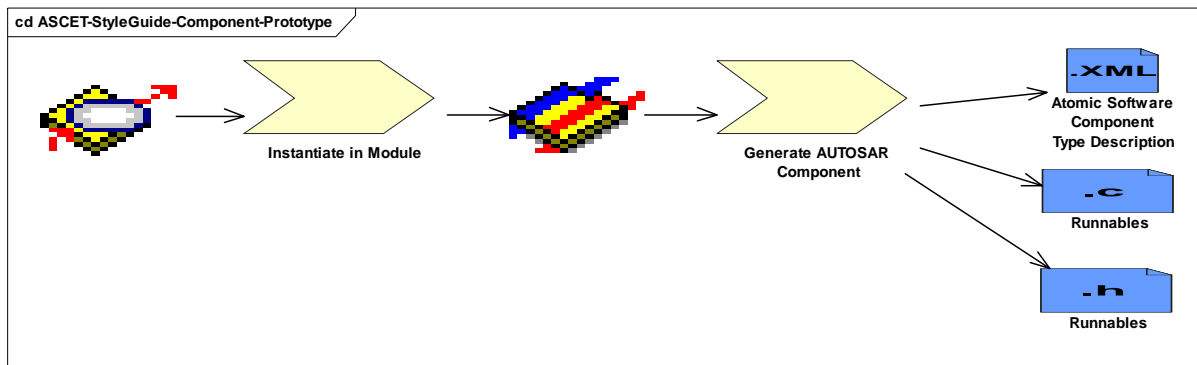


**Figure 34: Graphical Implementation of the ClientServerInterface of the SeatHeater ComponentType**

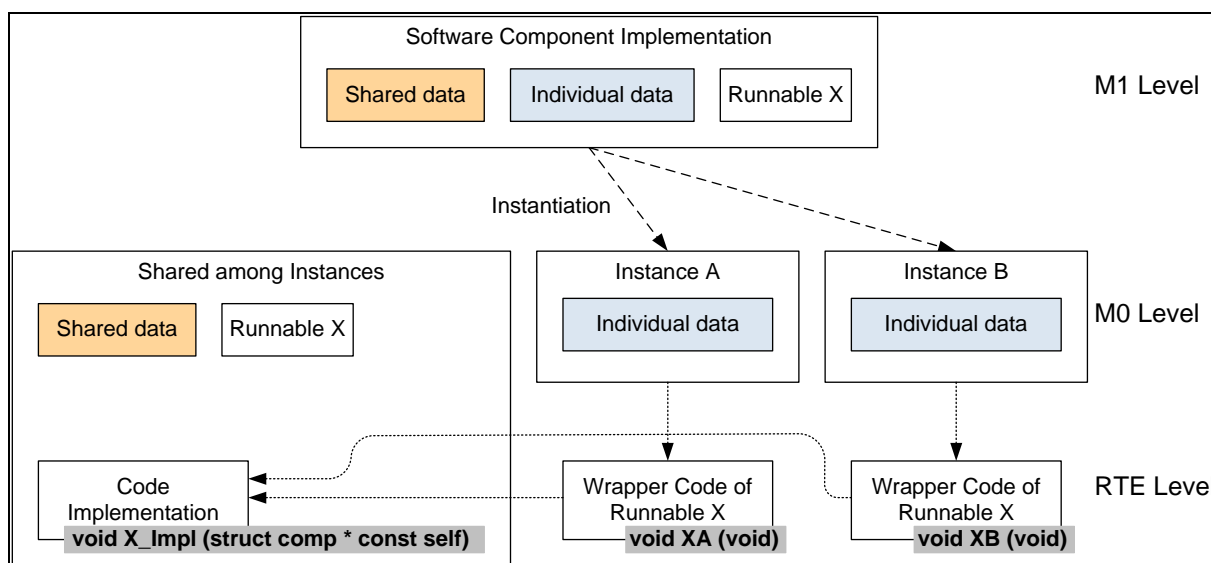
## 6.3 Creating SoftwareComponentPrototype Entities in ASCET

### 6.3.1 Wrapper Runnables

Prototypes of AtomicSoftwareComponent entities are realized in ASCET by modules. Besides the single instance case described in section 6.2.7.1, where a module can also be seen as component-type to have an associated internal behavior, modules can be seen as “wrapper” for ComponentType entities with the ASCET limitations as described in section 6.2.7.2 and 6.2.7.3.

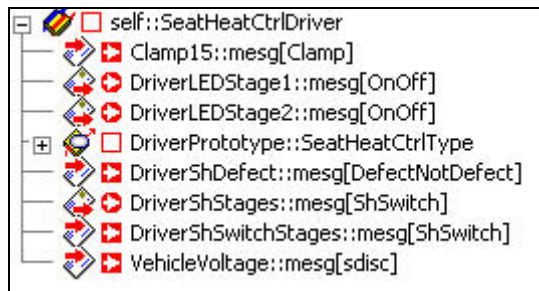


ASCET employs a type/prototype concept as being explained in chapter 6.8.2 (Multiple Instantiation) in the software component template specification[8]. The RunnableEntity of the InternalBehavior referencing an AtomicSoftwareComponentType (and requiring an instance handle to identify the appropriate ComponentPrototype) is embedded in a RunnableEntity for a single instance ComponentPrototype. This single-instance-runnable provides wrapper code which uses the appropriate instance handle when calling the RunnableEntity of the InternalBehavior referencing the AtomicSoftwareComponentType. This principle is shown in Figure 35.

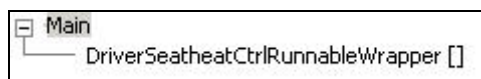


**Figure 35: Wrapper Runnables**

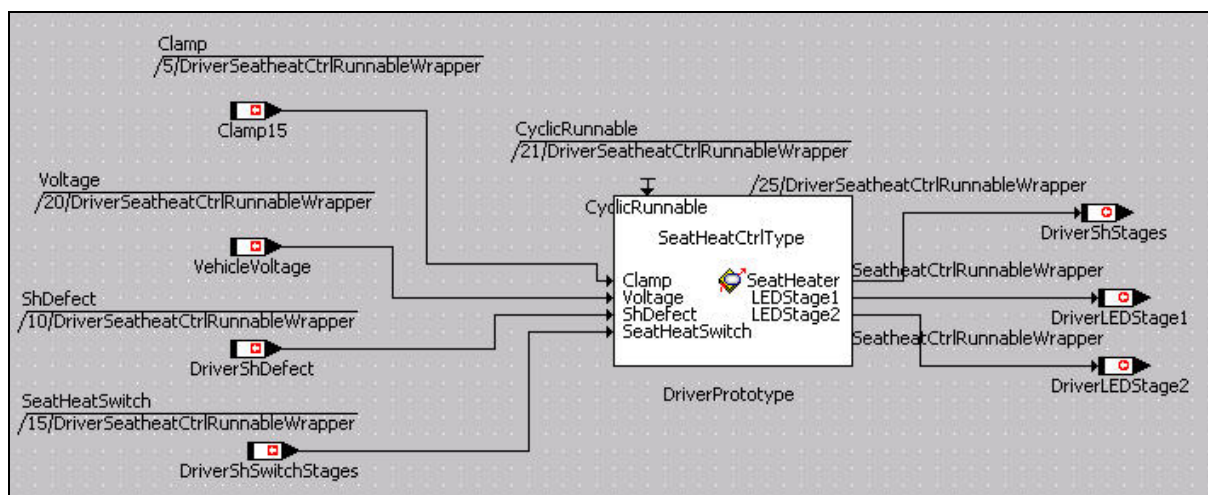
Furthermore, this wrapper-code transfers (in case of SenderReceiverInterfaces) the DataElementPrototype entities from an instance individual port, i.e. a RPortPrototype of a single instance software component, to the RPortPrototype of the ComponentPrototype. In ASCET the shared code and the in instance-individual data has to be modeled explicitly as ASCET class. On top of that a module has to be created for each software component prototype. The processes which are “void-void-functions” will call the methods of class instances with a handle to the individual data.



**Figure 36: Driver ComponentPrototype of the SeatHeatCtrl Component Type**



**Figure 37: Wrapper Runnable of Driver SeatHeatCtrl Prototype**



**Figure 38: Graphical Wrapper Code of the Driver SeatHeatCtrl ComponentPrototype**

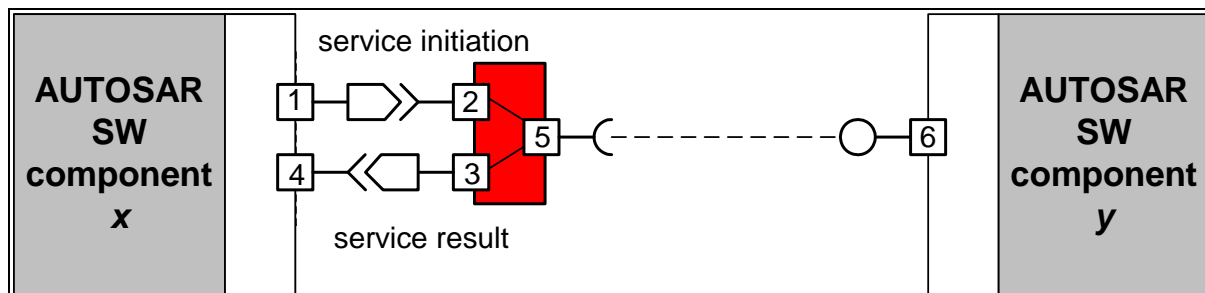
Figure 36 shows the wrapper module `DriverSeatHeatCtrlPrototype` with instance individual `PortPrototype` entities, i.e. the send- and receive-messages, as well as the `ComponentPrototype` `DriverPrototype` of `SeatHeatCtrlType`. Since the wrapper module is also a single-instance `ComponentType` in the sense of chapter 6.2.7.1, the wrapper runnable as shown in Figure 37 provides the `InternalBehavior`.

Figure 38 shows the (graphical) wrapper code of the wrapper runnable realizing the instance-handle administration and the transfer of the instance individual “port-data” to the `PortPrototypes` of the `ComponentPrototype` in providing- and requiring direction.

### 6.3.2 Port-Type-Converters

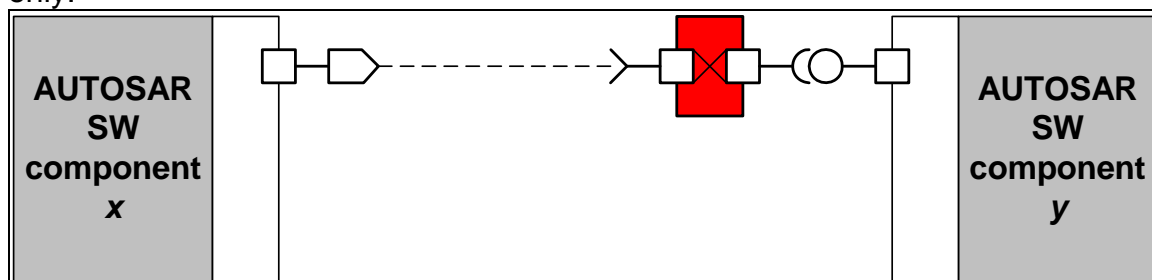
As explained in section 6.2.7.3, ASCET supports the specification of `AtomicSoftwareComponentType` entities having one implicit `PPortPrototype` realizing a `Client-ServerInterface`. If such a `ComponentType` is used in ASCET as `ComponentProto-`

type, an ASCET wrapper module has to be constructed. The wrapper-code then realizes also the port-type-converter concept as specified in the AUTOSAR Virtual Functional Bus Description[1] and shown in Figure 39. DataElementPrototype entities in a SenderReceiverInterface of a RPortPrototype are transferred to an IN ArgumentPrototype of an OperationPrototype in a ClientServerInterface. An OUT ArgumentPrototype will be transferred to a DataElementPrototype in a SenderReceiverInterface of a PPortPrototype.



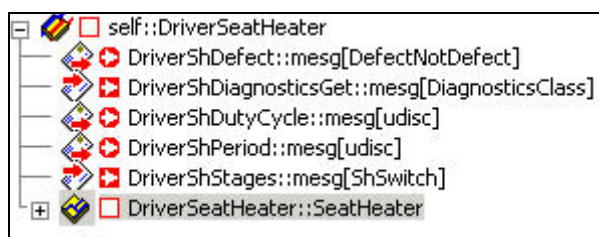
**Figure 39: Port-Type Converter Sender-Receiver Client**

As shown in Figure 40, Port-Type-Converters can also be specified in one direction only.



**Figure 40: Port-Type Converter Sender Client**

In the seat-heating function example, the port-type-converter concept in conjunction with a wrapper module has been used for the SeatHeater (Actuator) SoftwareComponent. There are messages for the DriverShDefect of user-defined enumeration DefectNotDefect, the DriverShStages of user-defined enumeration ShSwitch, as well as DriverShDiagnosticsGet for the user-defined enumeration DiagnosticsClass. The other messages are needed to define the PWM-communication with IoHwAbstractionLayer

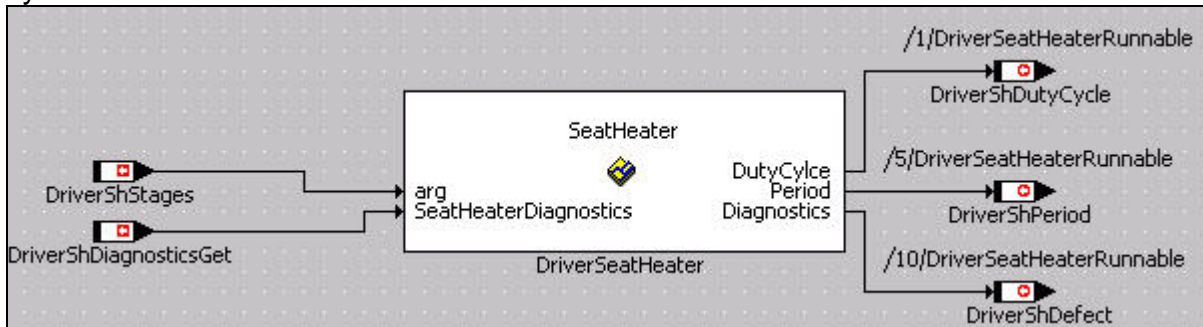


**Figure 41: Wrapper Module for Driver SeatHeater ActuatorSoftwareComponentPrototype**



**Figure 42: Wrapper Runnable implementing a Port-TypeConverter**

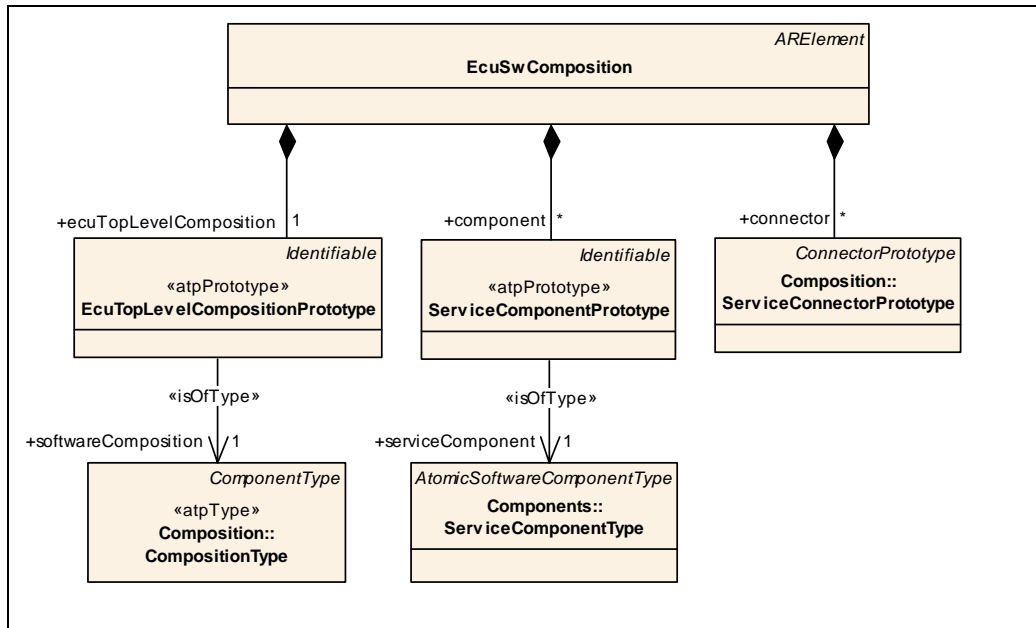
The semantics are similar to the example with the ComponentPrototype *DriverPrototype* of the ComponentType *DriverSeatHeatCtrl*. However, the actual values of the send-messages, realizing the DataElementPrototype in a SenderReceiverInterface of the RPortPrototype (with the same name as the DataElementPrototype) are copied to the ArgumentPrototype of the OperationPrototype of the ComponentPrototype *DriverSeatHeater*. The wrapper runnable has to ensure that all DataElementPrototype entities are given “synchronously” to the ArgumentPrototype entities. However, from the runnable point of view, the client-server call has sender-receiver semantics with additional synchronization.



**Figure 43: Graphical Implementation of Wrapper Code realizing the PortType Conversion**

## 7 The ECU Composition

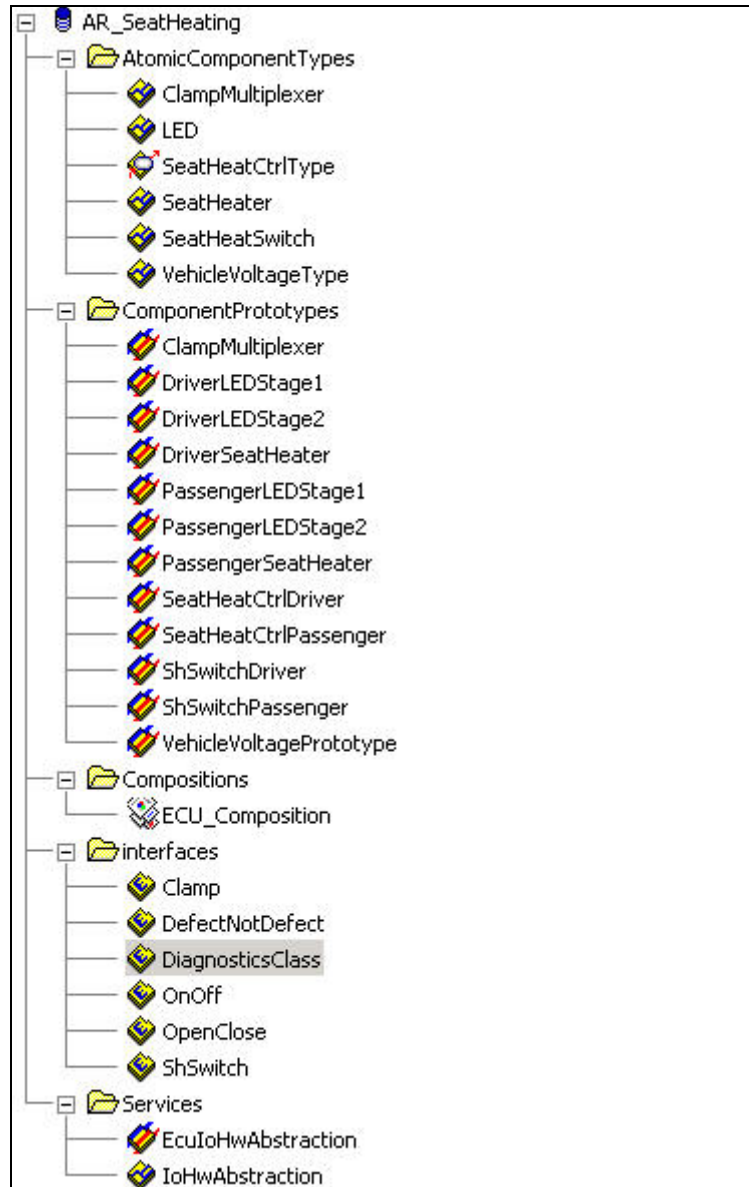
An ASCET project can be seen as an EcuSwComposition (see Figure 44) in the integration tool use-case. All wrapper modules constitute the EcuTopLevelComposition-Prototype. From the ASCET point of view, it does not matter whether the wrapper module for the IoHardwareAbstractionLayer is part of the EcuTopLevelComposition-Prototype or the ServiceComponentPrototype. Dedicated ServiceConnectorPrototype entities can be generated from the message name matching.



**Figure 44: The ECU Software Composition**

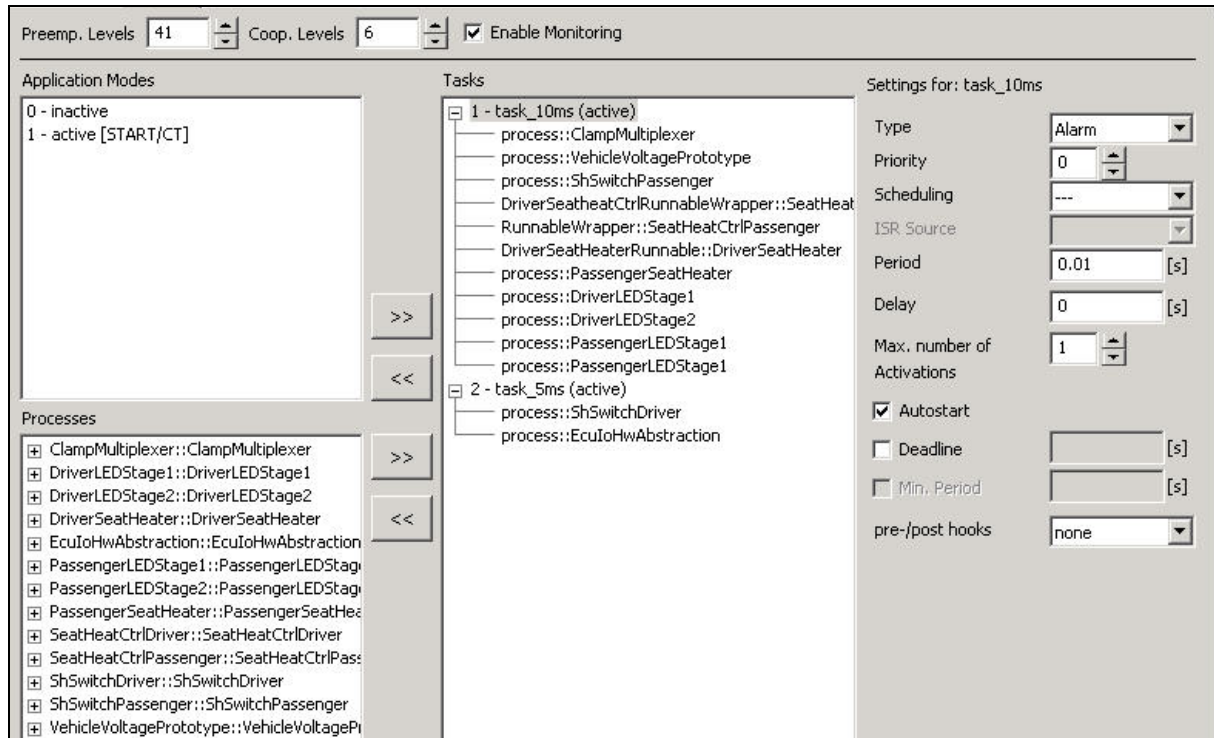
Applying the integration tool use-case to the seat-heating-system results in a system as shown in Figure 45 and lists all atomic software component types as classes, the prototypes as wrapped modules and a project as EcuSwComposition. Furthermore, the I/O hardware abstraction layer is put under services, though it is not an AUTOSAR service literally.





**Figure 45: ASCET model elements realizing the Seat heating Functionality**

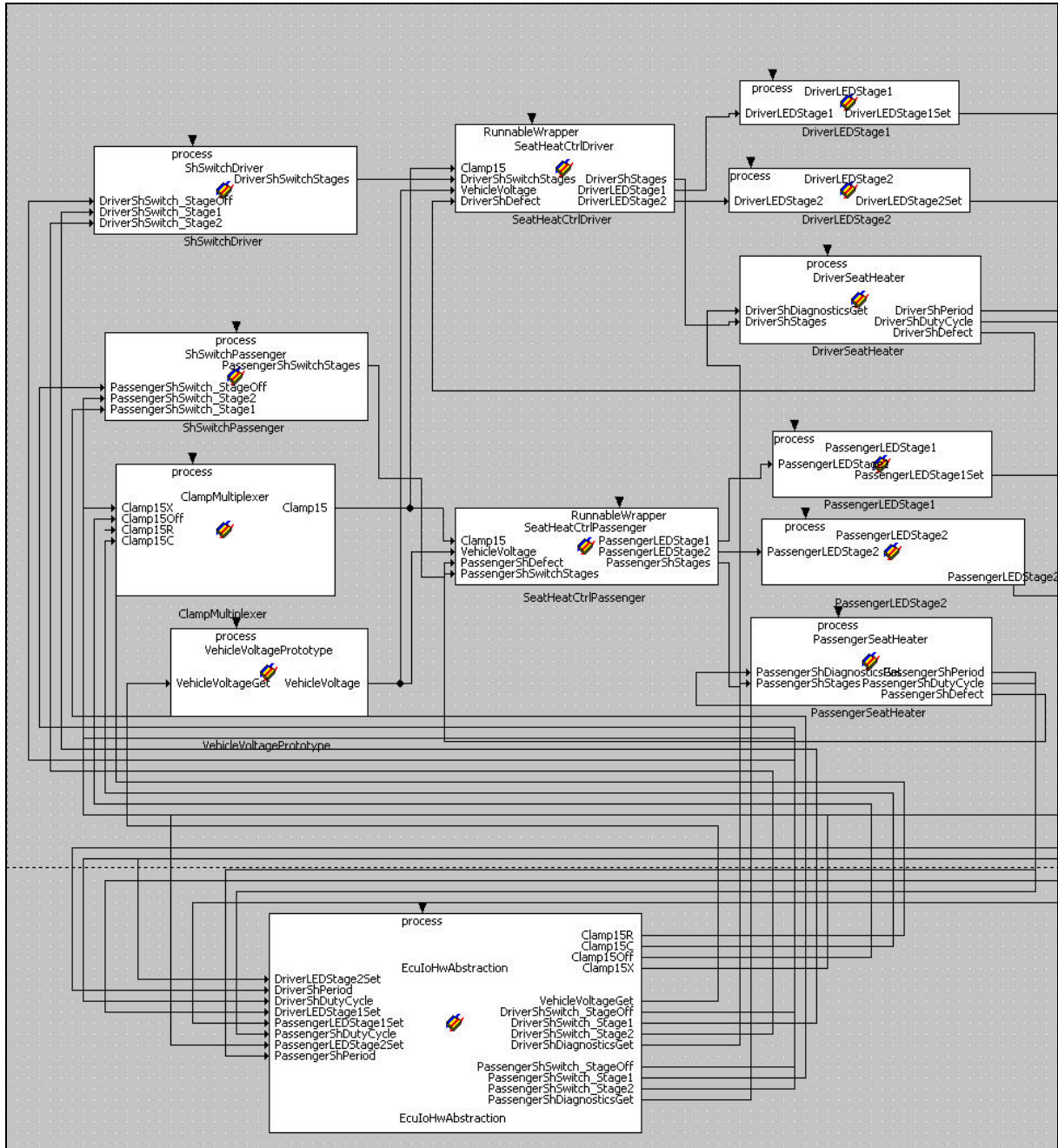
The Composition of the seat heating function is shown in Figure 47. This view shows all AtomicSoftwareComponentPrototype entities. This composition can be transformed into an ECU-composition allowing the ECU-Configuration of the RTE. For example, the RTEEvent entities of the processes are given by the period of the task. The allocation of the wrapper-runnables to tasks can also be seen here. Since the wrapper-runnables of the ComponentPrototype entities hide the instance handles of the RunnableEntity elements, they do not need any further arguments. However, for each ComponentPrototype, a wrapper-runnable has to be assigned to the task. The wrapper-runnable name can be used to derive the SoftwareComponentInstanceRef of the RunnableEntityMapping container as described in the ECU-C-ParameterDefinitions document.



**Figure 46: Allocation of Wrapper-Runnables to Tasks**

Using ASCET as authoring tool, an RTEEvent of a RunnableEntity can only be determined if runnables are mapped to an OS task. In the EcuSwComposition shown in Figure 47 consists of the ECUTopLevelCompositionPrototype which is the CompositionType aggregating all ComponentPrototypes. The IoHwAbstraction is wrapped as ServiceComponentPrototype, where the get- and set functions of the I/O hardware abstraction layer is wrapped by a runnable, which means that the application software components use sender-receiver communication





**Figure 47: The ECU Composition of the Seat heating Function**

## 8 Summary

ASCET can be used in AUTOSAR for the use-cases integration tool and additional programmer. ASCET supports the type/prototype concepts by means of wrapper-runnables and port-type-converts. In the integration tool use-case, the ASCET project serves as EcuTopLevelCompositionPrototype whereas in the additional programmer use-case, the ASCET project represents a single-instance prototype of an Atomic-SoftwareComponent, clustering all other ComponentPrototypes.

In additional programmer use-case, port-prototypes are represented by dedicated input- and output modules.

## 9 References

- [1] Specification of the Virtual Functional Bus  
AUTOSAR\_Spec\_of\_VFB.pdf
- [2] Specification of the RTE  
AUTOSAR\_SWS\_RTE.pdf
- [3] Specification of Interaction with Behavioral Models  
AUTOSAR\_InteractionBehavioralModels.pdf
- [4] Metamodel  
AUTOSAR\_Metamodel.eap
- [5] Glossary  
AUTOSAR\_Glossary.pdf
- [6] Requirements on Interoperability of Authoring Tools  
AUTOSAR\_RS\_InteroperabilityAuthoringTools.pdf
- [7] Methodology  
AUTOSAR\_Methodology.pdf
- [8] Software Component Template  
AUTOSAR\_SoftwareComponentTemplate.pdf
- [9] Specification of System Template  
AUTOSAR\_SystemTemplate.pdf
- [10] Specification of Feature Definition of Authoring Tools  
AUTOSAR\_FeatureDefinition.pdf
- [11] Specification of Graphical Notation  
AUTOSAR\_GraphicalNotification.pdf
- [12] Specification of the Runtime Environment  
AUTOSAR\_SWS\_RTE.pdf
- [13] Specification of the I/O Hardware Abstraction Layer  
AUTOSAR\_SWS\_IOHWAbstraction.pdf
- [14] Specification of the ECU Configuration  
AUTOSAR\_ECU\_Configuration.pdf