Document Title	Specification of BSW Module Description Template
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	089
Document Classification	Standard

Document Version	1.1.0
Document Status	Final
Part of Release	3.0
Revision	7

Document Change History			
Date	**Version**	**Changed by**	**Description**
07.08.2010	1.1.0	AUTOSAR Administration	• Added option to MemorySection
27.11.2007	1.0.0	AUTOSAR Administration	• Initial Release

Table of Contents

Bibliography

[1] Requirements on Basic Software Module Description Template
AUTOSAR_RS_BSW_ModuleDescription.pdf

[2] General Requirements on Basic Software Modules
AUTOSAR_SRS_General.pdf

[3] Methodology
AUTOSAR_Methodology.pdf

[4] Glossary
AUTOSAR_Glossary.pdf

[5] Software Component Template
AUTOSAR_SoftwareComponentTemplate.pdf

[6] Template UML Profile and Modeling Guide
AUTOSAR_TemplateModelingGuide.pdf

[7] Model Persistence Rules for XML
AUTOSAR_ModelPersistenceRulesXML.pdf

[8] List of Basic Software Modules
AUTOSAR_BasicSoftwareModules.pdf

[9] Meta Data Exchange Format for Software Module Sharing V1.0 (MDX V1.0)
ASAM-AE-MDX-V1_0_0.pdf

[10] Specification of ECU Configuration
AUTOSAR_ECU_Configuration.pdf

[11] Specification of BSW Scheduler
AUTOSAR_SWS_BSWSch.pdf

[12] System Template
AUTOSAR_SystemTemplate.pdf

[13] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

[14] Design Specification for the ECU Resource Template
AUTOSAR_ResourceTemplateECU.pdf

1 General Information

1.1 Document Scope

This is the documentation of the template for the Basic Software Module Description (BSWMDT).

The BSWMD is a formal notation of all information, which belongs to a certain BSW artifact (BSW module or BSW cluster) in addition to the implementation of that artifact. There are several possible use cases for such a description.

The BSWMDT is the standardized format which has to be used for this description in AUTOSAR. The template is represented in UML as part of the overall AUTOSAR meta-model and is part of the XML schema generated out of this meta-model. This document describes all the elements used in that template.

Some elements of the BSWMDT, for example for the description of implementation aspects and resource consumption, are used also within the Software Component Template (SWCT). These elements belong to the `CommonStructure` package of the meta-model. These common elements are also described within this document, as far as it is useful to understand the BSWMDT.

This document addresses people who need to have a deeper understanding of the BSWMDT part of the meta-model, for example tool developers and those who maintain the meta-model. It is not intended as a guideline for the BSW developers who will have to provide the actual BSWMD, i.e. who have to ”fill out” the template.

For further information on the overall goal of this document refer to the related requirements document, see [1].

1.2 Input Documents

The following input documents have been used to develop the BSWMDT:

- Requirements on BSW Module Description Template [1]
- General Requirements on Basic Software Modules [2]
- AUTOSAR Methodology [3]
- AUTOSAR Glossary [4]
- Software Component Template [5]
- AUTOSAR Template UML Profile and Modeling Guide [6]
- AUTOSAR Model Persistence Rules for XML [7]

1.3 Abbreviations

Abbreviation	*Meaning*
BSW	Basic Software
BSWMD	Basic Software Module Description
BSWMDT	Basic Software Module Description Template
ECU	Electronic Control Unit
ECUC	ECU Configuration
ICC1, ICC2, ICC3	AUTOSAR Implementation Conformance Class 1. . . 3
ISR	Interrupt Service Routine
ICS	Implementation Conformance Statement
MSR	Manufacturer Supplier Relationship
OS	Operating System
SW-C	Software Component
SWCT	Software Component Template
UML	Unified Modeling Language
XML	Extended Markup Language

2 Requirements Traceability

The following table references the requirements specified in AUTOSAR BSWMD Requirements [1] and denotes how they are satisfied by the meta-model.

Requirement	Description	Satisfied by
[BSWMD0001]	Main source of information on BSW Module ECU Configuration activity and integration	Complete BSWMDT
[BSWMD0005]	Description of the memory needs of the BSW Module implementation	MM:ResourceConsumption. stackUsage, MM:ResourceConsumption. objectFileSection
[BSWMD0007]	Provide vendor-specific published information	MM:BswImplementation. preconfiguredConfiguration
[BSWMD0008]	BSW Module Description SHALL be tool processable	Generated XML schema
[BSWMD0009]	Description of peripheral register usage	MM:BswImplementation. requiredHW
[BSWMD0010]	Compiler version and settings	MM:Implementation. compiler
[BSWMD0011]	Guaranteed execution context of API calls	MM: BswModuleDependency. requiredEntry. executionContext
[BSWMD0013]	Describe configuration class of ECU Configuration Parameters	MM:BswImplementation. vendorSpecificModuleDef
[BSWMD0014]	Support of BSW Module clusters	Complete BSWMDT
[BSWMD0016]	Timing guarantees	MM:ResourceConsumption. executionTime
[BSWMD0024]	Support description of module specific published information	MM:BswImplementation. vendorSpecificModuleDef
[BSWMD0025]	Support for shipment information	This is not specific for the shipment of BSWMD. It is handled in general by the root element of an AUTOSAR description MM:AUTOSAR. adminData
[BSWMD0026]	Description of supported hardware	MM:BswImplementation. requiredHW
[BSWMD0027]	Provide vendor-specific ECU Configuration Parameter Definition	MM:BswModuleDescription. vendorSpecificModuleDef
[BSWMD0028]	Development according to the AUTOSAR Metamodeling Guide	Complete BSWMDT
[BSWMD0029]	Transformation of BSWMD modeling according to the AUTOSAR Model Persistence Rules for XML	Implicitly solved by having the BSWMDT in the same EAP file as all templates
[BSWMD0030]	Publish resource needs for the BSW Scheduler	MM:BswBehavior
[BSWMD0031]	Description of used memory section names	MM:ResourceConsumption. objectFileSection
[BSWMD0032]	Recommended ECU Configuration Values	MM:BswImplementation. recommendedConfiguration
[BSWMD0033]	Pre-configured ECU Configuration Values	MM:BswImplementation. preconfiguredConfiguration
[BSWMD0034]	ECU Configuration Editor and Generation supported tool version information	MM:Implementation. implementationDependency

Requirement	Description	Satisfied by
[BSWMD0035]	Provide standardized ECU Configuration Parameter Definition	MM:BswImplementation. vendorSpecificModuleDef. refinedConfiguration
[BSWMD0037]	Needed libraries	MM:Implementation. implementationDependency
[BSWMD0038]	Required execution context of API calls	MM: BswModuleDescription. providedEntry. executionContext
[BSWMD0039]	Identification of implemented API and functions	MM:BswModuleDescription. providedEntry
[BSWMD0040]	Identification of required API and functions	MM:BswModuleDescription. bswModuleDependency. requiredEntry
[BSWMD0041]	Declaration of the provided API argument data types	MM:BswModuleDescription. providedEntry
[BSWMD0042]	Description of the required API argument data types	MM:BswModuleDescription. bswModuleDependency. requiredEntry
[BSWMD0043]	Support description of common published information	MM: Attributes of BswImplementation
[BSWMD0045]	Publish resources needed from AUTOSAR Services	MM: BswModuleDependency. serviceItem
[BSWMD0046]	Publish OS resource usage	MM:BswBehavior...
[BSWMD0047]	Modeling of call-chain dependencies between BSW Modules	MM:BswModuleEntity. calledEntry
[BSWMD0048]	Tagging of vendor-specific ECU Configuration Parameter Definition	Solved in the ECU Parameter Definition Template, MM:ConfigParameter. origin
[BSWMD0050]	Allow vendor-specific modification of standardized ECU Configuration Parameter Definition	MM:BswImplementation. vendorSpecificModuleDef

3 Use Cases and Modeling Approach

3.1 Use Cases

There are several possible use cases for the BSWMDT. The following uses cases can be applied for BSW modules (ICC3 conformance class) or for BSW clusters (ICC2 conformance class).

- It can be used to *specify* a BSW module or cluster in terms of interfaces and dependencies before it is actually implemented. Details of the implementation are not filled out for this use case.
- It can be used as input for a *conformance* test, which tests the conformance of the product (a BSW module or cluster) with respect to the AUTOSAR standard. The work products include the implementation (source or object code) and its BSWMD (in XML). In other words this means that for a conformance test the BSWMD must be usable as an ICS (implementation conformance statement). Note that this use case is different from the following one (the integration use case) because conformance test cases will typically cover a wider range of functionality and configuration values than required for the integration on a specific ECU.
- It can be used to describe an *actually implemented* BSW module or cluster given to the integrator of an AUTOSAR ECU. It will contain details of the actual implementation and constraints w.r.t. the specification. Especially, there may be more than one implementation (for example for different processors) which have the same specification.
- It may be used by the *integrator* to add further information which has not been filled out by the deliverer of the module (this is maybe against the idea of the BSWMD as being a description of a deliverable, but it is in principle possible).

Details of the work flow for the different use cases are not in the scope of this document (please refer to [3]), but the information to be provided in these various steps influences the meta-model of the BSWMDT.

There is ony limited use for the BSWMDT to describe software according to ICC1 conformance class, because in this case the complete BSW (including RTE) on an ECU consists of one single cluster, so that no interfaces or dependencies within the BSW can be described by this template, which means that the relevant parts of the template will be empty. However, even in this case the BSWMDT may be used to document implementation aspects (e.g. the required Compiler, resource consumption or vendor specific configuration parameters).

3.2 Three Layer Approach

The meta-model of the BSWMDT consists of three abstraction layers similar to the SWCT. This approach allows for a better reuse of the more abstract parts of the description. An overview is shown in Figure 3.1.

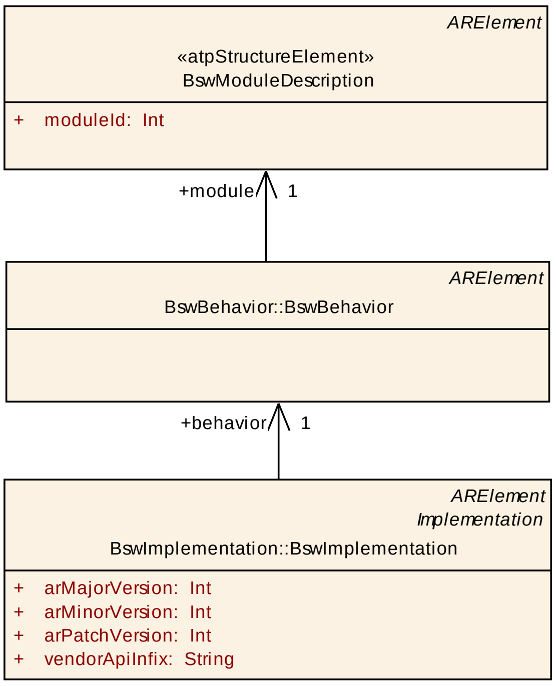

Figure 3.1: Three Layers of the BSW Module Description

The upper layer, the `BswModuleDescription`, contains the specification of all the provided and required interfaces including the dependencies to other modules.

The middle layer, the `BswBehavior`, contains a model of some basic activity inside the module. This model defines the requirements of the module for the configuration of the OS and the BSW Scheduler. There may be several different instances of `BswBehavior` fulfilling the same `BswModuleDescription`. The term "behavior" has been chosen in analogy to a similar term in the SWCT. Note that it is restricted only to the scheduling behavior here and does not descibe the behavior of the module or cluster completely.

The bottom layer, the `BswImplementation` contains information on the individual code. Again, there may be several instances of `BswImplementation` for the same `BswModuleBehavior`.

The usage of references between these layers instead of aggregations allows for more flexibility in the XML artifacts: If for example the `BswBehavior` would aggregate `BswImplementation`, a concrete XML artifact of a `BswBehavior` would have to be duplicated for every instance of `BswImplementation`. With references, the layers may be kept in separate files. This is analog to the inclusion of header files in a C-source file: Several implementation files can share the same header file, which typically declares more abstract things as function prototypes and the like.

3.3 Several Implementations of the same BSW Module or BSW Cluster

According to the three layer approach, the meta-class `BswModuleDescription` and an associated `BswModuleBehavior` describe a type of a BSW module or cluster, for which different implementations may exist which are represented by different `BswModuleImplementations` (note that the name of the meta-class `BswModuleDescription` is misleading here, because this meta-class does not contain the complete description of a module or cluster).

In case the different implementations of a BSW module or cluster are compiled for different CPUs, the corresponding BSWMDs can be treated as separate artifacts which may share the `BswModuleDescription` and/or `BswModuleBehavior`.

In case the implementations are compiled for the same CPU, i.e. are integrated on the same ECU and same address space (for example CAN drivers for several CAN channels), their BSWMDs still should share the `BswModuleDescription` and `BswModuleBehavior`, but there must be a mechanism to ensure, that the globally visible C symbols derived from the `BswModuleDescription` are unique. This is handled with `infixes` defined in the implementation part of the BSWMDT (see chapter 7).

3.4 Relation to SoftwareComponentType

Some BSW modules or clusters not only have interfaces to other BSW modules or clusters, but have also more abstract interfaces accessed from application SW-Cs via the RTE. These BSW modules or clusters can be AUTOSAR Services, part of the ECU Abstraction, or Complex Drivers.

The more abstract interfaces required here are called AUTOSAR Interfaces (see [5] and [4]). These AUTOSAR Interfaces are described by means of the Software Component Template (SWCT), they consist of ports, port interfaces and their further detailing. The root classes of the SWCT used to describe these elements for BSW modules are `ServiceComponentType`, `EcuAbstractionComponentType` and `ComplexDeviceDriverComponentType` (see [5]) which all are derived from `AtomicSoftwareComponentType`.

In addition, the function calls from the RTE into these BSW module must be modeled as so-called Runnable Entities (”runnables”), which is also contained in the SWCT. The root class of the SWCT used to describe the runnables (and a few other things) is called `InternalBehavior`.

Thus for BSW modules or clusters which can be accessed via AUTOSAR Interfaces there must be an XML-artifact defining an `AtomicSoftwareComponentType` and an `InternalBehavior` *in addition* to the BSWMD. These additional descriptions are required to generate the RTE. Note that the content of these additional descriptions can vary between different ECUs (for example due to the number of ports the RTE has to create for an AUTOSAR Service) and thus must be created per ECU. The detailed steps for creating these artifacts are described in [5].

In order to trace the dependencies between these additional SWCT descriptions and the associated BSWMD, there is a reference from the classes `ServiceComponentType`, `EcuAbstractionComponentType` and `ComplexDeviceDriverComponentType` to `BswModuleDescription` and from `InternalBehavior.runnable` to its counterpart in the BSWMDT, the class `BswModuleEntity`. Note that there are no references the other way round, because the SWCT descriptions may be created after delivery of a BSWMD.

It should be noted, that there is a certain ambiguity in the architecture regarding scheduling in that a scheduled functionality can be triggered by two different mechanisms: By the BSW Scheduler with the help of an event model defined in the BSWMDT (see chapter 6 in this document) or by the RTE with an event model defined in the `InternalBehavior` of the SWCT. For the AUTOSAR Services defined up to now (AUTOSAR release 3.0) triggering by the RTE is used only for function calls directly related to communication via ports, whereas for e.g. cyclic events the BSW Scheduler shall be used. It is however out of the scope of this document, to define such a rule for the BSW parts which are not standardized (ECU Abstraction and Complex Drivers).

Another special case arises, if a cyclic function triggered by the BSW Scheduler or an interrupt routine has to call into the RTE in order to access an SW-C. In order to generate the RTE API with the means of the current SWCT (Release 3.0), it is required to specify a runnable entity in this case even if it is not triggered by an RTE event. Also in this case the runnable entity must have a reference to the associated `BswModuleEntity`, as mentioned above.

4 BSW Module Description Overview

Figure 4.1 and the following class table show all the relations of the BSWMDT top layer, the BswModuleDescription.

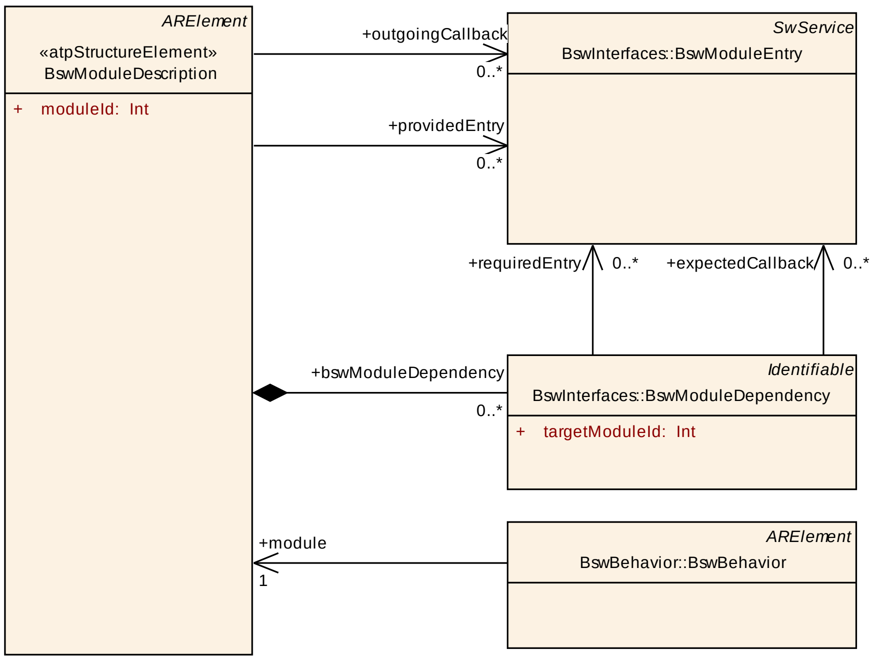

Figure 4.1: BSW Module Description Overview

First of all, the `BswModuleDescription` contains an attribute `moduleId` which refers to the identifier of the standardized AUTOSAR modules according to [8]. This identifier can also be used to distinguish modules which are not standardized (i.e. they belong to the ECU Abstraction or are Complex Device Drivers) or to identify ICC2 clusters. In this case the idenfifier must be chosen differently from the ones given in [8]. In any case, this identifier in the BSWMD shall be used to document the relation of an artifact to the standard and thus is a useful information for the conformance test.

The class `BswModuleEntry` describes a single C-function prototype. The interface exported by a `BswModuleDescription` is a set of `providedEntries` provided for the usage by other modules (including ”main”-functions called by the BSW Scheduler) and of `outgoingCallbacks`, which are declared by this module and will be called, if required by other modules.

With the help of class `BswModuleDependency` it is possible to describe the requirements of a given BSW module onto another BSW module, which among other things includes the interface imported from the other module, namely a set of `requiredEntries` and `expectedCallbacks`. Further details are described in chapter 5.

By the association of class `BswBehavior` to `BswModuleDescription` it is possible to add scheduling aspects to the description.

Class	⟨⟨**atpStructureElement**⟩⟩ **BswModuleDescription**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswOverview			
Class Desc.	Root element for the description of a single BSW module or BSW cluster. In case it describes a BSW module, the short name of this element equals the name of the BSW module.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
bswMod-uleDepen-dency	BswMod-uleDepen-dency	*	aggregation	Describes the dependency to another BSW module.
moduleId	Integer	1	aggregation	Refers to the BSW Module Identifier defined by the AUTOSAR standard.
outgoing Callback	BswMod-uleEntry	*	reference	Specifies a callback, which will be called from this module if required by another module.
provided Entry	BswMod-uleEntry	*	reference	Specifies an entry provided by this module which can be called by other modules. This includes ”main” functions and interrupt routines, but not callbacks (because the signature of a callback is defined by the calller).

Table 4.1: BswModuleDescription

5 BSW Interface

This chapter describes the meta-model elements which are used to define the interface level of a BSW module: The description of `providedEntries`, `outgoingCallbacks` and the dependencies from other modules.

5.1 BSW Module Entry

The class `SwService`[1] from MSR is used to model the signature of a C-function call, see figure 5.1.

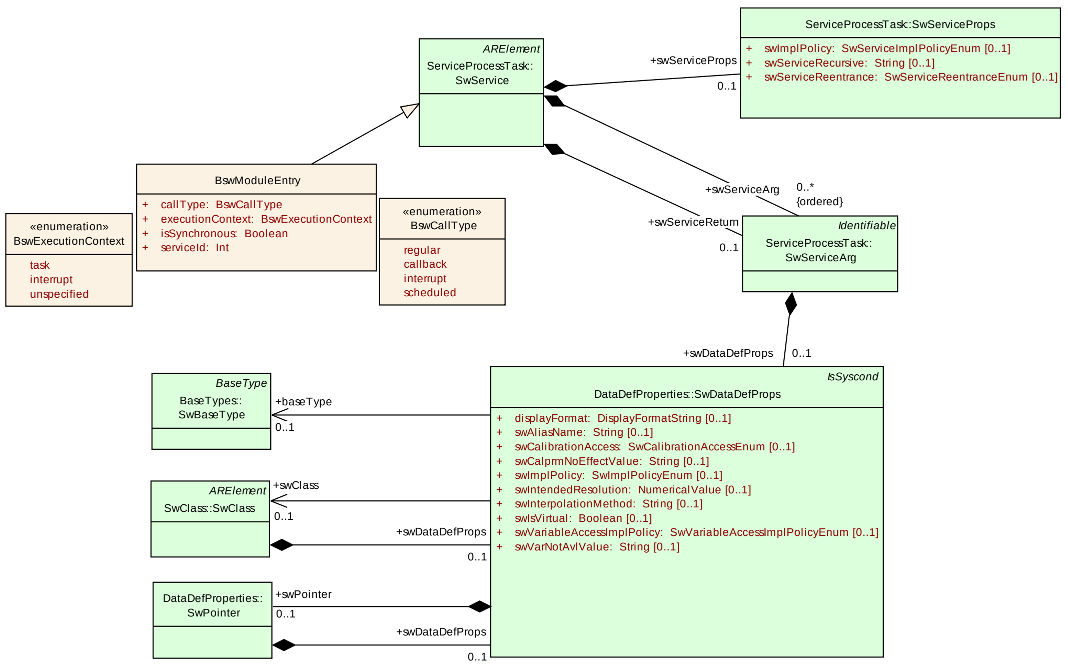
ServiceProcessTask::SwServiceProps
+ swImplPolicy: SwServiceImplPolicyEnum [0..1]
+ swServiceRecursive: String [0..1]
+ swServiceReentrance: SwServiceReentranceEnum [0..1]
AREIement
ServiceProcessTask::SwService
+swServiceProps
0..1
+swServiceArg
0..*
{ordered}
+swServiceReturn
0..1
Identifiable
ServiceProcessTask::SwServiceArg
BswModuleEntry
+ callType: BswCallType
+ executionContext: BswExecutionContext
+ isSynchronous: Boolean
+ serviceId: Int
«enumeration»
BswExecutionContext
task
interrupt
unspecified
«enumeration»
BswCallType
regular
callback
interrupt
scheduled
+swDataDefProps
0..1
IsSyscond
DataDefProperties::SwDataDefProps
+ displayFormat: DisplayFormatString [0..1]
+ swAliasName: String [0..1]
+ swCalibrationAccess: SwCalibrationAccessEnum [0..1]
+ swCalprmNoEffectValue: String [0..1]
+ swImplPolicy: SwImplPolicyEnum [0..1]
+ swIntendedResolution: NumericalValue [0..1]
+ swInterpolationMethod: String [0..1]
+ swIsVirtual: Boolean [0..1]
+ swVariableAccessImplPolicy: SwVariableAccessImplPolicyEnum [0..1]
+ swVarNotAvlValue: String [0..1]
BaseType
BaseTypes::SwBaseType
+baseType
0..1
AREIement
SwClass::SwClass
+swClass
0..1
+swDataDefProps
0..1
DataDefProperties::SwPointer
+swPointer
0..1
+swDataDefProps
0..1

Figure 5.1: Details of class BswModuleEntry

The class `BswModuleEntry` is a subclass of `SwService`. It contains the AUTOSAR specific attributes shown in the following table. The attribute `serviceId` is used to identify the C-function and thus is an important information for an AUTOSAR conformance test. For standardizes interfaces, the identifier is defined in the AUTOSAR Software Specification (SWS) of the module. In case the C-function prototype represented by the entry is not standardized, it still can be used optionally, but its value must differ from the standardized ones.

Class	**⟨⟨atpObject⟩⟩ BswModuleEntry**

[1] `SwService` and its attributes belong to the meta-model part re-engineered from MSR-SW. This subset of MSR-SW is defined by the AUTOSAR meta-model and the XML schema published as part of an AUTOSAR release. The relevant classes are shown as green in the class diagrams. See [5] and [9] for more explanation.

Package	M2::AUTOSARTemplates::BswModuleTemplate::BswInterfaces			
Class Desc.	This class represents a single API entry (C-function prototype) into the BSW module or cluster. The name of the C-function is equal to the short name of this element with one exception: In case of multiple instances of a module on the same CPU, special rules for ”infixes” apply, see description of class BswImplementation.			
Base Class(es)	SwService			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
callType	BswCall Type	1	aggregation	the type of call associated with this service
execution Context	BswEx-ecution Context	1	aggregation	Specifies the excution context which is required (in case of entries into this module) or guaranteed (in case of entries called from this module) for this service.
isSyn-chronous	Boolean	1	aggregation	true: This calls a synchronous service, i.e. the service is completed when the call returns. false: The service (on semantical level) may not be complete when the call returns.
serviceId	Integer	1	aggregation	Refers to the service identifier of the Standardized Interfaces of AUTOSAR basic software.

Table 5.1: BswModuleEntry

Enumeration	**BswExecutionContext**
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswInterfaces
Enum Desc.	specifies the excution context required or guaranteed for the call associated with this service
Literal	**Description**
interrupt	interrupt context always
unspecified	the execution context is not specified by the API
task	task context always

Enumeration	**BswCallType**
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswInterfaces
Enum Desc.	Denotes the mechanism by which the entry into the Bsw module shall be called.
Literal	**Description**
regular	regular API call
callback	callback (i.e. the caller specifies the signature)
interrupt	interrupt routine
scheduled	called by the scheduler

The attributes of `SwService` taken from MSR are used to describe the complete signature of a call. Not all attributes and classes shown in figure 5.1 will always be required. The most important ones are:

`SwServiceProps.swServiceReentrance` declares, whether the C-function guarantees reentrancy or not. For standardized BSW calls, this must match to the corresponding definition given in the BSW Specifications. Explanation from [9]: Reentrance enables or prohibits the service to be invoked again, before the service has finished and delivered a result. Valid values are:

- REENTRANCE

If this element is not defined the service cannot be invoked when it is executing.

Class SwServiceArg is used to declare the properties of the function arguments as well as of the return value.

SwServiceArg.swDataDefProps.baseType can be used to refer to the underlying basic data type (for more information on SwBaseType see [5]). Because it is attached via reference, it is a reusable type.

SwServiceArg.swDataDefProps.swClass can be used to relate the data definition to a reusable type definition, if it is not a basic data type (corresponds to a C typedef). Because SwClass is an ARElement and itself contains SwDataDefProps, it is possible to declare the required data properties as part of an SwClass and reuse it as a data type by referring to it.

SwServiceArg.swDataDefProps.swPointer is used to declare an argument or return type as a pointer. The class SwPointer in turn contains an element swDataDefProps which is used to describe the properties of the data to which the pointer refers. If the pointer refers to a reusable type, again baseType or swClass are used to describe this.

5.2 BSW Module Dependency

Figure 5.2 and the following table show the details of class BswModuleDependency. This class represents the expectations of one BSW module or cluster from another BSW module or cluster. It should be noted, that dependencies are not expressed by associations between instances of BswModuleDescription. In other words, the meta-model does not define compositions of BSW modules, which would be required to own such associations. This allows to maintain each BSWMD separately.

Class	⟨⟨**atpObject**⟩⟩ **BswModuleDependency**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswInterfaces			
Class Desc.	This class collects the dependencies of a BSW module or cluster on a certain other BSW module in an abstract way.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
expected Callback	BswMod-uleEntry	*	reference	Indicates a callback expected to be called from another module and implemented by this module.
required Entry	BswMod-uleEntry	*	reference	Indicates an entry into another modules which is required by this module.

service Item	Service Needs	*	aggregation	A single item (example: Nv block) for which the quality of a service is defined.
targetModuleId	Integer	1	aggregation	AUTOSAR identifier of the target module of which the dependencies are defined.

Table 5.2: BswModuleDependency

The set of `requiredEntries` and `expectedCallbacks` represent the interface imported from another module. Note that `requiredEntries` and `expectedCallbacks` do also include calls in interrupt context. An example could be as follows:

Consider we want to describe the callback-dependencies of an external EEPROM driver module from the (standardized) AUTOSAR SPI module. Consider the SPI driver offers an outgoing callback ”EndJobNotification” always called in interrupt context. To describe the dependency we would have to create an instance `BswModuleDescription.bswModuleDependency` and do the following assignments:
`bswModuleDependency.targetModuleId` = module identifier of the SPI driver
`bswModuleDependency.expectedCallback` = signature+name of ”‘EndJobNotification”
`bswModuleDependency.expectedCallback.executionContext` = ”interrupt” (i.e. the required context)
`bswModuleDependency.expectedCallback.callType` = ”callback”

The set of `serviceItems` repesents the abstract requirements which the module has on the configuration of AUTOSAR Services like NVRAM Manager or Watchdog Manager. The class `ServiceNeeds` is also used by the SWCT, because an AUTOSAR Service has to be configured per ECU for the needs of both BSW and SWCs. Therefore this class and its derivatives is defined in the `CommonStructure` package of the meta-model. These classes are shown in figure 5.3 and the following tables.

Note that the `ServiceNeeds` describes only the source data of an abstract dependency. How this is actually traced down to the configuration parameters is specified by the configuration parameters of the dependent modules itself. For a description of this mechnism see topic ”Derived Parameter Definition” in [10]. To get the complete picture, it should be noted that also other templates can define source data for dependencies, for example the configuration of the COM stack depends on information defined via the AUTOSAR System Template.

Class	⟨⟨**atpObject**⟩⟩ **ServiceNeeds**
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds
Class Desc.	This expresses the abstract needs that a Software Component or Basic Software Module has on the configuration of an AUTOSAR Service to which it will be connected. ”Abstract needs” means, that the model abstracts from the Configuration Paramaters of the underlying Basic Software.

Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.3: ServiceNeeds

Class	⟨⟨**atpObject**⟩⟩ **NvBlockNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of a single Nv block.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
nDataSets	Integer	1	aggregation	number of data sets to be provided by the NVRAM manager for this block
readonly	Boolean	1	aggregation	true: data of this block are write protected for normal operation (but protection can be disabled) false: no restriction
reliability	NvBlock Needs Reliability Enum	1	aggregation	Reliability against data loss on the non-volatile medium.
resistantTo Changed Sw	Boolean	1	aggregation	Defines whether an Nv block shall be treated resistant to configuration changes (true) or not (false). For details how to handle initialization in the latter case, refer to the NVRAM specification.
restoreAt Start	Boolean	1	aggregation	Defines whether the associated RAM mirror block shall be implictly restored during startup by the basic SW or not. Only relevant if a RAM mirror block (PerInstanceMemory) is associated with this port.
writeOnly Once	Boolean	1	aggregation	Defines write protection after first write: true: This block is prevented from being changed/erased or being replaced with the default ROM data after first initialization by the SWC. false: No such restriction.
writing Frequency	Integer	1	aggregation	Provides the amount of updates to this block from the application point of view. It has to be provided in ”number of write access per year”.
writing Priority	NvBlock Needs Writing Priority Enum	1	aggregation	Requires the priority of writing this block in case of concurrent requests to write other blocks.

Table 5.4: NvBlockNeeds

Class	⟨⟨**atpObject**⟩⟩ **SupervisedEntityNeeds**
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds

Class Desc.	Specifies the abstract needs on the configuration of the Watchdog Manager for one specific Supervised Entity (SE).			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
activateAt Start	Boolean	1	aggregation	true/false: supervision activation status of SE shall be enabled/disabled at start
enableDe-activation	Boolean	1	aggregation	true: SWC shall be allowed to deactivate supervision of this SE false: not
expected AliveCycle	Float	1	aggregation	Expected cycle time of alive trigger of this SE (in seconds)
maxAlive Cycle	Float	1	aggregation	Maximum cycle time of alive trigger of this SE (in seconds)
minAlive Cycle	Float	1	aggregation	Minimum cycle time of alive trigger of this SE (in seconds)
tolerated FailedCy-cles	Integer	1	aggregation	Number of consecutive failed alive cycles for this SE which shall be tolerated until the supervision status of the SE is set to EXPIRED (see WdgM documentation for details). Note that this has to be recalculated w.r.t. the WdgMs own cycle time for ECU configuration.

Table 5.5: SupervisedEntityNeeds

Class	⟨⟨**atpObject**⟩⟩ **ComMgrUserNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Communication Manager for one ”user”.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
maxComm Mode	MaxComm Mode Enum	1	aggregation	Maximum communication mode requested by this ComM user

Table 5.6: ComMgrUserNeeds

Class	⟨⟨**atpObject**⟩⟩ **EcuStateMgrUserNeeds**
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds
Class Desc.	Specifies the abstract needs on the configuration of the ECU State Manager for one ”user”. This class currently contains no attributes. Its name can be regarded as a symbol identifying the user from the viewpoint of the component or module which owns this class.
Base Class(es)	ServiceNeeds

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.7: EcuStateMgrUserNeeds

Class	⟨⟨**atpObject**⟩⟩ **DiagnosticEventNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Diagnostic Event Manager for one diagnostic event. Its name can be regarded as a symbol identifying the diagnostic event from the viewpoint of the component or module which owns this class.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.8: DiagnosticEventNeeds

Class	⟨⟨**atpObject**⟩⟩ **FunctionInhibitionNeeds**			
Package	M2::AUTOSARTemplates::CommonStructure::ServiceNeeds			
Class Desc.	Specifies the abstract needs on the configuration of the Function Inhibition Manager for one Function Identifier (FID). This class currently contains no attributes. Its name can be regarded as a symbol identifying the FID from the viewpoint of the component or module which owns this class.			
Base Class(es)	ServiceNeeds			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 5.9: FunctionInhibitionNeeds

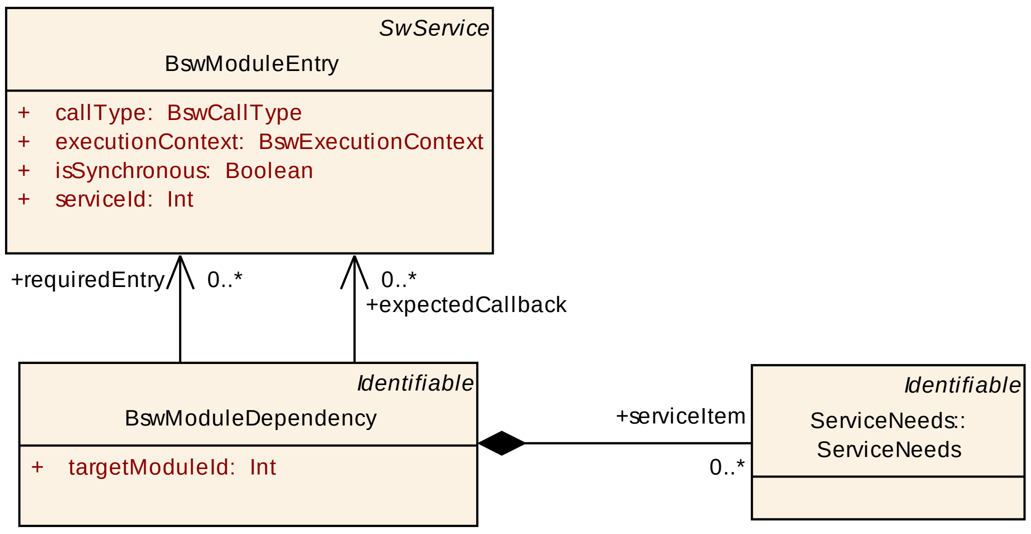

Figure 5.2: Details of class BswModuleDependency

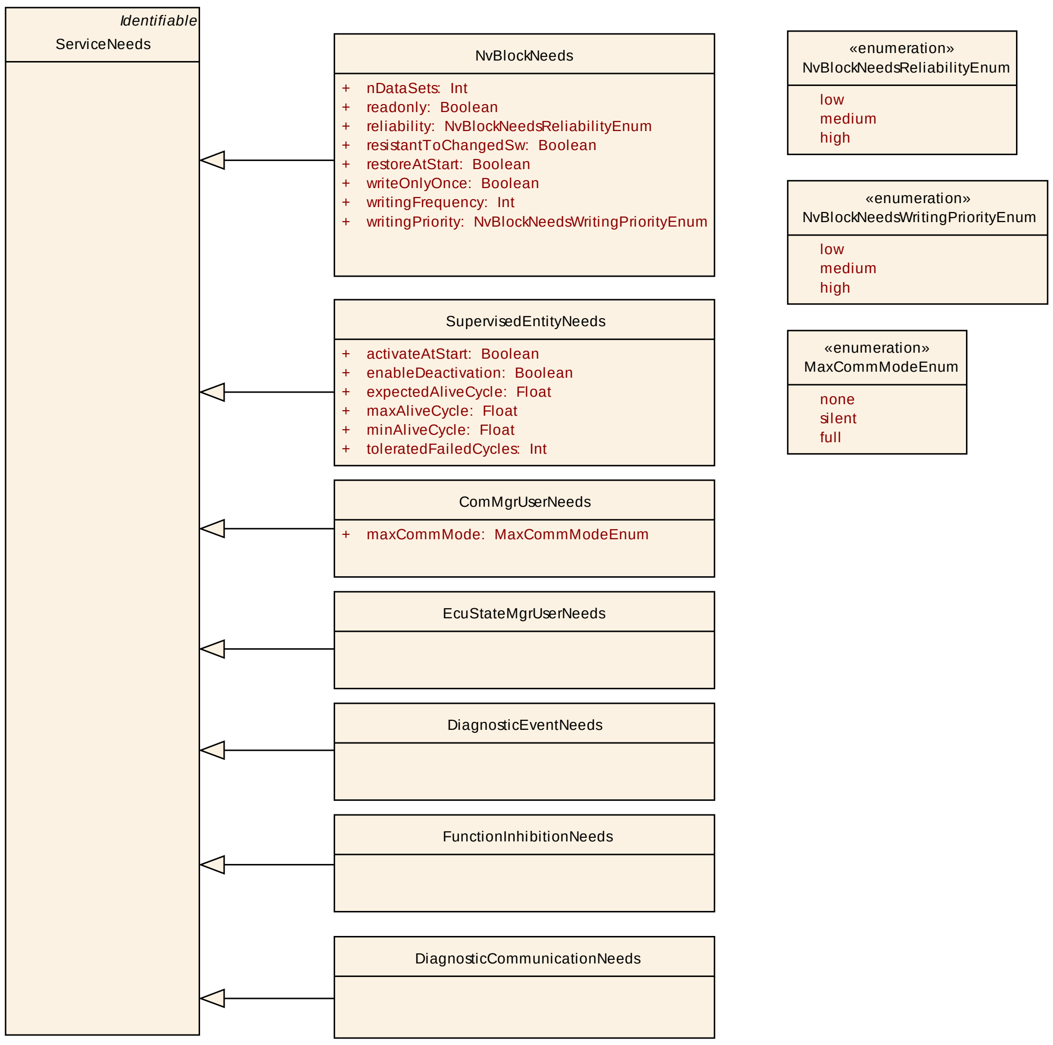
Identifiable
ServiceNeeds
NvBlockNeeds
+ nDataSets: Int
+ readonly: Boolean
+ reliability: NvBlockNeedsReliabilityEnum
+ resistantToChangedSw: Boolean
+ restoreAtStart: Boolean
+ writeOnlyOnce: Boolean
+ writingFrequency: Int
+ writingPriority: NvBlockNeedsWritingPriorityEnum
SupervisedEntityNeeds
+ activateAtStart: Boolean
+ enableDeactivation: Boolean
+ expectedAliveCycle: Float
+ maxAliveCycle: Float
+ minAliveCycle: Float
+ toleratedFailedCycles: Int
ComMgrUserNeeds
+ maxCommMode: MaxCommModeEnum
EcuStateMgrUserNeeds
DiagnosticEventNeeds
FunctionInhibitionNeeds
DiagnosticCommunicationNeeds
«enumeration» NvBlockNeedsReliabilityEnum
low
medium
high
«enumeration» NvBlockNeedsWritingPriorityEnum
low
medium
high
«enumeration» MaxCommModeEnum
none
silent
full

Figure 5.3: class `ServiceNeeds` from `CommonStructure`

6 BSW Behavior

6.1 BSW Behavior Overview

Figure 6.1 and the following class table show the attributes and description of class `BswBehavior`. Note that `BswBehavior.entity` and `BswBehavior.exclusiveArea` describe properties of the actual code whereas `BswBehavior.event` can be seen as a requirement to the BSW Scheduler to implement such an event.

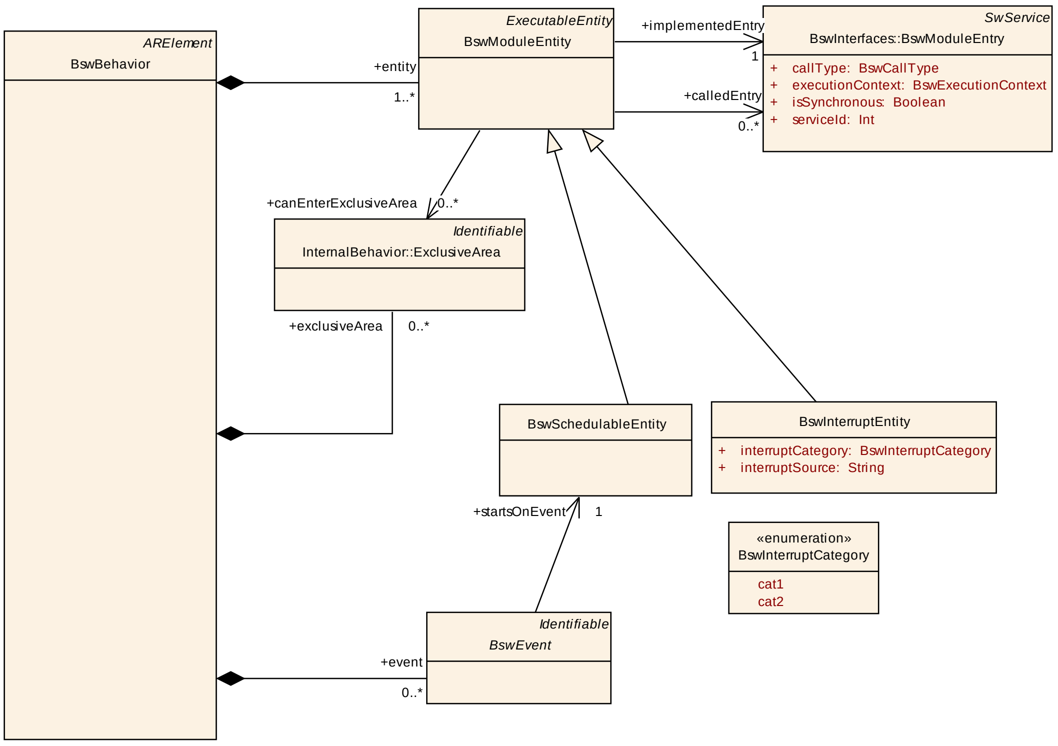

Figure 6.1: Overview of class BswModuleBehavior

Class	⟨⟨**atpObject**⟩⟩ **BswBehavior**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	Specifies the behavior of a BSW module or a BSW cluster w.r.t. the code entities visible by the BSW Scheduler. It is possible to have several different BswBehaviors referring to the same BswModuleDescription.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
entity	BswModuleEntity	1..*	aggregation	A code entity for which the behavior is described
event	BswEvent	*	aggregation	An event required by this module behavior.
exclusive Area	Exclusive Area	*	aggregation	This specifies an ExclusiveArea for this BswBehavior. The exclusiveArea is local to the module or module cluster.

module	BswModuleDescription	1	reference	The module specification fulfilled by this behavior.

Table 6.1: BswBehavior

6.2 BSW Module Entity

The next class tables shows the attributes of `BswModuleEntity` and its specializations for scheduled and interrupt entities. These attributes are mainly required to configure the BSW Scheduler.

It is important to understand the difference between `BswModuleEntity` and `BswModuleEntry`: The first one describes properties of a code fragment wheras the second one describes only the interface used to invoke a code fragment.

The attribute `BswModuleEntity.calledEntry` allows to declare which entry of another module (or the same module) is called by this code entity. Note that this is not a mandatory information in order to be able to integrate a module, but it is a very important information if an integrator wants to analyze a call chain among several modules in order to setup a proper scheduling. It is further important to note that this attribute contains additional information in comparison to `BswModuleDescription.bswModuleDependency`, because the latter only denotes the dependencies between the module interfaces whereas `calledEntry` shows from which code fragment a call is invoked.

Class	**⟨⟨atpObject⟩⟩ BswModuleEntity**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	Specifies the smallest code fragment which can be described for a BSW module or cluster within AUTOSAR.			
Base Class(es)	ExecutableEntity			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
activation Point	BswSporadicEvent	*	reference	The module entity can activate this event.
calledEntry	BswModuleEntry	*	reference	The entry of another (or the same) BSW module which is called by this entry (usually via C function call). This information allows to set up a model of call chains.
canEnter Exclusive Area	Exclusive Area	*	reference	The BswModuleEntity can enter/leave the referenced exclusive area through explicit API calls.
cancellation Point	BswSporadicEvent	*	reference	The module entity can cancel the activation of the event (this only makes sense, if the event has a non-zero delay time).
implemented Entry	BswModuleEntry	1	reference	The entry which is implemented by this module entity.

Table 6.2: BswModuleEntity

Class	⟨⟨**atpObject**⟩⟩ **BswSchedulableEntity**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	BSW module entity, which is designed for control by the BSW Scheduler. It implements a so-called "main" function.			
Base Class(es)	BswModuleEntity			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***

Table 6.3: BswSchedulableEntity

Class	⟨⟨**atpObject**⟩⟩ **BswInterruptEntity**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	BSW module entity, which is designed to be triggered by an interrupt.			
Base Class(es)	BswModuleEntity			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
interrupt Category	BswIn-terrupt Category	1	aggregation	Category of the interrupt
interrupt Source	String	1	aggregation	Allows a textual documentation of the intended interrupt source.

Table 6.4: BswInterruptEntity

Enumeration	**BswInterruptCategory**
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior
Enum Desc.	category of the interrupt service
Literal	***Description***
cat1	Cat1 interrupt routines are not controlled by the OS and are only allowed to make a very limited selection of OS calls to enable and disable all interrupts. The BswInterruptEntity is implemented by the interrupt service routine, which is directly called from the interrupt vector (not via the OS).
cat2	Cat2 interrupt routines are controlled by the OS and they are allowed to make OS calls. The BswInterruptEntity is implemented by the interrupt handler, which is called from the OS.

The class `ExclusiveArea` is not specific for the Basic Software, it is imported from the `CommonStructure` package of the meta-model:

Class	⟨⟨**atpObject**⟩⟩ **ExclusiveArea**
Package	M2::AUTOSARTemplates::CommonStructure::InternalBehavior
Class Desc.	Prevents an executable entity running in the area from being preempted.

Base Class(es)	Identifiable			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**

Table 6.5: ExclusiveArea

6.3 BSW Event

Figure 6.2 and the following tables show the inheritance and attributes of `BswEvent` and its relation to `BswModuleEntity`. Note the difference in the activation of sporadic and cyclic events: A `BswModuleEntity` can trigger or cancel a `BswSporadicEvent` (of the same module) with the help of an API generated by the BSW Scheduler, whereas a `BswCyclicEvent` is directly triggered by the BswScheduler (via the OS timer). Further information can be found in [11].

Note that `BswCyclicEvents` does not include recurring events with variable cycle time or from an external trigger (e.g. crank-shaft). The input information required to configure these kind of recurring events in the BSW Scheduler is curently not specified in [11].

Despite of that, external events can directly trigger a `BswInterruptEntity` by the means of an interrupt.

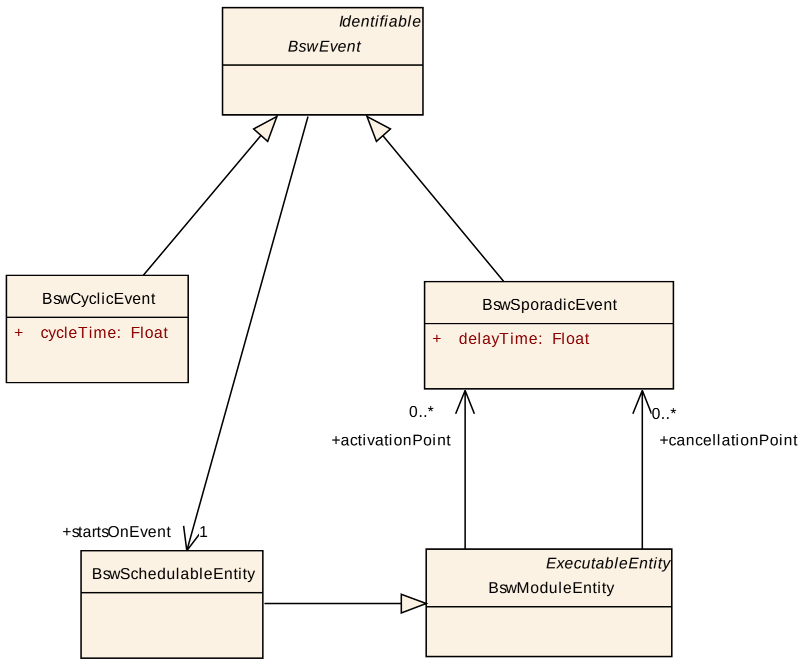

Figure 6.2: BswEvents and their relations to BswModuleEntities

Class	⟨⟨**atpObject**⟩⟩ **BswEvent (abstract)**

Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	Defines an event which is used to trigger a schedulable entity of this BSW module or cluster. The event is local to the BSW module or cluster. The short name of the class instance is intended as an input to configure the required API of the BSW Scheduler.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
startsOn Event	Bsw Schedula-bleEntity	1	reference	This entity is started by the event.

Table 6.6: BswEvent

Class	⟨⟨**atpObject**⟩⟩ **BswSporadicEvent**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	A BSW event, which can happen sporadically. The event is activated/cancelled by explicit calls from the module to the BSW Scheduler. There a two purposes for such an event: - cause a context switch, e.g. from an ISR context into a task context - implement a time delay			
Base Class(es)	BswEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
delayTime	Float	1	aggregation	Requirement for the delay time (in seconds) after which this event is triggered. The delay is counted from the activation of this event by a BswModuleEntitiy until the actual triggering of another BswModuleEntity associated by the event.

Table 6.7: BswSporadicEvent

Class	⟨⟨**atpObject**⟩⟩ **BswCyclicEvent**			
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswBehavior			
Class Desc.	A cyclically recurring BSW event. The cyclic activity has to be implemented by the BSW Scheduler.			
Base Class(es)	BswEvent			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
cycleTime	Float	1	aggregation	Requirement for the cycle time (in seconds) by which this event is triggered.

Table 6.8: BswCyclicEvent

7 BSW Implementation

The template elements to be used by the developer in order to document the actual implementation of a BSW module or cluster are very similar to what is needed for the same purpose in the case of SW-Cs. Therefore it is based on the `CommonStructure` part or the meta-model. This includes also the documentation of resource consumption. The generic classes of the meta-model used to document implementation and resource consumption are described in chapter 8 and chapter 9 in this document.

There are however some special features in describing the implementation of BSW. This is the purpose of class `BswImplementation` (see Figure 7.1 and the following class table).

The AUTOSAR version information (minor/major/patch) is specific for AUTOSAR BSW and specified for the `BswImplementation`.

Note that in case a BSW module is used in multiple implementations on the same ECU (which means, that the code has to be there multiple times with the exception of shared libraries), for each module implementation there has to be a separate instance of `BswImplementation`. This allows to define name expansions required for global symbols via the attribute `vendorApiInfix`.

The attribute `requiredHW` allows to document special hardware dependencies of a BSW module or cluster in addition to what can be expressed by the generic attributes `Implementation.processor` and `Implementation.resourceConsumption` (see also chapter 9). The intended use case of this attribute is to document hardware dependencies of BSW modules or clusters which cover firmware layers, namely MCAL, ECU abstraction or Complex Drivers.

Finally it is possible to specify vendor specific configuration parameter definitions and predefined or recommended configuration parameter values within the scope of BSW implementation. This is expressed by the associations from `BswImplementation` to `ModuleDef` and to `ModuleConfiguration` which are specified in the ECU Configuration Specification document [10]. Note that different implementations of the same `BswModuleDescription` can have different parameter values and different sets of vendor specific configuration parameters. Of course it is also possible that different implementations of the same module refer to the same configuration parameter definitions resp. to the same predefined or recommended configuration parameter values.

In addition the `ModuleConfiguration` from the ECU Configuration Template can refer to the `BswImplementation` for which it defines the configuration parameters. This relation is intended to be used by the integrator or tester to indicate for which BswImplementation an actual ECU configuration has been set up.

Class	⟨⟨**atpObject**⟩⟩ **BswImplementation**
Package	M2::AUTOSARTemplates::BswModuleTemplate::BswImplementation
Class Desc.	Contains the implementation specific information in addition to the generic specification (BswModuleDescription and BswBehavior). It is possible to have several different BswImplementations referring to the same BswBehavior.

Base Class(es)	Implementation , ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
arMajor Version	Integer	1	aggregation	Major version number of AUTOSAR specification on which this implementation is based on.
arMinor Version	Integer	1	aggregation	Minor version number of AUTOSAR specification on which this implementation is based on.
arPatch Version	Integer	1	aggregation	Patch level version number of AUTOSAR specification on which this implementation is based on.
behavior	BswBehav-ior	1	reference	The behavior of this implementation.
preconfigured Configura-tion	Module Configura-tion	0..1	reference	Reference to the module configuration that contains preconfigured (i.e. fixed) configuration parameters.
recommended Configura-tion	Module Configura-tion	0..1	reference	Reference to the recommended configuration for this module.
requiredH W	HWEle-ment	*	reference	Hardware resource required by this BswImplementation
vendorApi Infix	String	1	aggregation	In driver modules which can be instantiated several times on a single ECU, BSW00347 requires that the name of APIs is extended by the VendorId and a vendor specific name. This parameter is used to specify the vendor specific name. In total, the implementation specific name is generated as follows: <ModuleName>_<VendorId>_<VendorApiInfix><Api name from SWS>. E.g. assuming that the VendorId of the implementor is 123 and the implementer chose a VendorApiInfix of "v11r456" a api name Can_Write defined in the SWS will translate to Can_123_v11r456Write. This attribute is mandatory for all modules with upper multiplicity > 1. It shall not be used for modules with upper multiplicity =1.
vendor Specific ModuleDef	ModuleDef	1	reference	Reference to the Vendor Specific ModuleDef used in this BSW module description.

Table 7.1: BswImplementation

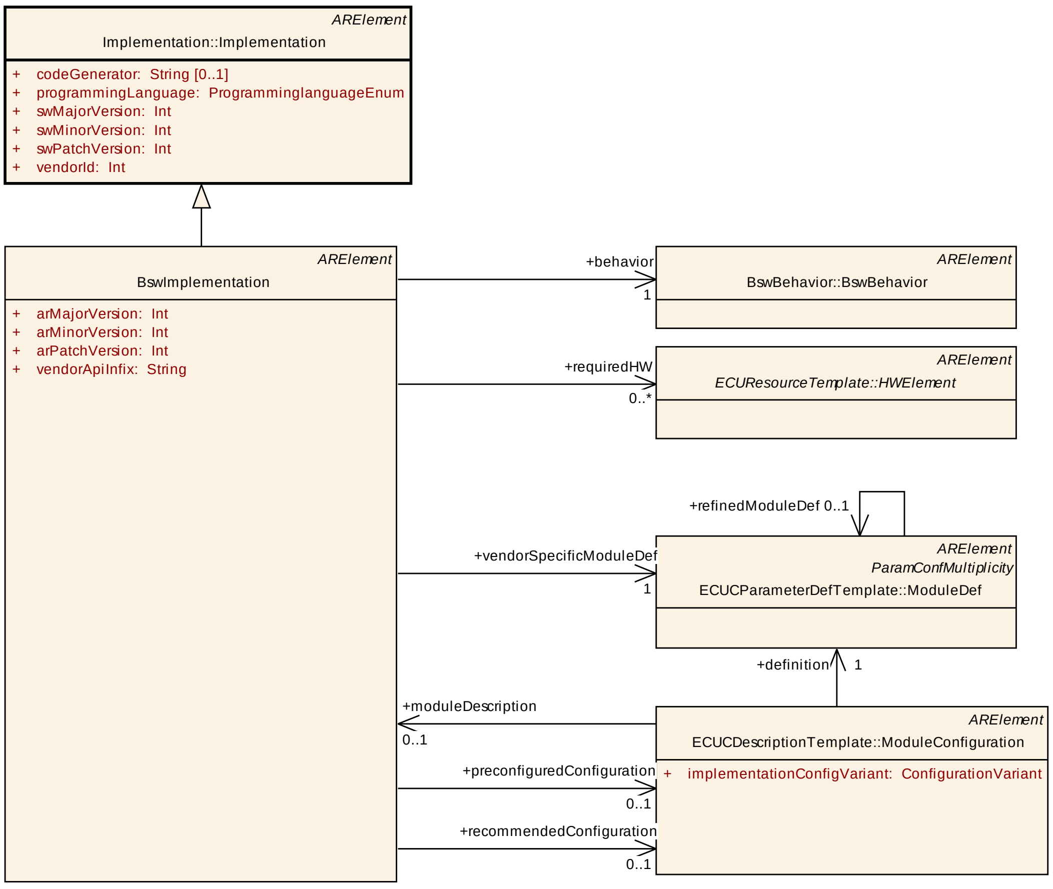

Figure 7.1: Overview of class BswImplementation

8 Implementation

8.1 Introduction

This chapter explains, how the implementation details of AUTOSAR software components and Basic Software can be described. While AUTOSAR contains various component types, only atomic software components and Basic Software Modules possess an `Implementation`. In the meta model this means that `Implementation` can be provided for `AtomicSoftwareComponentType` or derived classes and `BswModuleDescription` only. On the other hand, compositions simply structure and encapsulate their contained components in a hierarchical manner, without adding any implementation relevant behavior or functionality. So they cannot be implemented directly. Instead, the leaf components in such a composition tree, which by definition are again atomic, are implemented.

8.2 Implementation Description Overview

The `Implementation` class shown in Figure 8.1 serves the following main purposes:

- provide information about the resource consumption (chapter 9)
- link to code (source code, object code) (section 8.5)
- specify required libraries (section 8.6)
- specify the build environment (section 8.7)

As the figure shows, `Implementation` is derived from `ARElement`, i.e. it may be shipped as a separate engineering artifact, e.g. independent of the description of interfaces, ports and the component type.

The following table lists all attributes shown in Figure 8.1, thereby explaining the meaning of the remaining simple assertions and requirements of class `Implementation`.

Class	⟨⟨**atpObject**⟩⟩ **Implementation**			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Class Desc.	Description of an implementation a single software component or module.			
Base Class(es)	ARElement			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
codeDescriptor	Code	1..*	aggregation	Specifies the provided implementation code.
codeGenerator	String	0..1	aggregation	Optional: code generator used.
compiler	Compiler	*	aggregation	Specifies the compiler for which this implementation has been released

implementation Dependency	Dependency	*	aggregation	Specifies details on dependent software, modules or libraries.
linker	Linker	*	aggregation	Specifies the linker for which this implementation has been released.
processor	Processing Unit	*	reference	The processor the implementation is compatible with.
programming Language	Programminglanguage Enum	1	aggregation	Programming language the implementation was created in.
resource Consumption	Resource Consumption	1	aggregation	All static and dynamic resources for each implementation are described within the ResourceConsumption class.
swMajor Version	Integer	1	aggregation	Major version number of this implementation. The numbering is vendor specific.
swMinor Version	Integer	1	aggregation	Minor version number of this implementation. The numbering is vendor specific.
swPatch Version	Integer	1	aggregation	Patch version number of this implementation. The numbering is vendor specific.
vendorId	Integer	1	aggregation	Vendor ID of this Implementation according to the AUTOSAR vendor list

Table 8.1: Implementation

8.3 Assertions and Requirements

For some of the attributes mentioned below it is ambiguous whether they describe a requirement on the target environment or whether they are assertions made by the particular component implementation. The `Implementation` description's `Compiler` attribute is an example for this: does it describe a requirement for source code to be compiled with the named compiler, or is this simply information which compiler was used in the process of creating an object file? The simple answer is: if possible, this is derived from the context. Otherwise the attribute needs to have proper documentation. For the `Compiler` example just mentioned, the situation is straightforward: for source code, the attribute describes a requirement, for object code it is historic information. The same needs to be applied to all attributes in this section.

8.4 Implementation of a Software Component

Probably the most important information in `Implementation` is which Atomic Software Component or BSW Module is actually implemented. At first glance, this

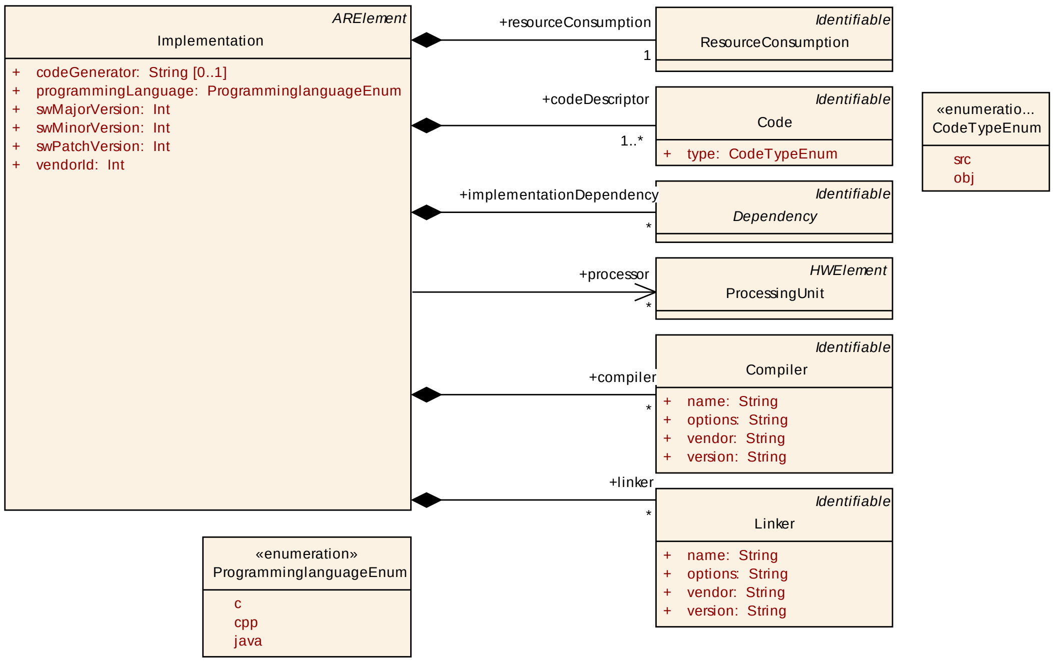

Figure 8.1: Overview of implementation description

link seems to be missing in the overview in Figure 8.1. However, implementations are actually given for a particular component behavior, specified through the class `InternalBehavior` respectively `BswBehavior`. The contents of such a behavior are not of interest here, but as Figure 8.2 shows, it in turn is associated with a single `AtomicSoftwareComponentType` or `BswModuleDescription`.

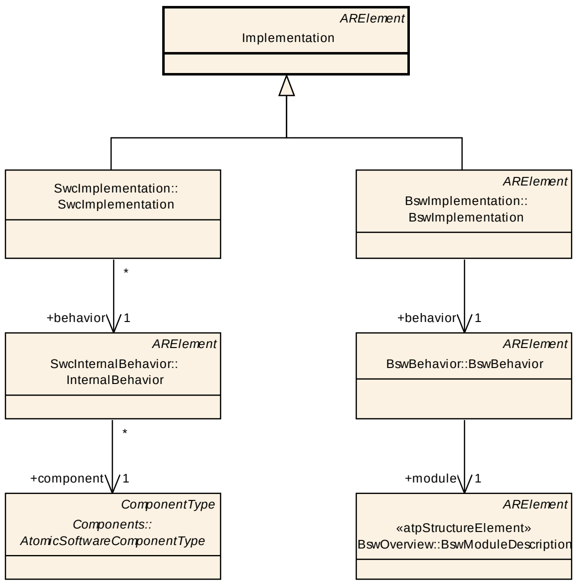

Figure 8.2: An implementation is associated with a single software component

8.5 Linking to Code

When a component is released the descriptions are accompanied by actual implementation code. This code can come in different ways: source code in C, C++ or Java, object code or even executable code[1].

Figure 8.3 shows how an `Implementation` is linked to `Code` files. For each available form of component code a `Code` element is used. If for instance a component implementation is given as source code only, then the respective `Implementation` would contain exactly one `Code`, whose `type` attribute would have been set to `src`. For each code description, all relevant files are then referenced in form of a standard URL. For relative URLs the path will start at the containing XML file.

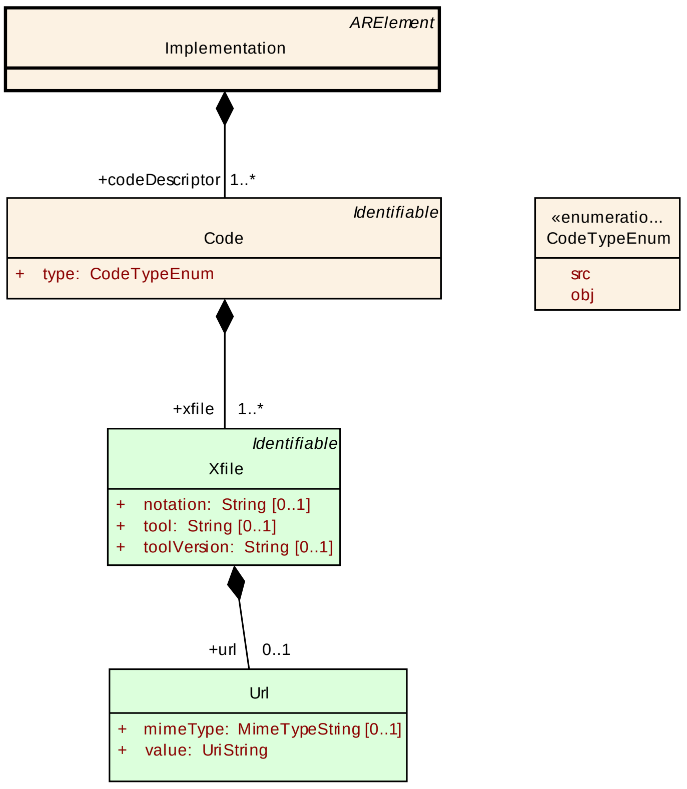

Figure 8.3: An `Implementation` references the code files through the `Code` class

Class	⟨⟨**atpObject**⟩⟩ **Code**			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Class Desc.	A generic code descriptor.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
type	CodeType Enum	1	aggregation	The type of described code.

[1] How such linked code would be embedded in the ECU is not in scope of the template specification. If required further attributes need to be added at a later time to support such a process.

xfile	Xfile	1..*	aggregation	The files belonging to this code descriptor.

Table 8.2: Code

Class	⟨⟨**atpObject**⟩⟩ **Xfile**			
Package	M2::AUTOSARTemplates::GenericStructure::CommonPatterns::InlineTextModel::Inlines			
Class Desc.	Use <xfile> , to reference an external file.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
tool	String	0..1	aggregation	This element describes the tool which was used to generate the corresponding <xfile> . Kept as a string.
toolVersion	String	0..1	aggregation	This element describes the tool version which was used to generate the corresponding <xfile> . Kept as a string.
url	Url	0..1	aggregation	This element specifies the Uniform Resource Locator (URL) of the context contained in the <url> element.

Table 8.3: Xfile

Class	⟨⟨**atpObject**⟩⟩ **Url**			
Package	M2::AUTOSARTemplates::GenericStructure::CommonPatterns::InlineTextModel::Inlines			
Class Desc.	This element specifies the Uniform Resource Locator (URL) of the context contained in the <url> element.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
mimeType	MimeTypeString	0..1	aggregation	this denotes the mime type of the resource located by the url.
value	UriString	1	aggregation	This is the url itself

Table 8.4: Url

8.6 Dependencies

By specifying dependencies an implementation can depend on certain other artifacts or model features. The concrete meaning of this is detailed by particular kind of dependency, as shown in Figure 8.4.

Class	⟨⟨**atpObject**⟩⟩ **Dependency (abstract)**
Package	M2::AUTOSARTemplates::CommonStructure::Implementation

Class Desc.	General dependency, typically on the existence of another artifact.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
usage	Dependency Usage Enum	1..*	aggregation	Specification during for which process step(s) this dependency is required.

Table 8.5: Dependency

An implementation can generally depend on files. Such files could for example be required header files or configuration files. The URL points to the place where the files are expected, or simply contains the name of the file, in case the path is not relevant. For libraries, like e.g. a `math.lib`, a minimum and maximum version number can be specified, therefore trying to ensure compatibility. Note that the specification of version numbers is a meta-information about certain artifacts, for which a more general solution may be found in the future (e.g. as part of a catalog description). So the current solution has to be seen as a first and rough approach only.

Class	⟨⟨**atpObject**⟩⟩ **DependencyOnFile**			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Class Desc.	Dependency on the existence of a certain file.			
Base Class(es)	Dependency			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
xfile	Xfile	1	aggregation	The specified file needs to exist.

Table 8.6: DependencyOnFile

Class	⟨⟨**atpObject**⟩⟩ **DependencyOnLibrary**			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Class Desc.	A specific file dependency: without the library that implementation cannot be used (compiled, linked, executed, ...).			
Base Class(es)	DependencyOnFile			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
maxVersion	String	1	aggregation	Maximum version compatible with implementation. If not set, there is limitation on the upper version.
minVersion	String	1	aggregation	Minimum version compatible with implementation.

Table 8.7: DependencyOnLibrary

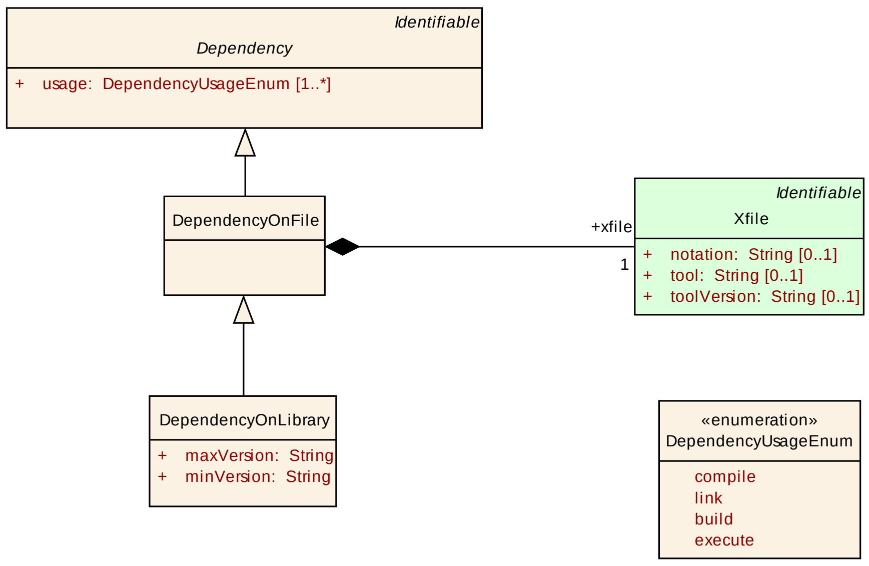

Figure 8.4: Dependencies of an `Implementation`

8.7 Compiler

For the specification of the used (or to be used) compiler the `Compiler` element shall be used:

Class	⟨⟨**atpObject**⟩⟩ **Compiler**			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Class Desc.	Specifies the compiler attributes. In case of source code this specifies requirements how the compiler shall be invoked. In case of object code this documents the used compiler settings.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
name	String	1	aggregation	Compiler name (like gcc).
options	String	1	aggregation	Specifies the compiler options.
vendor	String	1	aggregation	Vendor of compiler.
version	String	1	aggregation	Exact version of compiler executable.

Table 8.8: Compiler

8.8 Linker

For the specification of the to be used linker the `Linker` element shall be used:

Class	⟨⟨**atpObject**⟩⟩ **Linker**			
Package	M2::AUTOSARTemplates::CommonStructure::Implementation			
Class Desc.	Specifies the linker attributes used to decribe how the linker shall be invoked.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
name	String	1	aggregation	Linker name.
options	String	1	aggregation	Specifies the linker options.
vendor	String	1	aggregation	Vendor of linker.
version	String	1	aggregation	Exact version of linker executable.

Table 8.9: Linker

9 ResourceConsumption

AUTOSAR software needs to be mapped on ECUs at some point during the development. Application software components can be basically mapped to any ECU available within the car. The mapping freedom is limited by the *System Constraints* [12] and the available resources on each ECU. BSW Modules are present in each ECU which provides the corresponding service. The `ResourceConsumption` element provides information about the needed resources concerning memory and execution time for each `SwcImplementation` or `BswImplementation`.

9.1 Static and Dynamic Resources

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 a good example, where different tasks 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).

9.2 Resource consumption overview

In Figure 9.1, the meta-model of the `ResourceConsumption` description is depicted. The `ResourceConsumption` is attached to an `Implementation`. For each `Implementation`, there can be one `ResourceConsumption` description.

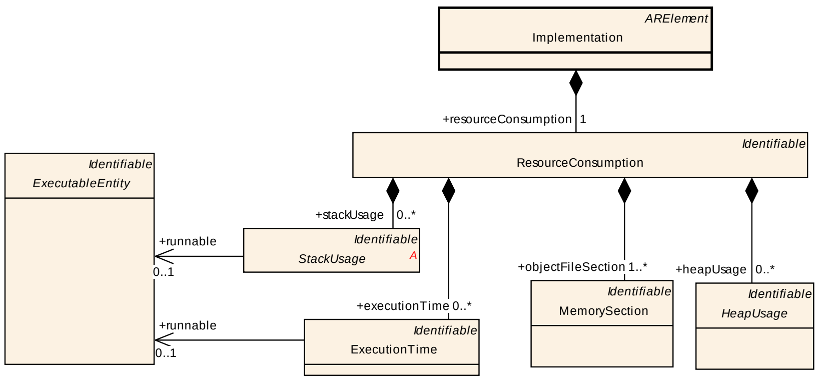

Figure 9.1: Resource consumption overview

As depicted by Figure 9.1, all resources are described within the `ResourceConsumption` meta-class.

`ExecutionTime` (section 9.5) and `StackUsage` (section 9.4.2) are used to provide information on the implementation specific resource usage of the `ExecutableEntity` defined in the `InternalBehavior` of SW-Component respectively in the `BswBehavior` of BSW Module.

`MemorySection` (section 9.3.2) documents the resources needed to load the object file containing the implementation on the ECU.

`HeapUsage` (section 9.4.3) describes the dynamic memory usage of the software.

Class	⟨⟨**atpObject**⟩⟩ **ResourceConsumption**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption			
Class Desc.	Description of consumed resources by one implementation of a software.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
executionTime	ExecutionTime	*	aggregation	Collection of the execution time descriptions for the runnable entities of this implementation.
heapUsage	HeapUsage	*	aggregation	Collection of the heap memory allocated by this implementation.
objectFileSection	MemorySection	1..*	aggregation	Provides additional information to the sections of the object-file containing the implementation of the SW-Component
stackUsage	StackUsage	*	aggregation	Collection of the stack memory usage for each runnable entity of this implementation.

Table 9.1: ResourceConsumption

9.3 Static Memory Needs

9.3.1 General

This sub-chapter describes how the static memory needs for the `Implementation` are described. This includes all memory needs of software for code or data both at the class and at the instance level except for:

- stack space needed in the task that activates an `ExecutableEntity` of the implementation (see chapter 9.4.2)
- dynamic heap-behavior of the software (in case the software uses `malloc/free` to get/free buffers from the heap, see chapter 9.4.3[1])

9.3.2 Memory Sections

Memory will be needed to load the object-file containing an implementation of the software on an ECU. In which kind of memory the code and data of the software have to be allocated has to be defined in the source code of the software according to the *Specification of Memory Mapping* (see [13]).

To support the integration and configuration of the software component the used memory sections and their attributes have to be described using the `MemorySection` element from figure 9.2.

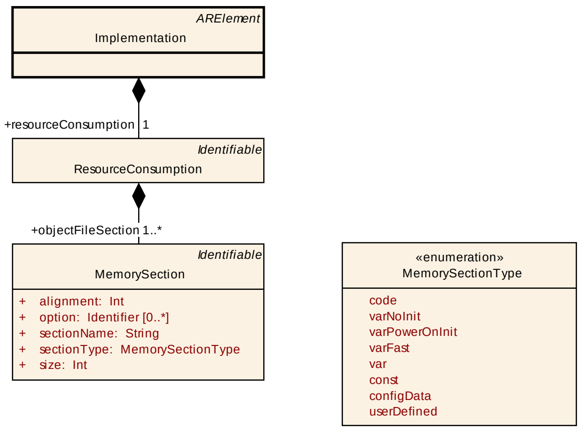

Figure 9.2: Meta-model related to the `MemorySection`

The attributes of `MemorySection` are shown below:

Class	⟨⟨**atpObject**⟩⟩ **MemorySection**
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::Memory SectionUsage

[1] This is often problematic in embedded and real-time systems: most software will only need static memory blocks and stack-size but will not require dynamic memory allocation

Class Desc.	The MemorySection provides description of the Memory Sections used in the Implementation.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
alignment	Integer	1	aggregation	The alignment (typically 1, 2, 4,...)
option	Identifier	*	aggregation	This attribute introduces the ability to specify further intended properties of this MemorySection. The following value is standardized (to be used for code sections only): * INLINE - The code section is declared with the compiler abstraction macro INLINE. The expansion of INLINE depends on the compiler specific implementation of the macro. Depending on this, the code section either corresponds to an actual section in memory or is put into the section of the caller. See AUTOSAR_SWS_CompilerAbstraction for more details.
section Name	String	1	aggregation	This is the name of the section in the Implementation.
section Type	Memory Section Type	1	aggregation	Memory section type of the described MemorySection.
size	Integer	1	aggregation	The size in bytes of the section.
swAddr Method	SwAddr Method	*	reference	This assocation indicates all objects (e.g. calibration parameters, data element prototypes) being assigned to this SwAddrMethod shall be placed in this memory Section.

Table 9.2: MemorySection

Enumeration	**MemorySectionType**
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::Memory SectionUsage
Enum Desc.	Enumeration to specify the different types of memory classes available in the AUTOSAR Memory Mapping.
Literal	***Description***
code	To be used for mapping code to application block, boot block, external flash etc.
varNoInit	To be used for all global or static variables that are never initialized.
varPowerOn Init	To be used for all global or static variables that are initialized only after power on reset.

varFast	To be used for all global or static variables that have at least one of the following properties: - accessed bitwise - frequently used - high number of accesses in source code Some platforms allow the use of bit instructions for variables located in this specific RAM area as well as shorter addressing instructions. This saves code and runtime.
var	To be used for global or static variables that are initialized after every reset (the normal case).
const	To be used for global or static constants.
configData	Constants with attributes that show that they reside in one segment for module configuration.
userDefined	No specific categorization of sectionType possible.

The attribute `sectionType` is used to define which default section this memory segment shall be mapped to. Since all of the provided `MemorySectionType` (except for `userDefined` do match to a section form the *Specification of Memory Mapping* (see [13]), this information can be used to create a default mapping of each `MemorySection` to some ECU memory segemnt during ECU configuration.

In case the `userDefined` `sectionType` is used additional documentation is needed to support the integrator in selecting the proper memory segment from the ECU.

9.4 Dynamic Memory Needs

9.4.1 General

The dynamic memory is mainly divided into two categories, the stack and the heap. While the stack is almost always used in embedded software, the heap is avoided as much as possible due to the complexity of its implementation and segmentation issues. The dynamic memory consumption of software has a much different quality than the static memory consumption. The amount of the static memory consumption can be retrieved from the compiler and is only dependent on the compiler and processor used as well as on the number of instances.

Dynamic memory consumption is heavily dependent on the actual code being executed, which is dependent on the state of the software and the parameters. With the introduction of recursive concepts the uncertainty is even higher. Therefore the approach for dynamic memory consumption is far more related to the description of the execution time introduced in section 9.5.

9.4.2 Stack

The stack is an area in memory that is used to store temporary information like parameters and local variables of function calls. Therefore the stack usage is highly dependent on the calling hierarchy and the nesting level of function calls. The stack is organized in a LIFO (last in first out) manner. So each time a function is called the necessary stack memory is occupied. After leaving the function also the associated memory area is freed again and can be used for the next function call. Therefore segmentation is not a problem for a stack. Only the available amount of stack memory is relevant from the software point of view.

Different mechanisms can be used to describe the stack memory needs of software. Needed stack size can either be *calculated*, *measured* or *estimated*. This is shown in Figure 9.3.

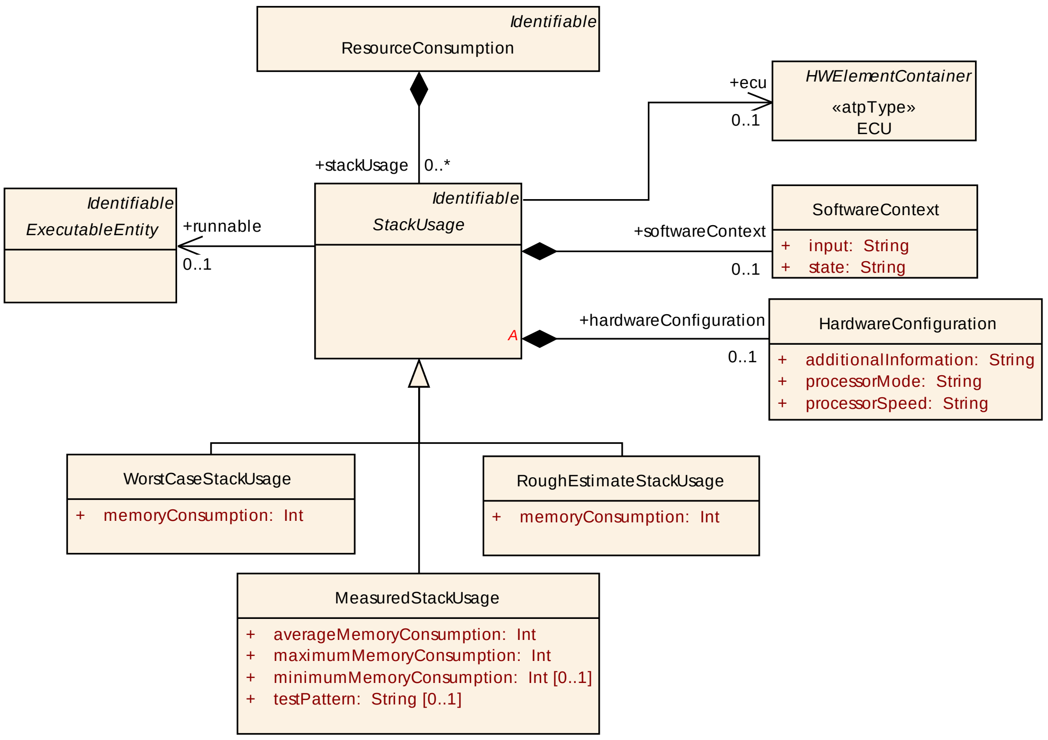

Figure 9.3: Stack Memory Consumption

The given stack memory consumption is dependent on the ECU, the software context and maybe also on the hardware configuration. The software context and the hardware configuration describe the state of the software and hardware under which the given stack usage was gathered. So for each given stack memory consumption these environmental descriptions have to be provided.

Class	**⟨⟨atpObject⟩⟩ StackUsage (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::StackUsage			
Class Desc.	Describes the stack memory usage of a software.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
ecu	ECU	0..1	reference	Reference to the ECU description this implementation is provided for.

hardware Configuration	Hardware Configuration	0..1	aggregation	Contains information about the hardware context this stack usage is describing.
runnable	Executable Entity	0..1	reference	Reference to the runnable this stack usage is provided for.
software Context	Software Context	0..1	aggregation	Contains details about the software context this stack usage is provided for.

Table 9.3: StackUsage

Class	⟨⟨**atpObject**⟩⟩ **WorstCaseStackUsage**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::StackUsage			
Class Desc.	Provides a formal worst case stack usage.			
Base Class(es)	StackUsage			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
memory Consumption	Integer	1	aggregation	Worst case stack consumption.

Table 9.4: WorstCaseStackUsage

Class	⟨⟨**atpObject**⟩⟩ **MeasuredStackUsage**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::StackUsage			
Class Desc.	The stack usage has been measured.			
Base Class(es)	StackUsage			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
average Memory Consumption	Integer	1	aggregation	The average stack usage measured.
maximum Memory Consumption	Integer	1	aggregation	The maximum stack usage measured.
minimum Memory Consumption	Integer	0..1	aggregation	The minimum stack usage measured.
testPattern	String	0..1	aggregation	Description of the test pattern used to aquire the measured values.

Table 9.5: MeasuredStackUsage

Class	⟨⟨**atpObject**⟩⟩ **RoughEstimateStackUsage**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::StackUsage			
Class Desc.	Rough estimation of the stack usage.			
Base Class(es)	StackUsage			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
memory Consump- tion	Integer	1	aggregation	Rough estimate of the stack usage.

Table 9.6: RoughEstimateStackUsage

9.4.3 Heap

Heap is the memory segment that is used to cover dynamic memory needs with explicit memory allocation and de-allocation. Since the allocation of the memory is controlled by the application program it also survives changes in the context of invocation from entering a function nesting level and leaving it again. So a memory block allocated in the subroutine can be used in the calling routine after the subroutine has returned. Also the allocated memory can be freed again in a different context.

Because of the independence of the heap consumption from processes and tasks only the whole software component or BSW Module heap consumption is provided in the description. The meta-model is shown in Figure 9.4.

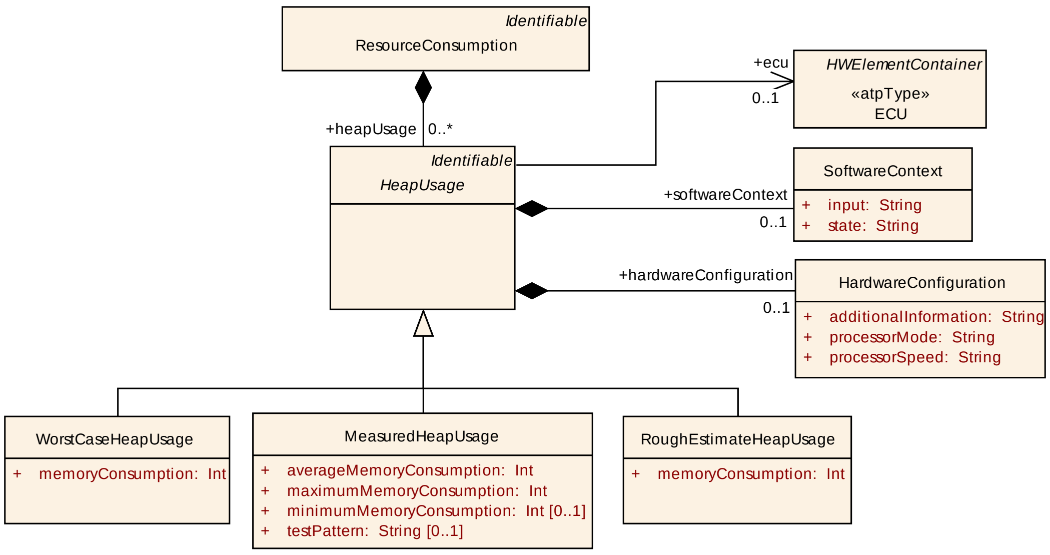

Figure 9.4: Heap Memory Consumption

The heap memory consumption also depends on the ECU, the software context and the hardware configuration.

Due to the highly dynamic nature of heap memory one problem is the segmentation of the available memory area. So in some cases there can be not enough memory allocated, even though the total amount of free heap memory is big enough, because the available memory space is not available continuously.

Class	⟨⟨**atpObject**⟩⟩ **HeapUsage (abstract)**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::HeapUsage			
Class Desc.	Describes the heap memory usage of a SW-Component.			
Base Class(es)	Identifiable			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
ecu	ECU	0..1	reference	Reference to the ECU description this implementation is provided for.

hardware Configuration	Hardware Configuration	0..1	aggregation	Contains information about the hardware context this heap usage is describing.
software Context	Software Context	0..1	aggregation	Contains details about the software context this heap usage is provided for.

Table 9.7: HeapUsage

Class	⟨⟨**atpObject**⟩⟩ **WorstCaseHeapUsage**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::HeapUsage			
Class Desc.	Provides a formal worst case heap usage.			
Base Class(es)	HeapUsage			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
memory Consumption	Integer	1	aggregation	Worst case heap consumption.

Table 9.8: WorstCaseHeapUsage

Class	⟨⟨**atpObject**⟩⟩ **MeasuredHeapUsage**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::HeapUsage			
Class Desc.	The heap usage has been measured.			
Base Class(es)	HeapUsage			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
average Memory Consumption	Integer	1	aggregation	The average heap usage measured.
maximum Memory Consumption	Integer	1	aggregation	The maximum heap usage measured.
minimum Memory Consumption	Integer	0..1	aggregation	The minimum heap usage measured.
testPattern	String	0..1	aggregation	Description of the test pattern used to aquire the measured values.

Table 9.9: MeasuredHeapUsage

Class	⟨⟨**atpObject**⟩⟩ **RoughEstimateHeapUsage**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::HeapUsage			
Class Desc.	Rough estimation of the heap usage.			
Base Class(es)	HeapUsage			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
memory Consump-tion	Integer	1	aggregation	Rough estimate of the heap usage.

Table 9.10: RoughEstimateHeapUsage

9.5 Execution Time

9.5.1 General

This subsection defines a model to describe the `ExecutionTime` of a specific `ExecutableEntity` of a specific `Implementation`.

Section 9.5.3 describes the goals and scope of the `ExecutionTime` description proposed.

Section 9.5.4 lists all the thoughts and observations that lead to the actual model which is described in section 9.5.5.

9.5.2 Preliminaries

This subsection assumes that the reader is familiar with the definition of the following terminology (please see the AUTOSAR Glossary [4] for details):

- task
- thread
- process
- executable entity
- (worst case) execution time
- (worst case) response time

9.5.3 Scope

9.5.3.1 Assertions Versus Requirements

The `ExecutionTime` 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, on the hardware or on the scheduling policy.

9.5.3.2 In Scope

This section proposes a description of the `ExecutionTime` of a `ExecutableEntity` of an `Implementation`. Very roughly, this description includes:

- 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 is to find 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. The following sections analyze what this context is and provide an appropriate structure for this information.

9.5.3.3 Out of Scope

It is however not in the scope of this section to specify how the execution time of a runnable entity can be or should be measured or analyzed. We will not discuss what tools or techniques can be used to find the execution time or worst-case execution time of a piece of software.

It also is not in the scope of this section to define how information about execution times is used when integrating various software onto one ECU. Similarly this section does not deal with the response time of the system to certain events. The response time does not only depend on the execution times of the involved software but also on the infrastructure overhead and on the scheduling policies which are used.

The focus also is on the description of the execution time of assembly instructions (typically generated out of compiled C or C++ code). The execution time of e.g. Java byte-code on a virtual machine has not been explicitly considered.

9.5.4 Background

This section provides some background to the proposed solution. Readers who want to skip to the result should go to section 9.5.5. 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 unless a precise compiler (and compiler options) are also provided so that a unique set of assembly instructions can be generated out of the source-code. In addition, the execution time of such a sequence of assembly instructions depends on:

1. the hardware-platform
2. the hardware state
3. the logical (software) context
4. execution time of external pieces of code called from the software

These dependencies are discussed in detail in the following sections.

9.5.4.1 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
  - 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 eventually accessed *as memory* by the I/O hardware abstraction, the speed of those devices potentially has a large influence on the execution time of software.

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 versus execution out of ROM might have very different execution times
- when caching is present, the relative physical location of data accessed in memory might also influence the execution time

9.5.4.2 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

Although this influence is not relevant on simple or deterministic processors (without cache), the influence of the cache state on modern processors can be enormous (an order of magnitude difference is not impossible). 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 we will not provide explicit attributes for this purpose.

9.5.4.3 Dependency on Logical Context

This logical context includes:

1. the input parameters with which the runnable is called
2. also 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 software 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.

9.5.4.4 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:

1. Do not support this case at all: only code that does not rely on external libraries can be given an execution time
2. Support a description of the execution time for a very specific version (again at object-code level) of the libraries. The exact versions of external libraries used would be described together with the execution time. In addition, the relative location in memory of the runnable and the library, the HW-state with respect to the library (e.g. whether this code is in cache or not) and the logical state of the library might have an influence.
3. Conceptually, it might be possible to support a description of the software, which explicitly describes the dependency on the execution times of the library. This description would include:
   (a) the execution time of the code provided by the software itself
   (b) a specification of which external library-calls are made (with what parameters, how often, in what order, ...)

Option 3 is deemed unrealistic and impractical and is not supported. Option 2 however is important as many software might depend on very simple but very common external libraries (like a math-library that provides floating-point capability in software). Option 2 will therefore be supported for the case that the external library does not have an additional logical context which influences its execution time.

9.5.5 Description-Model for the Execution Time

9.5.5.1 Inclusion in the Overall Model

Figure 9.5 shows how the `ExecutionTime` is part of the overall description of the `Implementation` of software. The description of the `Implementation` references the description of the `ExecutableEntity`.

Each description of such an `ExecutableEntity` (of a specific `Implementation`) can include an arbitrary number of `ExecutionTime` descriptions. Thereby this `ExecutionTime` description may also depend on code or data variant of the `Implementation`.

It is expected that many `ExecutableEntity` will not have `ExecutionTime` descriptions. For `ExecutableEntity` that do have `ExecutableEntity` descriptions, the software-implementor could provide several `ExecutionTime` descriptions: for example one per specific ECU on which the `Implementation` can run and on which the time was measured or estimated.

If an `ExecutableEntity` is defined to be running in an `ExclusiveArea` the `ExecutionTime` of the whole `ExecutableEntity` can be considered to allow the scheduler configuration an optimization of the data consistency mechanism.

If an `ExecutableEntity` is defined to be able to enter an `ExclusiveArea` the `ExecutionTime` can be specified for each section. The time provided is the time consumed AFTER the call to enter the `ExclusiveArea` and BEFORE the call to leave the `ExclusiveArea`.

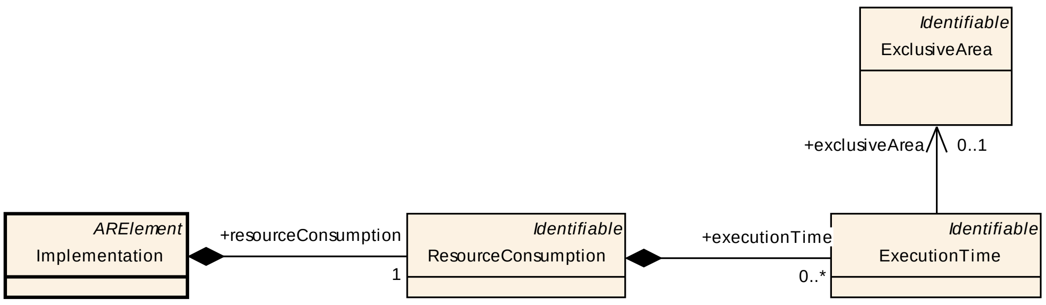

Figure 9.5: Position of `ExecutionTime` description in the overall model

9.5.5.2 Detailed Structure of an Execution-Time Description

Figure 9.6 shows the details of an execution time description. The following paragraphs describe aspects of this model in more detail.

The following shows the attributes of the `ExecutionTime` in tabular form:

Class	⟨⟨**atpObject**⟩⟩ **ExecutionTime**

Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::ExecutionTime			
Class Desc.	Base class for several means how to describe the ExecutionTime of software. The required context information is provided through this class.			
Base Class(es)	Identifiable			
Attribute	**Datatype**	**Mul.**	**Link Type**	**Description**
ecu	ECUPrototype	1	aggregation	Provides information on a ECUPrototype based on one ECU type.
exclusive Area	Exclusive Area	0..1	reference	Reference to the ExclusiveArea this execution time is provided for.
external Library	Dependency OnLibrary	*	reference	If this dependency is specified, the execution time of the library code is included in the execution time data for the runnable.
hardware Configuration	Hardware Configuration	1	aggregation	Provides information on the HardwareConfiguration used to specify this ExecutionTime.
memory Section Location	Memory Section Location	*	aggregation	Provides information on the MemorySectionLocation which is involved in the ExecutionTime description.
runnable	Executable Entity	0..1	reference	Reference to the runnable this execution time is provided for.
software Context	Software Context	1	aggregation	Provides information on the detailed SoftwareContext used to provide the ExecutionTime description.

Table 9.11: ExecutionTime

9.5.5.3 ExecutionTime References an ”ECU”

The `ExecutionTime` references an `ECU` (the concept `ECU` is defined by the ECU-Resource-Template [14]). This `ECU`-reference uniquely describes the hardware for which the `ExecutionTime` is provided. This includes: the kind of processor, the type of MMU, the type of caches, type of memory available,...

Note that this reference to an `ECU` has a different semantic than the attribute `processor` in the `Implementation`. The `processor` defines the family of processors on which the provided implementation may run (it is a requirement on the hardware on which the component may be deployed). The `ECU` on the other hand (of which the processor only is one part) is a statement on the context of the `ExecutionTime`. Of course, the processor of the `ECU` should be equal to the processor specified in the `Implementation`. Note that the `ECU` might include specific hardware that has no influence on the `ExecutionTime`. Despite this, it seems currently better to specify a reference to the entire hardware-platform used rather than introduce another hardware sub-system that includes all hardware-elements that influence the `ExecutionTime` of software.

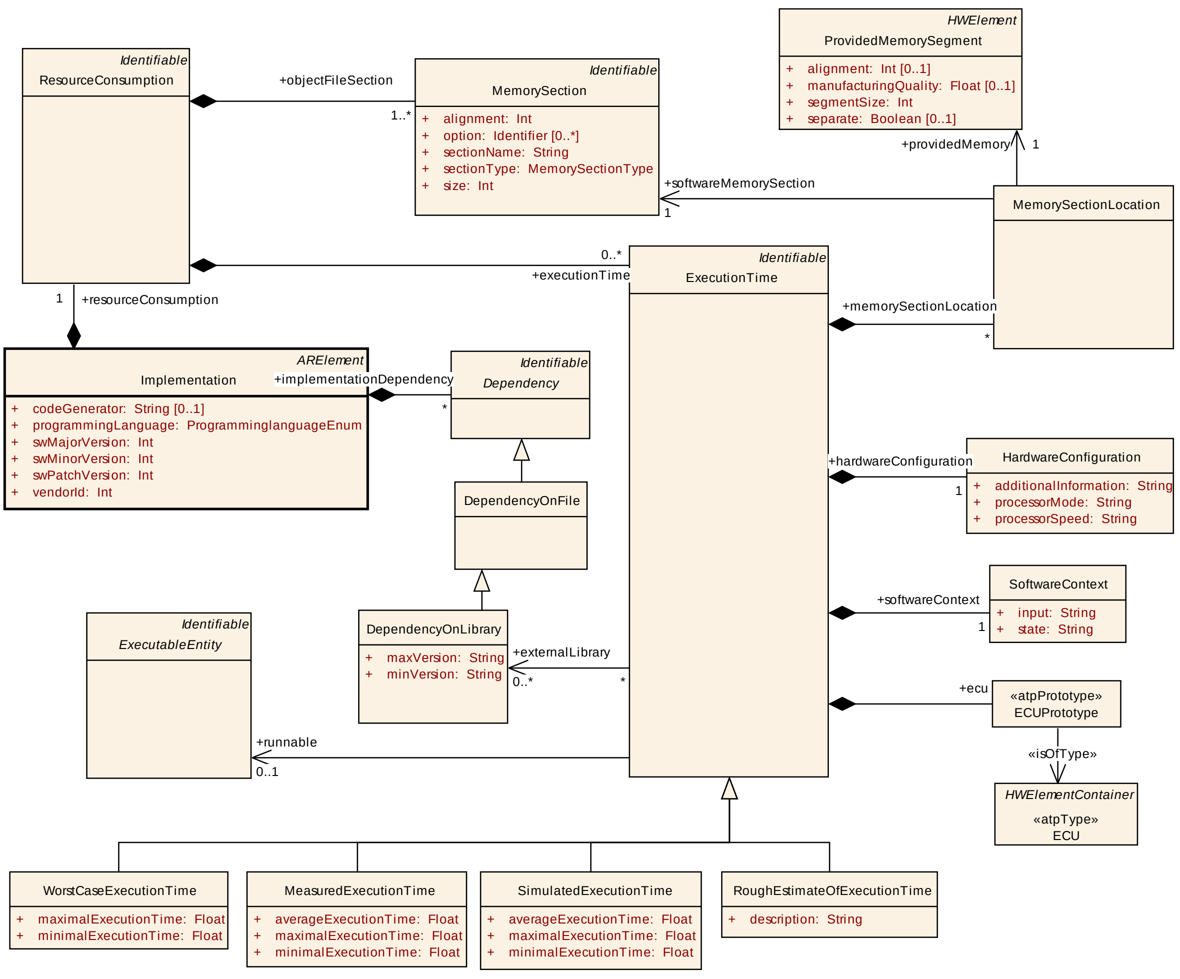

Figure 9.6: Detailed structure of an `ExecutionTime` description

9.5.5.4 ExecutionTime Includes a HW-Configuration

The `ECU` described through the `ecu` attribute can still run in several HW-modes. For example, many ECUs can run in several ”speed”-modes (for example a normal fast-mode and a low-power slow mode). The goal of the HW-Configuration is to describe this. The attributes `processorSpeed` and `processorMode` should describe the specific mode of the `ECU`.

Because of the potential dependency on many other HW-Configuration settings (such as caching policy, MMU-settings, ...), a generic attribute `additionalInformation` is provided. Because the exact structure of the information seems to depend so much on the specific case, all attributes are unstructured text.

Class	⟨⟨**atpObject**⟩⟩ **HardwareConfiguration**
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption
Class Desc.	Describes in which mode the hardware is operating while providing the ExecutionTime.
Base Class(es)	ARObject

Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
additional Information	String	1	aggregation	Specifies additional information on the HardwareConfiguration.
processor Mode	String	1	aggregation	Specifies in which mode the processor is operating.
processor Speed	String	1	aggregation	Specifies the speed the processor is operating.

Table 9.12: HardwareConfiguration

9.5.5.5 ExecutionTime Includes a MemorySectionLocation

For each `memorySection` of the `Implementation`, the `ExecutionTime` must specify where this section was located on the physical memory of the `ECU`. The `memorySection` on the software are described in the `softwareMemorySection` of the `Implementation`. The available memory-regions on the hardware are described inside the description of the `ECU`. The `ExecutionTime` contains descriptions of the location of the memory sections `MemorySectionLocation` which link a software memory section to a hardware memory section on the `ECU`.

Class	⟨⟨**atpObject**⟩⟩ **MemorySectionLocation**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::ExecutionTime			
Class Desc.	Specifes in which hardware ProvidedMemorySegment the softwareMemorySection is located.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
provided Memory	Provided Memory Segment	1	reference	Reference to the hardware ProvidedMemorySegment.
software Memory Section	Memory Section	1	reference	Reference to the MemorySection which is mapped on a certain hardware memory segment.

Table 9.13: MemorySectionLocation

9.5.5.6 ExecutionTime Includes a SoftwareContext

The `SoftwareContext` is the logical context for which the `ExecutionTime` is given. This includes two aspects:

1. the values of the input-parameters to the software
2. the state the logic of the runnable depends on

In the current form, both attributes are of type String and can contain free-form text describing this state. For the attribute input, it might be appropriate to refine this into a more formal description of the values of the parameters. For the attribute state, it is difficult to go beyond an informal text-field, because the state is a private matter of the component and there currently is no explicit mechanism in AUTOSAR to describe the value of this state. Further, it is possible to provide several execution times of a runnable entity, for example, in case of different values of the input-parameters. This is one of the reasons why the template supports an arbitrary number of ExecutionTime.

Class	⟨⟨**atpObject**⟩⟩ **SoftwareContext**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption			
Class Desc.	Specifes the context the software is whose ExecutionTime is provided.			
Base Class(es)	ARObject			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
input	String	1	aggregation	Specifies the input vector which is used to provide the ExecutionTime.
state	String	1	aggregation	Specifies the state the software is in when the ExecutionTime is provided.

Table 9.14: SoftwareContext

9.5.5.7 Dependency on External Libraries

The ExecutionTime measurements can depend on the precise version of external libraries (such as a math-emulation library) that have been used. This information can be included by adding a reference to an object of type DependencyOnLibrary which must be aggregated by the corresponding Implementation.

If such a reference is specified, the ExecutionTime includes the execution time of that specific library version.

In case the Implementation aggregates attributes of type DependencyOnLibrary, to which the ExecutionTime does not refer, it means that the execution time of the library code is NOT included in the execution time of the ExecutableEntity.

9.5.5.8 Several Qualities of Execution Times

9.5.5.8.1 WorstCaseExecutionTime

The WorstCaseExecutionTime is used to build the application schedule. It is an overall approximation of an ExecutableEntity which will be running on a hardware ECU context.

Further ”worst-case” means that an ”analytic” method was used to find the worst-case (=guaranteed) boundaries. But this boundary has a lower-limit and an upper-limit.

Considering the cache processor ECU, an execution time could be computed, and it depends on cache level. A `maximalExecutionTime` and a `minimalExecutionTime` has to be filled.

Class	⟨⟨**atpObject**⟩⟩ **WorstCaseExecutionTime**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::Execution Time			
Class Desc.	WorstCaseExecutionTime provides an analytic method for specifying the minimum and maximum execution time.			
Base Class(es)	ExecutionTime			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
maximal Execution Time	Float	1	aggregation	Maximum WorstCaseExecutionTime.
minimal Execution Time	Float	1	aggregation	Minimum WorstCaseExecutionTime.

Table 9.15: WorstCaseExecutionTime

9.5.5.8.2 MeasuredExecutionTime

The `MeasuredExecutionTime` describes the `ExecutableEntity` runtime on `ECU`.

Class	⟨⟨**atpObject**⟩⟩ **MeasuredExecutionTime**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::Execution Time			
Class Desc.	Specifies the ExecutionTime which has been gathered using measurement means.			
Base Class(es)	ExecutionTime			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
average Execution Time	Float	1	aggregation	Average MeasuredExecutionTime.
maximal Execution Time	Float	1	aggregation	Maximum MeasuredExecutionTime.
minimal Execution Time	Float	1	aggregation	Minumum MeasuredExecutionTime.

Table 9.16: MeasuredExecutionTime

9.5.5.8.3 SimulatedExecutionTime

A `SimulatedExecutionTime` describes the time information which are coming from a simulation. Simulation could be based on:

- `ExecutableEntity` model on specific hardware with time weighting to simulate processor time behavior
- `ExecutableEntity` model before generation code

Class	⟨⟨**atpObject**⟩⟩ **SimulatedExecutionTime**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::ExecutionTime			
Class Desc.	Specifies the ExecutionTime which has been gathered using simulation means.			
Base Class(es)	ExecutionTime			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
averageExecutionTime	Float	1	aggregation	Average SimulatedExecutionTime.
maximalExecutionTime	Float	1	aggregation	Maximum SimulatedExecutionTime.
minimalExecutionTime	Float	1	aggregation	Minimum SimulatedExecutionTime.

Table 9.17: SimulatedExecutionTime

9.5.5.8.4 RoughEstimateOfExecutionTime

A `RoughEstimateOfExecutionTime` describes the time information which are based on some estimation.

Class	⟨⟨**atpObject**⟩⟩ **RoughEstimateOfExecutionTime**			
Package	M2::AUTOSARTemplates::CommonStructure::ResourceConsumption::ExecutionTime			
Class Desc.	Provides a description of a rough estimate on the ExecutionTime.			
Base Class(es)	ExecutionTime			
Attribute	***Datatype***	***Mul.***	***Link Type***	***Description***
description	String	1	aggregation	Provides description on the rough estimate of the ExecutionTime.

Table 9.18: RoughEstimateOfExecutionTime