HYBRIS - Efficient Specification and Analysis of Hybrid Systems – Part I: The HybridUML Profile for UML 2.0

Kirsten Berkenkötter Stefan Bisanz Ulrich Hannemann Jan Peleska
University of Bremen
P.O.B. 330 440
28334 Bremen, Germany
{kirsten, bisanz, ulrichh, jp}@informatik.uni-bremen.de

Figure 1: HYBRIS kicks.

Contents

Part I

Introduction

Chapter 1

Hybrid Systems and UML

1.1 Modeling Hybrid Systems

A real-time system is called *hybrid* if it processes *time-continuous* variables in addition to discrete-range parameters. The (piecewise) continuous evolution over dense time of real or complex observables occurs naturally in physical models and in the development of (embedded) control systems monitoring some continuous observables (e.g. temperature, speed) via analog sensors and setting others (e.g. voltage, thrust) using actuators.

Hybrid systems have been studied extensively in various research communities since the early nineties. The definition and investigation of the Duration Calculus (see [ZRH93, Rav95, RRS03] and further references given there) provided fundamental contributions to understanding Hybrid Systems. The introduction of Hybrid Automata [Hen96] demonstrated the feasibility of verification by model checking for hybrid specifications. The applicability of hybrid automata to large-scale systems was improved by the introduction of hierarchical hybrid specifications [AGLS01]. Alternative hierarchical approaches closer to the Statecharts formalism have been described in [KMP00] (together with a proof theory) and [BBB$^+$99] (verification by model checking).

Though today numerous formalisms and verification approaches are available for hybrid systems, their application in an industrial "real-world"-context is still rare. According to our analysis, two main causes are responsible for this situation:

- The syntax developed for hybrid formalisms within research communities was too specialized and not supported by conventional software engineering tools available to practitioners.
- While the underlying theories supported formal verification by theorem proving or model checking, they did not support the development of optimized code for embedded control systems.

With respect to the first cause we suggest to augment existing well-accepted formalisms of software engineering by new specification constructs describing time-continuous behavior. From today's point of view, the Unified Modeling Language UML 2.0 (see [OMG05b, OMG05a])is the best candidate for such an approach: It is currently the most widely known software-engineering formalism supported by a variety of tools. Furthermore, language extension is an inherent feature of UML, therefore well-constructed UML tools should support this extension as well.

The second cause is related to both practical and theoretical considerations: From a practitioner's point of view, the effort invested into formal specification and verification – which will certainly be considerably higher than the effort spent on elaborating informal conventional specifications – is only justified if the specifications can be easily transformed into executable systems. For example, we do not expect that the amount of time required for developing executable code by step-wise refinement will ever be widely accepted among project leaders and developers of embedded systems.

From a theoretic point of view, the problem is even more subtle: If a transformation into executable code is available, how can the consistency between high-level specification semantics and execution behavior of the low-level implementation using conventional programming languages and operating systems be ensured? A practical consequence of this problem consists in the fact that the simulation facilities provided by many case tools never declare which formal high-level semantics has been used as a reference for the encoded simulation behavior.

In "classical" UML, the definition of a universal formal semantics has been deliberately avoided. Instead, the various language constructs are only associated with a general informal meaning so that their purpose in various modeling situations becomes clear. In [RJB99, pp. 105] this approach is motivated by the fact that the

semantic interpretation of specification constructs depends on the specific project context, and precise behavior is only obtained by transformation into the target programming language. While this avoids the obligation to prove consistency between executable system and high-level specification semantics, it still poses the problem that in general, it will be infeasible to capture the potential behavior of software written in Java, C/C++, or Ada, when executed in a specific target environment.

1.2 UML 2.0 and Real-Time

UML 2.0 offers a wide range of possibilities for modeling software systems. Nevertheless, real-time and hybrid constructs are not covered. Real-time systems are systems whose results are not only dependent on computation but also on the time that the computation needs. Hybrid systems are a specialization of real-time systems as they consist of time-discrete and time-continuous observables. Both kinds of system are often used in domains like avionics, chemical processes, or automotive control, so there is also a safety-critical background in many cases.

For this reason, specification, simulation, verification, etc. are strongly needed for developing these kinds of system. This is potentially possible when using UML as specification language, but not without extensions. We therefore propose a UML 2.0 profile that is capable of modeling real-time and hybrid systems; HybridUML. With this profile it is possible to develop formal specifications with UML that can be used for further purposes like model checking.

HybridUML is based on CHARON [AGLS01, ADE+01, ADE+03] which is a formal specification language for modeling hybrid systems. It offers ways of describing both structure and behavior of such a system in a hierarchical way, so that models become scalable and more manageable. The profile takes a subset of UML, modifies it according to CHARON and gives it precise semantics. It also adds useful constructs for modeling hybrid systems not covered by CHARON like events.

Modeling with HybridUML requires using only the elements defined in the profile as only these have semantics. It is highly recommended to work with these elements and not using other constructs known from common UML. It is also an aim of the profile to reduce modeling elements to a set of well-understood ones as this makes models better understandable.

1.3 UML 2.0 Profiles

UML 2.0 offers profiles as a powerful extension mechanism for tailoring UML to specific working areas. Based on a metamodel like the Meta Object Facility (MOF, [OMG06]) or usually UML itself, a profile specifies new model elements called stereotypes.

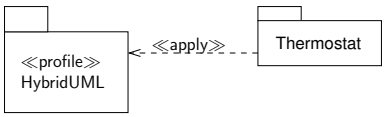

Figure 1.1: Profile Application by Package

Stereotypes customize the used metamodel in different ways: introducing a new terminology, e.g. for Enterprise Java Beans, introducing new syntax, either for elements without syntax or new symbols for elements with syntax, introducing new semantics and constraints, or adding further information like transformation rules from model to code. A set of stereotypes forms a profile.

A profile can be applied by a model or a package in a model. All stereotypes can be used as modeling elements. As every stereotype extends an already known element, the model is still a valid UML model if the profile is taken away. Profile application is visualized by a dependency with the keyword ≪*apply*≫ attached (see Fig. 1.1). The profile itself is a package and therefore depicted like this with the keyword ≪*profile*≫ above its name.

As described before, stereotypes extend elements of the metamodel in use, i.e. they extend a class of the metamodel. Information can be added but not taken away as the model has to be valid without the profile. Generalization respectively specialization of stereotypes is allowed. In the profile, the keyword ≪*stereotype*≫ marks the extended element while its name is given below (see Fig. 4.2). In the model that applies the profile, the name of the stereotype in guillemets is used as a keyword (see Fig. 4.1).

An extension is always binary, i.e. a stereotype is dependent on exactly one element of the underlying metamodel. It is depicted by an arrow with filled arrowhead. The extension can be marked as {*required*}, i.e.

the stereotype is always created if an instance of the extended class is created. In other words, the extension is mandatory in this case.

1.4 Outline

Part II

HybridUML

Chapter 2

HybridUML Overview

Besides the UML 2.0 Profile, we define an independent metamodel for HybridUML. This is done to visualize the HybridUML modeling elements and their relationships independently. On the one hand, the HybridUML profile tailors UML 2.0 to the area of hybrid systems, so it can be used with UML 2.0 tools that support profiles. On the other hand, the metamodel gives us the possibility to show exactly the used elements without the need to support other obfuscating constructs.

To achieve a consistent view of the syntax and semantics of HybridUML, the metamodel consists of three main parts:

- *Abstract Syntax*
  The backbone of HybridUML is defined by means of the Meta Object Facility (MOF). Model elements and their relationships between one another are defined.
- *Concrete Syntax*
  The concrete Syntax of HybridUML is given by a mapping from the HybridUML Profile to the HybridUML abstract syntax.
- *Semantics*
  Semantics of HybridUML are given with respect to the abstract syntax.

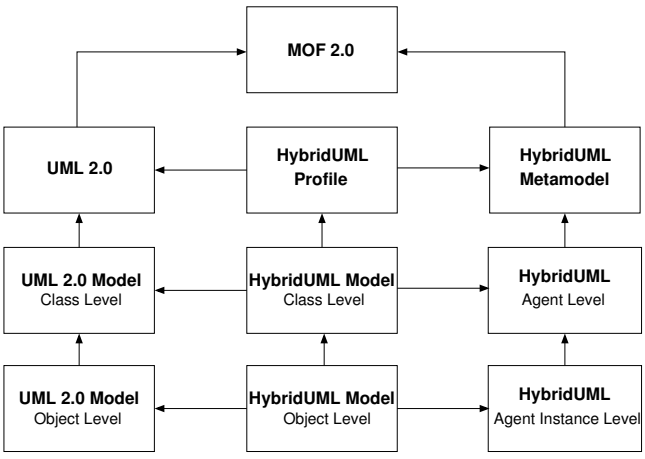

Figure 2.1: HybridUML with respect to UML metalevels

If we consider HybridUML in the UML family of languages as shown in Fig. 2.1, MOF is the basis for all modeling approaches. We also speak of a metametamodel. The UML metamodel is an instance of MOF, just as the HybridUML metamodel. At this level, we define (specification) languages. The HybridUML profile is also located at the metamodel level, but it is not a language itself but an extension of UML. It is also related to the HybridUML metamodel, as it defines its concrete syntax. This is done by a mapping from the HybridUML metamodel constructs to the HybridUML profile.

The next step is the model level, where we define concrete models with our specification language developed on the metamodel level. In case of UML, this would be classes, in case of the HybridUML metamodel, we speak

of agents. In the profile, specific agent classes are used that extend UML classes to fulfill the needs of agents. The last step is the instance level, where the classes, respectively agents are instantiated. On this level, we have objects, respectively agent instances.

Chapter 3

HybridUML MOF-based Metamodel

The HybridUML metamodel describes all elements that can be used in a HybridUML model and represents the abstract syntax of HybridUML. These elements are further subdivided into six packages to group them with respect to their purpose. There are the *Basics* package, the *Types* package, the *Expressions* package, the *Data* package, the *Communications* package, and the *Modes and Agents* package.

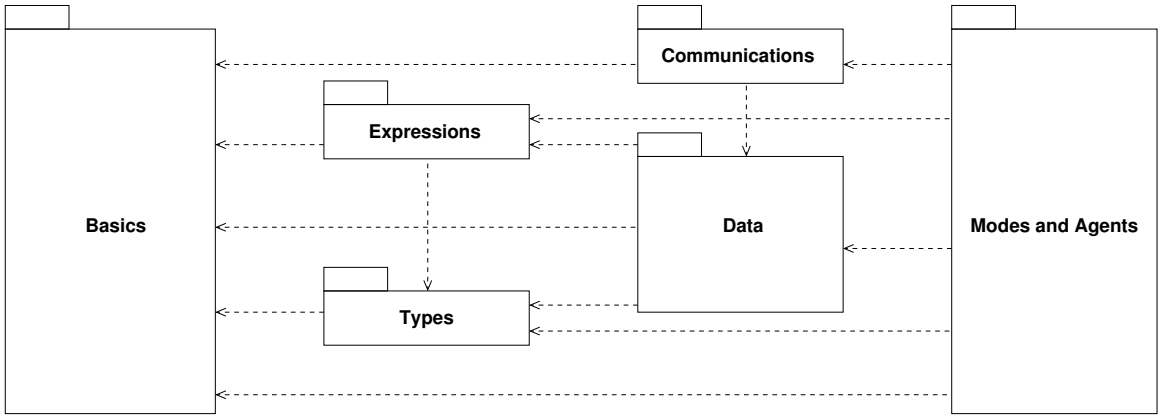

Figure 3.1: HybridUML Metamodel

The << *import* >>-dependencies between these packages show that a package depends on another one, i.e. it uses classes defined there. To give an example, the Communications package uses classes defined in the Basics package and in the Data package while it is used itself by the Modes and Agents package.

3.1 Basics

The *Basics* package of the HybridUML metamodel introduces basic modeling concepts like *NamedElements* or *RedefinableElements* shared by other modeling constructs. Most of these concepts are modeled as abstract classes, so the concrete classes can use them by means of inheritance. Solely *Generalization* is a concrete class that can be used directly.

Most important is *ModelElement* as this is the most basic class at all. Every element used in a HybridUML model must be inherited from ModelElement.

3.1.1 ContainableElement

Description

A ContainableElement is a kind of NamedElement that is capable of hierarchical modeling, i.e. containing other ContainableElements. A ContainableElement can be contained by at most one container, whereas each container can include many ContainableElements. ContainableElement is abstract and must be used by its concrete specializations.

Associations

- containedElement:ContainableElement[0..*]
- container:ContainableElement[0..1]

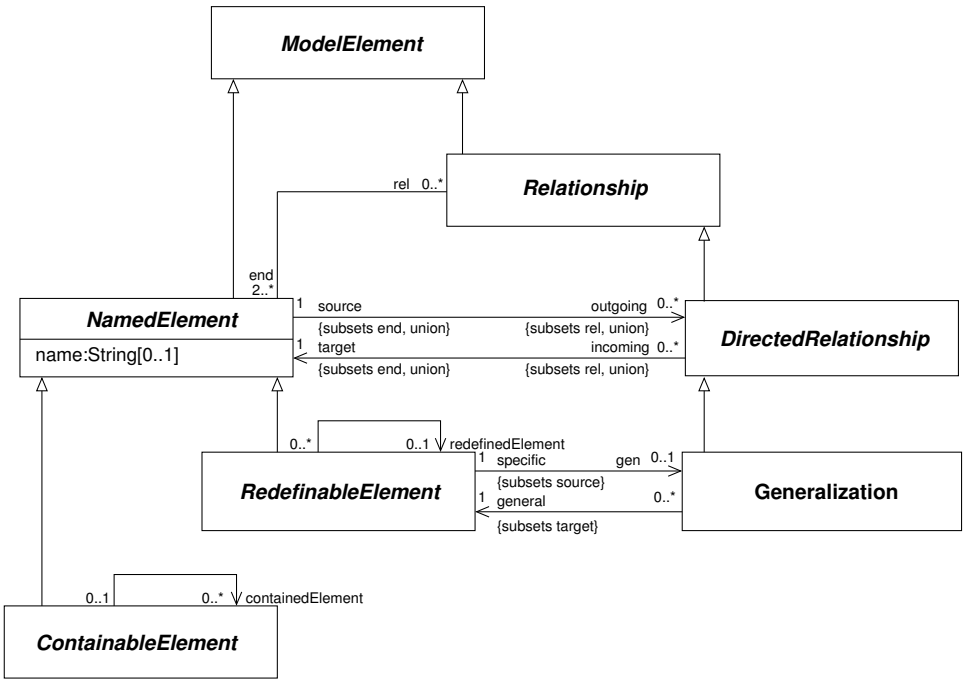

Figure 3.2: Basics package

Attributes

None.

Wellformedness Rules

- `containedElement->forAll(c1,c2 | c1<>c2)` All containedElements differ from each other.
- `containedElement->forAll(c1 | c1<>self)` A ContainableElement cannot contain itself.

3.1.2 DirectedRelationship

Description

Directed Relationship is a kind of Relationship that works only in one direction, from a source to a target that are both NamedElements. These can be part of zero or many DirectedRelationships which are a subset of all Relationships of an element. DirectedRelationship is abstract and must be used by its concrete specializations.

Associations

- target:NamedElement[1]

Attributes

None.

Wellformedness Rules

None.

3.1.3 Generalization

Description

Generalization is a kind of DirectedRelationship that models inheritance. The source of a generalization is a specific RedefinableElement and its target a general RedefinableElement. The source inherits from the target, the concrete inheritance is dependant on the concrete RedefinableElement that uses generalization.

Associations

- general:RedefinableElement[1]

Attributes

None.

Wellformedness Rules

None.

3.1.4 ModelElement

Description

A model consists of ModelElements. Therefore ModelElement is the most basic element in this package. Every element used in a model must be derived from ModelElement. ModelElement itself is abstract and must be used by its concrete specializations.

Associations

None.

Attributes

None.

Wellformedness Rules

None.

3.1.5 NamedElement

Description

NamedElement is a kind of ModelElement that may have a name. It is abstract and must be used by its concrete specializations. NamedElement can be parts of Relationships and DirectedRelationships.

Associations

- rel:Relationship[0..*]
- outgoing:DirectedRelationship[0..*]

Attributes

- name:String[0..1]

Wellformedness Rules

- `name->size()=1 implies (name<>'')` — A name cannot be empty.

3.1.6 RedefinableElement

Description

RedefinableElement is a kind of NamedElement with the additional possibility to be redefined. They can be used in Generalizations. RedefinableElement is abstract and can only be used by its concrete specializations.

Associations

- redefinedElement:RedefinableElement[0..1]
- gen:Generalization[0..1]

Attributes

None.

Wellformedness Rules

- `redefinedElement->size()=1 implies (redefinedElement<>self)` — A RedefinableElement cannot redefine itself.
- `gen->forAll (g:Generalization | g->target<>self)` — A Generalization that originates at a NamedElement cannot have the same NamedElement as a target.

3.1.7 Relationship

Description

Relationship is a kind of ModelElement that is used to model Relationships between NamedElements. A Relationship relates at least one ModelElement (to itself). Each ModelElement can be part of zero or many Relationships.

Associations

- end:NamedElement[2..*]

Attributes

None.

Wellformedness Rules

None.

3.2 Types

In the package *Types*, the idea of *TypedElements* and *DataTypes* is introduced. Several primitive types like *Real*, *Integer*, *Boolean*, and *String* are defined. To cope with the continuous-valued variables and time, *AnalogReal* and it subtypes *Clock* and *CounterClock* are used. In addition, *enumerations* are a data type that has a set of literals as its value domain.

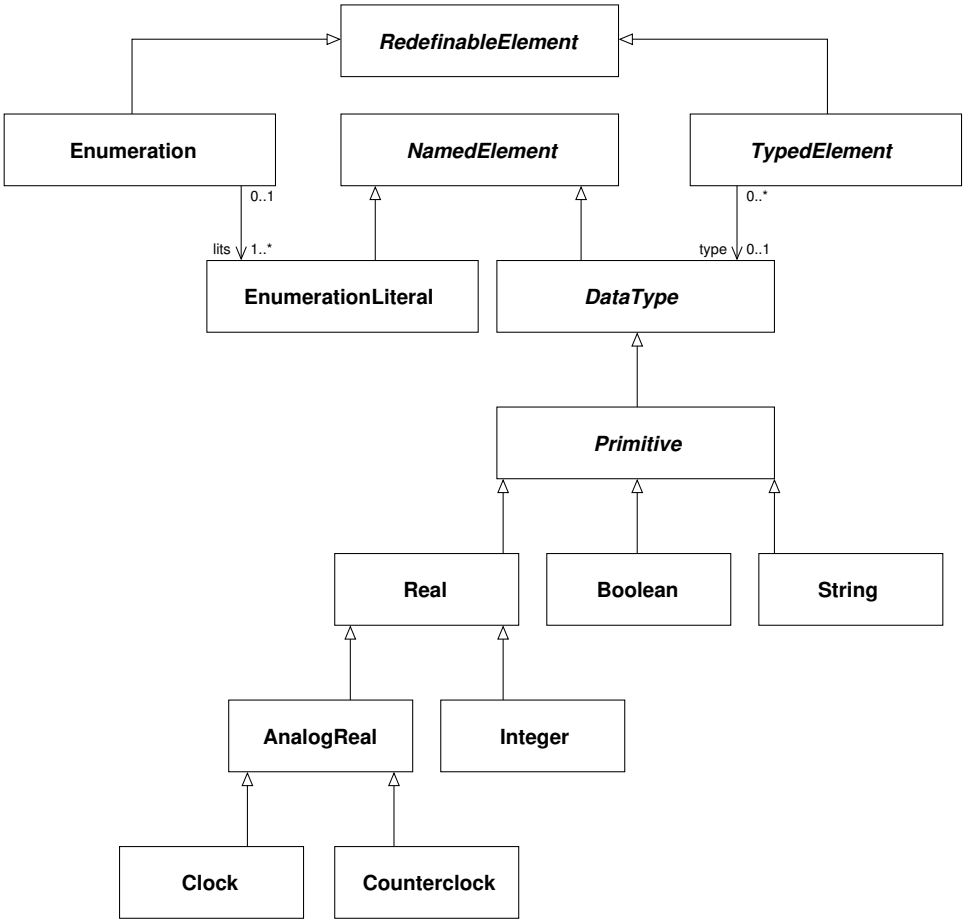

Figure 3.3: Types package

3.2.1 AnalogReal

Description

AnalogReal is a kind of Real and therefore also a Primitive, a DataType, and a NamedElement by inheritance. It is used as data type of continuous-valued variables.

Associations

None.

Attributes

None.

Wellformedness Rules

- `name='AnalogReal'` The name of the data type is AnalogReal.

3.2.2 Boolean

Description

Boolean is a kind of Primitive and therefore also a DataType and a NamedElement by inheritance. It is used as data type of variables with values in $\{true, false\}$.

Associations

None.

Attributes
None.
Wellformedness Rules
- `name='Boolean'` The name of the data type is Boolean.

3.2.3 Clock

Description
Clock is a kind of AnalogReal and therefore also a Real, a Primitive, a DataType, and a NamedElement by inheritance. It is used as data type of variables that model time.
Associations
None.
Attributes
None.
Wellformedness Rules
- `name='Clock'` The name of the data type is Clock.

3.2.4 Counterclock

Description
CounterClock is a kind of AnalogReal and therefore also a Real, a Primitive, a DataType, and a NamedElement by inheritance. It is used as data type of variables that model time, but in contrast to Clock it is an egg-timer where time flows downwards.
Associations
None.
Attributes
None.
Wellformedness Rules
- `name='CounterClock'` The name of the data type is CounterClock.

3.2.5 DataType

Description
DataType is a NamedElement whose name gives the type of a variable. It is abstract and can only be used by its concrete specializations.
Associations
None.
Attributes
None.
Wellformedness Rules
- `name->size()=1` A DataType must have a name.

3.2.6 Enumeration

Description
Enumeration is a DataType and therefore also a NamedElement. The name of the Enumeration is the name of the specified DataType. An Enumeration consists of EnumerationLiterals that together form the set of possible values a variable of this type can possess. Enumeration is also a RedefinableElement.
Associations
- lits:EnumerationLiteral[1..*]

Attributes
None.
Wellformedness Rules

- `name->size()=1` — An Enumeration must have a name.
- `lits->forAll(l1,l2 | l1.name<>l2.name)` — All EnumerationLiterals must have distinct names.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf(Enumeration) and let r:Enumeration= redefinedElement.oclAsType(Enumeration) in ((r.lits->size()>self.lits->size()) and (r.lits->forAll(l1 | self.lits->exists (l2 | l1=l2))))` — An Enumeration can only redefine another Enumeration by adding additional EnumerationLiterals. Removing of EnumerationLiterals is not allowed.
- `(redefinedElement->size()=1 implies (gen ->size()=1)) and (gen->size()=1 implies (redefinedElement->size()=1)) and gen->forAll (g:Generalization | g.specific implies redefinedElement->includes(g.specific))` — An Enumeration is redefined by means of generalization.
- `rel->size()=0` — There are no relationships.

3.2.7 EnumerationLiteral

Description

An EnumerationLiteral is a NamedElement whose name defines a value of an Enumeration.

Associations

None.

Attributes

None.

Wellformedness Rules

- `name->size()=1` — An EnumerationLiteral must have a name.
- `rel->size()=0` — There are no relationships.

3.2.8 Integer

Description

Integer is a kind of Real and therefore also a Primitive, a DataType, and a NamedElement by inheritance. It is used as data type of variables with values in **Z**.

Associations

None.

Attributes

None.

Wellformedness Rules

- `name='Integer'` — The name of the data type is Integer.

3.2.9 Primitive

Primitive is a DataType and therefore also a NamedElement. Is is abstract and can only be used by its concrete specializations.

Description

Associations

None.

Attributes

None.

Wellformedness Rules

- `name->size()=1` — An Primitive must have a name.
- `rel->size()=0` — There are no relationships.

3.2.10 Real

Description
Real is a kind of Primitive and therefore also a DataType and a NamedElement by inheritance. It is used as data type of variables with values in $\mathbf{R}$.
Associations
None.
Attributes
None.
Wellformedness Rules

- `name='Real'` — The name of the data type is Real.

3.2.11 String

Description
String is a kind of Primitive and therefore also a DataType and a NamedElement by inheritance. It is used as data type of variables with values in $\mathcal{A}^*$ where $\mathcal{A}^*$ is the set of finite sequences of Alphabet $\mathcal{A}$. The alphabet is given by concrete syntax.
Associations
None.
Attributes
None.
Wellformedness Rules

- `name='String'` — The name of the data type is String.

3.2.12 TypedElement

Description A TypedElement is a kind of RedefinableElement and therefore also a NamedElement by inheritance. Each TypedElement has a DataType. TypedElement is abstract and must be used by its concrete specializations.
Associations

- type:DataType[0..1]

Attributes
None.
Wellformedness Rules

- A TypedElement can only redefine another TypedElement. In this case, either the data types conform to another by specialization (e.g. Integer conforms to Real) or by redefinition (the type of the redefining variable is a redefinition of the type of the redefined variable).

```
redefinedElement->size()=1 implies
   (redefinedElement.oclIsTypeOf
      (TypedElement)
   and
   let r:TypedElement=
      redefinableElement.oclAsType
         (TypedElement)
   in
      (((r.type.oclIsKindOf(Real)) and
         (self.type.oclIsTypeOf(Integer)))
      or
      (self.type.redefinedElement.oclIsKindOf(r.type)))))
```

3.3 Expressions

The *Expressions* package defines elements used to model expressions and constraints. In general, expressions are used to model the concrete expressions needed in a model like algebraic terms, while constraints are used to attach Expressions to ModelElements.

Concrete elements used in HybridUML models are *Constraints* that attach *Expressions* to ModelElements. These can be either *AlgebraicExpressions* that model algebraic terms which evaluate to real numbers, *IntegerExpressions* that model algebraic terms which evaluate to integers, *DifferentialExpressions* that model algebraic terms evaluating to real numbers over time, and *BooleanExpressions* that model logical terms which evaluate to true or false. Furthermore *StringExpressions* evaluate to strings.

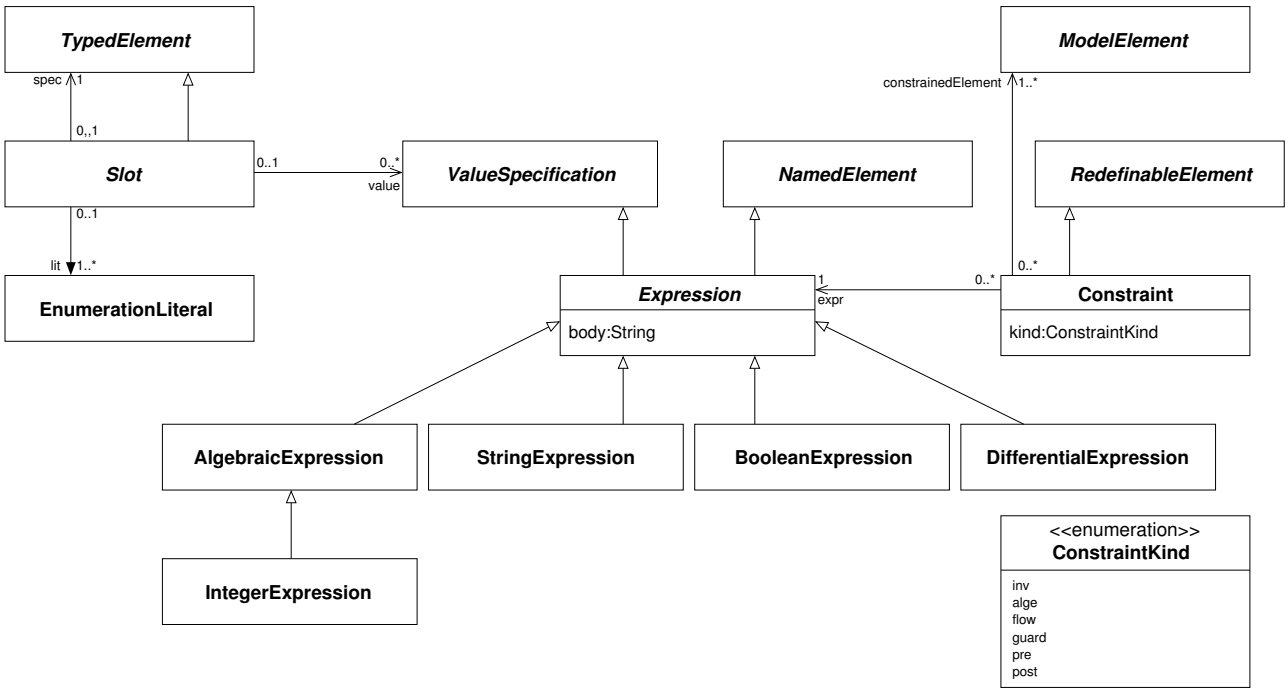

Figure 3.4: Expressions package

3.3.1 AlgebraicExpression

Description

AlgebraicExpression is a kind of Expression and therefore also a ValueSpecification, a TypedElement, and a NamedElement by inheritance. It is used to model expressions that can be evaluated to a real value.

Associations

None.

Attributes

None.

Wellformedness Rules

- `type.oclIsKindOf(Real)` An AlgebraicExpression evaluates to Real.

3.3.2 BooleanExpression

Description

BoolanExpression is a kind of Expression and therefore also a a ValueSpecification and a NamedElement by inheritance. Is is used to model expressions that can be evaluated to a boolean value.

Associations

None.

Attributes

None.

Wellformedness Rules

- `type.oclIsTypeOf(Boolean)` A BooleanExpression evaluates to Boolean.

3.3.3 Constraint

Description

A Constraint is used to attach an expression to ModelElements. An expression can be contained by zero or many constraints while a constraint holds always one expression. Constraints are RedefinableElements and therefore also NamedElements by inheritance.

Associations

- constrainedElement:ModelElement[1..*]
- expr:Expression[1]

Attributes

- kind:ConstraintKind[1]

Wellformedness Rules

- `(redefinedElement->size()=1) implies (redefinedElement->oclIsTypeOf(Constraint) and let r:Constraint= redefinedElement.oclAsType(Constraint) in ((r.expr<>self.expr) and (r.constrainedElement= self.constrainedElement)))` — A Constraint can only redefine another Constraint by changing the associated Expression. The constrainedElement cannot be changed.
- `((kind=inv) or (kind=guard) or (kind=pre) or (kind=post)) implies expr.oclIsTypeOf(BooleanExpression)` — A Constraint that is an invariant, a guard, a precondition, or a postcondition has an attached BooleanExpression.
- `(kind=flow) implies expr.oclIsTypeOf(DifferentialExpression)` — A Constraint that is a flow condition has an attached DifferentialExpression.
- `(kind=alge) implies expr.oclIsTypeOf(AlgebraicExpression)` — A Constraint that is an algebraic condition has an attached AlgebraicExpression.

3.3.4 ConstraintKind

Description

ConstraintKind is an enumeration that lists all possible kinds a Constraint can have, i.e. *flow* condition, *alge*braic expression, *inv*ariant, *guard* condition, *post* condition, or *pre* condition.

Associations

None.

Attributes

- inv
- flow
- alge
- guard
- pre
- post

Wellformedness Rules

None.

3.3.5 DifferentialExpression

Description

DifferentialExpression is a kind of Expression and therefore also a ValueSpecification and a NamedElement by inheritance. Is is used to model expressions that can be evaluated to a real value over time.

Associations

None.

Attributes

None.

Wellformedness Rules

- `type.oclIsKindOf(Real)` — A DifferentialExpression evaluates to Real.

3.3.6 Expression

Description

Expression is a kind of ValueSpecification and of NamedElement. It is used to model all kinds of expressions. Expression is abstract and must be used by its concrete subtypes.

Associations

None.

Attributes

- body:String[1]

Wellformedness Rules

- `rel->size()=0` — There are no outgoing relationships.
- `type->size()=1` — Expressions have a type.

3.3.7 IntegerExpression

Description

IntegerExpression is a kind of AlgebraicExpression and therefore also an Expression, a ValueSpecification, and a NamedElement. Is is used to model expressions that can be evaluated to an integer value.

Associations

None.

Attributes

None.

Wellformedness Rules

- `type.oclIsKindOf(Integer)` — An IntegerExpression evaluates to Integer.

3.3.8 Slot

Description

A Slot is a TypedElement that attaches ValueSpecifications to a TypedElement, i.e. it links a TypedElement and its value(s). It is possible that a value is absent, it is also possible that there is more than one value attached to a TypedElement. A Slot is a RedefinableElement and a NamedElement by inheritance. Slot is abstract and must be used by its concrete specializations.

Associations

- spec:TypedElement[1]
- value:ValueSpecification[0..*]
- lit:EnumerationLiteral[0..*]

Attributes

None.

Wellformedness Rules

- `redefinedElement->size()=0` — A Slot is not redefined.
- `outgoing->size()=0` — A Slot is not part of DirectedRelationships.
- `name=spec.name` — A Slot has the same name as its associated TypedElement that is used as specification.
- `(type=spec.type)` — The type of a Slot conforms to the type of its associated TypedElement.
- `spec.type.oclIsTypeOf(String) implies (lit->size()=0) and (value->forAll(oclIsTypeOf (StringExpression)))` — If the associated TypedElement has as type String, the value specification must be a StringExpression.
- `spec.type.oclIsTypeOf(Real) implies (lit->size()=0) and (value->forAll(oclIsKindOf (AlgebraicExpression)))` — If the associated TypedElement has as type Real, the value specification must be an AlgebraicExpression.
- `spec.type.oclIsKindOf(AnalogReal) implies (lit->size()=0) and (value-> forAll(oclIsKindOf (DifferentialExpression) or value->forAll(oclIsKindOf (AlgebraicExpression)))` — If the associated TypedElement has as type Real, the value specification must be a DifferentialExpression or an AlgebraicExpression.
- `spec.type.oclIsTypeOf(Integer) implies (lit->size()=0) and (value->forAll(oclIsKindOf (IntegerExpression)))` — If the associated TypedElement has as type Integer, the value specification must be an IntegerExpression.
- `spec.type.oclIsTypeOf(Boolean) implies (lit->size()=0) and (value->forAll(oclIsTypeOf (BooleanExpression)))` — If the associated TypedElement is of type Boolean, the value specification must be a BooleanExpression.
- `spec.type.oclIsTypeOf(Enumeration) implies ((value->size()=0)` — If the associated TypedElement is of type Enumeration, the value specification must be an EnumerationLiteral.

3.3.9 StringExpression

Description

StringExpression is a kind of Expression and therefore also a a ValueSpecification and a NamedElement by inheritance. Is is used to model expressions that can be evaluated to a String.

Associations

None.

Attributes

None.

Wellformedness Rules

- `type.oclIsKindOf(String)` — A StringExpression evaluates to String.

3.3.10 ValueSpecification

Description A ValueSpecification is a TypedElement and therefore also a RedefinableElement and a NamedElement by inheritance. A ValueSpecification describes the definition of a value.

Associations

None.

Attributes

None.

Wellformedness Rules

- `type->size()=1` — A ValueSpecification has a type.
- `rel->size()=0` — There are no outgoing relationships.
- `redefinedElement->size()=0` — ValueSpecifications are not redefined.
- `name->size()=0` — A ValueSpecification does not have a name.

3.4 Data

The package *Data* describes the usage of data and data types in HybridUML Models. This can be either in form of *Variables* or in form of *Signals* equipped with *Parameters*. We distinguish *ActualParameters* and *FormalParameters*.

Variables can be grouped in *Structures* to define a new data type.. Here, *Operations* can be used to work on these variables. Pre- and postconditions of operations are defined by constraints. *VariableSlots* are used to assign ValueSpecifications to Variables.

Variables, signals, and operations are *MultiplicityElement*s, i.e. there is an lower and upper bound. Access to variables is given by its *VariableAccessKind* that can be either *readOnly* or *readWrite*. Similarly, signals have a *SignalAccessKind* that is *send* or *receive*.

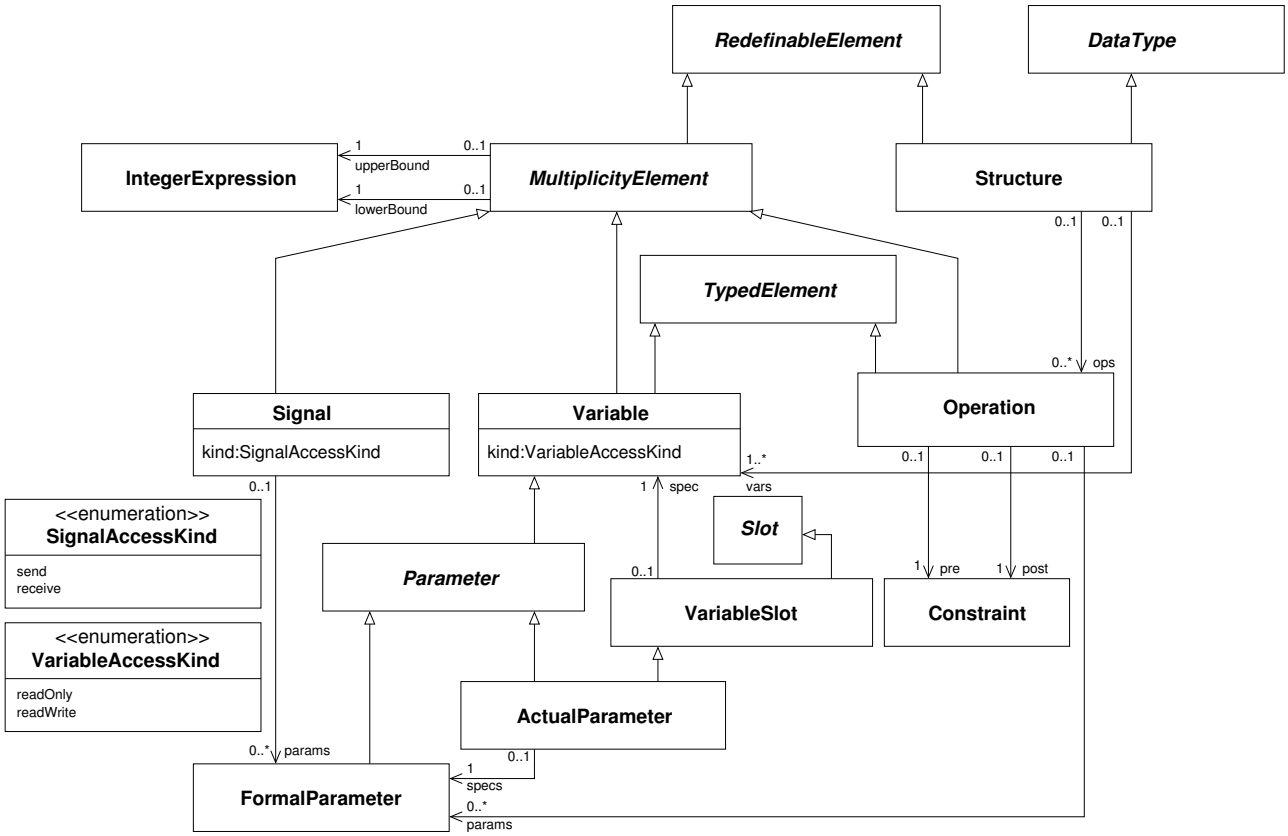

Figure 3.5: Data package

3.4.1 ActualParameter

Description

ActualParameter is a kind of Parameter and of VariableSlot and therefore also a Slot, a Variable, a MultiplicityElement, a TypedElement, a RedefinableElement, and a NamedElement by inheritance. It is used as a actual parameter, i.e. it assigns ValueSpecifications to FormalParameters.

Associations

- spec:FormalParameter[1]

Attributes

None.

Wellformedness Rules

- `redefinedElement->size()=0` An ActualParameter is not redefined.

3.4.2 FormalParameter

Description

FormalParameter is a kind of Parameter and therefore also a Variable, a MultiplicityElement, a TypedElement, a RedefinableElement, and a NamedElement by inheritance. It is used for specifying formal parameters of Agents, Modes, Operations, and Signals.

Associations

None.

Attributes

None.

Wellformedness Rules

- `name->size()=1` — A FormalParameter must have a name.

3.4.3 MultiplicityElement

Description

MultiplicityElement is RedefinableElement and therefore also a NamedElement. Is is used to model sets of elements with a given upper and lower boundary to model the size() of the set. To allow expression like $n + 4$, the boundaries are IntegerExpressions. MultiplicityElement is abstract and can only be used by its concrete specializations.

Associations

- upperBound:IntegerExpression[1]
- lowerBound:IntegerExpression[1]

Attributes

None.

Wellformedness Rules

- `upperBound`$\geq$`lowerBound` — The upper boundary must be greater or equal than the lower boundary.
- A MultiplicityElement can only redefine another MultiplicityElement by replacing the IntegerExpressions for the upper and lower boundary.

```
redefinedElement->size()=1 implies
   (redefinedElement.
      oclIsTypeOf(MultiplicityElement)
   and
   let r:MultiplicityElement=
      redefinedElement.oclAsType
         (MultiplicityElement)
   in
      ((r.upperBound<>self.upperBound) or
      (r.lowerBound<>self.lowerBound)))
```

3.4.4 Operation

Description

An Operation is a TypedElement and a MultiplicityElement and therefore also a NamedElement and a RedefinableElement by inheritance. An Operation may have parameters and is associated to a pre- and post-condition to describe the behavior of the Operation. The type of the return value is the type of the operation, if there is no return value, the operation has no type.

Associations

- pre:Constraint[1]
- post:Constraint[1]
- params:FormalParameter[0..*]

Attributes

None.

Wellformedness Rules

- `name->size()=0` — An Operation must have a name.
- `pre.kind=pre` — The kind of the precondition constraint must be pre.
- `post.kind=post` — The kind of the postcondition constraint must be post.
- `lowerBound=1` — The lower boundary is one.
- `rel->size()=0` — An Operation is not involved in Relationships.
- `params->forAll(p1, p2 | p1.name<>p2.name)` — Parameters must have distinct names.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf(Operation)) and (let o:Operation= redefinedElement.oclAsType(Operation) in ((o.pre<>self.pre) or (o.post<>self.post) or ((o.params->size() ≤ self.params->size()) and (o.params->forAll(p1 | self.params->includes(p1) or self.params->one(p2 | p2.redefinedElement=p1))))` — An Operation can only redefine another Operation by changing the pre- or postcondition or by redefining or adding parameters.

3.4.5 Parameter

Description

A Parameter is a Variable and therefore also a MultiplicityElement, a RedefinableElement, and a NamedElement. Parameters are always read-only. Parameter is abstract and can only be used by its concrete specializations.

Associations

None.

Attributes

None.

Wellformedness Rules

- `kind=readOnly` — Parameters are always read-only.
- `redefinableElement->size()=1 implies redefinableElement.oclIsTypeOf(Parameter)` — Parameters can only redefine other Parameters.
- `rel->size()=0` — A Parameter is not part of Relationships.
- `type.oclIsTypeOf(Real) or type.oclIsTypeOf(Integer) or type.oclIsTypeOf(Boolean) or type.oclIsTypeOf(String) or type.oclIsTypeOf(Enumeration) or type.oclIsTypeOf(Structure)` — A Parameter is either an Integer, a Real, a Boolean, a String, an Enumeration or a Structure.
- `type.oclIsTypeOf(Structure) implies type->flatten()->select(x | x.oclIsKindOf(Variable))->forAll (v | not v.oclIsKindOf(AnalogReal))` — If the type of the variable is Structure, then no part of this structure may be AnalogReal.

3.4.6 Signal

Description

Signal is a MultiplicityElement and therefore also a RedefinableElement and a NamedElement by inheritance. A Signal has zero or many FormalParameters. Its kind is given as a SignalAccessKind that is either *receive* or *send*.

Associations

- params:FormalParameter[0..*]

Attributes

- kind:SignalAccessKind[1]

Wellformedness Rules

- `name->size()=1` — A Signal must have a name.
- `lowerBound->size()=1` — The lowerBound is always 1.
- `params->forAll (p1,p2 | p1.name<>p2.name)` — Parameters must have distinct names.
- `rel->size()=0` — Signals are not part of Relationships.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf(Signal) and let r:Signal= redefinedElement.oclAsType(Signal) in ((r.kind=self.kind) and (r.params->size()≤ self.params->size()) and (r.params->forAll (p1 | self.params->includes(p1) or self.params->one(p2 | p2.redefinedElement=p1))))` — A Signal can only redefine another Signal of the same kind. The number of parameters cannot be reduced and all parameters of the redefined signal must be included in the parameters of the redefining signal or redefined by that. Parameters can be added but not removed.

3.4.7 SignalAccessKind

Description

A signal can be either *sent* or *received*.

Associations

None.

Attributes

- send
- receive

Wellformedness Rules

None.

3.4.8 Structure

Description

Structure is a kind of DataType and therefore also a NamedElement by inheritance. It is also a PackageableElement and a RedefinableElement. Each Structure consists of Variables and Operations. The name of a Structure is the name of the DataType is supports. Structures can be used in Generalizations.

Associations

- vars:Variable[1..*]
- ops:Operation[0..*]

Attributes

None.

Wellformedness Rules

- `name->size()=1` — Structures must have a name.
- `vars.forAll(v1,v2 | v1.name<>v2.name)` — The name of all included variables must be distinct.
- `ops.forAll(o1,o2 | o1.name<>o2.name)` — The name of all included operations must be distinct.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf (Structure) and let r:Structure= redefinedElement.oclAsType (Structure) in ((r.vars->size()≤ self.vars->size()) and (r.vars->forAll(v1 | self.vars->includes(v1) or self.vars->one(v2 | v2.redefinedElement=v1))) and (r.ops->size()≤ self.ops->size()) and (r.ops->forAll(o1 | self.ops->includes(o1) or self.ops->one(o2 | o2.redefinedElement=o2))))` — A Structure can only redefine another Structure. Variables can only be added but not removed, variables can also be redefined. The same holds for operations.
- `(redefinedElement->size()=1 implies (gen->size()=1)) and (gen->size()=1 implies (redefinedElement->size()=1)) and gen->forAll (g:Generalization | g.specific implies redefinedElement->includes(g.specific))` — A Structure is redefined by means of generalization.

3.4.9 Variable

Description

Variable is a kind of MultiplicityElement and a kind of TypedElement and therefore also a RedefinableElement and a NamedElement by inheritance. Each Variable has a name and a type.

Associations

None.

Attributes

None.

Wellformedness Rules

- `name->size()=1` — Each variable must have a name.
- `lowerBound=1` — The lowerBound of a variable is always one.
- `rel->size()=0` — Variables are not part of Relationships.
- `redefinedElement->size()=1 implies redefinedElement.oclIsTypeOf(Variable)` — A Variable can only redefine another Variable.

3.4.10 VariableAccessKind

Description

A variable can be either *readOnly* or *readWrite*.

Associations

None.

Attributes

- readOnly
- readWrite

Wellformedness Rules

None.

3.4.11 VariableSlot

Description

A VariableSlot is a kind of Slot and therefore also a TypedElement, RedefinableElement, and a NamedElement by inheritance. It is used to associate a Variable with a ValueSpecification.

Associations

- spec:Variable

Attributes

None.

Wellformedness Rules

- `rel->size()=0`

  A VariableSlot is not involved in Relationships.

- `spec.oclIsTypeOf(Structure) and spec.value->size()≥1 implies spec.vars->flatten()->forAll`

  If the type of the VariableSlot is Structure, then there exists one ValueSpecification for each Variable in the structure that conforms to the type of this variable.

```
(v:Variable |
   v.oclIsTypeOf(Real) implies
      value->isUnique(s |
         s.oclIsTypeOf(AlgebraicExpression))
   and
   v.oclIsTypeOf(Integer implies
      value->isUnique(s |
         s.oclIsTypeOf(IntegerExpression))
   and
   v.oclIsTypeOf(AnalogReal) implies
      value->isUnique(s |
         s.oclIsTypeOf(AlgebraicExpression)
         or
         s.oclIsTypeOf(DifferentialExpression)))
   and
   v.oclIsTypeOf(Boolean) implies
         value->isUnique(s |
         s.oclIsTypeOf(BooleanExpression))
   and
   v.oclIsTypeOf(String) implies
      value->isUnique(s |
         s.oclIsTypeOf(StringExpression))
   and
   v.oclIsTypeOf(Enumeration) implies
      lit->isUnique(l)))
```

3.5 Communications

The *Communications* package describes the way in which communication structures are modeled. Basically, there are *AgentConnectors* that connect *AgentPorts* and *AgentPortSlots* that are the instances of AgentPorts. Each port owns an *AgentInterface* that determines the direction of communication. These classes are all abstract and must be used by their concrete specializations.

There are two ways to communicate: by shared variables and by signal transmission. Therefore concrete communication is modeled either by *VariableConnectors*, *VariablePorts*, and *VariableInterfaces*, or by *SignalConnectors*, *SignalPorts*, and *SignalInterfaces*.

3.5.1 AgentConnector

Description

AgentConnectors are Relationships that are used to model communication lines between two or more AgentPorts or AgentPortSlots. They are abstract and can only be used by their concrete specializations.

Associations

- end:AgentPort[2..*]

Attributes

None.

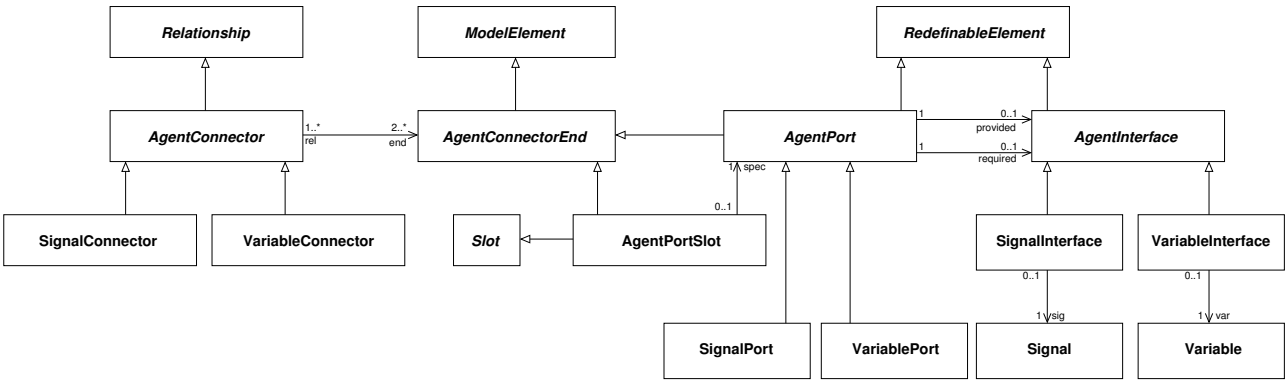

Figure 3.6: Communications package

Wellformedness Rules

- `end->forAll(e1,e2 | e1<>e2)` — All ends of the connector must be distinct.
- `end->select(e | e.oclIsTypeOf(AgentPort))->size()<=1` — There is at most one end of the connector that is an AgentPort.

3.5.2 AgentConnectorEnd

Description

AgentConnectorEnd is a ModelElement that is used to model the end of a connector. AgentConnectorEnd is abstract and can only be used by its concrete specializations.

Associations

None.

Attributes

None.

Wellformedness Rules

None.

3.5.3 AgentInterface

Description

AgentInterface are RedefinableElements and therefore also NamedElements by inheritance. They are used to model interfaces which support either shared variable or signal communication. AgentInterface is abstract and can only be used by its concrete specializations.

Associations

None.

Attributes

None.

Wellformedness Rules

- `redefinedElements->size()=1 implies redefinedElement.oclIsTypeOf (AgentInterface)` — An AgentInterface can only redefine another AgentInterface.

3.5.4 AgentPort

Description

AgentPorts are RedefinableElements and therefore also NamedElements by inheritance. They are used to bind interfaces to Agents and are therefore interaction points of Agents. Each AgentPort owns one required or provided interface. AgentPort is abstract and can only be used by its concrete specializations.

Associations

- required:AgentInterface[0..1]
- provided:AgentInterface[0..1]

Attributes

None.

Wellformedness Rules

- `required->size()+provided->size()=1` — There is exactly one required or provided AgentInterface.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf(AgentPort) and (required->size()=1 implies (self.required.redefinedElement= redefinedElement.required)) and (provided->size()=1 implies (self.provided.redefinedElement= redefinedElement.provided)))` — An AgentPort can only redefine another AgentPort by redefining the associated AgentInterface.
- `outgoing->size()=0` — AgentPorts are not used in DirectedRelationships.
- `(required->size()=1 implies (required.name=self.name)) and (provided->size()=1 implies (provided.name=self.name))` — The name of the port is the name of the associated interface.

3.5.5 AgentPortSlot

Description

An AgentPortSlot is the instance specification of an AgentPort. It is a kind of Slot and therefore also a TypedElement, a RedefinableElement and a NamedElement by inheritance.

Associations

- spec:AgentPort[1]

Attributes

None.

Wellformedness Rules

- `name=spec.name` — The name of the AgentPortSlot is the name of its associated AgentPort.
- `redefinedElement->size()=0` — An AgentPortSlot is not redefined.
- `rel->size()=0` — An AgentPortSlot is not part of relationships.
- `spec.oclIsTypeOf(SignalPort) implies value->size()=0 and lit->size()=0` — If the associated AgentPort is a SignalPort, then there is no value of literal in the slot.
- `spec.oclIsTypeOf(AgentPort) implies required.var->union(provided.var)->flatten()->forAll(v:Variable |` — If the associated AgentPort is a VariablePort, there exists a ValueSpecification for this variable.

```
v.oclIsTypeOf(Real) implies
   value->isUnique(s |
      s.oclIsTypeOf(AlgebraicExpression))
and
v.oclIsTypeOf(Integer implies
   value->isUnique(s |
      s.oclIsTypeOf(IntegerExpression))
and
v.oclIsTypeOf(AnalogReal) implies
   value->isUnique(s |
      s.oclIsTypeOf(AlgebraicExpression)
      or
   s.oclIsTypeOf(DifferentialExpression)))
and
v.oclIsTypeOf(Boolean) implies
   value->isUnique(s |
      s.oclIsTypeOf(BooleanExpression))
and
v.oclIsTypeOf(String) implies
   value->isUnique(s |
      s.oclIsTypeOf(StringExpression))
and
v.oclIsTypeOf(Enumeration) implies
   lit->isUnique(l)))
```

3.5.6 SignalConnector

Description

SignalConnector is a kind of AgentConnector and therefore also a Relationship by inheritance. Is is used to connect SignalPorts and their instances to model signal communication ways.

Associations

- end:AgentPort[2..*]

Attributes

None.

Wellformedness Rules

- `end->forAll(e |` — All ends are SignalPorts or instances of SignalPorts.

```
   e.oclIsTypeOf(SignalPort) or
   (e.oclIsTypeOf(AgentPortSlot) and
e.spec.oclIsTypeOf(SignalPort)))
```

3.5.7 SignalInterface

Description

SignalInterface is a kind of AgentInterface and therefore also a RedefinableElement and a NamedElement by inheritance. Each SignalInterface represents one Signal.

Associations

- signal:Signal[1]

Attributes

None.

Wellformedness Rules

- `name=signal.name` — The name of the interface is the name of the associated signal.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf (SignalInterface) and let r:SignalInterface= redefinedElement.oclAsType (SignalInterface) in (self.signal.redefinedElement= r.signal))` — A SignalInterface can only redefine another SignalInterface by redefining the associated signal.

3.5.8 SignalPort

Description

SignalPort is a kind of AgentPort and therefore also a RedefineableElement and a NamedElement by inheritance. Each SignalPort owns one required or provided SignalInterface.

Associations

None.

Attributes

None.

Wellformedness Rules

- `required->union(provided)->forAll (oclIsTypeOf(SignalInterface))` — The owned interface is a SignalInterface.
- `redefinedElement->size()=1 implies redefinedElement.oclIsTypeOf (SignalPort)` — A SignalPort can only redefine another SignalPort.

3.5.9 VariableConnector

Description

VariableConnector is a kind of AgentConnector and therefore also a Relationship by inheritance. Is is used to connect VariablePorts and their instances to model shared variable communication ways.

Associations

- end:AgentPort[2..*]

Attributes

Wellformedness Rules

- `end->forAll(e | e.oclIsTypeOf(VariablePort) or (e.oclIsTypeOf(AgentPortSlot) and e.spec.oclIsTypeOf(VariablePort)))` — All ends are VariablePorts or their instances.

3.5.10 VariableInterface

Description

VariableInterface is a kind of AgentInterface and therefore also a RedefinableElement and a NamedElement by inheritance. Each VariableInterface represents one shared Variable.

Associations

- var:Variable[1]

Attributes

None.

Wellformedness Rules

- `name=var.name` — The name of the interface is the name of the associated variable.
- `redefinedElement->size()=1 implies (redefinedElement.oclIsTypeOf (VariableInterface) and let r:VariableInterface= redefinedElement.oclAsType (VariableInterface) in (self.var.redefinedElement=r.var))` — A VariableInterface can only redefine another VariableInterface by redefining the associated shared variable.

3.5.11 VariablePort

Description

VariablePort is a kind of AgentPort and therefore also RedefinableElements and NamedElements by inheritance. Each VariablePort owns one required or provided VariableInterface.

Associations

None.

Attributes

None.

Wellformedness Rules

- `required->union(provided)->forAll (oclIsTypeOf(VariableInterface))` — The owned interface is a VariableInterface.
- `redefinedElement->size()=1 implies redefinedElement.oclIsTypeOf (VariablePort)` — A VariablePort can only redefine another VariablePort.
- `type.oclIsKindOf(Primitive) or type.oclIsTypeOf(Enumeration) or type.oclIsTypeOf(Structure)` — The type of a Variable is Primitive, Enumeration or Structure.

3.6 Modes and Agents

The package *Modes and Agents* includes the main modeling elements in HybridUML: *Agent*s, *AgentInstance*s, and *Mode*s. An Agent is used to model the structure of a system, it is itself structured hierarchically by AgentInstances. An Agent that contains AgentInstances is called *composite* Agent in contrast to *basic* Agents. The behavior of a basic Agent is given by a Mode. The behavior of a composite Agent is given by the parallel composition of the Modes of the included basic Agents.

A Mode is a hierarchical statemachine, a Mode can therefore include other Modes. Each Mode owns a set of different constraints: *flow* conditions that describe the valuation of a variable over time, *alge*braic conditions that are evaluated continously, and *inv*ariants that must evaluate to *true* as long as this Mode is active. The interaction points of Modes are *ModePseudostates*, namely *entry* and *exit* points. Each Mode has at least one *default entry* and one *default exit* point. *ModeTransitions* start and end at ModePseudostates. Each ModeTransition has a *guard* constraint that determines if the transition is activated or not. A signal is used to trigger a transition. If the transition is taken, one or more *ModeActivities* can be performed that describe operation calls, signal sending or variables updates.

3.6.1 Agent

Description

An Agent is a kind of RedefinableElement, a kind of ContainableElement, and a kind of DataType and therefore also by inheritance a NamedElement. It is the main modeling construct for modeling structure. An Agent consists of zero to many Parameters, zero to many local Variables, zero to many AgentPorts that provide global Variables and Signals, and zero to many Operations. An Agent that owns a hierarchical internal structure is a composite Agent and owns also connectors. An Agent without internal structure is a basic Agent that owns a Mode.

Associations

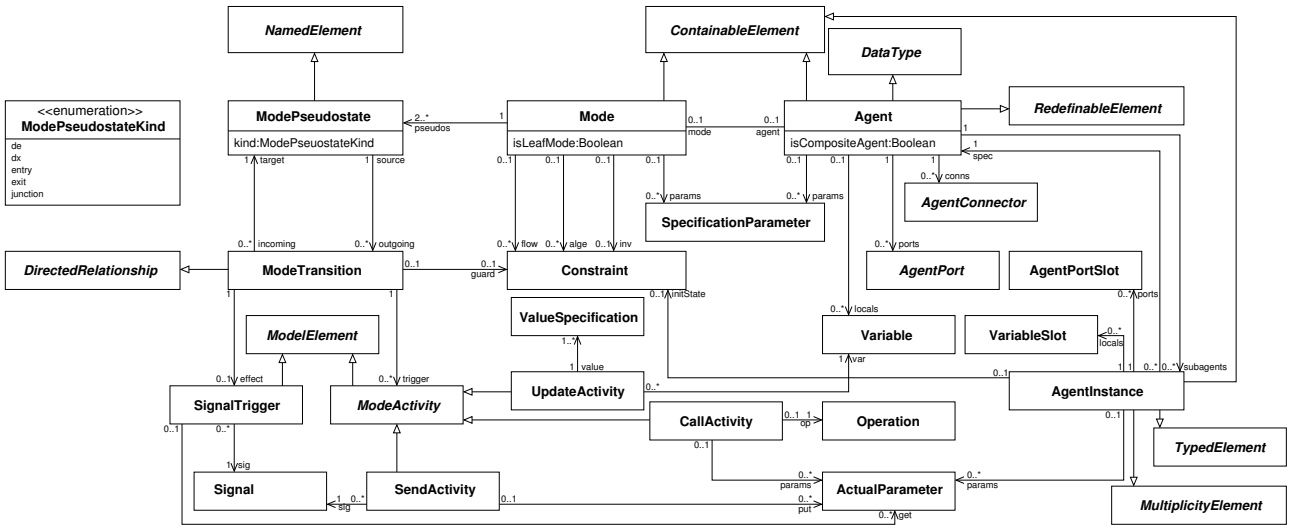

Figure 3.7: Modes and Agents package

- params:SpecificationParameter[0..*]
- locals:Variable[0..*]
- ports:AgentPort[0..*]
- mode:Mode[0..1]
- subagents:AgentInstance[0..*]
- conns:AgentConnector[0..*]
- ops:Operation[0..*]

Attributes

- /active:Boolean
- /isComposite:Boolean

Wellformedness Rules

- `name->size()=1` — An Agent must have a name.
- `isComposite: subagents->size()≥1` — An Agent is a composite Agent if there is at least one subagent.
- `active: mode->size()=1` — An Agent is active if there is an associated Mode.
- `isComposite=true implies mode->size()=0` — A composite Agent owns no Mode.
- `params->union(locals)->union (ports->select(p | p.oclIsTypeOf(VariablePort))-> forAll(v1,v2 | v1.name<>v2.name)` — All parameters, local and global variables have distinct names.
- `containedElement->forAll(c | c.oclIsTypeOf(Agent))` — All containedElements are Agents.
- `subagents->forAll(s | containedElement->includes(a | s.spec=a))` — All contained AgentInstances are instances of contained Agents.
- `redefinedElement->size()=1 implies redefinedElement.subagents->size()=0 and redefinedElement.mode->size()=0` — An Agent can only be redefined if there are no subagents and no mode.
- `redefinedElement->size()=1 implies ((redefinedElement.locals->size() ≤locals->size()) and (redefinedElement.ports->size() ≤ports->size()) and (redefinedElement.params->size() ≤params->size()) and (redefinedElement.ops->size() ≤ops->size()) and (redefinedElement.locals->forAll(l1 | locals->includes(l1) or locals->includes(l2 | l2.redefinedElement=l1))) and (redefinedElement.ports->forAll(p1 | ports->includes(p1) or ports->includes(p2 | p2.redefinedElement=p1))) and (redefinedElement.params->forAll(p1 | params->includes(p1) or params->includes(p2 | p2.redefinedElment=p1))) and (redefinedElement.ops->forAll(o1 | ops->includes(o1) or ops->includes(o2 | o2.redefinedElement=o1))))` — An Agent is redefined by adding or redefining variables, ports, parameters, and operations.
- `redefinedElement->size()=1 implies gen->size()=1` — An Agent is redefined by Generalization.

3.6.2 AgentInstance

Description

An AgentInstance is MultiplicityElement and a TypedElement and therefore also a RedefinableElement and a NamedElement by inheritance. The type of an AgentInstance is given by the corresponding Agent that serves as specification for the AgentInstance. The initial state of variables can be given as a Constraint.

Associations

- spec:Agent[1]
- initState:Constraint[0..1]
- ports:AgentPortSlot[0..*]
- locals:AgentInstance[0..*]
- params:ActualParameter[0..*]

Attributes

None.

Wellformedness Rules

- `ports->forAll(p1 | spec.ports->isUnique(p2 | p2.spec=p1))` — For each AgentPortSlot of the instance there exists an AgentPort in the associated Agent.
- `locals->forAll(l1 | spec.locals->isUnique(l2 | l2.spec=l1))` — For each VariableSlot of the instance there exists a Variable in the associated Agent.
- `params->forAll(p1 | spec.params->isUnique(p2 | p2.spec=p1))` — For each ActualParameter of the instance there exists a FormalParameter in the associated Agent.
- `initState.kind=BooleanExpression` — The initState is a BooleanExpression.
- `type.oclIsKindOf(spec)` — The type of the AgentInstance is the associated Agent.
- `outgoing->size()=0` — An AgentInstance is not involved in DirectedRelationships.
- `redefinedElment->size()=0` — An AgentIntstance is not redefined.

3.6.3 CallActivity

Description

A CallActivity is a ModeActivity that is used to model an operation call with actual parameters.

Associations

- op:Operation[1]
- params:ActualParameter[0..*]

Attributes

None.

Wellformedness Rules

- `op.params->forAll(p1 | params->isUnique(p2 | p2.spec=p1))` — For each parameter of the Operation there is an ActualParameter that has as spec this parameter.

3.6.4 Mode

Description

A Mode is a kind of ContainableElement, and therefore also a NamedElement by inheritance. It is used as main modeling element for modeling behavior. A Mode consists of at least two ModePseudostates that are the default entry and exit point and may own more ModePseudostates. It may have zero to many flow conditions, zero to many algebraic condition, and at most one invariant condition. In addition, FormalParameters can be used.

Associations

- params:SpecificationParameter[0..*]
- flow:Constraint[0..*]
- alge:Constraint[0..*]
- inv:Constraint[0..1]
- agent:Agent[0..1]
- pseudos:ModePseudostate[2..*]

Attributes

- isLeafMode:Boolean
- isTopLevelMode:Boolean

Wellformedness Rules

- `name->size()=1` — A Mode has a name.
- `rel->size()=0` — A Mode is not involved in Relationships.
- `containedElement->forAll(e | e.oclIsTypeOf(Mode))` — All contained elements are Modes.
- `isLeafMode: containedElement->size()=0` — A Mode without contained Submodes is a leaf mode.
- `isTopLevelMode: container->size()=0` — A Mode without container is called top-level Mode.
- `containedElement->forAll(e | e.agent=self.agent)` — Submodes belong to the same Agent as their Mode.
- `params->forAll(p1,p2 | p1.name<>p2.name)` — All parameters have distinct names.
- `flow->forAll(kind=flow)` — The flow constraints are of kind flow.
- `flow->forAll(f|` — A flow constraint constrains the Mode itself and a local or global variable of the Agent the Mode belongs to. The variable must be of type AnalogReal.

```
f.constrainedElement->includes(self) and
f.constrainedElement->includes(c |
   agent->locals->includes(c) or
   agent->ports->includes(p |
      p.oclIsTypeOf(VariablePort) and
         p.var=c) and
   c.oclIsKindOf(AnalogReal)))
```

- `alge->forAll(kind=alge)` — The algebraic constraints are of kind alge.
- `alge->forAll(f|` — An algebraic constraint constrains the Mode itself and a local or global variable of the Agent the Mode belongs to. The variable must be of kind Real.

```
f.constrainedElement->includes(self) and
f.constrainedElement->includes(c |
   agent->locals->includes(c) or
   agent->ports->includes(p |
      p.oclIsTypeOf(VariablePort) and
         p.var=c) and
   c.oclIsKindOf(Real)))
```

- `inv->forAll(kind=inv)` — The invariant is of kind inv.
- `inv->forAll(f| f.constrainedElement->includes(self))` — An invariant constraint constrains the Mode itself.
- `pseudos->one(p | p.kind=de)` — There is one ModePseudostate that is of kind default entry.
- `pseudos->one(p | p.kind=dx)` — There is one ModePseudostate that is of kind default exit.
- `isLeafMode=true implies (pseudos.size()=2)` — A leaf Mode has exactly two ModePseudostates: a default entry one and a default exit one.
- `isTopLevelMode=true implies pseudos->size()≥3 and pseudos->includes(p | p.kind=entry)` — A top-level Mode has at least three ModePseudostates: one default entry point, one default exit point, and one entry point that marks the initial point.
- `pseudos->forAll(p | p.outgoing->forAll(t | t.trigger->forAll(tr |` — The triggers of all transitions of a Mode must refer to a signal supported by a SignalPort of the associated Agent.

```
agent.ports->includes(p |
   p.oclIsTypeOf(SignalPort) and
   p.sig=tr.sig))))
```

- `pseudos->forAll(p | p.outgoing->forAll(t | t.effect->select(e |` — The SendActivities of all transitions of a Mode must refer to a signal supported by a SignalPort of the associated Agent.

```
e.oclIsTypeOf(SendActivity))->
   forAll(s |
      agent.ports->includes(p |
         p.oclIsTypeOf(SignalPort)
         and
         p.sig=s.sig))))
```

- The CallActivities of all transitions of a Mode must refer to an operation supported by the associated Agent.

```
pseudos->forAll(p |
  p.outgoing->forAll(t |
    t.effect->select(e |
      e.oclIsTypeOf(CallActivity))->
        forAll(c |
          agent.ops->includes(o |
            o=c.op))))
```

- The UpdateActivities of all transitions of a Mode must refer to a variable supported by the associated Agent either as local or as global variable.

```
pseudos->forAll(p |
  p.outgoing->forAll(t |
    t.effect->select(e |
      e.oclIsTypeOf(UpdateActivity))->
        forAll(u |
          agent.vars->includes(v |
            v=u.var) or
          agent.ports->includes(p |
            p.oclIsTypeOf(VariablePort)
            and
            p.var=v.var))))
```

3.6.5 ModeActivity

Description

ModeActivity is a ModelElement that is used to describe the activities that are performed if a ModeTransition is taken. ModeActivity is abstract and must be used by its concrete specializations.

Associations

None.

Attributes

None.

Wellformedness Rules

None.

3.6.6 ModePseudostate

Description

A ModePseudostate is a NamedElement that is used to model entry, exit, and junction points of Modes. ModePseudostates are sources and targets of ModeTransitions.

Associations

- outgoing:ModeTransition[0..*]

Attributes

- kind:ModePseudostateKind[1]

Wellformedness Rules

- All relationships are ModeTransitions.

```
rel->forAll(oclIsTypeOf(ModeTransition))
```

- Two transitions with the same target must have either distinct triggers or the same trigger but distinct guards.

```
outgoing->forAll(t1,t2 |
  (t1.target=t2.target) implies
    ((t1.trigger.signal<>
      t2.trigger.signal) or
    ((t1.trigger.signal=
      t2.trigger.signal) and
      (t1.guard.expr<>
        t2.guard.expr))))
```

- Only (default) exit points can own transition.

```
outgoing->size()≥1 implies
  ((kind=dx) or
  (kind=exit))
```

- Exit points must have outgoing transitions.

```
((kind=exit) implies
  (outgoing->size()≥1)
```

3.6.7 ModePseudostateKind

Description

ModePseudostateKind describes the different kinds of ModePseudostates. There are default entry points *de*, default exit points *dx*, entry points, and exit points.

Associations

None.

Attributes

- de
- dx
- entry
- exit

Wellformedness Rules

None.

3.6.8 ModeTransition

Description

ModeTransition is a kind of DirectedRelationship that is used to connect a target and a source ModePseudostate. It has at most one guard, at most one trigger and zero to many effects that either send signals, call operations, or update variables.

Associations

- target:ModePseueostate[1]
- trigger:SignalTrigger[0..1]
- guard:Constraint[0..1]
- effect:ModeActivity[0..*]

Attributes

None.

Wellformedness Rules

• `guard->size()=1 implies guard.kind=guard`	A Constraint that describes a guard must be of kind *guard*.
• `guard->constrainedElement=self`	The guard condition constrains the ModeTransition.
• `(target.kind=de) or (target.kind=entry)`	A target of a ModeTransition is a (default) entry point.

3.6.9 SendActivity

Description

A SendActivity is a kind of ModeActivity. It assigns actual parameters to a signal.

Associations

- sig:Signal[1]
- put:ActualParameter[0..*]

Attributes

None.

• `sig.params->forAll (p1 | put.spec=p1)`	For each parameter of the associated signal there exists an actual parameter.

3.6.10 SignalTrigger

Description

SignalTrigger is a ModelElement that is used to describe that a specific Signal triggers a ModeTransition. The signal has actual parameters.

Associations

- sig:Signal[1]
- get:ActualParameter[0..*]

Attributes

None.

Wellformedness Rules

• `sig.params->forAll (p1 | get.spec=p1)`	For each parameter of the associated signal there exists an actual parameter.

3.6.11 UpdateActivity

Description

An UpdateActivity is a kind of ModeActivity and that associates a Variable with a ValueSpecification.

Associations

- var:Variable[1]
- value:ValueSpecification[1..*]

Attributes

None.

Wellformedness Rules

- `value.spec=var` — The variable of the UpdateActivity is the specification for the associated ValueSpecification.

Chapter 4

HybridUML Profile

In this part, we define the HybridUML profile, based on the UML Superstructure Specification [OMG05a] as the reference metamodel. We define stereotypes for modeling datatypes not provided by UML 2.0, for time, for specific expressions and constraints, for communication based on shared variables and signals, for agents, and modes.

In a HybridUML model, these stereotypes as well as the datatypes mentioned in section 4.1 should be used. Other modeling elements do not have semantics with respect to HybridUML.

4.1 Data

In HybridUML, only typed variables are used. UML 2.0 defines Integer, UnlimitedNatural, Boolean, and String as datatypes. Integer, Boolean, and String are needed out of these. Instances of Integer-typed variables are values out of $\mathbb{N}$ while instances of Boolean-typed variables are values out of $\{true, false\}$. Instances of String-typed variables are finite sequences of characters from a finite alphabet, e.g. Unicode.

Further, real-valued variables are needed as these are not covered by UML 2.0. We extend PrimitiveType for getting a Real datatype in HybridUML. To distinguish between real-valued variables that can be changed exclusively discretely and real-valued variables whose value may also vary while time is passing, also AnalogReal variables are introduced as a further specialization of Real.

For better readability, StructuredDataType is introduced in HybridUML as an extension of DataType. Variables which represent related information are merged in a structure that can be used as datatype afterwards.

4.1.1 AnalogReal

Description AnalogReal is a specialization of Real (see Fig. 4.2) that defines an analog real number. AnalogReal variables are used in Modes to describe the time-continuous behavior of analog variables in flow conditions, algebraic expressions, and invariants.
Attributes No additional attributes.
Associations No additional associations.
Constraints No additional constraints.
Semantics AnalogReal instances are values of $\mathbb{R}$. The value is changing over time according to a given RTExpression in the context of a Mode. See Mode for further details.
Notation

Graphical Notation AnalogReal will appear as the type of attributes (see Fig. 4.1). The corresponding RTExpression that describes the behavior of an AnalogReal-typed variable is given in Modes.

Textual Notation

4.1.2 Real

Description Real is an extension of Classes::Kernel::PrimitiveType (see Fig. 4.2) that defines a real-valued number.
Attributes No additional attributes.
Associations No additional associations.

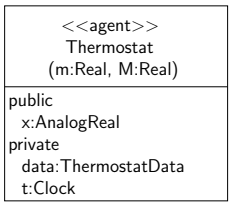

Figure 4.1: Usage of AnalogReal Variables

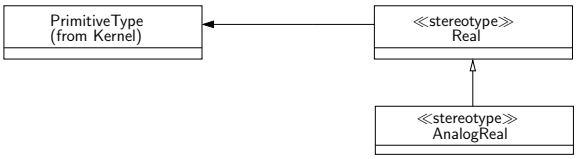

Figure 4.2: Stereotype for Real Numbers

Constraints No additional constraints.
Semantics Real instances are values of $\mathbb{R}$.
Notation

Graphical Notation Real appears as the type of attributes (see Fig. 4.1 and Fig. 4.4).

Textual Notation

4.1.3 StructuredDataType

Description StructuredDataType is an extension of Classes::Kernel::DataType (see Fig. 4.3) that defines a datatype consisting of different variables, i.e. a structure or - object-oriented - a class without methods.

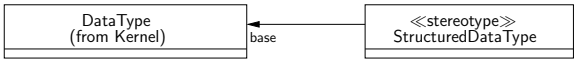

Figure 4.3: Stereotype for StructuredDataType

Attributes No additional attributes.
Associations No additional associations.
Constraints

- There are only attributes, no operations, i.e.
  `base.ownedOperation->size = 0`
- All attributes are typed. i.e.
  `base.ownedAttribute->forAll(type->size = 1)`
- All attributes are typed by DataType, i.e. PrimitiveType, Enumeration, or StructuredDataType:
  `base.ownedAttribute->forAll(type->forAll(oclIsKindOf(DataType)))`
- The type of an attribute is not UnlimitedNatural, i.e.
  `not base.ownedAttribute->forAll(type->exists (oclIsTypeOf(UnlimitedNatural)))`

Semantics StructuredDataType groups variables to a structure for simplifying usage of variables. Semantics of variables of the structure are according to the types of the variables.
Notation

Graphical Notation In a HybridUML Model, StructuredDataType is used as a stereotype. Elements with this stereotype will have a name (see Fig. 4.4) and are grouped in a class diagram named *package StructuredDataTypes*.

In diagrams of a HybridUML model, these names will be given as the datatype of the corresponding variable of an Agent (see Fig. 4.1).

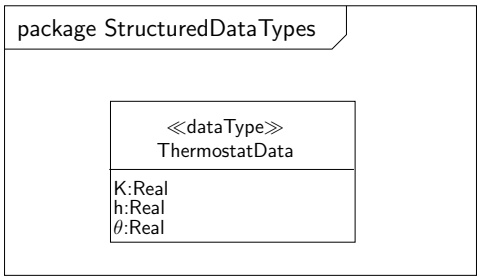

Figure 4.4: Notation of StructuredDataType in Definition

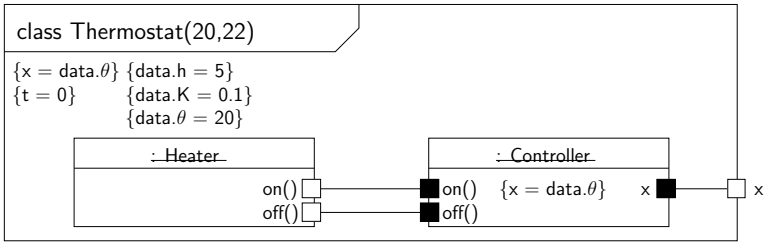

Figure 4.5: Usage of StructuredDataTypes and Constraints

Parts of the structure are referenced in dot-notation, i.e. name of the variable whose type is a StructuredDataType, followed by a dot, followed by the name of the part (see Fig. 4.5).

Textual Notation

4.2 Expressions and Constraints

For describing variables whose values evolve continuously over time, different expressions are needed, i.e. differential expressions and algebraic expressions. Furthermore, invariant expressions are used for defining state invariants. We therefore introduce RTExpression as an abstract extension of Expression. DifferentialExpression, AlgebraicExpression, and InvariantExpression are the concrete subtypes used in modeling.

Often, Expressions are used in combination with Constraints to attach them to a model element, e.g. invariants in states. We therefore introduce RTConstraint as a Constraint that always is used in combination with RTExpression.

4.2.1 AlgebraicExpression

Description AlgebraicExpression is a specialization of RTExpression (see Fig. 4.8). It describes non-differential terms that can be dependent on variables that change over time. AlgebraicExpressions are evaluated continuously to assign a value to an AnalogReal at every moment in time.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- Attribute *symbol* contains the AlgebraicExpression. It must give a mathematical, non-differential term. All variable names in *symbol* must correspond to variables in the model:
  Let V be the set of variables in the Mode owning the AlgebraicExpression and V_{num} the set of variables of type Real, AnalogReal, or Integer. Then $V_{num} \subseteq V$. If the set of variables in *symbol* is V_s, then $V_s \subseteq V_{num}$. Furthermore, the evaluated variable v must be of type AnalogReal, i.e. $v \in V_a$.

Semantics An AlgebraicExpression is a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$

Notation

Graphical Notation Mathematical term, non-differential, e.g. $x = f(y, z)$ where y and z may be time-continuous variables.

Textual Notation

4.2.2 DifferentialExpression

Description DifferentialExpression is a specialization of RTExpression (see Fig. 4.8). It describes differential terms dependent on time.

Attributes No additional attributes.
Associations No additional associations.
Constraints

- Attribute *symbol* contains the DifferentialExpression. It must give a differential equation dependent on time. All variable names in *symbol* must correspond to variables in the model:
  Let V be the set of variables in the Mode owning the DifferentialExpression and V_{num} the set of variables of type AnalogReal, Real, or Integer. Then $V_{num} \subseteq V$. Let V_s be the set of variables in *symbol*. Then $V_s \subseteq V_{num}$. Furthermore, the differentiated variable v must be of type AnalogReal, i.e. $v \in V_a$.

Semantics A DifferentialExpression is a differentiable function $f : \mathbb{R}^n \rightarrow \mathbb{R}$.
Notation

Graphical Notation Mathematical term, e.g. $\dot{x} = f(x, u)$, where $\dot{x}$ *is* $\frac{dx}{dt}$.

Textual Notation

4.2.3 InvariantExpression

Description InvariantExpression is a specialization of RTExpression (see Fig. 4.8). It is used to model invariants in Modes. **Attributes** No additional attributes.
Associations No additional associations.
Constraints

- Attribute *symbol* contains the InvariantExpression. It must give an invariant as logical expression. All free variable names in *symbol* must correspond to variables in the model:
  Let V be the set of variables in the Mode owning the InvariantExpression and V_s the set of free variables in *symbol*. Then $V_s \subseteq V$.

Semantics An InvariantExpression is an invariant given as logical expression.
Notation

Graphical Notation Logical expression, e.g. $x \leq c$.

Textual Notation

4.2.4 RTConstraint

Description RTConstraint is an extension of Classes::Kernel::Constraint. It holds an RTExpression that defines the constraint.

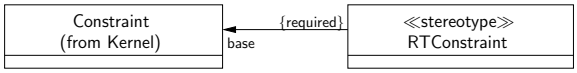

Figure 4.6: Stereotype for RTConstraint

Attributes No additional attributes.
Associations No additional associations
Constraints

- The given specification must be an RTExpression, i.e.
  `base.specification->forAll(oclIsKindOf(RTExpression))`

Semantics According to the included RTExpression. For details, see RTExpression, Agent, and Mode.
Notation

Graphical Notation RTConstraint is visualized in the same way as UML 2.0 constraints, i.e. an RTExpression term given in curly brackets (see Fig. 4.5). In Modes, brackets are used (see Fig. 4.7).

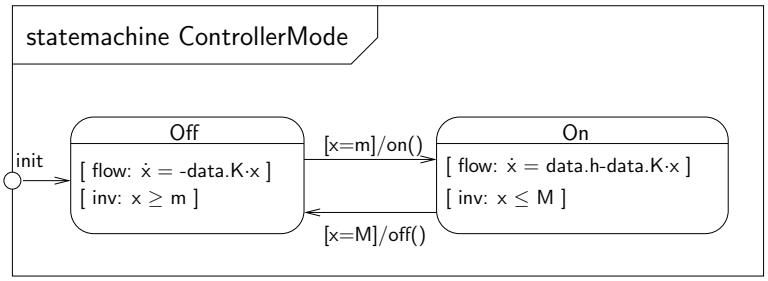

Figure 4.7: Usage of Constraints in Modes

Textual Notation

4.2.5 RTExpression

Description RTExpression is an extension of Classes::Kernel::Expression (see Fig. 4.8). It defines mathematical and logical terms that may be dependent on time. RTExpression is an abstract metaclass that cannot be instantiated.

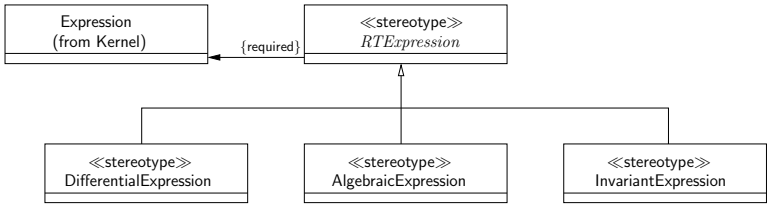

Figure 4.8: Stereotypes for RTExpressions

Attributes No additional attributes.
Associations No additional associations.
Constraints

- The real-time expression is given in attribute symbol as a string, just as in Expression. The expression must be mathematically or logically evaluable.

Semantics Semantics are given by the concrete subtypes.
Notation Notation is given by the concrete subtypes.

4.3 Time

For modeling time, we need clocks. This is done by using a variable of type AnalogReal that uses a differential equation for modeling the flow of time. Therefore we inherit from AnalogReal to get Clock. Beside clocks, timers are useful. A timer is set with a value and counts downwards until null. This is also modeled by inheriting from AnalogReal to achieve Timer.

We do not use the UML 2.0 time model as it has no formal semantics and is not powerful enough for our purposes.

4.3.1 Clock

Description A Clock is a specialization of AnalogReal (see Fig. 4.9). The flow of time is specified by a DifferentialEquation.

Attributes No additional attributes.
Associations No additional associations.

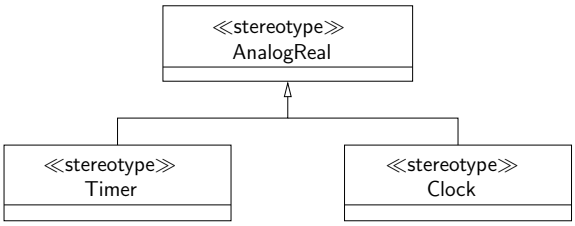

Figure 4.9: Stereotype for Clock

Constraints Let t be the value of a Clock instance. Then the following expression always holds:
$\dot{t} = 1$

Semantics Time flow is expressed by DifferentialExpression $\dot{t} = 1$ where t is the value of a Clock instance. As Clock instances are variables, their value can be set in assignments.

Notation

Graphical Notation Clocks are modeled as attribute of type Clock, e.g. x : Clock. As the differential equation is explicitly given, it is not added as a constraint following the variable (see Fig. 4.10).

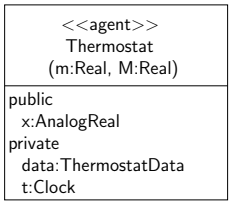

Figure 4.10: Usage of Clock Variables

Textual Notation

4.3.2 Timer

Description A Timer is a specialization of AnalogReal (see Fig. 4.9). The flow of time is specified by a DifferentialEquation.

Attributes No additional attributes.

Associations No additional associations.

Constraints Let t be the value of a Timer instance. Then the following expression always holds:
$\dot{t} = -1 \wedge t \geq 0$

Semantics Time flow is expressed by DifferentialExpression $\dot{t} = -1$ where t is the value of a Timer instance. t cannot undershoot 0. As each Timer instance is a variable, its value can be set in assignments.

Notation

Graphical Notation Timers are modeled as attribute of type Timer, e.g. x : Timer. As the differential equation is explicitly given, it is not added as a constraint following the variable. Therefore the notation is the same as for Clocks (see Fig. 4.10).

Textual Notation

4.4 Communication Structures

There are two different mechanisms Agents use for communication purposes. The first one is communication over shared variables, the second one is communication via signals. Both of these are implemented in HybridUML in the same way.

In UML 2.0, ports are used for defining communication structures. These can own required and provided interfaces. Ports are linked by connectors whose ends are connector ends. We therefore introduce AgentPort,

AgentInterface, AgentConnector and AgentConnectorEnd as abstract stereotypes. Concrete subclasses are SignalPort, VariablePort, SignalInterface, etc.

Communication via shared variables works by using VariableInterfaces that own exactly one variable each which is the shared variable. A VariablePort owns exactly one VariableInterface. This is either required or provided. A required VariableInterface means read access to the variable, a provided VariableInterface means read/write access. This is visualized by a white- respectively black-filled port. The lollipop notation for interfaces is not used. VariableConnectors are solid lines that link VariablePorts of the same type, i.e. they own VariableInterfaces that mirror the same global variable.

Communication via signals work in the same way. A SignalInterface owns exactly one signal of type RTSignal. A SignalPort owns exactly one required or provided SignalInterface. Here, required means the ability of receiving that signal while provided means the ability to send that signal. SignalConnectors link SignalPorts. There is exactly one sender involved in a connection.

We introduce RTSignal as a means for asynchronous communication between Agents. They are extensions of Signal whose parameters are either Integer, Real, AnalogReal, Boolean, String, or StructuredDataType. SignalEvents that are extensions of SignalTrigger carry RTSignals. They are used as triggers in Modes. Nevertheless, we prefer the term event as this is more common in statemachines.

4.4.1 AgentConnector

Description AgentConnector is an extension of CompositeStructures::InternalStructures::Connector (see Fig. 4.11). AgentConnector is an abstract class that cannot be instantiated. Instead, its concrete subclasses SignalConnector and VariableConnector are used.

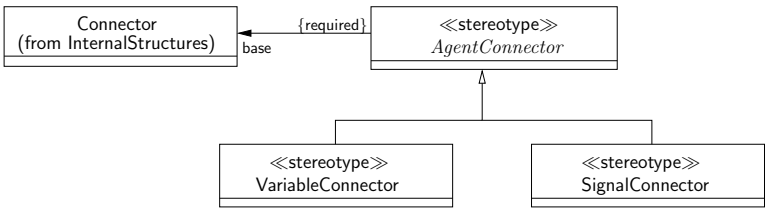

Figure 4.11: Stereotype for AgentConnector

Attributes No additional attributes.
Associations No additional associations.
Constraints

- All ConnectorEnds are AgentConnectorEnds, i.e.
  `base.end->forAll(oclIsTypeOf(AgentConnectorEnd))`

Semantics Semantics are given by the subtypes VariableConnector and SignalConnector.
Notation Notation is given by the subtypes VariableConnector and SignalConnector.

4.4.2 AgentConnectorEnd

Description AgentConnectorEnd is an extension of CompositeStructures::InternalStructures::ConnectorEnd (see Fig. 4.12). AgentConnectorEnds are the ends of AgentConnectors and always attached to AgentPorts. AgentConnector is an abstract class that cannot be instantiated. Instead, its subtypes SignalConnector and VariableConnector are used.

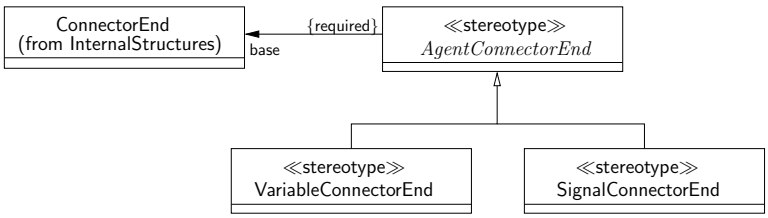

Figure 4.12: Stereotype for AgentConnectorEnd

Attributes No additional attributes.

Associations No additional associations.
Constraints

- All AgentConnectorEnds are attached to AgentPorts, i.e.
  `base.role->forAll(oclIsTypeOf(AgentPort))`

Semantics AgentConnectorEnds are the ends of AgentConnectors. They are always linked to AgentPorts.
Notation AgentConnectorEnd does not have a notation.

4.4.3 AgentInterface

Description AgentInterface in an extension of Classes::Interfaces::Interface (see Fig. 4.13). An AgentInterface is either a VariableInterface used for communication via shared variables or a SignalInterface used for communication via signals. AgentInterface is an abstract class that cannot be instantiated. Instead, its concrete subtypes SignalInterface and VariableInterface are used.

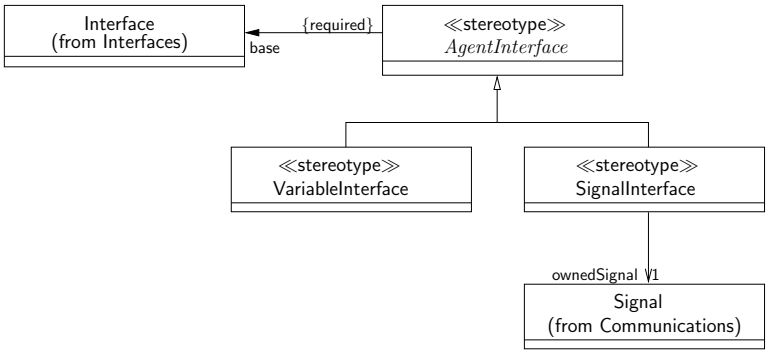

Figure 4.13: Stereotype for AgentInterface

Attributes No additional attributes.
Associations No additional associations.
Constraints

- AgentInterfaces do not own operations, i.e.
  `base.ownedOperation->size = 0`
- AgentInterfaces are not nested, i.e.
  `base.nestedInterfaces->size = 0`
- Each AgentInterface owns exactly one attribute, i.e.
  `base.ownedAttribute->size = 1`

Semantics AgentInterfaces are used by AgentPorts in HybridUML. Concrete semantics are given by the subtypes VariableInterface and SignalInterface.
Notation Notation is given by the subtypes VariableInterface and SignalInterface.

4.4.4 AgentPort

Description An AgentPort is an extension of CompositeStructures::Ports::Port (see Fig. 4.14). It owns only required and provided AgentInterfaces. AgentPorts are always both service ports and behavioral ports, i.e. they are interaction points of Agents and connected to a statemachine called Mode. Each AgentPort owns exactly one AgentInterface. AgentPort is an abstract class that cannot be instantiated. Instead, its concrete subtypes SignalPort and VariablePort are used.
Attributes No additional attributes.
Associations No additional associations.
Constraints

- AgentPorts are service ports, i.e. they specify the functionality of the Agent they belong to:
  `base.isService = true`

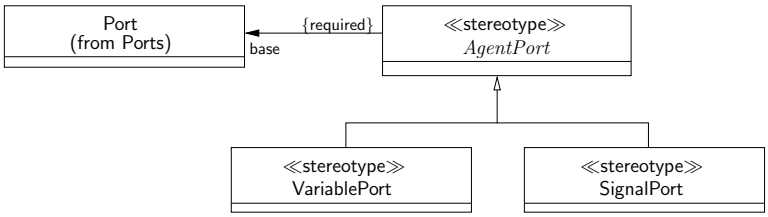

Figure 4.14: Stereotype for AgentPort

- AgentPorts are behavior ports, i.e. they are connected to the Mode of the Agent they belong to:
  `base.isBehavior = true`
- AgentPorts own only AgentInterfaces, i.e.
  `base.required->forAll(oclIsTypeOf(AgentInterface)) and`
  `base.provided->forAll(oclIsTypeOf(AgentInterface))`
- Each AgentPort owns exactly one required or provided AgentInterface, i.e.
  `base.exists(required) implies (base.required->size = 1 and base.provided->size = 0) and`
  `base.exists(provided) implies (base.required->size = 0 and base.provided->size = 1)`
- AgentPorts own exactly one AgentInterface at all, i.e.
  `if base->exists(required) then not(base->exists(provided)) endif and`
  `if base->exists(provided) then not(base->exists(required)) endif`

Semantics AgentPorts are used in connection with AgentInterfaces. They are access points for Agents. AgentPorts are connected by AgentConnectors. Concrete Semantics are given by VariablePort and SignalPort.

Notation Notation is given by the subtypes VariablePort and SignalPort.

4.4.5 RTSignal

Description RTSignal is an extension of CommonBehaviors::Communications::Signal (see Fig. 4.15). It defines an asynchronous message that may carry parameters.

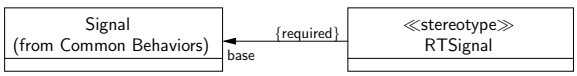

Figure 4.15: Stereotype for RTSignal

Attributes No additional attributes.
Associations No additional associations.
Constraints

- The parameters owned by VariableInterface are typed, i.e.
  `base.ownedAttribute->forAll(type->size = 1)`
- All attributes are typed by DataType, i.e. PrimitiveType, Enumeration, or StructuredDataType:
  `base.ownedAttribute->forAll(type->forAll(oclIsKindOf(DataType)))`
- The type of an attribute is not UnlimitedNatural, i.e.
  `not base.ownedAttribute->forAll(type->exists (oclIsTypeOf(UnlimitedNatural)))`

Semantics RTSignals have semantics in combination with SignalEvents in Modes. See SignalEvent and Mode for further details.

Notation

Graphical Notation Signals are given by their name, followed by the parameters in parenthesis (see Fig. 4.30).

Textual Notation

4.4.6 SignalConnector

Description SignalConnector is a specialization of AgentConnector (see Fig. 4.11). It links SignalPorts whose required and provided interfaces must be instances of the same SignalInterface, i.e. they send or receive the same RTSignal.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- All ConnectorEnds are SignalConnectorEnds, i.e.
  `base.end->forAll(oclIsTypeOf(SignalConnectorEnd))`
- All SignalConnectorEnds mirror the same SignalInterface, i.e. the same global variable:
  `base->forAll(e1, e2:SignalConnectorEnd | e1.role = e2.role)`
- There is exactly one sender involved in a connection, i.e.
  `base->exists(e1:SignalConnectorEnd | e1.role->exists(provided)`
  `and forAll(e2:SignalConnectorEnd | e1 <> e2 implies not e2.role->exists(provided)))`

Semantics A SignalConnector connects SignalPorts. All ends of the connector are used to send or receive the same RTSignal. This signal is defined by the SignalInterface owned by the SignalPorts.

Notation

Graphical Notation SignalConnectors are solid lines between ports (see Fig. 4.16). It is possible that one or both ends of the connector are attached to ports of Agent instances with a multiplicity. This is a shorthand notation that must be expanded.

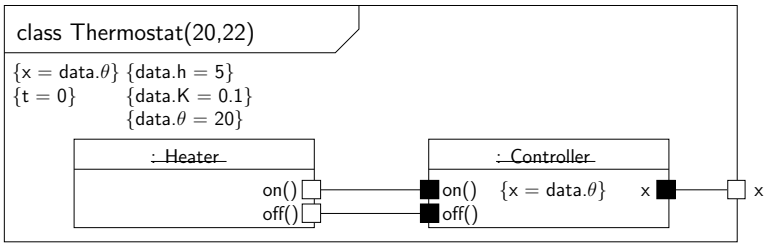

Figure 4.16: Usage of Connectors, Interfaces, and Ports

In the first case, there is a multiple Agent instance at exactly one end of the connector (see Fig. 4.17). This means that there are in fact as many Agent instances as the multiplicity defines. If the non-multiple end of the connector is a sending port, there is exactly one connector end for each of the multiple Agent instances (see Fig. 4.18).

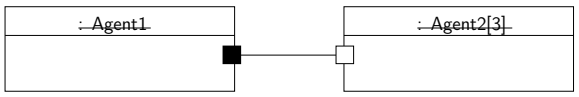

Figure 4.17: Shorthand Notation Case 1a)

If the multiple end is a sending port (see Fig. 4.19), there is exactly one connector from each sending port to the receiving port (see Fig. 4.19):
In the second case, there are multiple Agent instances at both ends of the connector (see Fig. 4.21. This means that there are in fact as many Agent instances as the multiplicities define on both ends. On one end, there must be a sending port (black-filled). Then, there is one connector from each sending port to each port of the receiving instances (see Fig. 4.22).

Textual Notation

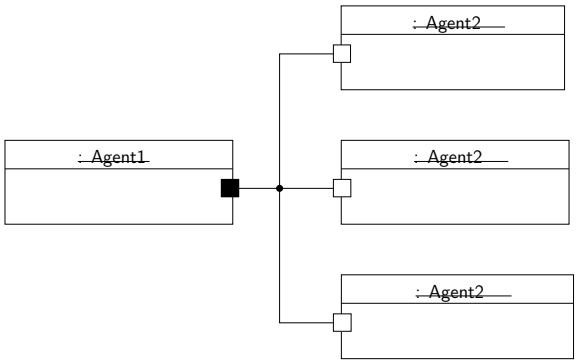

Figure 4.18: Shorthand Notation Expanded Case 1a)

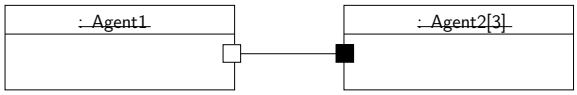

Figure 4.19: Shorthand Notation 1b)

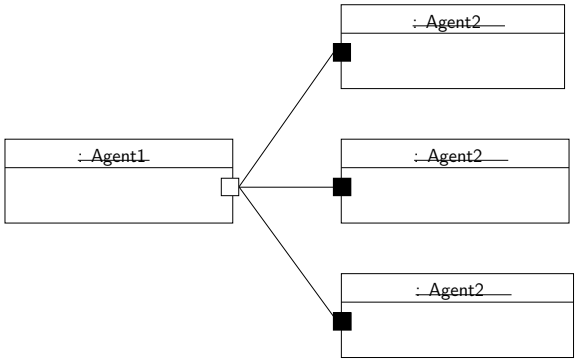

Figure 4.20: Shorthand Notation Expanded Case 1b)

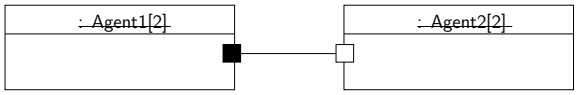

Figure 4.21: Shorthand Notation Case 2

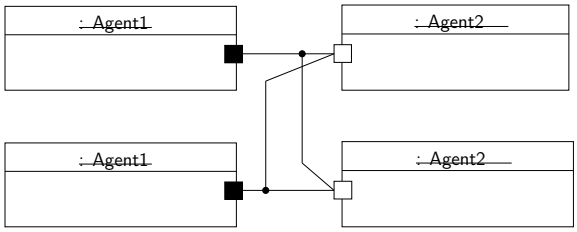

Figure 4.22: Shorthand Notation Expanded Case 2

4.4.7 SignalConnectorEnd

Description SignalConnectorEnd is a specialization of AgentConnectorEnd (see Fig. 4.12). It is an end of a SignalConnector. SignalConnectorEnds are always attached to SignalPorts.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- All SignalConnectorEnds are attached to SignalPorts, i.e.

`base.role->forAll(oclIsTypeOf(SignalPort))`

Semantics SignalConnectorEnds are the ends of SignalConnectors. They are always linked to SignalPorts.
Notation
SignalConnectorEnd does not have a notation.

4.4.8 SignalEvent

Description SignalEvent is an extension of SignalTrigger (see Fig. 4.23). It is used in correspondence with ModeTransitions.

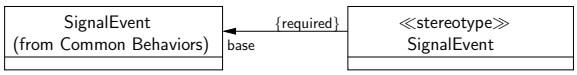

Figure 4.23: Stereotype for SignalEvent

Attributes No additional attributes.
Associations No additional associations.
Constraints

- The associated signal is an RTSignal, i.e.
  `base.signal.oclIsTypeOf(RTSignal)`

Semantics Semantics are given in ModeTransition and Mode.
Notation

Graphical Notation SignalEvents have notation in correspondence to their associated RTSignal, i.e. the name of the RTSignal with the parameters in parenthesis is used.

Textual Notation

4.4.9 SignalInterface

Description SignalInterface is a specialization of AgentInterface (see Fig. 4.13). A SignalInterface is associated to one RTSignal.
Attributes No additional attributes.
Associations

- ownedSignal:Signal[1]

Constraints

- The owned signal of the SignalInterface is an RTSignal, i.e.
  `self.ownedSignal.oclIsTypeOf(RTSignal)`
- SignalInterfaces do not own attributes, i.e.
  `base.ownedAttribute->size = 0`

Semantics SignalInterfaces are used by SignalPorts in HybridUML. They represent asynchronous messages send from one Agent to another one. Each interface owns an RTSignal for that purpose. Concrete semantics are given in SignalConnector.
Notation

Graphical Notation SignalInterfaces own exactly one RTSignal, its name is given as the name of the SignalInterface, respectively as the name of the SignalPort that owns the interface. SignalInterfaces are only visualized in combination with ports (see Fig. 4.16).

Textual Notation

4.4.10 SignalPort

Description A SignalPort is a specialization of AgentPort (see Fig. 4.14). It owns one required or provided SignalInterface.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- SignalPorts own only SignalInterfaces, i.e.
  `base.required->forAll(oclIsTypeOf(SignalInterface)) and`
  `base.provided->forAll(oclIsTypeOf(SignalInterface))`
- A required interface means sending a signal.
- A provided interface means receiving a signal.

Semantics SignalPorts are used in connection with SignalInterfaces. They are access points for Agents. SignalPorts are connected by SignalConnectors.

Notation

Graphical Notation In composite structure diagrams, SignalPorts are depicted like ports, i.e. a rectangle on the boundary of the owning classifier. Instead of visualizing the attached SignalInterfaces in lollipop-notation, a required SignalInterface is a white-filled rectangle and a provided SignalInterface is a black-filled rectangle (see Fig. 4.16). As every port is connected to a Mode, the state symbol that indicates behavior ports will be omitted.
In class diagrams, only the RTSignal owned by the SignalInterface of the port will be shown (see Fig. 4.27).

Textual Notation

4.4.11 VariableConnector

Description VariableConnector is a specialization of AgentConnector (see Fig. 4.11). VariableConnectors link VariablePorts whose required or provided VariableInterfaces must be instances of the same VariableInterface, i.e. they mirror the same global variable.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- All ConnectorEnds are VariableConnectorEnds, i.e.
  `base.end->forAll(oclIsTypeOf(VariableConnectorEnd))`
- All VariableConnectorEnds mirror the same VariableInterface, i.e. the same global variable:
  `base->forAll(e1, e2:VariableConnectorEnd | e1.role = e2.role)`

Semantics A VariableConnector connects VariablePorts. All ends of the connector mirror the same variable. This variable is defined by the VariableInterface owned by a VariablePort.

Notation

Graphical Notation AgentConnectors are solid lines between ports (see Fig. 4.16).
If the port belongs to an Agent instance with multiplicity, this is a shorthand notation with the following meaning: there is exactly one connector end for every instance involved.

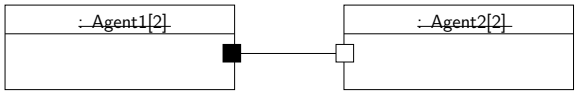

Figure 4.24: Shorthand Notation Case

Textual Notation

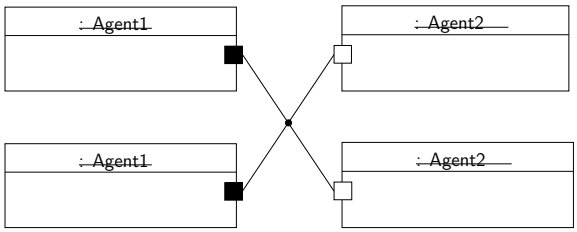

Figure 4.25: Shorthand Notation Expanded Case

4.4.12 VariableConnectorEnd

Description VariableConnectorEnd is a specialization of AgentConnectorEnd (see Fig. 4.12). VariableConnectorEnds are the ends of VariableConnectors. They are always attached to VariablePorts.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- All VariableConnectorEnds are attached to VariablePorts, i.e.
  `base.role->forAll(oclIsTypeOf(VariablePort))`

Semantics VariableConnectorEnds are the ends of VariableConnectors. They are always linked to VariablePorts.

Notation VariableConnectorEnd does not have a notation.

4.4.13 VariableInterface

Description VariableInterface is a specialization of AgentInterface (see Fig. 4.13). A VariableInterface is associated to one global variable that must be of type Integer, Real, AnalogReal, Boolean, String, or StructuredDataType.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- The property owned by VariableInterface is typed, i.e.
  `base.ownedAttribute->forAll(type->size = 1)`
- All attributes are typed by DataType, i.e. PrimitiveType, Enumeration, or StructuredDataType:
  `base.ownedAttribute->forAll(type->(oclIsKindOf(DataType)))`
- The type of an attribute is not UnlimitedNatural, i.e.
  `not base.ownedAttribute->forAll(type->exists (oclIsTypeOf(UnlimitedNatural)))`

Semantics VariableInterfaces are used by VariablePorts in HybridUML. They represent global variables. Each interface owns a property that mirrors the value of a global variable, i.e. the properties of connected interfaces must have the same value. Concrete semantics for this is given in VariableConnector.

Notation

Graphical Notation VariableInterfaces own exactly one property, its name is given as the name of the VariableInterface, respectively as the name of the VariablePort that owns the interface. VariableInterfaces are only visualized in combination with VariablePorts (see Fig. 4.16).

Textual Notation

4.4.14 VariablePort

Description A VariablePort is a specialization of AgentPort (see Fig. 4.14). It owns one required or provided VariableInterface.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- VariablePorts own only VariableInterfaces, i.e.
  `base.required->forAll(oclIsTypeOf(VariableInterface)) and`
  `base.provided->forAll(oclIsTypeOf(VariableInterface))`
- A required interface means read access, i.e.
  ```
  if (base->exists(required)) then
    post:  self.required.ownedAttribute = self.required.ownedAttribute@pre
  ```
- A provided interface means read/write access, i.e. normal behavior.

Semantics VariablePorts are used in connection with VariableInterfaces. They are access points for Agents. VariablePorts are connected by VariableConnectors.

Notation

Graphical Notation In composite structure diagrams, VariablePorts are depicted like ports, i.e. a rectangle on the boundary of the owning classifier. Instead of visualizing the attached VariableInterface in lollipop-notation, a required VariableInterface is a white-filled rectangle and a provided VariableInterface is a black-filled rectangle (see Fig. 4.16. As every port is connected to a Mode, the state symbol that indicates behavior ports will be omitted. The name of the global variable mirrored by the attached VariableInterface is given near the port.
In class diagrams, only the variable owned by the VariableInterface of the port will be shown (see Fig. 4.27).

Textual Notation

4.5 Agents

The main building block for modeling architectural structure within HybridUML is the *Agent*. Agents can be combined of other Agents, respectively Agent instances, by parallel composition, and can be grouped together enclosing them with a hiding operator. For precise interface descriptions we distinguish local and global variables and signals. HybridUML allows communication between concurrent agents via shared variables as well as via message passing to model multicasting of signals. The behavior of an Agent is described by a set of Modes, i.e. statecharts, and the set of allowed initial states might be restricted.

In HybridUML, Agents are extensions of classes. They consist of VariablePorts, SignalPorts, private variables, Modes, initState, and parameters.

VariablePorts are interaction points for communication via shared variables, whereas SignalPorts are interaction points for communication via asynchronous messages, i.e. signals. Beside the VariablePorts, there are also private variables only used by the Agent and its Mode(s). Modes are used for describing the behavior of an Agent. initState specifies the allowed initial values of variables in Agent instances. Parameters are used for better scalability. They specify constants that can be used in invariants and other expressions used in the Agent instance and its Mode(s).

4.5.1 Agent

Description An Agent is an extension of CompositeStructures::StructuredClasses::Class (see Fig. 4.26) that can own an internal structure (see Fig. 4.28). The internal structure consists of Agent instances. Agents communicate by AgentPorts and AgentConnectors, i.e. by shared variables (VariablePorts and VariableConnectors) or signals (SignalPorts and SignalConnectors).

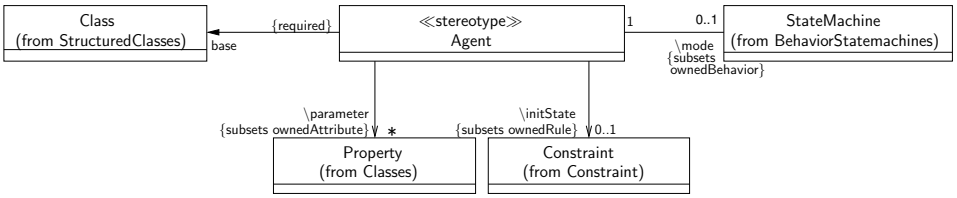

Figure 4.26: Stereotype for Agent

Attributes No additional attributes.
Associations

- \parameter:Property[*] {subsets ownedAttribute}
- \initState:Constraint[0..1] {subsets ownedRule}
- \mode:StateMachine[0..1] {subsets ownedBehavior}

Constraints

- Agents do not hold operations, i.e.
  `base.ownedOperation->size = 0`
- All attributes and parameters are typed, i.e.
  `base.ownedAttribute->forAll(type->size = 1)`
- All attributes are typed by DataType, i.e. PrimitiveType, Enumeration, or StructuredDataType:
  `base.ownedAttribute->forAll(type.oclIsKindOf(DataType))`
- The type of an attribute is not UnlimitedNatural, i.e.
  `not base.ownedAttribute->forAll(type->exists(oclIsTypeOf(UnlimitedNatural)))`
- All parameters are read-only, i.e.
  `self.parameter->forAll(isReadOnly = true)`
- All parts of the internal structure are Agent instances, i.e.
  `base.part->forAll(oclIsTypeOf(Agent))`

- All ports of the Agent are AgentPorts, i.e.
  `base.ownedPort->forAll(oclIsTypeOf(AgentPort))`
- All connectors are AgentConnectors, i.e.
  `base.ownedConnector->forAll(oclIsTypeOf(AgentConnector))`
- The behavior of an Agent is described by a StateMachine called Mode, i.e.
  `self.mode->forAll(oclIsTypeOf(Mode))`
- There is no behavior besides Modes, i.e.
  `base.ownedBehavior->size = 0`
- If the Agent has an internal structure, there is no Mode, i.e.
  `base->exists(part) implies self.mode->size = 0`
- If the Agent has no internal structure, there is exactly one Mode, i.e.
  `not base->exists(part) implies self.mode->size = 1`
- The Mode of an Agent has exactly the same attributes as the Agent, i.e.
```
self.mode->size = 1 implies
  self.mode->forAll(ownedAttribute->forAll (a1 | base.ownedAttribute->exists
                                              (a2 | a1 = a2)))
```
- All parameters of the Mode of an Agent are parameters of the Agent itself, i.e.
```
self.mode->size = 1 implies
  self.mode->forAll(parameter->forAll (p1 | self.parameter->exists (p2 | p1 = p2)))
```
- Each Agent that owns a Mode is a Thread, i.e.
  `self.mode->size = 1 implies base.isActive = true`
- initState is an InvariantExpression, i.e.
  `self.initState->forAll(oclIsTypeOf(InvariantExpression))`
- The expression given in initState must correspond to the variables of the Agent:
  Let V_i be the set of variables in initState and V the set of variables of Agent A. Then $V_i \subseteq V$.

Semantics
Notation

Graphical Notation Agents are depicted like UML classes with internal structure. In a class diagram, the internal structure is visualized as aggregated classes (see Fig. 4.27). The parameter list of each Agent is given behind its name in parenthesis in the first compartment of the class symbol.

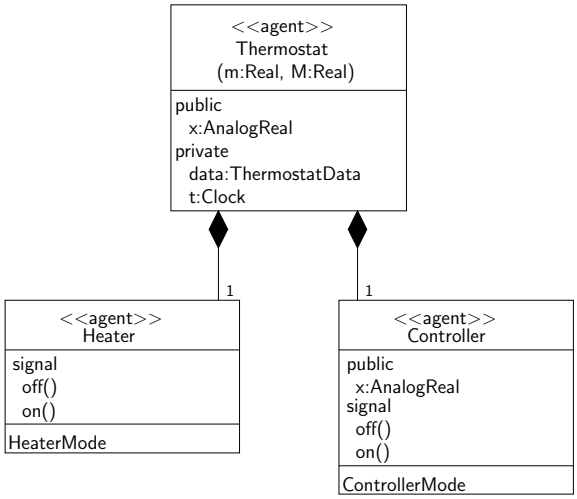

Figure 4.27: Usage of Agents in Class Diagram

VariablePorts and their included VariableInterfaces and variables are given as attributes in the second compartment of the class symbol. In the class diagram, only the name and type of the included variable is given.
The same holds for Signal ports and their included SignalInterfaces and signals. They are also given as attributes in the second compartment of the class symbol. In the class diagram, only the name and parameters of the included signal is given.
Attributes of the Agent, i.e. private variables, are also listed in the second compartment of the class symbol. They are listed with name and type.
To better distinguish between global variables, private variables, and signals, the second compartment is subdivided in maximal three parts named *public*, *private*, and *signal*. The corresponding variables and signals are listed beneath the keyword.
Optionally, in the third compartment of the class symbol the Mode of the Agent is given. This is the name of the Mode followed by concrete parameters listed inside parentheses. A parameter of a Mode may also be a parameter of the Agent, i.e. the concrete value is given in an Agent instance.

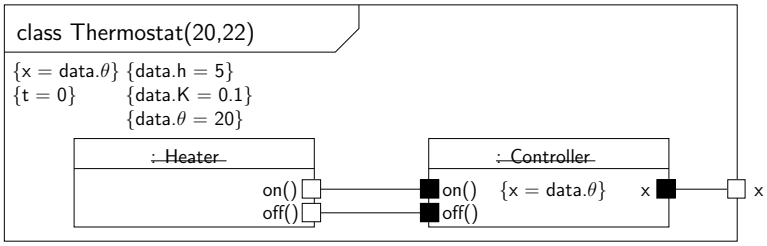

Figure 4.28: Usage of Agents in Composite Structure Diagram

The internal structure of composite Agents is shown in a composite structure diagram. The name of the Agent is given in the upper left corner with the keyword *class* before it. After that, the concrete parameters of the composite Agent follow. The internal structure is given in the main compartment.
Agent instances are visualized as objects in composite structure diagrams (see figure 4.28). Behind the objects' name and type the concrete parameters are given in parenthesis in the first compartment of the object symbol. If there are multiple instances of an Agent, the multiplicity is given in brackets after the Agent instances name. This is a shorthand notation as we do not want to show instances with the same parameters and initial values repeated. In fact, there are as many Agent instances as the multiplicity defines. In the second compartment, the respective initState is given as a constraint, i.e. in curly brackets. Here read/write access of global variables is shown as ports with required and provided interfaces (see Fig. 4.28). The same holds for posting and receiving signals.
The Mode of the Agent is given in a statechart diagram. The name of the Mode with the keyword *statemachine* before it is given in the upper left corner of the diagram (see Fig. 4.30).

Textual Notation

4.6 Modes

The behavior of a basic agent is described by a Statecharts formalism, in a fashion which became popular with Harel's Statemate statecharts []. In particular the use of hierarchy allows for improved structuring ....

Modes describe sets of states by grouping them according to constraints....

4.6.1 Mode

The behavior of Agents is described by hierarchical StateMachines called Mode. Each Mode contains exactly one Region, so there is no parallel behavior modeled inside. It is entered and left by control points, which are partitioned into entry and exit points. Every Mode has a default entry point *de* and a default exit point *dx*.

A Mode that is not contained by any other Mode is called top-level Mode. Each top-level Mode has a single non-default entry point called *init* point, and no non-default exit point. A Mode that is contained by another Mode is called Submode. A Mode without Submodes is called leaf Mode.

Top-level Modes are connected to an Agent. They use the variables defined in this Agent. Analog variables are updated according to constraints while the state machine is in a Mode. Discrete variables are only updated when a transition is taken. Modes can have parameters for better scalability. Preemption is modeled by using the default exit point *dx* as source of a group transition.

To achieve this behavior, we extend StateMachine to Mode and consequently Region to ModeRegion, State to ModeState, Pseudostate to ModePseudostate, PseudostateKind to ModePseudostateKind, and Transition to ModeTransition. Furthermore, we extend Activity to ModeTransitionActivity and its subtypes ModeUpdateActivity and ModeSendActivity for modeling the allowed activities while taking a transition, i.e. assigning values to variables and posting RTSignals.

Description Mode is an extension of StateMachines::BehaviorStatemachines::StateMachine. It describes the behavior of an Agent. Each Mode contains exactly one region, i.e. there is no parallel behavior inside a Mode. Modes can be built up hierarchically. As each Mode has exactly one default entry and one default exit point, there are at least two connection points. Modes can have parameters that are set by the Agent the Mode belongs to.

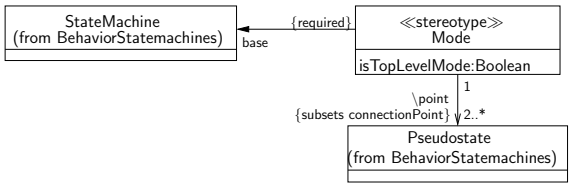

Figure 4.29: Stereotype for Mode

Attributes

- isTopLevelMode:Boolean

Associations

- point:Pseudostate [2..*] subsets connectionPoint

Constraints

- All parameters are typed. i.e.
  `base.formalParameter->forAll(type->size = 1)`
- All parameters are typed by DataType, i.e. PrimitiveType, Enumeration, or StructuredDataType:
  `base.formalParameter->forAll(type->forAll(oclIsKindOf(DataType)))`
- The type of a parameter is not UnlimitedNatural, i.e.
  `not base.formalParameter->forAll(type->exists (oclIsTypeOf(UnlimitedNatural)))`
- All parameters are constants, i.e.
  `base.parameter->forAll(isReadOnly = true)`
- There is no return result, i.e.
  `base.returnResult->size = 0`
- There is no specification beside the Mode itself, i.e.
  `base.specification->size = 0`
- Modes will not be redefined, i.e.
  `base.redefinedBehavior->size = 0`
- There are no operations, i.e.
  `base.ownedOperation->size = 0`
- Each Mode has exactly one region, i.e.
  `base.region->size = 1`
- All Pseudostates are ModePseudostates, i.e.
  `self.point->forAll(oclIsTypeOf(ModePseudostate))`
- Each Mode has exactly one default entry and exit point, i.e.
  `self.point->select(v | v.kind = defaultEntry)->size = 1 and`
  `self.point->select(v | v.kind = defaultExit)->size = 1`
- Each top-level mode has exactly one non-default entry point and no non-default exit point, i.e.

```
self.isTopLevelMode=true implies
  self.point->select(p | p.kind = entryPoint)->size = 1 and
  not(self.point->exists(p | p.kind = exitPoint))
```

- Each Mode has a *default transition* from *de* to *dx*, i.e.
  `base.region->forAll(transitions->exists`
  `(t | t.target.kind = defaultExit and t.source.kind = defaultEntry))`
- We do not redefine Modes, i.e. it is not possible to inherit from a Mode:
  `base.redefinedClassifier->size = 0`

Semantic Domain

.

Notation

Graphical Notation Modes are visualized the same way as UML 2.0 StateMachines (see Fig. 4.30). The identity transition from *de* to *dx* is not visualized explicitly as every Mode has it. Parameters are given behind the name of the Mode in parenthesis. The invariant is marked *inv*, the flow conditions with *flow* and the algebraic expressions with *alge*. As both are constraints, they are given in brackets.

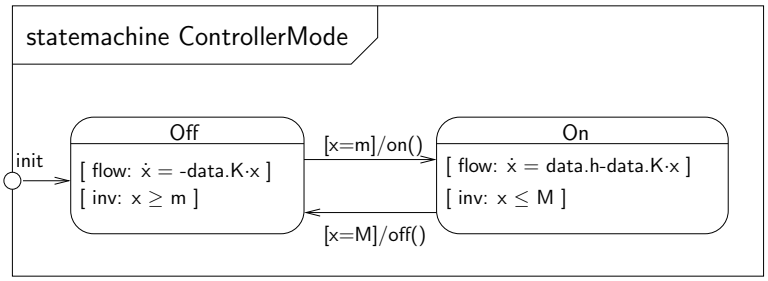

Figure 4.30: Notation for Mode

Textual Notation

4.6.2 ModePseudostate

Description ModePseudostate is an extension of StateMachines::BehaviorStatemachines::Pseudostate. ModePseudostates are used as entry and exit points for Modes. Further, they connect multiple transitions into more complex ones.

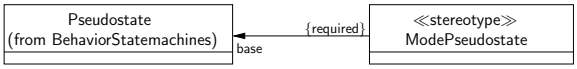

Figure 4.31: Stereotype for ModePseudostate

Attributes No additional attributes.
Associations No additional associations.
Constraints

- Each ModePseudostate is of kind ModePseudostateKind, i.e.
  `base.kind.oclIsTypeOf(ModePseudostateKind)`
- A ModePseudostate is either entryPoint, exitPoint, defaultEntry, defaultExit, or junction.
  `base.kind = (entryPoint or exitPoint or defaultEntry or defaultExit or junction)`

Semantics Semantics are given in Mode.
Notation

Graphical Notation entryPoints are depicted as small circles on the border of a Mode (see Fig. 4.30) with an optional name attached to it.
exitPoints are depicted as small solid black-filled circles on the border of a Mode with an optional name attached to it.
defaultEntry points are not depicted explicitly as every Mode has exactly one; transitions to the defaultEntry point end at the boundary of the Mode. defaultExit points are not depicted explicitly as every Mode has exactly one; transitions starting at the defaultExit point start at the boundary of the Mode.
junction points are depicted as small black-filled circles inside Regions.

Textual Notation

4.6.3 ModePseudostateKind

Description ModePseudostateKind is an extension of StateMachines::BehaviorStatemachines::PseudostateKind. It extends ModePseudostateKind by the following literal values:

- defaultEntry
- defaultExit

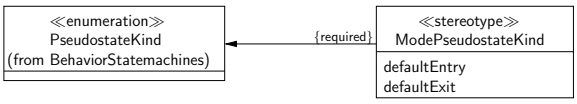

Figure 4.32: Stereotype for ModePseudostateKind

Attributes No additional attributes.
Associations No additional associations.

4.6.4 ModeRegion

Description ModeRegion is an extension of StateMachines::BehaviorStatemachines::Region. It contains ModeStates and ModeTransitions. Each Mode consists of one ModeRegion. Orthogonal ModeRegions are not allowed.

Figure 4.33: Stereotype for ModeRegion

Attributes No additional attributes.
Associations No additional associations.
Constraints

- All transitions are ModeTransitions, i.e.
  `base.transitions->forAll(oclIsTypeOf(ModeTransition))`
- All vertices are either ModePseudostates or ModeStates, i.e.
  `base.subvertex->forAll(v | v.oclIsTypeOf(ModePseudostate) or v.oclIsTypeOf(ModeState))`
- All ModePseudostates in a ModeRegion are junction points, i.e.
  `base.subvertex->select(v | v.oclIsTypeOf(ModePseudostate))->forAll(kind = junction)`

Semantics Semantics are given in Mode.
Notation As orthogonal regions are now allowed, there is no notation for region. They are just the space inside a Mode.

4.6.5 ModeState

Description ModeState is an extension of StateMachines::BehaviorStatemachines::State. Each Mode consists of ModeStates. ModeStates are always Submodes.

In addition to invariants, there are flows and algebraic expressions. A flow is a DifferentialExpression that describes how the values of analog variables changes over time. An AlgebraicExpression is a non-differential expression that includes variables that change over time. Both are evaluated while the Mode resists in this state.

Attributes No additional attributes.
Associations

- stateFlow: Expression[*]
- algExpression: Expression[*]

Constraints

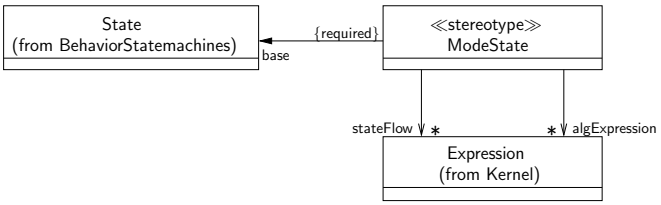

Figure 4.34: Stereotype for ModeState

- ModeState is never a composite state, i.e.
  `base.isComposite = false`
- ModeState is never orthogonal, i.e.
  `base.isOrthogonal = false`
- ModeState is never simple, i.e.
  `base.isSimple = false`
- ModeState is always a Submode, i.e. a submachine:
  `base.isSubmachineState = true`
- There is no do activity, i.e.
  `base.doActivity->size = 0`
- There is no entry activity, i.e.
  `base.entry->size = 0`
- There is no exit activity, i.e.
  `base.exit->size = 0`
- ModeStates have exactly one region, i.e.
  `base.region->size = 1`
- ModeStates are attached to a Mode, i.e.
  `base.submachineState->forAll(oclIsTypeOf(Mode)) and`
  `base.submachineState->size = 1`
- Triggers are not deferred, i.e.
  `base.deferrableTrigger->size = 0`
- There is at most one invariant of type RTConstraint, i.e.
  `base.stateInvariant->forAll(oclIsTypeOf(RTConstraint)) and`
  `base.stateInvariant->forAll(specification->forAll(oclIsTypeOf(InvariantExpression)))`
- Each stateFlow is a DifferentialExpression, i.e.
  `self.stateFlow->forAll(oclIsTypeOf(DifferentialExpression))`
- Each algExpression is an AlgebraicExpression, i.e.
  `self.algExpression->forAll(oclIsTypeOf(AlgebraicExpression))`

Semantics Semantics are given in Mode.
Notation ModeStates are always Submodes, therefore see Fig. 4.30).

4.6.6 ModeTransition

Description ModeTransition is an extension of StateMachines::BehaviorStatemachines::Transition. It connects a target and a source ModePseudoState. ModeTransitions are taken due to guard constraints or SignalEvents. A ModeTransition may have an associated effect that updates some variables or emits a SignalEvent.

A transition that originates at a default exit point of a Mode is called group transition. Group transitions are taken to interrupt the execution of that Mode. After that, the Mode must be entered again through the default entry point to resume the execution of the Mode.

Attributes No additional attributes.
Associations No additional associations.
Constraints

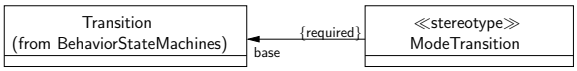

Figure 4.35: Stereotype for ModeTransition

- All triggers are SignalEvents, i.e.
  `base.trigger->forAll(oclIsTypeOf(SignalEvent))`
- All guards are InvariantExpressions, i.e.
  `base.guard->forAll(oclIsTypeOf(InvariantExpression))`
- All effects are ModeTransitionActivities, i.e.
  `base.effect->forAll (oclIsTypeOf(ModeTransitionActivity))`
- The source of a ModeTransition is a ModePseudostate, i.e.
  `base.source.oclIsTypeOf(ModePseudostate)`
- The target of a ModeTransition is a ModePseudostate, i.e.
  `base.target.oclIsTypeOf(ModePseudostate)`

Semantics Semantics are given in Mode.
Notation

Graphical Notation ModeTransition is depicted by an arrow with open arrowhead (see Fig. 4.30). The guard constraint is given in brackets. After that, the SignalEvent is given. The effect, i.e. a ModeSendActivity or a ModeUpdateActivity, is separated from the guard and the event by a slash.
The default transition from *de* to *dx* of each Mode is not visualized.

Textual Notation

4.6.7 ModeSendActivity

Description ModeSendActivity is a specialization of ModeTransitionActivity. It is an Activity that is used for sending RTSignals.
Attributes No additional attributes.
Associations No additional associations.
Constraints

- The String given as *body* of the Activity must be sending an RTSignal. Let S_s be the set of RTSignals in *body*, then $S_s \subseteq S$

Semantics The given RTSignal is posted.
Notation

Graphical Notation ModeSendActivity is visualized by the RTSignal (see Fig. 4.30). See RTSignal for further details.

Textual Notation

4.6.8 ModeTransitionActivity

Description ModeTransitionActivity is an extension of CommonBehaviors::BasicBehaviors::Activity (see Fig. 4.36). It is an abstract class that cannot be instantiated. Instead, its concrete subtypes ModeSendActivity and ModeUpdateActivity are used.
Attributes No additional attributes.
Associations No additional associations.
Constraints No additional constraints.
Semantics Semantics are given by the subtypes ModeSendActivity and ModeUpdateActivity.
Notation Notation is given by the subtypes ModeSendActivity and ModeUpdateActivity.

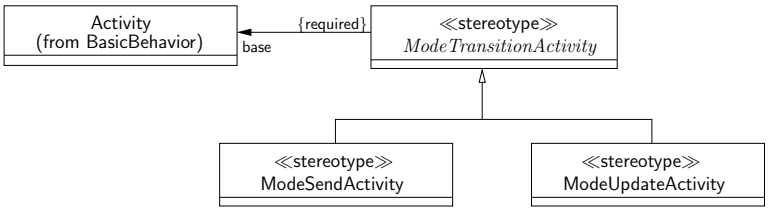

Figure 4.36: Stereotype for ModeTransitionActivity

4.6.9 ModeUpdateActivity

Description ModeUpdateActivity is a specialization of ModeTransitionActivity (see Fig. 4.36). It updates variable values of Agents.

Attributes No additional attributes.

Associations No additional associations.

Constraints

- The String given as *body* of the Activity must be an expression that updates variables V of an Agent: Let V_s be the set of variables in *body*, then $V_s \subseteq V$. Let Q_s be the set of valuations in *body*, then $Q_s \subseteq Q$ with Q valuations of Agent.

Semantics

Variables of the Agent that owns the Mode to that the ModeTransitionActivity belongs are updated.

Notation

Graphical Notation ModeUpdateActivity is an assignment that assigns a new value to a variable, e.g. $x = y + 10$

Textual Notation

Bibliography

[ADE+01] R. Alur, T. Dang, J. Esposito, R. Fierro, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling of embedded systems. *Lecture Notes in Computer Science*, 2211:14–31, 2001.

[ADE+03] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical hybrid modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, January 2003.

[AGLS01] Rajeev Alur, Radu Grosu, Insup Lee, and Oleg Sokolsky. Compositional refinement for hierarchical hybrid systems. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, volume 2034 of *Lecture Notes in Computer Science*, pages 33–48, 2001.

[BBB+99] Tom Bienmller, Jrgen Bohn, Henning Brinkmann, Udo Brockmeyer, Werner Damm, Hardi Hungar, and Peter Jansen. Verification of automotive control units. In *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 319–341, 1999.

[Hen96] Thomas A. Henzinger. The theory of Hybrid Automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[KMP00] Y. Kesten, Z. Manna, and A. Pnueli. Verification of clocked and hybrid systems. *Acta Informatica*, 36(11):836–912, 2000.

[OMG05a] Object Management Group. Unified Modeling Language: Superstructure, version 2.0. http://www.omg.org/docs/formal/05-07-04.pdf, July 2005.

[OMG05b] Object Management Group. Unified Modeling Language (UML) Specification: Infrastructure, version 2.0. http://www.omg.org/docs/ptc/04-10-14.pdf, July 2005.

[OMG06] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/docs/formal/06-01-01.pdf, January 2006.

[Rav95] A. P. Ravn. Design of embedded real-time computing systems. Technical Report ID-TR 1995-170, ID/DTU, Lyngby, Denmark, October 1995. dr. techn. dissertation.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language – Reference Manual*. Addison-Wesley, 1999.

[RRS03] Mauno Rönnkö, Anders P. Ravn, and Kaisa Sere. Hybrid Action Systems. *Theoretical Computer Science*, 290:937–973, January 2003.

[ZRH93] Chaochen Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 763 of *Lecture Notes in Computer Science*, pages 36–59. The Computer Society of the IEEE, 1993. Extended abstract.