Document Title	Template UML Profile and Modeling Guide
Document Owner	AUTOSAR GbR
Document Responsibility	AUTOSAR GbR
Document Identification No	121
Document Classification	Auxiliary

Document Version	2.2.0
Document Status	Final
Part of Release	3.0
Revision	0001

Document Change History

Date	Version	Changed by	Change Description
20.12.2007	2.2.0	AUTOSAR Administration	• Removed stereotype <<atpIdentifiable>> • Added stereotypes <<import>> • Refined stereotypes <<instanceRef…> • Optimized consistency of document • Added concept for documentation of model evolution • Added stereotype <<splitable>> • Document meta information extended • Small layout adaptations made
31.01.2007	2.1.0	AUTOSAR Administration	• Detailed modeling of instanceRefs and indexedRefs added • Example on instanceRefs added • Legal disclaimer revised • Release Notes added • “Advice for users” revised • “Revision Information” added
18.10.2005	2.0.0	AUTOSAR Administration	• Formal reorganization of document • Alignment with UML2 • Formal UML profile • Extraction of common patterns into separate document • Methodology added for model subset specification
09.05.2005	1.0.0	AUTOSAR Administration	Initial release

Page left intentionally blank

Disclaimer

Any use of these specifications requires membership within the AUTOSAR Development Partnership or an agreement with the AUTOSAR Development Partnership. The AUTOSAR Development Partnership will not be liable for any use of these specifications.

Following the completion of the development of the AUTOSAR specifications commercial exploitation licenses will be made available to end users by way of written License Agreement only.

Advice to users of AUTOSAR Specification Documents:

AUTOSAR Specification Documents may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).
Any such exemplary items are contained in the Specification Documents for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such Specification Documents, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

1 Introduction 7
1.1 Origins and Goals 7
1.2 Document Guide 8
1.3 Terminology 8
1.3.1 Terms 8
1.3.2 UML Diagrams 9
1.3.3 Profile Classes 9
1.3.4 Specification Items 9
2 Requirements Traceability 11
3 AUTOSAR Template Models at a Glance (informative) 12
3.1 Scope 12
3.2 Usage of Packages 12
3.3 Classes and Attributes 12
3.4 Enumerations 13
3.5 Associations and Composite Aggregations 13
3.6 Dependencies 14
3.7 Types and Prototypes 14
3.8 Structure Elements 19
3.9 Constraints 20
3.10 Metamodel Evolution 21
4 Alignment with UML and MOF 22
5 UML Profile for AUTOSAR Templates 24
5.1 Supported Modeling Constructs 24
5.2 UML Profile Specification 24
5.2.1 Classes 25
5.2.2 Class Attributes 27
5.2.3 Mixed Content 30
5.2.4 Types, Prototypes, and Structure Elements 31
5.2.5 Associations 33
5.2.6 IsOfType Association 35
5.2.7 IndexedRef Association 36
5.2.8 InstanceRef Association 36
5.2.8.1 Detailed Representation of InstanceRef Association 37
5.2.8.2 Short Representation of InstanceRef Association 41
5.2.8.3 42
5.2.8.4 Constraints on instanceRef associations 42
5.2.9 Constraints 46
5.2.10 Dependencies 48
5.2.11 Packages 49
5.2.12 Primitive Types 50
5.2.13 Enumerations 53
5.2.14 Tagged Values 53
5.2.15 Splitable attributes, associations and aggregations 53
5.3 Summary 55
6 Conventions 57

6.1 Naming Conventions ... 57
6.1.1 Language ... 57
6.1.2 Model Element Names ... 57
6.1.3 Class Names ... 57
6.1.4 Diagram Names ... 58
6.2 Modeling Conventions ... 58
6.2.1 Unambiguous Models ... 58
6.2.2 Abstract Classes ... 58
6.2.3 Enumerations ... 58
6.2.4 Class Attributes vs. Associations ... 59
6.2.5 Physical Quantities ... 59
6.2.6 Modeling Aggregations and Associations in Enterprise Architect ... 60
6.2.7 Constraints in Metamodel ... 60
6.2.7.1 Context Class ... 60
6.2.7.2 Entering Information ... 60
6.2.8 Implicit & Explicit Stereotypes ... 62

7 Template Model Infrastructure ... 63
7.1 Overview ... 63
7.2 Packaging ... 63
7.3 Identifiable Classes ... 64

8 AUTOSAR Template Subsets ... 65
8.1 Goal ... 65
8.2 Related concepts ... 65
8.3 Subset Notation ... 65
8.3.1 Importing Classes ... 65
8.3.2 Attaching Constraints ... 66
8.3.2.1 Attribute Value Constraints ... 67
8.3.2.2 Multiplicity constraints ... 67
8.3.2.3 Location of constraints ... 67

9 Specification Items ... 69

10 Appendix ... 71
10.1 Design Rationales (informative) ... 71
10.1.1 Choice of Metamodeling Mechanism ... 71
10.1.1.1 Instantiation of MOF Entities ... 71
10.1.1.2 Instantiation of UML Entities ... 72
10.1.1.3 Extending UML through UML Profiles ... 72
10.1.1.4 Templates as Direct UML Profiles ... 73
10.1.1.5 Conclusion ... 73
10.1.2 Modeling Prototype Attributes ... 74
10.1.2.1 Explicit Prototype Attributes ... 74
10.1.2.2 InstanceRef ... 75
10.1.2.3 Conclusion ... 75
10.1.3 AUTOSAR Subset Modeling ... 75
10.1.3.1 Tagged values ... 75
10.1.4 Stereotypes vs. Tagged Values ... 76
10.2 References ... 76
10.2.1 Normative References ... 76
10.2.2 Informative References ... 77

1 Introduction

1.1 Origins and Goals

AUTOSAR attempts to allow for a very flexible yet stable and reliable software engineering lifecycle through precise and formal description of all relevant aspects of a distributed system of embedded controllers and the corresponding executed software units.
The descriptions range from high level requirements on interfaces of software components to low level constraints on certain bits of a specific bus message. Various work packages in AUTOSAR determine the information that needs to be captured in the different descriptions.
For instance, in [18] it is defined how AUTOSAR software components need to be described, e.g. when a requirements specification is exchanged between OEM and supplier. The collection of attributes required specifying various AUTOSAR relevant artifacts like software components, ECUs and so on is called an *AUTOSAR template*. Once information is available a template is said to be filled out, leading to an *AUTOSAR description*.
The templates defined by AUTOSAR are expressed in form of UML class diagrams[1] defined in [1], or in a certain subset to be more precise. The resulting UML model is called the *template model*.
This document formally specifies the allowed subset of UML features and also gives guidelines on various modeling questions when creating template models.

Scope
While giving an exact definition about template modeling capabilities in AUTOSAR this document will *not* address the following issues:

- Models of AUTOSAR specific content, e.g. which relation needs to exist between a port interface and a software component[2]. Such content is described by the responsible work packages only. Exceptions are certain common patterns that need to be shared between multiple work packages. Those are collected in [24].
- Definition of the mapping from a UML based template model into other description domains, e.g. XML DTDs or database schemas. [19] exactly describes this mapping, including the UML tags and values to control DTD/XSD creation.

Who should read this specification?
This specification is relevant for:

- template creating work packages
- tool vendors that need to implement and navigate template models

It is not relevant for

- automotive engineers building systems with AUTOSAR technology
- process engineers creating the AUTOSAR process

[1] There are many introductions into UML. E.g. see [12] for an overview, [16] gives an excellent explanation in German.
[2] These terms are randomly chosen from an existing template [18], their actual meaning does not matter here at all.

- conceptual work in AUTOSAR (e.g. defining how timing is addressed)

In particular, the reader will not be able to learn good modeling with constructs provided by UML simply by studying this specification. Instead, UML – at least the subset of class diagrams – is a mandatory prerequisite for successful application of the concepts introduced here.

1.2 Document Guide

This specification gives information about modeling AUTOSAR templates on three related levels, thereby addressing the needs of different audiences.
If your interest is mainly focused on the actual template modeling task, you should read chapter *3* AUTOSAR Template Models at a Glance (informative), which gives a quick overview of the modeling patterns used for AUTOSAR templates without much delving into the formal details and reasons. Chapter *6* Conventions gives further required detail on how certain features have to be realized in the model.

```
The formal groundwork has been laid in chapter 0 value   := “(since ” date
“) ” comment
```

For example:

```
deprecated = (since 20.10.2007) Please use ApplicationSoftwareComponent
                                instead.
```

Alignment with UML and MOF, chapter *5* UML Profile for AUTOSAR Templates and chapter *6* Conventions.
All specification items given in this document are finally listed in chapter *9* Specification Items, which allows for a systematic model compliance check as well as formal references to the particular items.

1.3 Terminology

1.3.1 Terms

In this specification the key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when EMPHASIZED are to be interpreted as described in RFC 2119 [6]. In addition, the terms defined in the following list are used throughout this specification:

metamodel

A metamodel is a precise definition of the constructs and rules needed for creating semantic models [17]. In AUTOSAR, templates defining constructs like *software component* and *ECU* are used to create models of AUTOSAR software and hardware systems.

metalevel

A model and its metamodel are said to be on different metalevels. A standard setup of metalevels is the so-called *four-layer metamodel hierarchy*, consisting of the four metalevels *M0*, *M1*, *M2* and *M3* [2], where entities in M0 are expressed in terms of M1 entities, M1 is expressed in terms of M2 entities and so on.

There may be more or less than four metalevels (e.g. the hierarchy described in this specification has five layers). It is a feature of the particular metamodels defined by the OMG[3] that their hierarchy (so far) always stops at M3.

reflective metamodel

If a metamodel can be described in terms of its own entities it is called reflective [2]. A reflective metamodel is required to end the metamodel hierarchy. E.g. in OMG's four-layer metamodel hierarchy the metamodel at M3 is reflective.

UML profile

As defined in [2] profiles can be used to extend the set of entities provided by the UML specification in order to adapt to a certain domain. The new metaclasses defined in a profile are called *stereotypes*. They always extend existing metaclasses by adding semantics and constraints. A metamodel created in form of a profile exists on the same metalevel as the metamodel it is derived from.

1.3.2 UML Diagrams

The diagrams in this specification are consistent with UML 2.0. The underlying models are assumed to be complete even though certain elements might not be shown in a particular diagram to simplify understanding.
This is particularly true for diagrams that show extracts of the UML 2.0 metamodel. Due to the highly modular structure and the extensive use of redefinition of classes, attributes and associations in UML 2 it is sometimes difficult to show a certain aspect in just one or two diagrams while still preserving readability. For this reason, the shown UML 2.0 metamodel extracts are simplified to contain only the parts relevant to the topic at hand. Applied simplifications include:

- Association constraints *{subsets…}* and *{union}* are not shown. It is still shown whether an attribute or an association is derived (forward slash) and/or *{ordered}*.
- In case of derived associations or roles, it is possible that only the derived or the deriving element is shown, whatever fits the discussed context better.
- Conceptual relations to abstract classes may not be shown if none of the concrete subclasses will be part of the template profile.

1.3.3 Profile Classes

The classes defined as part of the AUTOSAR template profile are marked with a prefix *atp* for *AUTOSAR template profile*.

1.3.4 Specification Items

At the end of each relevant section a list of specification items is summarizing the requirements on models resulting from this section. Chapter 9 collects those items, which then in turn can be used to quickly check the conformance of a given model.

[3] Object Management Group, http://www.omg.org/

The id as well as the short description of a specification items is listed in the following form:

ATPS-xxxx	This is the description of a specification item.

2 Requirements Traceability

This document MUST satisfy the relevant requirements from [8] and [9], which have the purpose of defining how AUTOSAR tools can successfully and efficiently create and exchange data. This creates requirements on the used metamodel.

Requirement	***Satisfied by***
[ATUC_007] Repository for AUTOSAR models	Formal definition of meta model as UML profile, leading to a repository model (see chapter 5).
[ATI0003] Support for model validity checks	Definition of how formal and informal constraints have to be used in model (see section 6.2.7).
[ATI0016] Documentation of authoring tool SHOULD describe supported features	Explicit concept supporting subset definition (see chapter 8).

3 AUTOSAR Template Models at a Glance (informative)

This section provides a quick overview of the rules and limitations on AUTOSAR template models introduced by this document. The intention is to ease actual application of the document without requiring in-depth UML metamodel knowledge.
To avoid repeating parts of the normative sections of this document, especially the modeling conventions in section 6, they will simply be referenced in case they fit the style of this chapter.

3.1 Scope

AUTOSAR template metamodels are expressed in form of simplified UML class diagrams. This section assumes that general UML knowledge is available to the reader. The concepts of classes, attributes, associations and aggregations are *not* explained. Instead the limitations and required elements in such diagrams are pointed out.
However, reading this chapter is *not* sufficient for validation of a template model against the profile defined in here. For applications like this refer to the normative parts of the document, namely chapters 5, 6, 7 and 8, and finally – in form of a short but formal summary – chapter 9.

3.2 Usage of Packages

You are invited to structure your models in UML packages. However, note that all classes must have a unique name within the model.
While in regular UML models a package forms a namespace and classes in different packages can still be distinguished, this is not the case for AUTOSAR template models. The classes in the model will eventually end up in form of database tables or XML elements, and it is assumed that the target platform (databases, XML, programming language, …) is not necessarily supporting namespaces.

3.3 Classes and Attributes

Figure 3-1 shows a simple class. The following rules are defined in addition to standard UML class diagrams:
All elements in a template model have to be public. This includes packages, classes, attributes, associations and so on.
The names of classes and attributes have to follow certain rules, like the so-called “camel-case” convention. Class names must start with a capital letter, attribute names with a lower letter.

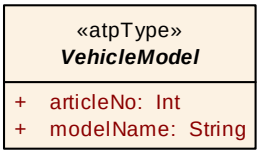

Figure 3-1: Example class in a metamodel.

Abstractness of classes is used to point out that a metaclass must not be instantiated.
Attributes can have the following primitive types: *Int*, *Float*, *String*, *Boolean*, *Identifier* or *UnlimitedNatural*.
An *Identifier* is a special form of a *String*, which basically has to satisfy the requirements for names used in typical programming languages. Identifiers have a maximum length of 32 characters.
An *UnlimitedNatural* is a natural numbers that also may have the value +infinity, expressed in form of an asterisk "*" (without quotes).
The stereotype <<*atpType*>> will be explained further down.

3.4 Enumerations

Enumerations are modeled as shown in Figure 3-2. See sections 6.2.3 and 6.2.4 for more information.

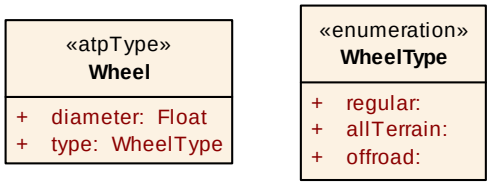

Figure 3-2: Example of an Enumeration on M2.

3.5 Associations and Composite Aggregations

Associations are used to express that the source class is referencing the target class. In AUTOSAR template models, associations are always unidirectional, as shown in Figure 3-3.

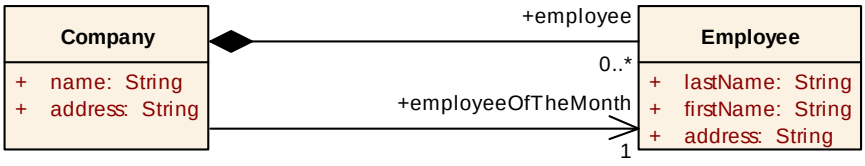

Figure 3-3: Example association and composite aggregation.

Aggregations must be composite aggregations, shared aggregations (diamond not filled) are not allowed. Navigability from whole to part is assumed and therefore does not need to be shown through an arrow.
For both, associations and aggregations, it is not permitted to assign names to the connection. Instead role names are required for navigable ends, the same is true for an explicitly given cardinality.
The documenting comments for associations and aggregations have to be given for the navigable end, not for the connector itself.

3.6 Dependencies

Dependencies are used very rarely in AUTOSAR template models. Dependencies which are not decorated with a stereotype are used for documentation only. They are used in overview diagrams in order to indicate a relationship between two elements in the template model. The detailed relationship should be described in separate diagrams.
Dependencies which are decorated with a stereotype have special semantics:

- If the stereotype <<*import*>> is applied then the UML2.0 semantics of PackageImport /ElementImport is applied: The referenced package/element is imported. These dependencies are e.g. used to describe subsets of the metamodel or to reuse elements defined in other standardization organizations such as ASAM MSR-SW
- If the stereotype <<*instanceRef*>> is applied, then the dependency shows an overview of an instance-reference. The detailed representation is described in diagrams that are contained in the "_instanceRef" package. Details about instance-references are described in the following section.

For more detailed description of dependencies and its stereotypes please read section 5.2.10.

3.7 Types and Prototypes

Creating template models involves being clear whether one is modeling some kind of reusable type or an actual usage of a type. In UML the typical terms are *types* and *roles*. For historical reasons, AUTOSAR is not using *role* but *prototype*.
A typical example of a model involving types and prototypes is shown in Figure 3-4, where some of the classes mentioned before show up in their context.
The model describes a point-of-sales selling the vehicles it has in store. Each vehicle has a VIN (vehicle identification number) and obviously is of a certain type (the vehicle's model).
Let's now look at two possible vehicle types, e.g. a car and a bike. Both types have an article number (model number) and a model name. Also, all vehicles have a set of wheels, but how many and how/where they are used differs between types.
How and where they are used is expressed in class *WheelUsage*. For instance the attribute *wheelName* could be "front left" for a car, while the name would be "front" for a bike. On the other hand, the four wheels of a car are basically all of the same wheel type, as is expressed in the class *Wheel*.
The Wheel*RepairOrder* class is discussed below.

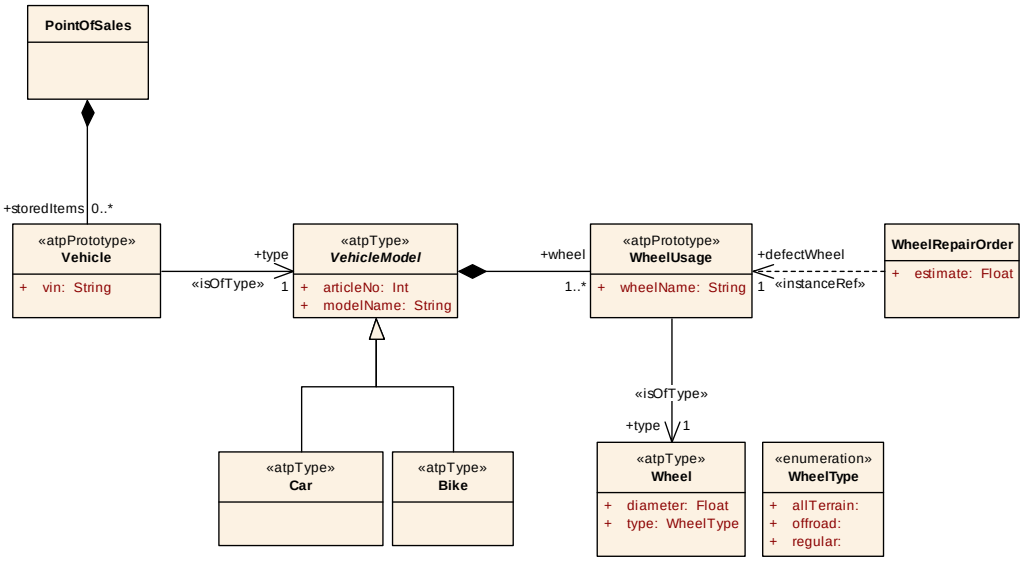

Figure 3-4: Sample model involving types and prototypes.

Some of the classes and associations carry stereotypes to indicate certain semantics. In particular we have:

- <<*atpType*>> indicates that a class describes a reusable type.
- <<*atpPrototype*>> indicates that a class described a certain usage of another class, its type.
- <<*isOfType*>> indicates the special relation of a prototype and its defining type. See the association between *Vehicle* and *VehicleModel*. Associations stereotyped <<*isOfType*>> always start at a prototype and end at a type. The type role name is always *type* and the cardinality is 1 by design.

For the final stereotype shown in Figure 3-4 we need to extend the example to some degree. Assuming our point-of-sales also has a garage where cars and their parts are repaired.

If we imagine the case that the assignment to fix a broken wheel is kept as an instance of class *WheelRepairOrder*, carrying e.g. the estimated price it will cost the customer, then this order needs to point out exactly which wheel needs the repair. Because wheels are always parts of vehicles, this means that a repair order needs to identify a particular wheel of a particular vehicle.

Now, a simple reference to e.g. the “front left” *WheelUsage* would not do the job because it would not identify the *context* of that front-left wheel, that is it would not identify a particular vehicle to which the wheel belongs. This is because, as shown in Figure 3-4, *WheelUsage* is aggregated by *VehicleModel,* the *type* of *Vehicle*, and not by *Vehicle* itself, meaning that *any* vehicle of the type at hand, in our example any car of the some given model, has a front-left wheel. The repair order needs to identify a particular car.

To indicate that a reference needs a context in order to identify an actual instance of a class the <<*instanceRef*>> relation is used:

- <<*instanceRef*>> indicates that the reference needs a context in order to identify an actual instance of the target class.The target of instance references always end at prototypes or at structure elements (see Sec. 3.8 below). The context references always end at prototypes. Figure 3-5 shows the short and

detailed representation of the instanceRef association to the *WheelUsage* as explained above. The short representation is shown as a dashed line which points to the target only. The detailed representation describes the target (*WheelUsage*) and the context in which the Wheel is used (*Vehicle*).

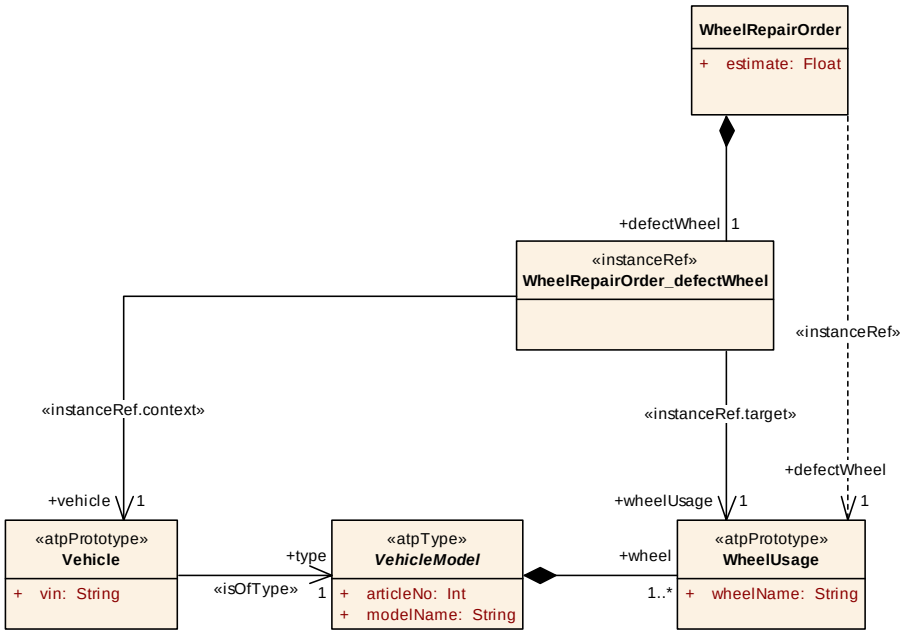

Figure 3-5: Short and detailed representation of an instanceRef association

The next example is taken directly from the AUTOSAR metamodel. Figure 3-6 shows the structure of *component types* – a fundamental abstraction in the metamodel. An AUTOSAR application is described as a composition of components, which themselves may be composed of smaller components and so on until the leaf-level of atomic components is reached. This structure is obtained through the interplay of types and prototypes: a composition type is made out of component prototypes which are typed by component types. Or: component types are used in composition types via component prototypes.

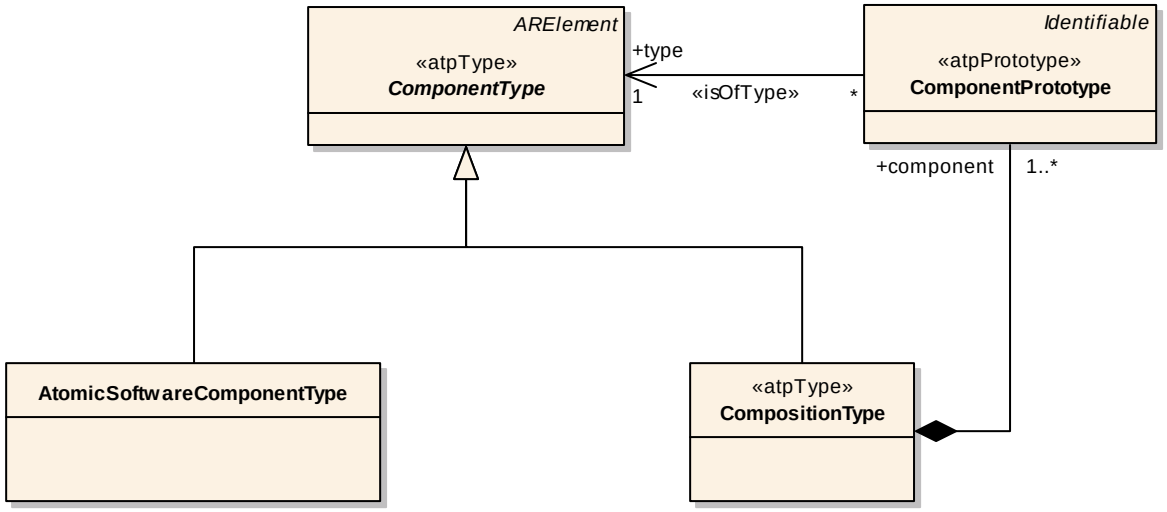

Figure 3-6: Component types and prototypes

Using this structure any number of component types may be defined. At the end, the system designer specifies exactly one composition type to be used as the system's software composition. This is shown in Figure 3-7.

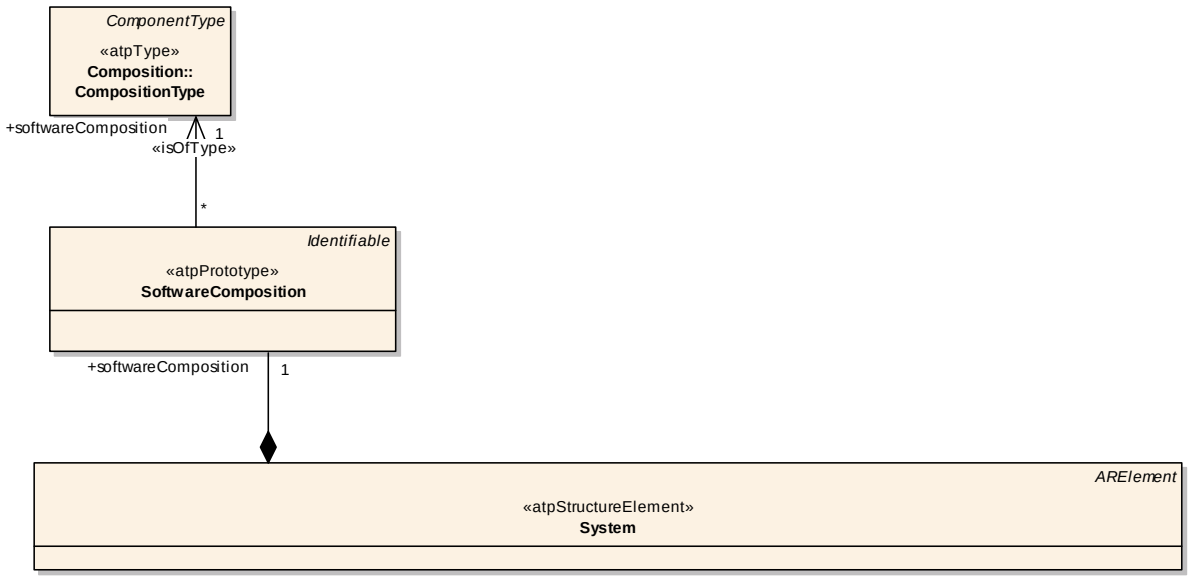

Figure 3-7 - The system's software composition

A next important step in the design process is the allocation of atomic component prototypes -- representing the leaves of the structure to be generated from a composition type -- to ECUs. This is achieved via a metamodel class called *SwCompToEcuMapping* which ties a particular ECU instance to a particular component prototype. The modeler defines one instance of this metaclass for every leaf of the tree, and the sum of all those mapping instances defines the allocation of all components to ECUs. This is shown in Figure 3-8.

In order to identify particular would-be instances of atomic component types it is not enough to specify component prototype – a context specifying the path from the software composition down to the leaves is needed. Thus, InstanceRefs are needed, which are shown in their short representation in Figure 3-8. As shown there, ECU instances also need a context but we will not get into this here.

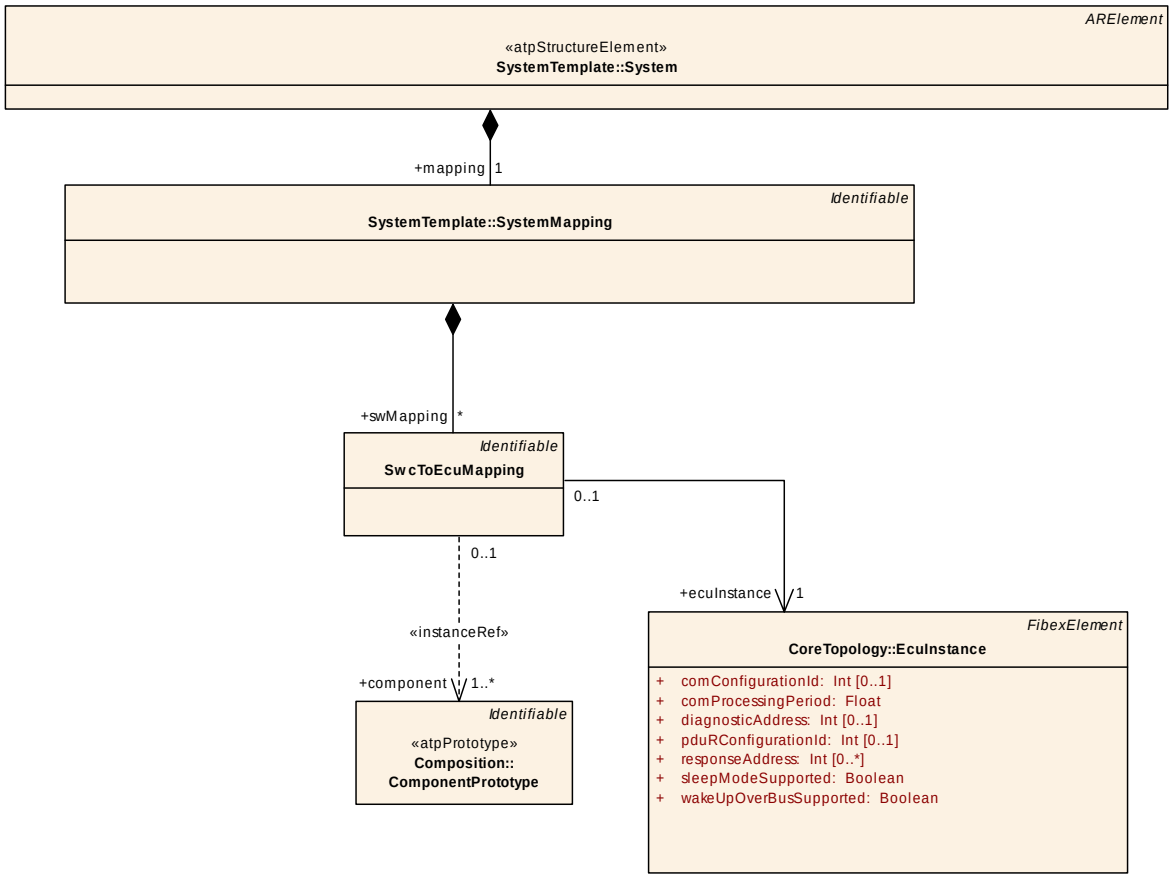

Figure 3-8: Mapping components to ECUs

Figure 3-9 shows the detailed representation of the InstanceRef identifying a component prototype in context. Note that we have here an example of a context path of unbounded length.

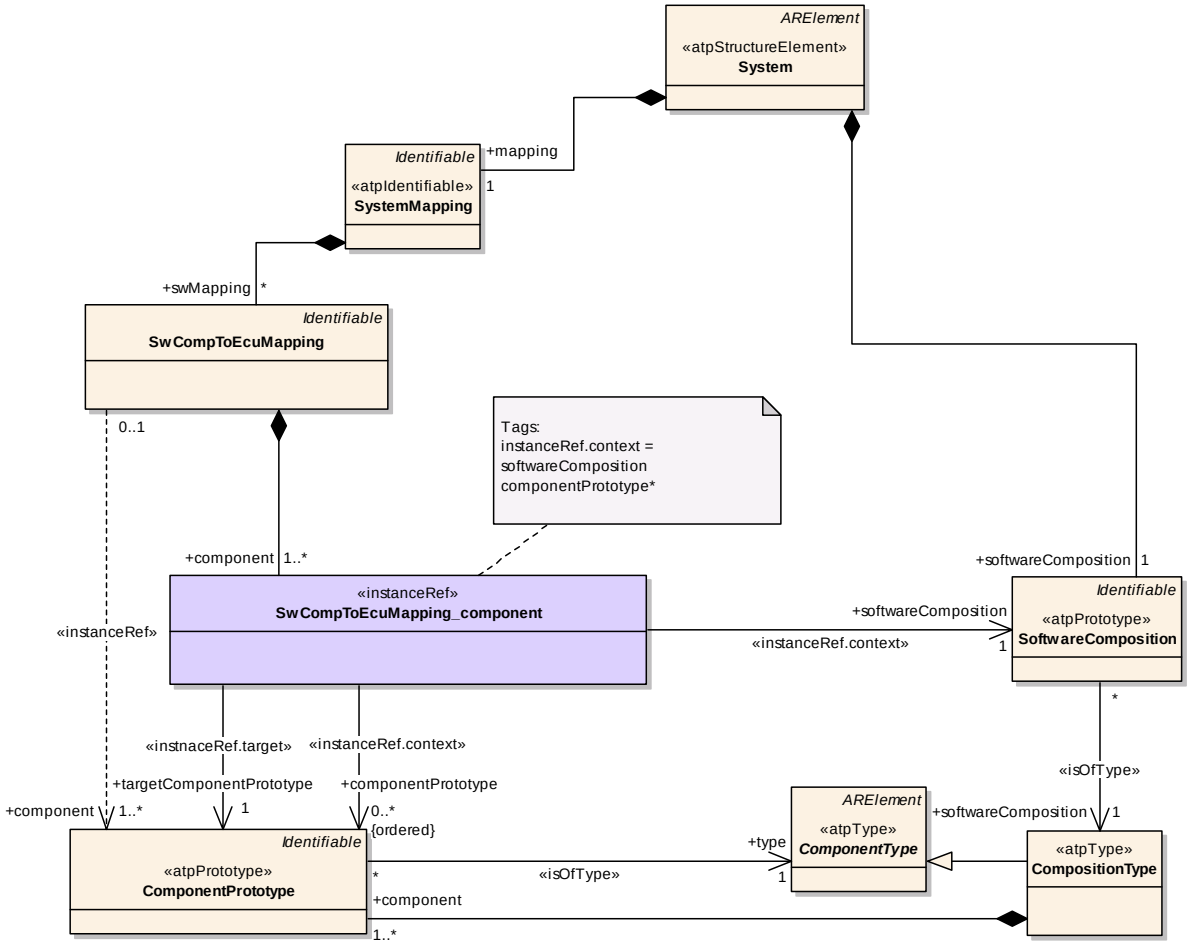

Figure 3-9: Detailed representation of the InstanceRef identifying a component prototype

3.8 Structure Elements

As explained above, the stereotype <<*atpType*>> indicates that a reusable type is being modeled. This reusability aspect of <<*atpType*>> is realized by the fact that a given class stereotyped with <<*atpType*>> may, via <<*isOfType*>>, be used to type prototypes, which are aggregated by other classes. So *Wheel* types *WheelUsage* which is aggregated by *VehicleModel*. But as this example shows, <<*atpType*>>s may be used not only to type prototypes but also to aggregate them, as does *VehicleModel*. Thus, an <<*atpType*>> has another aspect: it represents a container, something that has structure. To summarize, an <<atpType>> is something that (1) is reusable, and (2) can have structure.

In the type-prototype relationship, it is always the type which aggregates the prototype. That is, a type has structure and its aggregated prototypes define what that structure is. Each aggregated prototype defines a *feature* of the aggregating type and the sum of all features of a given type defines its structure. Each prototype in turn is typed by another type which in turn may aggregate prototypes within it, and so on. In the lower, instance level, such a chain results in a (possibly deeply) nested structure.

Given some concept of the problem domain, a wheel for example, the need to model it via two elements, a type – *Wheel* -- and a prototype – *WheelUsage*, stems from the fact that a wheel may play several roles in a vehicle. It can be the front-left wheel, or the rear-right wheel, and so on. But in many cases an element in the structure hierarchy just needs to be there once, without playing different roles.
In Figure 3-10 class *Chassis* has been inserted between *VehicleModel* and *WheelUsage*, introducing additional structure into the model. The new element is stereotyped <<*atpStructureElement*>>. This stereotype indicates that the modeled element can be part of structure hierarchies, that it can both aggregate and be aggregated. Classes stereotyped with <<*atpStructureElement*>> combine the “has structure” aspect of <<*atpType*>> and the structuring, feature defining ability of <<*atpPrototype*>>, leaving reusability out.

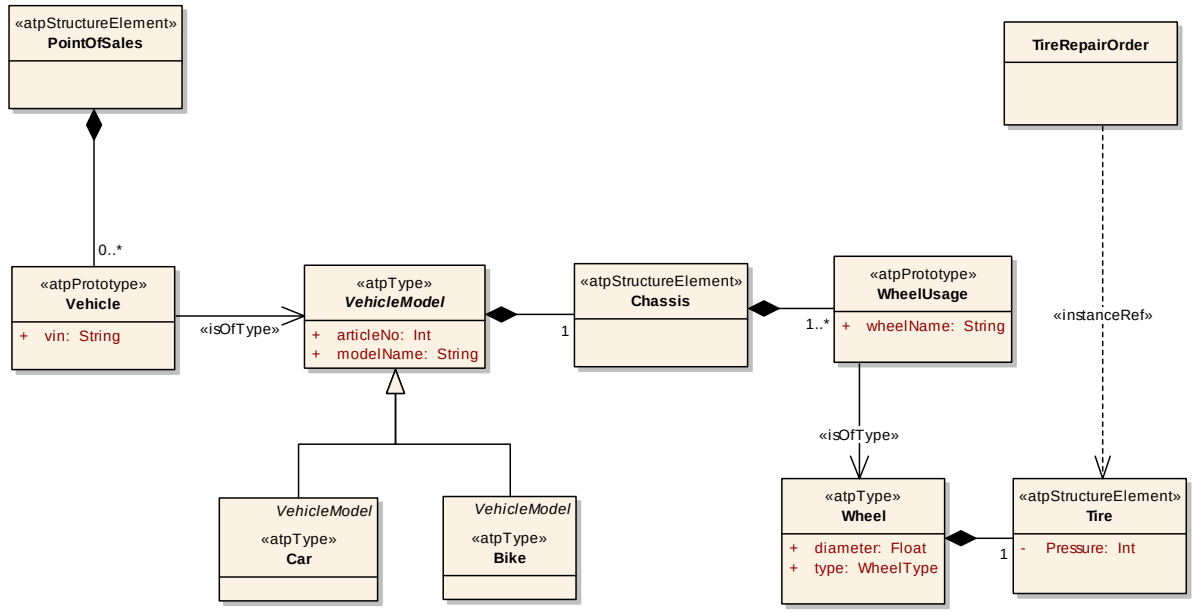

Figure 3-10: Sample model involving structure elements

As already mentioned in Sec. 3.7, the target of an <<*instanceRef*>> association must either be an <<*atpPrototype*>> or an <<*atpStructureElement*>>, and the context path must be composed out of references to <<atpPrototype>>. Another new element in Figure 3-10 is the class *Tire*, stereotyped by <<*atpStructureElement*>> and aggregated by the <<*atpType*>> *Wheel*. The <<*instanceRef*>> *WheelRepairOrder* has been replaced by *TireRepairOrder*. Its target now is *Tire*, an <<atpStructureElement>>. *PointOfSales* is also stereotyped this way.

3.9 Constraints

The model can be annotated with formal and informal constraints as explained in section 6.2.7.

3.10 Metamodel Evolution

In order to allow for evolution of the metamodel without breaking backwards compatibility old concepts SHOULD not be removed immediately. They SHOULD be marked with the tagged value *deprecated* and left in the model until the next major release. The tagged value indicates that these elements SHOULD no longer be used actively by an AUTOSAR tool. However, old data remains valid until the next major release is available. The value of the tagged value SHOULD follow the following rules:

```
value   := “(since ” date “) ” comment
```

For example:

```
deprecated = (since 20.10.2007) Please use ApplicationSoftwareComponent
                                instead.
```

4 Alignment with UML and MOF

In this section the relation of AUTOSAR template modeling and the OMG specifications UML and MOF [3] is explained. AUTOSAR templates use a UML profile that is specified in the next chapter. See *10.1.1 Choice of Metamodeling Mechanism* for a discussion why the UML profile mechanism was chosen.
The complete metamodel hierarchy for AUTOSAR templates is shown in Figure 4-1. Unlike the classical four-layer architecture used by OMG, five metalevels are shown. Starting at the lowest, most concrete metalevel those are:

M0: AUTOSAR objects

This is the realization of an AUTOSAR system at work: real ECUs executing a software image containing for instance the windshield wiper control software.

M1: AUTOSAR models

Models on this metalevel are built by the AUTOSAR end-user (automotive engineers). They may define a software component called “windshield wiper” with a certain set of ports that is connected to another software component and so on. On this level all artifacts required to describe an AUTOSAR system are detailed, including re-usable types as well as specific instances.

M2: AUTOSAR metamodel

On this metalevel the *vocabulary* is defined that later can be used by AUTOSAR end-users. E.g., it is defined that in AUTOSAR we have an entity called “software component” and another entity called “port”. The relation between those entities as well as their semantics is part of such an overall model.

M3: UML profile for AUTOSAR templates

The templates on M2 are built with the metamodel defined on M3. As discussed before this is UML plus a particular UML profile to better support template modeling work. Formally a template on M2 is still an instance of UML, but at the same time the template profile is *applied*, i.e. that additionally rules set out by the stereotypes in the profile need to be observed.

M4: Meta Object Facility

Just for completeness, OMG’s MOF sits on the final metalevel M4. No further metalevels are required since MOF is designed to be reflective.

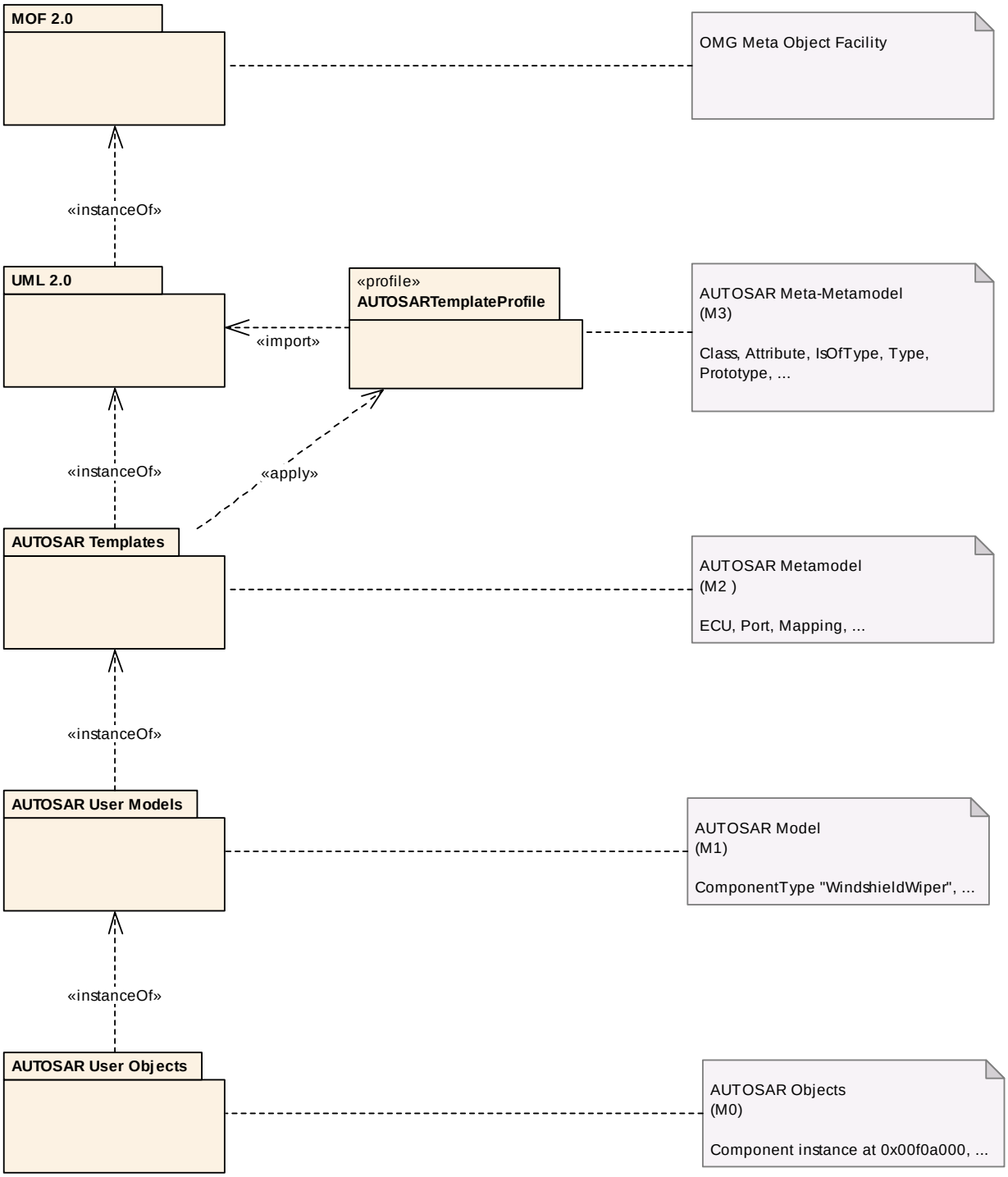

Figure 4-1: AUTOSAR metamodel hierarchy.

5 UML Profile for AUTOSAR Templates

This chapter specifies the UML profile that supports defining AUTOSAR template models.

5.1 Supported Modeling Constructs

Formal metamodels as for example in [1] or [3] are typically given in form of class diagrams. While even class diagrams may contain advanced features like derived associations, powertypes or generalization sets, this specification will limit the available modeling constructs to the smallest subset possible.
It is expected that over time the more involved modeling concepts may be re-enabled in the profile once it has been proven they are required.
The following modeling constructs are supported in the UML profile:

- Meta classes with attributes, but without operations,
- class stereotypes supporting *types* and *prototypes*,
- comments,
- constraints,
- inheritance, single and multiple,
- aggregation,
- aggregation stereotype <<*splitable*>>
- associations,
- association stereotypes <<*splitable*>>, <<*isOfType*>>, <<*instanceRef.context*>> and <<*instanceRef.target*>>,
- dependencies,
- dependency stereotypes <<*import*>>, <<*instanceRef*>>
- packages,
- tagged values,
- primitive types and literals,
- new primitive type *Float*,
- enumerations,
- mixed content.

Not part of regular template models but required by tool vendors:

- Definition of valid subsets, including specification of cardinalities and optional elements.

The semantic meaning of the various modeling entities listed above is explained in the next sections actually defining the UML profile.
Other parts of UML, e.g. use-case or sequence diagrams but also advanced structural features like composite structure diagrams MUST NOT be used, among other things to allow a fairly straightforward mapping into other structural domains like XML or databases. Hence, the profile MUST prevent the usage of those UML constructs.

[ATPS-01]	Template models MUST be expressed in form of UML class diagrams

5.2 UML Profile Specification

This section will show how each of the required modeling constructs listed before is satisfied by the UML profile for AUTOSAR templates.

5.2.1 Classes

The ability to define classes is provided by UML in form of the metaclass *Class*, which is shown in the simplified class diagram in Figure 5-1. In AUTOSAR however, only a subset of the features that are possible with a UML class need to be supported.
To formally limit the template modeler to this subset, a stereotype *atpClass* is introduced. This extension is shown in Figure 5-2.

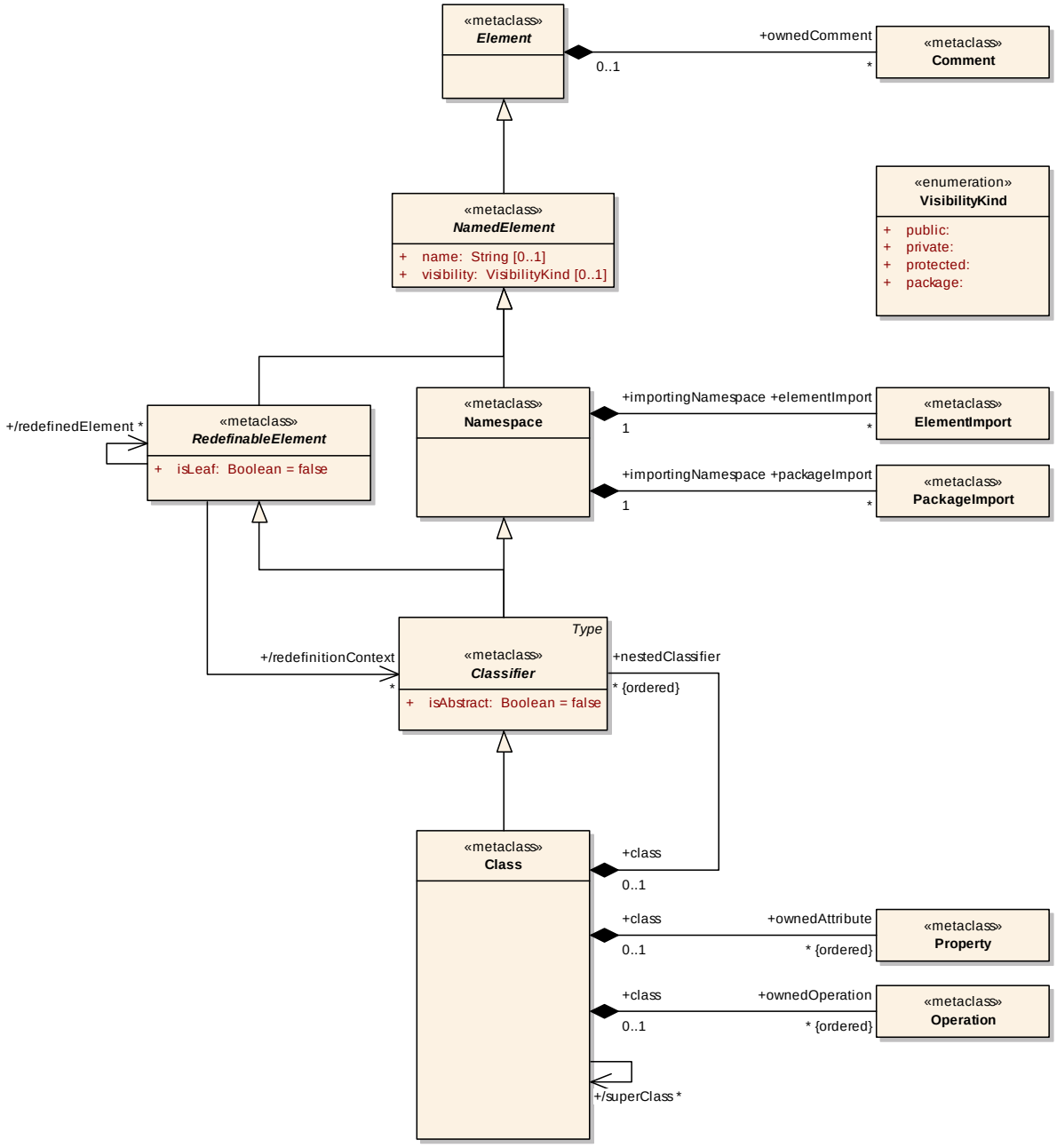

Figure 5-1: Core facilities of *Class* in UML 2 metamodel.

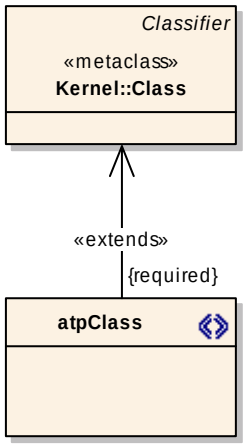

Figure 5-2: Stereotypes *atpClass*.

The stereotype *atpClass* now needs to carry a number of semantic constraints that carefully tailor the metaclass for AUTOSAR. This is done in the following table.

	Feature	Context	Constraint Description	Constraint in OCL (in Context)
1	having a name	`NamedElement`		
2	owning comments	`Element`		
3	visibility	`NamedElement`	All elements in the metamodel MUST be public.	`visibility=public`
4	allowing for redefinition	`RedefinableElement`	Elements MUST NOT be redefined.	`isLeaf=true` `redefinedElement->size()=0`
5	importing packages and elements	`Namespace`	Package imports MUST NOT be used, classes are always visible.	`elementImport->size()=0` `packageImport->size()=0`
6	abstract classes	`Classifier`		
7	owning class attributes	`Class`		
8	owning operations	`Class`	Operations MUST NOT be used.	`ownedOperation->size()=0`
9	generalization	`Class`		

Table 1: Features of UML 2 metaclass *Class*.

The table has the following columns:

Feature
The particular ability that is provided by the metaclass.

Context
Name of the class providing the aforementioned feature, either the UML 2 class in discussion or one of its base classes.

Constraint Description
A textual description of what is constrained.

Constraint in OCL
A formal expression in OCL [5] – a formal constraint language – limiting the feature or possibly removes it altogether.

It needs to be noted that the feature “owning class attributes” does not yet include associations and aggregations. Those are discussed further down.

[ATPS-07]	All elements in model MUST be public.
[ATPS-08]	Redefinition of model elements MUST NOT be used.
[ATPS-09]	Packages imports MUST NOT be used to put model elements in scope.
[ATPS-11]	Classes MUST NOT have operations.

5.2.2 Class Attributes

So far it has only been defined that a class may have attributes (the *Properties* of UML 2). Now their particular capabilities need to be analyzed and tailored in the way this has been done above for classes.
In UML 2 this capability is provided through metaclass *Property*, which is shown in Figure 5-3.

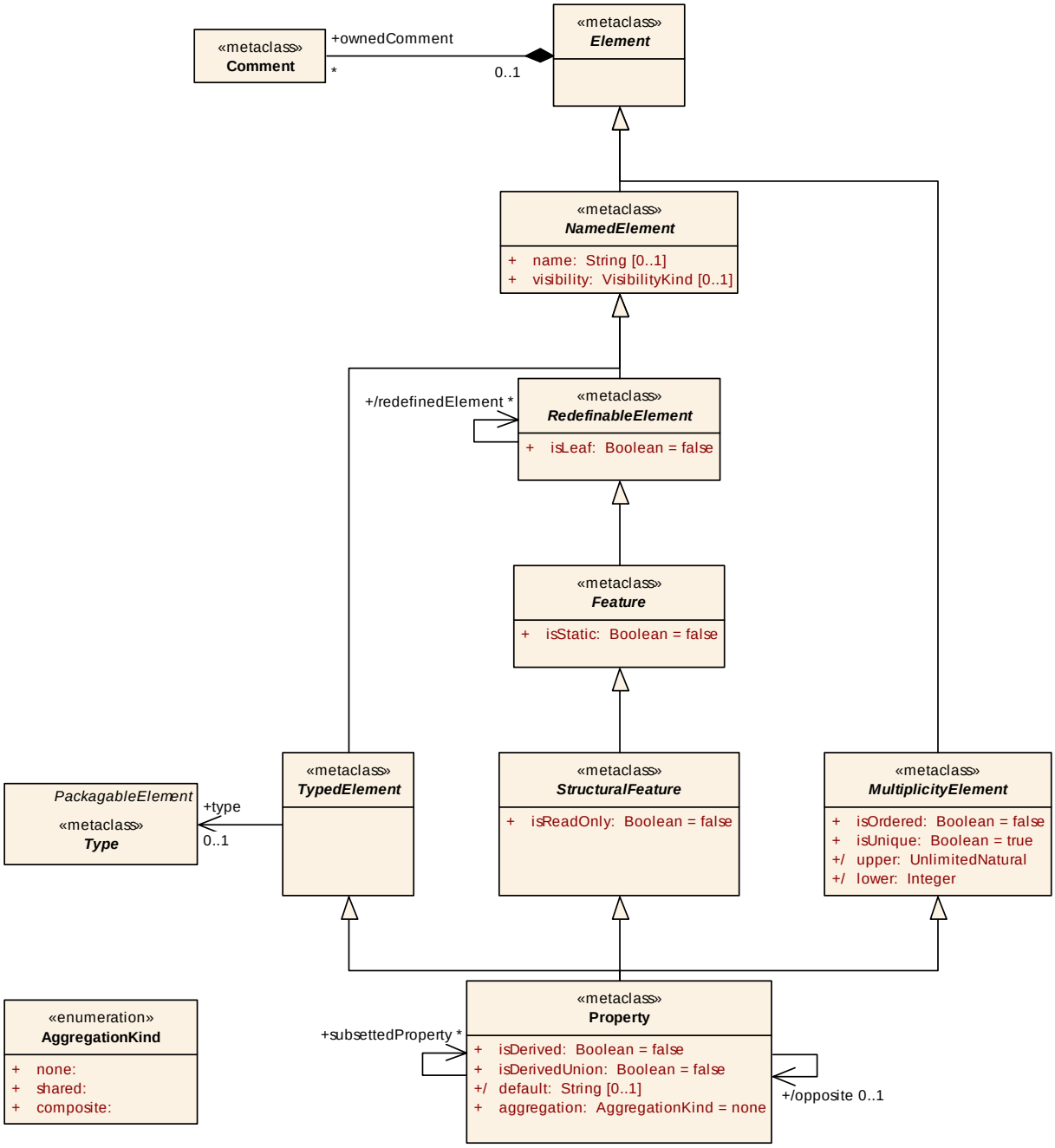

Figure 5-3: UML 2 metaclass *Property*.

For use in AUTOSAR some limitations need to be put on UML properties. Figure 5-4 shows the corresponding stereotype, which again is tagged *{required}*, i.e. it MUST be used instead of *Property* in AUTOSAR template models.

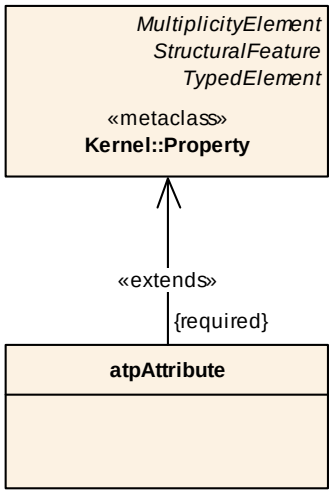

Figure 5-4: *atpAttribute* extends UML 2 metaclass *Property*.

The constraints in Table 2 are attached to the new stereotype *atpAttribute*. Note that we also have a constraint for a feature that *is* desired in the profile, but still needs to be limited in terms of allowed values (*aggregation*).

	Feature	Context	Constraint Description	Constraint in OCL (in Context)
1	having a name	`NamedElement`		
2	owning comments	`Element`		
3	visibility	`NamedElement`	All elements in the metamodel MUST be public.	`visibility=public`
4	allowing for redefinition	`RedefinableElement`	Elements MUST NOT be redefined.	`isLeaf=true` `redefinedElement->size()=0`
5	static (class level) properties	`Feature`	Properties MUST NOT be defined static.	`isStatic=false`
6	properties can be typed	`TypedElement`	In AUTOSAR, attributes MUST be typed.	`type->size()=1`
7	properties can be read only (fixed attributes)	`StructuralFeature`	Properties MUST be writeable.	`isReadOnly=false`
8	ordered multiplicity	`MulitplicityElement`		
9	uniqueness of property	`MulitplicityElement`	An object MUST NOT be part of a multivalued property more than once, i.e. all properties are always unique.	`isUnique=true`
10	specification of upper and lower boundaries of multiplicity	`MulitplicityElement`		
11	derived properties	`Property`	Derived properties MUST NOT be given in template model.	`isDerived=false` `isDerivedUnion=false`
12	default values	`Property`	Default values MUST NOT be used.	`default->size()=0`
13	aggregation kinds: o none o shared o composite	`Property`	Due to imprecise semantics the aggregation kind *shared* MUST NOT be used.	`aggregation=none or` `aggregation=composite`
14	subsetting a property	`Property`	Property subsets MUST NOT be defined.	`subsettedProperty->size()=0`
15	definition of opposite property	`Property`	An opposite MUST NOT be defined.	`opposite->size()=0`

Table 2: Features of UML 2 metaclass *Property*.

[ATPS-07]	All elements in model MUST be public.
[ATPS-08]	Redefinition of model elements MUST NOT be used.
[ATPS-13]	Template class attributes and association roles MUST NOT be defined *static* (UML default).
[ATPS-14]	Regular class attributes MUST be typed. The only possible exception is when modeling enumerations.
[ATPS-15]	Attributes MUST NOT be defined *readonly* (UML default), i.e. no fixed attributes are allowed.
[ATPS-16]	Attributes MUST be defined *unique* (UML default.
[ATPS-17]	Attributes MUST not be derived from other attributes.
[ATPS-18]	Default values MUST NOT be defined for attributes.
[ATPS-25]	Aggregations MUST either be of type *none* or *composite*. Aggregation type *shared* MUST NOT be used.
[ATPS-26]	Attribute subsets MUST NOT be used.
[ATPS-27]	Opposite attributes MUST NOT be defined

5.2.3 Mixed Content

If a model requires to describe documentation like information, it often will need to mix formal content and text. An example of such a model is HTML: markup of formal information bits is mixed into regular text, as shown in the following sample[4]:

```
[…]meet    <a    href="/wiki/Runtime"    title="Runtime">runtime</a>
requirements of automotive devices[…]
```

The sample indicates correctly, that this feature is well known from the XML world, where it is called *mixed content*[5].
The following list indicates the features of mixed content from a modeling point of view. Within a mixed content instance

- a set of formally defined model elements may appear an arbitrary number of times in arbitrary order,
- but the actually present order is relevant in terms of semantics of the whole object, and
- unqualified text may be mixed in between any of formally defined elements.

This mechanism is supported in AUTOSAR through stereotypes <<*atpMixedString*>> and <<*atpMixed*>>. The latter stereotype does not allow for mixed-in text, but keeps the definition of order in terms of syntax and semantics.
A mixed content class will aggregate or reference a number of other classes in a template model. The target cardinalities of those relations are typically 1, since the overall number of occurrences is arbitrary by definition.
If, however, multiplicity is different from 1, a required grouping is specified. E.g. if the target multiplicity is 2, always a pair of those objects (not just a single object) must be put into the mixed content, and so on.

[4] Taken from Wikipedia's description of AUTOSAR: http://en.wikipedia.org/wiki/AUTOSAR.
[5] E.g. see http://www.w3schools.com/schema/schema_complex_mixed.asp.

The concepts introduced above MAY be used in a template model. Alternatives, typically including ordered references to an abstract base class, MAY also be used, if applicable.

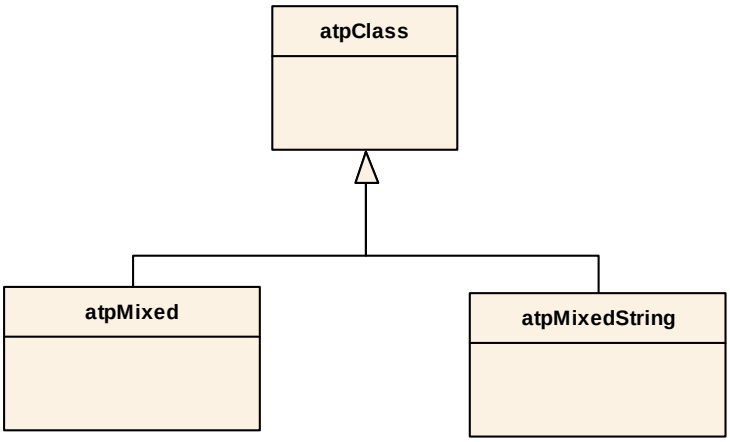

Figure 5-5: Definition of stereotypes for mixed content in a template model.

5.2.4 Types, Prototypes, and Structure Elements

AUTOSAR follows a strict concept of reusability and structure. Certain metamodel elements are designed to be explicit *type* declarations, which means M1 instances of those classes (like a *Windshieldwiper* software component) are reusable definitions of structured elements.

Types can then be put to use in the form of roles AKA occurrences AKA prototypes (from MSR [20]) in the context of other types. Compared to UML, prototypes are very similar to UML 2 *Properties*: they are typed elements aggregated by a type. The main difference is that AUTOSAR prototypes are so-called *First Class Objects* (FCOs) [21], meaning that they can carry additional M2 properties (attributes or association ends) if required.

If an instance of a prototype is created its allowed “values” are constrained by the used type (see definition of *TypedElement* in [1]). In terms of attributes this means that attributes of a type’s instance specify which attributes values are possible for a prototype instance[6].

The fact that types in AUTOSAR may aggregate other elements make them structured entities. In this respect they are similar to UML’s StructuredClassifier (see [1]). But unlike UML, where a structured classifier is necessarily a type, i.e. reusable, AUTOSAR allows for non-reusable elements in structure hierarchies, called *Structure Elements*. A structure element may be thought of as a simultaneous definition of a type an the single role it can play. Structure elements may be used in an M2 metamodel wherever either a type or a prototype may be used.

The related stereotypes, *<<atpType>>*, *<<atpPrototype>>, and <<atpStructureElement>>* extend the UML 2 metaclass *Class* as shown in Figure 5-6. To better categorize the elements at play, two more M3 abstract metaclasses

[6] Note that type and prototype are constructs on M3, i.e. instances of those concepts exist on M2, e.g. in form of template model elements.

are used: <<*atpStructured*>> and <<*atpStructuringFeature*>>. The former corresponds to elements which have structure and the latter to those that give structure. <<*atpType*>> has structure and is reusable. <<*atpPrototype*>> gives structure and must be typed by an <<*atpType*>>. <<*atpStructureElement*>> may both have and give structure but is not reusable.
These stereotypes make it possible to characterize the target and context of <<*instanceRef*>> associations as follows:

- The target of an <<instanceRef>> must be stereotyped with a subclass of <<*atpStructuringFeature*>>.
- The context of an <<instanceRef>> must be stereotyped with <<atpPrototype>>
- The first element of the context must be aggregated by an element stereotyped by a subclass of <<*atpStructured*>>.

Intuitively it means that in M1, an <<*instanceRef*>> expresses a reference to a nested feature (an <<*atpStructuringFeature*>>) in some containing structured element (an <<*atpStructured*>>). Given an M0 instance of the latter, the former identifies a part of it, i.e. a sub-instance within it.

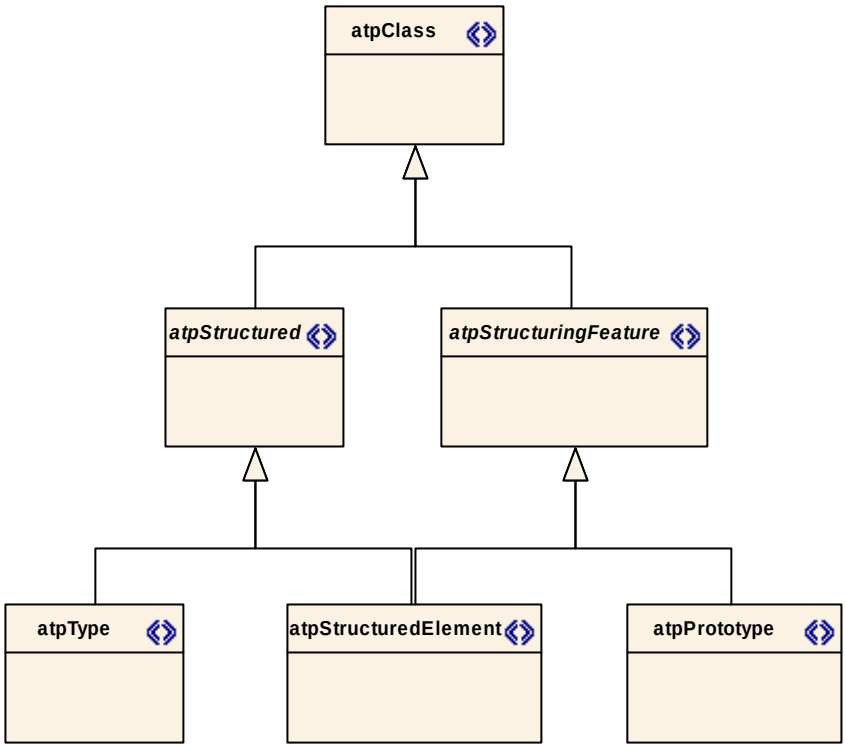

Figure 5-6: New stereotypes *for types and prototypes.*

Note that the stereotype *atpClass* is not abstract classes, which means the profile allows defining metaclasses that are deliberately chosen to be non of type, prototype, or structure element[7].
The constraints related to types, prototypes, and structure elements are not given here, as they are not simple restrictions with respect to what regular classes can do. Instead sections 5.2.6 and 5.2.7 present two new stereotypes for associations that result in constraints on *atpType*, *atpPrototype,* and *atpStructureElement*.

[7] Another way of modeling would be to allow prototypes without type reference, as done for UML's *TypedElement*.

[ATPS-28]	For template classes representing a type, stereotype <<*atpType*>> MUST be used.
[ATPS-29]	For template classes representing a prototype, stereotype <<*atpPrototype*>> MUST be used.

5.2.5 Associations

Classes in the metamodel need to be related to each other. This is done through associations. For AUTOSAR template models we allow only for two kinds of binary associations:

(a) Composite aggregation, forming a whole-part relationship, and

(b) Regular association, expressing a reference from the associating to the associated model element.

The features of the original UML 2 metaclass *Association* are show in Figure 5-7.

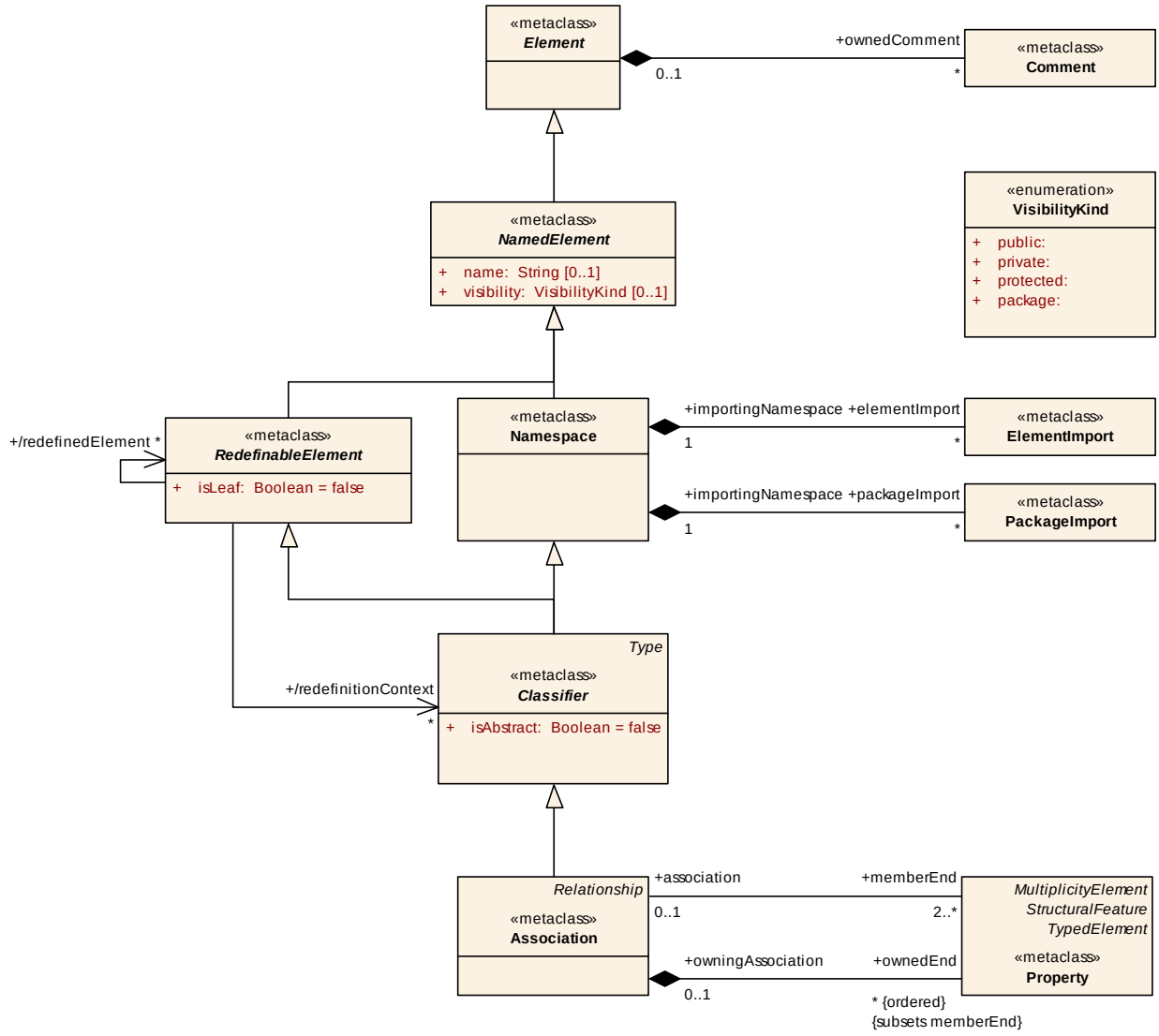

Figure 5-7: Features of the UML 2 metaclass *Association*

While in the UML Diagrams section in the beginning of this specification it was stated that no *subset* clauses are shown for simplification reasons, the diagram above does contain such an annotation, since it is crucial to distinguish the subset and the subsetting roles:

- *Association.memberEnd* holds all participants of the association, at least two for a binary association, three for a ternary and so on. In AUTOSAR we only allow binary associations.
- *Association.ownedEnd* holds those participants among the set given already in *memberEnd* that additionally are owned by the association. This means they are not owned by one of the associated classes and are therefore not navigable from there. In AUTOSAR there is always exactly one such end: the aggregating part in case of a composite aggregation and the referring part in case of a regular (non-aggregating) association.

The aforementioned limitations and other constraints are enforced when the required stereotype *atpAssociation* is used, shown in Figure 5-8.

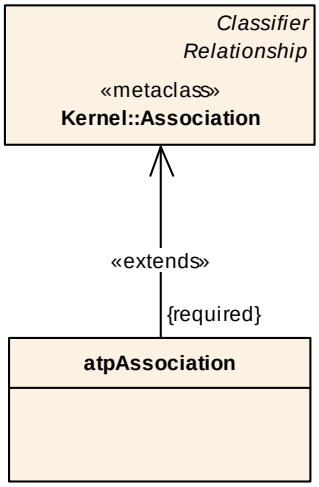

Figure 5-8: New stereotype *atpAssociation*.

The following table lists the required constraints on the new stereotype:

	Feature	Context	Constraint Description	Constraint in OCL (in Context)
1	having a name	`Element`	Associations MUST NOT carry a name.	`name.size()=0`
2	owning comments	`Element`		
3	visibility	`NamedElement`	All elements in the metamodel MUST be public.	`visibility=public`
4	allowing for redefinition	`RedefinableElement`	Elements MUST NOT be redefined.	`isLeaf=true` `redefinedElement->size()=0`
5	importing packages and elements	`Namespace`	Package imports MUST NOT be used, classes in all packages are always visible from all other packages.	`elementImport->size()=0` `packageImport->size()=0`
6	abstract associations	`Classifier`	Associations in the metamodel MUST be concrete.	`abstract=false`
7	having at least two ends	`Association`	Associations MUST be binary (the associating and the associated class).	`memberEnd->size()=2`
8	types of association ends	`Association`	The navigable end of an association MUST be of type identifiable.	`(memberEnd-ownedEnd)->forAll(oclIsKindOf(identifiable))`

9	having non-navigable ends	`Association`	Associations MUST have exactly one non-navigable end (the associating/ aggregating class).	`ownedEnd->size()=1`

Table 3: Constraints on *atpAssociation*.

[ATPS-07]	All elements in model MUST be public.
[ATPS-08]	Redefinition of model elements MUST NOT be used.
[ATPS-09]	Packages imports MUST NOT be used to put model elements in scope.
[ATPS-30]	Associations MUST not be named.
[ATPS-31]	Associations MUST be binary.
[ATPS-32]	Associations MUST have exactly one non-navigable end.

5.2.6 IsOfType Association

As explained before, classes in an M2 template model can be categorized as reusable type definitions or as using prototypes within a certain context. While each type MAY be used in form of a prototype an arbitrary number of times, every such prototype MUST refer to exactly one type. This association from prototype to type is very special in terms of semantics: the type defines the attributes and M2 semantics of the prototype, much like the datatype *Integer* defines the value range of a variable *a* of type *Integer*.
To clearly show this kind of relation in the model, a new stereotype <<*isOfType*>> is introduced in the UML 2 profile. First, the simple example in Figure 5-9 shows the M2 usage of the stereotype, here between two classes taken from [18], also showing the required cardinalities.

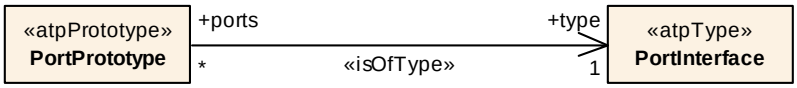

Figure 5-9: Example of using stereotype <<*isOfType*>> on M2.

On M3, the metalevel of the UML 2 Profile for AUTOSAR templates, this is enabled through the formal introduction of the new stereotype and its associated constraints.

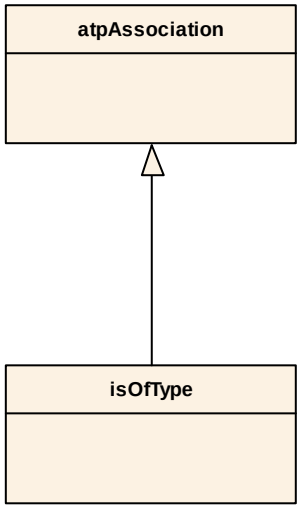

Figure 5-10: Introduction of new stereotype *<<IsOfType>>*.

IsOfType is specializing *atpAssociation*, i.e. the constraints put on this formerly defined stereotype also apply. In addition the following constraints MUST be satisfied:

Constraint Description	Constraint in OCL
The non-navigable (source) association end MUST be an *atpPrototype* (a type does not know about its using prototypes).	`-- Note: atpAssociation allows only ownedEnd->size() = 1.` `ownedEnd->forAll( c | c.oclIsKindOf( atpPrototype ) )`

The navigable (target) association end MUST be an *atpType* (it is possible to navigate from prototype to type).	`-- Note: atpAssociation restricts Association to be binary`
	`--  already. This means the following statement essentially`
	`--  evaluates to: 2 – 1 = 1.`
	`memberEnd->size() – memberEnd->count( atpType ) = ownedEnd->size()`

There MUST be exactly one type referred (by this association in general and by an *atpPrototype* in particular).	`( memberEnd – ownedEnd )->forAll( p | p.lower=1 and p.upper=1 )`
	`context atpPrototype inv:`
	`  ownedAttribute.association->count(IsOfType)=1`

Table 4: Additional constraints for *IsOfType*.

[ATPS-33]	Every prototype MUST refer to its defining type via an association typed *<<isOfType>>*.
[ATPS-34]	The *<<isOfType>>* association MUST be used only to refer from an *atpPrototype* to an *atpType*.
[ATPS-35]	The target cardinality of an *<<isOfType>>* association MUST be one.

5.2.7 IndexedRef Association

The stereotype <<indexedRef>> is used for references to elements that may occur s times in an ordered sequence on M0

5.2.8 InstanceRef Association

If the AUTOSAR template model makes use of reusable type definitions, simple references are often no longer sufficient. References that contain information about

the target within the reusable type and the context in which the type is actually used are required.
Example: The AUTOSAR template model defines a *SenderReceiverInterface* as a type of *PPortPrototype*. In order to reference a *DataElementPrototype* that is communicated via the *PPortPrototype*, the reference needs information about

- the *DataElementPrototype* (the target) within a *SenderReceiverInterface* and
- the *PPortPrototype* (context) which is typed by that *SenderReceiverInterface* .

Those references are supported in AUTOSAR template models through two different representations: a detailed and a short representation: The detailed representation describes the target and the context of the reference. The short representation shows the target only. The detailed representation is mandatory whereas the short representation is optional.

5.2.8.1 Detailed Representation of InstanceRef Association

The support for modeling detailed representations of instanceRef associations could be realized using n-ary associations (associations with more than two association ends, see Figure 5-11).

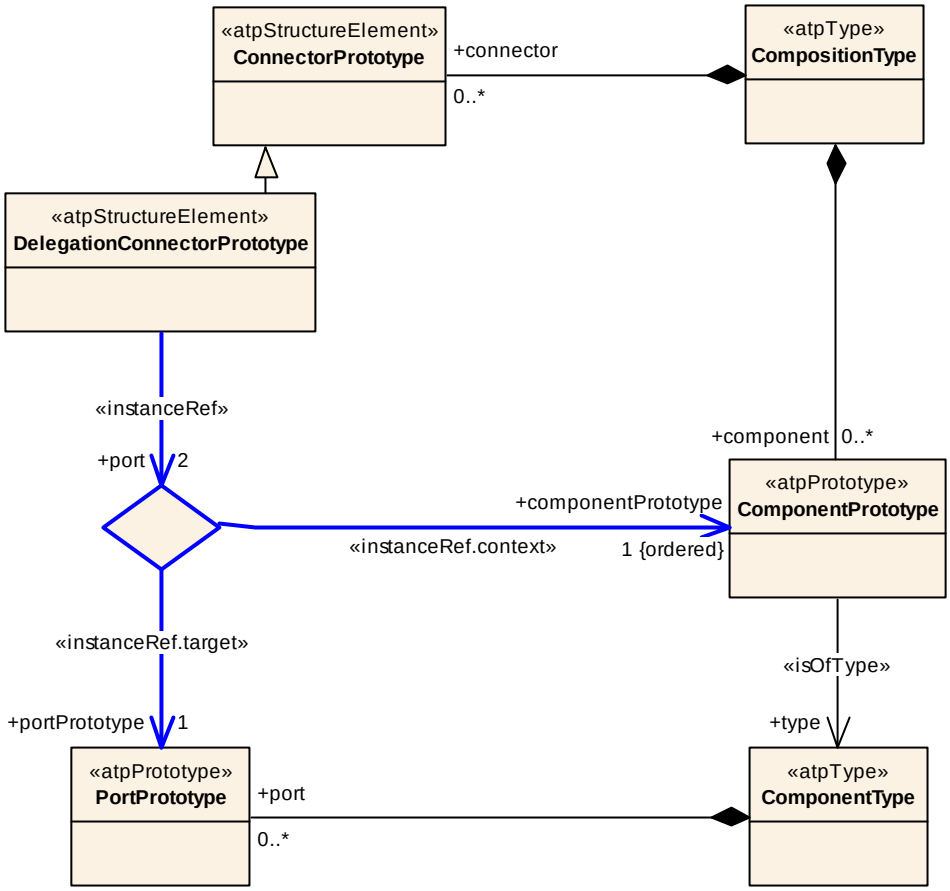

Figure 5-11: Example of detailed modelling of an instanceRef association using a 4-ary association (not supported by AUTOSAR template profile)

The AUTOSAR template profile limits itself to modelling concepts that are supported by OMG MOF[3]. Therefore n-ary associations are replaced by a metaclass which represents the n-ary associations.
Figure 5-12 shows an example of a detailed representation of an instanceRef association in the AUTOSAR metamodel (M2).

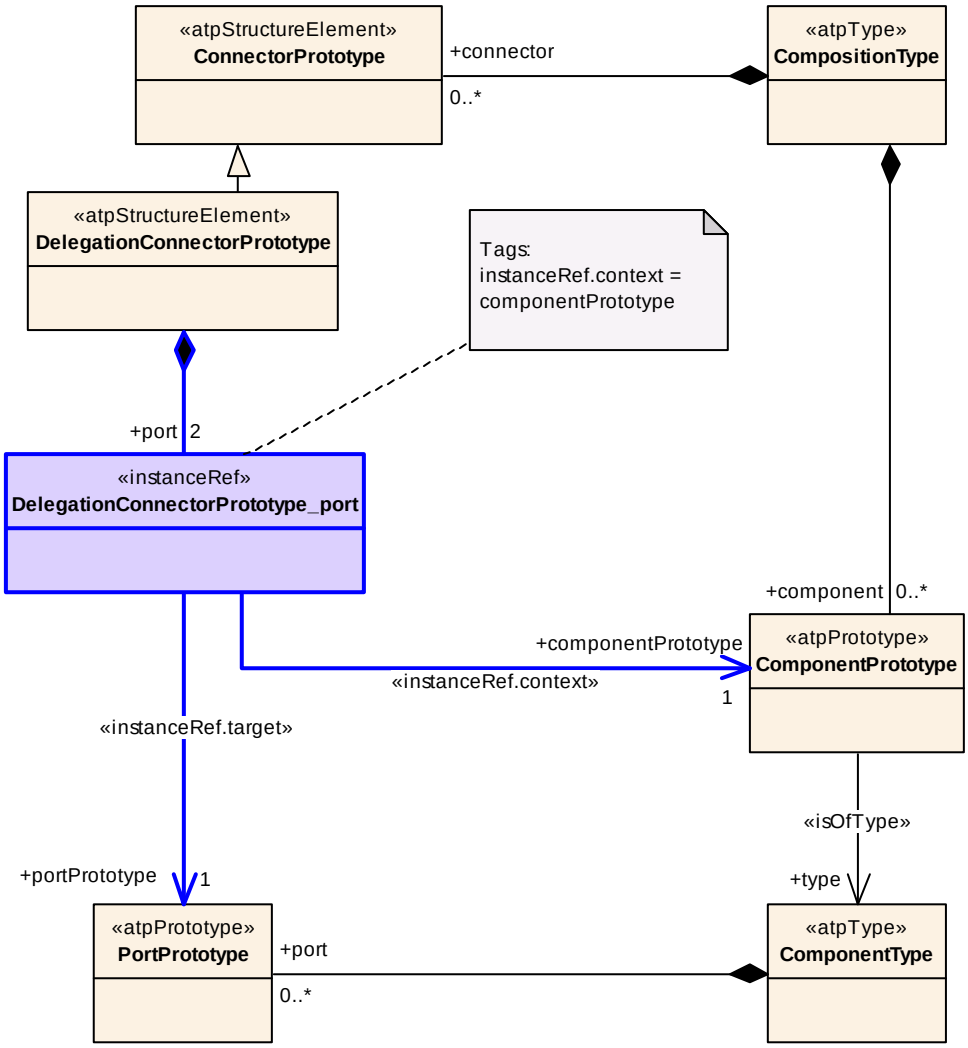

Figure 5-12: Example of detailed modeling of an instanceRef association using a meta class that owns references to the target and the context

Some prototypes or structure elements have a *multiplicity* defined on them via an dedicated attribute stereotyped <<atpMaxMultiplicity>>. When the multiplicity is ordered it is possible to use such prototypes in context paths or as target via indexing. To allow for that, an intermediate metaclass stereotyped <<indexedRef>> is introduced between the <<instanceRef>> metaclass and the context or target element.
An <<indexedRef>> encapsulates a direct, indexed reference to a prototype or structure element. The <<instanceRef.context>> and <<instanceRef.target>> associations coming from an <<instanceRef>> class then point to this new kind of

metaclass rather than directly to the prototype or structure element itself. In addition those two associations become aggregations from the <<instanceRef>> class. Figure 5-13 shows an instanceRef which can point to a particular element in a chain of nested arrays. The <<indexedRef>> metaclass should be used when and only when the prototype or structure element is in fact a multiplicity element.

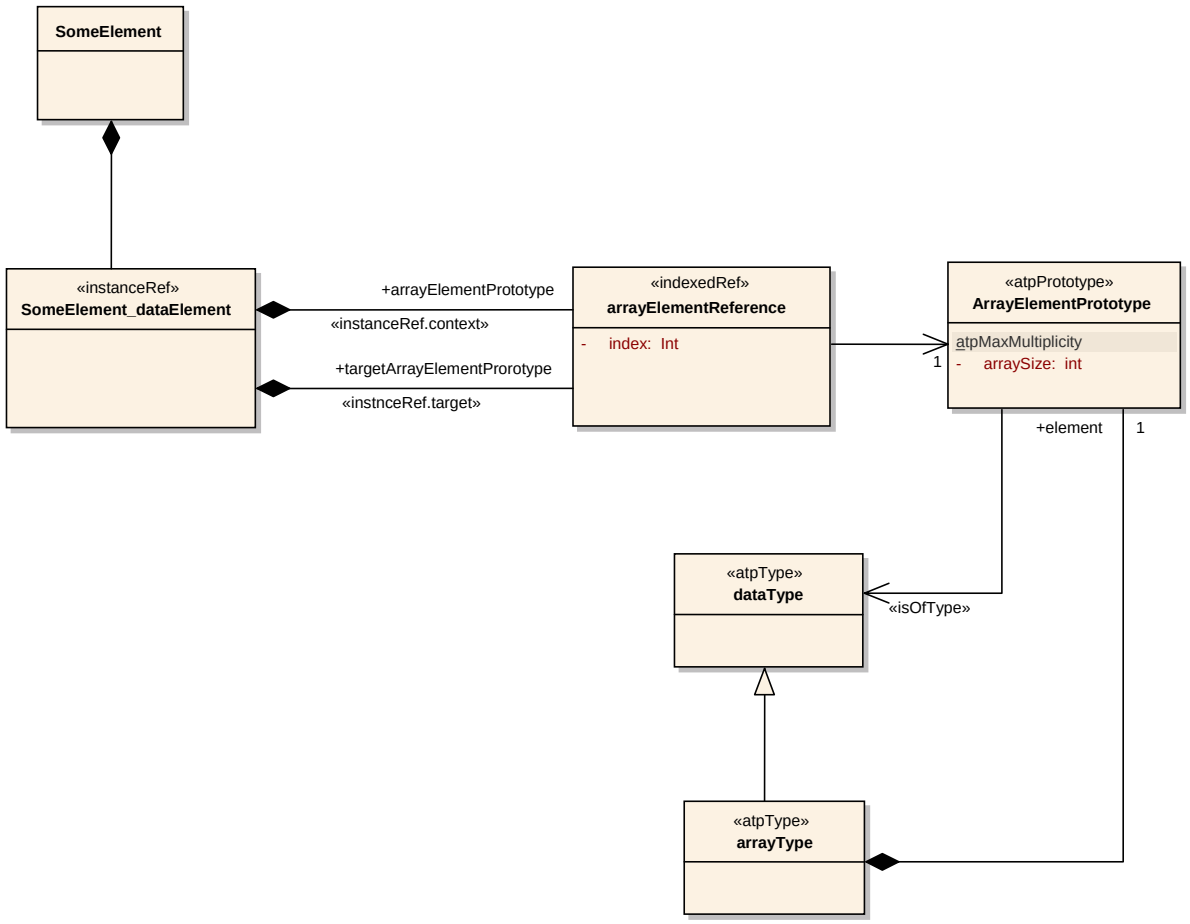

Figure 5-13 - InstanceRef with a multiplicity prototype

The detailed representation of an instanceRef is characterized by the following rules:

1. The instanceRef association is represented by a metaclass with stereotype <<instanceRef>>.
    a. This instanceRef metaclass is aggregated by the source metaclass of the instanceRef association.
    b. The role and multiplicity of the referenced model element are specified on the aggregation between the source and the instanceRef metaclass.
    c. The name of the instanceRef metaclass is derived from the name of the source metaclass and the role of the aggregation:
    <name of instanceRef metaclass> :=
    <name of source metaclass> + "_" + <role of instanceRef association>
    d. The instanceRef metaclass is contained in a package called "_instanceRef". This package is contained in the package that contains the source of the instanceRef association.

2. The target of the instanceRef association is described by a reference from the instanceRef metaclass to either the target metaclass directly or to an aggregated intermediate metaclass stereotyped <<indexedRef>> which in turn has a reference to the target metaclass.

    a. If the target metaclass does not have an <<atpMaxMultiplicity>> attribute then the stereotype <<instanceRef.target>> is used to reference the target metaclass directly.
    b. If the target metaclass has an <<atpMaxMultiplicity>> attribute then the stereotype <<instanceRef.target>> is used to aggregate an intermediate metaclass stereotyped <<indexedRef>> which in turn has a reference with multiplicity 1 to the target metaclass.
    c. The role name of the <<instanceRef.target>> reference in either case is derived from the target metaclass. However the first character of the role name should be lower case:
    <role name> :=
    firstCharacterLowerCase (<referenced metaclass name>)
    d. If this role name conflicts with a role name defined by context references (see 3) then the role name is defined by prefixing the referenced metaclass name by “target”.
    <role name> := “target” + <referenced metaclass name>
    e. The target metaclass must be stereotyped by a subclass of <<atpStrucuringFeature>>, i.e. by an <<atpPrototype>> or an <<atpStructureElement>>.
    f. The multiplicity of this reference is 1.

3. The context of the instanceRef association is described by references from the instanceRef metaclass to either the context metaclass directly or to an aggregated intermediate metaclass stereotyped <<indexedRef>> which in turn has a reference to the context metaclass.

    a. If the context metaclass does not have an <<atpMaxMultiplicity>> attribute then the stereotype <<instanceRef.context>> is used to reference the context metaclass directly.
    b. If the context metaclass has an <<atpMaxMultiplicity>> attribute then the stereotype <<instanceRef.context>> is used to aggregate an intermediate metaclass stereotyped <<indexedRef>> which in turn has a reference with multiplicity 1 to the context metaclass.
    c. The role name of the <<instanceRef.context>> reference in either case is derived from the context metaclass. However the first character of the role name should be lower case:
    <role name> :=
    firstCharacterLowerCase (<referenced metaclass name>)
    d. The context metaclasses must be stereotyped by <<atpPrototype>> (see also 5.2.8.4)
    e. The multiplicity of the reference depends on the structure of the context. See section 5.2.8.4 for more details. If the upper multiplicity of a reference is bigger than one, then the keyword **ordered** must be indicated.
    f. If there is more than one context reference then the tagged value “instanceRef.context” must be given a value specifying an order between the referenced elements. It lists, in some order, all the *roles* by

which context elements are referenced, separated by spaces. In order to improve the readability the value must contain additional information on the multiplicity of the roles:

i. multiplicity = 1 then no additional symbol required
ii. multiplicity = 0..1 then the symbol '?' is added
iii. multiplicity = 0..* the the symbol '*' is added
iv. multiplicity = 1..* the the symbol '+' is added

g. The value of instanceRef.context forms a regular expression which can be used to check the correctness of instanceRef contexts. The tool needs to concatenate the rolenames within the context in the order as they are stored in XML or in the tool. Applying the regular expression validates the context.

5.2.8.2 Short Representation of InstanceRef Association

In addition to the detailed representation a short representation may be used. The short representation can be derived from the detailed representation by the following algorithm (Figure 5-14 shows the short representation of the instanceRef association described in
Figure 5-12):

1. The short representation of an instanceRef association is represented by a dependency, stereotyped by <<instanceRef>>.
   a. The multiplicity of this instanceRef dependency equals the multiplicity of the aggregation between the source metaclass and the instanceRef metaclass.
   b. The name of the instanceRef dependency equals the name of the aggregation between the source metaclass and the instanceRef metaclass.

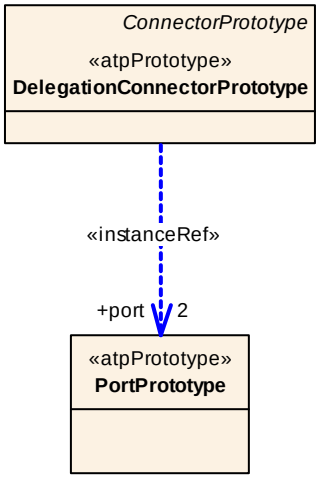

Figure 5-14: Example of the short representation of an instanceRef association.

class with the stereotype *<<instanceRef>>.*

5.2.8.4 Constraints on instanceRef associations

The following constraints define rules for the existence, correctness and completeness of instanceRef associations. If the following constraints are violated then the instanceRef association is not valid. The following sections formally describe these constraints.

Basic relations
Let's define some basic relations on metaclasses that are based on relations defined in the AUTOSAR metamodel:

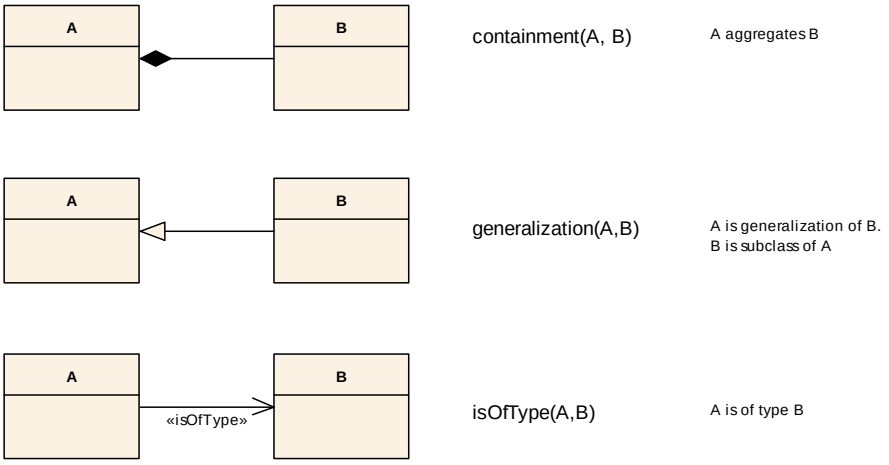

Figure 5-15: Basic relations

Direct context relation
The direct context of a metaclass is defined by metaclasses that are reachable

- by navigating a containment aggregation towards the containing metaclass
- by navigating a generalization towards the baseclass or
- by navigating a <<isOfType>> reference towards its source

The following relation defines if one metaclass *A* is in the direct context of another metaclass *B*:

directContext (A, B) := containment (A, B) ***OR***
generalization (A, B) ***OR***
isOfType(A, B)

Full context paths
The set of all possible full context paths beginning at the metaclass *B* is defined by the following expression. A full context path is defined by navigating the metamodel using the relations defined by the directContext relation. Within the full context path at least one isOfType relation must be navigated.

fullContextPaths (B) :=
{ (X_1, X_2, …,X_{n-1}, X_n) | X_1, X_2, …,X_{n-1}, X_n elementOf Metaclass
AND directContext(X_n, X_{n-1})

....
AND directContext(X_2, X_1)
AND X_n == B
AND
(***exists*** X_a, X_b ***elementOf*** { X_1, X_2, …,X_{n-1}, X_n } | isOfType(X_a, X_b))
}

Note that while each path is finite, the set fullContextPaths(B) may be infinite. This will be the case when the graph defined by the direct context relation contains a cycle.

InstanceRef context paths

In order to unambiguously describe the full context path of an instanceRef it is not required to list all classes within the path. Instead it is sufficient to only list the references to <<atpPrototype>>s within the path. This more compact representation of context paths is described by the expression *contextPaths(B)*.

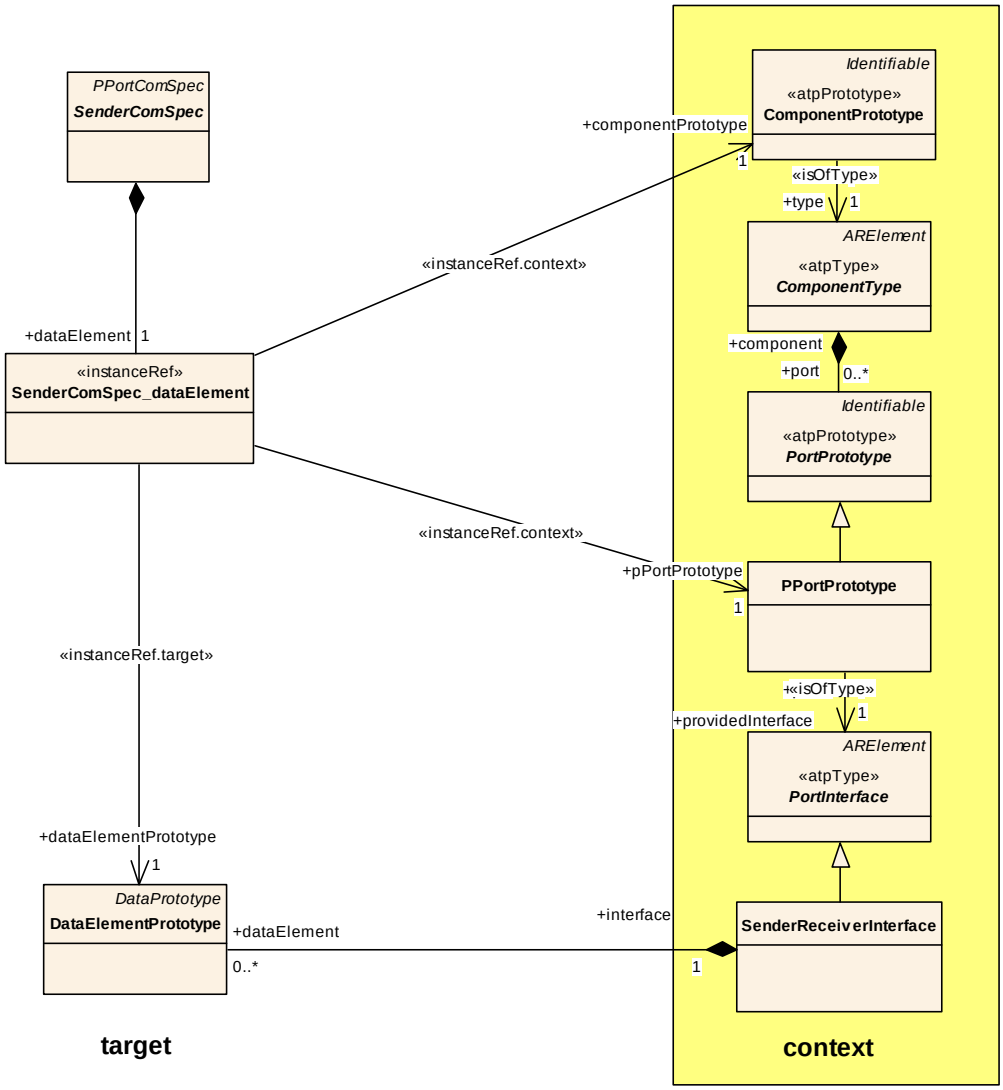

Figure 5-16: Example of a context path of DataElementPrototype

Figure 5-16 shows an example of a contextPath of the metaclass *DataElementPrototoype.* In order to unambiguously describe the contextPath of the instanceRef association from SenderComSpec to *DataElementPrototype* it is sufficient to identify the *PPortPrototype* and the *ComponentPrototype.*

Constraints

A given detailed M2 representation of an instanceRef association is valid if the following constraints hold.

1. *Completeness:* The set of context references is complete if it can describe a valid context path.
   - **Exists** (X_1, X_2, …,X_{n-1}, X_n) **elementOf** contextPaths(target) |
     ForAll Xi **Exists** context reference from instanceRef metaclass to Xi

   Note that this condition implies that contextPaths(target) is nonempty.
3. *Multiplicities*: The multiplicity of each context reference SHALL NOT exceed the maximum occurrence of the referenced prototype within possible context paths.
4. *Order*: Not all sequences of a complete set of context references are allowed. The value of the tagged value “instanceRef.context” must describe a path or a set of paths which is contained in contextPaths(target).
   - Paths as described by “instanceRef.context”
     subsetOf
     contextPaths(target)

Example

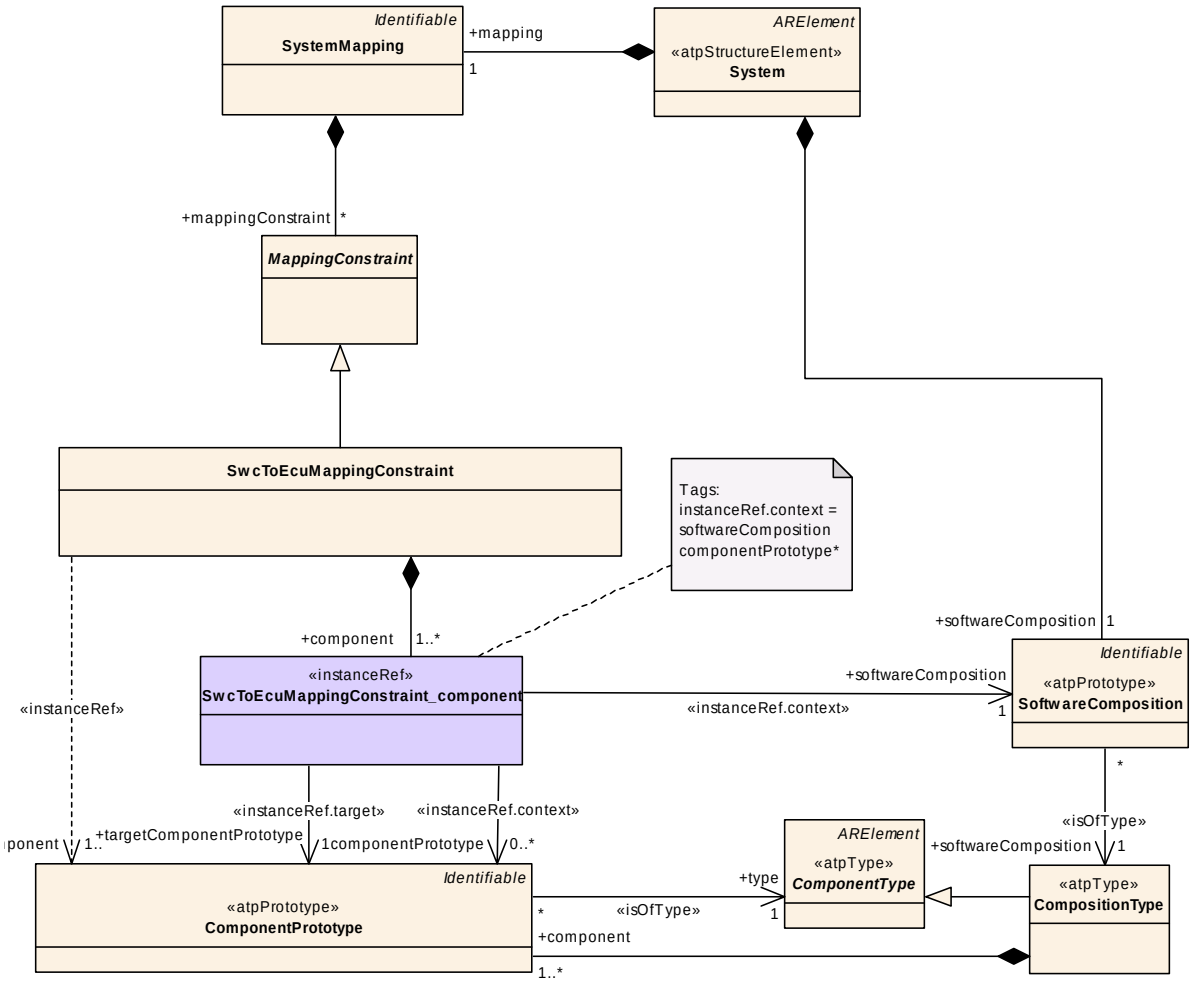

Figure 5-17: Example of detailed and short modelling (see below) of an InstanceRef association

Figure 5-17 shows the detailed and short representation of an instanceRef association. This instanceRef association points from the outside to a *ComponentPrototype* which might be deeply nested in the type hierarchy of *CompositionTypes*. The modeled instanceRef association is valid because:

1. Completeness
    a. Modeled context references:
        i. *componentPrototype* with multiplicity *
        ii. *softwareComposition* with multiplicity 1
    b. Valid context paths:
    contextPaths(*ComponentPrototype*) =
    { (*SoftwareComposition),*
    (SoftwareComposition, ComponentPrototype),
    (*SoftwareComposition, ComponentPrototype, ComponentPrototype),*
    … }
    c. The modeled instanceRef association requires at least one context reference to a *SoftwareComposition* and zero or more context references to *ComponentPrototypes.* Context reference paths described by the

instanceRef association are a subset of contextPaths (ComponentPrototype).

2. Multiplicities:
   Multiplicities of context references do not exceed the maximum occurrence of referenced prototypes.
3. Order:
   The tagged value *instanceRef.context* defines the order as *softwareComposition componentPrototype*
   This doesn't allow for creating sequences of context reference that are not covered by the valid paths as defined in section "completeness".

5.2.9 Constraints

Constraints are basically taken as they are defined in [1], with one additional limitation: constraints in an AUTOSAR template model MUST be expressed in OCL or another supported language, mainly to support automatic evaluation.
Formally this is once again provided by introducing a new stereotype *atpConstraint*. Figure 5-18 shows in a simplified diagram how constraints are defined in UML 2 while Figure 5-19 depicts the new stereotype.

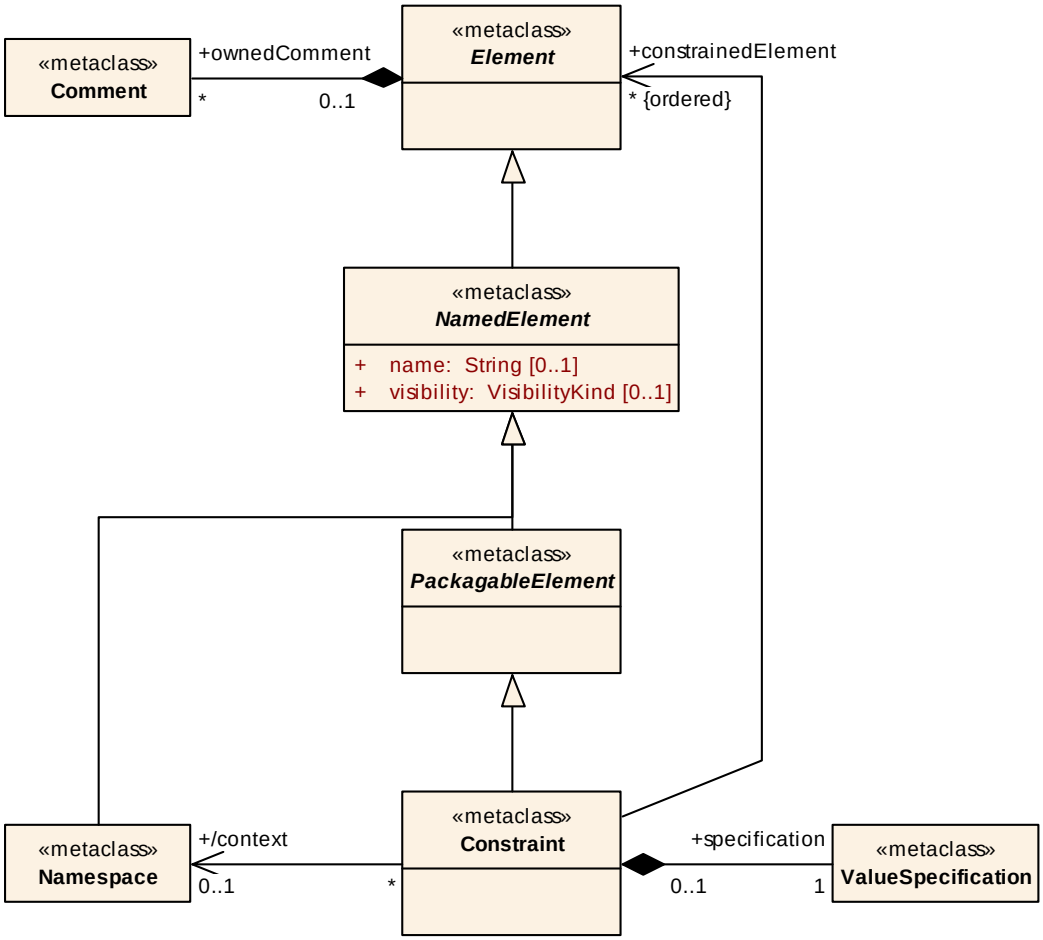

Figure 5-18: The *Constraint* metaclass as defined in UML 2.

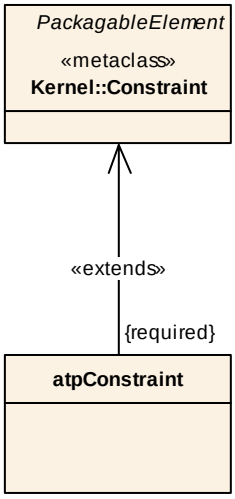

Figure 5-19: New stereotype <<*atpConstraint*>>.

Note that the stereotype *atpConstraint* MUST be used in AUTOSAR template models, which essentially means a supported language MUST be used, as described in the following table:

	Feature	Context	Constraint Description	Constraint in OCL (in Context)
1	having a name	`NamedElement`		
2	owning comments	`Element`		
3	definition with context	`Constraint`		
4	constraint expression	`Constraint`	Constraints must be expressed in OCL or another supported language.	`specification.oclIsKindOf(OpaqueExpression) and (specification.oclAsType(OpaqueExpression).language="OCL" or specification.oclAsType(OpaqueExpression).language="Java" or specification.oclAsType(OpaqueExpression).language="informal")`

Table 6: Constraints on stereotype *<<atpConstraint>>*.

For information about the supported languages and how to express constraints see section 6.2.7.

[ATPS-39]	Constraints MUST be expressed in OCL (preferred), Java or informal text.

5.2.10 Dependencies

Dependencies are extensively described in UML 2. AUTOSAR template models MAY use dependency stereotypes that have been defined within the context of UML already, namely:

<<trace>>

A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other [1].

<<use>>

A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation [1].

<<import>>

This dependency is allowed only for template subset models (see 8.3.1) and for reusing packages and elements from other standardization organizations such as ASAM MSR-SW.

Dependencies without stereotypes MAY be used, however, there is no formal semantics associated, i.e. those kind of dependencies are simply treated as model documentation.

Dependencies with stereotypes not listed above MUST NOT be used.

[ATPS-40]	Dependencies MAY be used. They MUST be used either without stereotype, or MUST be stereotyped <<*import*>> or <<*instanceRef*>>.

5.2.11 Packages

Packages are a grouping mechanism for model elements. AUTOSAR template models quickly grow large, so packages are required here as well. See Figure 5-20 for the original definition of packages in UML 2.

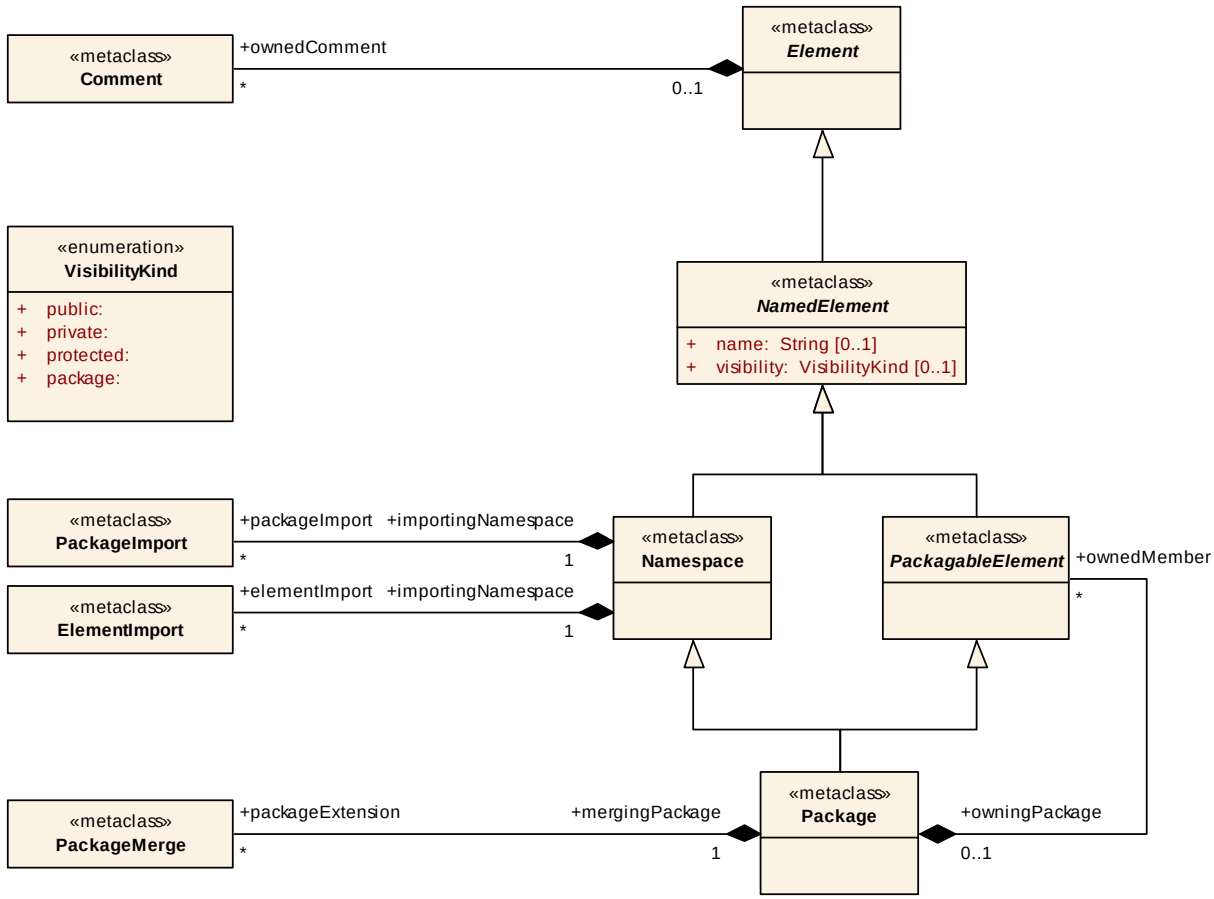

Figure 5-20: Original definition of *Package* in UML 2.

Not all of the features provided by UML 2 are desired for the simplified modeling of AUTOSAR templates. In particular, imports and merges are not required and will be excluded for the new stereotype *atpPackage*.

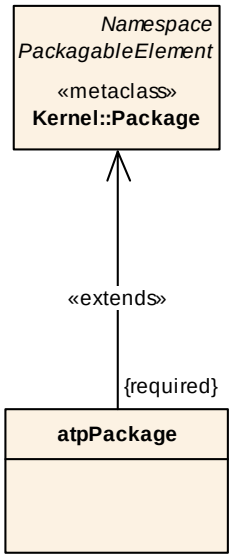

Figure 5-21: Definition of stereotype <<atpPackage>>.

In the following table the features and constraints for *atpPackage* are listed:

	Feature	Context	Constraint Description	Constraint in OCL (in Context)
1	having a name	`NamedElement`	Name MUST be a valid Identifier.	
2	owning comments	`Element`		
3	visibility	`NamedElement`	All elements in the metamodel MUST be public.	`visibility=public`
4	importing packages and elements	`Namespace`	Package imports MUST NOT be used, classes are always visible.	`elementImport->size()=0` `packageImport->size()=0`
5	merging with other packages	`Package`	Packages MUST NOT be merged.	`packageExtension->size()=0`

Table 7: Constraints on features of packages.

In addition to those constraints, *atpPackage* has semantics different from the UML 2 package:
Due to limitations/design guidelines of the model processing toolchains (DTD generators, …) namespaces are not controlling access, i.e. an M2 class *SWComponentType* is visible from all packages in the M2 model. This can formally be implemented by assuming a global import of all M2 packages from all other packages.

[ATPS-07]	All elements in model MUST be public.
[ATPS-09]	Packages imports MUST NOT be used to put model elements in scope.
[ATPS-10]	Package merges MUST NOT be used.

5.2.12 Primitive Types

UML 2 already defines the basic primitive types that may be used for attributes of classes. Those include:

Integer

An instance of *Integer* is an element in the (infinite) set of integers (…, -2 , -1 , 0, 1, 2, …). It is used for integer attributes and integer expressions in the metamodel [1]. An allowed alias for *Integer* is *Int*.

UnlimitedNatural

An instance of *UnlimitedNatural* is an element in the (infinite) set of naturals (0, 1, 2…). The value of infinity is shown using an asterisk (‘*’) [1].

Boolean

In the metamodel, Boolean defines an enumeration that denotes a logical condition. Its enumeration literals are:

- *true*: The Boolean condition is satisfied.
- *false*: The Boolean condition is not satisfied.

It is used for Boolean attribute and Boolean expressions in the metamodel [1].

String

An instance of String defines a piece of text. The semantics of the string itself depends on its purpose, it can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel [1].

For all primitive types corresponding literals are defined that allow specifying literal values for a primitive type. These are also used as defined in UML 2.

In addition to what is defined by UML, AUTOSAR requires a few more primitive types:

Float

It extends the semantics of *String* by requiring the string to be in a valid float number format.

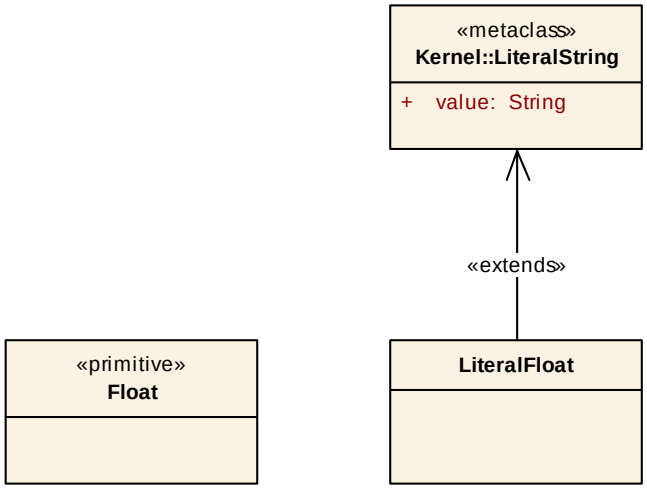

Figure 5-22: New primitive type *Float* and its value stereotype <<*LiteralFloat*>>.

The single constraint is listed in the following table.

Constraint Description	Constraint in OCL

The value string must be consistent with IEEE 754.	n/a.

Table 8: Constraints on a *Float* value.

Identifier

Similar to *Float* an *Identifier* puts additional constraints on the string format. *Identifiers* are supposed to be used whenever a string, typically some kind of name will later be processed by tools or in a programming language.

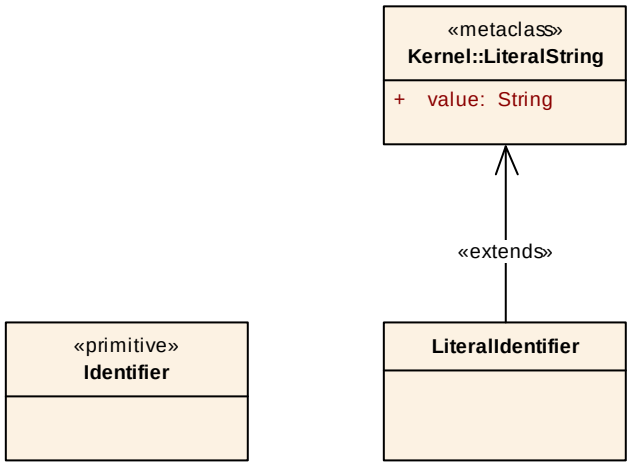

Figure 5-23: New primitive type *Identifier* and its value <<*LiteralIdentifier*>>.

The following formal constraint applies to the literal representation of an *Identifier*:

Constraint Description	Constraint in OCL

The value must satisfy the following syntax (in Extended Backus Naur Form [7]):
`identifier := letter (letter | digit | uscore)*`
 	
`letter := "a" | "A" | "b" | "B" | "c" | "C" | "d" |`
`"D" | "e" | "E" | "f" | "F" | "g" | "G" |`
`"h" | "H" | "i" | "I" | "j" | "J" | "k" |`
`"K" | "l" | "L" | "m" | "M" | "n" | "N" |`
`"o" | "O" | "p" | "P" | "q" | "Q" | "r" |`
`"R" | "s" | "S" | "t" | "T" | "u" | "U" |`
`"v" | "V" | "w" | "W" | "x" | "X" | "y" |`
`"Y" | "z" | "Z"`
 	
`digit := "0" | "1" | "2" | "3" | "4" | "5" | "6" |`
`"7" | "8" | "9"`
 	
`uscore := "_"`
	n/a.

An identifier has a maximum length of 32 characters. This is compliant with MISRA-C conventions [11].	`context LiteralIdentifier`
	`inv:`
	`value.size()<=32`

Table 9: Constraints on an *Identifier*.

[ATPS-20]	The primitive types from UML have been extended by two new types *Float* and *Identifier*, which MAY be used.
[ATPS-21]	An *Identifier* name MUST NOT exceed 32 characters.
[ATPS-22]	The characters used for an *Identifier* MUST be consistent with the rules given in Table 9: Constraints on an *Identifier*.

5.2.13 Enumerations

Enumerations are imported from UML 2 without refinement. This implies the notation within an M2 metamodel. Refer to the modeling convention given in 6.2.3 on how to express enumerations in an AUTOSAR template model.
Once an enumeration is defined the corresponding type can be used just like a primitive type.

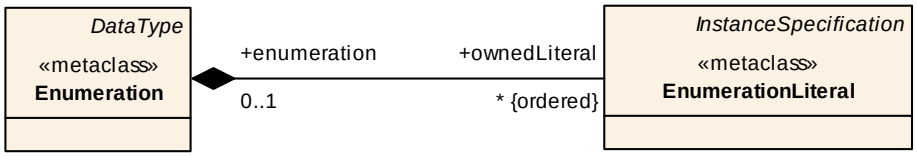

Figure 5-24: Definition of enumerations in UML 2.

5.2.14 Tagged Values

Tagged values are simply taken as they are defined by OMG as part of the *Extension* package defined in [3]. There, tagged values are a feature of the MOF metaclass *Object*, which is the common base class of all UML metaclasses.
Tagged values are typically used to annotate the model in order to control later code generation. E.g. a tagged value called *xml.name* could control the XML element name used for a class. The allowed sets of tagged values are defined in the persistence rules corresponding to a particular target platform. XML tag names are defined in [19].

[ATPS-47]	Platform specific (e.g. XML specific) model information MAY be added to the model in form of tagged values only.

5.2.15 Splitable attributes, associations and aggregations

By using the stereotype <<splitable>> the metamodel can explicitly define how instances of the metamodel may be distributed over several files. By default all data is stored in one single file. If the stereotype <<splitable>> is applied, then the associated or aggregated information may be stored in different files.

Example: Figure 5-25 shows how an AUTOSAR SenderReceiverInterface is integrated into the AUTOSAR infrastructure. It is not required to describe the elements *AUTOSAR*, *ARPackage* within one single file. From all these classes there exists a *splitable* containment path to the root element *AUTOSAR*. The

SenderReceiverInterface MUST be stored completely in one single file. There are no <<*splitable*>> aggregations, attributes or associations within the context of the *SenderReceiverInterface*.

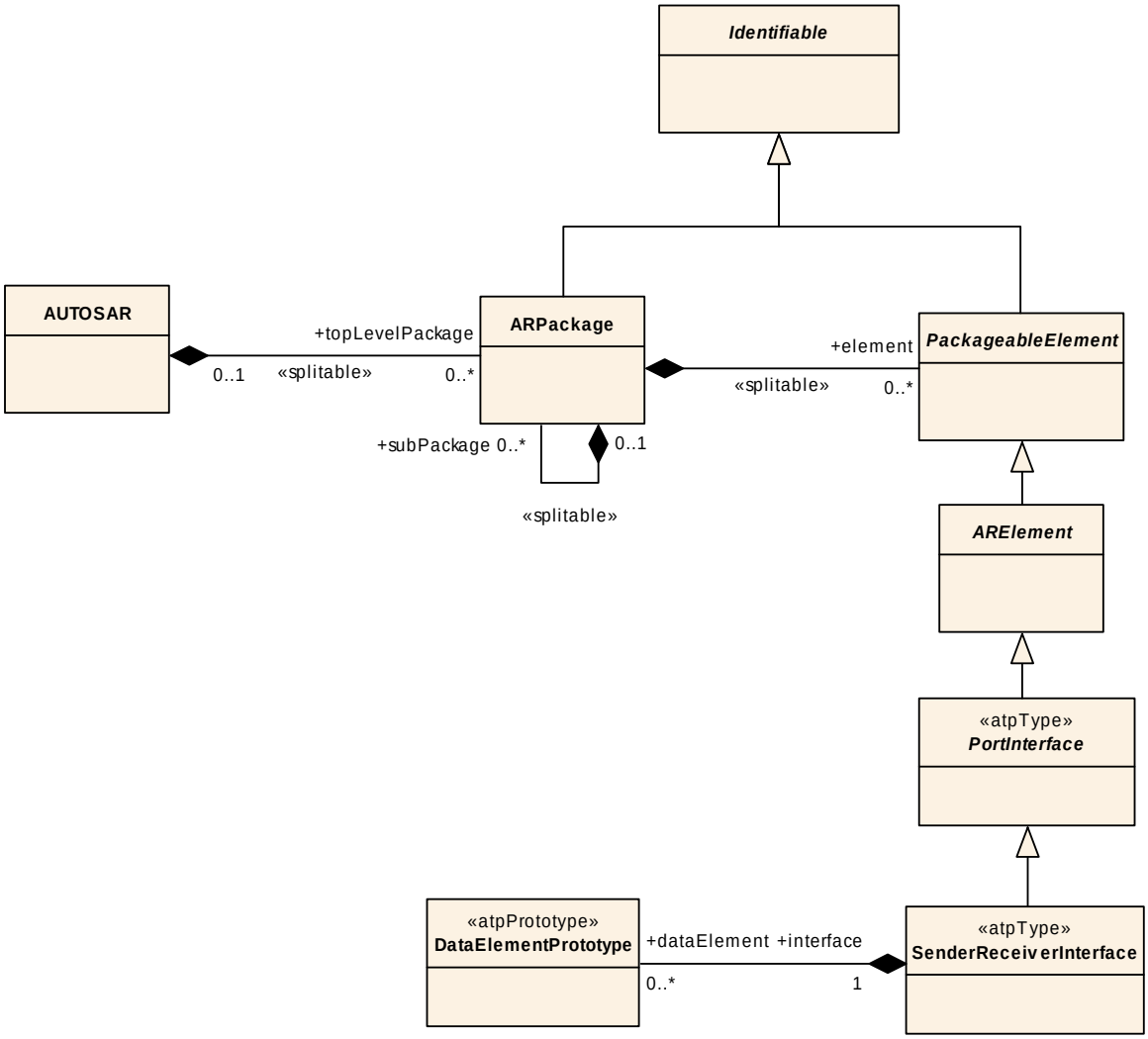

Figure 5-25: Example of splitable aggregations

[ATPS-53]	For all elements which are marked with the stereotype <<splitable>> there MUST exist a containment path to the root element where all navigated containments are stereotyped by <<splitable>>.

5.3 Summary

After having addressed all modeling features in detail this section will summarize the resulting UML Profile for AUTOSAR Template Models.

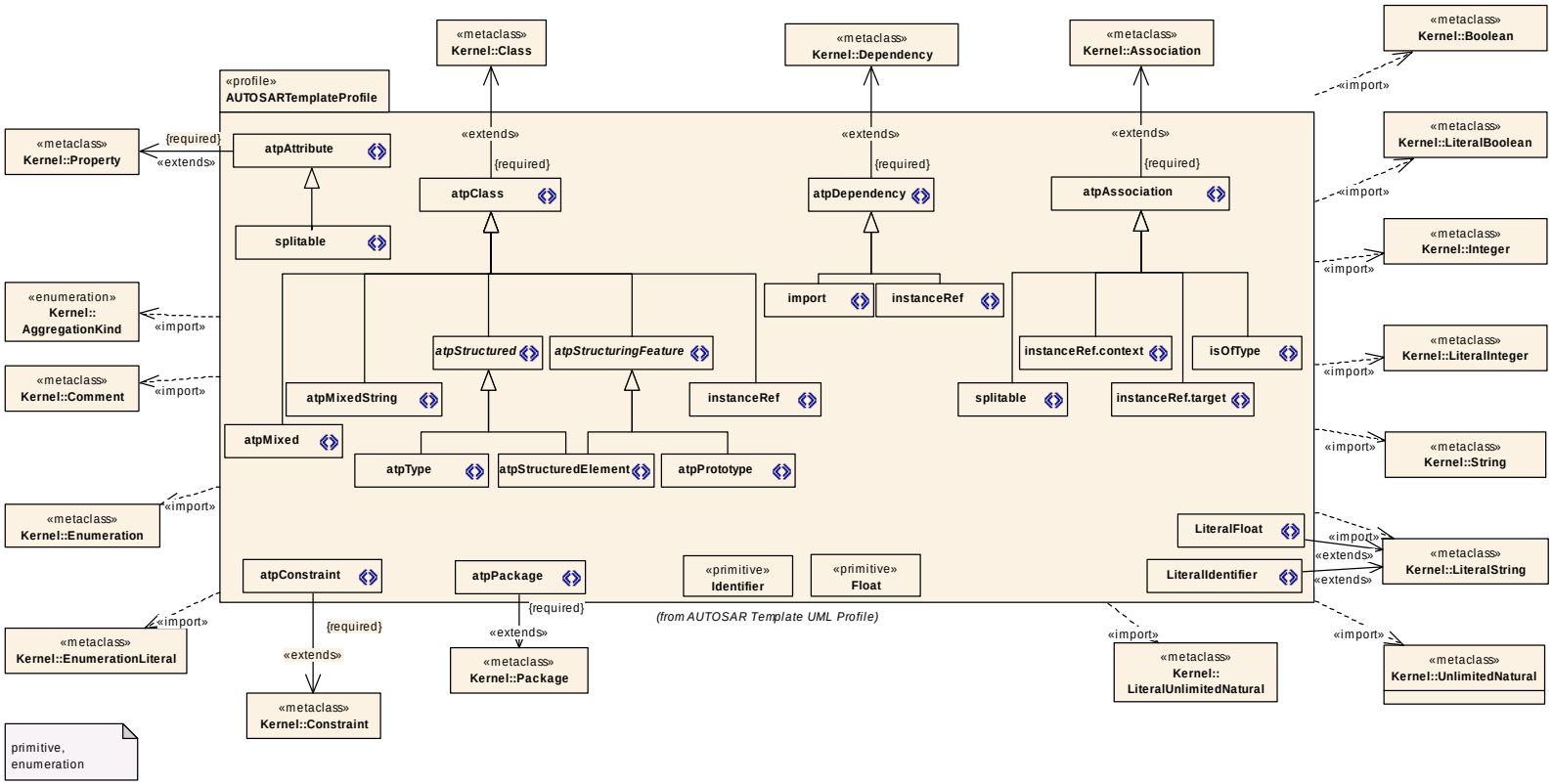

Figure 5-26: Overview of the UML Profile for AUTOSAR Templates.

Classes that are imported into the profile MAY be used as they are defined in UML 2. These are:

- *AggregationKind*
- *Boolean*
- *Integer*
- *String*
- *Enumeration*
- *LiteralBoolean*
- *LiteralInteger*
- *LiteralString*
- *LiteralUnlimitedNatural*
- *EnumerationLiteral*
- *Comment*

The remaining classes from UML 2 are extended through required stereotypes, i.e. the extended metaclasses MUST NOT be used in template models, instead the extending stereotype MUST be applied.
The new stereotypes defined in the profile are:

- *atpClass*: a class without operations
- *atpType*: a class representing a type declaration
- *atpPrototype*: a class representing the usage or an occurrence of *atpType*
- *atpMixed*: mixed content without inclassified strings
- *atpMixedString*: mixed content including unclassified strings
- *atpAssociation*: a concrete, binary association, with one navigable end
- *InstanceRef*: a particular kind of association referring to an instance of *atpPrototype*
- *IsOfType*: a particular kind of association referring from *atpPrototype* to its *atpType*
- *atpAttribute*: an attribute of a class, without redefinition, always public
- *atpConstraint*: a constraint that is always expressed in OCL
- *atpPackage*: packages of this kind imply importing each other (accessibility)
- *Float*: a float valued primitive datatype
- *LiteralFloat*: a literal value for *Float*
- *Identifier*: a string valued primitive datatype suitable for use in programming languages
- *LiteralIdentifier*: the string value of an *Identifier*

In addition the following primitive types have been added to the profile:

- *Float*: a rational number
- *Identifier*: a string that can be used in typical programming and tooling environments

6 Conventions

This chapter will list a number of modeling conventions that are not yet covered by the formal UML profile.

6.1 Naming Conventions

6.1.1 Language

The AUTOSAR standard language is US English. This means that all model elements, comments and all text in general MUST be written in English with US spelling.
For instance “behavior” MUST be used instead of “behaviour”, or “specialized” instead of “specialised".

[ATPS-03]	Model documentation and element names MUST be written in US English.

6.1.2 Model Element Names

Names of model elements MUST adhere to the following rules:

(a) Template model element names must satisfy the definition of an *Identifier*, defined in 5.2.12, except for the length limitation. Especially (but not exclusively), this means that none of the following character is allowed to be used: “,”, “.”, “:”, “-“, “(“, “)”, “{“, “}” and “ “.
(b) If a name consists of multiple terms, each of those terms starts with a capital letter, all other letter are not capitalized (e.g. *PortInterface*, or *DataElementPrototype*).
(c) Class names MUST begin with a capital letter.
(d) Attribute names MUST begin with a lower letter (e.g. *isService*).
(e) Role names MUST begin with a lower letter.
(f) Even if the multiplicity of an attribute or role is greater 1, the singular form of the respective name MUST be used (e.g. *port*, *connector*).

[ATPS-04]	Model elements MUST follow the specified naming conventions.
[ATPS-52]	Attribute and role names MUST be given in singular form, even if the corresponding cardinality is greater 1.

6.1.3 Class Names

In addition to the rules given above, class names must be unique in a template model.

[ATPS-06]	Class names MUST be unique within the combined set of all class names in the template model.

6.1.4 Diagram Names

Since the generation of various specification documents is partly automated, the names of all diagrams in the complete model – this includes M3 packages, scratch pads, … -- MUST be unique. Package hierarchy is not observed in our diagram extraction scripts.

[ATPS-05]	Diagram names MUST follow the specified naming conventions.

6.2 Modeling Conventions

6.2.1 Unambiguous Models

An AUTOSAR template model MUST be unambiguous. This means with a given template model that MUST be only a single way to represent specific semantic information through instantiation of the template.

[ATPS-02]	Template models MUST be unambiguous.

6.2.2 Abstract Classes

Metaclasses MUST be marked as *abstract* if it they are not intended to be instantiated directly in an M1 model. Abstract classes are used to capture attributes and semantics that are common to all derived subclasses.

[ATPS-12]	Classes MUST be marked abstract if they are not intended to be instantiated on M1.

6.2.3 Enumerations

If for instance for an enumeration called “Direction” the literals “forward” and “backward” need to be specified in UML, Figure 6-1 shows what this would look like: Essentially, enumerations are visualized as classes with a set of public attributes that make up their literals. The literals have no type. If the modeling tool supports it the “+” for public and the “: “ indicating the type MAY be omitted.

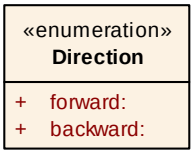

Figure 6-1: Example of an Enumeration on M2.

Enumerations are modeled as classes with a set of public attributes. As such the same rules apply for their names: they MUST be valid *Identifiers* and their names must be unique within the unified set of class and enumeration names.
The names of enumeration literals do not need to be unique within the full model, only within the namespace of the containing enumeration.

[ATPS-19]	Enumerations MUST be modeled as explicit classes with a public attribute for each literal.
[ATPS-14]	Regular class attributes MUST be typed. The only possible exception is when modeling enumerations.
[ATPS-51]	Enumeration names MUST be unique, also with respect to names of existing regular classes.

6.2.4 Class Attributes vs. Associations

In general it is pure modeling taste whether an aggregation is shown as class attribute or explicit association. For an AUTOSAR template model, the two options are used in the following situations:

(a) Primitive types (Integer, Float, Boolean, String) MUST be aggregated in form of direct class attributes.
(b) Metaclasses defined within the same M2 model MUST be aggregated in form of explicit associations and aggregations.
(c) Exception to (b): Enumerations, while defined as M2 classifier, MUST be aggregated as a direct class attribute.

Figure 6-2 shows examples of the cases distinguished above.

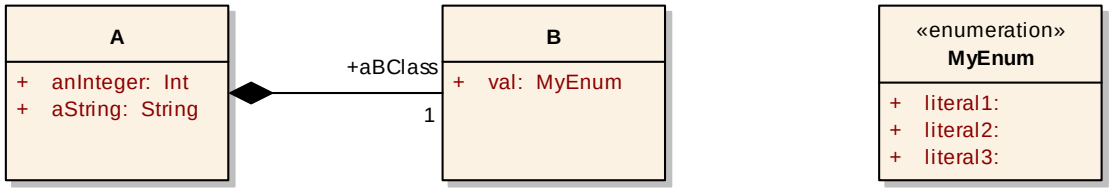

Figure 6-2: Examples of when to use which aggregation method.

[ATPS-23]	Primitive types and enumerations MUST be used by a class in form of direct class attributes.
[ATPS-24]	Classes MUST NOT be used in form of direct class attributes but are always referenced through an association.

6.2.5 Physical Quantities

Class attributes that represent physical quantities MUST use primitive type *Float* and be expressed in SI units [10].

[ATPS-49]	When representing continuous physical quantities in template models (in form of class attributes), SI units and datatytpe *Float* MUST be used.

6.2.6 Modeling Aggregations and Associations in Enterprise Architect

Sparx Systems Enterprise Architect (see [22]) is currently the standard UML tool to create AUTOSAR template models.
While UML defines which adornment of an association end means what (composition type, role name, cardinalities, constraints, …) the used modeling tool Enterprise Architect leaves some room how the visual representation is actually created. The following guidelines MUST be followed when adding associations and aggregations to a model:

- An association MUST have direction “source->destination”.
- Navigability along aggregations SHOULD be omitted[8]. It is assumed that logical navigability exists from container to part class.
- Each navigable association end MUST have a role name and cardinality.
- Each navigable association end MUST be documented.
- The source end of an aggregation (that is the *aggregated* end) must carry a role name and cardinality.
- The source end of an aggregation must be documented.

Note that documentation of the association itself is not required.

[ATPS-41]	Associations (with composite or no aggregation) MUST comply with the specified rules with respect to use of names, cardinality and documentation.

6.2.7 Constraints in Metamodel

6.2.7.1 Context Class

Due to OCL’s ability to completely navigate across model elements, it is possible to express constraints in the context of an arbitrary class. The following rules should guide the selection of the annotated metaclass.
In case rules contradict each other in a certain context, they MUST be applied in the prioritized order given here.

1. Constraints SHOULD be given in the context of the constraining class, not the constrained class.
2. The length of the OCL expression SHOULD be minimized through selection of the context metaclass.

[ATPS-42]	Constraints MUST be put into the model at the specified locations

6.2.7.2 Entering Information

Figure 6-3 shows the dialog in Enterprise Architect allowing to add a constraint to a model element. The following dialog fields are used to enter semantic information:

[8] This is an allowed representation option defined in [1].

- Field *Constraint*: the name of the constraint.
- Field *Type*: please select “OCL”.
- Field *Notes* (unnamed in figure): the actual constraint described in OCL plus additional data encoded in OCL comments. If it is not possible to describe the constraint using OCL any other preferably formal language may be used. This constraint needs to be encoded as OCL comments as well. Additionally the note shall state which language is used.

Key-value pairs follow the following syntax (in EBNF, see [7]):

```
kv-pair := “-- ” key “: ” value “\n”
key     := “id” | “severity”
value   := identifier
```

For the definition of `identifier` see section 5.2.12. The semantics of the defined keywords are:

- *id*: a unique (model-wide) id which will be displayed by tools in case the constraint is violated. The id should start with *ARSema (*Autosar Semantic Constraint).
- *severity*: The severity defines the behavior of a processing tool in case the constraint is violated. See [8] for exact definition of severities.

The last lines of the *Notes* field may be used for additional informal description of the constraint. The description starts with `“-- description: \n”` at the beginning of a new line.

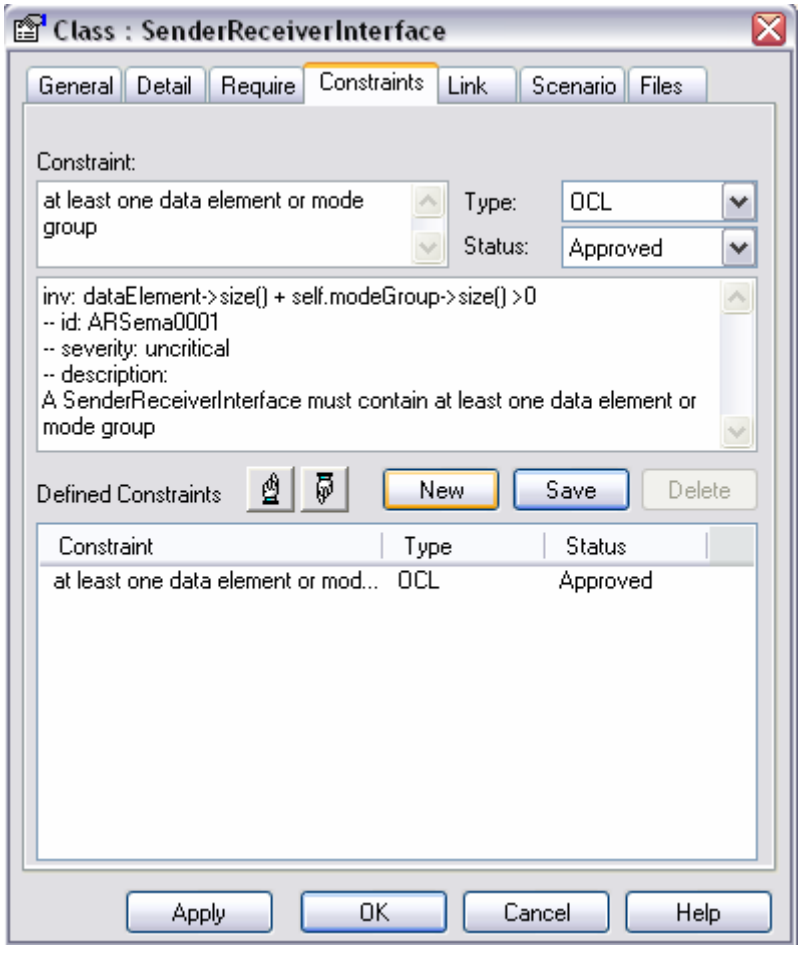

Figure 6-3: Constraint dialog in Enterprise Architect.

[ATPS-43]	Constraints MUST be formatted as specified.

6.2.8 Implicit & Explicit Stereotypes

For convenience reasons certain stereotypes MAY be omitted. They are assumed to be implicit in AUTOSAR template models. Implicitly stererotypes are:

- *atpClass*
- *atpAttribute*
- *atpAssociation*
- *atpPackage*
- *atpConstraint*

Stereotypes that MUST be explicitly part of template models are:

- *atpType*
- *atpMixed*
- *atpMixedString*
- *atpPrototype*
- *InstanceRef, instanceRef.target, instanceRef.context*
- *IsOfType*
- *import*
- *splitable*

It is ALLOWED to omit even enforced stereotypes, if one of the base classes has been appropriately stereotyped.

[ATPS-44]	The list of stereotypes given in MUST be observed with respect to which stereotypes are required to be given in model (explicit) and which are optional (implied).
[ATPS-45]	Additional stereotypes MUST NOT be defined.
[ATPS-46]	Stereotypes of base classes MAY be omitted for derived classes. This is true even for enforced stereotypes.

7 Template Model Infrastructure

7.1 Overview

The basic infrastructure of an AUTOSAR template is shown in Figure 7-1. The root container class for all templates is the class *AUTOSAR*, shown on the left end of the diagram.

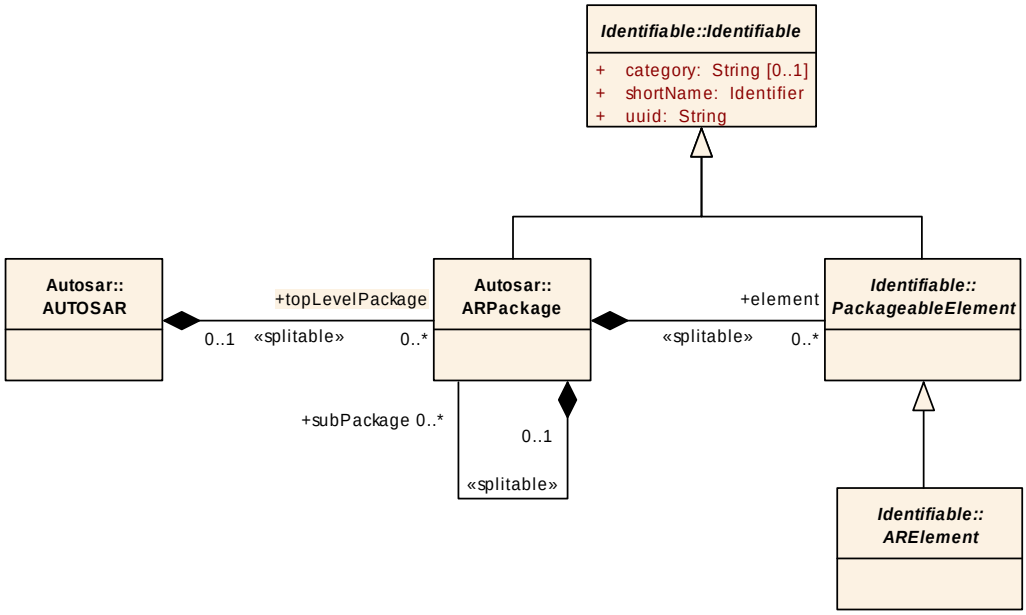

Figure 7-1: Infrastructure of an AUTOSAR template model.

7.2 Packaging

Class *AUTOSAR* is composed of a number of packages, represented by class *ARPackage*. This class is not to be confused with the M3 class *atpPackage* defined in 5.2.11.

While the latter refines the meaning and possibilities of using packages in a template UML model, *ARPackage* is actually part of such a model. It allows defining packages at AUTOSAR system modeling time, e.g. in form of an OEM or project specific package containing entities like a windshield wiper software component.

The self aggregation (role *subPackage*) shows that packages may in fact contain other subpackages. Besides those, a package may contain an arbitrary number of elements, represented by the abstract class *ARElement*.

Such an element is an entity for which it makes sense to be defined in its own semantic context (stand-alone). An example for such an *ARElement* is the definition of a reusable software component type, while on the other hand a parameter of an operation does not make sense to be defined stand-alone: its semantics is defined within and therefore highly dependent on the enclosing context – the operation.

7.3 Identifiable Classes

In addition to the structural classes, one more is shown in Figure 7-1: class *Identifiable*.
An *Identifiable* is an abstract base class that represents the ability to be the target of an association. *Identifiables* have a name and therefore can be referenced. This identifying name is represented in form of the attribute *shortName*. *Identifiables* have further attributes that are not explained here, since they have no structural or referential relevance. Instead refer to the AUTOSAR template patterns given in [24] for details.

[ATPS-48]	Referenced classes (at the target end of an association) MUST be derived from class *Identifiable*.

M1 Application Notes for Identifiable References
The *Identifiable*-based referencing mechanism is the only *official* mechanism in AUTOSAR templates. In an M1 model, references to an *Identifiable* therefore must be given in form of a shortname (or a sequence of shortnames).
Identifiable is also by definition a namespace for other contained identifiable classes. Only within such a namespace the name of an *Identifiable* must be unique.
When traversing a number of namespaces, the delimiter "/" is used to prepend the names of the namespace hierarchy containing the object.
Absolute references begin with "/", which essentially refers to the instance of class *AUTOSAR* in a description. Relative references, i.e. references *not* beginning with "/", are then resolved according to the following search order:

(a) all *Identifiables* declared in the namespace of the referring object
(b) all *Identifiables* declared in the namespace declaring the namespace that holds the referring object
(c) recursive application of (b).

Please note that the application notes given here form requirements in the M1 model only. Therefore, no further ATPS specification items for an M2 template model are created from them.

8 AUTOSAR Template Subsets

8.1 Goal

The complete AUTOSAR metamodel allows for the description of complex technical elements and their contexts across multiple abstraction levels with different levels of detail and so on. As a result the full model consists of a few hundred classes and even more attributes of those classes and is often overwhelming to new AUTOSAR users.
It therefore may be useful to hide all parts of the model that are not applicable to a certain problem or even implementation of AUTOSAR. This can be done through the definition of AUTOSAR subsets defined in the following sections.

8.2 Related concepts

MSR [20] implements this *model limiting* capability in form of its MSR Use Cases. Through the definition of additional constraints on the model only certain values, datatypes and model elements are allowed in a particular MSR Use Case.
UML allows for limitations on existing models (e.g. on UML itself) through *UML Profiles*. A profile explicitly imports models and model elements that can be used. Additional constraints can be added to those elements, e.g. in form of OCL statements.

8.3 Subset Notation

The notation defined here was initially compared to other approaches. See 10.1.3.1 for details why they were not chosen.

8.3.1 Importing Classes

In analogy to the definition of UML profiles, imports of metaclasses are used to define the required subsets. Those imports can be done in separate class diagrams showing <<*import*>> dependencies from the template subset to the enabled metaclasses and packages.
Figure 8-1 is an example of a subset of the existing AUTOSAR template allowing for logical definition of software components only, without the details of scheduling or implementation[9].
Once a set of classes is imported, the following model entities (classes and attributes) are part of the defined subset:

1. Directly imported classes are part of the subset.
2. Classes derived from directly imported classes are part of the subset.
3. All baseclasses of directly imported classes are part of the subset.

[9] These samples contain details of the AUTOSAR Software Component Template. It is not important that the user knows or understands those concepts. They are shown just for illustration of the subset concept. Also, the shown extract are not guaranteed to be valid.

4. All relations (associations and aggregations) between the classes in the subset are part of the subset.
5. All attributes (direct and inherited) of classes in the subset are part of the subset. If such an attribute requires another class (e.g. an enumeration), this class is part of the subset.

No other classes and attributes are part of the subset.

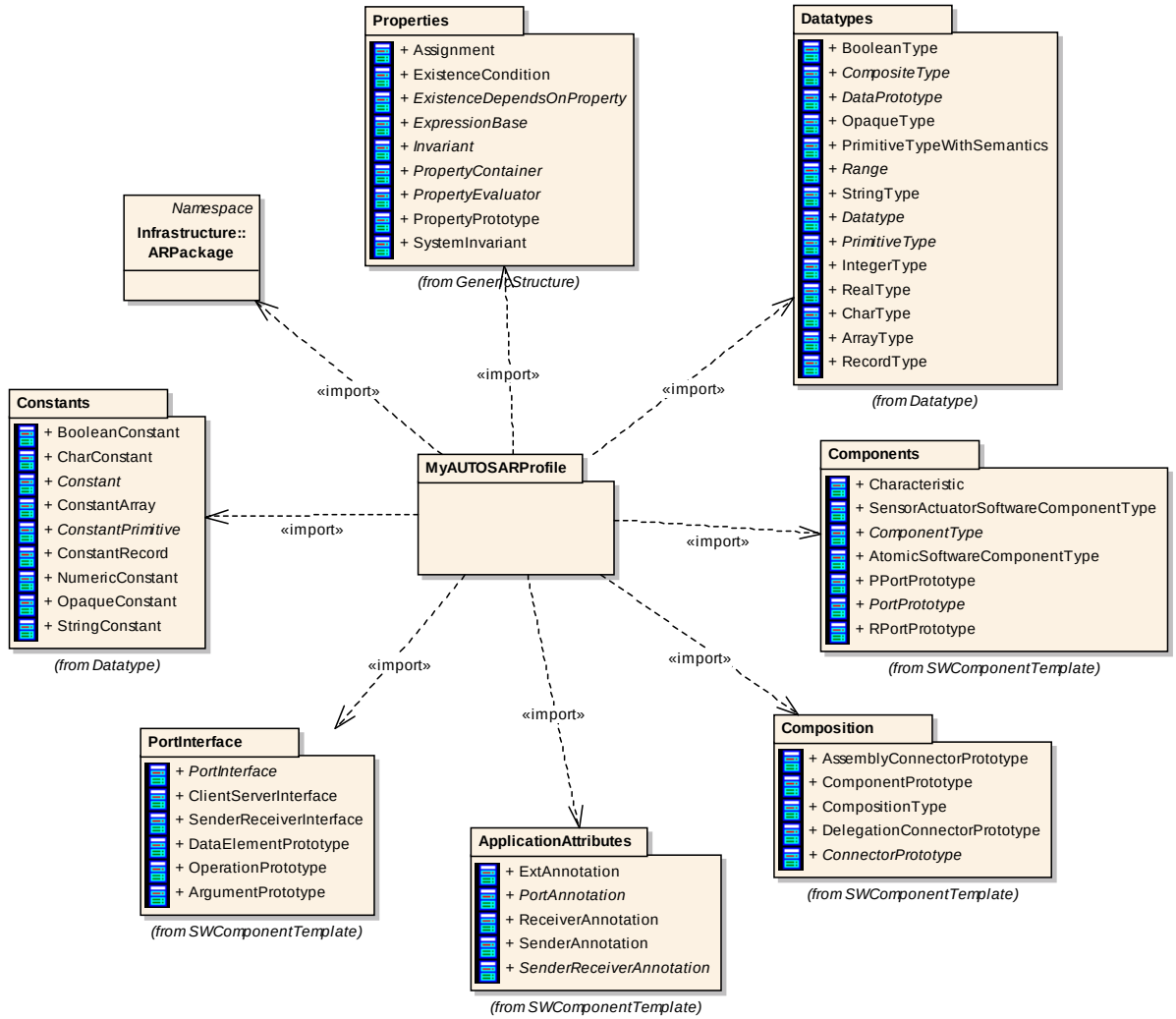

Figure 8-1: Sample AUTOSAR template subset for software components.

8.3.2 Attaching Constraints

Frequently there is need to attach additional constraints to the created subset, for instance to:

- restrict the allowed range of an attribute value, possibly to a single fixed value, or to
- constrain multiplicity.

Those constraints can then be attached to the subset package, as is shown below.

8.3.2.1 Attribute Value Constraints

To constrain the values of a certain attribute (which in this context refers to a direct attribute of a meta class), simple OCL expressions can be used [5]. For instance if a class *PortInterface* is part of the subset, its attributes *isService* needs to be removed, the following expression will do the job:

```
context PortInterface
     inv: isService = false
```

This defines an invariant in the context of metaclass *PortInterface*, demanding that for a valid model the *isService* flag must be false. This effectively removes the attribute from the remaining model in a defined way.
If instead of fixed attributes the range needs to be constrained, a similarly simple expression can be used. For example three kinds of directions are allowed for a class *ArgumentPrototype*, namely *in*, *out* and *inout*. If the reference like direction *inout* needs to be disallowed, the following expression can be used:

```
context ArgumentPrototype
     inv: (direction = in) or (direction = out)
```

Alternatively, the following statement can be used:

```
context ArgumentPrototype
     inv: direction <> inout
```

Obviously, the two alternatives are only equivalent if the original set of allowed values equals 3.

8.3.2.2 Multiplicity constraints

Very similar constraints on multiplicity of attributes and roles can be written. For instance if we have a class *ComponentType* which aggregates another class through the role name *ports* with original cardinality “*”, but the subset is supposed to restrict the allowed number of ports to less or equal 5 (for whatever reason), then this can be written like so:

```
context ComponentType
     inv: ports->size() <= 5
```

If an attribute or a role in the original metamodel has an upper multiplicity greater 1, the corresponding OCL type is called a collection. OCL defines a set of predefined operations that can be applied to collections and *size()* is one of them, returning the number of elements in the collection.

8.3.2.3 Location of constraints

Since the original metamodel is not supposed to be changed by the definition of a new element subset and the constraints defined for it, those constraints must not be attached to the actually constrained model elements.
With the *context* keyword of OCL it is possible to write constraints almost anywhere in and even outside of the model, e.g. in a separate document. A logical place for the constraints of a newly defined subset is the subset package itself. This way the constraints can still be kept in the model.

The following example shows how this could look for the subset and the three constraints defined above.

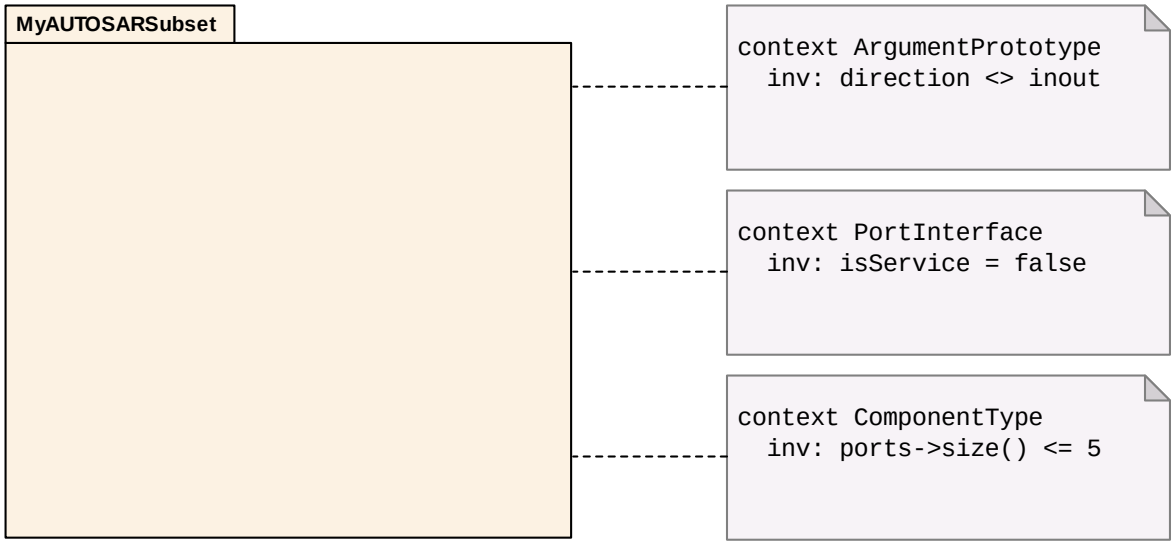

Figure 8-2: Constraints can be attached to the defined subset.

Note that Enterprise Architect provides a feature that allows linking the text in a UML note to a feature of the annotated element, which can be used to implement single source constraints[10], since it is updated as soon as the element feature is changed.

[ATPS-50]	Subsets of AUTOSAR template models MUST be defined using the specified notation.

[10] Right click on the note link between the note and the annotated element and select “Link this note to en element feature”. There, select the feature type “Constraint” and lastly the constraint to show.

9 Specification Items

All specification items given in this document are briefly listed in this section to allow referencing them from other AUTOSAR documents as well as systematic checking for model compliance.

[ATPS-01] Template models MUST be expressed in form of UML class diagrams.
[ATPS-02] Template models MUST be unambiguous.
[ATPS-03] Model documentation and element names MUST be written in US English.
[ATPS-04] Model elements MUST follow the specified naming conventions.
[ATPS-05] Diagram names MUST follow the specified naming conventions.
[ATPS-06] Class names MUST be unique within the combined set of all class names in the template model.
[ATPS-07] All elements in model MUST be public.
[ATPS-08] Redefinition of model elements MUST NOT be used.
[ATPS-09] Packages imports MUST NOT be used to put model elements in scope.
[ATPS-10] Package merges MUST NOT be used.
[ATPS-11] Classes MUST NOT have operations.
[ATPS-12] Classes MUST be marked abstract if they are not intended to be instantiated on M1.
[ATPS-13] Template class attributes and association roles MUST NOT be defined *static* (UML default).
[ATPS-14] Regular class attributes MUST be typed. The only possible exception is when modeling enumerations.
[ATPS-15] Attributes MUST NOT be defined *readonly* (UML default), i.e. no fixed attributes are allowed.
[ATPS-16] Attributes MUST be defined *unique* (UML default.
[ATPS-17] Attributes MUST not be derived from other attributes.
[ATPS-18] Default values MUST NOT be defined for attributes.
[ATPS-19] Enumerations MUST be modeled as explicit classes with a public attribute for each literal.
[ATPS-20] The primitive types from UML have been extended by two new types *Float* and *Identifier*, which MAY be used.
[ATPS-21] An *Identifier* name MUST NOT exceed 32 characters.
[ATPS-22] The characters used for an *Identifier* MUST be consistent with the rules given in Table 9: Constraints on an *Identifier*.
[ATPS-23] Primitive types and enumerations MUST be used by a class in form of direct class attributes.
[ATPS-24] Classes MUST NOT be used in form of direct class attributes but are always referenced through an association.
[ATPS-25] Aggregations MUST either be of type *none* or *composite*. Aggregation type *shared* MUST NOT be used.
[ATPS-26] Attribute subsets MUST NOT be used.
[ATPS-27] Opposite attributes MUST NOT be defined.
[ATPS-28] For template classes representing a type, stereotype <<*atpType*>> MUST be used.
[ATPS-29] For template classes representing a prototype, stereotype <<*atpPrototype*>> MUST be used.
[ATPS-30] Associations MUST not be named.
[ATPS-31] Associations MUST be binary.

[ATPS-32] Associations MUST have exactly one non-navigable end.

[ATPS-33] Every prototype MUST refer to its defining type via an association typed *<<isOfType>>*.

[ATPS-34] The *<<isOfType>>* association MUST be used only to refer from an *atpPrototype* to an *atpType*.

[ATPS-35] The target cardinality of an *<<isOfType>>* association MUST be one.

[ATPS-36] When referring to an instance of a prototype – and not to its definition in another enclosing type – an association stereotyped *<<instanceRef>>* MUST be used.

[ATPS-37] The source end of an *<<instanceRef.context>> or <<instanceRef.target>>* association MUST be a class with the stereotype *<<instanceRef>>*.

[ATPS-38] The target end of an *<<instanceRef.context>>* association MUST be an *atpPrototype*.

[ATPS-39] Constraints MUST be expressed in OCL (preferred), Java or informal text.

[ATPS-40] Dependencies MAY be used. They MUST be used either without stereotype, or MUST be stereotyped *<<import>>* or *<<instanceRef>>*.

[ATPS-41] Associations (with composite or no aggregation) MUST comply with the specified rules with respect to use of names, cardinality and documentation.

[ATPS-42] Constraints MUST be put into the model at the specified locations.

[ATPS-43] Constraints MUST be formatted as specified.

[ATPS-44] The list of stereotypes given in MUST be observed with respect to which stereotypes are required to be given in model (explicit) and which are optional (implied).

[ATPS-45] Additional stereotypes MUST NOT be defined.

[ATPS-46] Stereotypes of base classes MAY be omitted for derived classes. This is true even for enforced stereotypes.

[ATPS-47] Platform specific (e.g. XML specific) model information MAY be added to the model in form of tagged values only.

[ATPS-48] Referenced classes (at the target end of an association) MUST be derived from class *Identifiable*.

[ATPS-49] When representing continuous physical quantities in template models (in form of class attributes), SI units and datatytpe *Float* MUST be used.

[ATPS-50] Subsets of AUTOSAR template models MUST be defined using the specified notation.

[ATPS-51] Enumeration names MUST be unique, also with respect to names of existing regular classes.

[ATPS-52] Attribute and role names MUST be given in singular form, even if the corresponding cardinality is greater 1.

[ATPS-53] For all elements which are marked with the stereotype <<splitable>> there MUST exist a containment path to the root element where all navigated containments are stereotyped by <<splitable>>.

10 Appendix

10.1 Design Rationales (informative)

This section will list the different design decisions applied throughout this specification.

10.1.1 Choice of Metamodeling Mechanism

One of the design goals of this specification is that existing and widely accepted modeling standards MUST be incorporated, namely the various mechanisms of UML. Within this restriction the following approaches for the definition of a specialized metamodel for AUTOSAR templates have been analyzed.

10.1.1.1 Instantiation of MOF Entities

In this approach the new metamodel is an instance of MOF, therefore lives in parallel and in particular independently of UML, which is also an instance of MOF, as shown in Figure 10-1. While almost any kind of metamodel can be built with MOF the independency of UML is problematic in terms of tool support.
The number of UML modeling tools is constantly growing and professional tools exist today. If however peculiar meta constructs are modeled, a standard UML tool may very well not be able to express models built on those constructs [13].
There are tools that span two metalevels (e.g. GME [14], which was used in the ITEA EAST-EAA project [15]), therefore allowing to build metamodel and subsequent user models, but today the number, quality and commercial applicability of those tools is questionable.

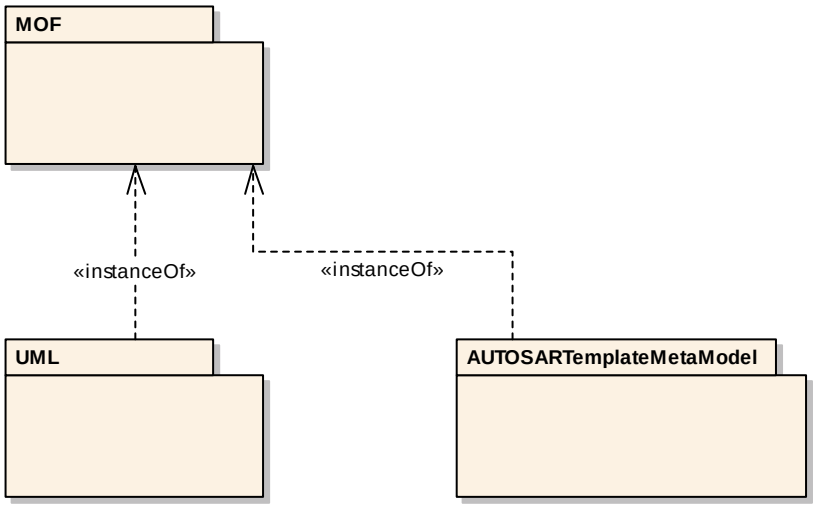

Figure 10-1: AUTOSAR template model as MOF instance.

10.1.1.2 Instantiation of UML Entities

This is a very similar approach to what has been described above, just that UML constructs are used to build the metamodel. While on a different metalevel (see Figure 10-2), the problems are the same, namely missing tool support.

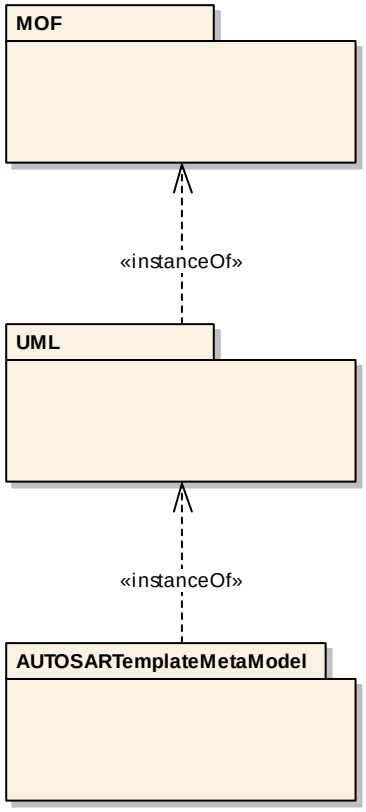

Figure 10-2: AUTOSAR template model as UML instance.

10.1.1.3 Extending UML through UML Profiles

Unlike the approach in 10.1.1.2, where UML is instantiated to create a new metamodel on a *lower* metalevel, a *UML profile* extends UML *on the same* metalevel by basically adding new metaclasses (see Figure 10-3).
The profile mechanism is designed in such a way that a model built with it is still a valid UML instance. This requirement limits the additions a profile can bring to an existing metamodel:

- New metaclasses that extend existing metaclasses. Those new metaclasses are called *stereotypes*.
- Stereotypes must not define additional attributes, but only own the attributes that the extended metaclass already has[11].
- Additional constraints on the model, e.g. fixed values for certain class attributes or restricted ranges for values and cardinalities.

For a detailed introduction into UML profiles see [13] as well as the formal specification by OMG [2].

[11] A common workaround for this limitation is the use of tagged values, not explicitly explained here.

While the modification options of a profile are limited by the aforementioned requirement, this at the same time has the extreme advantage that profiled models can be built, exchanged and visualized by standard UML tools and even further processed by any UML (or XMI [4] for that matter) toolchain.
Besides the technical advantage of finding ready-to-use toolchains when applying the UML profile mechanism, there is also a political advantage: it will be much easier to defend a metamodel that is built on standard UML with has been extended through standard mechanisms instead of a metamodel that is built from scratch.

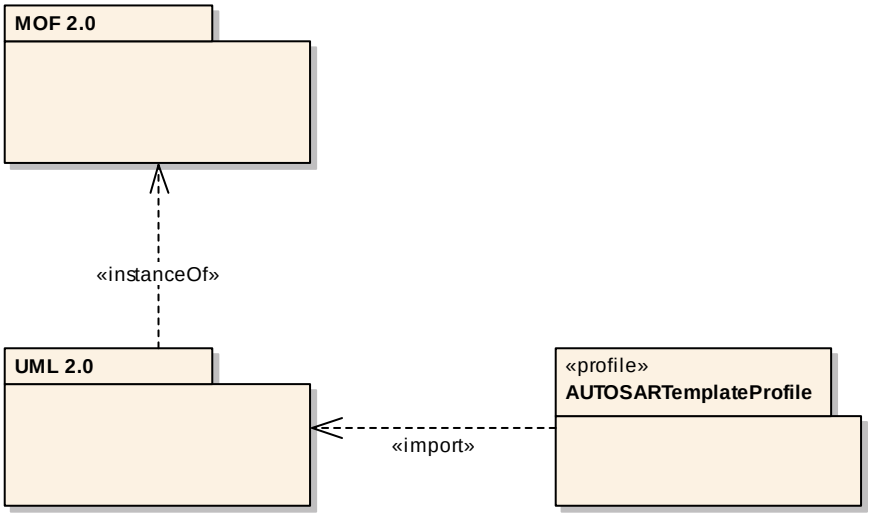

Figure 10-3: AUTOSAR template model as UML profile.

10.1.1.4 Templates as Direct UML Profiles

Just for completeness this fourth approach is also mentioned here: it is theoretically also possible to abandon the explicit templating metamodel and create the actual AUTOSAR template directly as UML profiles. This would e.g. lead to stereotypes called “software component”, “port” and “interface” in case of the software component template.
The major disadvantage – besides asking template modelers to become proficient in handling UML profiles – is that for instance all associations between the stereotypes need to be derived from existing associations. Since those existing associations are very general (associating any number of classes), the particular attributes of relations, namely cardinality, which classes can be connected and aggregation type would need to be captured in form of constraints on the general association.
This makes this approach basically unmanageable.

10.1.1.5 Conclusion

For the reasons given above, alternative *10.1.1.3 Extending UML* has been chosen.

10.1.2 Modeling Prototype Attributes

The semantics of the *IsOfType* association from *atpPrototype* to *atpType* implies that if an instance of *atpType* has a certain attribute (either in form of an *atpAttribute*, or through a *atpAssociation*), a corresponding attribute is available on the respective *atpPrototype*.
E.g., taken from the metamodel described in [18], if a software component *type* has ports, then obviously one expects a corresponding software component *prototype* to also expose ports.
Frequently, those attributes need to be referred to in the model, e.g. a connector will need to connect the ports of component prototypes instead of connecting the ports of reusable, unconnected component types!
Two options have been analyzed to allow this, which are discussed in the following sections.

10.1.2.1 Explicit Prototype Attributes

This is the most straight-forward approach. To stick with the above example: if component types have ports, then also component prototypes need to have ports modeled. A semantic constraint needs to then make sure that the two port sets are consistent, as shown in Figure 10-4.

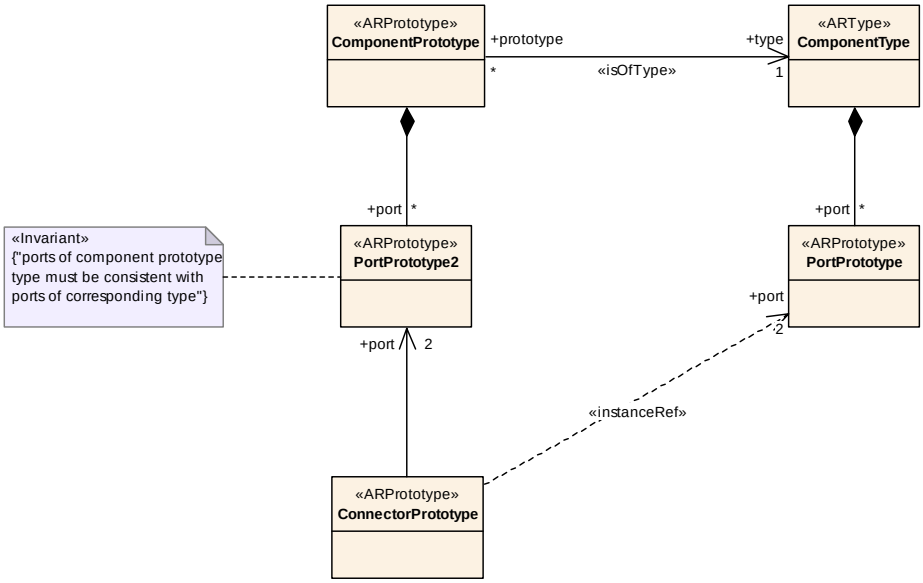

Figure 10-4: Alternative of prototypes having explicitly modeled attributes.

The definitive advantage of such an approach is how easy it is to understand. The major disadvantage is that if a metamodel (like the AUTOSAR templates) contains many kinds of types and prototypes, then large portions of the model will be duplicated, at least in terms of attributes and associations (similar as to how in the above example *ComponentPrototype* now also has an composite aggregation of ports, which is not in the original model).

10.1.2.2 InstanceRef

Another approach is to specialize associations to instances through a new stereotype <<*instanceRef*>>, which simply implies the semantics of what is explicitly modeled and constrained in the previous section.
This means that no explicit prototype attributes are modeled unless they are really attributes of the prototype only, not also of the referenced type. The example would then look like .

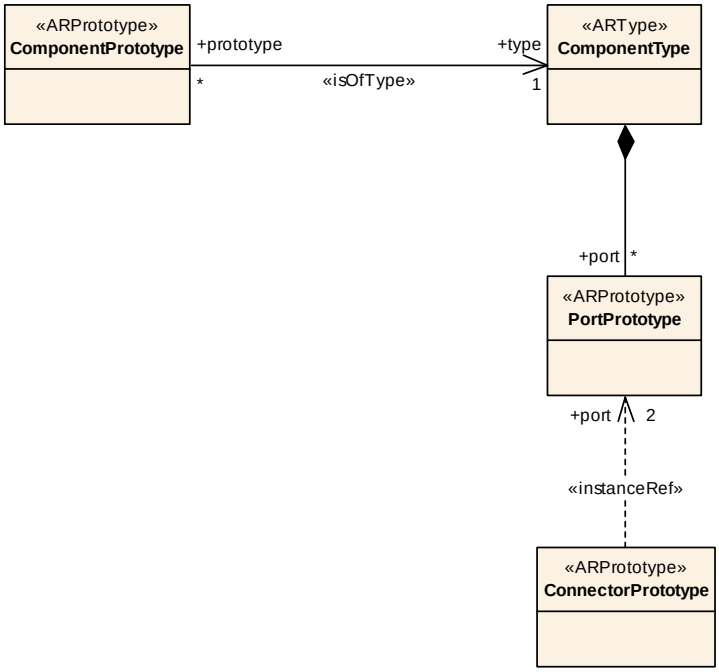

Figure 10-5: Reference to prototype instances through a instanceRef association.

10.1.2.3 Conclusion

It was decided to go with the latter alternative to keep template models as simple as possible. This knowingly shifts the difficulty to the evaluating tool chain which needs to implement the described *InstanceRef* semantics.

10.1.3 AUTOSAR Subset Modeling

In this section the disadvantages of other approaches leading to template subsets (see 8.3) are listed.

10.1.3.1 Tagged values

An option that was originally discussed is the application of tagged values. Every UML model element, including the AUTOSAR metaclasses, allows for adding so-called *tagged values*, consisting of simple key-value pairs.

While less trivial keys are certainly possible, a simple approach would be the addition of a tagged value like the following to indicate a model element is in fact part of a certain profile:

```
PART-OF-PROFILE-XY = TRUE
```

Tools can then check the model for the existence of the corresponding tag and deal with it appropriately.
There are a number of disadvantages of this approach:

- By adding tagged elements to a meta class, the meta class is in fact modified. This is problematic. In fact, in a version-controlled environment a new revision of the meta class would need to be committed just because a new profile was defined.
- Tagged values are inherently text-based. This means that tools may be able to display them in UML diagrams, but whether a tool is able to filter by tagged values is questionable (EA is not). This means that in case tagged values are shown, the tagged values for all profiles and all other tagged values are shown for all classes in the diagram. This gets messy very fast.

10.1.4 Stereotypes vs. Tagged Values

Frequently, when mapping the metamodel to a specific platform like XML, modelers are quick to invent new stereotypes expressing platform specifics, e.g. that a container class in the model should result in an XML *or-group*.
As there is some ambiguity to adding a stereotype to a class or alternatively adding a (Boolean) tagged value to express membership, we have the following findings:

- A stereotyped class is essentially an instance of a metaclass in the underlying metamodel.
- As this profile is intended to be target platform independent (comparable to a PIM in MDA [23]), the instantiated metamodel and therefore the defined stereotypes are also always platform independent.
- Platform dependent information is expressed in form of tagged values.
- The tags available are defined in the persistence rule document for a specific target platform.

10.2 References

In this section the references used in this specification are listed. They are separated into normative and non-normative (informative) references.

10.2.1 Normative References

[1] Unified Modeling Language: Superstructure, Version 2.0, OMG Adopted Specification, ptc/03-08-02.
[2] UML 2.0 Infrastructure Specification, OMG Adopted Specification, ptc/03-09-15.
[3] Meta Object Facility (MOF) 2.0 Core Specification, OMG Adopted Specification, ptc/03-10-04.

[4] UML 2.0 Diagram Interchange Specification, OMG Adopted Specification, ptc/03-09-01.
[5] UML 2.0 OCL Specification, OMG Adopted Specification, ptc/03-10-14.
[6] Key words for use in RFCs to Indicate Requirement Levels. Network Working Group, S. Brandner, Harvard University, 1997. (http://www.ietf.org/rfc/rfc2119.txt).
[7] ISO/IEC 14977:1996: Extended BNF, freely available at http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153
[8] Requirements on Interoperability of Authoring Tools, https://svn2.autosar.org/repos2/22_Releases/, AUTOSAR_RS_InteroperabilityAuthoringTools.pdf
[9] Specification of Interoperability of Authoring Tools, https://svn2.autosar.org/repos2/22_Releases/, AUTOSAR_InteroperabilityAuthoringTools.pdf
[10] Guide to the SI, with a focus on usage and unit conversions: NIST Special Publication 811, 1995 Edition, by Barry N. Taylor. Guide for the Use of the International System of Units (SI).
[11] Guidelines for the Use of the C Language in Critical Systems (“MISRA C Guidelines”), ISBN 0-9524156-2-3, October 2004.

10.2.2 Informative References

[12] UML 2 Toolkit. H.-E. Eriksson, M. Penker, B. Lyons, D. Fado, OMG Press, Wiley Publishings, Indianapolis, 2004.
[13] An Introduction to UML Profiles. L. Fuentes, A. Vallecillo, Universidad de Málaga, Spain, 2003. (http://se2c.uni.lu/tiki/se2c-bib_download.php?id=1421).
[14] The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/
[15] ITEA EAST-EAA, Embedded Electronic Architecture, http://www.east-eea.net/start.asp
[16] UML 2 glasklar. M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins, Carl Hanser Verlag, München, 2004.
[17] What is metamodeling and what is it good for? metamodel.com, 2002. (http://www.metamodel.com/staticpages/index.php?page=20021010231056977)
[18] Software Component Template, https://svn2.autosar.org/repos2/22_Releases/, AUTOSAR_SoftwareComponentTemplate.pdf
[19] Model Persistence Rules for XML, https://svn2.autosar.org/repos2/22_Releases/, AUTOSAR_ModelPersistenceRulesXML.pdf.
[20] MSR-SW Element Attribute Documentation V2.2.2. (http://www.msr-wg.de/medoc/download/)
[21] Definition of term “First-class object”, Wikipedia. (http://en.wikipedia.org/wiki/First-class_object)
[22] Sparx Systems homepage: http://www.sparxsystems.com.au
[23] MDA Guide Version 1.0.1, OMG Document, omg/03-06-01

[24] Template Modeling Patterns,
https://svn2.autosar.org/repos2/22_Releases/,
AUTOSAR_TemplateModelingPatterns.pdf.