Specification of an Architecture Meta-Model*

Raphael Weber, Eike Thaden,
Philipp Reinkemeier, Andreas Baumgart

Tuesday 31st January, 2012

*The research reported in this document has mostly been performed in the project SPES2020 (www.spes2020.de) which is funded by the Ministry of Science and Culture. Also, the projects CESAR (www.cesarproject.eu) and SPEEDS (www.speeds.eu.com) had major influences on the results of this work.

Specification of an Architecture Meta-Model

Contents

1 Introduction

This document specifies the SPES Meta-Model (SPESMM) including the Heterogeneous Rich Component (HRC) basis and its extensions.

1.1 Overview

The document starts with this introduction in which the purpose and the scope will be presented. The scope motivates the need for various modeling concepts based on requirements on the meta-model. It shows how existing assets meet these requirements and how modeling concepts and different meta-models can be integrated. The second chapter provides a specification of the SPES Meta-Model.

1.2 Purpose

The purpose of this document is to motivate the need for a new meta-model based on HRC from the SPEEDS project [Pro07] and to specify the SPES Meta-Model.

1.3 Scope

In the context of the SPES2020 project a so called Reference Architecture (for better understandability and correctness we call it a meta-model) is to be specified. Tools and services being connected to this meta-model are able to interoperate with each other. This requires means for data exchange and common interfaces satisfied by the meta-model.

The SPES Meta-Model which is specified in this document covers the need for a System Meta-Model in SPES2020. In the following we show the coverage of the SPES Meta-Model and provide integration concepts. The formal description of a modeling theory (ZP-AP1), the description of a model-based requirements engineering approach (ZP-AP2), the specification of efficient safety analysis methods (ZP-AP4), or the description of the realization of multi-core realtime systems including methods to analyse them (ZP-AP5) are not scope of this document.

1.3.1 Motivation

The SPESMM provides interoperability between tools and services. Many tools have internal meta-models or provide connectivity to open meta-models such as MARTE [Obj08b] or SysML [Obj08a]. Other tools or services might provide support for such meta-models but this is not the general case. When connecting tools and services with a heterogeneous set of meta-models, there is a strong need for data exchange. Point-to-point connections between specific tools are always possible and have to be regarded as well. But the support of a new tool with a foreign meta-model requires providing data exchange solutions to all tools which shall communicate with the tool. In cases where the full semantics of the model have to be supported and specific artifacts have to be accessed this is useful.

Generally, tools do not know all foreign meta-model concepts. So an interface to all kinds of models would either be only structural without semantics or has to provide all possible semantics which is difficult to capture and such a "world interface" would not be useful. In many cases it is only relevant to identify more abstract concepts and artifacts with generic semantics i. e. when browsing models or for common analysis and simulation techniques. For this purpose the SPES Meta-Model will provide a technique to access common concepts, artifacts and relationships in models of meta-models. Such meta-models are integrated by performing a tailoring process in which meta-model artifacts and relationships are mapped to the concepts of the SPES Meta-Model. The SPES Meta-Model provides core structure and semantics. Models can therefore be completely stored in the SPESMM for full model exchange and have furthermore a generic interface as defined by the SPES Meta-Model.

In order to generally support a large number of modeling tools the SPES meta-model is also presented as a UML Profile, relying on UML models extended through stereotypes. Consequently, the profile approach is not as flexible as the meta-model, but we were able to identify missing UML concepts present in the SPESMM and to integrate them into the profile. Thus, in some SPESMM descriptions there are hints as to how the specific meta-model artifact is represented in the profile.

1.3.2 Meta-Model Requirements

Recent meta-model approaches like EAST-ADL2 [ATE08], AADL [FGH06], SysML [Obj08a] or HRC [JMM08] contain structural features such as components, ports, and connectors. In conjunction with the requirements from Deliverable D3.1.A [WTR09] we deduced general meta-model requirements for the SPESMM. In the following we provide a concept of common structural artifact kinds and a description of the composition concepts.

The SPESMM will enable heterogeneous component design. Each of the mentioned meta-models contains artifacts which resemble components. The kind of a component thereby depends on the abstraction level and on the subject architecture in which the component shall be placed. For instance, on the one hand a functional draft can con-

sist of a set of abstract black boxes without internal behavior, on the other hand a concrete design can contain software functions or hardware modules with a behavioral description.

The SPESMM, when classified into the schemes of different compositions (see [Ode98]) resembles the component-integral object and the member-bunch composition. The latter allows differing configurations — this is true for higher abstraction levels of the SPESMM, since there is no information about the specific place of a component. At the lower abstraction levels this information is added and the composition concept reflects a component-integral object composition, containing more details on the configuration of components.

As depicted in Figure 1.1 components can have ports as a communication means which contain data or which provide respectively invoked services. A port must have a data type. The applicable data type depends on the component which the port belongs to. In general, functional components exchange data using higher data types like numbers, strings or complex compound objects. Furthermore, they can communicate using services for function calls. Lower level components might talk about bit arrays and electrical components expect certain voltages on the lines of their interfaces. In order to enable communication between two components there has to be a connection between the ports of a component. Components may contain attributes as a port for internal use. All ports can be typed. Furthermore, a port has a name and can be referenced. The internal behavior of a component can be described by a behavioral description from which the ports of the respective component can be referenced.

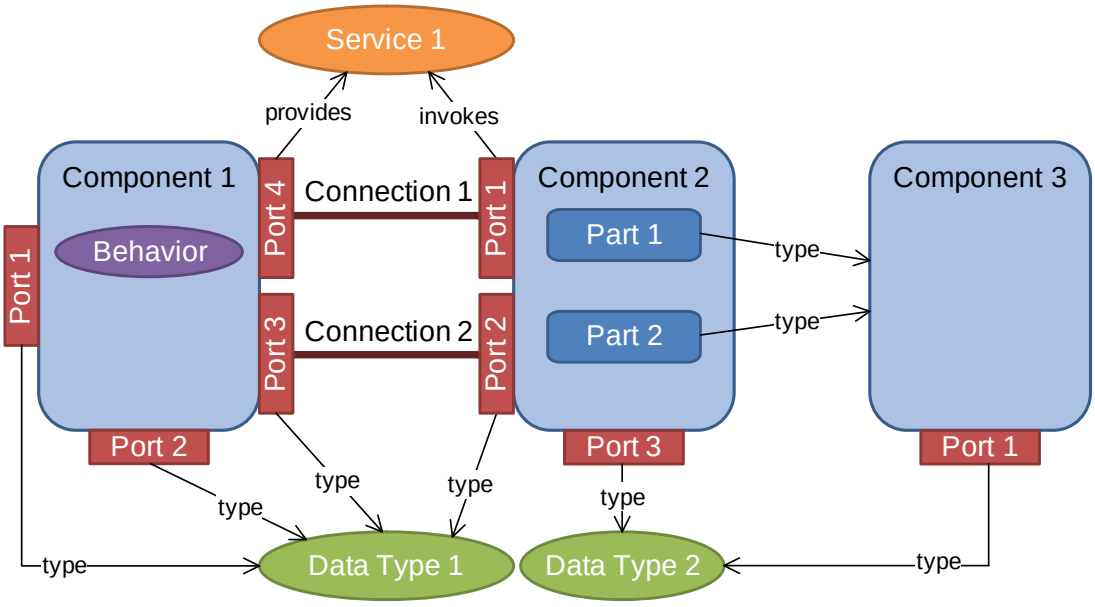

Figure 1.1: Concept of components and ports.

Components can contain parts typed by other components, enabling decomposition.

It is often useful to instantiate one component several times. For instance, a component design contains two actuators of the same kind. The respective component is defined once and instantiated twice. This provides high reuse capabilities.

1.3.3 Integration Concepts

The SPES Meta-Model will consist of a common concept core meta-model, eventual general core extensions as well as domain specific extensions.

The SPESMM Meta-Model contains common modeling concepts which are generally needed such as component based design, data types, expressions, requirements and tracebility. These concepts are based on HRC, EAST-ADL2, and MARTE.

Domain specific meta-models like EAST-ADL2 can be tailored to extend the SPESMM core meta-model. This allows domain specific models as well as a general interface to such models based on the SPES Meta-Model and therefore enables model-based tool interoperability.

For the purpose of clear packaging meachanisms meta-models which are tailored for the SPES Meta-Model shall only reference own classes, classes of parallel respectively own sub packages or classes which belong to more general concepts like the common core. Thus classes of a generalized meta-model package shall not reference classes of an extension package because this would make the extension package belong to the generalized package.

In order to integrate a foreign meta-model into the SPES artifacts and relationships which are related to common concepts of the SPES core meta-model have to be identified. Based on such a concept identification the respective meta-model can be tailored to extend the common concepts.

1.4 Differences between the Meta-Model and the Profile

Since the SPESMM is a meta-model it relies on its own semantics. Therefore a conversion of the meta-model to a profile is not an easy step since UML already provides some semantics and structures which are not always compatible with the SPESMM semantics. A second issue is the absence of a decompositional context reference.

UML only allows for a component to be decomposed *one* level deeper. So if a modeler talks about an instance within a type (part of a class) he can not know how many instances of the described type itself exist. In a deep decomposition hierarchy the user has to be able to identify the context in which the instance is used. Therefore we introduced part and port references for the mapping relation, see Section 2.2.9 for more details on the mapping relation. Since the profile would hardly be usable with such a deep context reference decomposition we “flattened” these context references, so they now are tags in the mapping stereotype.

Expressions in the meta-model are a powerful tool to specify all kinds of side-effect-free constraints or values. To make it usable in a profile for UML we decided to convert all expression associations into String attributes. These String attributes can later be parsed to expressions for further analysis.

1.5 How to read this Document

This section provides a short overview about how to read the following sections. These will show concepts of the SPES Meta-Model, the involved modeling artifacts and their relationships. Modeling artifacts will be denoted by meta-classes which are shown in class diagrams. Each meta-class which belongs to a respective context will have a description and a specification of its relationship. The description is an informal text, the relationship specification is expressed in formatted lists as listed below. Meta-classes can be concrete or abstract. Abstract meta-classes are denoted by an appended $\{abstract\}$. Meta-classes can have generalizations. Attributes have a name, a type, a multiplicity and a description. Aggregations mean that an element of a target meta-class belongs to the aggregating meta-class. Associations only mark a reference. Aggregations and associations have a name, a target, a multiplicity and a description. Meta-classes may have operations where the respective name is marked by additional brackets (). An operation has a return type and an operation which may be specified in OCL. Furthermore, there can be a numbered list of constraints on a meta-class where every constraint has a description which can additionally be specified in OCL. Enumerations are marked with an additional $\{Enumeration\}$ and have a list of elements (literals) which are described respectively.

1.6 Example Description

SPESMetaClass/SPESMetaClass $\{abstract\}$**/SPESMEnumeration** $\{Enumeration\}$

Description of the SPESMetaClass/SPESEnumeration.

Generalizations: SPESSuperMetaClass

Attributes

- attribute : Type [m..n] Attribute description.

Aggregations

- aggregation : Type [m..n] Aggregation description.

Associations

- association : Type [m..n] Association description.

EnumerationLiterals

enumLiteral Enumeration literal description

Operations

- operation(ParameterType(s)): ReturnType Description of the operation.

```
...
context SomeContext
    def: involves(someParameter: ParameterType): ReturnType
    ...
```

Constraints SPESMetaClasses are subject to the following constraints:

1. Constraint description:

```
...
context SPESMetaClass inv someInv:
    self.value.type().conformsTo(self.target.type())
    ...
```

2 Specification of the SPES Meta-Model

In this chapter concepts of the current SPES Meta-Model are described. As already mentioned before the SPES Meta-Model was created based on an integration of EAST-ADL2 from the ATESST project and HRC from the SPEEDS project. Since EAST-ADL2 is a language which is intended for the usage in the development of embedded electronic systems in the automotive domain several terms are related to cars (such as vehicle). On the contrary HRC provides a more general and formal means to describe component based design. This SPES Meta-Model is the result of an approach to unite domain specific terms and concepts of EAST-ADL2 with general formal mechanisms of HRC.

The SPES core meta-model provides a general modeling concept for embedded system design. Figure 2.1 shows an overview on the logical sections of the meta-model. These logical sections are "Components" to cover component based design, "Requirements" to cover requirements engineering and traceability as well as "Verifciation and Validation" to cover analysis techniques. They are supported by the general concept of "System Modeling" which covers the engineering process.

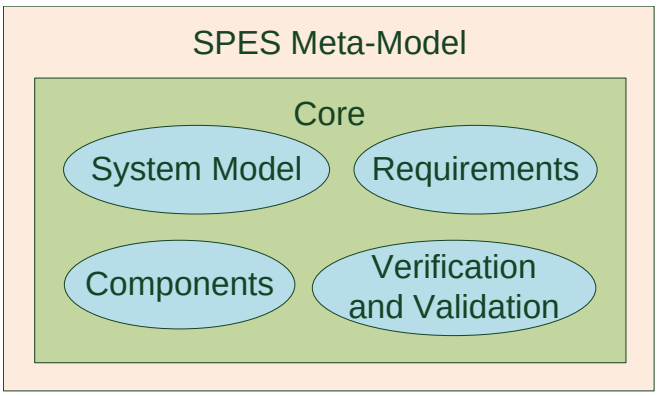

Figure 2.1: General overview over the SPESMM packages.

The core packages (SPESMM core) allow rich component models in conjunction with requirement descriptions, traceability links and the specification of verification and validation test cases. The core meta-model provides an abstraction of domain specific meta-models. It can be used as an exchange format for general analysis techniques and provides a common concept interface on various domain specific modeling

approaches. The specification of abstraction levels, perspectives, and aspects (also called viewpoints) is also possible with the SPESMM core.

2.1 Component Meta-Model

In this section a modeling concept for rich component design is presented. The approach is based on the HRC meta-model.

2.1.1 Model Elements

Model elements are elements which have specific semantics for modelling such as components, requirements, trace links, attributes etc. As depicted in Figure 2.2 they can be extended by specific domain-, user-, or tool-specific extension types and by respective extension attribute values.

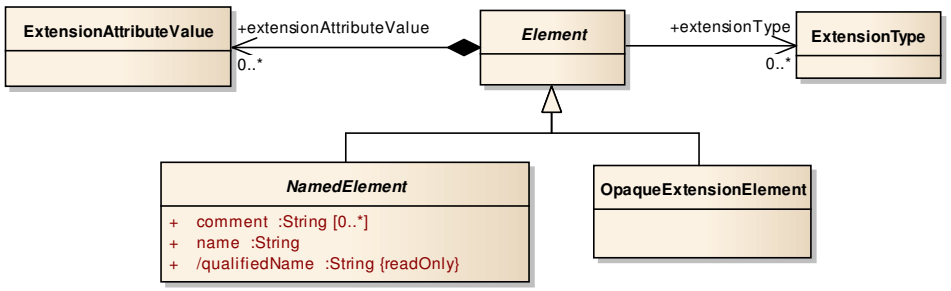

Figure 2.2: Elements.

2.1.1.1 Element {*abstract*}

An element is a model entity with defined semantics such as components, requirements, ports, attributes or trace links. An element can be extended by extension types. These define specific semantics und structure of an element. The attributes of an extension type can be used for the element by defining extension attribute values.

Element is an abstract meta-class. Its sub-classes define concrete semantics. Extension types define specific semantics for an element. Extension attribute values that are owned by an element must be instances of attributes that belong to referenced extension types of an element. There are two sub-classes of Element, namely NamedElement and OpaqueExtensionElement.

Aggregations

- extensionAttributeValue : ExtensionAttributeValue [0..*] A set of attribute values coming with domain-, user-, or tool-specific extensions.

Associations

- extensionType : ExtensionType [0..*] A set of domain-, user-, or tool-specific extension types which define additional interpretation of the model element.

Constraints Elements are subject to the following constraints:

1. Each ExtensionAttributeValue of an Element must reference an attribute of an ExtensionType that is referenced by the Element as extension type.

2.1.2 Namespaces

A name-space is a region or context within a model wherein a certain set of elements may be identified by their names; that is to say, their names serve as identifiers as depicted in Figure 2.3. To this end the elements in question must each have a name attribute with an appropriate type (String) and these names must be guaranteed to be unique within the region or context of the name-space. No explicit meta-class is provided for name-spaces. Whenever a named element is owned by some other model element, the requirement for uniqueness of names in the context of the owner will be expressed by a constraint associated with the owning meta-class.

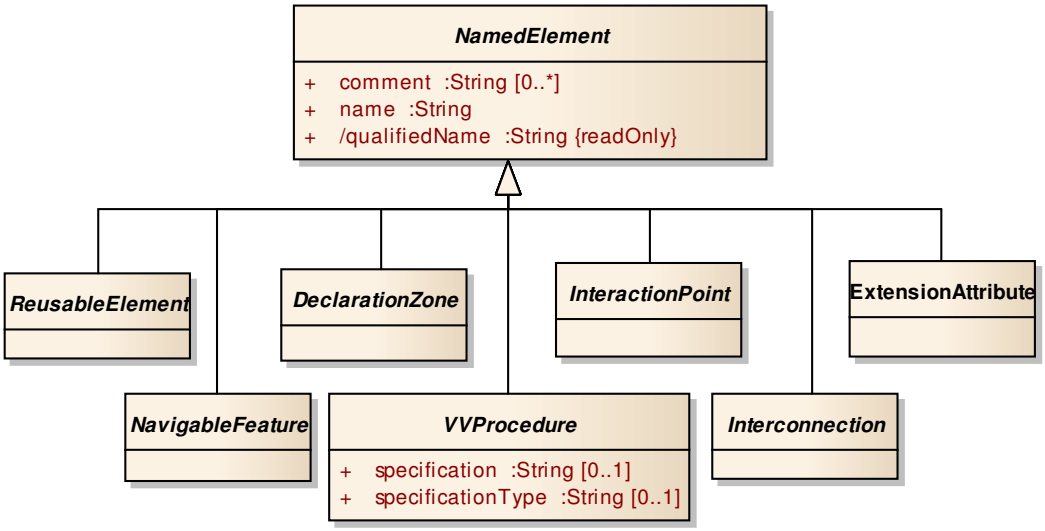

Figure 2.3: Named elements.

2.1.2.1 NamedElement {*abstract*}

Meta-class NamedElement defines the ability of a model element to have a name for the purpose of serving as its identifier. Moreover, a named element may also own a

comment string to illustrate briefly for example its purpose, rationale, speciality, etc. The name of a named element is mandatory and required not to be an empty string. This reflects the intention that named elements be uniquely identifiable by their names within the context of their owning name spaces.

Generalizations: Element

Attributes

- comment : String [0..1] An optional comment.
- name : String [1] The name of an element.
- qualifiedName : String [1] A qualified name derived from the hierarchical structure of named elements.

Constraints Named elements are subject to the following constraints:

1. The name must not be an empty string:

   context NamedElement **inv** NamesNotEmpty : `self.name <> ""`

2.1.3 Packages

As depicted in Figure 2.4 this section deals with the modularization and coarse-grained structure of HRC models. HRC models are divided into declaration zones that contain certain model elements. The model elements that are contained in a declaration zone are called reusable elements because they can be used in different contexts. Reusable elements are data types, functions, HRC blocks, port specifications, and rich components.

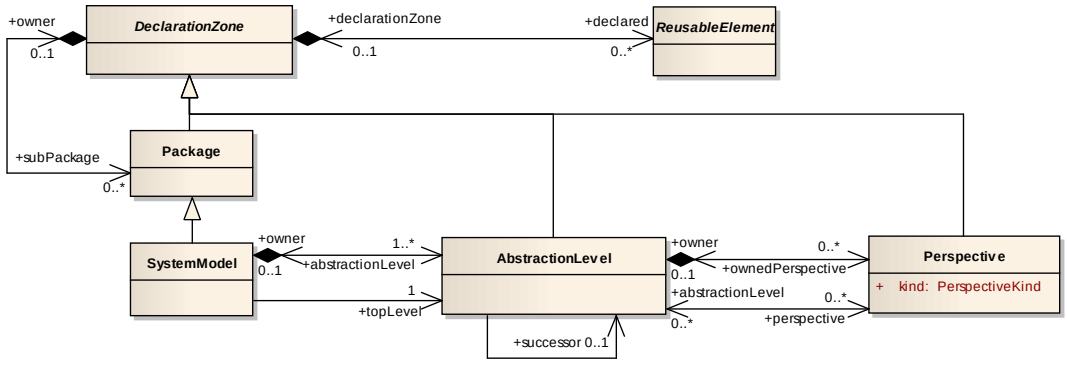

Figure 2.4: Packages.

2.1.3.1 DeclarationZone {*abstract*}

A declaration zone is a named modularization unit that contains a set of HRC reusable elements. In addition, a declaration zone can be further divided into packages. DeclarationZone is an abstract meta-class that has one subclass: Package.

A declaration zone is either a package or a system. A system is a top-level declaration zone that designates a rich component as its root. A package is a declaration zone that can be contained by another declaration zone (package or system). Packages thus allow to structure HRC models as trees.

The top-level element of an HRC model is a either a system or a package. In the latter case, the HRC model is just a set of reusable elements: it is thus a library.

This part of the meta-model does not explicitly deal with the physical partitioning of models into multiple files. As this is a purely technical (non-conceptual) issue, it is unclear to us whether it should have an impact on the meta-model.

Generalizations: NamedElement

Aggregations

- declared : ReusableElement [0..*] The set of reusable elements contained by the declaration zone.
- subPackage : Package [0..*] The set of sub-packages contained by the declaration zone.

2.1.3.2 Package

A package is a declaration zone that can be contained in another declaration zone. This meta-class allows to structure HRC models as trees. When not contained by another declaration zone, a package is the top-level element of its HRC model.

Generalizations: DeclarationZone

Associations

- owner : DeclarationZone [0..1] The declaration zone that owns the package.

2.1.3.3 ReusableElement {*abstract*}

A reusable element is an element that has a name and can be directly owned by a declaration zone. A reusable element is reusable in the sense that it can be used in different contexts. In addition, because of the ownership by DeclarationZone, reusable elements are the constituents of HRC libraries (see Package).

The subclasses of ReusableElement are BehaviorBlock, VVTarget, InterconnectionSpecification, PortSpecification, ValueFunction, FailureScope, Dimension, Unit, SystemArtefact, DataType, Constant, MappingBlock, Aspect, VVCategory, Entail, Mapping, Stakeholder, .Decompose, InformalAssertion, OperationalBlock, Derive, Satisfy and Refine.

Generalizations: NamedElement

Associations

- declarationZone : DeclarationZone [0..1] The declaration zone that owns the reusable element.

2.1.4 System Design

An embedded system is modeled along the development process. Such a development process has several steps. As depicted in Figure 2.5 a system model covering such a process therefore has several abstraction levels. In one abstraction level different kinds of model compositions can be created or regarded coming from other abstraction levels. Such model compositions belong to different perspectives such as operational, functional, logical, technical and geometrical. Models are created based on system artifacts such as requirements, rich components and VVCases. In one perspective there is one root system artifact denoting the entry element of the model of the perspective. For the model in one perspective several aspects can be regarded which refer to specific system artifacts.

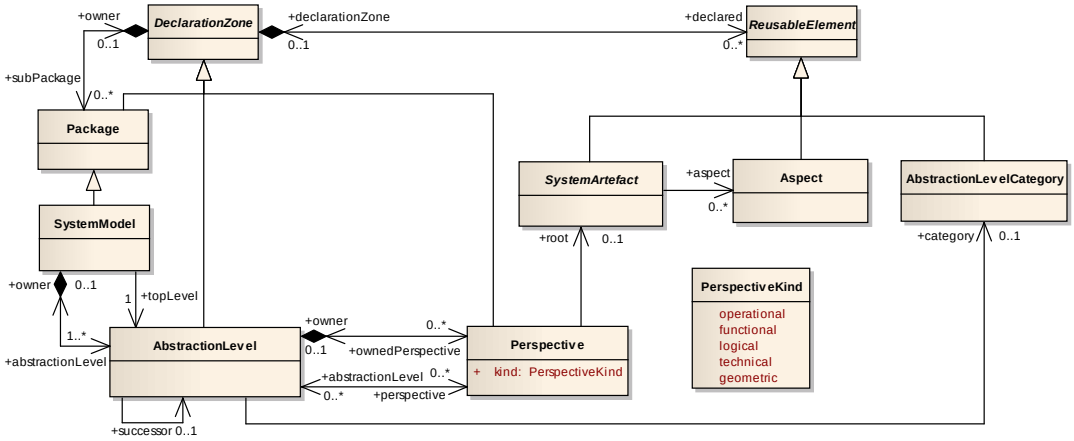

Figure 2.5: System design elements.

The evolution of an embedded system development process and related models is illustrated in Figure 2.6. The real system with the system under design is projected to

a model world. In this model world different kinds of models are created and refined until finally the system can be built in the real world. These models evolve along three axes covering the abstraction levels of the process steps and the viewpoints. As a first axis there is an ordered set of abstraction levels. These abstraction levels generally start with the definition of very coarse models and end with very low level and detailed models. A transition from one abstraction level to another level is always a refinement of models. Models defined on a lower level of abstraction realize models of the next higher abstraction level. Several kinds of composed models can be regarded on one abstraction level which cover different perpectives of the system as a second axis of the model evolution. The models of the different perspectives are related to each other via allocation links. Furthermore, within the models several non-functional aspects are regarded which are covered by different artifacts of the models. The coverage of aspects in the models therefore provides a third axis of model evolution.

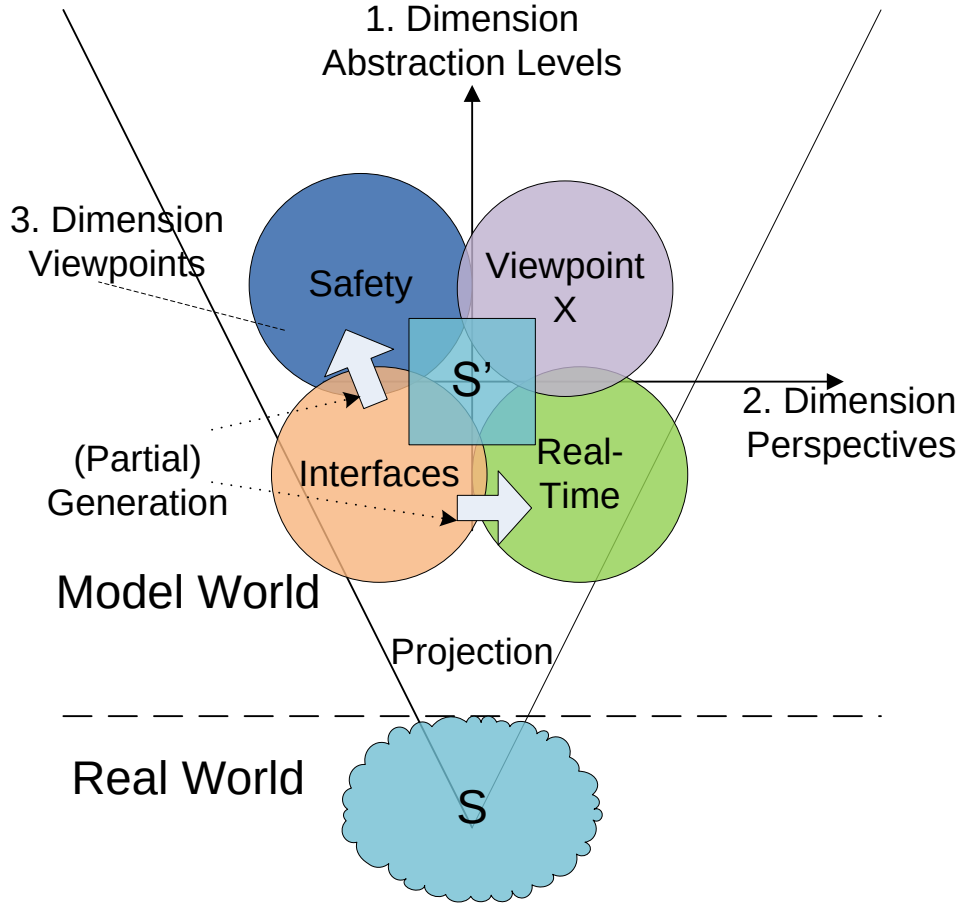

Figure 2.6: Embedded system design model evolution.

The models that are created during such a development process can be covered by a SPESMM system model. The presence of requirements and VV cases is considered the whole time. They can be introduced in any step of the process and therefore be related

to the structural elements where they are needed by using requirements traceability means as described in Section 2.3. The SPESMM methodology paper (to appear) gives a detailed description how to use abstraction levels, perspectives, and aspects in a system model.

2.1.4.1 SystemModel

The System Model of the SPESMM aims to provide the same features as the System Package in EAST-ADL2. It is supposed to be a template for how engineering information is organized and represented. The organization thereby refers to the presence of different abstraction layers and system aspects. While the number and names of abstraction layers of EAST-ADL2 are fixed, SPESMM provides a variable number of abstraction layers which can have arbitrary names. The system model only referes to the topmost abstraction level to start with.

Through the abstraction level names and the artifacts associated with each abstraction level the system architect can distinguish concerns and relevance for stakeholders. A system model is a package and can therefore be arbitrarily structured by other subpackages and contain reusable elements. As a package a system model can belong to another package if there shall be a hierarchy of system models available.

Generalizations: Package

Aggregations

- abstractionLevel : AbstractionLevel [1..*] The set of abstraction levels in the system development process.

Associations

- topLevel : AbstractionLevel [1] The highest level of abstraction of the development process. Serves as an entry point for the process.

2.1.4.2 AbstractionLevel

Related to the ISO/IEC 42010 IEEE Std 1471-2000 Standard for Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems [ISO07] an abstraction layer is a granularity/detail viewpoint constraining a component view based on level of detail or granularity criteria and regarding descriptive means. Abstraction levels are ordered, ranging from simplified to very detailed. At least one level of abstraction contributes to a design process phase. Models can be created in one abstraction level but also be referenced from different abstraction levels. Such models are part of different perspectives which can be decomposed independently.

An abstraction level can be a part of a development process or a system representation with a certain amount of information. Note that the system is represented completely in each abstraction level. However, not all perspectives may be present in each abstraction level. To determine the order of abstraction levels in a system package the system package has the topLevel association. Furthermore, each abstraction level has an association to its successor abstraction level which enables a list-like concatenation of abstraction levels.

Generalizations: DeclarationZone

Aggregations

- ownedPerspective : Perspective [0..*] A set of perspectives contained in the model of the abstraction level.

Associations

- owner : SystemModel [0..1] The owning system model of the abstraction level.
- successor : AbstractionLevel [0..1] The succeeding level of abstraction in an ordered set of abstraction levels.
- perspective : Perspective [0..*] A set of perspectives which are not neccessarily contained in the model of the abstraction level (it might be useful)
- category : AbstractionLevelCategory [0..1] The (optional) category to which this abstraction level belongs.

2.1.4.3 AbstractionLevelCategory

An abstraction level category refers to certain development processes. It thereby helps to sub divide the abstraction levels in process steps. They can also be used to clarify, where on which abstraction level(s) specific analysis tools can be utilized.

Generalizations: ReusableElement

2.1.4.4 Perspective

A perspective (related to the ISO/IEC 42010 IEEE Std 1471-2000) is a structural/architectural viewpoint constraining a component view for at least one abstraction level based on structural/architectural criteria. These structural elements can be decomposed

independently from other elements of one abstraction level. Elements of one perspective can be allocated to elements of another perspective. Furthermore, all elements of a specific perspective may constitute a target architecture description.

Perspectives allow for different description parts of the same system in one abstraction level. Thereby different modelling artifacts are used. On major distinction between perspectives is that the set of structural hierarchy driving artifacts may differ, i. e. the hierarchry of requirements may differ from the hierarchy of components. The preliminary set of available perspectives include: Operational, Functional, Logical, Technical, and Geometric. It must be evaluated if this set contains all needed perpectives or if a generic approach is more appropriate.

In order to give analysis tools a starting point in each perspective a rootElement is contained. It denotes the topmost element of the decompositional hierarchy within the abstraction level's perspective.

Generalizations: DeclarationZone

Attributes

- kind : PerspectiveKind [1] The kind of the perspective.

Associations

- root : SystemArtefact [0..1] The topmost element of the hierarchy within this perspective.
- abstractionLevel : AbstractionLevel [0..*] A set of abstraction levels which refer to the model of the perspective.
- owner : AbstractionLevel [1] The abstraction level that owns this perspective.

2.1.4.5 PerspectiveKind {*Enumeration*}

Each perspective has a perspective kind which (loosely) gives the user a hint which model artifacts to use in this perspective, e. g. the operational perspective kind allows to use artifacts like activities. The (preliminary) set of perspective kinds includes the following literals:

EnumerationLiterals

operational Denotes the operational perspective containing activities contributing to the operation of the system under design.

functional Denotes the functional perspective containing functional requirements (not necessarily user-visible) and internal functionality.

logical Denotes the logic perspective containing structural and behavioral component descriptions.

technical Denotes the technical perspective containing hardware components along with resource and scheduling modeling artifacts.

geometric Denotes the geometric perspective containing geometric information about the system.

2.1.4.6 Aspect

An aspect is a non-functional viewpoint constraining a component view for at least one perspective based on functional/non-functional criteria. It projects the dynamics specifications of architecture design from a certain point of view. It is reusable and refers to a set of system artifacts, which are relevant for the concerned aspect, which is given by the name of the aspect. SPES supports generic aspects such as safety, realtime, and others.

Generalizations: ReusableElement

2.1.4.7 SystemArtefact {*abstract*}

SystemArtefact is an abstract meta-class. Concrete sub-classes are Requirement, RichComponent and VVCase.

Generalizations: ReusableElement

Associations

- aspect : Aspect [0..*] The aspect(s) motivating the presence of the system artifact.

2.1.5 Types

Types provide the ability to declare a set of possible valuations for data items and allowed structures for elements. This section describes the type concepts which are provided by the SPES Meta-Model. Figure 2.7 displays an outline.

2.1.5.1 Type {*abstract*}

A type denotes data values or element structures. Type is an abstract meta-class and has two subclasses: DataType and ExtensionType.

Generalizations: ReusableElement

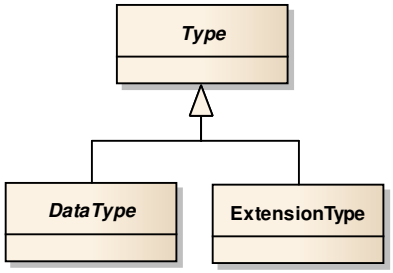

Figure 2.7: Types.

2.1.5.2 DataType {*abstract*}

A data type denotes a set of data values. DataType is an abstract meta-class and has four subclasses: Array, Record, PrimitiveType, and Enumeration. See Section 2.5 for details on data types.

Generalizations: Type

Operations

- isBoolean(): Boolean Overridden in subclasses.

  context PrimitiveType::isBoolean(): **Boolean**
  post: result = (self.kind = PrimitiveTypeKind::boolean)

- isReal(): Boolean Overridden in subclasses. Specified as for isBoolean() above.
- isInteger(): Boolean Overridden in subclasses. Specified as for isBoolean() above.
- isString(): Boolean Overridden in subclasses. Specified as for isBoolean() above.
- isVoid(): Boolean Overridden in subclasses. Specified as for isBoolean() above.
- isNumber(): Boolean Returns true if the type in question is intended to be used with arithmetic operators.

  context DataType::isNumber(): **Boolean**
  post: result = (self.isInteger() **or** self.isReal())

- conformsTo(DataType): Boolean Specifies whether the types denoted by self and other are compatible.

2.1.6 Constants

As depicted in Figure 2.8 Constant values such as a natural number are reusable elements, which can be referenced by Expressions. These entities, also known as rigid variables, could be shared across multiple HRC models by packaging them into libraries.

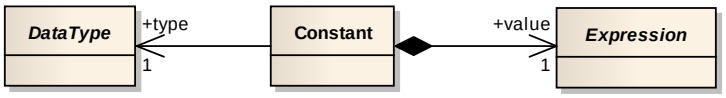

Figure 2.8: Constants.

2.1.6.1 Constant

A Constant inherits from NavigableFeature and ReusableElement. A Constant is associated to exactly one DataType by its type and owns exactly one value, which is a statically computable Expression.

Generalizations: NavigableFeature, ReusableElement

Aggregations

- value : Expression [1] The value of the constant.

Associations

- type : DataType [1] The type of the constant.

Constraints Constants are subject to the following constraints:

1. The value of a Constant must be statically computable.

2.1.7 Textual Elements

Some elements contain textual descriptions. Such a textual description can be optional but there are special opaque elements which are only represented by their contained text. Figure 2.9 gives an overview over all textually representable elements.

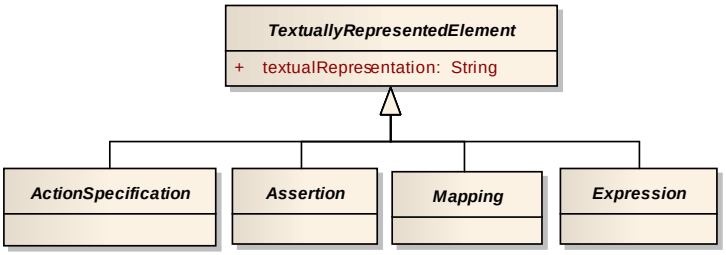

Figure 2.9: Textually represented element.

2.1.7.1 TextuallyRepresentedElement {*abstract*}

This abstract meta-class allows model elements to carry a textual representation of themselves. This representation is stored in a String attribute that is allowed to be empty. This means that the textual representation is actually optional. This meta-class has six subclasses: Assertion, Mapping, Expression, Requirement, and ActionSpecification.

Attributes

- textualRepresentation : String [1] The textual representation as a String.

2.1.8 Values

This section deals with a general value concept, which considers expressions and element referencing. Figure 2.10 depicts the general meta-class structure of values in the SPESMM Core. Note for the profile usage all expressions are represented as Strings!

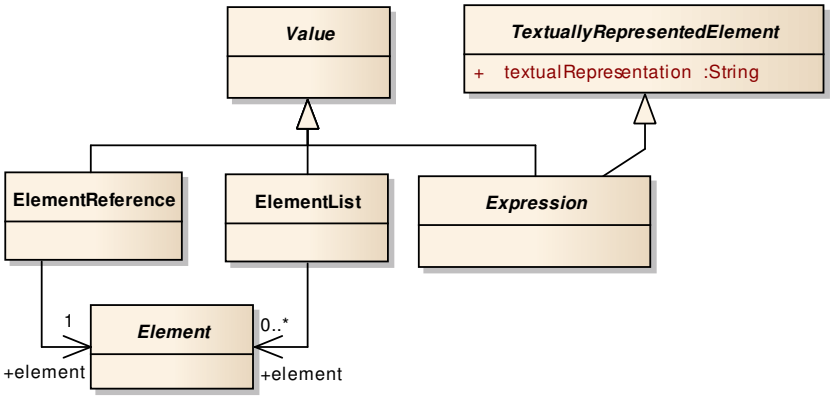

Figure 2.10: Values.

2.1.8.1 Expression {*abstract*}

The abstract Expression meta-class denotes all the available expressions in Contract models. The evaluation of an expression is guaranteed to be side-effect-free, by contrast with action specifications.

Expression has sub-classes that correspond to the syntactical means by which expressions are specified: constant literals, instance specifications for records and arrays, built-in binary or unary expressions, navigation expressions and function calls.

An expression may carry a textual representation of itself since Expression inherits from TextuallyRepresentedElement.

Whenever subclasses of Expression have associations with Expression or a subclass, these associations are always defined as composite aggregations, which ensures that expressions are organized in trees.

Generalizations: Value

Operations

- type(): DataType Determines the data type of the tested expression. Every expression has a data type. This abstract query is redefined in subclasses.
- conformsTo(DataType): Boolean Determines whether the tested expression has a type that conforms to a given data type.

  context Expression::conformsTo(dt: CoreDataType): **Boolean**
  post: result = (self.type().conformsTo(dt))

- intValue(): Integer If the expression has Integer type, this operation returns the value obtained by evaluating the expression.
- boolValue(): Boolean If the expression has type Boolean, this operation returns the value obtained by evaluating the expression.
- realValue(): Real If the expression has type Real, this operation returns the value obtained by evaluating the expression.
- stringValue(): String If the expression has type String, this operation returns the value obtained by evaluating the expression.

Constraints Expressions are subject to the following constraints:

1. Expressions must be correctly typed. This general constraint is defined precisely in sub-classes.

2.1.8.2 ElementReference

An Element Reference denotes a referenced element. Element Reference inherits from Value.

Generalizations: Value

Associations

- element : Element [1] The referenced element.

2.1.8.3 ElementList

An Element List denotes a list of referenced elements. Element List inherits from Value.

Generalizations: Value

Associations

- element : Element [0..*] The referenced elements.

2.1.9 Navigable Elements

Some elements can be directly referenced and therefore referenced from expressions. Such elements are called navigable features. Figure 2.11 gives an overview.

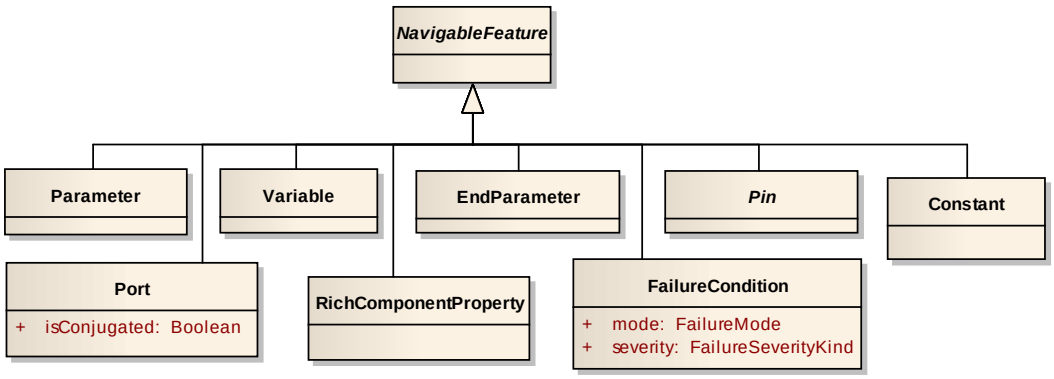

Figure 2.11: Navigable elements.

2.1.9.1 NavigableFeature {*abstract*}

This meta-class encompasses the features (model elements) which can be navigated, or referenced, in expressions. The subclasses are: Parameter, Port, RichComponent-Property, Pin, FailureCondition, Constant, EndParameter, and Variable.

Generalizations: NamedElement

Operations

- type(): Type Returns the data type of the navigable feature, if existant. Otherwise, returns oclUndefined.

2.1.10 Templates

A template defines a re-usable parameterized definition of a family of model elements as depicted in Figure 2.12. Members of such families are similar and differ from one another only in so far as they may have different values associated with the parameters of the template.

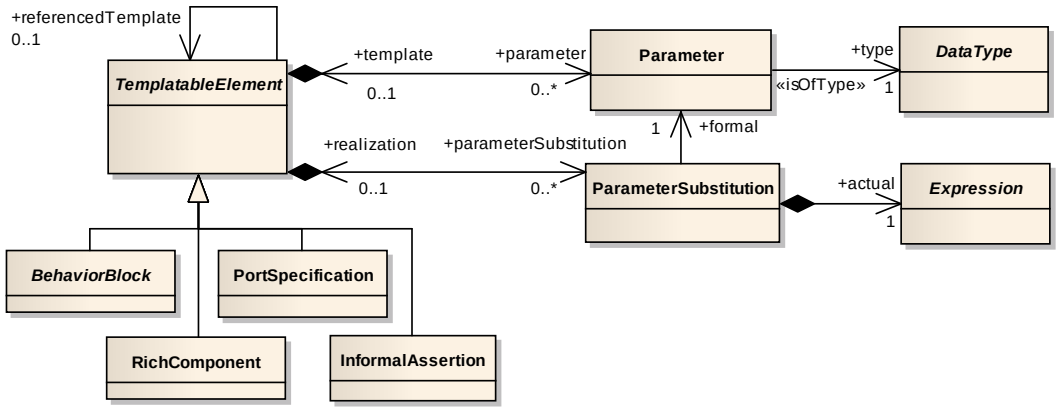

Figure 2.12: Templates.

2.1.10.1 TemplatableElement {*abstract*}

TemplatableElement defines model elements that can be created in the form of a template. A templatable element, in the role of template, may own a number of parameters.

And in the role of template realization, a templatable element may own a number of parameter substitutions which are the model elements that serve to bind formal parameters to actual values. A Templatable element TI in the role of template realization will also have a template association with the templatable element T, in the role of

template, whose parameters are being bound by the parameter substitutions owned by TI.

Aggregations

- parameter : Parameter [0..*] The parameters of this template.
- parameterSubstitution : ParameterSubstitution [0..*] The bindings of this templatable element realization.

Associations

- template : TemplatableElement [0..1] If present, the templatable element at the end of the template association is the template that this templatable element is realizing.

Operations

- isTemplate() : Boolean Returns true iff this templatable element is playing the role of template, i. e. if it has parameters.

  context TemplateableElement
  post: result = self.parameter→notEmpty()

- isRealization() : Boolean Returns true iff this templatable element is playing the role of template realization, i. e. if it has parameter substitutions.

  context TemplateableElement
  post: result = self.parameterSubstitution→notEmpty()

Constraints Templatable elements are subject to the following constraints:

1. A templatable element that is playing the role of template realization must have a corresponding template (a templatable element that is playing the role of template accessible via the template role):

   context TemplatableElement **inv** RealisationsHaveTemplates:
   self.isRealisation() **implies** self.template→notEmpty()

2. All the parameter substitutions of a template realization are for parameters that belong to the template of this realization:

   context TemplatableElement **inv** MySubstitutionsAreForMyTemplateFormals:
   self.parameterSubstitution→forAll(ps |
   self.template.parameter→includes(ps.formal))

2.1.10.2 Parameter

Instances of Parameter are formal parameters in the model. Parameters are named elements with an association with DataType that indicates the intended type of the actual elements. Formal parameters are associated with templates (TemplatableElement), interconnection specifications, services, functions and rich connectors.

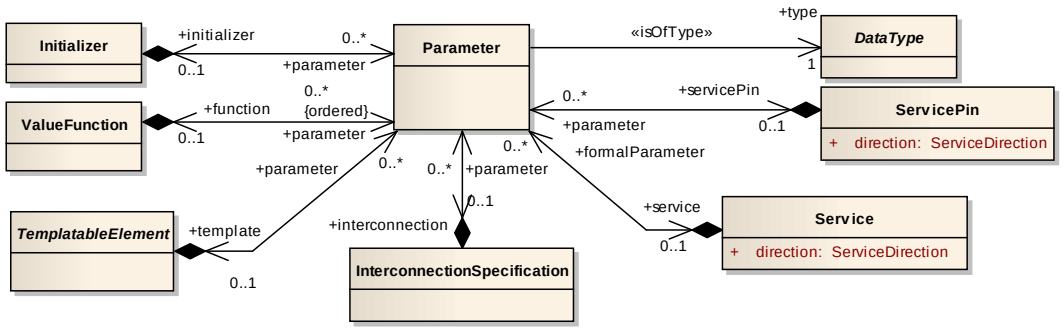

Figure 2.13: Parameters.

Generalizations: NavigableFeature

Associations

- type : DataType [1] The type of values that will be bound to this parameter.
- service : Service [0..1] If present, the service that owns this parameter.
- servicePin : ServicePin [0..1] If present, the service pin that owns this parameter.
- function : Function [0..1] If present, the function that owns this parameter.
- template : TemplatableElement [0..1] If present, the template that owns this parameter.
- interconnection : InterconnectionSpecification [0..1] If present, the interconnection specification that owns this parameter.
- initializer : Initializer [0..1] If present, the initializer that owns this parameter.

2.1.10.3 ParameterSubstitution

A parameter substitution is a model element that serves to bind the formal parameters of a template or interconnection specification with actual values in the context of an realization of that template.

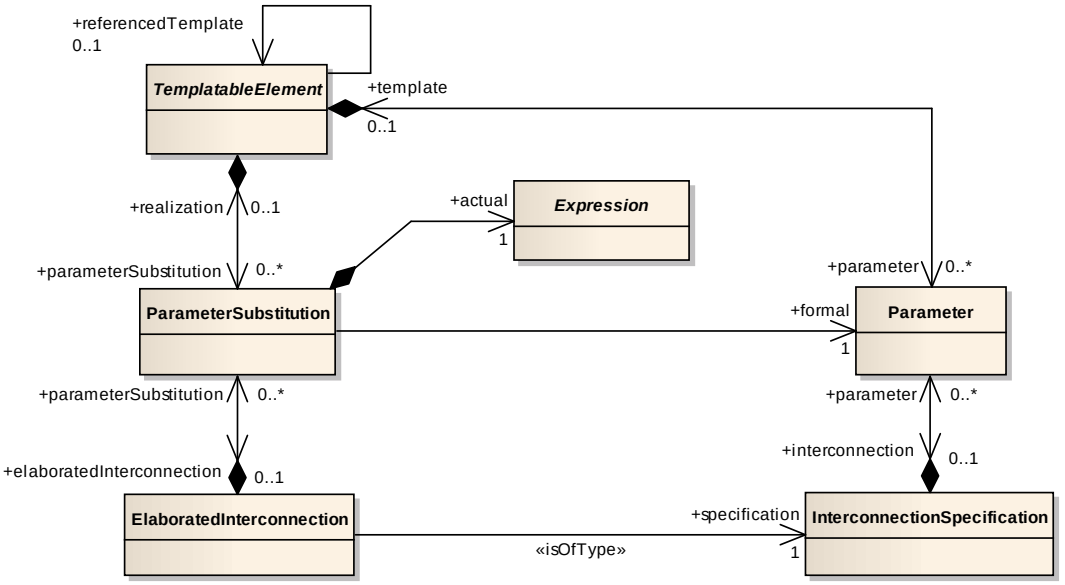

Figure 2.14: Parameter Substitution.

Aggregations

- actual : Expression [1] An expression which gives the actual value to which the formal parameter.

Associations

- formal : Parameter [1] The formal parameter being bound.
- template : TemplatableElement [0..1] If present, templatable element, in the role of template realization, who owns this parameter substitution.
- elaboratedInterconnection : ElaboratedInterconnection [0..1] If present, the elaborated interconnection that owns this parameter substitution.

Constraints Parameter substitutions are used by both by templatable elements and by elaboration specifications as illustrated in Figure 2.14. Parameter substitutions are subject to the following constraints:

1. A parameter substitution is owned either by a templatable element (in the role of actualization) or by an elaborated interconnection:

   context ParameterSubstitution **inv** ownedByTemplateOrInterconnection:
   self.template→notEmpty() **xor** self.elaboratedInterconnection→notEmpty()

2. The formal parameter of a parameter substitution that is owned by a template realization, must be owned by the corresponding template:

   context ParameterSubstitution **inv** templateOwnsFormals:
   self.template→notEmpty() **implies**
   self.template.template.parameter→includes(self.formal)

3. The formal parameter of a parameter substitution that is owned by an elaborated interconnection, must be owned by the corresponding interconnection specification:

   context ParameterSubstitution **inv** interconnectionOwnsFormals:
   self.elaboratedInterconnection→notEmpty() **implies**
   self.elaboratedInterconnection.specification.parameter→includes(self.formal)

4. The actual parameter must be statically computable:

   context ParameterSubstitution **inv** actualIsStaticallyComputable:
   self.actual.isStaticallyComputable()

5. The types of the actual and formal match:

   context ParameterSubstitution **inv** ActualTypeMatchesFormal:
   self.actual.type().conformsTo(self.formal.type)

2.1.11 Multiplicities

Certain of the elements of a Contract model serve as the specification that a number of model elements should exist in a real system. The number of elements specified is indicated by an integer size associated with the element.

The multiplicity elements are rich component properties and ports. This means a rich componentmay be defined with a part p of size 3, for example, which defines a component with three ports of name p, each with the same port specification and accessible via the notation p[0], p[1] and p[2].

2.1.11.1 MultiplicityElement {*abstract*}

MultiplicityElement is the abstract super-class of all model elements that have a size for the purpose of indicating the number of instances to be created in the model for execution and simulation purposes.

Aggregations

- size : Expression [1] An expression which holds the actual size of this multiplicity element.

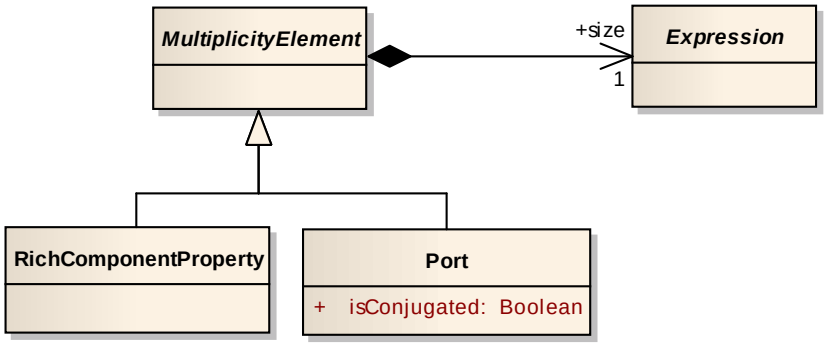

Figure 2.15: Multiplicities.

Constraints Multiplicity elements are subject to the following constraints:

1. The size expression must be of Integer type:

   context MultiplicityElement **inv** sizeIsInteger:
   self.size.type().isInteger()

2. The size expression must be statically computable:

   context MultiplicityElement **inv** sizeIsStaticallyComputable:
   self.size.isStaticallyComputable()

2.1.12 Rich Components

In this section rich components and their dependencies are presented. As depicted in Figure 2.16 a component reflects structural modeling but also behavioral aspects which are shown in Figure 2.17.

Figure 2.17 shows component behavior concepts like component initialization, service implementations as well as behavior descriptions with behavior links that are described in Section 2.2.8 and component operation concepts like failure conditions as described in Section 2.4.

2.1.12.1 RichComponent

A rich component describes a type of structural unit (by contrast to an HRC block that denotes a type of behavior unit) that share the same characterization of features, constraints, and dynamics.

A rich component is both, a reusable element and a templatable element. A rich component in the role of template realization implicitly specifies the features and dynamics defined by its template, in which all template parameters are replaced by the actual ones.

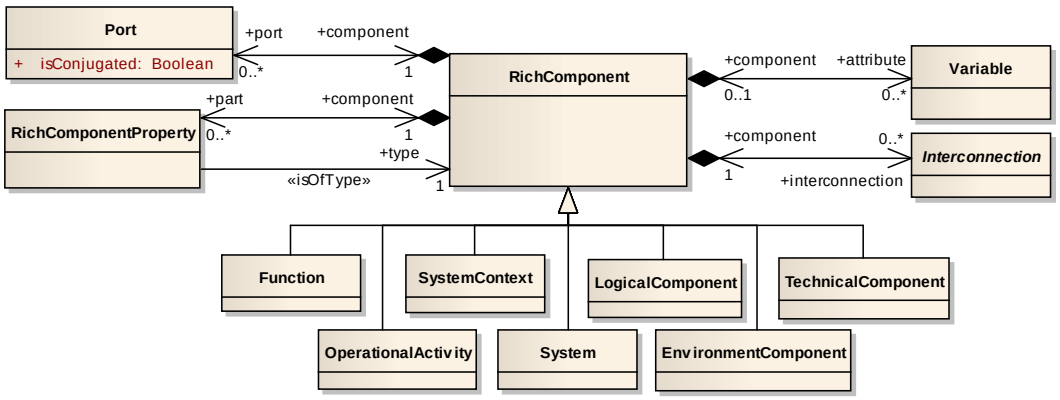

Figure 2.16: Rich Component structure.

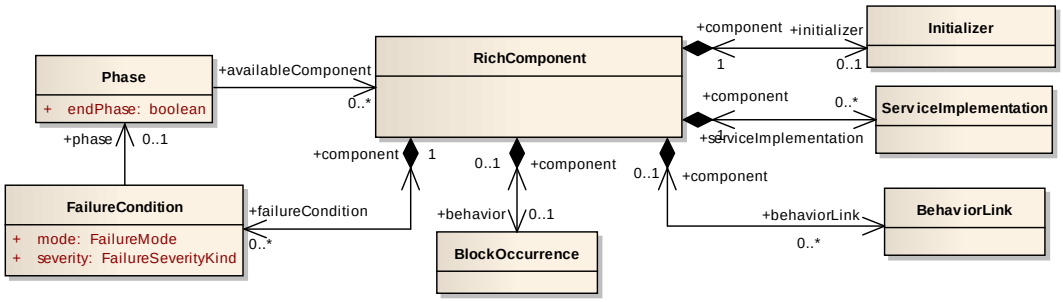

Figure 2.17: Rich Component behavior.

The structural features of a rich component include rich component properties, ports, and local variables. Rich components communicate data-typed values with their environments via the interaction points aggregated in their ports. That is the flows or services owned by the port specifications that type these ports. In later course, we also call these interaction points directly "the flows (or the services) of the rich component". Flows can be of either the following three directions: in, out, or bidirectional, indicating the owning component either inputs, outputs, or does both, values via the flows. By contrast, a service is either provided or required by the owning rich component. When a service is provided, the rich component meanwhile owns a corresponding service implementation for it.

In addition, a rich component may own local variables that are of data types, and rich component properties whose types are also rich components. The former define the attributes of the rich component, and the latter identify the parts of the rich component, i. e. its sub-components. Note that both attributes and parts of a rich component are private, in the sense that neither of themare visible outside the rich component. The only access points between the rich component and its environment are the interaction

points in the ports of the rich component.

A rich component may optionally own an initializer. The purpose of the initializer is to assign initial values to the attributes of the instances of the rich component. Moreover, if the rich component has nested sub-components, in the body of the initializer, the corresponding initializers of the rich components that type these nested sub-components can also be invoked to initialize the sub-instances (see Initialization-Call).

The inter-connection configuration of a rich component and its sub-components are specified by its interconnections. Two kinds of interconnections can be specified statically: the finer one — bindings (including both flow bindings and service bindings), which bind the interaction points of the component or its sub-components; and the coarser one — connectors among ports, which serve as a short-cut to bind all the interconnection points in each connected ports with the same names. All these interconnections are owned by the rich component. In addition, a rich component may also specify its inter-connection configuration dynamically in terms of "elaboration codes".

The dynamics of a rich component is depicted by its contracts and its behavior. Contracts are fulfilled by the rich component and expressed in terms of assumptions and promises, requiring the environment to behave as assumed, and guaranteeing the rich component to behave as promised. There are two kinds of contracts: an atomic contract that owns a pair of assertions for the assumption and promise respectively, and a composite contract that is a composition of contracts using a number of operators such as: glb, parallel, and fusion. Please refer to ContractCompositionOperator for details. Finally, according to the perspective from which a contract talks about the specification of the component, contracts are categorized into viewpoints.

In contrast to contracts, the behavior of a rich component defines the inherent dynamics of the rich component, where the behavior is specified by a block occurrence to describe the dynamics at an abstracted level.

An HRC block exposes dynamics on its owned pins. Then an instance of such a pin, which is owned by block occurrences of this HRC block, will be linked to either a flow, a service, or an attribute of the rich component that owns this block occurrence, to state that the dynamics of the component on the flow, the service, or the attribute is as specified by the HRC block on the linked pin. Such links are owned by the same rich component.

Generalizations: TemplatableElement, ReusableElement

Aggregations

- attribute : Variable [0..*] The set of attributes owned by the rich component.
- part : RichComponentProperty [0..*] The set of sub-components owned by the rich component.
- port : Port [0..*] The set of ports owned by the rich component.

- interconnection : Interconnection [0..*] The set of interconnections owned by the rich component.
- initializer : Initializer [0..1] The initializer of the rich component.
- behavior : BlockOccurrence [0..1] The behavior owned by the rich component.
- serviceImplementation : ServiceImplementation [0..*] The implementations of services owned by the rich component.
- behaviorLink : BehaviorLink [0..*] The set of links owned by the rich component.
- failureCondition : FailureCondition [0..*] Set of conditions for failures during the operation of the component.

Operations

- isGrayBox() : Boolean This operation, used in constraints, tells whether a rich component is a gray box or not (if not, then it is black box).

```
context RichComponent
    post: result = self.part→notEmpty()
```

Constraints Rich components are subject to the following constraints:

1. The operation hasSubcomponentOfType(rc: RichComponent) is used to determine whether the given rich component appears as the type of a direct or indirect part (i. e. a subcomponent) of the current rich component:

```
context RichComponent
    def: hasSubcomponentOfType(rc: RichComponent) =
        self.part→exists(p | p.type = rc or p.hasSubcomponentOfType(rc))
```

2. A rich component can never own, directly or recursively, a part/sub-component of its own type:

```
context RichComponent inv noRecursiveParts:
    not self.hasSubcomponentOfType(self)
```

2.1.12.2 OperationalActivity

An operational activity is a rich component that denotes an activity in the operational perspective.

Generalizations: RichComponent

2.1.12.3 Function

A function is a rich component that denotes a function in the functional perspective that shall be fulfilled by the system.

Generalizations: RichComponent

2.1.12.4 LogicalComponent

A logical component is a rich component that denotes a logical system component in the logical perspective.

Generalizations: RichComponent

2.1.12.5 System

A system is a rich component that denotes a system under design in the logical perspective. A system may be decomposed by instances of logical components.

Generalizations: RichComponent

2.1.12.6 EnvironmentComponent

An environment component is a rich component that denotes a component of the environment, such as an actor, that interacts with the system(s) under design in the logical perspective.

Generalizations: RichComponent

2.1.12.7 TechnicalComponent

A logical component is a rich component that denotes a technical system component in the technical perspective. The TechnicalComponent is used to describe functional hardware. The TechnicalComponent is a structural entity that is part of an electrical architecture. Through its ports it can be connected to other electrical hardware components. Its sub-meta-classes add more information about which hardware component is represented. The TechnicalComponent is only to be used in case the designer does not know more specific details about the hardware to represent (the sub-meta-classes should be the primarily used hardware modeling artifacts).

Generalizations: RichComponent

2.1.13 RichComponents and Parts

The definition of component parts provides a means for arbitrary hierarchies of component composition. As depicted in Figure 2.18 RichComponentProperties are parts of RichComponents and as well typed by respective RichComponents. Thus composed components can be reused in another component context by being assigned to a component part as a type. Furthermore, a RichComponentProperty can be assigned a size which denotes its multiplicity and expresses that a component owns a RichComponentProperty typed by one RichComponent several times.

Figure 2.18: Rich Component Property.

2.1.13.1 RichComponentProperty

A rich component property specifies a part/sub-component of a rich component. A rich component property is both a navigable feature, and a multiplicity element.

Generalizations: NavigableFeature, MultiplicityElement

Associations

- type : RichComponent [1] The type of the rich component property.
- component : RichComponent [1] The rich component that owns the rich component property as a part.

2.1.14 Componenets and Ports

The concept of ports provides a means of interface specification for components. As depicted in Figure 2.19 a ports type can be specified. A port can be referenced from expressions as well. Furthermore a port can have asize which denotes its multiplicity and expresses that a component owns a port of one type several times.

2.1.14.1 Port

A port is owned by a rich component. It aggregates a set of interaction points of the owning rich component. A port is typed by a port specification, which specifies the set of interaction points in terms of flows or services.

A port is both a navigable feature, and a multiplicity element.

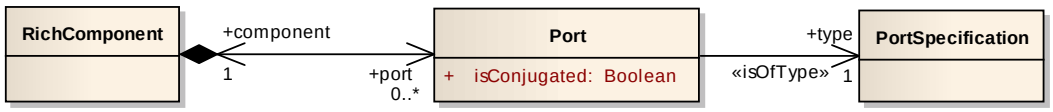

Figure 2.19: Ports.

Generalizations: NavigableFeature, MultiplicityElement

Attributes

- isConjugated : Boolean [1] If true, the flow/service directions defined in the corresponding port specification are all treated as if reversed. By default, the value is false. This facility allows to specify peer ports (i. e. with the same flows/services but complementary directions.) For flows, the complementary directions are: in for out, out for in, and bidirectional for bidirectional. For services, the complementary directions are: required for provided, and vice versa.

Associations

- type : PortSpecification [1] The port specification that types the port.
- component : RichComponent [1] The rich component that owns the port.

2.1.15 Components and Attributes

In this section the concept of variables as depicted in Figure 2.20 is described.

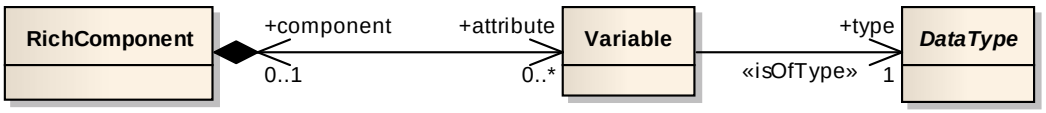

Figure 2.20: Variables.

2.1.15.1 Variable

A variable associates a name with a data type. It represents a permanent storage unit for a value of the data type. The name is inherited from meta-class NamedElement via NavigableFeature.

Variables are used to define attributes for rich components, local variables for composite actions and local variables for machine blocks.

Generalizations: NavigableFeature

Associations

- type : DataType [1] The data type of the variable.
- component : RichComponent [0..1] If present, specifies the rich component that owns the variable as an attribute.
- compositeAction : CompositeAction [0..1] If present, specifies the composite action that owns this local variable.
- block : MachineBlock [0..1] If present, specifies the machine block that owns this local variable.

2.1.16 Components and Interconnections

As described in Section 2.1.13 properties of components and component parts can be interconnected. In this section interconnections between ports as depicted in Figure 2.21 are described.

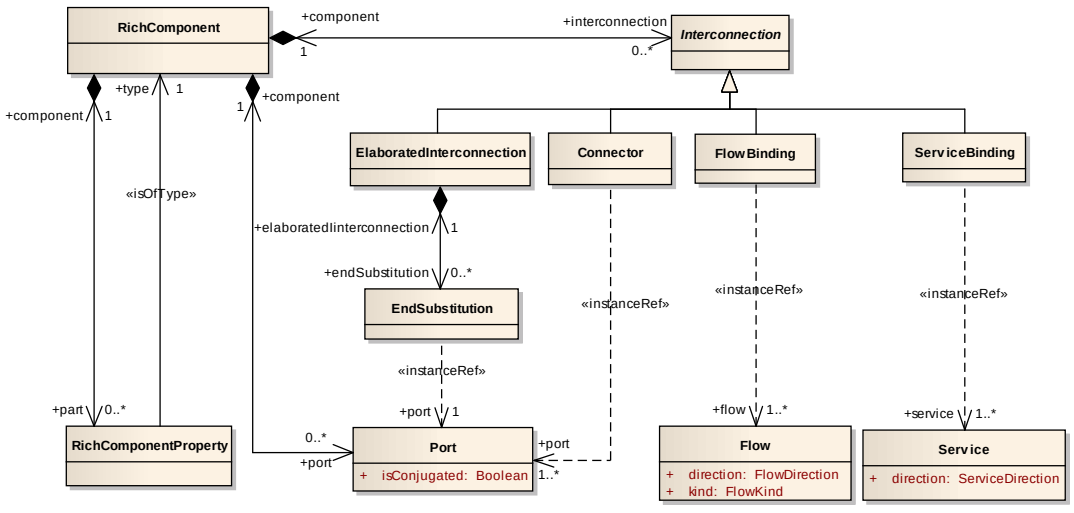

Figure 2.21: Interconnections.

2.1.16.1 Interconnection {*abstract*}

An interconnection is owned by a rich component. It specifies how the interaction points of the rich component and its sub-components are inter-connected.

An interconnection can be fixed interconnection which statically specifies how a rich component and its sub-components are inter-connected. A fixed interconnection owns

two or more interconnection ends. There are four concrete sub-classes are defined for fixed interconnections: FlowBinding that connects (instances of) flows; ServiceBinding that connects (instances of) services; Connector that connects (instances of) ports; and ElaboratedInterconnection (see Section 2.1.18). An interconnection is also a named element.

Generalizations: NamedElement

Associations

- component : RichComponent [1] Specifies the rich component that owns the interconnection.

2.1.16.2 Connector

A connector is a kind of fixed interconnections. The interconnection ends owned by a connector must be 2 or more port ends, which represent one instance (or an array of instances) of a port of the rich component that owns the connector, or of a subcomponent of the owning component.

Generalizations: Interconnection

Aggregations

- end : Connector_port [1..*] Specifies the ends of the interconnection.

Constraints Connectors are subject to the following constraints:

1. The interconnection ends owned by a connector must be port ends.
   Formal OCL constraint TBD.

2.1.16.3 Connector_port

Connector_port is an InterconnectionEnd and references exactly one port of the component that owns the connector or one instantiated port of the component's parts. In the second case the associated component parts denotes the instantiated context of the component port which is the target of the reference.

Since a ports belongs to a component specification an interconnection which shall be linked to ports of rich component parts requires a reference to the respective RichComponentProperty which is typed by the RichComponent. This is called an instance reference. Such a mechanism is realized by a connector end which references both the instantiated context and the concrete target port of the respective component as depicted in Figure 2.22.

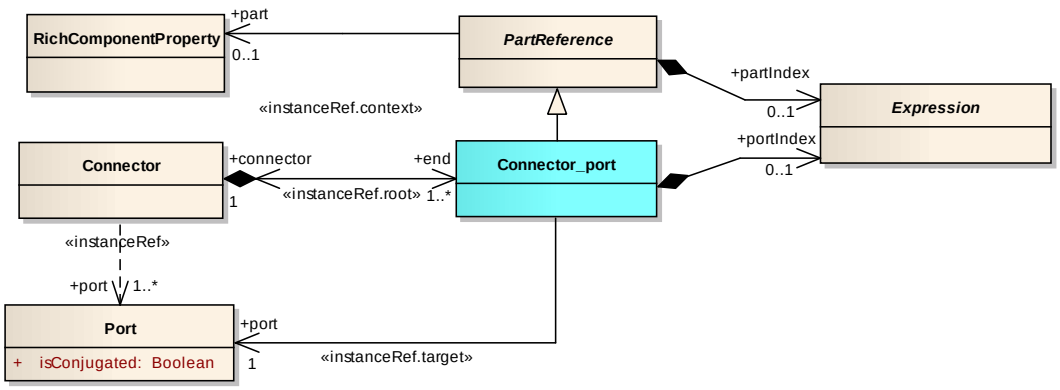

Figure 2.22: Connector references to ports of components and component parts.

Furthermore, ports and component parts are MultiplicityElements and can therefore be instantiated several times by assigning a size to the RichComponentProperty or to the Port. Such a multiple instantiation is denoted by only one RichComponentProperty respectively by one Port. When wanting to create an interconnection to one port of a multiply instantiated port which might even belong to one part of a multiple component part, the respective index of the port or part has to be given.

Generalizations: PartReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

Assosiations

- port : Port [1] Specifies the referenced port.

2.1.16.4 FlowBinding

A flow binding is a kind of fixed interconnections. The interconnection ends owned by a flow binding must be 2 or more flow ends, each of which denotes either one instance or an array of instances of a given flow referenced by the end. Instances of flows referred to by flow ends of a flow binding are all bound/inter-connected, hence should synchronize on their values for discrete/continuous flows or on their signals for event flows.

Generalizations: Interconnection

Aggregations

- end : FlowBinding_flow [1..*] Specifies the ends of the FlowBinding.

Constraints Flow bindings are subject to the following constraints:

1. The interconnection ends owned by a flow binding must be flow ends:

   context FlowBinding **inv** flowEnds:
   self.end→forAll(oclIsKindOf(FlowBinding_flow))

2. A flow binding has at least 2 ends:

   context FlowBinding **inv** atLeastTwoEnds:
   self.end→size() > 1

2.1.16.5 FlowBinding_flow

FlowBinding_flow is an InterconnectionEnd of a FlowBinding which references the exact instance of a flow. Such an instance is denoted by the the service, its containing port and the respective RichComponent or RichComponentProperty as depicted in Figure 2.23.

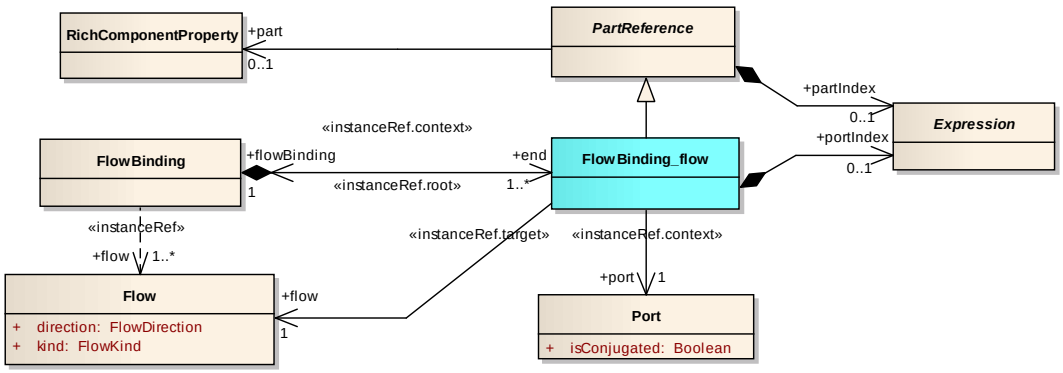

Figure 2.23: FlowBinding_flow.

Generalizations: PartReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

Assosiations

- port : Port [1] Specifies the referenced port.
- flow : Flow [1] Specifies the flow referenced.

2.1.16.6 ServiceBinding

A service binding is a kind of fixed interconnections. The interconnection ends owned by a service binding must be 2 or more service ends, each of which denotes either one instance or an array of instances of a given service referenced by the end. Note the interconnection configuration specified by service bindings whose ends referring to arrays of instances of services is similar to flow bindings, as exemplified above.

Generalizations: Interconnection

Aggregations

- end : ServiceBinding_service [1..*] Specifies the ends of the ServiceBinding.

Constraints Service bindings are subject to the following constraints:

1. The interconnection ends owned by a service binding must be service ends:

   context ServiceBinding **inv** serviceEnds:
   self.end→forAll(oclIsKindOf(ServiceBinding_service))

2. A service binding has at least 2 ends:

   context ServiceBinding **inv** atLeastTwoEnds:
   self.end→size() > 1

2.1.16.7 ServiceBinding_service

ServiceBinding_service is an InterconnectionEnd of a ServiceBinding which references the exact instance of a service. Such an instance is denoted by the the service, its containing port and the respective RichComponent or RichComponentProperty as depicted in Figure 2.24.

Generalizations: PartReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

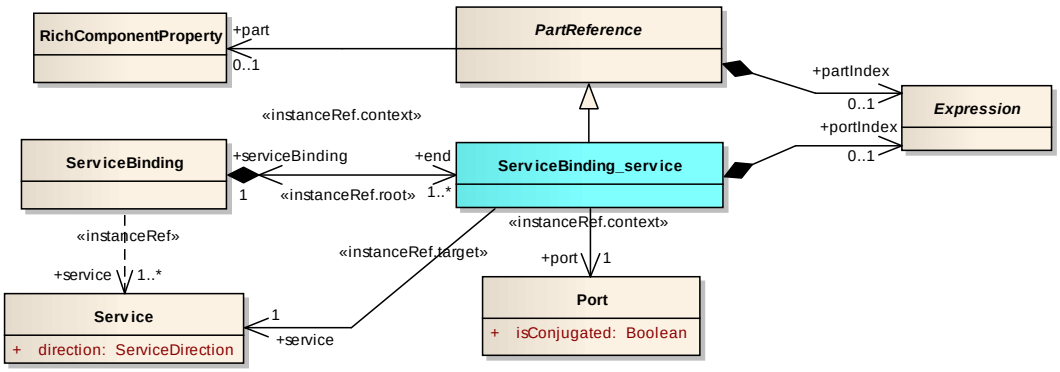

Figure 2.24: ServiceBinding_service.

Assosiations

- port : Port [1] Specifies the referenced port.
- service : Service [1] Specifies the service referenced.

2.1.16.8 PartReference {*abstract*}

PartReference is an abstract meta-class which provides the ability to reference a RichComponentProperty as a context for further references to properties of its RichComponent type. Concrete part references are Connector_port, FlowBinding_flow, ServiceBinding_service, BehaviorLink_flow and BehaviorLink_service.

Aggregations

- partIndex : Expression [0..1] In case the size of the rich component property associated is not 1, specifies the index in the multiple element.

Assosiations

- part : RichComponentProperty [0..1] In case the part of a rich component shall be referenced the respective rich component property is referenced with this association.

2.1.17 Port Specifications

As described in Section 2.1.14 a port is typed by a PortSpecification. Such a port specification denotes the data flows and services of a component port with respective types and directions as depicted in Figure 2.25.

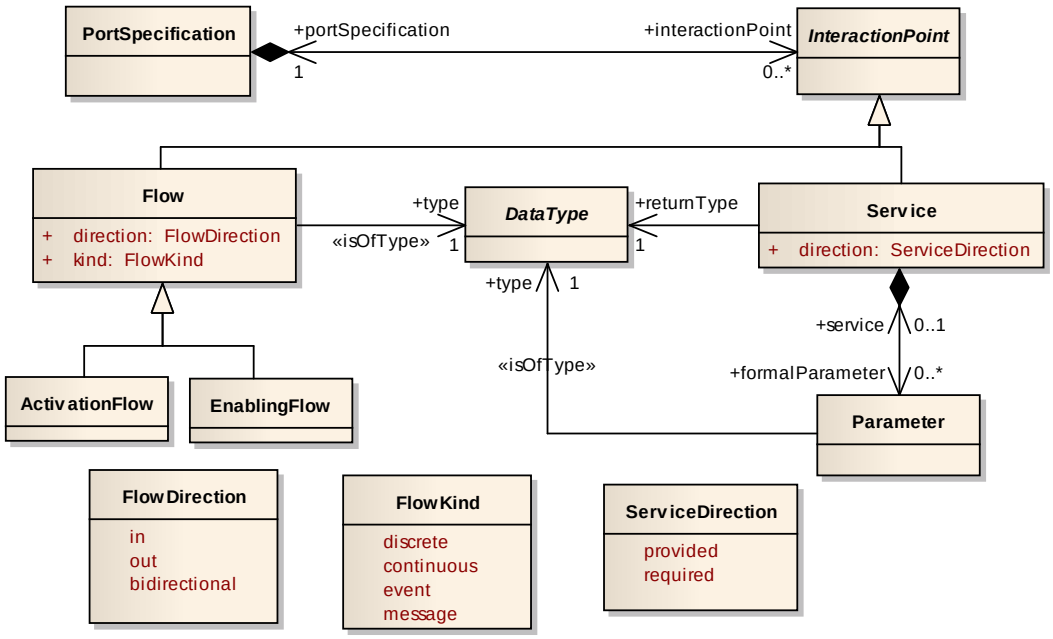

Figure 2.25: Port Specifications.

2.1.17.1 PortSpecification

A port specification owns a set of interaction points and is used to type ports. It is both a reusable element and a templatable element.

A port specification defines a (non-empty) set of interaction points that describe the interactions of any ports that have this specification as their type. A port specification can be a template and may have template parameters that specify, for example, the size of array types for its interaction points. Port specification templates are realized by instances of PortSpecification in the role of template realization, in which case they may or may not define additional interaction points.

Generalizations: TemplatableElement, ReusableElement

Aggregations

- interactionPoint : InteractionPoint [0..*] The set of interaction points owned by the port specification.

Constraints Port specifications are subject to the following constraints:

1. An instance of PortSpecification must always have at least one associated interaction point, except in the case where the port specification is playing the role of a template realization:

```
context PortSpecification inv notEmptyUnlessRealization:
    self.interactionPoint→isEmpty() implies self.isRealization()
```

2. A PortSpecification owns either Services or Flows as its interaction point, but not both.
   Formal OCL constraint TBD.

2.1.17.2 InteractionPoint {*abstract*}

InteractionPoint is an abstract class. It has two sub-classes: Flow and Service. Interaction points aggregate together in a port specification, which is used to type a port. For any rich component that owns such a port, we also informally say that this component owns the interaction points: flows and services, of the corresponding port specification. An interaction point can be referred to as an end of a binding that binds other ends referring to other interaction points. We say these interaction points are bound. There are two types of bindings: flow bindings and service bindings, to bind flows to flows, and services to services respectively. For the semantics of bound interaction points, please refer to individual sub-classes.

Moreover, an interaction point can also be referred to as an end of a link where the other end referring to a block pin with compatible meta-type, namely flow to flow pin and service to service pin. In this case, the behavior of the owning component on the interaction point is the same as the behavior of the block on the pin. An interaction point is a named element.

Generalizations: NamedElement

Aggregations

- portSpecification : PortSpecification [1] The port specification that owns the interaction points.

2.1.17.3 Flow

A flow specifies an interaction point. A flow is characterized by its data type, its kind, and its direction. The data type defines the type of values that appear on the flow. The kind defines when values are available on the flow, and how they change. And the direction defines whether the owning rich component does input/output/both on the flow.

Bound flows can be considered as one shared communication point among the owning components. As a consequence, bound flows always have the same values at any time.

A flow inherits from NavigableFeature. This allows it to initialize flows within the body of an Initializer.

Generalizations: NavigableFeature

Attributes

- direction : FlowDirection [1] Specifies the direction of the flow.
- kind : FlowKind [1] Specifies the kind of the flow.

Assosiations

- type : DataType [1] Specifies the data type of the flow.

2.1.17.4 FlowDirection {*Enumeration*}

FlowDirection is an enumeration with elements that can be used as literals for specifying input and output directions of flows. Such directions include: in, out, and bidirectional, called the three abstract flow directions.

EnumerationLiterals

in indicates that for any instance of the flow, which is owned by a port owned by a rich component, the dynamics of this rich component can only do input on this interaction point.

out indicates that for any instance of the flow, which is owned by a port owned by a rich component, the dynamics of this rich component can only do output on this interaction point.

bidirectional indicates that for any instance of the flow, which is owned by a port owned by a rich component, the dynamics of this rich component can (but not forced to) do both input and output on this interaction point.

2.1.17.5 FlowKind {*Enumeration*}

FlowKind is an enumeration with elements that can be used as literals to specify the kind of a flow. Such kinds include: discrete, continuous, and event, called the three abstract flow kinds.

EnumerationLiterals

discrete indicates an interaction point where there is always a value available and the values change only discretely at some time instants.

continuous indicates an interaction point where there is always a value available and the values either evolve continuously during some time duration or change discretely at some time instants, i. e. the trajectory is piece-wise continuous.

event indicates an interaction point where only at certain time instants there is a value available and the values may change from instant to instant.

2.1.17.6 ActivationFlow

An activation flow is the communication point through which the rich component that owns the port that owns it, communicates to the environment the possibility to execute, i. e. there is a valid outgoing transition from the current active state. Thus the direction of an activation flow is always out, the kind is discrete and the type is Boolean. ActivationFlow inherits form Flow.

Generalizations: Flow

Constraints Activation flows are subject to the following constraints:

1. An activation flow must be of kind discrete:

   context ActivationFlow **inv** isDiscrete:
   self.kind = FlowKind::discrete

2. An activation flow must be of direction out:

   context ActivationFlow **inv** isOut:
   self.direction = FlowDirection::out

3. An activation flow must be of type Boolean:

   context ActivationFlow **inv** isBoolean:
   self.type.isBoolean()

2.1.17.7 EnablingFlow

An enabling flow is a flow constituting the communication point through which the environment can control the execution of a rich component. The rich component that owns an enabling flow disables its execution (is forced to take an empty self-loop transition) if the value of this enabling flow is true. The enabling flow can be set only by the environment, thus the direction is in. The kind is discrete and the type is Boolean, since it is the counterparting coordination port of the activation flow.

Generalizations: Flow

Constraints Activation flows are subject to the following constraints:

1. An enabling flow must be of kind discrete:

   context EnablingFlow **inv** isDiscrete:
   self.kind = FlowKind::discrete

2. An enabling flow must be of direction in:

   context EnablingFlow **inv** isIn:
   self.direction = FlowDirection::in

3. An enabling flow must be of type Boolean:

   context EnablingFlow **inv** isBool:
   self.datatype.isBoolean()

2.1.17.8 Service

A service is the declaration and the implementation of a service that is provided or required by a rich component through a port specification.

Generalizations: InteractionPoint

Attributes

- direction : ServiceDirection [1] Specifies whether the service is provided or required by the hosting rich component.

Aggregations

- formalParameter : Parameter [0..*] {ordered} The input arguments of the service.

Assosiations

- returnType : DataType [1] The type the returned value of the service.

2.1.18 Elaboration of Architectures

Sometimes it is useful to be able to specify a complex internal structure for a component without explicitly creating and interconnecting all of its sub-components and their ports. This is particularly useful when one wishes to avoid specifying the number of interconnected sub-components in advance by, for example, creating a template that can be instantiated later for a specific number of sub-components.

This requirement can be, at least partially, addressed by means using specifications for certain patterns of interconnection among components (architectures) that can be used in specific contexts where a component has multiple parts (rich component properties) and where the size of the parts (number of sub-components to be created) is specified by a template parameter. In such contexts the pattern of interconnection among the rich component instances created as specified by the parts can be specified by an elaborated interconnection, which refers to an interconnection specification that determines the imperative code that will be executed to effect the interconnection of components, and the ports of those rich components that are to be connected (see Figure 2.26).

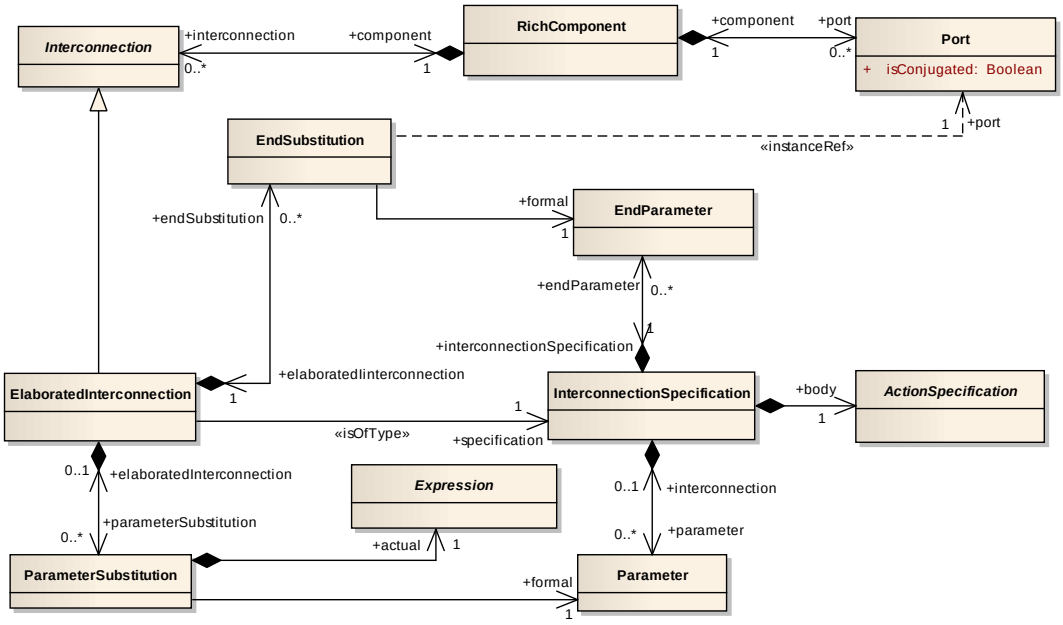

Figure 2.26: Elaboration of Architectures.

2.1.18.1 ElaboratedInterconnection

An elaborated interconnection represents the intention to interconnect certain ports in a model in a way that cannot easily be described using connectors.

The precise pattern of interconnection to be employed is defined by the interconnection specification associated by the spec association. The parameter substitutions and end substitutions serve to bind the formal parameters of the interconnection specification to actual values and ports (via port ends) in the context of this elaborated interconnection. Elaborated interconnections are owned by a rich component for whose parts it is specifying the interconnection.

Generalizations: Interconnection

Aggregations

- endSubstitution : EndSubstitution [0..*]
- parameterSubstitution : ParameterSubstitution [0..*]

Assosiations

- spec : InterconnectionSpecification [1]
- component : RichComponent [1]

2.1.18.2 InterconnectionSpecification

An interconnection specification is a reusable model element that describes a pattern of interconnection among multiple ports. It is envisioned that a standard library of interconnection specifications will be provided for use in Contract models covering such patterns as One-To-One connections, Grids, Total Graphs, Fan-In and Fan-Out, etc.

An interconnection specification has a number of end parameters, that indicate the requirement to specify ports to be connected, and may also have a number of normal parameters (those whose type is a Contract DataType) to configure interconnection process. The semantics of an interconnection specification is given by its body. The body is an action specification that typically involves loops and interconnection actions, that is connector declarations and connect actions, as illustrated in the example above. A connector declaration specifies the creation of a connector, while a connect action determines the ends of the connector, that is on which end parameters the connector is connected. Interconnection specifications are reusable elements that are owned directly by a DeclarationZone.

Generalizations: ReusableElement

Aggregations

- parameter : Parameter [0..*]
- endParameter : EndParameter [1..*]
- body : ActionSpecification [1]

2.1.18.3 EndParameter

An end parameter is a formal parameter of an interconnection specification that must be bound to a port reference by an elaborated interconnection in order to specify which ports of which rich component properties should be connected during the model's elaboration phase.

An end parameter has a name that describes its role with respect to its owning interconnection specification. The name is inherited from meta-class NavigableFeature. Inheritance from meta-class NavigableFeature also allows an end parameter to be navigated (like an array) in order to handle multiplicity. This allows to bind an end parameter to a port that has a multiplicity.

Generalizations: NavigableFeature

Assosiations

- interconnectionSpecification : InterconnectionSpecification [1]

2.1.18.4 EndSubstitution

An end substitution is a model element that serves to bind formal end parameters of an interconnection specification to port references that specify the port an d part (rich component property) that is to be connected.

Generalizations: ReusableElement

Aggregations

- actual : EndSubstitution_port [1]

Assosiations

- formal : EndParameter [1]
- elaboratedInterconnection : ElaboratedInterconnection [1]

2.1.18.5 EndSubstitution_port

The meta-class EndSubstitution_port (see Figure 2.27) denotes a port instance referenced by an EndSubstitution. It references exactly one port of the component that owns the ElaboratedInterconnection owning the Endsubstitution or one instantiated port of the component's parts. In the second case the associated component parts denotes the instantiated context of the component port which is the target of the reference. Since a port is multiplicity in the case of a given port size the exact port instance can be denoted by a portIndex.

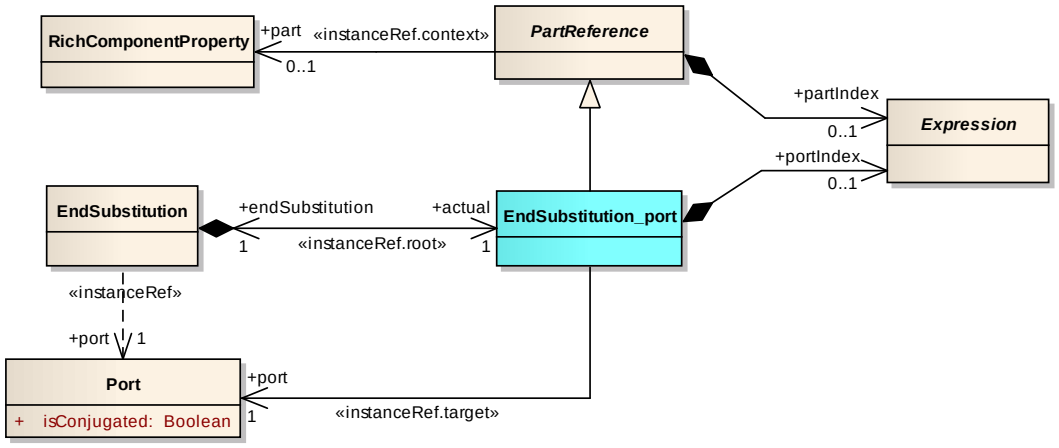

Figure 2.27: EndSubstitution_port.

Generalizations: PartReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

Assosiations

- port : Port [1] Specifies the port referenced.

2.1.19 Domain-, User-, and Tool-specific Extensions

This section deals with the coverage of domain-, user-, and tool-specific extensions for the SPESMM. The SPESMM is intended be the common meta-model for a systems engineering development process. It provides a common understanding for system engineering information information that is exchanged. Therefore it provides a common understandable abstraction from concrete domain-, user- or tool specific information. The original domain-, user-, or tool-specific information can be addressed by using the concepts that are provided by this section. These concepts and their relationship to other CMM elements are depicted in Figure 2.28.

The meta-model concepts provided by this section allow the definition of extension types. These extension types may have attributes, each being typed by a data type or by another extension type. Elements of the SPESMM can be typed by such extension types. According to the attributes being defined for the extension types elements may have instantiations of these extension attributes. These are extension attribute values.

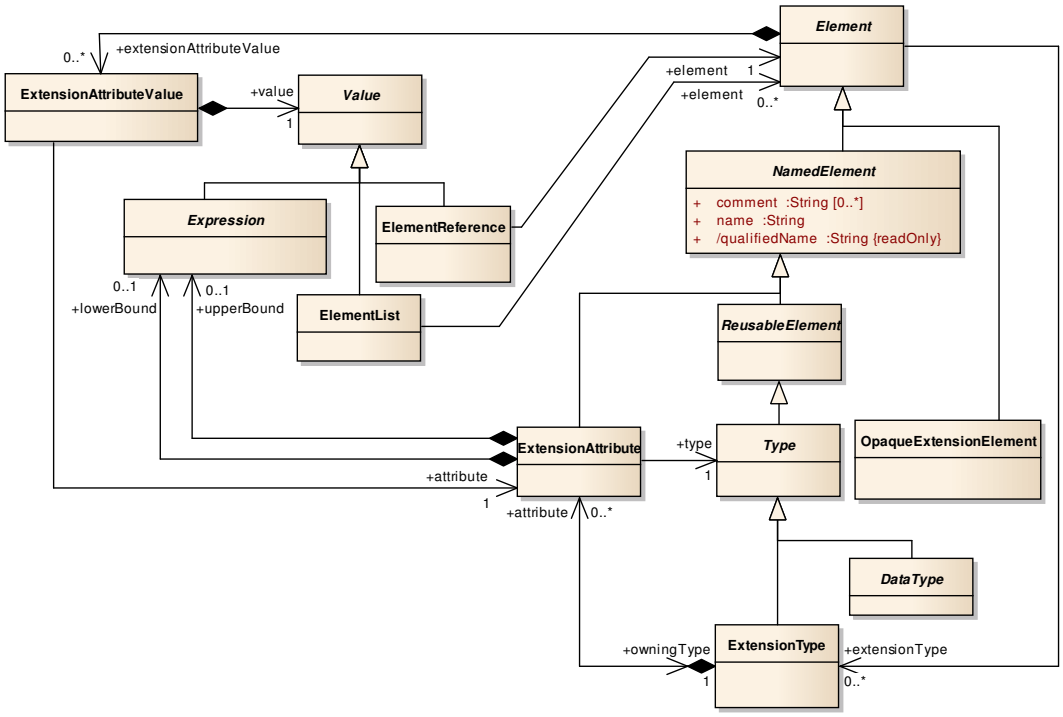

Figure 2.28: Elements for domain-, user-, and tool-specific extensions.

They define the value of an attribute for a specific SPESMM element. Extension attribute values can be either expressions or references to other elements. The types of the values are defined by the corresponding extension attributes. Therefore SPESMM elements may have attributes which are defined by another ontology or by another meta-model. This concept allows a type-safe extension of a SPESMM view by original domain-, user-, or tool-specific information which is not directly covered by the SPESMM concepts.

2.1.19.1 Extension Type

An extension type is a classifier of a SPESMM extension. Its semantics are externally defined. An extension type defines a set of extension attributes. Elements of the SPESMM can be typed by extension types in order to specialize their semantics and in order to use the attributes of the extension types.

Generalizations: Type

Aggregations

- attribute : ExtensionAttribute [0..*] The set of attributes which belong to the extension type.

2.1.19.2 ExtensionAttribute

An extension attribute defines an attribute of an extension type. It is typed by a data type or by an extension type. This defines allowed values for extension attribute values which are instantiations of extension attributes. The count of values can be limited by defining a lower and an upper bound.

Generalizations: NamedElement

Aggregations

- lowerBound : Expression [0..1] This denotes the lower bound of the attribute's multiplicity. It specifies a lower value for the allowed number of concrete values which belong to an extension attribute value when instantiating the extension attribute.
- upperBound : Expression [0..1] This denotes the upper bound of the attribute's multiplicity. It specifies an upper value for the allowed number of concrete values which belong to an extension attribute value when instantiating the extension attribute.

Assosiations

- type : Type [1] Reference to the type of the extension attribute, which can be a data type or an extension type.

2.1.19.3 ExtensionAttributeValue

An extension attribute value is an instantiation of an extension attribute. It defines a value for an attribute compliant to the referenced attribute of an extension type.

Values can be expressions, element references or element lists. This depends on the type of the referenced extension attribute. It the referenced attribute is typed by a data type then the value is defined by an expression. If the referenced extension attribute is typed by an extension type then ElementReference or ElementList is used. Whether ElementReference or ElementList is used depends on the lower and upper bound values of the extension attribute. If no lower or upper bound is defined, then implicitely a 1..1 multiplicity is considered which means that ElementReference is used. In other cases ElementList is used.

Aggregations

- value : Value [1] The denotes the concrete value of an extension attribute value. The allowed multiplicity is given by the lowerBound and the upperBound of the referenced extension attribute.

Assosiations

- attribute : ExtensionAttribute [1] Reference the instantiated ExtensionAttribute.

2.1.19.4 OpaqueExtensionElement

An opaque extension element defines a model element whose semantics is only defined by a referenced extension type.

Generalizations: Element

Constraints Opaque extension elements are subject to the following constraints:

1. An OpaqueExtensionElement must reference at least one ExtensionType.

2.2 Component Behavior Meta-Model

Rich components can be both: They are structured and may have behavior. In the following means to describe behavior, implementations and how to reference system artefacts is described.

2.2.1 Value Functions and Calls

This section deals with function descriptions and their invocations. Figure 2.29 gives an outline.

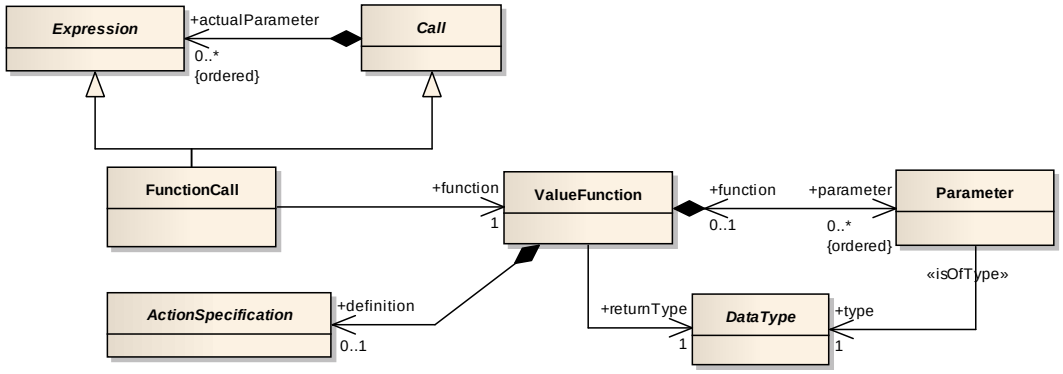

Figure 2.29: Functions and Calls.

2.2.1.1 FunctionCall

FunctionCall represents expressions corresponding to function invocations. Such expressions are syntactically obtained by relating a function (the callee) and an ordered list of actual parameter expressions.

Generalizations: Expression, Call

Associations

- function : ValueFunction [1] The value function being called.

Operations

- callee(): ValueFunction Inherited from super-meta-class Call.

```
context FunctionCall def:
    callee(): OclAny = self.function
```

Constraints Function calls are subject to the following constraints:

1. The number of the actual parameters must be the same as the number of the parameters declared for the function:

```
context FunctionCall inv formalsAnActuals:
    self.actualParameter→size() = self.function.parameter→size()
```

2. Each actual parameter must be type compatible to the corresponding formal parameter:

```
context FunctionCall inv paramTypesMatch:
    self.actualParameter→forAll(ap |
        let fp: Parameter =
            function.parameter→at(actualParameter→indexOf(ap))
        in ap.conformsTo(fp.type)
```

3. The return type is the type of the function:

```
context FunctionCall::type(): DataType
    post: result = self.function.returnType
```

2.2.1.2 ValueFunction

A value function is a reusable element that represents a pure (side-effect free) computational unit which, as usual, computes a return data value corresponding to an ordered set of argument values. A value function is specified by its name (inherited from NamedElement), the data type of the return value and data types and names of its formal parameters. A value function is owned by a declaration zone.

Generalizations: ReusableElement

Aggregations

- parameter : Parameter [0..*] $\{ordered\}$ Specifies the ordered set of parameters.
- definition : ActionSpecification [0..1] The optional action that defines this value function if the value function is a service.

Associations

- returnType : DataType [1] Specifies the return type of the value function.

Constraints Value functions are subject to the following constraints:

1. Parameter names are unique with respect to their owning value function:

   context ValueFunction **inv** uniqueParameterNames:
   self.parameter→isUnique(p | p.name)

2.2.1.3 Call $\{abstract\}$

Call is an abstract meta-class. A call relates a callee (which can be implicit) with a list of actual parameters.

Aggregations

- actualParameter : Expression [0..*] $\{ordered\}$ The actual parameter list.

Operations

- callee(): Void Determines the callee. This query must be redefined by subclasses.

  context Call def: callee(): **OclAny** = OclUndefined

2.2.2 Component Initialization

In this section the initialization of components is described.

2.2.2.1 Initializer

An initializer allows specifying the initial value of the attributes of the owning rich component. An initializer may have formal parameters. Its behavior (body) is defined by an action specification. The role of that action is, according to the actual values of the parameters, to assign attributes of the rich component and call initializers of the direct sub-components (rich component properties) via initialization calls. This recursive mechanism allows the initialization of the whole tree structure of components in a component-based system.

The initializer of a rich component is executed when it is called by the body of another initializer. The initializer of the root component of a system is implicitly called whenever the entire system needs to be instantiated and initialized, e. g. for simulation. Figure 2.30 displays the initializer and its relations.

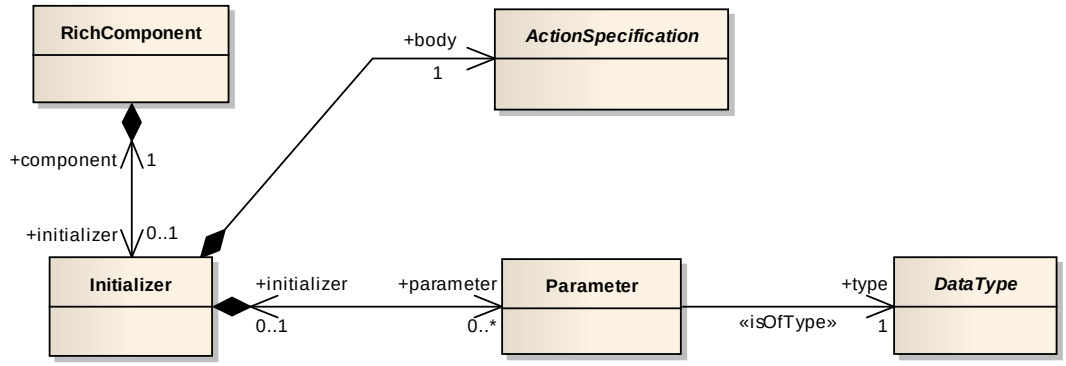

Figure 2.30: Initializer.

Aggregations

- body : ActionSpecification [1] The body of the initializer.
- parameter : Parameter [0..*] {ordered} The ordered set of formal parameters of the initialize.

Assosiations

- component : RichComponent [1] The rich component that owns the initializer.

2.2.3 Service Implementations

In this section the implementation of services is described.

2.2.3.1 ServiceImplementation

A service implementation is the implementation of a function and it is associated to a rich component. Figure 2.31 displays the service implementation meta-class and its relations.

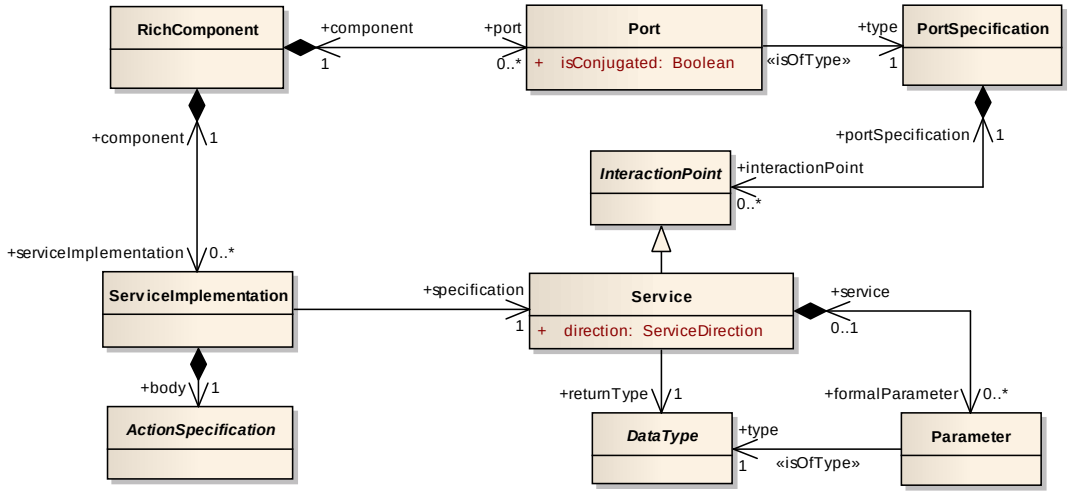

Figure 2.31: Service Implementation.

Aggregations

- body : ActionSpecification [1] Associates to the function implementation the actual code of the service.

Assosiations

- specification : Service [1] The service owing this implementation.
- component : RichComponent [1] The rich component that owns the implementation.

2.2.4 Behavior Definitions

A behavior denotes the dynamics of a rich component. It can refer to an aspect to indicate that the concrete behavior is related to this specific aspect as depicted in Figure 2.32.

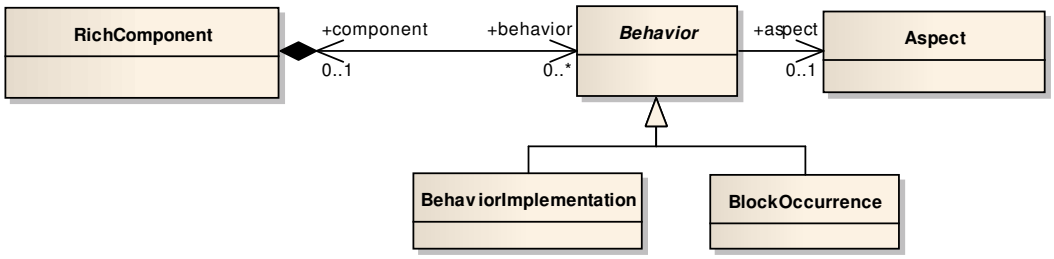

Figure 2.32: Behavior definitions.

2.2.4.1 Behavior $\{abstract\}$

A behavior provides a description of the dynamics of a rich component and can be related to an aspect. Behavior is an abstract meta-class. Specializations are BehaviorImplementation and BlockOccurrence.

Assosiations

- aspect : Aspect [0..1] An aspect which the behavior refers to.
- component : RichComponent [0..1] The component that owns the behavior.

2.2.5 Behavior Implementations

Behavior implementations can be used to describe the implemented behavior of a rich component. A behavior implementation knows about the ports of a rich component with its flows and services. It can be any behaviour implementation like C-Code or a Simulink model. Figure 2.33 gives an overview.

2.2.5.1 BehaviorImplementation

A behavior implementation provides a textual representation of an implemented behavior. The interpretation of this textual representation is given by its category, i. e. C-Code, or external Simulink model. Being a named element a behavior implementation has a name and may own a set of comments.

Generalizations: Behavior, TextuallyRepresentedElement, NamedElement

Assosiations

- category : BehaviorImplementationCategory [1] The category of the behavior implementation.

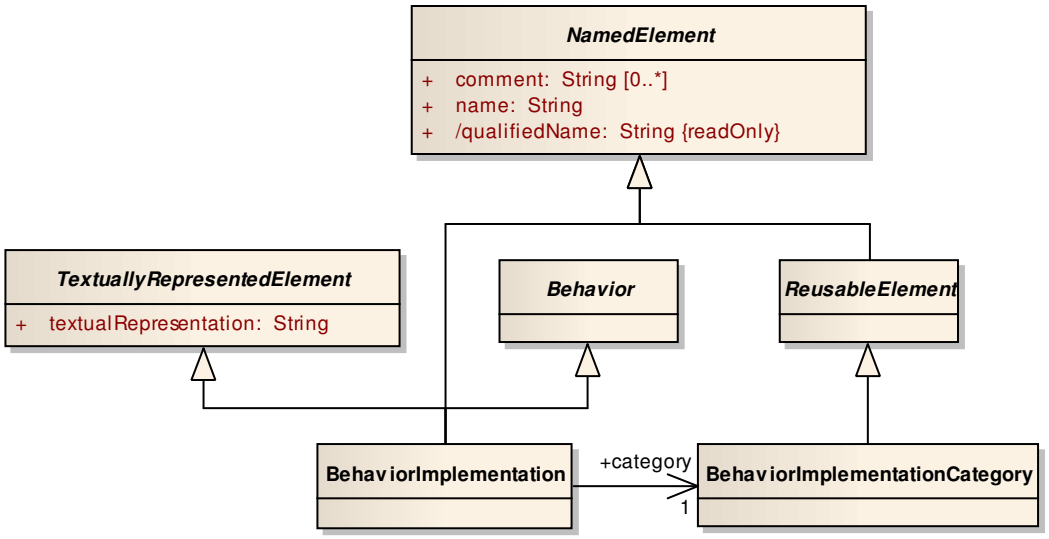

Figure 2.33: Behavior implementation.

2.2.5.2 BehaviorImplementationCategory

A behavior implementation category provides information about the interpretation of the textual representation of a behavior implementation.

Generalizations: ReusableElement

2.2.6 Behavior Blocks

In this section behavioral blocks and their structure are described. Figure 2.34 gives an overview.

2.2.6.1 BehaviorBlock

A behavior block denotes a type of behavior units (by contrast to a rich component which denotes a type of structural units). An instance of a behavior block, that is one particular behavior unit of this type, is denoted by a block occurrence, which can be used in the following circumstances: to specify the formal description of an assertion that is used either as the assumption or promise of an atomic contract of a rich component; to specify the behavior of a rich component; to be used as an operand in an operation on behavior blocks in order to build compound behaviors from basic ones. Such operations on behavior blocks include: conjunction, disjunction, negation, renaming and hiding. Please refer to BlockCompositionOperator for details.

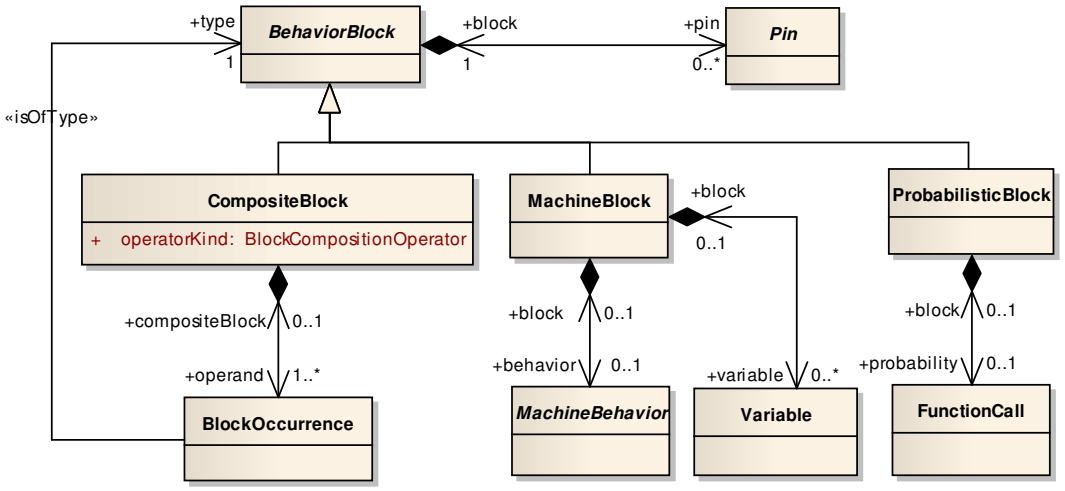

Figure 2.34: Behavior blocks.

A behavior block owns pins that are the interaction points for which the block describes behavior. Each block occurrence of a behavior block owns an instance of each pin owned by the behavior block.

The BehaviorBlock is an abstract meta-class with three concrete sub-classes: MachineBlock, ProbabilisticBlock, and CompositeBlock. Please refer to corresponding sub-classes to understand how behaviors are specified for each kind of behavior blocks.

The fact that a behavior block only talks about dynamics with respect to the pins it owns reflects the encapsulation of the behavior units, hence favors reuse. When a block occurrence is used in the context of a rich component to either specify its contract or behavior, the pin instances of the block occurrence will be linked to the interaction points (i. e. flows and services) or attributes of the rich component, on which the dynamics are actually to be defined.

A behavior block is both a reusable element and a templatable element. A behavior block in the role of template realization may have no pins, normal behavior blocks and behavior block templates must define at least one pin.

Generalizations: TemplatableElement, ReusableElement

Aggregations

- pin : Pin [0..*] The set of pins that are owned by the behavior block.

Constraints Behavior blocks are subject to the following constraints:

1. A behavior block must have at least one pin unless it is in the role of template realization:

```
context BehaviorBlock inv pinsUnlessRealization:
    self.pin→isEmpty() implies self.isRealization()
```

2.2.6.2 MachineBlock

The behavior of a machine block is specified by the runs of a state machine owned by it. The state machine is optional because an instance of MachineBlock in the role of a template realization need not specify its own state machine, normal machine blocks and those in the role of template must specify a state machine. A machine block may also own local variables. Being local, these variables are only visible inside the owning machine block; specifically, they can be referenced only in the state machine owned by the same machine block.

Generalizations: BehaviorBlock

Aggregations

- behavior : StateMachine [0..1] The state machine that is owned by the machine block, which is used to describe the behavior of the machine block.
- variable : Variable [0..*] The (possibly empty) set of variables owned by the machine block.

Constraints Machine blocks are subject to the following constraints:

1. A machine block must have a state machine unless it is in the role of template realization:

```
context MachineBlock inv stateMachineUnlessRealization:
    self.behavior→isEmpty() implies self.isRealization()
```

2.2.6.3 ProbabilisticBlock

A probabilistic block owns a function call and a unique flow pin of kind event (see FlowKind). The behavior specified by a probabilistic block is the set of "runs" on the unique flow pin, which includes any sequences of random values generated according to the probability distribution inherent in the function referred to by the owned function call. An example of such function call could be "uniform(3, 10)", where "uniform" is the uniform distribution function on integers. Values generated at different time instants are independent from each other.

When a probabilistic block appears in the role of template realization, it may specify no function call (since it implicitly specifies the function defined by its template). Normal probabilistic blocks and those in the role of templates must specify a function call via the probability role.

Generalizations: BehaviorBlock

Aggregations

- probability : FunctionCall [0..1] The function call that specifies the probability distribution of the random values that are generated by the probabilistic block on its unique pin.

Constraints Probabilistic blocks are subject to the following constraints:

1. A probabilistic block owns exactly one flow pin:

   context ProbabilisticBlock **inv** ownsOneFlowPin:
   self.pin→size() = 1 **and** self.pin→any(true).oclIsTypeOf(FlowPin)

2. A probabilistic block must specify a function call via its probability role, unless it is a template realization:

   context ProbabilisticBlock **inv** callUnlessRealization:
   self.probability→isEmpty() **implies** self.isRealization()

2.2.6.4 CompositeBlock

A composite block is an application of a block composition operator on some block occurrences typed by other behavior blocks (the operands). No recursion is allowed in the sense that none of the block occurrences that play the role of operands is allowed to be typed by the same composite block.

Five block composition operators are defined: negation, disjunction, conjunction, renaming, and hiding. The behavior of a composite block is the result of composing the corresponding behaviors of the behavior blocks delegated by the operands, following the semantics of the specified block composition operator.

When the operator of a composite block is renaming, which takes exactly one block occurrence as operand and the purpose of this operator is to rename some pin of the operand, the effect of renaming is interpreted by links between pins. More specifically, if a pin instance of the operand, (i. e. the pin is owned by the HRC block that types the operand), is linked to a pin of the composite block with a different pin name, that means this pin is renamed. All such links are owned by the composite block.

Generalizations: BehaviorBlock

Attributes

- operator : BlockCompositionOperator [1] Specifies the operator used in the composite block.

Aggregations

- operand : BlockOccurrence [1..*] The set of block occurrences that play the role of operands in the composite block.

Associations

- link : Link [0..*] The set of links owned by the composite block.

Constraints Type, kind and direction compatibilities of flow pins in composite blocks are defined following the companion semantics paper. When considering the compatibilities of service pins, the constraints come as if we treat a service pin "sp(p1:t1, ..., pk=tk): t" as k + 1 flow pins: sp-p1: t1, ..., sp-pk=tk, sp-result: t. And for directions:

- if direction(sp) = required, then direction(parai) = out, for i = 1, ..., k, and direction(result) = in;
- if direction(sp) = provided, then direction(parai) = in, for i = 1, ..., k, and direction(result) = out.

As the soundness of such an approach still needs to be examined (at least semantically), and overall the design of composite blocks still needs to be checked by users and tools, we restrict ourselves to only informal English statements for the constraints on composite blocks in the following.

Composite blocks are subject to the following constraints:

1. A composite block whose operator is negation is called a negation block. A negation block NegB takes exactly one operand of type SubB. NegB denotes the complement runs of SubB.
   Well-formedness: We require NegB and SubB have the same set of pins. For flow pins: the same types, kinds, and directions. For service pins, same directions and signatures (i. e. same parameters including the types of parameters, and return types).

2. A composite block whose operator is disjunction is called a disjunction block. A disjunction block DisB takes two or more operands respectively of type SubB1 ,..., SubBn, $n >= 2$. DisB denotes the union runs of SubBi, $1 <= i <= n$.
   Well-formedness: We require that for any shared pin (i.e. more than one sub blocks own a pin with that name), it has the same meta-type, that is either all occurrences are as flow pins or as service pins. If flow pins, it has the same type and kind in all the sharing SubBi's, $1 <= i <= n$. And if service pins, it has the same directions and signatures in all the sharing SubBi's. Moreover, the set of pins of DisB is the union of the sets of pins of SubBi, $1 <= i <= n$, with

the same types and kinds for flow pins or same directions and signatures for service pins. As for directions, there is no restrictions on the directions of shared flow pins in sub blocks. But the direction of a flow pin p of DisB should be the least upper bound of the directions of the flow pin p's in all the sharing SubBi, $1 <= i <= n$, where the sub-direction relation is defined as: any direction is a sub-direction of itself; in is a sub-direction of bidirectional; out is a sub-direction of bidirectional.
Shared service pins must have the same directions in all the sub-blocks. Otherwise, consider a shared service pin sp by two sub-blocks SubB1 and SubB2. In SubB1, direction(sp) = required, i. e. direction(parai) = out, for i = 1, . . . , k, and direction(result) = in. In SubB2, direction(sp) = provided, i. e. direction(parai) = in, for I = 1, . . . , k, and direction(result) = out. Following the direction computation rule for pins of the disjunction block, i. e. the least upper bound of the directions of the pins in all the sharing sub-block, we should have direction(parai) = bidirectional, for i = 1, . . . , k, and direction(result) = bidirectional, which do not give a valid direction for a service pin.

3. A composite block whose operator is conjunction is called a conjunction block. A conjunction block ConB takes two or more operands respectively of type SubB1 , . . . , SubBn, $n >= 2$. ConB denotes the product/intersection runs of SubBi, $1 <= i <= n$.
   Well-formedness: the same constraints as in disjunction blocks for flow pins, except that for directions, shared out flow pins are not allowed. Moreover, the direction of a flow pin p of ConB should be the product of the directions of flow pin p in all the sharing SubBi's, $1 <= i <= n$, where the product is defined as:
   - out X in = out
   - out X bidirectional = out
   - in X in = in
   - in X bidirectional = bidirectional
   - bidirectional X bidirectional = bidirectional

   Service pins are forbidden to be shared among sub-blocks, and the set of service pins of ConB is the union of the sets of service pins of SubBi, $1 <= i <= n$.

4. A composite block whose operator is renaming is called a renaming block. A renaming block RnB takes exactly one operand of type SubB. RnB denotes the same runs as SubB, but renames some pins of SubB.
   Well-formedness: We require bijective links between the pins of RnB and the pin instances of the operand. That is, for any pin of RnB, it is linked to one and only one pin instance of the operand, and vice versa. Moreover, if pin p1 of RnB is linked to pin p2 of SubB (via its corresponding instance of the operand), we require they are of the same meta-type, and the same types, kinds, and directions if flow pins, and the same directions and signatures if service pins.

5. A composite block whose operator is hiding is called a hiding block. A hiding block HdB takes exactly one operand of type SubB. HdB denotes the same runs as SubB, but hides some pins of SubB (projection).
   Well-formedness: We require the set of pins of HdB is a subset of the pins of SubB. For flow pins, with the same types, kinds, and directions. For service pins, with the same directions and signatures.

6. A composite block owns links iff its operator is renaming.

7. A composite block may not be neither a template nor a template realization:

```
context CompositeBlock inv notTemplate:
    not self.isTemplate() and not self.isRealization()
        and self.template.isEmpty()
```

2.2.6.5 BlockCompositionOperator {*Enumeration*}

BlockCompositionOperator is an enumeration with elements that can be used as literals for specifying the operators applied in a composite block.

EnumerationLiterals

conjunction takes two or more operands and the resulting behavior is the intersection set of runs specified by the behaviors of the behavior blocks that type the operands.

disjunction takes two or more operands and the resulting behavior is the union set of runs specified by the behaviors of the behavior blocks that type the operands.

negation takes exactly one operand and the resulting behavior is the complementary set of runs specified by the behavior of the behavior block that types the operand.

renaming takes exactly one operand and the resulting behavior is the same set of runs specified by the behavior of the behavior block that types the operand. However, some pins of the behavior block may be renamed.

hiding takes exactly one operand and the resulting behavior is the same set of runs specified by the behavior of the behavior block that types the operand. However, some pins of the behavior block may be hidden (projected away).

2.2.6.6 BlockOccurence

A block occurrence is typed by a behavior block. It denotes an instance of the behavior unit specified by its typing behavior block, including corresponding instances of the pins owned by the behavior block, etc.

A block occurrence is either owned by a composite block to play the role of an operand; or owned by an assertion as its formal presentation; or owned by a rich component to specify the behavior of the rich component.

Associations

- type : BehaviorBlock [1] Description of the role of the associated meta-class.
- compositeBlock : CompositeBlock [0..1] If present, specifies the composite block that owns the block occurrence as its operand.
- component : RichComponent [0..1] If present, specifies the rich component that owns the block occurrence as its behavior.
- assertion : Assertion [0..1] If present, specifies the assertion that owns the block occurrence as its formal presentation.

Constraints Block occurrences are subject to the following constraints:

1. A block occurrence is exclusively either owned by a composite block, or an assertion, or a rich component. This exclusivity is ensured by the semantics of the aggregation relationships between block occurrence and block occurrence, composite block, or assertion.

2.2.6.7 MachineBehavior {*abstract*}

Machine behavior is the behavior specification of a machine block. MachineBehavior is an abstract meta-class. It's specializations allow the exact specification of the machine behavior. A MachineBehavior is represented by an UML state machine in the profile.

Associations

- block : MachineBlock [0..1] The machine block that owns the MachineBehavior.

2.2.7 Pins

In this section behavioral interaction points, which are called pins, are described. Figure 2.35 givs an overview.

2.2.7.1 Pin {*abstract*}

A pin is owned by a behavior block. It denotes an interaction point of the owning behavior block. The functionality of the interaction point is specified by its sub-classes: Flow pin and service pin. A pin is a navigable feature.

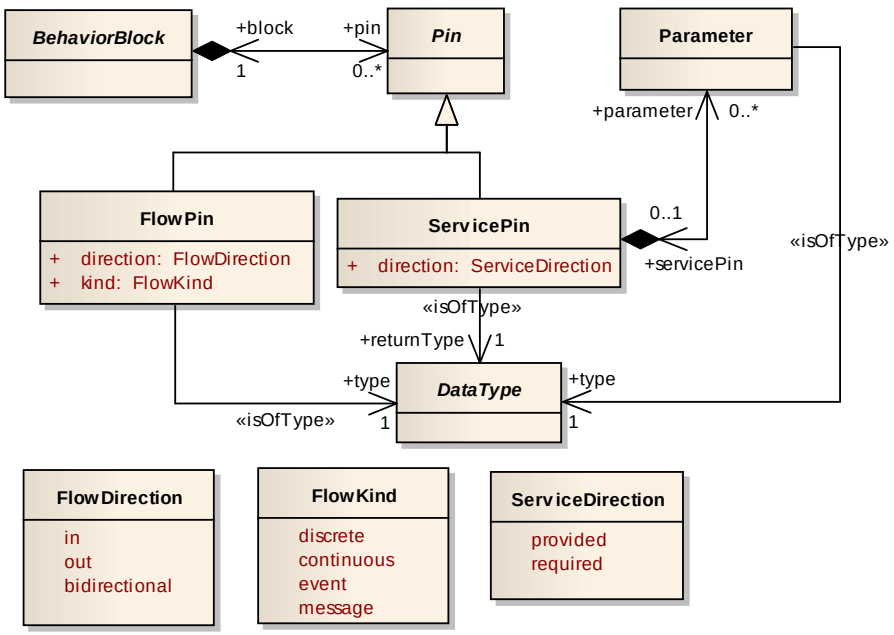

Figure 2.35: Pins.

Generalizations: NavigableFeature

Associations

- block : BehaviorBlock [1] Specifies the behavior block that owns the pin.

2.2.7.2 FlowPin

The functionality of a flow pin is specified by its kind. It either represents a discretely evolving variable, or a continuously evolving variable, or a communication point that carries signals (i. e. having values only at certain time instants) of the owning behavior block. The type association specifies the data type of the values carried by the flow pin, and the direction defines whether the owning behavior block does input/output/both on the flow pin.

Generalizations: Pin

Associations

- Type : DataType [1] Specifies the data type of the values that carried by the flow pin.

2.2.7.3 ServicePin

Service pins are the counterpart of services, where the services are owned by rich components, service pins are owned by BehaviorBlock. As a consequence, a service pin syntactically mimics a service, namely specified by a set of parameters, a return type, and a direction. Similarly, a service pin can be either provided or required by the owning behavior block depending on its direction as specified by ServiceDirection.

A provided service pin means that the owning block receives service calls on this pin in its behavior. Symmetrically, a required service pin allows the owning block to make calls on this pin in its behavior.

Generalizations: Pin

Attributes

- direction : ServiceDirection [1] Specifies if the behavior block calls or provides the service and takes values from the enumeration service direction.

Aggregations

- parameter : Parameter [1..*] These are the parameters that serve as arguments for the service call.

Associations

- returnType : DataType [1] Specifies the data type of the return value of the referred service.

2.2.8 Behavior Links

In this section links between behavioral descriptions and architectural interaction points and attributes are described. Figure 2.36 displays a logical diagram of behavior links.

2.2.8.1 BehaviorLink

A behavior link owns exactly two link ends, which are one of the following: Attribute ends, pin ends, flow link ends, or service link ends.

In general, behavior links are used in three circumstances:

- To associate the dynamics of a pin to a structural feature of a rich component RC. In this case, the behavior link L is owned by RC. The two link ends of L are of the following possibilities:

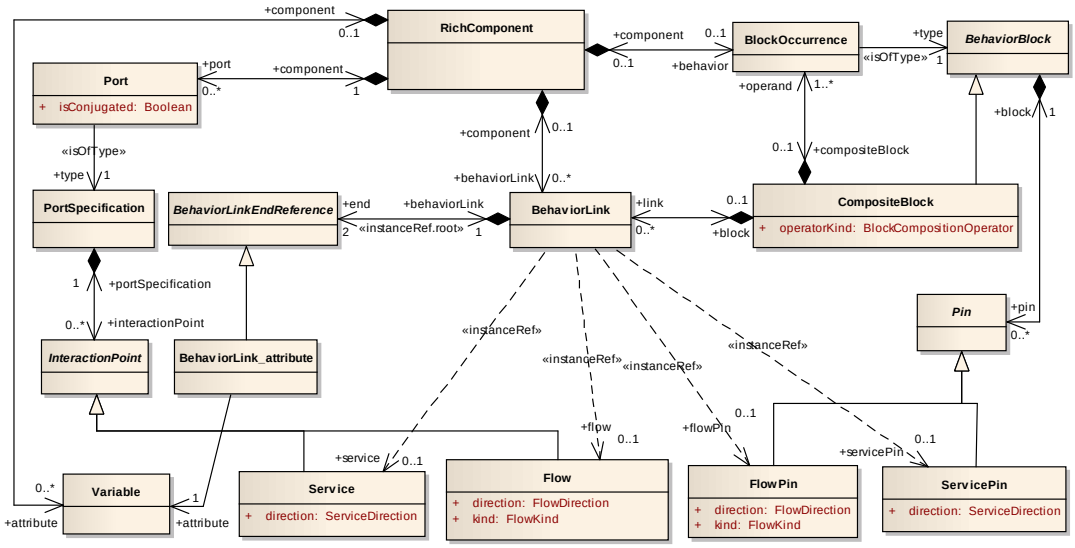

Figure 2.36: BehaviorLinks.

  - One end of L is a flow pin end. It denotes an instance of a flow pin owned by either the block occurrence that specifies the behavior of RC, or the block occurrence that specifies formal presentation of the assertion that is the assumption or promise of one contract of RC. The other end of L is a flow link end. It denotes an instance of a flow owned by a port of RC or one of its sub-components.
  - Similar case for services: one end of L is a service pin end, and the other end of L is a service link end.
  - One end of L is a service pin end, and the other end of L is an attribute end denoting an attribute of RC.

- To express the interaction between the two atomic contracts C1 and C2 of a rich component RC. In this case, the link L is owned by RC, and both ends are flow pin ends. One flow pin end denotes an instance of flow pin owned by the assumption or promise block occurrence of C1, and the other denotes an instance of flow pin owned by the assumption or promise block occurrence of C2.

- Inside a composite block that applies the renaming operator. See Subsection 2.2.4 for details.

Aggregations

- end : BehaviorLinkEndReference [2] Specifies the two ends of the behavior link.

Associations

- component : RichComponent [0..1] If present, specify the rich component that owns the behavior link.
- block : CompositeBlock [0..1] If present, specify the composite block that owns the behavior link.

2.2.8.2 BehaviorLinkEndReference {*abstract*}

A behavior link end reference is owned by a behavior link. BehaviorLinkEnd-Reference is an abstract meta-class with five sub-classes: BehaviorLink_attribute, BehaviorLink_flowPin, BehaviorLink_servicePin, BehaviorLink_flow and Behavior-Link_service.

Associations

- behaviorLink : BehaviorLink [1] Specify the behavior link that owns this behavior link end reference.

2.2.8.3 BehaviorLink_attribute

A BehaviorLink_attribute is a link end which references a variable (see Figure 2.37).

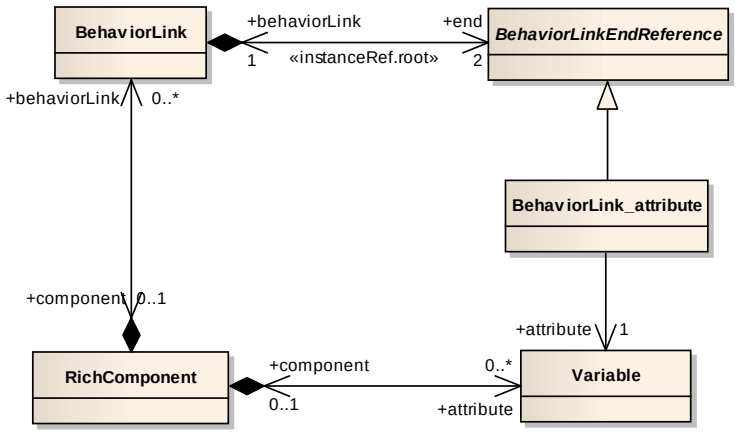

Figure 2.37: BehaviorLink_attributes.

Generalizations: BehaviorLinkEndReference

Associations

- attribute : Variable [1] Specify the attribute that is referenced by the attribute end.

2.2.8.4 BehaviorLink_flowPin

A BehaviorLink_flowPin refers to the instance of a flow pin. Figure 2.38 gives more detail.

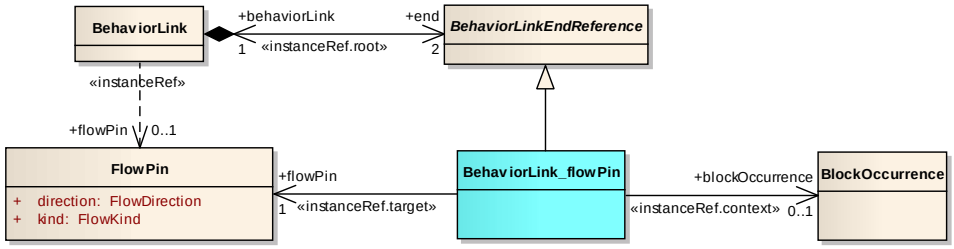

Figure 2.38: BehaviorLink_flowPins.

Generalizations: BehaviorLinkEndReference

Associations

- blockOccurrence : BlockOccurrence [0..1] Specify the block occurrence whose instance of pin is referenced. This association is optional in the case of renaming blocks, where for one end, the block occurrence is implicitly the renaming block itself.
- flowPin : FlowPin [1] Specify the flow pin whose instance is referenced by the flow pin end.

2.2.8.5 BehaviorLink_servicePin

A BehaviorLink_servicePin refers to the instance of a service pin (see Figure 2.39).

Generalizations: BehaviorLinkEndReference

Associations

- blockOccurrence : BlockOccurrence [0..1] Specify the block occurrence whose instance of pin is referenced. This association is optional in the case of renaming blocks, where for one end, the block occurrence is implicitly the renaming block itself.

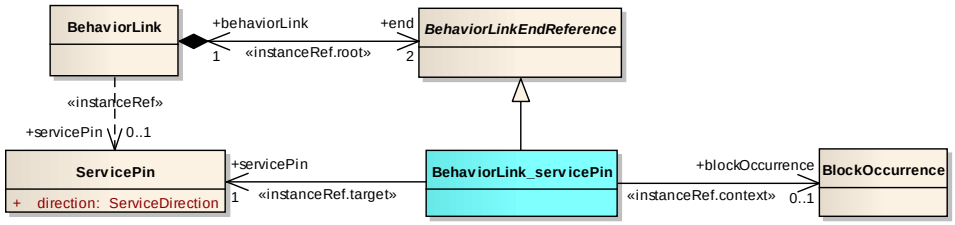

Figure 2.39: BehaviorLink_servicePins.

- servicePin : ServicePin [1] Specify the service pin whose instance is referenced by the service pin end.

2.2.8.6 BehaviorLink_flow

BehaviorLink_flow references the instance of a flow which is given by referencing the containing port and its owning rich component instance. If the size of the associated port is bigger than 1, then the optional "portIndex: Expression [0..1]" association must be present to specify exactly one instance in the port. BehaviorLink_flow plays the role of an end of a link that is owned by the same component that also owns the flow. Figure 2.40 displays the BehaviorLink_flow in more detail.

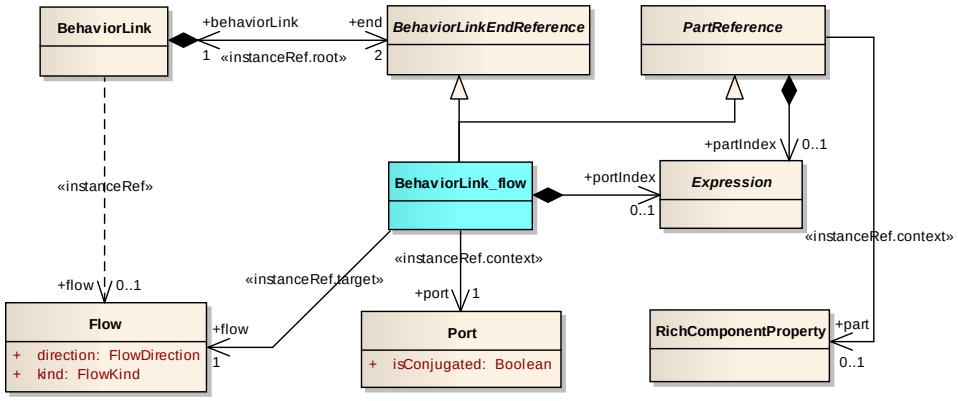

Figure 2.40: BehaviorLink_flows.

Generalizations: BehaviorLinkEndReference, PartReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

Associations

- port : Port [1] Specifies the port referenced.
- flow : Flow [1] Specifies the flow referenced.

2.2.8.7 BehaviorLink_service

BehaviorLink_service references the instance of a service which is given by referencing the containing port and its owning rich component instance. If the size of the associated port is bigger than 1, then the optional "portIndex : Expression [0..1]" association must be present to specify exactly one instance in the port. BehaviorLink_service plays the role of and end of a link that is owned by the same component that also owns the service. Figure 2.41 displays the BehaviorLink_service in more detail.

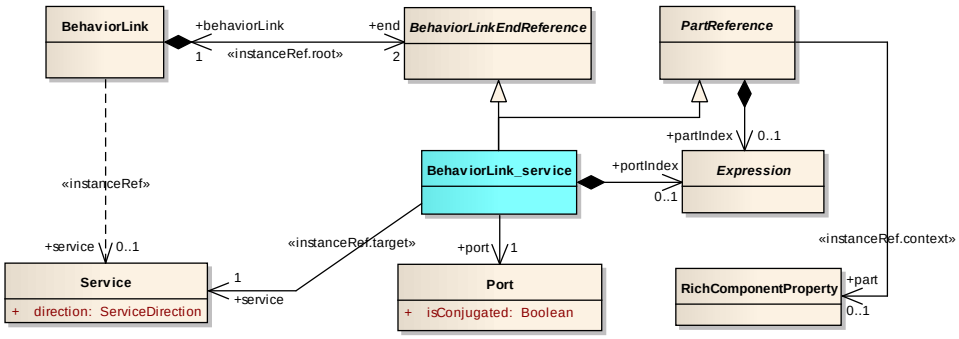

Figure 2.41: BehaviorLink_services.

Generalizations: BehaviorLinkEndReference, PartReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

Associations

- port : Port [1] Specifies the port referenced.
- service : Service [1] Specifies the service referenced.

2.2.9 Component Mapping

This section deals with relations between abstraction levels and perspectives. The general idea is to create some sort of mapping that observes a systems behavior on two different abstraction levels and validates if the observable behavior is causally equal. Another possible application of these mapping links is between perspectives. Note that this is not possible for all perspectives since some perspectives do not formally (as in typed data) specify observable behavior (e. g. the operational perspective does not always talk about formal interface specifications). Figure 2.42 shows the meta-model cutout of the mapping artifacts.

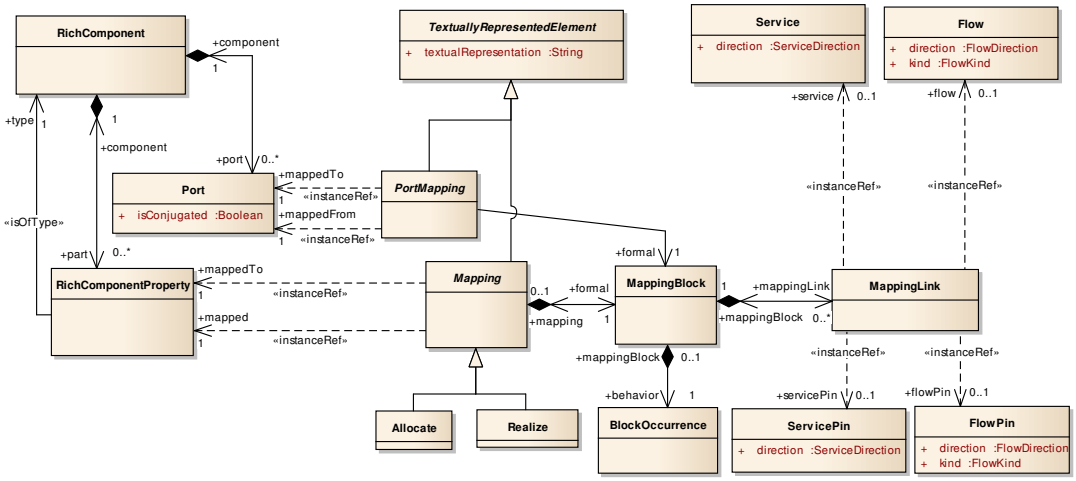

Figure 2.42: Component mapping relations.

Component mapping allows to describe solutions of how component parts in different kinds of rich component models can be are mapped to each other. Component mappings can be allocations and realizations. Since mappings become visible on data correlations a mapping is only complete if there is a formal specification of how data interaction points of one rich component model can be projected to corresponding data interaction points of another rich component model.

Figure 2.43 illustrates how rich component models can be mapped to each other. The way of description is the same for allocation and for realization. But the kinds of mapped models are different. In case of a realization there are different models on different abstraction levels mapped to each other. Such a mapping is useful when describing how rich components are realized after a refinement in a new model i. e. logical components being realized by concrete hard- and software components. In case of an allocation there are models of different perspectives mapped i. e. logical perspective to technical perspective. A mapping like an allocation or a refinement links to component parts of two models to identify the context for the mapping. The mapping is formally described by a mapping block, with links to the concrete data interaction

points and a block occurrence to which represents the mapping behavior. This behavior can be expressed by a statemachine which formally describes how the data of the mapped interaction points behave to each other respective how the projection looks like [BRR+10].

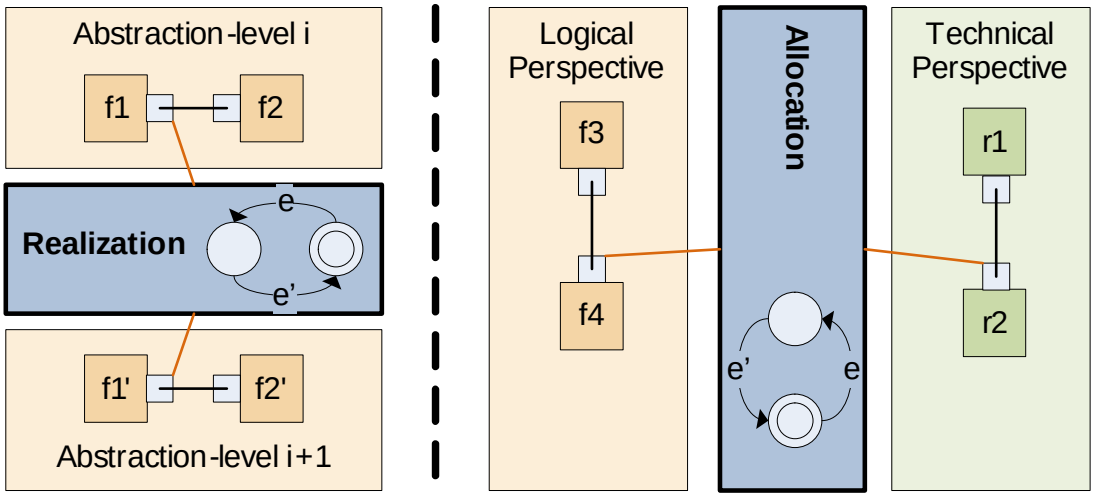

Figure 2.43: Mapping relations: Allocation and Realization.

A mapping describes the mapping solution between two rich component properties which belong to different component models such as allocations between component models of different perspectives and realizations between component models of different abstraction levels. The referenced rich component properties provide the context of the mapping. These context references are needed for the identification of data items which belong to the description of mapping. Since a mapping becomes visible with the evaluation of corresponding interaction points a mapping is only complete if there is a formal description of data mapping. This data mapping is formally provided by a mapping block and can furthermore be given by a textual representation.

2.2.9.1 Mapping {*abstract*}

The abstract mapping meta-class represents an element which may connect to two component instances of different abstraction levels or perspectives. An associated formal specification can specify how to interpret the behavior of both ports. Note that the references to Flow and Service exclude each other i. e. either there are references to two flows, two services, or neither. Mapping has two concrete subclasses denoting the relationships between perspectives (Allocate) or abstraction levels (Realize).

Generalizations: ReusableElement, TextuallyRepresentedElement

Aggregations

- formal : MappingBlock [1] The mapping block that formally defines the data mapping of the referenced rich component properties.
- mapped : Mapping_part [1] The rich component instance which provides the mapped context.
- mappedTo : Mapping_part [1] The rich component property which is the context for the mapping target.

2.2.9.2 Allocate

The allocate meta-class is a concrete mapping meant to be used only between two different perspectives on the same abstraction level.

Generalizations: Mapping

2.2.9.3 Realize

The realize meta-class is a concrete mapping meant to be used only between two different abstraction levels.

Generalizations: Mapping

2.2.9.4 Mapping_part

Mapping_part is an end reference of a mapping and references exactly one component part. Mapping_part is owned by a Mapping to reference both a rich component property as a mapped context and a rich component property as a mapping target. Figure 2.44 gives an overview over the mapping part element.

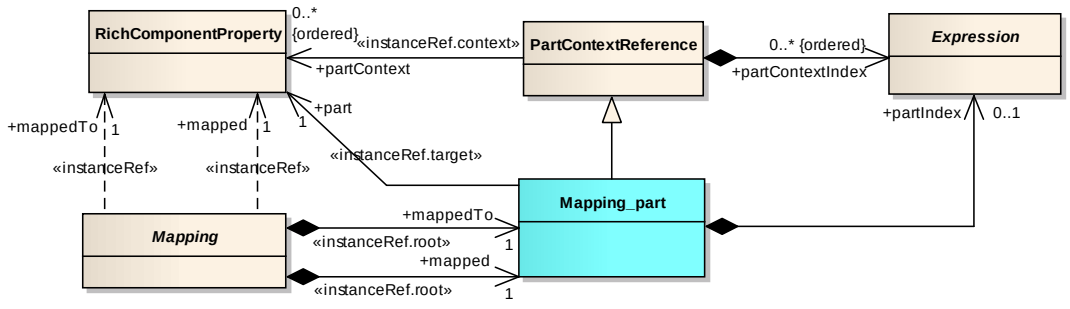

Figure 2.44: Mapping_part.

Since often a concrete instance in a very specific context shall be referenced the path of owning component paths must be given. Furthermore, rich component properties

are MultiplicityElements and can therefore be instantiated several times by assigning a size to the RichComponentProperty. Such a multiple instantiation is denoted by only one RichComponentProperty. When defining a mapping to a multiple component part, the respective index of the part and also the indices of the context parts have to be given.

Generalizations: PartContextReference

Aggregations

- partIndex : Expression [0..1] In case the size of the rich component property associated is not 1, specifies the index in the multiple element.

Associations

- part : RichComponentProperty [1] The referenced component part.

2.2.9.5 PartContextReference {*abstract*}

PartContextReference is an abstract meta-class whose subclasses have the ability to reference the concrete context of a rich component part. The part context is given by an ordered set of referenced RichComponentProperties. These referenced RichComponentProperties are a path a component parts and provide the distinct context information. The referenced part with the lower most index (0) is the part which contains the property that shall be addressed by the concrete subclass. The referenced part with the upper most index is the part that can be distinctively referenced by the part context reference. If there are parts which are multiply instantiated, for all elements of the part context path a part context index must be given. The order is the same as for the part context path. Non-multiple instances have the index 0.

Aggregations

- partContextIndex : Expression [0..*] {ordered} An ordered set of indices denoting the exact part of multiple part instances in the part context path.

Associations

- partContext : RichComponentProperty [0..*] {ordered} An ordered set of rich component properties denoting a concrete part context path.

2.2.9.6 MappingBlock

A mapping bock formally defines the mapping between structural interaction points of rich component parts. Thus a mapping block contains a block occurrence which defines how the interaction points behave to each other and therefore how data is projected between two rich component models. Furthermore, a mapping block owns a set of mapping links in order to connect the mapped structural interaction points to the pins that belong to the behaviour block which types the behaviour block occurrence of the mapping block.

A mapping block is either owned by a mapping such as an allocation or a realize link or, as a reusable element, declared by a declaration zone independent from an allocation or a realization.

Generalizations: ReusableElement

Aggregations

- mappingLink : MappingLink [0..*] The mapping links that are owned by the mapping block.
- behavior : BlockOccurrence [1] The behavior description of how mapped interaction points relate to each other.

Associations

- mapping : Mapping [0..1] Description of the role of the associated meta-class.

2.2.9.7 MappingLink

Mapping links allow to link between mapped structural interaction points like flows as well as services of rich component parts and pins of the block occurrence which belong to a mapping block. In contract to behavior links mapping links do not have a component scope and must therefore reference the exact part that has the interaction points. Figure 2.45 gives a more detailed view on mapping links.

A maping link connects data interaction points such as flows and services with respective flow and service pins that belong to the mapping behaviour description of the mapping block. In contrast to behaviour links a mapping links do not belong to the context of a rich component but to the context of a mapping block and therefore reference interaction points of parts that belong to a specific part context.

Aggregations

- mapped : MappingLinkInteractionPointReference [1] A reference to the mapped interaction point.

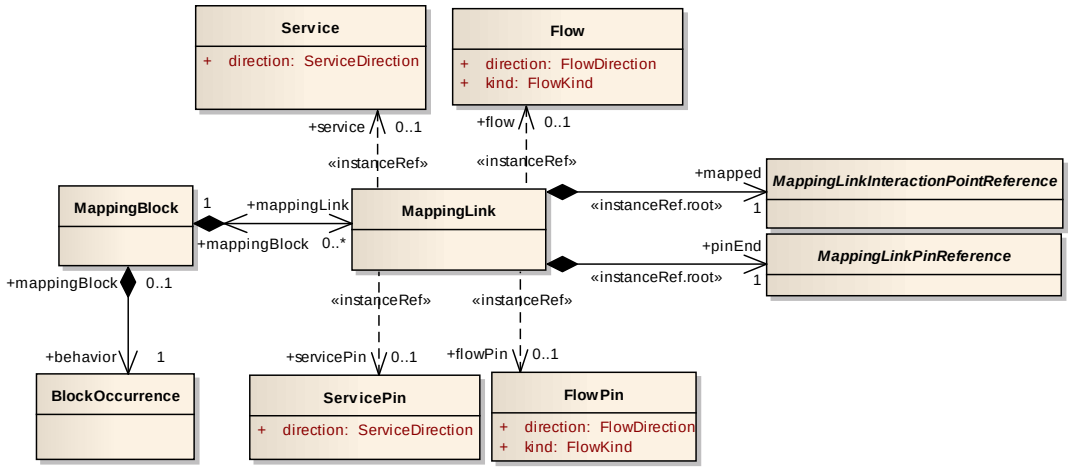

Figure 2.45: MappingLink.

- pinEnd : MappingLinkPinReference [1] A reference to the pin which belongs the behavior of the mapping block and shall be linked with the mapped interaction point.

Associations

- mappingBlock : MappingBlock [1] The mapping block that owns the link.

2.2.9.8 MappingLinkInteractionPointReference {*abstract*}

MappingLinkInteractionPointReference is an abstract meta-class. Its specializations reference the context of an interaction point which is given by a part, a part context and a port. If a port is instantiated as a multiplicity element a port index must be given to reference a specific port. A mapping link interaction point reference is owned as one end of a mapping link to reference an interaction point such as a flow or a service. Concrete subclasses are MappingLink_flow and MappingLink_service.

Generalizations: PartReference, PartContextReference

Aggregations

- portIndex : Expression [0..1] The index of the exact port if a port is a multiple instance.

Associations

- port : Port [1] The port which is context of the referenced interaction point.

- mappingLink : MappingLink [1] The mapping link which owns the reference.

2.2.9.9 MappingLink_flow

MappingLink_flow is an end of a mapping link which references a specific flow of a port that belongs to a concrete structural rich component part as depicted in Figure 2.46.

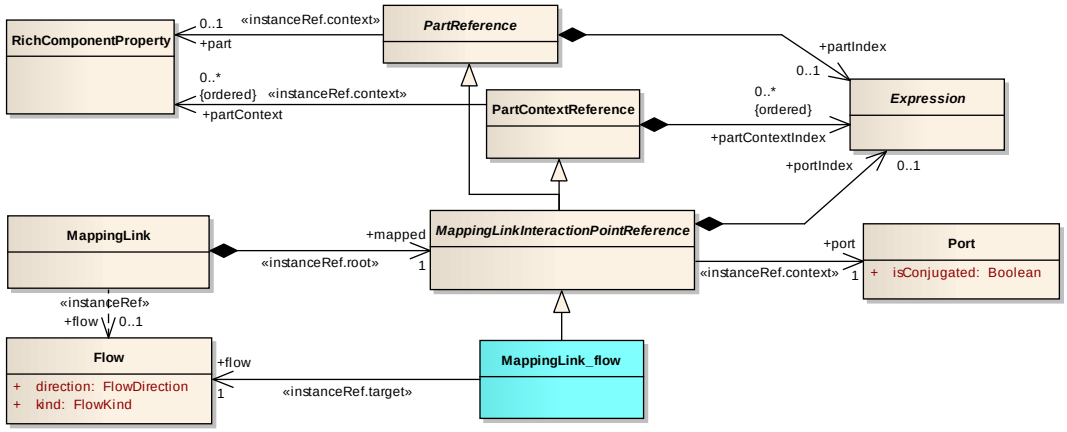

Figure 2.46: MappingLink_flow.

Generalizations: MappingLinkInteractionPointReference

Associations

- flow : Flow [1] The referenced flow.

2.2.9.10 MappingLink_service

MappingLink_service is an end of a mapping link which references a specific service of a port that belongs to a concrete structural rich component part as depicted in Figure 2.47.

Generalizations: MappingLinkInteractionPointReference

Associations

- service : Service [1] The referenced service.

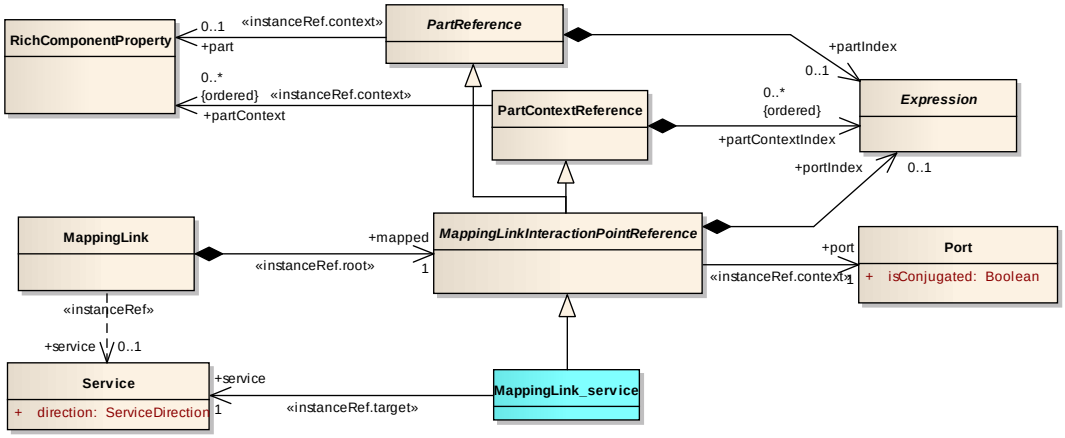

Figure 2.47: MappingLink_service.

2.2.9.11 MappingLinkPinReference {*abstract*}

MappingLinkPinReference is an abstract meta-class. Its specializations reference the behavior block occurrence of a mapping block as the context of a pin. A mapping link pin reference is owned as one end of a mapping link to reference a pin such as a flow or a service pin. Concrete subclasses are MappingLink_flowPin and MappingLink_servicePin.

Associations

- mappingLink : MappingLink [1] The mapping link that owns the reference.
- blockOccurrence : BlockOccurrence [1] The block occurrence which is the context of the referenced pin.

2.2.9.12 MappingLink_flowPin

MappingLink_flowPin is an end of a mapping link which references a specific flow pin of the behaviour block occurrence of the mapping block which owns the mapping link as depicted in Figure 2.48.

Generalizations: MappingLinkPinReference

Associations

- flowPin : FlowPin [1] The target flow pin of the mapping link.

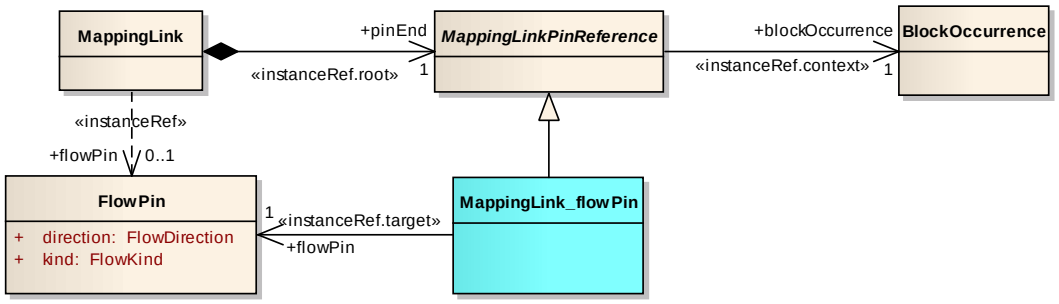

Figure 2.48: MappingLink_flowPin.

2.2.9.13 MappingLink_servicePin

MappingLink_servicePin is an end of a mapping link which references a specific service pin of the behaviour block occurrence of the mapping block which owns the mapping link as depicted in Figure 2.49.

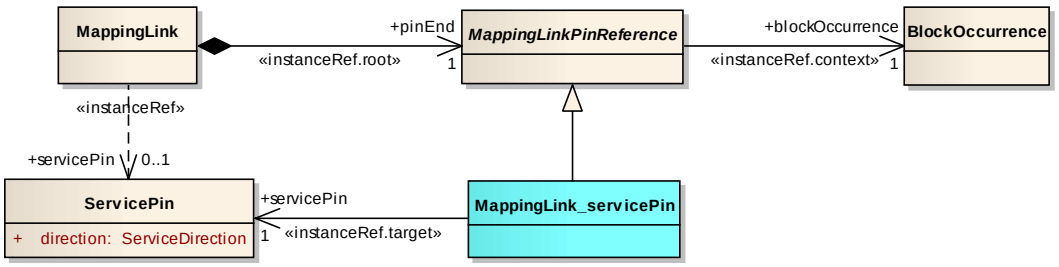

Figure 2.49: MappingLink_servicePin.

Generalizations: MappingLinkPinReference

Associations

- servicePin : ServicePin [1] The target service pin of the mapping link.

2.2.9.14 PortMapping {*abstract*}

In order to specify mappings directly between ports and thus reduce the complexity of the formal representation of the mapping there is an abstract meta-class PortMapping which is a super-meta-class of PortAllocate and PortRealize. Figure 2.50 displays an overview over port mappings.

Like the Mapping meta-class, PortMapping refers to a formal specification (MappingBlock) and is a subclass of TextuallyRepresentedElement and ReusableElement.

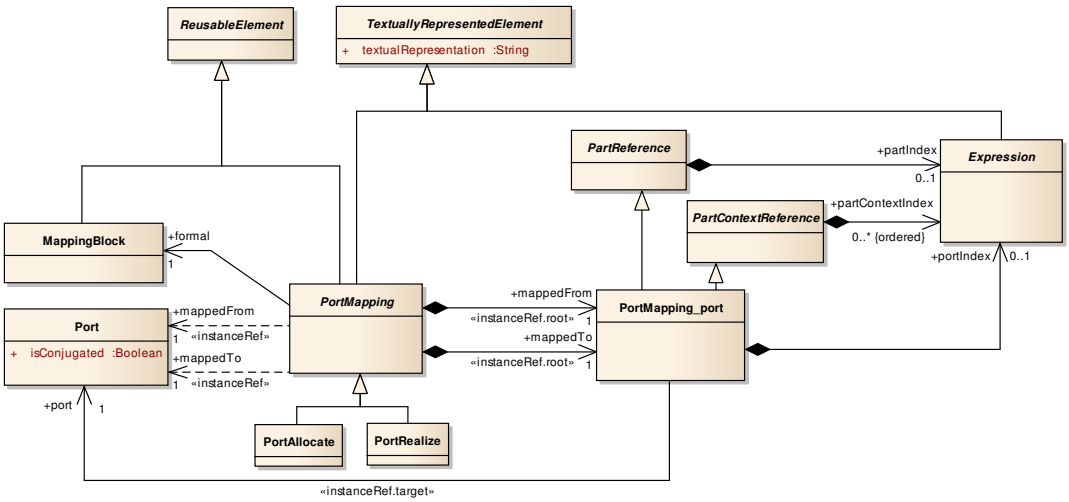

Figure 2.50: Port Mappings.

Generalizations: ReusableElement, TextualleRepresentedElement

Aggregations

- formal : MappingBlock [1] The mapping block that formally defines the data mapping of the referenced rich component properties.
- mappedFrom : PortMapping_port [1] The port of a rich component instance which provides the mapped context.
- mappedTo : PortMapping_port [1] The port of a rich component property which is the context for the mapping target.

2.2.9.15 PortAllocate

PortAllocate is a concrete mapping solution between ports of rich component properties in rich component models of different perspectives in one abstraction level.

Generalizations: PortMapping

2.2.9.16 PortRealize

PortRealize is a concrete mapping solution between ports of rich component properties in rich component models in different abstraction levels.

Generalizations: PortMapping

2.2.9.17 PortMapping_Port

The mapping port reference PortMapping_port is an end reference of a port mapping and references exactly one port of one component part. PortMapping_port is owned by a PortMapping to refer to a rich component property as a mapped context, a rich component property as well as port as a port mapping target.

Since often a concrete instance in a very specific context shall be referenced the path of owning component paths must be given. Furthermore, rich component properties are MultiplicityElements and can therefore be instantiated several times by assigning a size to the RichComponentProperty. Such a multiple instantiation is denoted by only one RichComponentProperty. When wanting to define a mapping to a multiple component part, the respective index of the part and also the indices of the context parts have to be given. The same applies for the referenced ports which also have to have and index.

Generalizations: PartContextReference

Aggregations

- portIndex : Expression [0..1] In case the size of the port associated is not 1, specifies the index in the multiple element.

Associations

- port : Port [1] The referenced port.

2.3 Requirements Meta-Model

This section deals with the SPESMM Requirements Meta-Model as part of the SPES Meta-Model. The Requirements Meta-Model, which will be described in the following, provides a general requirements concept with respec to the requirements engineering definitions from SPES ZP-AP2, Contracts as a means for formal system requirements as defined in HRC (SPEEDS) and furthermore a traceability concept.

2.3.1 Requirements

This section describes means to cover system specifications which are requirements and goals. System specifications are system artifacts in a system design. They refer to aspects such as safety, realtime or functional and can be related to a set of stakeholders. There is a disinction between goals and requirements. Requirements are grouped into process requirements and system requirements. Process requirements are textual descriptions of how the design process shall look like. System requirements describe what the system under design shall fulfill. They are structured as contracts. Being

a contract a system requirement consists of a set of assertions which are a guarantee and optional strong and weak assumptions. A strong assumption characterizes the allowed context of a component and a weak assumption considers integration context possibilities. The guarantee states the guaranteed behavior for the given assumptions. Assertions describe the assumed context and the guaranteed component behavior. Assertions have a textual description which can be informal or formal. An informal assertion carries informal text. It can be based on a template which provides information about its interpretation. A formal assertion carries a description which can be formally interpreted. A goal denotes a design goal for a component. Its content is described by an assertion. Figure 2.51 gives an overview.

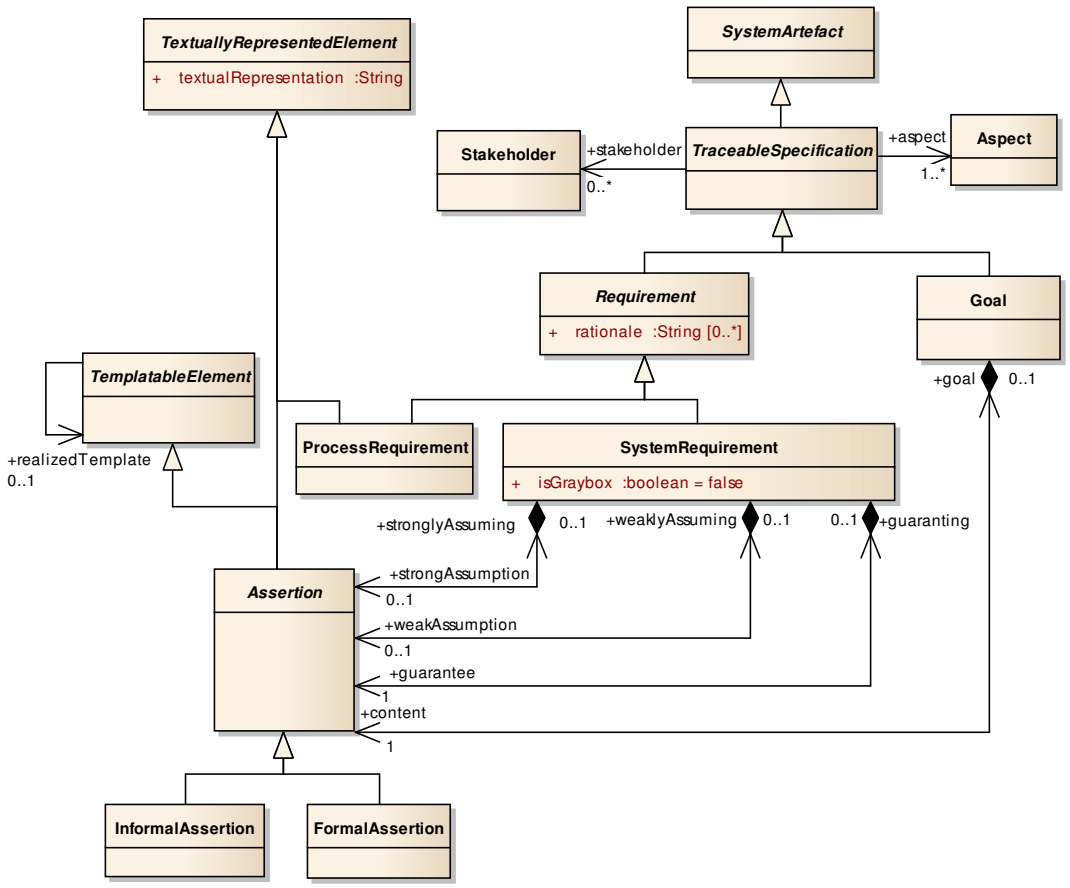

Figure 2.51: Requirements.

Different kinds of requirements such as natural language requirements, boilerplates or pattern based requirements can be covered using these structural requirement concepts. For each system requirement the meaning of its content may be the same but with different degree of formality. A natural language based user requirement simply contains informal assertions. A boilerplate requirement is also covered by informal

assertion but these are realizations of assertion templates which describe the structure and the relevant parameters of the boilerplate. Since a pattern provides a complete formal requirement description, the content of pattern based requirements is covered by formal assertions.

2.3.1.1 TraceableSpecification {*abstract*}

A traceable specification is a specification that is traceable during the development process. Trace links such as Refine, Derive and Evaluate are defined to link traceable specification. A traceable specification may reference a set of stakeholders which are involved in the context of the respective requirement and. It may also reference a set of aspects which it addresses.

TraceableSpecification is an abstract meta-class. Its sub-classes define concrete specifications. A TraceableSpecification is a SystemArtefact and therefore a ReusableElement. It can be traced by using respective trace links. There are two sub-classes, namely Goal and Requirement.

Generalizations: SystemArtifact

Associations

- aspect : Aspect [1..*] Specifies the aspects which are addressed by the specification.
- stakeholder : Stakeholder [0..*] Specifies the stakeholder who is responsable for the traceable specification.

2.3.1.2 Requirement {*abstract*}

A requirement is defined as a statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality assurance guidelines.

A requirement may reference aspects, which it is related to, and also a set of stakeholders which are involved in the context of the respective requirement. The general concept of a requirement provides a textual representation for the description of the requirement. Furthermore, requirements are reusable and can be traced by using respective trace links.

Requirement is an abstract class which. Concrete sub-classes provide explicit semantics.

Generalizations: TraceableSpecification

Attributes

- rationale : String [0..*] The rationale(s) for the requirement (i. e. why the requirement is present).

2.3.1.3 Stakeholder

A stakeholder represents a person which is involved in the development of a system architecture. A stakeholder is an individual that has right, share, claim, or interest in a system or in its ossession of characteristics that meet her/his needs and expectations.

Generalizations: ReusableElement

2.3.1.4 Goal

A goal denotes a design goal during a development process. Typically stakeholders of a system development process define goals for the system design. Such a goal describes an optimization target for the development of a system and its design entities. It declares intended functional and qualitative characteristica of the final system design. These characteristics are described by criteria for measurable properties which shall be optimized. Typically aspects such as cost or weight are addressed. Based on goals further system requirements can be derived which constrain the system design and which lead to specific design decisions. Goals allow considerations about the usage of existing library components regarding the component's properties, which are addressed by the goals.

Typically a system or a design entity that is subject to a goal cannot be directly verified on reaching the goal or not. Therefore, a goal analysis requires a measure. The measure is used for the evaluation of a system architecture. It allows arguing whether and in which degree a goal is reached by the properties of a design solution. Typically goals can be reached in different ways and therefore allow different architecture solutions. Therefore goal measures can be used to choose a design solution among design alternatives which reaches the goals in the best way.

A Goal is a specification which is traceable. The content of a goal is defined by an assertion. Goals can be refined into more formal goals. System requirements can be derived from goals. A Rich Component can be subject to goal which is denoted by a satisfy link.

Generalizations: TraceableSpecification

Aggregations

- content : Assertion [1] Specifies the assertion that is owned by the system requirement and which defines the content of the goal.

2.3.1.5 ProcessRequirement

A process requirement is a requirement related to the engineering (development/maintenance) process and imposed by corporate policy or practice. Process requirements include compliance with national, state or local laws; administrative requirements such as physical security; and specific work directives.

By now a process requirement can be textually described.

Generalizations: Requirement

2.3.1.6 SystemRequirement

System requirements encapsulate the internals of a component and abstract dynamics constraints by providing an abstract characterization of the component. Traceability regarding system requirements will be described in Section 2.3.2.

The system requirement specification is given in terms of a triple of a weak assumption, a strong assumption, and a promise, each of which is specified by an assertion owned by the system requirement. It states that the environment of a satisfying rich component must behave as assumed. Moreover, under such circumstance, the rich component itself behaves as promised.

Generalizations: Requirement

Aggregations

- strongAssumption : Assertion [1] Specifies the assertion that is owned by the system requirement as strong assumption.
- weakAssumption : Assertion [1] Specifies the assertion that is owned by the system requirement as weak assumption.
- promise : Assertion [1] Specifies the assertion that is owned by the system requirement as promise.

2.3.1.7 Assertion {*abstract*}

An assertion specifies a logical property. It is on one hand informally given by statements in English (specified by the textualRepresentation attribute inherited from TextuallyRepresentedElement). An assertion can be formal or informal.

Assertion is an abstract meta-class. Its specializations have concrete semantics. Sub-classes are InformalAssertion and FormalAssertion.

Generalizations: TextuallyRepresentedElement, TemplateableElement

Associations

- stronglyAssuming : SystemRequirement [0..1] If present, specifies the system requirement that owns the assertion as its strong assumption.
- weaklyAssuming : SystemRequirement [0..1] If present, specifies the system requirement that owns the assertion as its weak assumption.
- guaranting : SystemRequirement [0..1] If present, specifies the system requirement that owns the assertion as its promise.

2.3.1.8 InformalAssertion

An informal assertion provides a statement in an informal language (specified by the textualRepresentation attribute inherited from TextuallyRepresentedElement) for an assumption or for the promise of a system requirement.

As a templatable element an informal assertion can be defined as a template. Such a template may define a set of parameters which declare variable properties of the informal assertion. Being a reusable element such an informal assertion template can be stored in any declaration zone, i. e. a package. Realizations of such a template may be owned by system requirements. A realization references the respective template assertion and defines in parameter substitutions the values for the variable properties — the parameters that are declared in the template.

Generalizations: Assertion, ReusableElement

2.3.1.9 FormalAssertion

A formal assertion provides a formal definition of an assumption or a promise of a system requirement. This can be formally given by a block occurrence covering the textual representation.

Generalizations: Assertion

Aggregations

- formal : BlockOccurrence [1] Specifies the formal presentation of the assertion.

2.3.2 Requirements Traceability

The link types to be introduced cover relevant traceability concepts in existing meta-models like SysML [Obj08a], EAST-ADL2 [ATE08], or in the traceability reference model by Jarke and Ramesh [RJ01]. Regarding the contained traceability elements,

there are only few differences between these meta-models. In fact many approaches are related to each other. The MeMVaTEx requirements meta-model uses traceability links from SysML and the abstraction layers from EAST-ADL2. The EAST-ADL2 traceability links are very similar to SysML. It is hoped that more requirements towards traceability will origin from ZP-AP2 in the future. Figure 2.52 shows an overview.

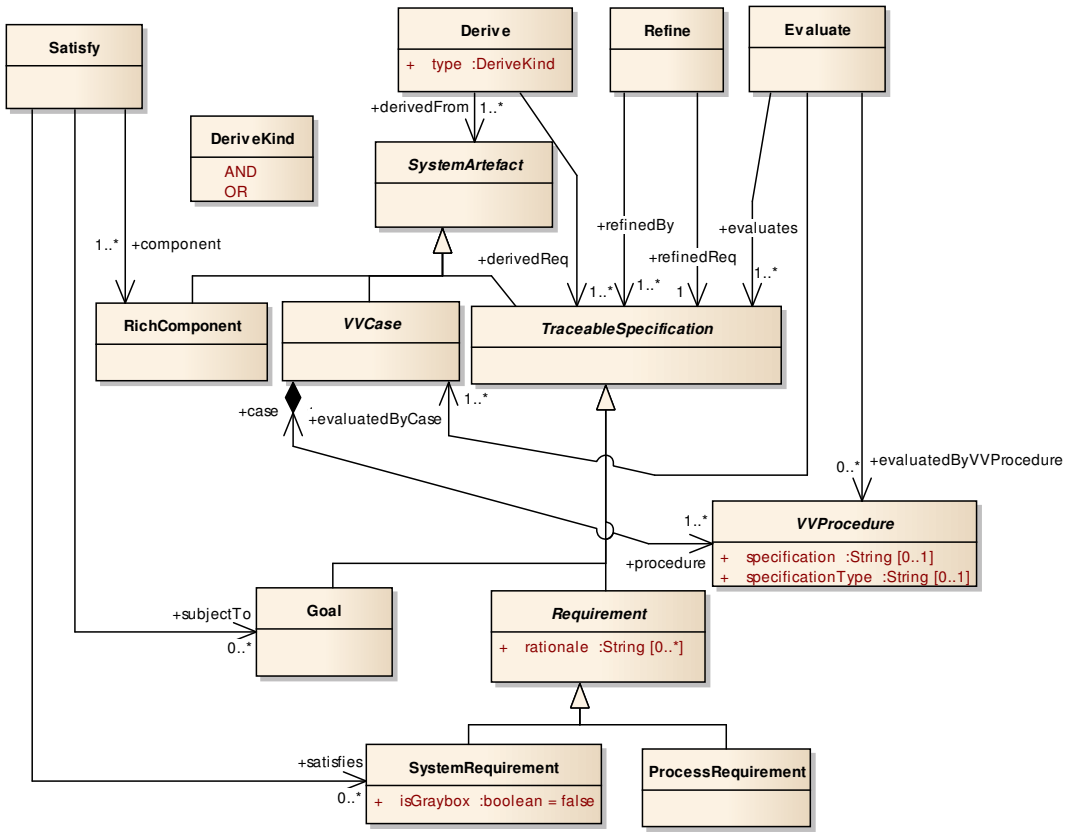

Figure 2.52: Traceability.

2.3.2.1 Refine

A refinement operation adds detail to an abstraction while preserving the semantics of the abstraction. The result of applying a refinement operation to an abstract item (say requirement AR1) is a new item (say RR1), such that RR1 has more detail than AR1, but RR1 has the same intent and meaning as what AR1 specifies. This link is e. g. used to trace requirement formalisations.

The refinement link represents a specific dependency between requirements on different level of abstractions where one or multiple target requirements add further detail to the source requirement. The existence of the resulting requirements is completely

defined by the superior source requirement. If the latter is changed, the resulting requirements of the refinement need to be changed, too.

Generalizations: ReusableElement

Associations

- refinedReq : TraceableSpecification [1] The source specification that is refined.
- refinedBy : TraceableSpecification [1..*] The resulting specification set that refines the source requirements.

2.3.2.2 Derive

This link is used to indicate that one or more new requirements are derived from a design decision or certain analysis results (e. g. realtime analysis).

Requirements are derived from existing requirements, certain design decisions or analysis results, where the modification of the requirement or the change of a made design decision could have an impact on the derived requirements.

Generalizations: ReusableElement

Associations

- derivedReq : TraceableSpecification [1..*] The set of specifications that are derived from the source spcifications.
- derivedFrom : SystemArtefact [1..*] The set of source system artifacts (Requirement is a special of SystemArtefact).

2.3.2.3 Satisfy

This link between components (component of different abstraction layers, source code) and system requirements indicates that the artifacts shall satisfy the requirements it is linked to.

This link indicates that the components connected to the requirement(s) shall satisfy the requirement(s). The justification of this link is done by verification means. A change to the requirement(s) could have an impact on the components and therefore lead to a complete re-verification whether the component still satisfies the requirement(s).

Generalizations: ReusableElement

Associations

- satisfies : SystemRequirement [1..*] The set of system requirements that shall be satisfied by the set of components.
- subjectTo : Goal [0..*] The set of goals which shall be reached by the component definition.
- component : RichComponent [1..*] The set of components that shall satisfy the defined set of system requirements.

Constraints Satisfy links are subject to the following constraints:

1. The satisfied Contract must be a SystemRequirement.

2.3.2.4 Evaluate

This link between V&V Cases and requirements indicates that the V&V Case verifies/validates the requirement. As VVCase is an abstract concept, it can represent both validation and verification procedures.

This link indicates that the VVCases connected to the requirement(s) shall verify/-validate the requirement(s). The question whether the VVCases really verifies/validates the requirement(s) is a different topic. Nonetheless, a change to the requirement(s) could have an impact on the results of the verification/validation and therefore lead to a complete re-verification/validation.

Generalizations: ReusableElement

Associations

- evaluates : TraceableSpecification [1..*] The set of specifications that is verified or validated.
- evaluatedByCase: VVCase [1..*] The set of VVCases that performs the validation/verification.
- evaluatedByProcedure : VVProcedure [0..*] The set of VVProcedures that performs the validation/verification.

2.4 Safety Aspects of the Meta–Model

This section deals with the safety aspects of the meta-model, first the verification and validation parts second with the failure modeling parts.

2.4.1 Verification and Validation

This section describes the part of the SPES Meta-Model that deals with the representation of verification and validation artifacts and results. The V&V package is adopted with some modifications from EAST-ADL2.1 and generally follows the same working hypothesis as EAST-ADL2: Many different verification and validation (V&V) techniques, methods and tools are applied during the development of electrical/electronic systems. Different techniques are applicable at different abstraction levels. Also, choosing a technique depends on the properties being validated and/or verified and which language was used to specify them. Furthermore, each partner in a project may develop and employ his own V&V processes and activities, what makes it impossible to define a common set of verification and validation types in the SPES Meta-Model core.

The current set of V&V concepts shown in Figure 2.53 describe the common parts of V&V techniques that allow the organization of V&V and the storage of V&V results. The main modifications compared to EAST-ADL are:

- Description of V&V categories to define what kind of V&V activity produced the VVCase. This could be e. g. a safety analysis, a weight assessment or any kind of requirement quality assurance technique.
- Concrete distinction between validation and verification, where validation represents all kind of requirement related quality assurance (unambiguity check, consistency check, etc.).

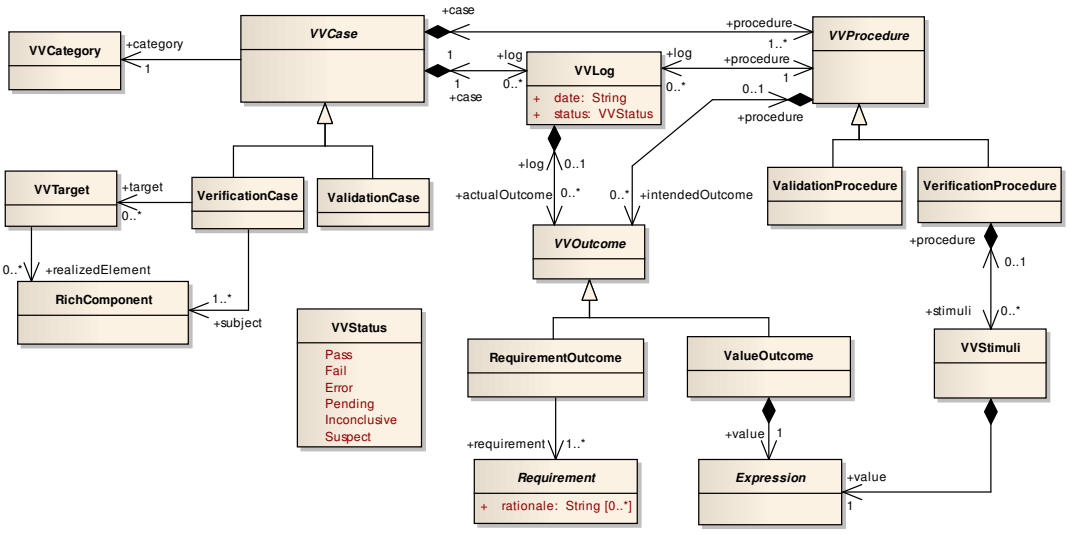

Figure 2.53: Verification and Validation.

To illustrate the main concepts and their usage, the following two usage scenarios are described:

Representation of Requirement Quality Assurance (e. g. manual review, automatic checks, etc.) In the first usage example (see Figure 2.54), the description and storage of requirement validation results with the V&V meta-model is shown. Requirement validation focusses either on characteristics of single requirements like unambiguity, atomic, verifiable, necessary, (internal) consistency, etc. or on characteristics of requirement sets like (external) consistency or completeness. The following figure shows the representation of such checks using the two single requirement checks unambiguity and verifiable and a consistency check on a set of requirements.

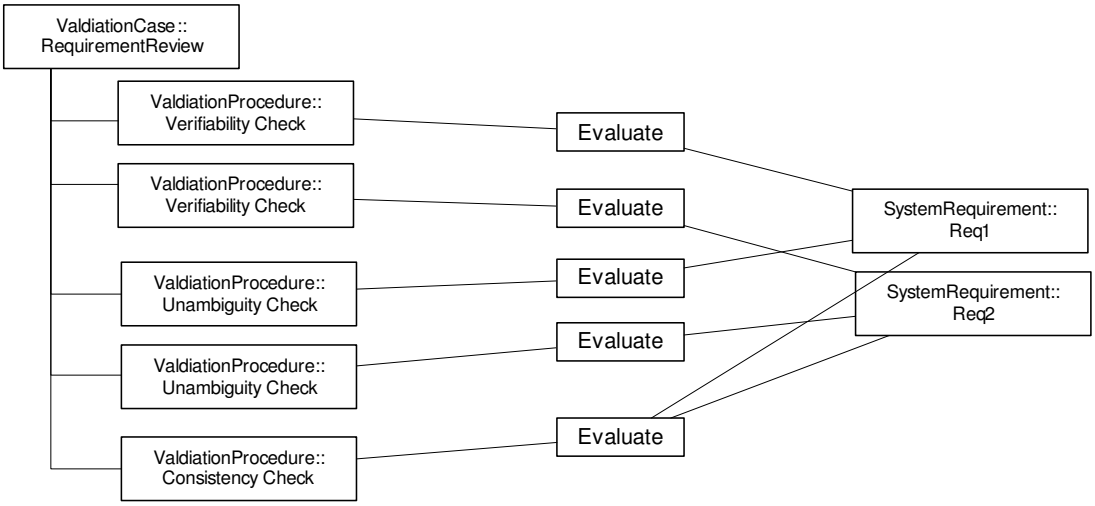

Figure 2.54: Requirement Quality Analysis represented with VV concepts.

Each time the checks are performed, the result is stored using the VVLog concept (not shown in the figure). The check itself can be manually or automatically. The concrete V&V method in use does not matter.

Architecture Assessment The next usage example describes the representation of analyses for architecture assessments. On the top, Figure 2.55 shows the description of non functional architecture assessments like cost or weight. For each cost or weight requirement, there is a defined verification procedure. Each of these verification procedure uses a registered metric/model to assess the as-is value of the requirement and stores the result in using VVLog and VVActualOutcome. The metric itself is marked as dashed lines as this is not directly part of the core V&V meta-model and has to be provided by an analysis service/meta-model extension. Below these non functional assessments, Figure 2.55 also indicates the evaluation of architecture mapping constraints. They are also described using VerificationProcedure and results are stored using VVLog and VVActualOutcome (not shown in the figure).

The following section introduces all elements of the V&V package and the last section gives some usage example on how to represent different kind of V&V activities.

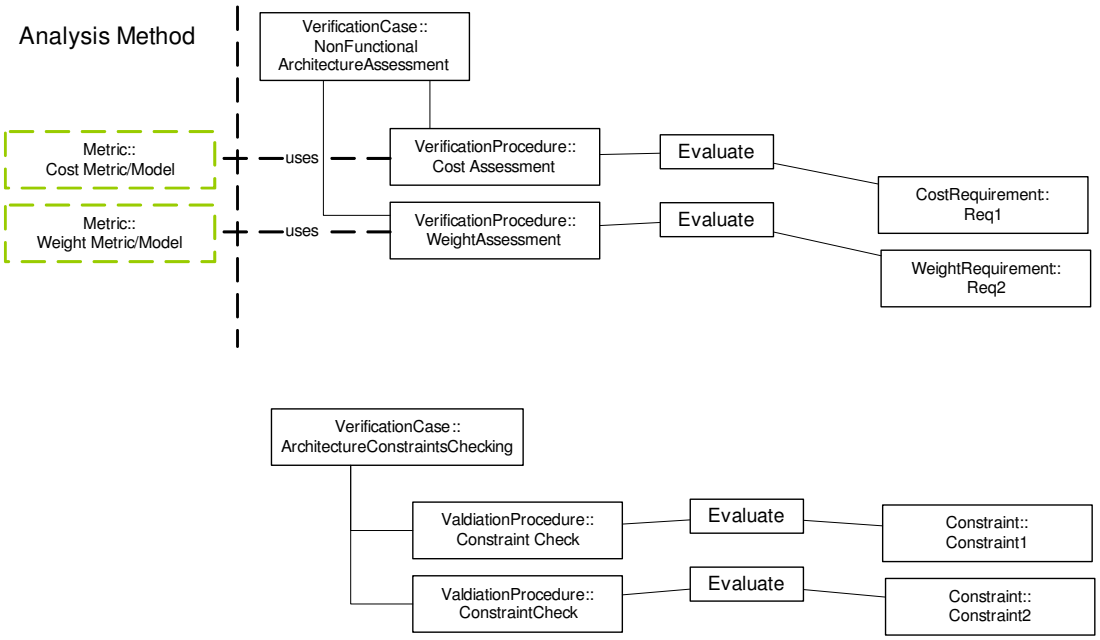

Figure 2.55: Architecture Assessments represented with VV concepts.

2.4.1.1 VVCase

A VVCase is an abstract representation of an overall V&V effort. It is typed by a VVCategory and stores the results of each "execution" of a corresponding VVProcedure using the VVLog concept. VVCase is a SystemArtefact which may contain a textal description and can be stored in a package.

Generalizations: SystemArtefact

Aggregations

- vvProcedure : VVProcedure [1..*] The VVProcedures for this VVCase.
- vvLog : VVLog [0..*] The VVLogs captured while executing this VVCase.

Associations

- category : VVCategory [0..1] The category describes the type of the VVCase (e. g. HIL Testing, Safety Analysis, Requirement Consistency Check, etc.)

2.4.1.2 VVStatus {*Enumeration*}

An enumeration depticting all possible status a VVProcedure can have. This status is attached to a VVLog to give information about the status at a given date.

EnumerationLiterals

Pass Means that the V&V procedure was executed successful with the specified intended outcome.

Fail Means that the V&V procedure was executed wrong i. e. the specified intended outcome was not met.

Error An error occurred during execution of the V&V procedure, i. e. the specification/implementation of the procedure was wrong.

Pending The V&V procedure is specified, but it was not yet executed.

Inconclusive The V&V procedure did not deliver a result, it can not be stated whether it passed or failed.

Suspect Due to a change, the V&V procedure itself and/or its result is possibly not valid any more and needs to be checked/rerun.

2.4.1.3 ValidationCase

A ValidationCase represents a validation effort to analyze whether we build the right product, i. e. whether the requirements are valid (consistent, complete, correct, etc.) as well as confirming that the product, as provided (or as it will be provided), will fulfil its intended use. Validation (of a product) ensures that the correct product is built. It may be performed in the operational environment or in a simulated operational environment.

Generalizations: VVCase

Constraints Validation cases are subject to the following constraints:

1. A ValidationCase can only own ValidationProcedures.

2.4.1.4 VerificationCase

VerificationCase represents a verification effort for design, implementation and product verification. A VerificationCase specifies concrete test subjects and targets and provides stimuli and the expected outcome on a concrete technical level. It aims at confirming that work products properly reflect the requirements specified for them. Verification (of a product) ensures that a product is correctly built.

Generalizations: VVCase

Associations

- target : VVTarget [0..*] The VVTargets for this VerificationCase. See association "subject" for more information.
- subject : RichComponent [1..*] The elements that are being verified and validated by this VerificationCase. Usually this will be a subset of those elements which are realized by the VVTarget(s) of the VerificationCase; but this need not always be the case. The difference between the vvSubjects and the entities which are realized by the case's VVTarget(s), is that the vvSubjects are related to the primary, overall objective of the VerificationCase, while the realized entities can comprise more than these. For example:
  - **(a)** For technical reasons additional entities need to be realized only to permit the testing of the entities of actual interest or
  - **(b)** If a VVTarget is reused among many VerificationCase and therefore realizes more entities than are actually being tested by any single VerificationCase.

Constraints Verification cases are subject to the following constraints:

1. A VerificationCase can only own VerificationProcedures.

2.4.1.5 VVProcedure

VVProcedure represents an individual task in a verification effort (represented by a VerificationCase), which has to be performed in order to achieve that effort's overall objective. A VVProcedure is defined with a concrete testing environment in mind and provides stimuli and the expected outcome of the procedure in a form which is directly applicable to this testing environment.

Generalizations: NamedElement

Aggregations

- vvStimuli : VVStimuli [0..*] Set of involved stimuli.
- intendedOutcome : VVOutcome [0..*] Set of intended outcomes.

Associations

- case : VVCase [1] The owning VVCase.

2.4.1.6 VVTarget

VVTarget represents a concrete testing environment in which a particular V&V activity can be performed. This can be physical hardware or it can be pure software in case of a test by way of design level simulations. Usually, a VVTarget will be a realization of one or more elements. However, there are cases in which this is not true, for example when a VVTarget represents parts of the system's environment. Therefore the association to element has a minimum cardinality of 0. VVTargets can be reused across several ConcreteVVCases.

Generalizations: ReusableElement

Associations

- realizedElement : RichComponent [0..*] The rich component this VVTarget realizes.

2.4.1.7 VVStimuli

VVStimuli represents the input values of the testing environment represented by VVTarget in order to perform the corresponding VVProcedure. The input values must be provided in a form that is directly applicable to the VVTarget(s) defined for the containing VerificationCase.

Aggregations

- value : Expression [1] The value of the VVStimuli.

2.4.1.8 VVLog

VVCase represents the description of a V&V effort and thus provides all necessary information to actually perform this V&V effort. However, it does not represent the actual execution of the effort. This is the purpose of the VVLog. Each VVLog meta-class represents an execution of a VVCase. For example, if the HIL test of the wiper system with a certain set of stimuli was performed on Friday afternoon and, for checkup, again on Monday, then there will be one VVCase describing the HIL test and two VVLogs describing the test results from Friday and Monday respectively.

Attributes

- date : String [1] Date and time when this log was captured.
- status : VVStatus [1] The status of the VVCase that owns this log.

Aggregations

- actualOutcome : VVOutcome [0..*] The actual outcome of the VVCase.

Associations

- case : VVCase [0..1] The VVCase owning this log.
- procedure : VVProcedure [1] The associated procedure.

2.4.1.9 VVOutcome {*abstract*}

VVOutcome represents a test output. This output can be intended for a specific VVProcedure or be an actual result belonging to a VVLog. Being an intended outcome of a VVProcedure a VVOutcome denotes an intended result of an inididual test task. In case of an actual test result the VVProcedure that is referenced by the VVLog denotes the task that is executed and which leads to the actual outcome. An actual test outcome should fulfill the intended outcome being referenced by a VVProcedure. As a test output in a verification step a VVOutcome can be produced by a testing environment as represented by VVTarget when triggered by the VVStimuli of the VerficationProcedure that is referenced by the VVLog.

VVOutcome is an abstract meta-class. Concrete subclasses are RequirementOutcome, CutSet, and ValueOutcome.

Associations

- log : VVLog [0..1] Denotes the VVLog to which the VVOutcome belongs being a actual test outcome.
- procedure : VVProcedure [0..1] Denotes the VVProcedure which has the VVOutcome as an intended test outcome.

Constraints VVOutcomes are subject to the following constraints:

1. A VVOutcome can either belong to a VVLog as an actual outcome or to a VVProcedure as an intended outcome.

2.4.1.10 RequirementOutcome

RequirementOutcome is a VVOutcome referring to a set of requirements.

Generalizations: VVOutcome

Associations

- requirement : Requirement [1..*] The set of requirements to which the actual outcome referes.

2.4.1.11 ValueOutcome

ValueOutcome is a VVOutcome with a specific value.

Generalizations: VVOutcome

Associations

- value : Expression [1] The value of the actual outcome.

2.4.1.12 VVCategory

A VV Category describes a verification or validation kind. This can be requirement validation analyses like requirement consistency, completeness or any other requirement quality attribute as well as analysis like HIL testing, requirement based testing or safety analyses. VVCategories depend on the concrete V&V process in a company and therefore need to be provided as a V&V category library.

Generalizations: ReusableElement

2.4.2 Safety Extension

The safety concept of the SPESMM provides means for the analysis of potential component failures. Figure 2.56 gives an overview.

2.4.2.1 SafetyCase

A safety case denotes a safety analysis for a specific component as its subject.

Generalizations: VVCase

Associations

- subject : RichComponent [1] The rich component that is subject to the safety case.

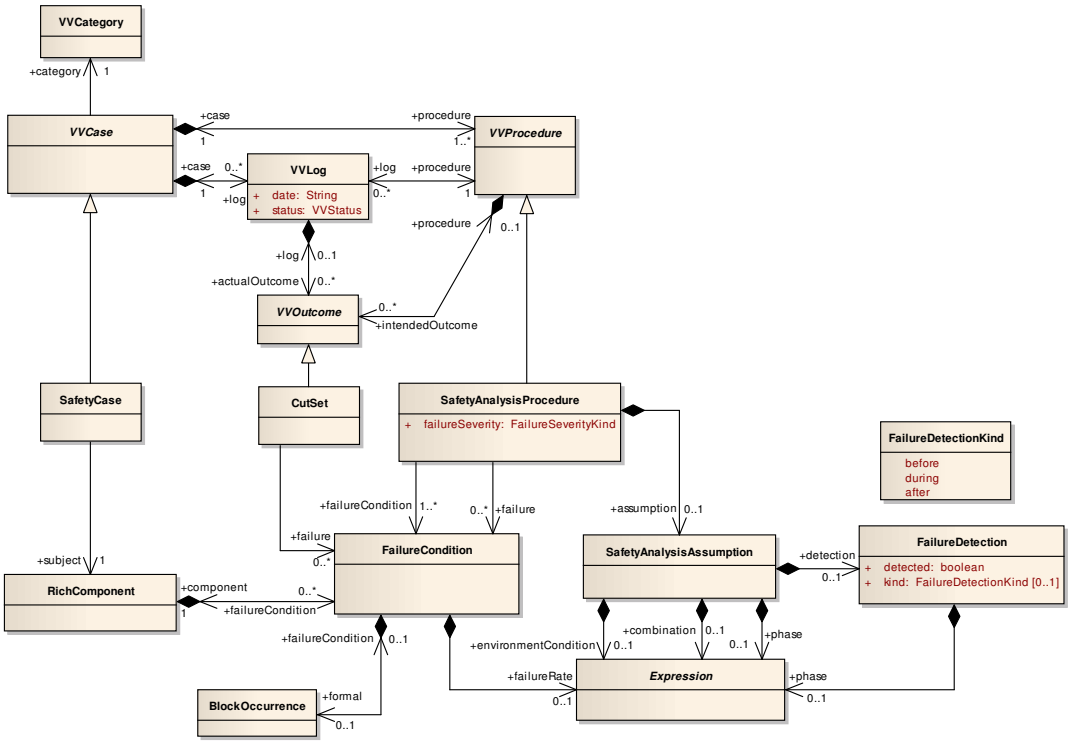

Figure 2.56: Safety concepts.

2.4.2.2 SafetyAnalysisProcedure

A safety analysis procedure is an individual safety task on the occurrence of specific failures. In an analysis these failures are top level events. Other failure conditions can be referenced as base failures which can potentially lead to these top level events. A safety analysis procedure can have a safety analysis assumption which qualifies the assumed occurrence of the failures. Furthermore, a safety analysis procedure can have intended outcomes which can be quantitative value or a qualitative value as well as an intended cut set. These intended outcomes are described as follows:

quantitative A real value expressing a quantitative safety requirement on the quantification of the failure occurrence. The value denotes an upper bound for the propability of the first occurrence of the particular failure within a specific time span. (i. e. "$10^{-}9$ *per hour*")

qualitative A natural number expressing a qualitative safety requirement on the size of the allowed minimal cut set which corresponds to the level of severity. A value of 1 means that no single failure shall lead to the top level event (TLO), a value of 2 means that no double failure shall lead to the TLO and so on.

cut set An intended cut set.

Generalizations: VVProcedure

Attributes

- failureSeverity : FailureSeverityKind [1] Safety impact classification of a failure.

Aggregations

- assumption : SafetyAnalysisAssumption [0..1] The assumption on the occurrence of a failure for an individual safety analysis procedure.

Associations

- failureCondition : FailureCondition [1..*] A set of failure conditions which describe top level events.
- failure : FailureCondition [0..*] A set of basis failures which can potentially lead to the top level events.

2.4.2.3 SafetyAnalysisAssumption

A safety assumption a qualifying assumption about the circumstances of failures in a particular safety analysis procedure, such as the operational phase, a classification of the detection detection, environment conditions for the particular failure condition or combinations with other failure conditions.

Aggregations

- detection : FailureDetection [0..1] A temporal classification for the detection of the failure (i. e. '$before\ speed = 80\frac{km}{h} reached$").

Associations

- environmentCondition : Expression [0..1] A value domain classification on external properties that might affect severity, detection etc. of the failure (i. e. "$acceleration > -3\frac{m}{s^2}$").
- combination : Expression [0..1] Denotes combinations with other failures that must also occure to make the particular failure critical (i. e. "*HandbrakeController_loss or FullThrottle_commission*")
- phase : Expression [0..1] The operational phase in which the failure occurs (i. e. "Landing").

2.4.2.4 CutSet

A cut set denotes a set of failures as a result of a SafetyCase.

Generalizations: VVOutcome

Associations

- failure : FailureCondition [0..*] The failures of the cut set.

2.4.2.5 FailureDetection

A failure detection provides information about the detection of a failure.

Attributes

- detected : Boolean [1] Denotes whether a failure is detected or not.
- kind : FailureDetectionKind [0..1] Denotes a temporal classification for the detection of a failure in combination with a phase description.

Aggregations

- phase : Expression [0..1] An operational phase description related to the failure detection kind denoting a temporal classification for the detection of a failure.

2.4.2.6 FailureDetectionKind {*Enumeration*}

A failure detection kind provides a temporal classification for the occurrence of a failure in combination with an operational phase description.

EnumerationLiterals

before Denotes the detection of a failure before a specific phase.

during Denotes the detection of a failure during a specific phase.

after Denotes the detection of a failure after a specific phase.

2.4.2.7 FailureSeverityKind {*Enumeration*}

The severity kind classifies the impact of a failure. Concrete values to be used in a development process are defined by domain requirements such as ARP. FailureSeverityKind provides a generic classification and therefore defines five severity kinds. Four of these kinds are compliant to the classification of ARP, EASA CS-25 (Certification Specification for Large Aeroplanes) and EASA AMC25.1309 (System design and analysis). Furthermore, a failure can have no safety impact.

EnumerationLiterals

none No safety impact.

minor Minor safety impact which is required to be probable.

major Major safety impact which is required to be remote.

hazardous Hazardous safety impact which is required to be extremely remote.

catastrophic Catastrophic safety impact which is required to be extremely improbable.

2.5 Data Type Specification

A data type describes values which data may have. The SPES Meta-Model provides a means to denote primitive types and enumerations as well as complex data types like records or arrays. Furthermore, a primitive type can be specified more precisely and it is possible to denote units and dimensions. Figure 2.57 gives an overview.

2.5.1 Data Types

This section describes the data type concepts which are provided by the SPES Meta-Model.

2.5.1.1 Enumeration

An enumeration is a data type whose instances are enumerated in the model as enumeration elments (also called literals).

Generalizations: DataType

Aggregations

- element : EnumerationElement [1..*] The set of elements that characterizes the enumeration.

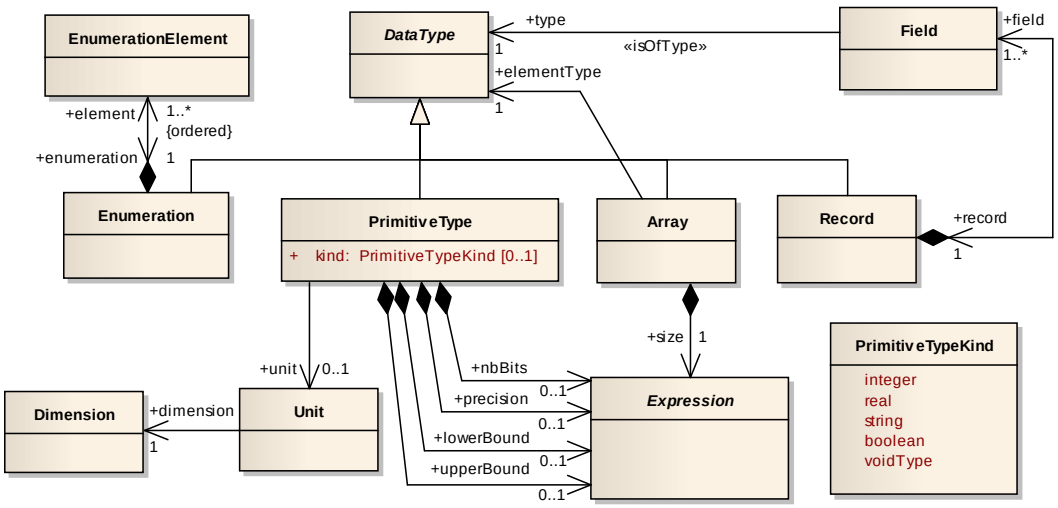

Figure 2.57: Data types.

2.5.1.2 EnumerationElement

An enumeration element denotes a user-defined value for an enumeration. The value is characterized by its name.

Generalizations: NamedElement

Associations

- enumeration : Enumeration [1] The enumeration that owns this element.

2.5.1.3 Record

A record is a data type that is composed of a fixed number of fields. Each field is identified by a name and is associated to a data type. A value of a record is composed of values for every field, where each value conforms to the type of its corresponding field. Values of records can be specified via the meta-class RecordInstance.

Generalizations: DataType

Aggregations

- field : Field [1..*] The non-empty set of fields owned by the record.

Constraints Recordss are subject to the following constraints:

1. All fields in a record must have distinct names:

   context Record **inv** uniqueFieldNames:
   self.field→isUnique(field | field.name)

2.5.1.4 Field

A field represents a part of a record.

Generalizations: NamedElement

Associations

- type : DataType [1] The data type of the field.
- record : Record [1] The record that owns the field.

2.5.1.5 Array

An array is a composite type whose values are a sequence of elements of the same data type. Individual elements are accessed by their position in the array, from 0 to $size - 1$, as defined by meta-class ArrayNavigation.

Generalizations: DataType

Aggregations

- size : Expression [1] The size of the array.

Associations

- elementType : DataType [1] The data type of all the elements in the array.

Constraints Arrays are subject to the following constraints:

1. The size expression must be of integer type:

   context Array **inv** typeIsInteger:
   self.size.type().isInteger()

2. The size expression must be statically evaluable to a non-negative, non-null value.

2.5.1.6 PrimitiveType

A primitive type is a data type which does not have any relevant substructure. A primitive type may have an upper bound and a lower bound to explicitly define the upper most and the lower most value of the type's range. It may define an implementation size on the hosting machine, in terms of number of bits. It may also define the precision of the numeric value that it represents and finally, a primitive type may be typed by a unit and a dimension.

A primitive type may have a kind. A kind determines what values conform to the primitive type. The possible kinds are Boolean, Integer, Real, String, and Void. Primitive types are therefore organized in families of a same kind. Two types of the same family are considered conformant. For example, an expression whose type is of kind Integer can always be the right-hand side of an assignment where the left-hand side is also of kind Integer, regardless of the name, unit or precision of the types. In addition, in that case the left-hand side can also be of kind Real because the family of kind Integer is considered to be a subtype of the family of kind Real.

Generalizations: DataType

Attributes

- kind : PrimitiveTypeKind [1] Specifies the kind of the primitive type.

Aggregations

- upperBound : Expression [0..1] The upper bound that defines the upper most value in the range of this type.
- lowerBound : Expression [0..1] The lower bound that defines the lower most value in the range of this type.
- precision : Expression [0..1] The precision that defines the smallest amount of information (quantum) that this type can express.
- nbBits : Expression [0..1] The size of the type in terms of number of bits.

Associations

- unit : Unit [1] The unit of measurement for this primitive type.

Operations

- isBoolean() : Boolean Returns true iff the tested data type is of kind Boolean. It is redefined from DataType.

```
context PrimitiveType::isBoolean(): Boolean
    post: result = (self.kind = PrimitiveTypeKind::boolean)
```

- isInteger() : Boolean Returns true iff the tested data type is of kind Integer. It is redefined from DataType.

```
context PrimitiveType::isInteger(): Boolean
    post: result = (self.kind = PrimitiveTypeKind::integer)
```

- isString() : Boolean Returns true iff the tested data type is of kind String. It is redefined from DataType.

```
context PrimitiveType::isString(): Boolean
    post: result = (self.kind = PrimitiveTypeKind::string)
```

- isReal() : Boolean Returns true iff the tested data type is of kind Real. It is redefined from DataType.

```
context PrimitiveType::isReal(): Boolean
    post: result = (self.kind = PrimitiveTypeKind::real)
```

- isVoid() : Boolean Returns true iff the tested data type is of kind Void. It is redefined from DataType.

```
context PrimitiveType::isVoid(): Boolean
    post: result = (self.kind = PrimitiveTypeKind::voidType)
```

- conformsTo(DataType) : Boolean Determines whether the primitive type conforms to another data type. When kinds are defined, a primitive type conforms to a primitive type of the same kind or of a "super-kind", where Real is a super-kind of Integer.

```
context PrimitiveType::conformsTo(other: DataType): Boolean
    post: result =
        if (other.oclIsKindOf(PrimitiveType)) then
            let otherKind : PrimitiveTypeKind =
                other.oclAsType(PrimitiveType).kind
            in self.kind = otherKind or
                (self.kind = PrimitiveTypeKind::integer and
                    otherKind = PrimitiveTypeKind::real) or
                (self.kind.oclIsUndefined() or otherKind.oclIsUndefined())
        else
            false
        endif
```

2.5.1.7 PrimitiveTypeKind {*Enumeration*}

An enumeration that defines the kinds supported by primitive types (see PrimitiveType). A kind identifies what values conform to a primitive type.

EnumerationLiterals

integer Integer values.

real Real values.

string String values.

boolean Boolean values.

voidType No value. This kind allows typing pure event flows (event flows which carry no value).

2.5.1.8 Real {*PrimitiveType*}

Real is a primitive type denoting floating point values.

2.5.1.9 Unit

This class describes the unit of measurement for a given type. It is typed by a dimension. It inherits from ReusableElement (which inherits from NamedElement), thus the dimension is defined using the name attribute.

Unit is a qualifier of measured values in terms of which the magnitudes of other quantities that have the same physical dimension can be stated. A unit often relies on precise and reproducible ways to measure the unit. For example, a unit of length such as meter may be specified as a multiple of a particular wavelength of light. A unit may also specify less stable or precise ways to express some value, such as a cost expressed in some currency, or a severity rating measured by a numerical scale.

Generalizations: ReusableElement

Attributes

- convFactor : Real [0..1] This parameter allows referencing measurement units to other base units by a numerical factor.
- convOffset : Real [0..1] This parameter allows referencing measurement units to other base units by applying an offset value to them.

Associations

- dimension : Dimension [1] Specifies the dimension of the unit.
- baseUnit : Unit [0..1] Represents the base unit by which a derived measurement unit is created. Basic units do not require this field to be set.

2.5.1.10 Dimension

A dimension is used to type a unit like for example length is the dimension for the unit meter.

It inherits from ReusableElement (which inherits from NamedElement) thus the dimension is defined using the name attribute.

Generalizations: ReusableElement

2.6 Technical Elements

Among other tasks of ZP-AP3 was the specification of a meta-model to allow modeling of technical elements such as hardware units, scheduling properties and means to allocate software components to hardware components. This mostly describes the logical and technical perspectives as introduced in the last section. In this section we describe these means to state scheduling properties and to model hardware elements.

2.6.1 Hardware Elements

This section will introduce some hardware elements like processors, busses, actuators or sensors. Figure 2.58 gives an overview.

As shown in the diagram, the hardware elements are specials of the model components introduced in Section 2.1 and 2.2.

2.6.1.1 Receiver

This element represents a communication part of a technical component which can receive data from a Transmitter or Transceiver.

Generalizations: TechnicalComponent

Aggregations

- rxDataRate : Expression [1] The data rate with wich this receiver can receive data (e. g. 3 Mb/s).

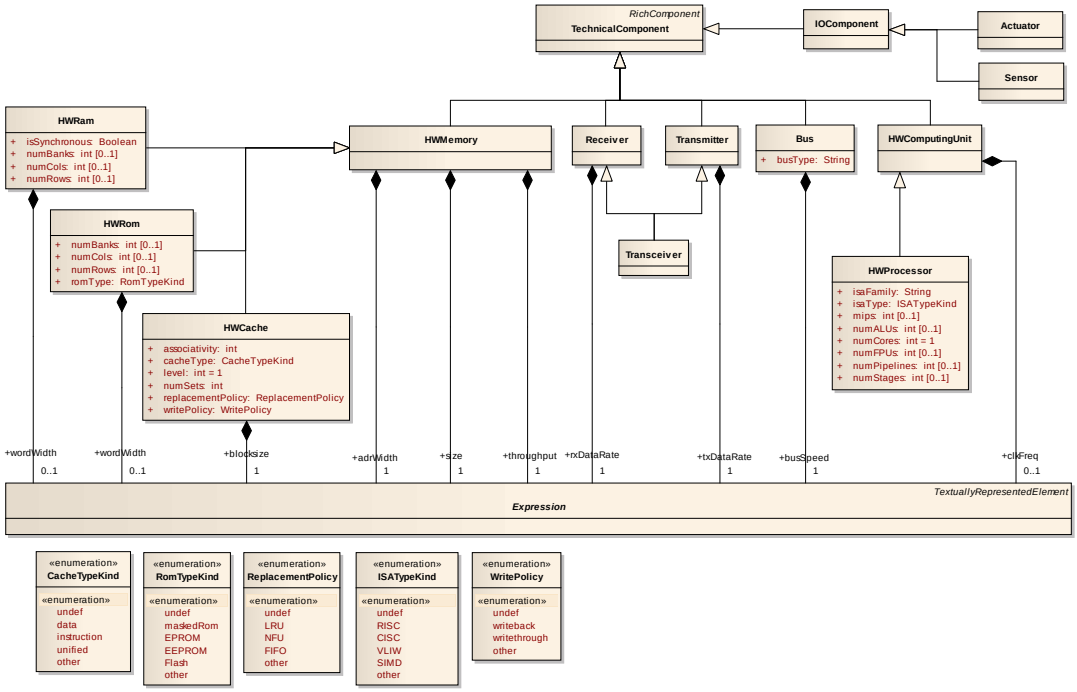

Figure 2.58: Common Hardware Elements.

2.6.1.2 Transmitter

This element represents a communication part of a technical component which can transmit data to a receiver or a transmitter.

Generalizations: TechnicalComponent

Aggregations

- txDataRate : Expression [1] The data rate with wich this transmitter can transmit data (e. g. 3 Mb/s).

2.6.1.3 Transceiver

The transceiver is a combination of receiver and transmitter. The mode decides whether the transceiver can receive and transmit at the same time or not (either receive or transmit at a time).

Generalizations: Receiver, Transmitter

2.6.1.4 Bus

The bus represents logical communication channels. It serves as an allocation target for connectors, i. e. the data exchanged between rich components. The bus carries data from any transmitter to all receivers. Transmitters and receivers are identified by the wires of the bus, i. e. the associated HwConnectors. The available busSpeed represents the maximum amount of useful data that can be carried. The busSpeed has already deducted speed reduction resulting from frame overhead, timing effects, etc. The busType attribute may contain information about the bus type (CAN, FlexRay, ADFX, etc.).

Generalizations: TechnicalComponent

Attributes

- busType : String [1] The type of the bus specified in a string.

Aggregations

- busSpeed : Expression [1] The data rate with wich this bus can transmit or receive data (e. g. 3 Mb/s).

2.6.1.5 IOComponent

An input/output component represents an interaction point with the environment. Its sub-meta-classes are Actuator and Sensor.

Generalizations: TechnicalComponent

2.6.1.6 Actuator

The Actuator is the element that represents electrical actuators, such as valves, motors, lamps, brake units, etc. Non-electrical actuators such as the engine, hydraulics, etc. are considered part of the plant model (environment). The Actuator meta-class represents the physical and electrical aspects of actuator hardware. The logical aspect is represented by the behavioral part.

Generalizations: IOComponent

2.6.1.7 Sensor

Sensor represents a hardware entity for digital or analog sensor elements. The Sensor is connected electrically to the electrical electrical of the hardware architecture (other TechnicalComponents). The logical aspect is represented by the behavioral part.

Generalizations: IOComponent

2.6.1.8 HwComputingUnit

Hardware computing units represent the computer nodes of the embedded electrical/-electronic system. They consist of processor(s) and may be connected to sensors, actuators and other computing units via a HwConnector.

Node denotes an electronic control unit that acts as a computing element executing Functions. In case a single CPU-single core computing unit is represented, it is sufficient to have a single, non-hierarchical Node. The computing unit element represents an allocation target of logical rich components.

Generalizations: TechnicalComponent

Aggregations

- clkFreq : Expression [1] The clock frequency with wich this HwComputingUnit is clocked (e. g. 50 MHz).

2.6.1.9 HwProcessor

A hardware processor is a hardware computing unit that can be described as a general purpose processor (no costum design, something like a commercial-of-the-shelf product).

Generalizations: HwComputingUnit

Attributes

- isaFamily : String [1] Denotes the instruction set architecture (ISA) family this processor is derived of.
- isaType : ISATypeKind [1] Indicates the type of the ISA.
- mips : int [0..1] Optionally indicates the how many instructions per second this processor can perform (million instructions per second). An average rate is stated.
- numALU : int [0..1] Optionally indicates how many arithmetic logic units (ALUs) this processor has.
- numCores : int [0..1] Optionally indicates how many cores this processor has.
- numFPUs : int [0..1] Optionally indicates how many floating point units (FPUs) this processor has.

- numPipelines : int [0..1] Optionally indicates how many pipelines this processor has.
- numStages : int [0..1] Optionally indicates how many pipeline stages this processor has.

2.6.1.10 HwMemory

HwMemory represents memory units present in a technical component. It also has information about its size, throughput and its address size.

Generalizations: TechnicalComponent

Aggregations

- adrWidth : Expression [1] The address size or width of the address specification (e. g. 64 Bit).
- size : Expression [1] The size of the memory unit (e. g. 64 Mb).
- throughput : Expression [1] The data rate with wich this memory unit can store or access data (e. g. 3 Mb/s).

2.6.1.11 HwRam

A Hardware RAM (Random Access Memory) represents a readable and writable memory unit.

Generalizations: HwMemory

Attributes

- isSynchronous : Boolean [1] Indicates whether the RAM is synchronous or not.
- numBanks : int [0..1] Optionally indicates how many banks there are on the RAM module.
- numCols : int [0..1] Optionally indicates how many collumns there are on the RAM module.
- numRows : int [0..1] Optionally indicates how many rows there are on the RAM module.

Aggregations

- wordWidth : Expression [0..1] The size or width of a date that can be accessed or stored in the RAM module (e. g. 64 Bit).

2.6.1.12 HwRom

A Hardware ROM (Read Only Memory) represents a readable only memory unit.

Generalizations: HwMemory

Attributes

- numBanks : int [0..1] Optionally indicates how many banks there are on the ROM.
- numCols : int [0..1] Optionally indicates how many collumns there are on the ROM.
- numRows : int [0..1] Optionally indicates how many rows there are on the ROM.
- romType : RomTypeKind [1] The type of ROM indicated by RomTypeKind.

Aggregations

- wordWidth : Expression [0..1] The size or width of a date that can be accessed in the ROM (e. g. 64 Bit).

2.6.1.13 HwCache

A Hardware Cache is a intermediate memory that stores data for a certain time (depending on the replacement policy) to serve them later in case they are accessed again. As a very fast and expensive hardware component it has a rather limited size.

Generalizations: HwMemory

Attributes

- associativity : int [1] Describes the number of blocks in a set of the cache.
- cacheType : CacheTypeKind [1] Indicates the type of cache.
- level : int [1] Indicates how "close" the cache is to the processing speed (lower level means closer the the actual processable speed).

- numSets : int [1] The number of sets of the cache.
- replacementPolicy : ReplacementPolicyKind [1] The policy (cache algorithm) defining how existing data shall be replaced.
- writePolicy : WritePolicyKind [1] The policy defining how new cache entries are to be written.

Aggregations

- blockSize : Expression [1] The size or width of a block in the cache (e. g. 64 Bit).

2.6.1.14 CacheTypeKind {*Enumeration*}

This enumeration descripes different cache types.

EnumerationLiterals

undef indicates that the cache type is unknown.

data indicates a pure data cache.

instruction indicates a pure instruciton cache.

unified indicates a cache containing both data and instructions.

other indicates that the cache type is not listed in this enumration.

2.6.1.15 RomTypeKind {*Enumeration*}

Describes the type or ROM.

EnumerationLiterals

undef indicates that the ROM type is unknown.

maskedROM indicates a type of ROM where the readable contents are code on the integrated circuit level.

EPROM indicates a erasable programmable ROM.

EEPROM indicates a electrical erasable programmable ROM.

Flash indicates a type of ROM which is writable but can only write much slowlier than read.

other indicates that the ROM type is not listed in this enumration.

2.6.1.16 ReplacementPolicy {*Enumeration*}

Denotes the type of cache replacement algorithm to be used when updating/replacing elements.

EnumerationLiterals

undef indicates that the replacement policy is unknown.

LRU indicates a least recently used (LRU) cache algorithm, if the cache is full, the least recently used data will be replaced by a new data.

NFU indicates a not frequently used (NFU) cache algorithm, if the cache is full, the data with the least average usage will be replaced by a new data.

FIFO indicates a first in first out (FIFO) cache algorithm, if the cache is full, the oldest data is replaced by the newest data.

other indicates that the replacement policy is not listed in this enumration.

2.6.1.17 ISATypeKind {*Enumeration*}

Denotes the instruction set architecture kind.

EnumerationLiterals

undef indicates that the ISA type is unknown.

RISC indicates that the ISA is a reduced instruction set computer (RISC).

CISC indicates that the ISA is a complex instruction set computer (CISC).

VLIW indicates that the ISA is a very long instruction word (VLIW) architecture.

SIMD indicates that the ISA is a single instruction multiple data (SIMD) architecture.

other indicates that the ISA type is not listed in this enumration.

2.6.1.18 WritePolicy {*Enumeration*}

The write policy denotes how the cache handles write operations (indicates the timing of write operations).

EnumerationLiterals

undef indicates that the write policy is unknown.

writeback indicates that writes are not immediately mirrored to the backing store. The overwritten data entry is only marked as “dirty”.

writethrough indicates that every write action causes a synchronous write to the backing store.

other indicates that the write policy is not listed in this enumration.

2.6.2 Resource Modeling and Scheduling

Resources like ComputingResource, CommunicationResource, StorageResource constitute an abstraction of the resources a platform provides and allocated behavior from logical components. Figure 2.59 gives an overview of resources.

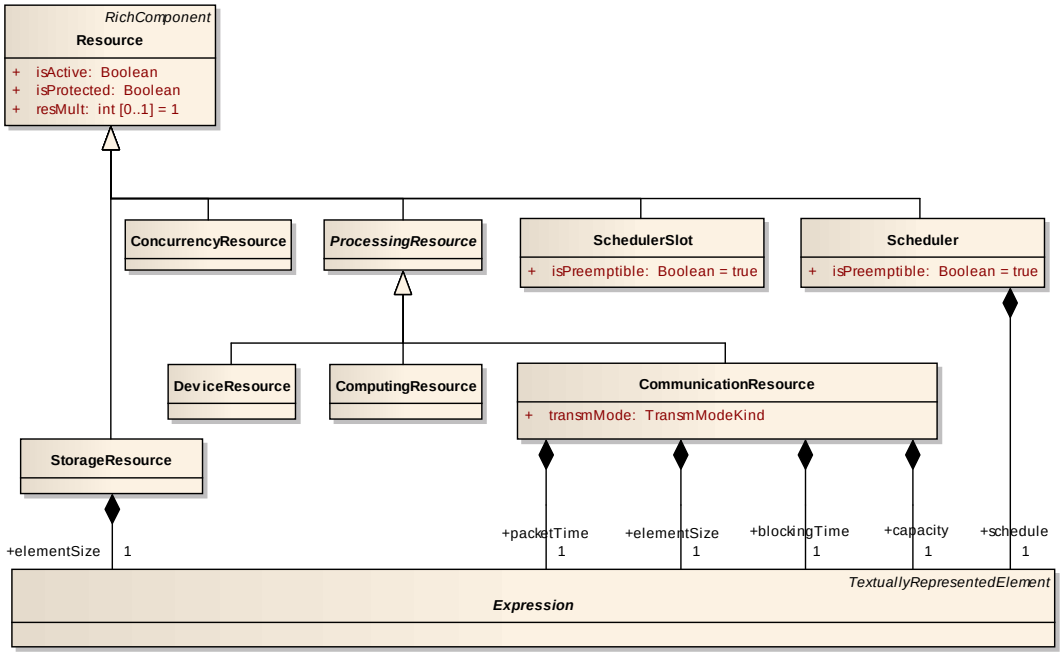

Figure 2.59: Resources.

Any resource is a specialized RichComponent as introduced in Section 2.1.

2.6.2.1 Resource

A Resource is an abstraction of a particular resource a platform provides and the allocated behaviour from components of the logical perspective. It is a structural entity,

that is part of a model of the technical perspective of a system. Through its ports it can be connected to other resources. The specializations of Resource add attributes and parameters typical for a certain kind of resource. The meta-model element Resource shall only be used directly, if none of the specializations does fit.

Generalizations: RichComponent

Attributes

- isActive : Boolean [1] If set to true, this indicates that the resource has its own course of action, i. e. thread of control.
- isProtected : Boolean [1] If true, this indicates that access to the resource must be arbitrated.
- resMult : int [0..1] Defaults to 1. If set to another value, this indicates the maximum number of elementary units of some type of resource accessible through its ports.

2.6.2.2 ProcessingResource {*abstract*}

A ProcessingResource is a generalization of the concepts ComputingResource, DeviceResource and CommunicationResource.

Generalizations: Resource

2.6.2.3 DeviceResource

A device resource is a general not exactly specifyable resource represented by some device (a scenario may be that the device resource is manufactured by a supplier, so that the inner parts are unknown).

Generalizations: Resource

2.6.2.4 ComputingResource

A ComputingResource is a protected active resource, that models a processing device capable of storing and executing some program.

Generalizations: ProcessingResource

2.6.2.5 CommunicationResource

A CommunicationResource models an entity transferring information from one location to another.

Generalizations: ProcessingResource

Attributes

- transmMode : TransmModeKind [1] The transmission mode.

Aggregations

- elementSize : Expression [1] The size of the communication quantum, that can be transmitted (e. g. width of a bus).
- capacity : Expression [1] Specifies the communication-capacity of the element.
- packetTime : Expression [1] The time it takes to transmit an element on this CommunicationResource.
- blockingTime : Expression [1] The time the CommunicationResource is blocked due to transmission of an element.

2.6.2.6 TransmModeKind {*Enumeration*}

TransmModeKind is an enumeration with elements, that can be used as literals for specifying the transmission mode of a CommunicationResource. Such modes are: simplex, half_duplex, full_duplex.

EnumerationLiterals

simplex indicates that data is transmitted in one direction.

half_duplex indicates that data can flow in one direction or the other, but not both directions at the same time.

full_duplex indicates that data can flow in both directions simultaneously. Thus, each end can transmit and receive at the same time.

2.6.2.7 ConcurrencyResource

A ConcurrencyResource is a protected active resource, that is capable of performing its execution concurrently with others. Processing capacity is provided by other resources (e. g. a ComputingResource).

Generalizations: Resource

Aggregations

- required : SchedulerRPort [0..*] {subsets RichComponent.port} The scheduling interface required by this ConcurrencyResource. This port should be connected to a SchedulerSlot, which provides access to its fraction of the capacity of some ProcessingResource on this port.

2.6.2.8 Scheduler

A Scheduler controls access to its arbitrated ProcessingResources following a certain scheduling policy.

Generalizations: Resource

Attributes

- isPreemptible : Boolean [1] Whether this scheduler and its associated clients is allowed to be preempted, e. g. by some higher level scheduler.

Aggregations

- policy : SchedulingPolicy [1] The policy of the scheduler under which it provide access to its arbitrated resource.
- schedule : Expression [0..1] Explicit specification of the schedule. This may have been calculated offline by some tool.
- provided : SchedulerPPort [1..*] {subsets RichComponent.port} The scheduling interface provided by this Scheduler. This port should be connected to some SchedulerSlot, which are clients of this scheduler and thus compete for access to the ProcessingResource managed by this Scheduler.
- required : SchedulerRPort [0..*] {subsets RichComponent.port} The scheduling interface required by this Scheduler. This port should be connected to a SchedulerSlot, which is in turn connected to some other Scheduler. This supports the modelling of hierarchical scheduling.

2.6.2.9 SchedulingPolicy

A SchedulingPolicy specifies the strategy of a scheduler to do its run-time arbitration of a ProcessingResource.

Attributes

- policy : SchedPolicyKind [1] Kind of scheduling policy.
- otherSchedPolicy : String [0..1] If the policy is none of that policies contained in the enumeration SchedulingPolicy, it can be specified here.

Aggregations

- scheduleLength : Expression [0..1] For some scheduling policies there exist a scheduling round, that is repeatedly executed (TimeTableDriven). This specifies the duration in time of such a round. Other policies like RoundRobin typically assign time-slices to each client in equal portions. If such a policy is selected, this attribute specifies the duration of a time-slice.

2.6.2.10 SchedPolicyKind {*Enumeration*}

SchedPolicyKind is an enumeration with elements, that can be used as literals for specifying the scheduling policy of a Scheduler.

EnumerationLiterals

EarliestDeadlineFirst This policy implies, that the client with the most urgent deadline will be granted access to the resource.

FixedPriority This policy implies, that the client with the highest priority will be granted access to the resource.

RoundRobin This policy implies, that access to the resource will be granted in a circular order fashion. Typically each client will be assigned a time-slice.

TimeTableDriven This policy implies, that there is a scheduling round, that will be repeated over and over again and access to the resource will be granted at time points relative to the beginning of such a period.

Undef This policy can be chosen, if the policy of a scheduler can not be categorized according to one of the previous policies.

Other This policy can be chosen, if the policy of a scheduler can not be categorized according to one of the previous policies. The attribute otherSchedPolicy of the aggregating SchedulingPolicy needs to be filled out then.

2.6.2.11 SchedulerSlot

A SchedulerSlot models the client of a scheduler. Thus it specifies necessary attributes used by its associated scheduler to carry out arbitration. The slot itself does not model the functional behaviour that is executed, when access to the associated ProcessingResource is granted. It just provides an execution context for connected ConcurrencyResources.

Generalizations: Resource

Attributes

- isPreemptible : Boolean [1] Whether this slot is allowed to be preempted, e. g. by some scheduler, that withdraws access to the ProcessingResource by this slots and grants access to another slot.

Aggregations

- schedParameters : SchedParameterSpec [1] Parameters used by some scheduler to do arbitration according to its policy.
- required : SchedulerRPort [1] {subsets RichComponent.port} The scheduling interface required by this SchedulerSlot. This port should be connected to a Scheduler, which exposes the behaviour of arbitration of access to its managed ProcessingResource on this port.
- provided : SchedulerPPort [0..*] {subsets RichComponent.port} The scheduling interface provided by this SchedulerSlot. This port should be connected to some ConcurrencyResource, which will actually make use of the processing capacity, once access is granted to this SchedulerSlot by a connected Scheduler.

2.6.2.12 SchedParameterSpec

A SchedParameterSpec is a specification of the parameters of some SchedulerSlot that are used by some scheduler to do arbitration according to its policy.

Generalizations: Variable

2.6.2.13 Task

A Task is a special ConcurrencyResource modeling a computation task. The computing capacity needed by the task is provided either directly by some ComputingResource or by a SchedulerSlot.

Generalizations: ConcurrencyResource

Aggregations

- executionTime : ExecutionTimeSpec [0..*] {subsets RichComponent.attribute} Attribute specifying the execution time of this task.

2.6.2.14 ExecutionTimeSpec

An ExecutionTimeSpec specifies the absolute amount of computational capacity needed by a task. As this specification depends on the actual ComputingResource, which the task is mapped to, a ComputingResource can be referenced as the scope of this execution time specification.

Generalizations: Constant

Associations

- scope : ComputingResource [0..1] The ComputingResouce for which this execution time has been measured/estimated/analyzed.

2.6.2.15 FrameTriggering

A FrameTriggering defines the manner of triggering and identification of a Frame on a CommunicationResource.

Generalizations: ConcurrencyResource

2.6.2.16 SchedulerPort {*abstract*}

A SchedulerPort provides means to specify the scheduling interface of a resource. A Scheduler can expose its arbitration of access to its managed ProcessingResource at such ports. SchedulerSlots connected to that ports in turn can expose their fraction of the capacity of the ProcessingResource granted by the Scheduler to connected ConcurrencyResources.

Generalizations: Port

Associations

- type : SchedulerPortSpec [1] {redefines Port.type} The type of this SchedulerPort.

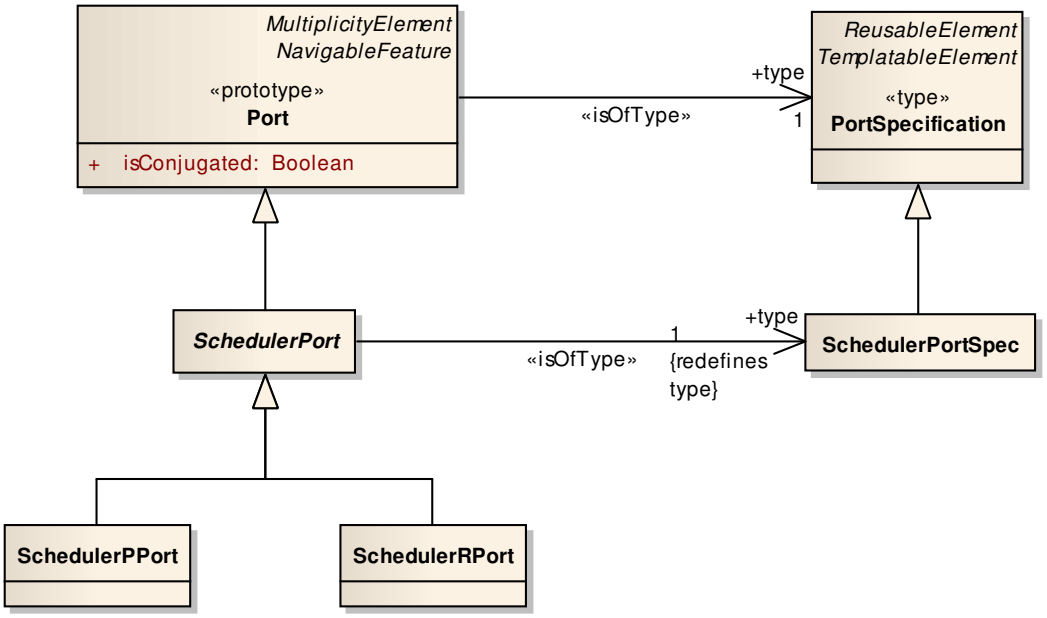

Figure 2.60: Scheduler Ports.

2.6.2.17 SchedulerPPort

A specialization of SchedulerPort that provides a scheduling interface conforming to SchedulerPortSpec to connected SchedulerSlots or ConcurrencyResources.

Generalizations: SchedulerPort

Constraints SchedulerPPorts are subject to the following constraints:

1. The attribute isConjugated inherited from Port must be false.
   Formal OCL constraint TBD.

2.6.2.18 SchedulerRPort

A specialization of SchedulerPort, that requires a scheduling interface conforming to SchedulerPortSpec from connected SchedulerSlots or Schedulers.

Generalizations: SchedulerPort

Constraints SchedulerRPort are subject to the following constraints:

1. The attribute isConjugated inherited from Port must be true.
   Formal OCL constraint TBD.

2.6.2.19 SchedulerPortSpec

A SchedulerPortSpec is a special PortSpecification that defines a scheduling interface for resources. It is used to type SchedulerPorts.

Generalizations: PortSpecification

Aggregations

- schedulerEvent : Flow [3..5] {subsets PortSpecification.interactionPoint} The flows, that constitute this scheduling interface. If preemption shall not be supported by this scheduling interface, only three flows are necessary. Otherwise five flows are needed.

Associations

- activateEvent : Flow [1] The referenced flow shall be triggered by some ConcurrencyResource, when it requires access to the scheduled capacity of some ProcessingResource.
- startEvent : Flow [1] The referenced flow shall be triggered by some Scheduler or forwarded by some SchedulerSlot, when access to the processing capacities of a managed resource is granted.
- finEvent : Flow [1] The referenced flow shall be triggered by some ConcurrencyResource and forwarded by some SchedulerSlot, after it has accessed the processing capacity of a resource. This is interpreted as an indication, that processing has finished and access to the ProcessingResource is relinquished.
- suspendEvent : Flow [0..1] The referenced flow shall be triggered by some Scheduler or forwarded by some SchedulerSlot, when access to the processing capacities of a managed resource has been granted before, but shall now be revoked in favour of another SchedulerSlot. This reference to a flow is only needed, if this scheduling interface shall support preemption.
- resumeEvent : Flow [0..1] The referenced flow shall be triggered by some Scheduler or forwarded by some SchedulerSlot, when access to the processing capacities of a managed resource has been granted and suspended before and now access to the ProcessingResource is granted again. This reference to a flow is only needed, if this scheduling interface shall support preemption.

Constraints SchedulerPortSpecs are subject to the following constraints:

1. The direction of the referenced flow under the role name *activateEvent* must be *FlowDirection::in*
2. The direction of the referenced flow under the role name *startEvent* must be *FlowDirection::out*
3. The direction of the referenced flow under the role name *finEvent* must be *FlowDirection::in*
4. The direction of the referenced flow under the role name *suspendEvent* must be *FlowDirection::out*
5. The direction of the referenced flow under the role name *resumeEvent* must be *FlowDirection::out*
   Formal OCL constraint TBD.

2.6.2.20 ComData {*abstract*}

A ComData provides means to specify pieces of information, that must be handled by the communication facilities of the system. It is a generalization of the concepts Signal, Message and Frame. Figure 2.61 gives an overview.

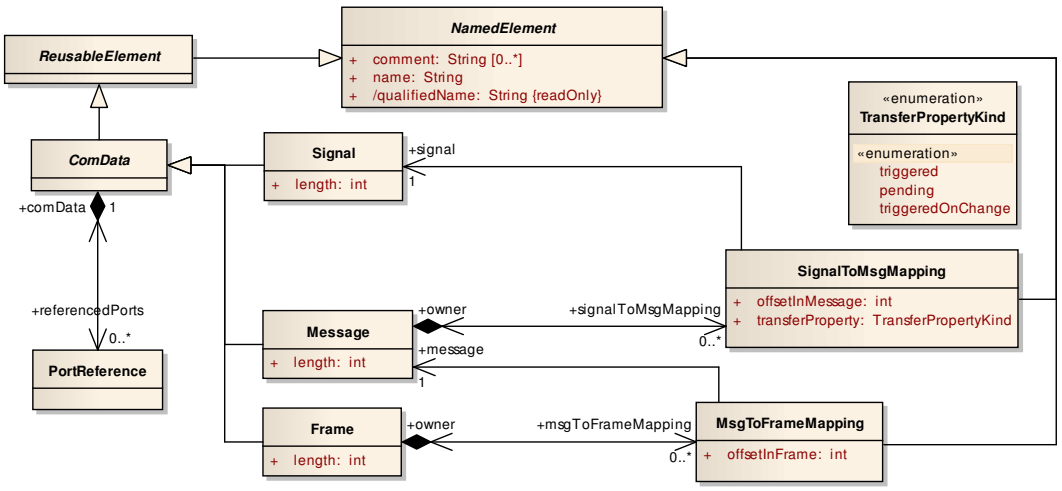

Figure 2.61: Communication Data.

Generalizations: ReusableElement

Aggregations

- referencedPorts : PortReference [0..*] References the ports containing information about the type and thus about the properties of the Signal/Message/Frame.

2.6.2.21 PortReference

A port reference references a port in a unambiguous way. This is done by referencing the complete context of the port as well. Figure 2.62 gives an overview.

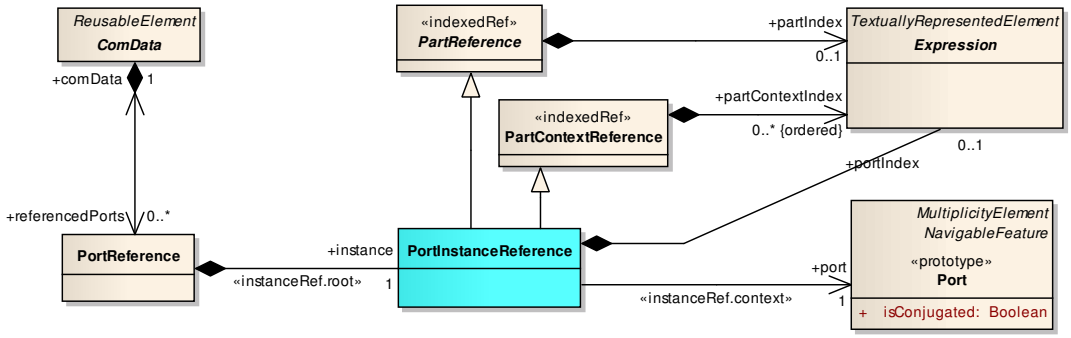

Figure 2.62: Port reference.

Aggregations

- instance : PortInstanceReference [1] References the actual instance of a port.

Associations

- comData : ComData [1] The comunication data this port reference belongs to.

2.6.2.22 PortInstanceReference

The meta-class actually references a port instance (referring to its part context and index).

Generalizations: PartReference, PartContextReference

Aggregations

- portIndex : Expression [0..1] The index of the exact port if a port is a multiple instance.

Associations

- port : Port [1] The port which is referred to.

2.6.2.23 Signal

A Signal models the input or output parameter of some component from the "communication" point of view. It is thus characterized by a sequence of bits.

Generalizations: ComData

Attributes

- length : int [1] The size of the Signal in bits.

2.6.2.24 Message

A Message consists of one or more signals. In a Frame it is simply a sequence of bytes.

Generalizations: ComData

Attributes

- length : int [1] The size of the Message in bytes.

Aggregations

- signalToMsgMapping : SignalToMsgMapping [0..*] The mappings of particular Signals to this Message.

Constraints Messages are subject to the following constraints:

1. At least for one of the aggregated SignalToMsgMappings the attribute *transferProperty* must NOT be set to *TransferPropertyKind::pending*.
   Formal OCL constraint TBD.

2.6.2.25 SignalToMsgMapping

A SignalToMsgMapping specifies how a particular Signal is mapped to a Message, that aggregates this mapping.

Generalizations: NamedElement

Attributes

- offsetInMessage : int [1] This specifies the bit-position of the referenced Signal within a Message.
- transferProperty : TransferPropertyKind [1] Specifies the triggering properties of the Message aggregating this mapping with repsect to the referenced Signal.

Associations

- signal : Signal [1] Reference to the Signal, that shall be mapped to the Message that aggregates this mapping.

2.6.2.26 TransferPropertyKind {*Enumeration*}

TransferPropertyKind is an enumeration with elements, that can be used as literals for specifying the triggering properties of a Message aggregating some mapping, with respect to the mapped Signal.

EnumerationLiterals

triggered This transfer property implies that the Message will be triggered immediately when the mapped Signal has been triggered.

pending This transfer property implies that the Message will NOT be triggered when the mapped Signal has been triggered.

triggeredOnChange This transfer property implies that the Message will be triggered immediately when the mapped Signal has been triggered and its value has changed.

2.6.2.27 Frame

A Frame consists of one or more Messages (and thereby Signals). It is the smallest piece of information, that can be transmitted by a CommunicationResource.

Generalizations: ComData

Attributes

- length : int [1] The size of the Frame in bytes.

Aggregations

- msgToFrameMapping : MsgToFrameMapping [0..*] The mappings of particular Messages to this Frame.

2.6.2.28 MsgToFrameMapping

A MsgToFrameMapping specifies how a particular Message is mapped to a Frame, that aggregates this mapping.

Generalizations: NamedElement

Attributes

- offsetInFrame : int [1] This specifies the byte-position of the referenced Message within a Frame.

Associations

- message : Message [1] Reference to the Message, that shall be mapped to the Frame, that aggregates this mapping.

3 Conclusion

In this document a meta-model for the SPES2020 project was specified. The meta-model specification consists of a common concept meta-model and furthermore provides a tailoring approach for meta-models that shall be mapped to the SPESMM.

In the ongoing work on the SPES Meta-Model for future versions of the meta-model specifications of the application projects (AWP-AV, AWP-AT, AWP-AU, AWP-MT, AWP-EN) will be integrated. Thus, the integration based on the application project's specification will be completed.

Together with the methodology document on how to use the SPESMM this document serves as a reference for all usable and non-usable model artifacts of the SPESMM.

List of Figures

Bibliography

[ATE08] The ATESST Consortium. *EAST ADL 2.0 Specification*, February 2008.

[BRR+10] Andreas Baumgart, Philipp Reinkemeier, Achim Rettberg, Ingo Stierand, Eike Thaden, and Raphael Weber. A model–based design methodology with contracts to enhance the development process of safety–critical systems. In Sang Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 59–70. Springer Berlin / Heidelberg, 2010.

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Carnegie Mellon University, Pittsburgh, USA, 2006.

[ISO07] ISO/IEC/IEEE. ISO/IEC Standard for Systems and Software Engineering – Recommended Practice for Architectural Description of Software–Intensive Systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1 –24, jul. 2007.

[JMM08] Bernhard Josko, Qin Ma, and Alexander Metzner. Designing Embedded Systems using Heterogeneous Rich Components. *Proceedings of the INCOSE International Symposium 2008*, 2008.

[Obj08a] Object Management Group. *OMG Systems Modeling Language (OMG SysML ™)*, November 2008. Version 1.1.

[Obj08b] Object Management Group. *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems*, June 2008. Beta 2.

[Ode98] James J. Odell. *Advanced Object–Oriented Analysis & Design Using UML*, chapter Six different kinds of aggregation, pages 139 – 149. Cambridge University Press, 1998.

[Pro07] Project SPEEDS: WP.2.1 Partners. SPEEDS Meta-model Behavioural Semantics — Complement do D.2.1.c. Technical report, The SPEEDS consortium, 2007.

[RJ01] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, 2001.

[WTR09] Raphael Weber, Eike Thaden, and Philipp Reinkemeier. Requirement Definition for the Reference Architecture. SPES 2020 Deliverable D3.1.A, The SPES 2020 Project, OFFIS, October 2009.