Contents

1 KobrA 2 Metamodel

Version of: June 26, 2009

In the following, the Metamodel for the KobrA 2 Single Underlying Model (SUM) is described in detail, as well as the metamodels for the KobrA 2 views and the transformation rules between the SUM and the views. Furthermore, consistency rules are specified for the SUM and the views. As a side effect of the modeling effort, the OCL 2 and UML 2 are aligned, which is not yet done - the OCL 2 specification draft still has elements from UML 1.4 in it and leaves the alignment with UML 2 to be done. Principles of the KobrA 2 model are:

- to separate the core metamodel and the view metamodels.
- to reuse as much elements as possible from the UML2. Elements not needed for KobrA 2 are excluded with OCL constraints and new elements are introduced by specializing UML2 elements.
- to avoid redundancy whenever possible, e.g. "derived" elements. Some exceptions are made in order to have significantly smaller and more concise OCL expressions.

1.1 SUM - Package Dependencies

The elements of the views are derived from the SUM and, in contrast to the SUM, there might be a lot of redundancy in the views and between overlapping views.

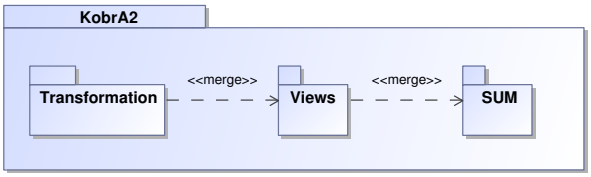

Figure 1.1: Contents of KobrA 2

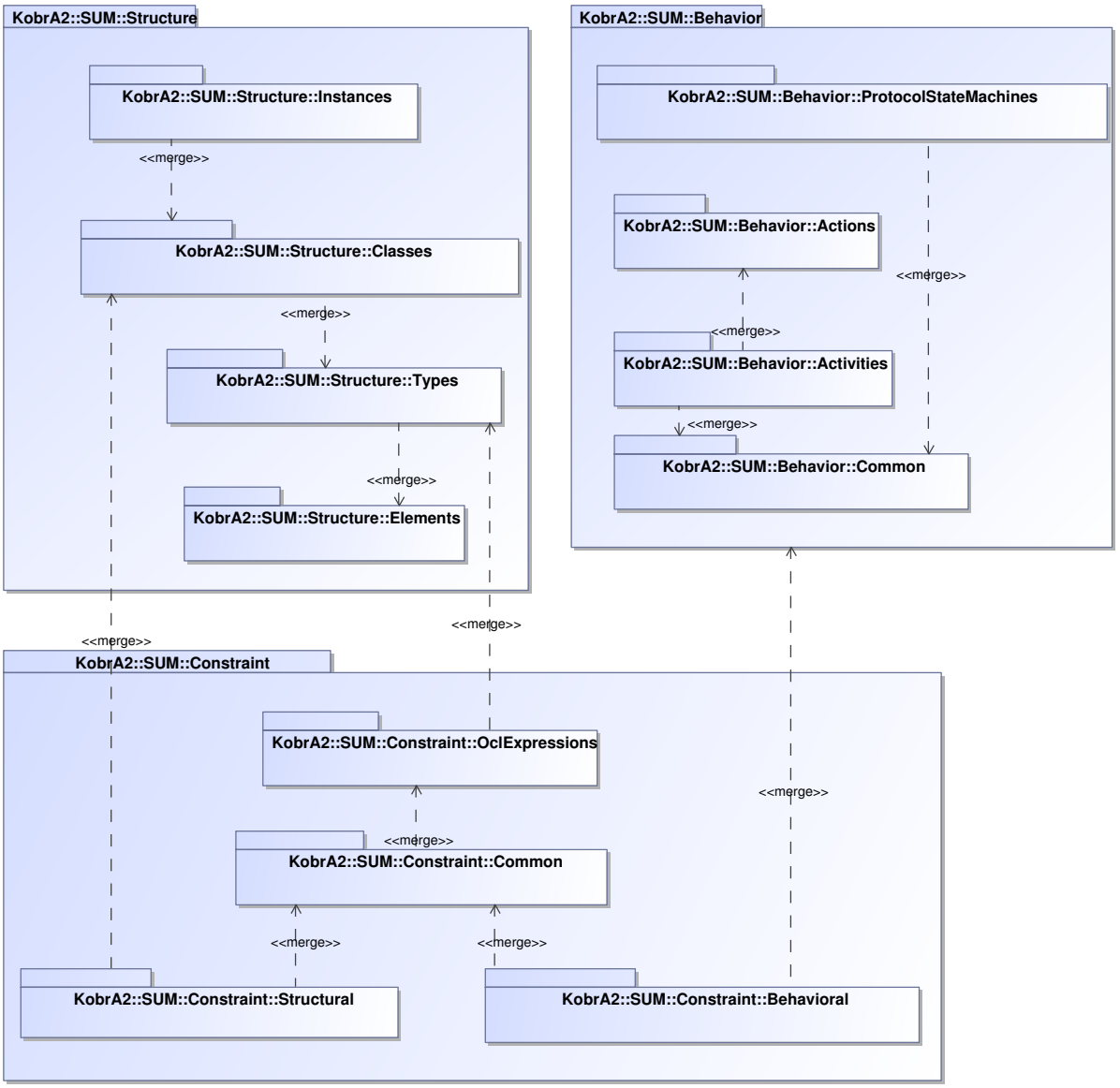

Figure 1.2: SUM Package Overview

To simplify our package structure, we flattened a few UML packages (e.g. the subpackages of Actions were collapsed into SUM::Actions), because in most cases it makes no sense for us to package only e.g. two elements together. On the other hand, the UML Kernel package from the UML superstructure contains a lot of elements which are not structured very well. Therefore, we divided it into several coherent subpackages.

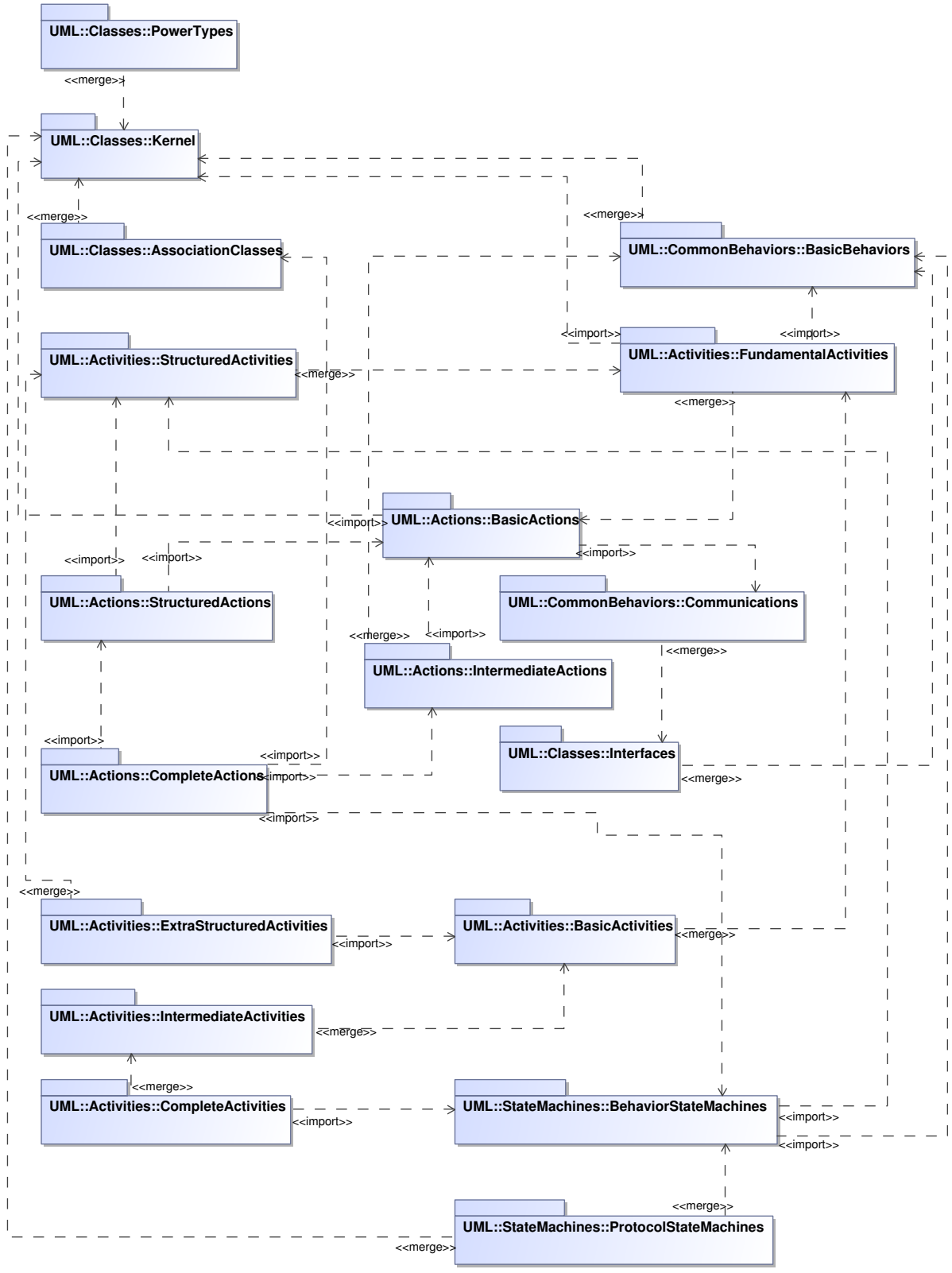

Figure 1.3: UML Package Overview

As the SUM contains far fewer elements than the UML, we tried to make the dependencies as simple as possible, in contrast to the rather complicated UML dependencies shown in fig 1.3 (and this figure only shows the packages (and their direct dependencies) of the UML that are reused in the SUM).

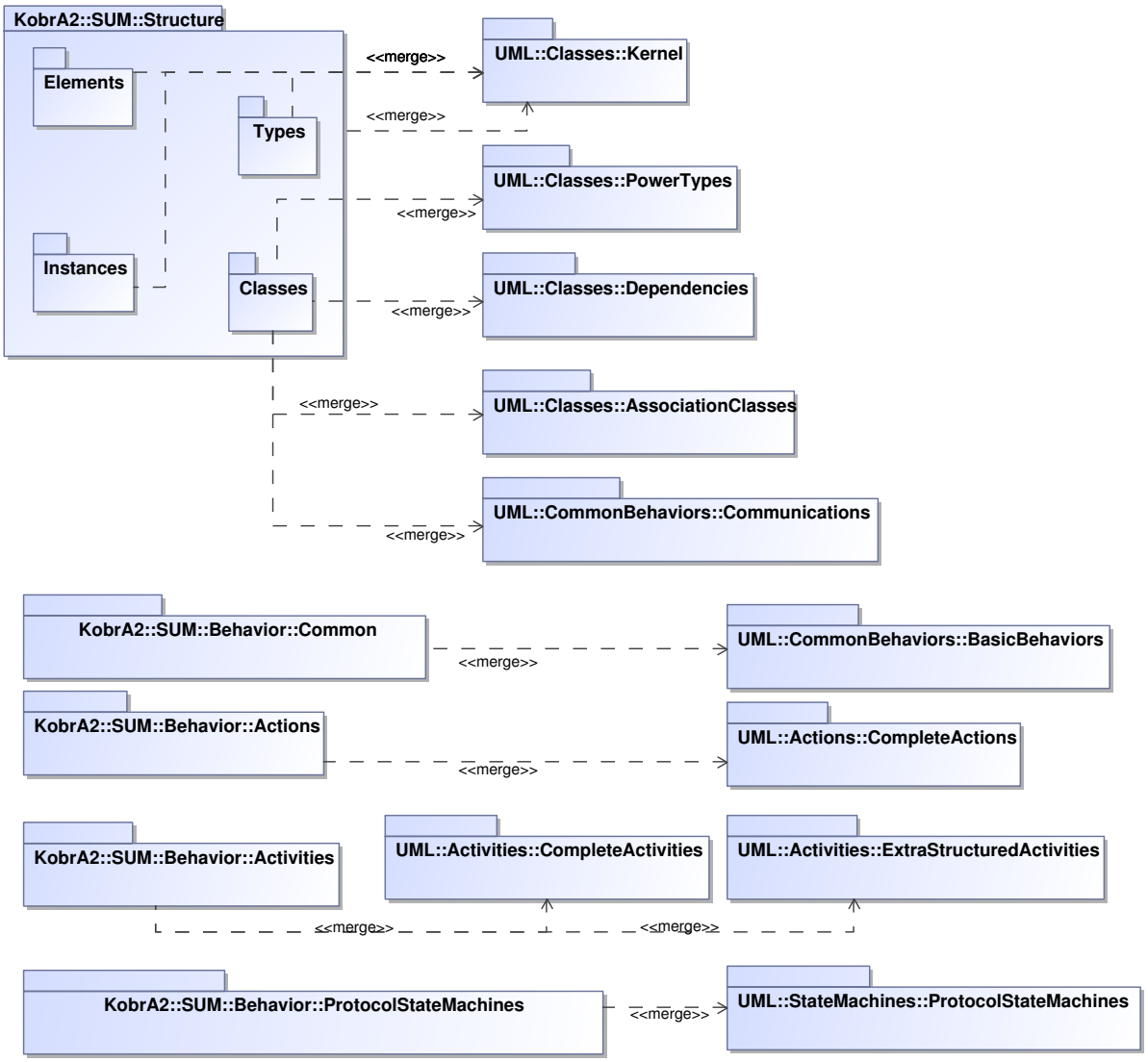

Figure 1.4: Dependencies between SUM and UML

The package overview shows exactly which packages from the UML are merged into which packages from the SUM. As a rule of thumb, a package in the SUM merges a package in the UML with the same name (if present), or a package in the UML which contains Elements with the same name. Thus all elements in a SUM package, for which an element in the UML exists with the same name, implicitly specialize the UML element. Elements in the SUM add new meta associations, new meta attributes, or new constraints.

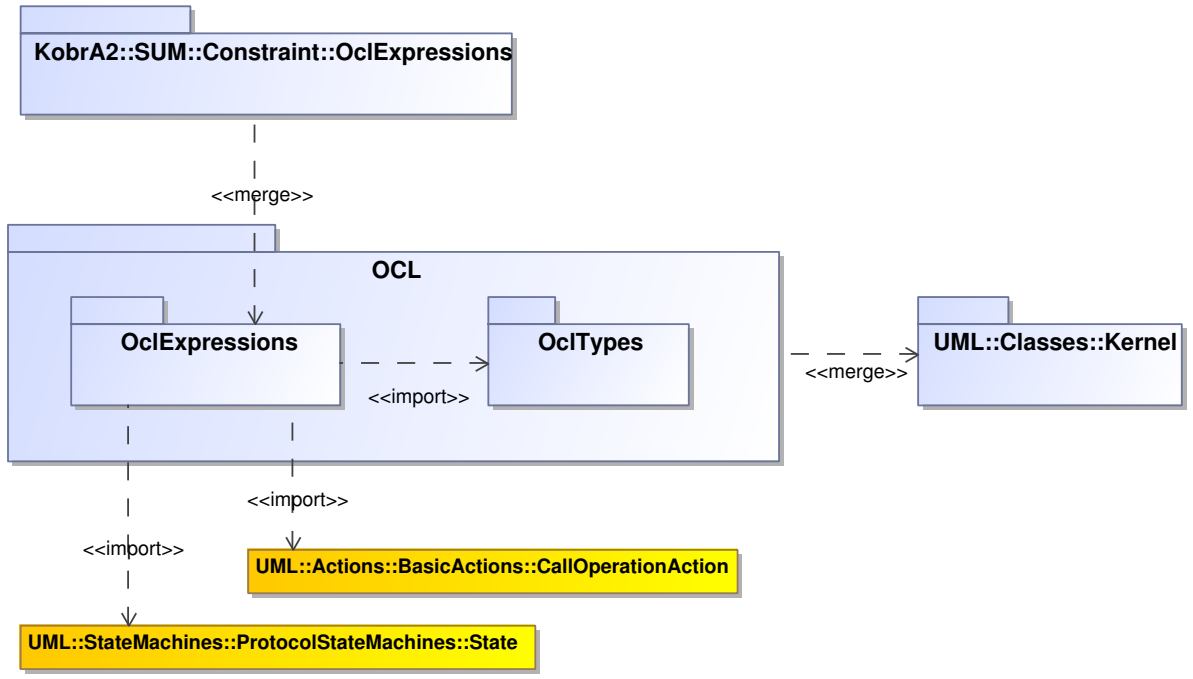

Figure 1.5: Package Dependencies between SUM constraints and UML

As in the current draft of the OCL specification UML and OCL are not aligned, another task of the KobrA 2 metamodel is to align OCL 2.0 with UML 2.0. As there is no package structure for OCL, we group the OCL metamodel in two packages: OCL::OclExpressions and OCL::OclTypes.

1.2 SUM - Structure

1.2.1 Element

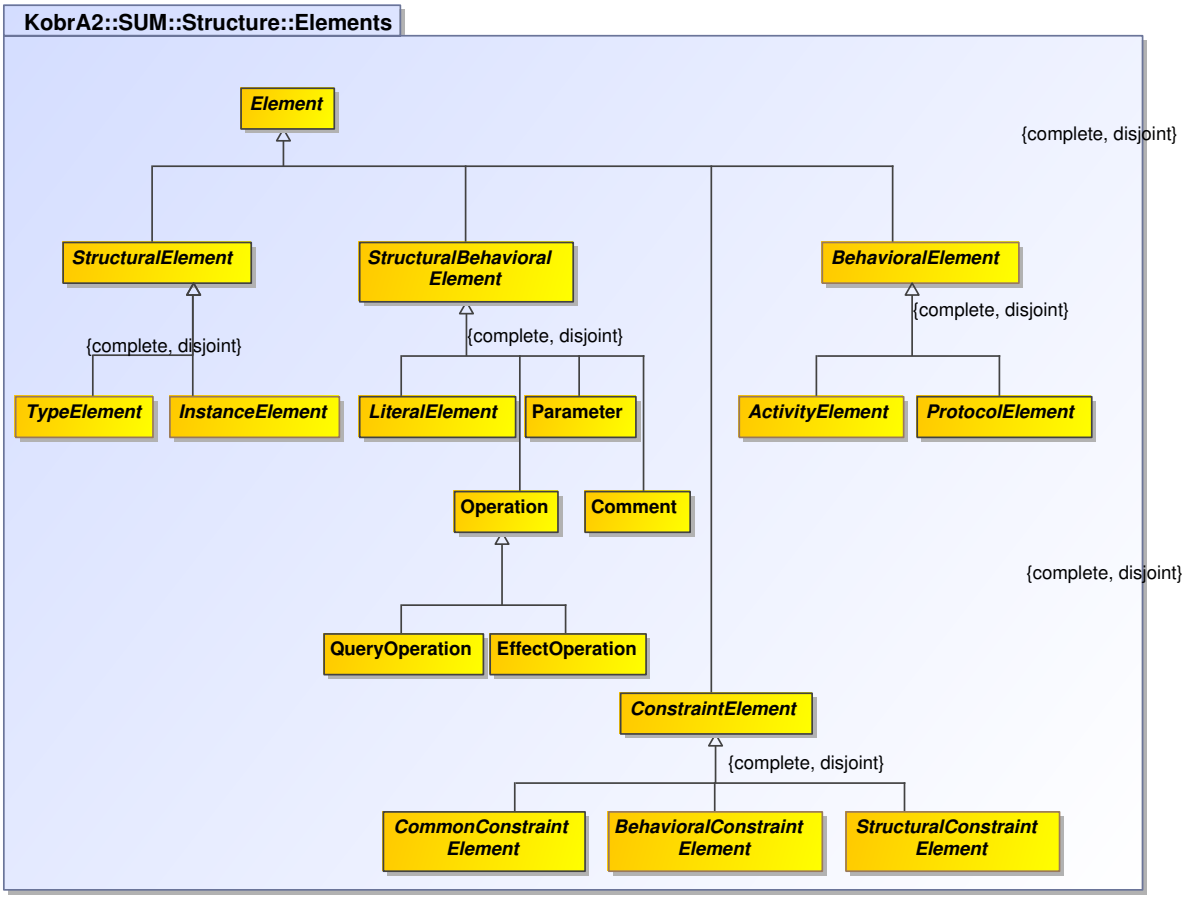

Figure 1.6: Element

As some elements appear in multiple constraints, we list them here and refer to them later. It is a design decision where to define the sets of reusable elements. As they are sets of elements and reused we defined them in the *Element* package.

Since views are defined as packages, all KobrA 2 elements must be either PackageableElements or owned by PackageableElements or owned by Elements that are in turn are owned by PackageableElements. PackageableElements are

- Elements that specialize *Classifier*: InvalidType, VoidType, TypeType, ElementType, AnyType, OrderSetType, SequenceType, BagType, SetType, TupleType, Boolean, UnlimitedNatural, Real, String, Enumeration, MessageType, ComponentClass, Property, Association (which also includes AssociationClass, Acquires and Nests), Generalization, GeneralizationSet, Usage, Activity, ProtocolStateMachine
- Elements that specialize InstanceSpecification: ComponentObject (which also includes Object and ObjectLink), Link (which also ObjectLink, AcquiresLink and NestsLink)
- Elements that inherit from ValueSpecification: ExpressionInOcl
- Elements that inherit from Constraint: Inv, Derived, Init, PropDef, OpDef, Post, Pre, Body

Elements owned by PackageableElements are

- Elements owned by any packageable element: Comment
- Elements owned by ComponentClass: Property, Generalization, QueryOperation, EffectOperation
- Elements owned by ComponentObject: Slot
- Elements owned by ExpressionInOcl: Variable, EnumLiteralExp, RealLiteralExp, UnlimitedNaturalExp, IntegerLiteralExp, StringLiteralExp, BooleanLiteralExp, NullLiteralExp, InvalidLiteralExp, CollectionLiteralExp, IfExp, VariableExp, TypeExp, IteratorExp, IterateExp, OperationCallExp, PropertyCallExp, LetExp, UnspecifiedValueExp, MessageExp
- Elements owned by Activity: CallSubActivityAction, CallOperationAction, ActionInOcl, InputPin, OutputPin, Activity, ActivityParameterNode, InitialNode, DecisionNode, MergeNode, ForkNode, JoinNode, FlowFinalNode, ActivityFinalNode, ControlFlow, ObjectFlow, ExceptionHandler, ActivityPartition
- Elements owned by ProtocolStateMachine: Region

Elements owned by Elements owned by PackageableElements are

- Elements owned by Operation and ActivityParameterNode: Parameter
- Elements owned by CollectionLiteralExp: CollectionRange, CollectionItem
- Elements owned by TupleLiteralExp: TupleLiteralPart
- Elements owned by Region: State (which includes FinalState), PseudoState, ProtocolTransition

The following constraints specify which elements are allowed in the SUM. We avoid oclIsTypeOf wherever possible (e.g. stereotypes are an exception, they are specified in the views with oclIsTypeOf) in order to facilitate later extensions of the metamodel. The SUM only stores concrete elements and we don't need to allow concrete elements which have no proper instance (i.e. oclIsTypeOf()). However, ObjectNode is in the list, because it is (in our opinion) in the UML erroneously specified as abstract. Even though we don't use Region, it is in our set of allowed Elements, as there must be at least one Region for every StateMachine.

Also, it is not necessary to have abstract UML elements with no concrete syntax in the above list, as they will never appear in views, and, if the SUM is only manipulated through those views, never appear in the SUM.

For clarity, one could fully qualify every Element in the sets below.

context Element
inv: let kindAllowed = **Set**{StructuralElement, StructuralBehavioralElement,
ConstraintElement, BehavioralElement}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context StructuralElement
inv: let kindAllowed = **Set**{TypeElement, InstanceElement}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context TypeElement
inv: let kindAllowed = **Set**{InvalidType, VoidType, TypeType, ElementType, AnyType, OrderSetType, SequenceType, BagType, SetType, TupleType, **Boolean**, UnlimitedNatural, **Real**, **String**, Enumeration, EnumerationLiteral, ComponentClass, Property, Association, Generalization, GeneralizationSet, Usage}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e)) **and not** notAllowed−>exists(e: Element | self.oclIsKindOf(e)))

context InstanceElement
−− *LiteralSpecification is already in OCL as LiteralExp*
inv: let kindAllowed = **Set**{ComponentObject, Slot, Link}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context StructuralBehavioralElement
inv: let kindAllowed = **Set**{Comment, LiteralElement, Parameter, QueryOperation, EffectOperation}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context LiteralElement
inv: let kindAllowed = **Set**{EnumLiteralExp, RealLiteralExp, UnlimitedNaturalExp, IntegerLiteralExp, StringLiteralExp, BooleanLiteralExp, NullLiteralExp, InvalidLiteralExp, CollectionLiteralExp, CollectionRange, CollectionItem, TupleLiteralExp, TupleLiteralPart}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context ConstraintElement
inv: let kindAllowed = **Set**{CommonConstraintElement, StructuralConstraintElement, BehavioralConstraintElement}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context CommonConstraintElement
inv: let kindAllowed = **Set**{ExpressionInOcl, Variable, IfExp, VariableExp, TypeExp, IteratorExp, IterateExp, OperationCallExp, PropertyCallExp, LetExp}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context StructuralConstraintElement
inv: let kindAllowed = **Set**{Inv, Derived, Init, PropDef}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context BehavioralConstraintElement
inv: let kindAllowed = **Set**{Post, Pre, Body, OpDef, UnspecifiedValueExp, MessageExp, MessageType}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context BehavioralElement
inv: let kindAllowed = **Set**{ActivityElement, ProtocolElement}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context ActivityElement
inv: let kindAllowed = **Set**{CallSubActivityAction, CallOperationAction, ActionInOcl, InputPin, OutputPin, Activity, ActivityParameterNode, InitialNode, DecisionNode, MergeNode, ForkNode, JoinNode, FlowFinalNode, ActivityFinalNode, ControlFlow, ObjectFlow, ExceptionHandler, ActivityPartition},

−− excluding ValuePin, a specialization of InputPin
notAllowed = **Set**{ValuePin}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e)) **and not** notAllowed−>exists(e: Element | self.oclIsKindOf(e)))

InterruptibleActivityRegion was deleted since the interruption can only be triggered by AcceptEventAction which was previously deleted.

context ProtocolElement
inv: let kindAllowed = **Set**{ProtocolStateMachine, Region, PseudoState, State, ProtocolTransition}
in (kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

1.2.2 Types

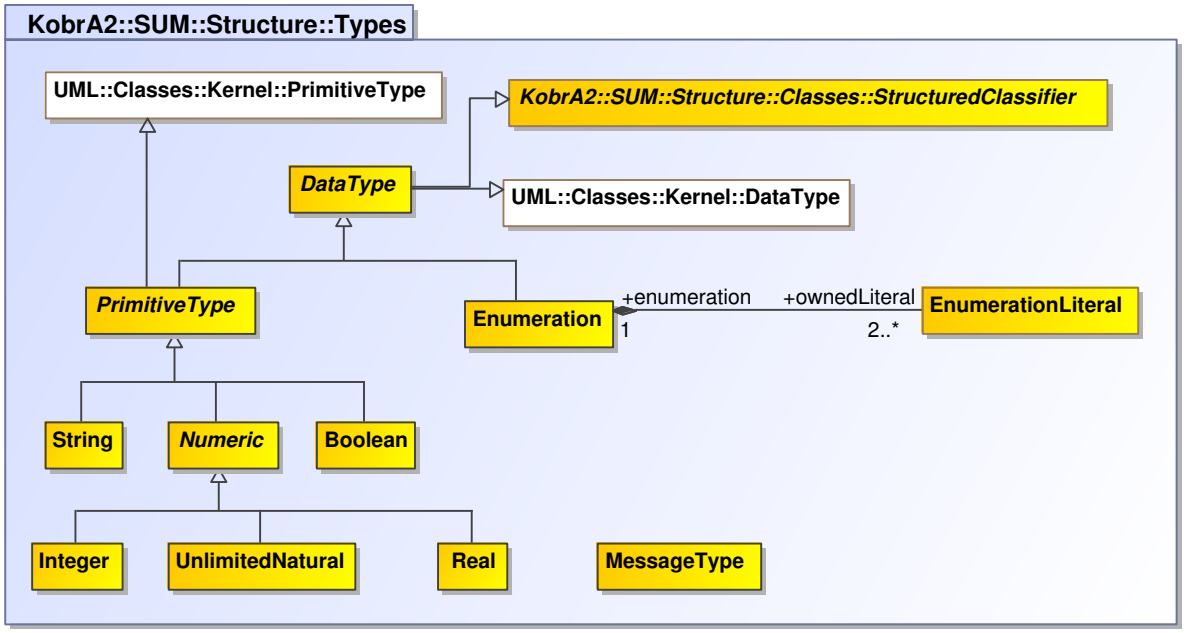

Figure 1.7: Types

context DataType
−− the UML DataType can have operations to represent abstract DataTypes with no OID, in KobrA all Instances of complex DataTypes are objects with OID, therefore we prohibit Operations in KobrA DataTypes
−− KobrA DataType specializes KobrA StructuredClassifier to conform to the fact that UML DataType specializes UML Classifier
inv: ownedOperation−>union(ownedAttribute)−>isEmpty
−− in the UML, DataType is concrete which seems to be a bug, so we make it abstract
inv: isAbstract

context PrimitiveType
−− in the UML, PrimitiveType is concrete which seems to be a bug, so we make it abstract
inv: isAbstract

context EnumerationLiteral
inv: specification .bodyExpression.oclIsKindOf(EnumLiteralExp)

context MessageType
inv: referredSignal−>isEmpty()

1.2.3 Classes

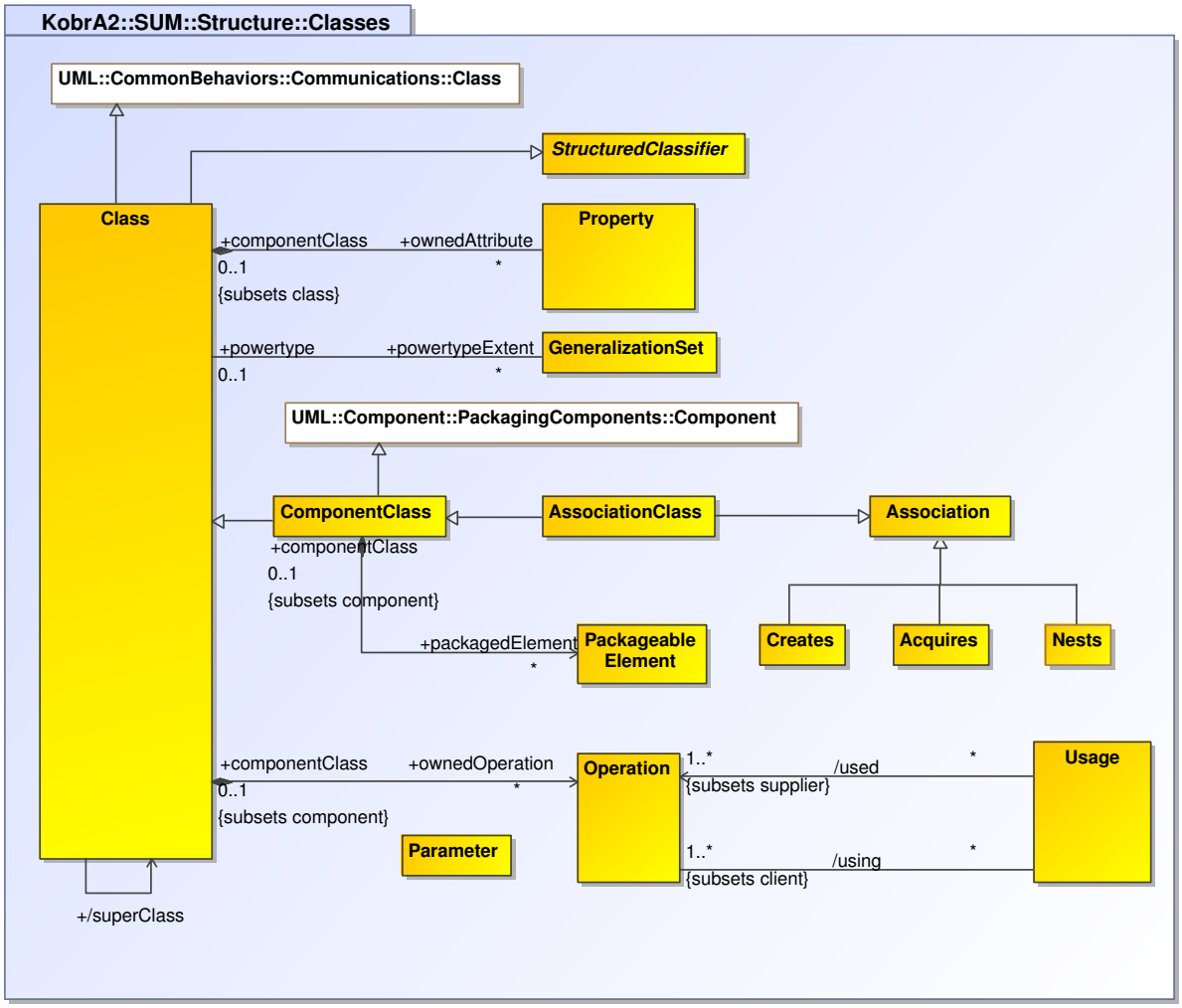

Figure 1.8: Classes

ComponentClasses encapsulate complex and reusable behaviors, whereas Classes encapsulate only non-reusable and/or very simple behaviors.

ComponentClass inherits (indirectly) from the Class from Communications so it has the *isActive* attribute.

context StructuredClassifier
−− *types of operation parameters that exclude BehavioralClassifiers (e.g. Activities and ProtocolStateMachines)*
inv: oclIsKindOf(ComponentClass) **or** oclIsKindOf(DataType)

context ComponentClass
−− *We only use ProtocolStateMachine and Activities as specification of behavior, (no Interactions, no BehavioralStateMachines, no Use Cases, etc.)*
inv: ownedBehavior−>forAll(oclIsKindOf(Actitivity) **or** oclIsKindOf(ProtocolStateMachine)

-- *all packageableElement must be related to their container by an explicit "Nests" association in order to be able to navigate to them with OCL, because OCL 2.0 doesn't discuss access to packaged elements, in addition nests associations are instantiable into nests-links whereas the relationship packagedElement is not, so it allows to have instance diagrams that follow the structure of corresponding service or type diagrams*

inv:
let nestedComponentClasses = packagedElement->select(oclIsKindOf(componentClass))
in nestedComponentClasses->forAll(n : ComponentClass |
let ends = ownedAttributes->select(association->notEmpty())
in ends->exists(e: Property | e.opposite = self.ownedAttribute **and** ownedAttribute.
association->oclIsKindOf(Nests)))

-- *take off everything from component except packagedElement and what it inherits from Class (from UML::Behavior::CommonBehavior::Communications), e.g. we don't need nestedClassifier, as there's no clear concrete syntax in the UML standard specification and most UML CASE tools provide easier-to-use concrete syntax support for packagedElement*

inv: nestedClassifier ->union(role)->union(part)->union(ownedConnector)->union(
collaborationUse)->union(representation)->union(realization)->union(required)->
union(provided)->isEmpty()

context ComponentClass::superClass
derive: generalization.general->select(oclIsKindOf(ComponentClass)

context Class::superClass
derive: generalization.general->select(oclIsKindOf(Class))

context Class
-- *must not have nested Classifiers, must not have an associated protocol, nor a packaged element*

inv: union(Protocol)->union(PackagedElement)->isEmpty()

-- *the only behaviors that a Class can own, are the activities that are the methods of its operations*

inv: ownedBehavior->forAll(oclIsKindOf(Activity))

-- *A Class can only be acquired (by a ComponentClass that is not a Class), but it cannot itself acquire any other ComponentClass; hence it must be located at the* navigable end of the acquired association.

inv: let p=k2OwnedAttribute, a=p.k2Association, me=a.k2memberEnd->select(e|e<>p),
kc=me.componentClass
-- *if there is a ComponentClass associated to this class ...*
in kc -> notEmpty() **implies**
-- *then if it's Acquires or Creates ...*
(a->forall(oclIsKindOf(Acquires) **or** oclIsKindOf(Creates)) **implies**
-- *this Class must be at it's navigableOwnedEnd*
(let target = a.navigableOwnedEnd.componentClass
in target->size() = 1 **and** target->includes(self)))

The attribute concurrency of BehavioralFeature (from Communications) of type CallConcurrencyKind is not needed.

context Parameter

−− must be either DataType or ComponentClass (or a subclass of those)
inv: type.oclIsKindOf(DataType) **or** type.oclIsKindOf(ComponentClass)

context Property
−− have all association ends owned by the ComponentClasses, to ease writing OCL constraints (navigate from ComponentClasses to Associations).
−− However, this might create problems when reusing existing CASE tools
inv: owningAssociation−>isEmpty()
inv: association.memberEnd−>size() = 2 **implies** opposite−>notEmpty()
−− Attribute types are limited to primitive types and Enumerations
−− All dependencies between ComponentClasses are modeled by associations, so we don't need InstanceValue (see constraint on Element)
inv: type−>oclIsKindOf(ComponentClass) **implies** association−>notEmpty()

Note for readers who are familiar with an intermediate KobrA version (between KobrA 1 and KobrA 2): *Provides* used to be there as well, as specialization of Association, but we think it is redundant with the opposite, which was called *Requires*. *Requires* used to be a generalization of *Creates* and *Acquires*.

context Association
−− navigate to ComponentClass; if that's a nested component, then we access its containing component by navigating from Classifier to ComponentClass (role name 'componentClass')
inv: let me=memberEnd, c = me.componentClass, containers = c.componentClass
in
−− all memberEnds of this assciation must have a common containing ComponentClass
forAll(containers−>isUnique())
or
−− OR one end must be a containing ComponentClass and the other ends (we allow associations with more than two ends) must be its nested ComponentClasses
−− among the associated components, suppose the existence of the containing one (container), retrieve all its association ends, from them, navigate to the associations, then get the other ends of these associations, then get the classifiers at these other ends and state that they must be nested classifiers of container
c−>exists(container: Component | let containerEnds = ownedAttribute−>select(association−>notEmpty()), otherEnds = self.memberEnd − containerEnds in container−>nestedClassifier−>includes(otherEnds. componentClass))

Note for readers familiar with KobrA 1: The creation of a component follows these rules: By default, a ComponentClass creates all its directly nested ComponentClasses and each Class creates the Classes with which it has a composite association (aggregationKind=#composite). However, other creation patterns e.g. factory pattern can be modeled explicitely by using the "creates" association.

context AssociationClass
−− AssociationClass is not allowed to inherit from Acquires and Nests
inv: let gg = generalization.general in
(gg.oclIsKindOf(Association) **and** (**not Set**{Acquires, Nests}−>exists(e: Element | gg. oclIsKindOf(e))))

A ComponentClass A acquires another ComponentClass B iff at least one operation of A calls at least one operation of B. In the modeling process, the developer usually first specifies the acquires relationships and then the calling relationships between operations in their OCL specifications, OCL realizations or activity realizations.

The element Usage is derived, we put it in the SUM to be able to reuse it for constraint checking among elements that later appear in different views.

One operation A is directly using another operation B, iff

- the OclExpression of the postcondition or body of A contains a OperationCallExp whose referredOperation is B
- OR the Activity of operation A (rolename "method") contains a CallOperationAction whose operation is B

```
context Usage
inv: (let constraints = using.precondition->union(using.body)
          ->union(using.postcondition),
      usedCalls = constraints->exists(specification.bodyExpression.containsCallTo(used)))
     or
     (let coa = using.method.node->select(oclIsKindOf(CallOperationAction))
     in coa->exists(operation=used))
     or
     (let scoa = using.method.subactivities().node->select(oclIsKindOf(
         CallOperationAction))
     in scoa->exists(operation=used))


context Activity::subactivities (): Set(Activity)
body: let cba = node->select(oclIsKindOf(CallSubActivityAction)),
          sub1 = cba.behavior->select(oclIsKindOf(Activity)),
          subN = sub1->collect(subactivities())
      in sub1->union(subN)->asSet()

context LoopExp::containsCallTo(o: Operation): Boolean
body: body.containsCallTo(o)

context CallExp::containsCallTo(o: Operation): Boolean
body: source.containsCallTo(o)

context OperationCallExp::containsCallTo(o: Operation): Boolean
body: referredOperation=o

context CollectionLiteralExp::containsCallTo(o: Operation): Boolean
body: let pr = part->select(oclIsKindOf(CollectionRange)),
          pi = part->select(oclIsKindOf(CollectionItem))
          embeddedExp = pr.first->union(pr.last)->union(pi.item)
      in embeddedExp->asSet()->exists(containsCallTo(o))

context LetExp::containsCallTo(o: Operation): Boolean
body: variable.initExpression.containsCallTo(o)

context MessageExpression::containsCallTo(o: Operation): Boolean
body: calledOperation.operation=o

context IfExp::containsCallTo(o: Operation): Boolean
```

body: condition.containsCallTo(o) **or** thenExpression.containsCallTo(o) **or** elseExpression.containsCallTo(o)

We chose Acquires to be an Association so that it can be instantiated and shown in an Instances Diagram. As Usage is derived and we don't need to show it, it is only a Dependency.

context Acquires
-- *must be a binary association*
inv: memberEnd->size()=2
-- *only between two ComponentClasses (but not their subclasses)*
inv: memberEnd->union(ownedEnd)->union(navigableOwnedEnd)->forAll(OclIsKindOf(ComponentClass) **and not** OclIsKindOf(Class))

inv: let acquiredOps=navigableOwnedEnd.componentClass.ownedOperation,
acquringOps=
memberEnd->reject(navigableOwnedEnd).componentClass.ownedOperation
in
acquiringOps.using.used->intersection(acquiredOps)->size() >= 1

context GeneralizationSet
-- *only Classes might be a powertype*
inv: powertype->forAll(oclIsKindOf(Class))

1.2.4 Instances

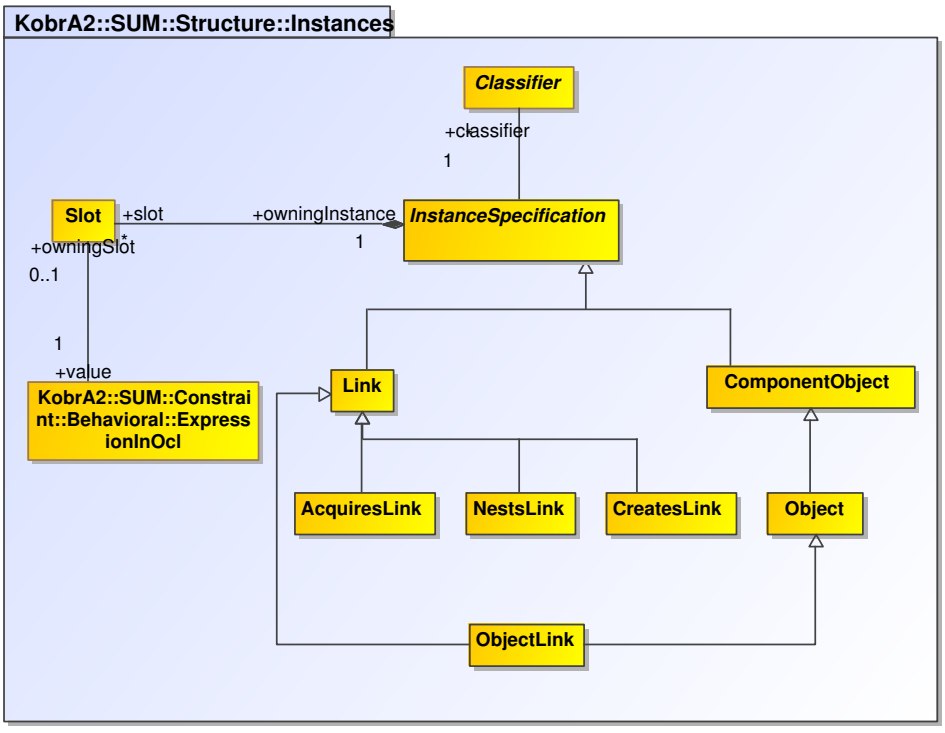

Figure 1.9: Instances

As a general rule for the following constraints, we only allow the corresponding classifiers from the SUM to be represented by elements of our instance package.

There is no concrete syntax in the UML for several classifiers for one InstanceSpecification, therefore we prohibit it.

All slots must have one ValueSpecification, otherwise there is no point in creating them. Also, there is no concrete syntax for several ValueSpecifications of a Slot, so we prohibit it.

context InstanceSpecification
-- *only allow classifiers of kind ComponentClass, Association or DataType to be represented by InstanceSpecification*
inv: classifier ->forAll(oclIsKindOf(ComponentClass) **or** oclIsKindOf(Association) **or** oclIsKindOf(DataType))
-- *since we have prohibited DataTypes to have attributes, their instances cannot have slots and their values are specified as LiteralExp in OCL (to allow real numbers, tuples and specialized types of collections)*
inv: classifier .oclIsKindOf(DataType) **implies** (
slot->isEmpty() **and**
let s = specification
in s->notEmpty() **and** s.oclIsKindOf(ExpressionInOcl) **and** s.bodyExpression(
oclIsKindOf(LiteralExp)))

context Slot
-- *values of a slot must be literal expressions in OCL (i.e. LiteralExp)*
inv: value.bodyExpression->oclIsKindOf(LiteralExp)

context Link
-- *slots are only useful for objects and not for links*
inv: classifier .oclIsKindOf(Association) **and** slot->isEmpty()

context NestsLink
inv: classifier .oclIsKindOf(Nests)

context AcquiresLink
inv: classifier .oclIsKindOf(Acquires)

context CreatesLink
inv: classifier .oclIsKindOf(Creates)

context ComponentObject
inv: classifier .oclIsKindOf(ComponentClass)
-- *ValueSpecification (role name 'specification') only makes sense for PrimitiveTypes and Enumeration, however the UML metamodel does not restrict it*

context Object
inv: classifier ->forAll(oclIsKindOf(KobrA2::SUM::Classes::Kernel::Classes::Class))

context ObjectLink
inv: classifier ->oclIsKindOf(AssociationClass)

1.3 SUM - Behavior

1.3.1 Common Behaviors

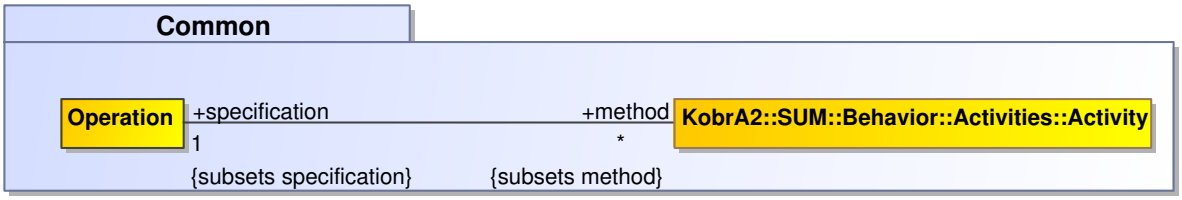

Figure 1.10: CommonBehaviors

In the UML, the only subclasses of BehavioralFeature are Operation and Reception.

context Operation
−− prohibit other behavior specifications (role name "method") than Activity
−− (the UML would also allow OpaqueBehavior, Interaction and StateMachine)
inv: method−>notEmpty() **implies** method−>forAll(oclIsKindOf(Activity))

−− bodycondition and postcondition are not allowed together
inv: bodycondition−>notEmpty() **implies** postcondition−>isEmpty()
inv: postcondition−>notEmpty() implie bodycondition−>isEmpty()

−− Operation must either have a postcondition, body, or a method, otherwise it is underspecified
inv: bodycondition−>notEmpty() **or** postcondition−>notEmpty() **or** method−>notEmpty()

−− if an Operation has Parameters and an Activity (role name "method"), then these Parameters must be the same as those of ParameterNode of the Activity
context Operation
inv: self .method−>forAll (act | act.oclIsKindOf(K2Activity) **implies**
act.oclAsType(K2Activity).node−>select(
oclIsKindOf(K2ActivityParameterNode))−>collect(node |
node.oclAsType(K2ActivityParameterNode).parameter)−>asBag()
=self.ownedParameter−>asBag())

1.3.2 Actions

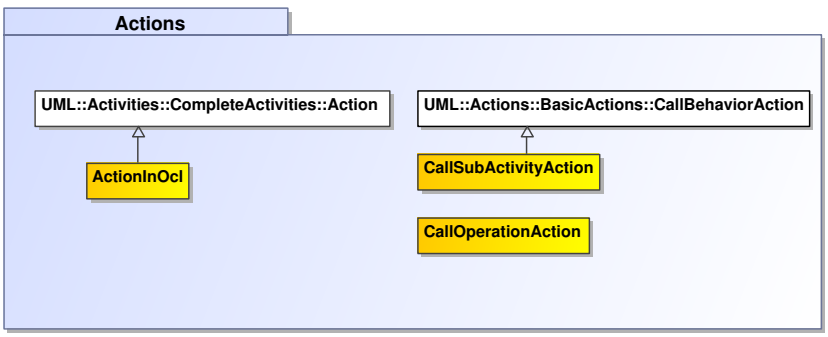

Figure 1.11: Actions

context CallSubActivityAction
−− *must call an Activity*
inv: behavior.oclIsKindOf(Activity)

Every Action must be either call a Behavior that is a subactivity (CallSubActivityAction which is a CallBehaviorAction from UML), or an Operation (CallOperationAction), or have an OCL expression as localPostcondition to specify it (e.g. call multiple Operations and algorithmically relate in one OCL expression). We neither use OpaqueAction nor OpaqueBehavior because their body is a String attribute instead of being an association to OpaqueExpression (which is in the SUM restricted to ExpressionInOcl).

context ActionInOcl
inv: localPostcondition−>notEmpty()

1.3.3 Activities

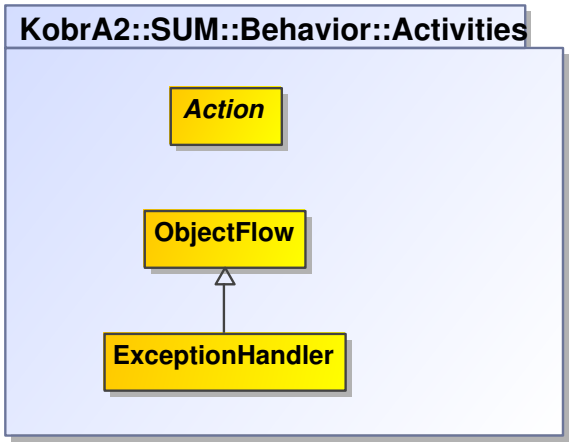

Figure 1.12: Complete Activities

context Activity
−− *direct subactivities, will be needed later*
def: subactivities : **Set**(Activity) = node−>select(oclIsKindOf(CallSubActivityAction).behavior)−>asSet()
def: subactivityClosure : **Set**(Activity) = subactivities−>union(subactivies.subactivityClosure)−>asSet()
−− *our convention is that the Activity must be named after the operation which behavior it implements*
inv: name = specification.name
−− *the activity is read only, iff the operation that it implements is a query*
inv: isReadOnly = specification.isQuery

In the UML ExceptionHandler is simply an element, even though its concrete syntax is similar to the InterruptingEdge of an InterruptibleActivityRegion. Therefore, we make it an ObjectFlow.

Note: Might be needed later on:

context InterruptibleActivityRegion::enclosingRegions()
let sg=superGroup
in
if sg−>isEmpty()

then Set{}
else sg−>union(sg.enclosingRegions())
endif

context Action
−− *The local precondition is redundant with the guard from ActivityEdge, thus we don't need it.*
inv: localPrecondition−>isEmpty()

1.3.4 Protocol State Machines

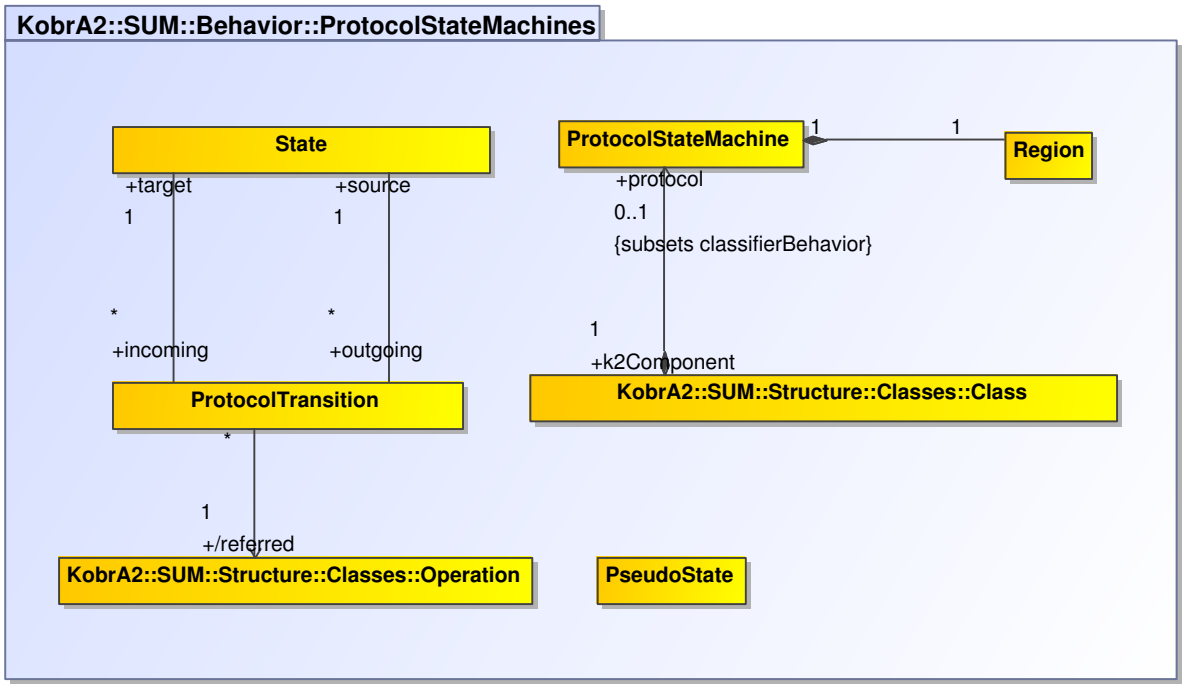

Figure 1.13: Protocol State Machines

The associations between State and ProtocolTransition are only drawn for convenience, they are the same as in the UML specification.

State invariants are derived in the view, please see the constraints in the Specification - Behavioral - Algorithmic view.

We created a new association between ComponentClass and ProtocolStateMachine. In the UML, ProtocolStateMachines are only indirectly related to Components through interfaces or ports.

Region is in the package only for the cardinality constraints of its association to ProtocolStateMachine.

We also changed the cardinality of the association between ProtocolTransition and Operation.

context ProtocolTransition
−− *for our protocol we only want transitions that exit the current (composite) source state*
inv: kind=#external
−− *a QueryOperation doesn't change the state*
inv: referred .oclIsKindOf(QueryOperation) **implies**
source=target
inv: source=target **implies**
referred .oclIsKindOf(QueryOperation)

−− *an EffectOperation does change the state to another one*
inv: referred.oclIsKindOf(EffectOperation) **implies**
source <> target
inv: source <> target **implies**
referred.oclIsKindOf(EffectOperation)
−− *the postcondition of a transition is implicitly derived from the state invariant of its target state*
inv: postcondition−>isEmpty()

As both a transition and an Operation can have a preconditions, they must be compatible. Compatibility here means that the precondition of the State must imply the precondition of the Operation.

context ProtocolTransition
inv: let te=preCondition.bodyExpression, pe=referred.precondition.bodyExpression
in
−− *if both Operation and the Transition have preconditions associated to them ...*
te−>notEmpty() **and** pe−>notEmpty()
implies
−− *the precondition of the Transition must imply the one of the Operation*
(te **implies** pe)

context ProtocolTransition
−− *The postcondition of the Transition must be equal to that of its referred Operation.*
inv: postCondition.bodyExpression=referred.postcondition.bodyExpression

context ProtocolStateMachine
−− *in the SUM, substates are not allowed*
inv: submachineState−>isEmpty()
−− *we don't need ports and interfaces*
inv: port−>isEmpty()
inv: interface−>isEmpty()

context PseudoState
−− *no PseudoStates other than "initial" allowed, because it is our restricted ProtocolStateMachine*
−− *note that FinalState is a subclass of State and not a kind of PseudoState*
inv: kind=#initial
−− *see connectionPoint of State*
inv: state−>isEmpty()
−− *we only have one region, so this region must be contained in the statemachine that contains the pseudostate*
inv: container.statemachine = statemachine

context State
−− *the container region inherited to State from the Vertex should be equal to the region of the State*
inv: region = container
−− *connectionPoint refers to the entry and exit connection points of composite states, which are prohibited in ProtocolStateMachines (even though UML forgot to specify it)*
inv: connectionPoint−>isEmpty()
−− *in the SUM, substates are not allowed*
inv: submachine−>isEmpty()

1.4 SUM - Constraint

1.4.1 OCL Expressions

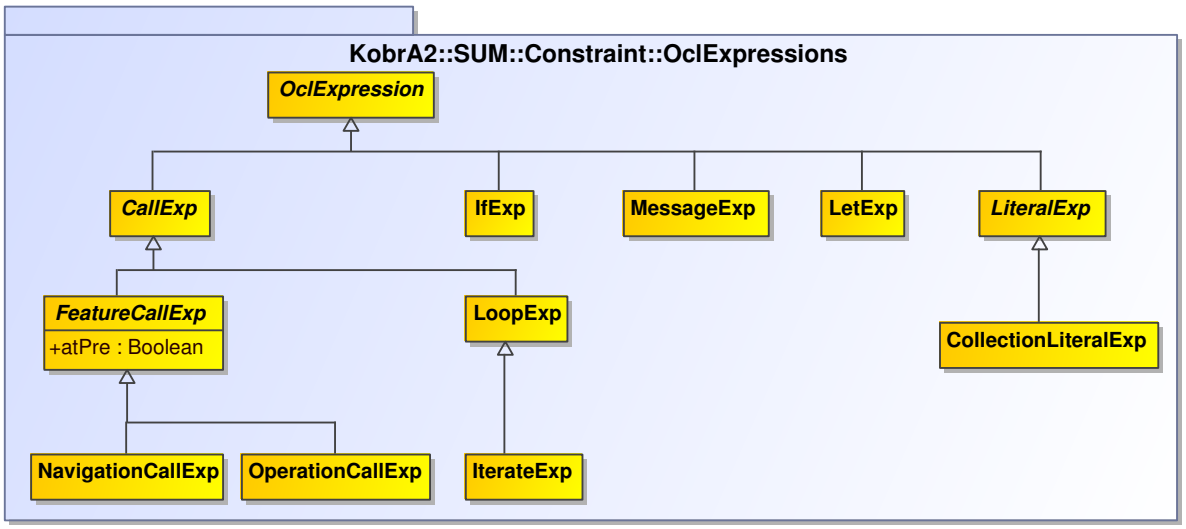

Figure 1.14: OCL Expressions

Note: Generalization relationships and LiteralExp are shown only for clarity in this diagram.

context OclExpression
derive: boolean = type.oclIsTypeOf(**Boolean**)

def: containsAtPre : **Boolean** =
subExps−>select((oclIsKindOf(PropertyCallExp) **and** atPre) **or**
(oclIsKinfOf(OperationCallExp) **and** referredOperation.isQuery
and atPre))−>notEmpty()

def: containsResult : **Boolean** =
subExps−>select(oclIsKindOf(VariableCallExp) **and**
referredVariable = resultVariable)−>notEmpty()

def: containsOclIsNew : **Boolean** =
subExps−>select(oclIsKindOf(OperationCallExp) **and**
referredOperation.name = 'OclIsNew')−>notEmpty()

def: containsMessageExp: **Boolean** =
subExps−>select(oclIsKinfOf(MessageExp))−>notEmpty()

derive: query = **not** (containsAtPre **or** containsResult **or** containsOclIsNew **or**
containsMessage Exp)

context CallExp
def: subExps : **Set**(**OclExpression**) = source−>union(source.subExps)

context IfExp
def: subExps : **Set**(**OclExpression**) =
let c = condition, t = thenExpression, e = elseExpression

```
        in c->union(c.subExps)->union(t)->union(t.subExps)->union(e)->union(e.
            subExps)

context MessageExp
-- we don't have Signals in KobrA 2
inv: sentSignal->isEmpty()
def: subExps : Set(OclExpression) =
        let t = target, a = argument
        in t->union(t.subExps)->union(a)->union(a.subExps)

context LetExp
def: subExps : Set(OclExpression) =
        let vie = variable.initExpression
        in in->union(in.subExps)->union(vie)->union(vie.subExps)

context FeatureCallExp
-- to be sure that we need the additional attribute atPre, one would need and understand the
-- many many pages in the OCL specifcation that refer to the @pre keyword, although they
-- are all in the concrete syntax chapter, or vague in the relation to UML chapter, or in the
-- semantics appendix

context LoopExp
def: subExps : Set(OclExpression) =
        let b = body, ii = Iterator.initExpression
        in b->union(b.subExps)->union(ii)->union(ii.subExps)

context NavigationCallExp
def: subExps : Set(OclExpression) = qualifier->union(qualifier.subExps)

context OperationCallExp
def: subExps : Set(OclExpression) = argument->union(argument.subExps)

context IterateExp
def: subExps : Set(OclExpression) =
        let ri = result.initExpression in ri->union(ri.subExps)

-- Check need for and inclusion of qualified associations in SUM

context CollectionLiteralExp
def: subExps : Set(OclExpression) =
        let f = part.first , l=part.last , i = part.item
        in f->union(f.subExps)->union(l)->union(l.subExps)->union(i)->union(i.
            subExps)
```

1.4.2 Common Constraints

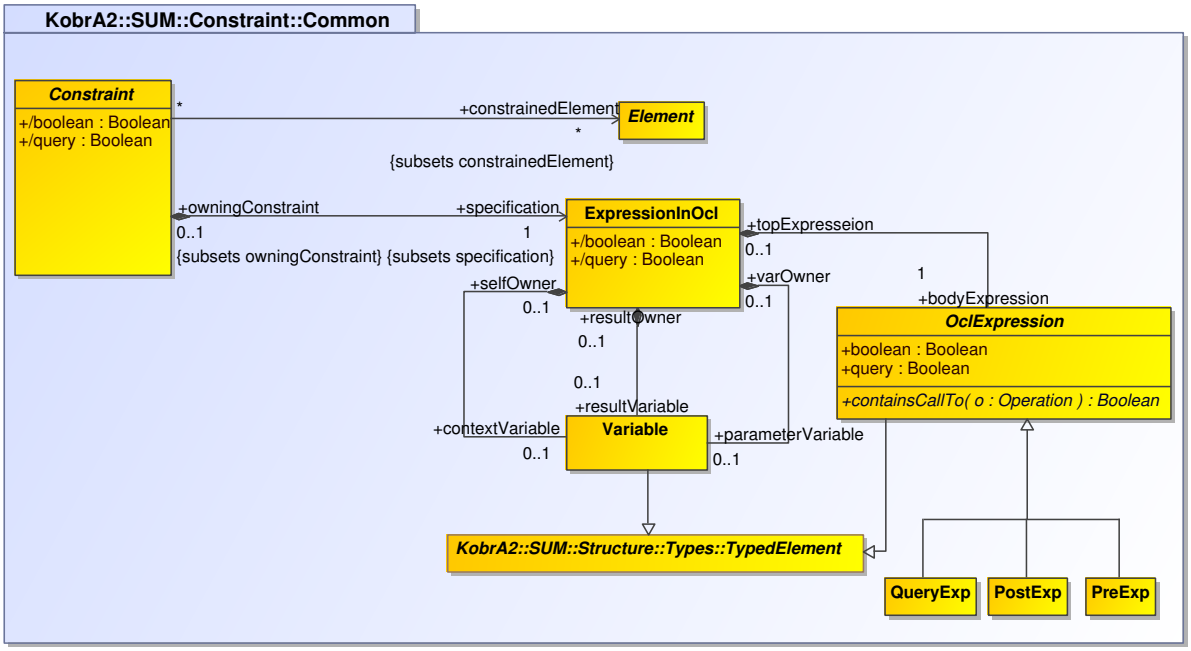

Figure 1.15: Common Constraints

We want all constraints to be written in OCL in order to automatically process them, thus the association *specifiation* to *ExpressionInOcl* (instead of ValueSpecification as in the UML).

context Constraint
−− ExpressionInOcl allows to specify all the value specifications that are in the UML
inv: specification −>oclIsKindOf(ExpressionInOcl)
−− only allow certain elements to be constrained
inv: let allowed= **Set** (ActivityParameterNode{ComponentClass, Property, Operation, LocalPostcondition, ProtocolTransition}
in allowed−>exists(e: Element | self.constrainedElement−>oclIsKindOf(e))
−− what follows the context keyword in the OclExpression, is the Element that the Expression constraints (this is an alignment between the OCL und UML metamodel)
inv: specification .contextVariable.type = **context**
derive: boolean = specification.boolean
derive: query = specification.query

context OclExpression
inv: boolean = type.oclIsTypeOf(**Boolean**)
−− contains neither result nor @pre nor oclIsNew() nor isSent nor message
derive: query = **not** (containsAtPre **or** containsMessage Exp)

context QueryExp **inv**: query
context PostExp **inv**: **not** query
context PreExp **inv**: query **and** boolean

context ExpressionInOcl
derive: boolean = bodyExpression.boolean
derive: query = bodyExpression.query
inv: boolean **implies** resultVariable.type = boolean

As a *Parameter* can have a *defaultValue* of type *ValueSpecification* in the UML, we constrain it to be an *ExpressionInOcl*.

context ValueSpecification
−− all ValueSpecification elements must be of kind ExpressionInOcl
−− ExpressionInOcl allows to specify all UML value specifications
inv: oclIsKindOf(ExpressionInOcl)

1.4.3 Structural Constraints

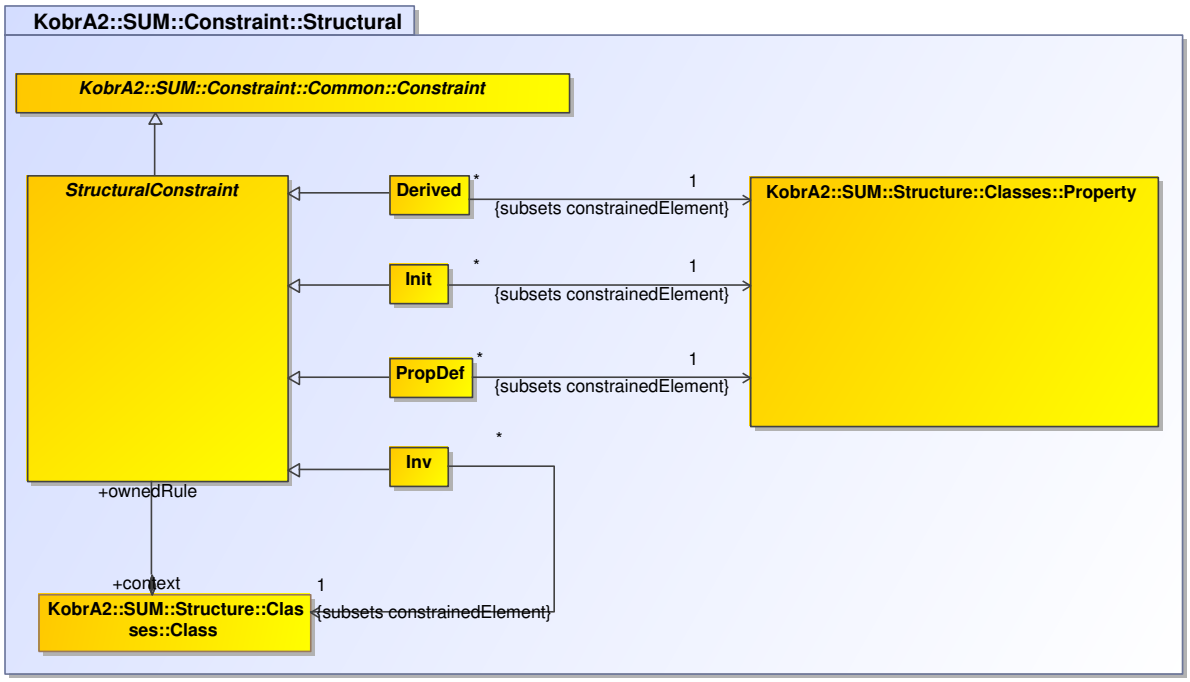

Figure 1.16: Structural Constraints

context StructuralConstraint
inv: specification −>oclIsKindOf(ExpressionInOcl)

−− The ComponentClass owns the constraints on its attributes
context Derived
inv: constrainedElement.componentClass = context
inv: constrainedElement.oclIsKindOf(Property) **and** specification.contextVariable.type.
oclIsKindOf(Property)
inv: **context**.ownedAttribute−>include(constrainedElement
−− the type of the constraint must be the type of the property which value it derives
−− the subsetted rolename constrainedElement is used to navigate
inv: type = constrainedElement.type
−− must be a query, see derived attribute of Constraint (in common constraint package)
inv: query

context Init
inv: constrainedElement.componentClass = context
inv: constrainedElement.oclIsKindOf(Property) **and** specification.contextVariable.type.
oclIsKindOf(Property)

inv: **context**.ownedAttribute−>include(constrainedElement)
inv: type = constrainedElement.type
inv: query

context PropDef
−− *as we define a new property not in the model with "def :", the constrainedElement can only be the ComponentClass (as in Inv) and not as in Init and Derived*
inv: constrainedElement = context
inv: constrainedElement.oclIsKindOf(Property) **and** specification.contextVariable.type.oclIsKindOf(ComponentClass)
inv: **context**.ownedAttribute−>include(constrainedElement)
inv: type = constrainedElement.type
inv: query

context Inv
−− *For an invariant, the constraints are owned by the constrainedElement*
inv: context = constrainedElement
inv: constrainedElement.oclIsKindOf(ComponentClass) **and** specification.contextVariable.type.oclIsKindOf(ComponentClass)
−− *see derived attributes boolean and query from KobrA ExpressionInOcl*
inv: boolean **and** query

1.4.4 Behavioral Constraints

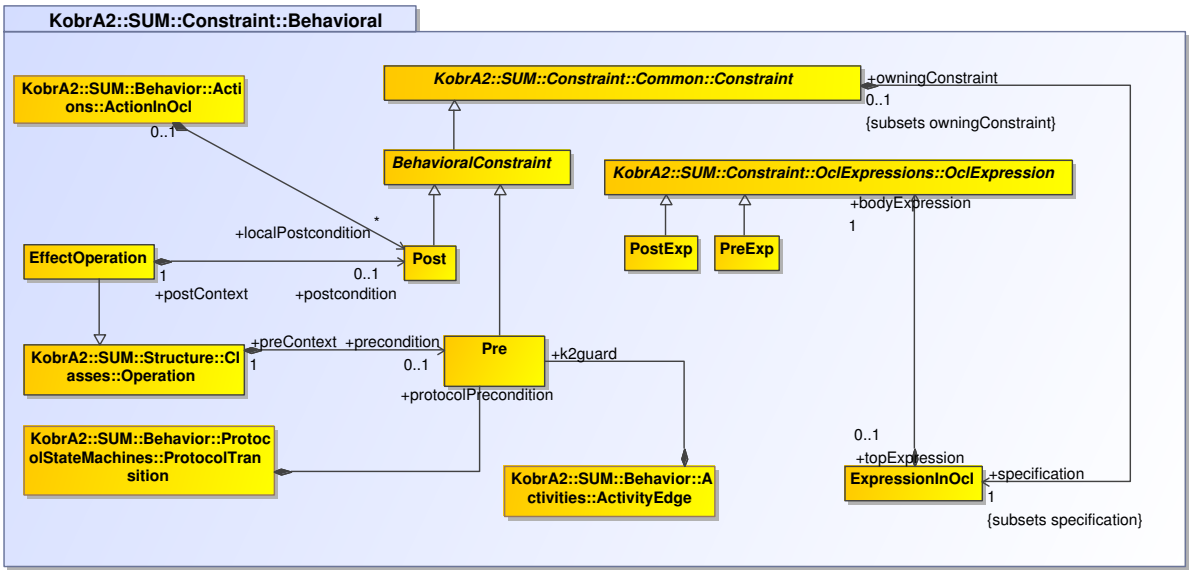

Figure 1.17: Pre- and postcondition constraints

context EffectOperation
−− *see QueryOperation*
inv: postCondition−>isEmpty() **implies** method−>notEmpty()
inv: (**not** isQuery) **and** postcondition.oclIsKindOf(Post)
inv: ownedParameter−>select(direction = return) = postcondition.specification.resultVariable

```
context ActivityEdge
inv: guard.oclIsKindOf(ExpressionInOcl) and guard.bodyExpression.oclIsKindOf(PreExp)

context Pre
-- the ExpressionInOcl of a Pre only contains the pre keyword
inv: contextVariable = preContext
inv: specification .bodyExpression->oclIsKindOf(PreExp)
inv: boolean and query
--
inv: let pc = preContext
     -- if Pre has a preContext (an Operation), then the constrainedElement and the context
        (which is subsetted by preContext in the UML metamodel) of Pre are that operation
     in if pc->size() = 1
        then constrainedElement = pc and context = pc and pc.oclIsKindOf(Operation)
        -- otherwise, the constrainedElement and the context of Pre are a ProtocolTransition
        else let pt = protocolTransition in
            if pt->size() = 1
            then constrainedElement = pt and context = pt and pt.oclIsKindOf(
                ProtocolTransition)
            else let ae = activityEdge in
              ae->size() = 1 implies
              constrainedElement = ae and context = ae.activity and ae.oclIsKindOf(
                  ActivityEdge)
        endif

context Post
-- the ExpressionInOcl of a Post only contains the post keyword
inv: contextVariable = postContext
inv: specification .bodyExpression->oclIsKindOf(PostExp)
inv: boolean and not query
-- a post constraint must constrain either an EffectOperation (postContext)
inv: let pc = postContext
     in if pc->size() = 1
        then constrainedElement = pc and context = pc and pc.oclIsKindOf(
            EffectOperation)
        -- or an ActionInOcl in an activity diagram (as a localPostcondition) in which case
           the context of the postcondition is the Activity containing the Action and the
           constrainedElement is the Action itself
        else let a = actionInOcl
        -- (because there is no rolename from the constraint that is the local postcondition of
           an Action and that Action "actionInOcl" navigates from the Post constraint to the
           ActionInOcl of which is the local postcondition)
            in if a->size() = 1
               then context = a.activity and constrainedElement = a
               -- or a State in a ProtocolStateMachine (as a StateInvariant) in which case
                  the constrainedElement and the context are the State itself
               else let os = owningState
                 in os->size() = 1 implies

                       constrainedElement = os and context = os and os.oclIsKindOf(State)
               endif
```

endif

context OpDef
inv: bodyExpression.type = constrainedElement.type

context PreExp
inv: type=**Boolean**

context PostExp
inv: type=**Boolean**

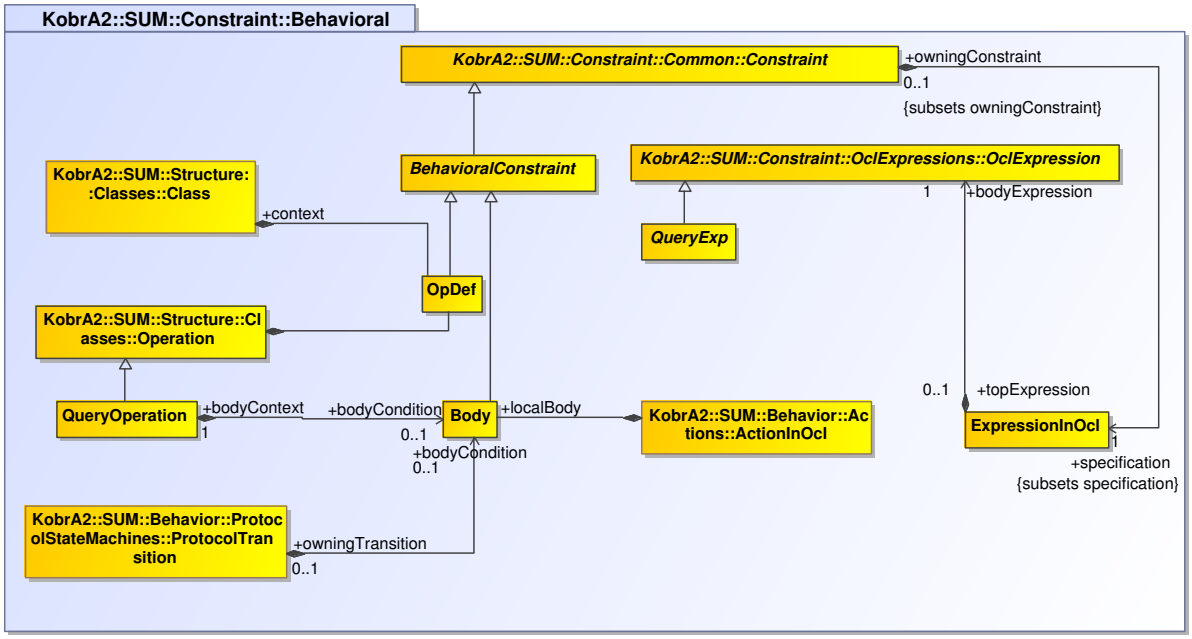

Figure 1.18: Operation definition and body constraints

A *QueryOperation* may not change Elements, thus we set the property *isQuery* to true and don't allow *out* or *inout* Parameters. In addition, we want a *QueryOperation* to be either specified in OCL or have a Behavior associated to it. Later on in this specification, we constrain this Behavior to be an Activity.

We don't need DefExp as it is a query expression with the return type of the operation, like BodyExp.

context QueryOperation
inv: isQuery=true
−− only "in" and "return" Parameters are allowed for a QueryOperation
inv: ownedParameter−>forAll(ParameterDirectionKind=#in **or** ParameterDirectionKind=#return)
−− either the body is specified in OCL or a Behavior (which is later constrained to be an Activity), must be associated to QueryOperation, or both
inv: bodyCondition−>isEmpty() **implies** method−>notEmpty()
inv: method−>isEmpty() **implies** bodyCondition−>notEmpty()

context Body
−− the ExpressionInOcl of a body constraint only contains the body keyword
inv: contextVariable = bodyContext

−− the return type of a body expression is the return type of the operation
inv: bodyExpression.type = bodyContext.type

context OpDef
inv: query
inv: constrainedElement.oclIsKindOf(Operation) **and context**.oclIsKindOf(ComponentClass)
inv: **context**.ownedOperation−>includes(constrainedElement)

For detailed information on how the OCL pre- and postcondition and bodyExpression constraints fit into the UML metamodel, please refer to the OCL specification, Sections 12.7 and 12.10.

context QueryExp
−− see definiton of the derived query attribute of OclExpression
inv: query

context BehavioralConstraint
−− an operation owns its constraints
inv: **context** = constrainedElement

1.5 Views

1.5.1 Package Dependencies

The views of a component in KobrA 2 are structured as follows.

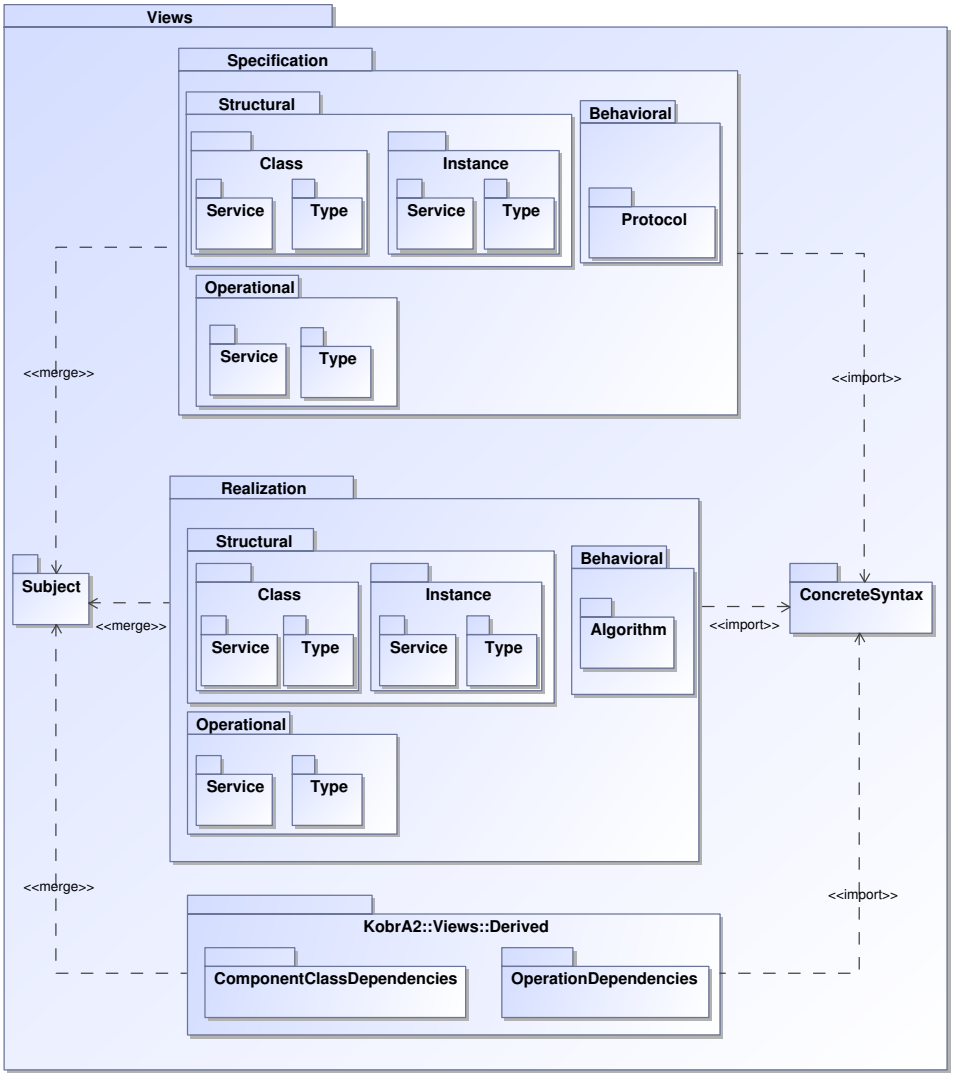

Figure 1.19: KobrA 2 Views Package Nesting

context views
inv: let spec = packagedElement−>oclIsKindOf(Specification)),
realize = packagedElement−>oclIsKindOf(Realization),
merge = packagedElement−>oclIsKindOf(PackageMerge)
in realize−>forall(r | exists(s: Specification, m:PackageMerge | spec−>includes(s)
and merge−>includes(m) **and**
m.receivingPackage = r **and** m.mergedPackage = s)

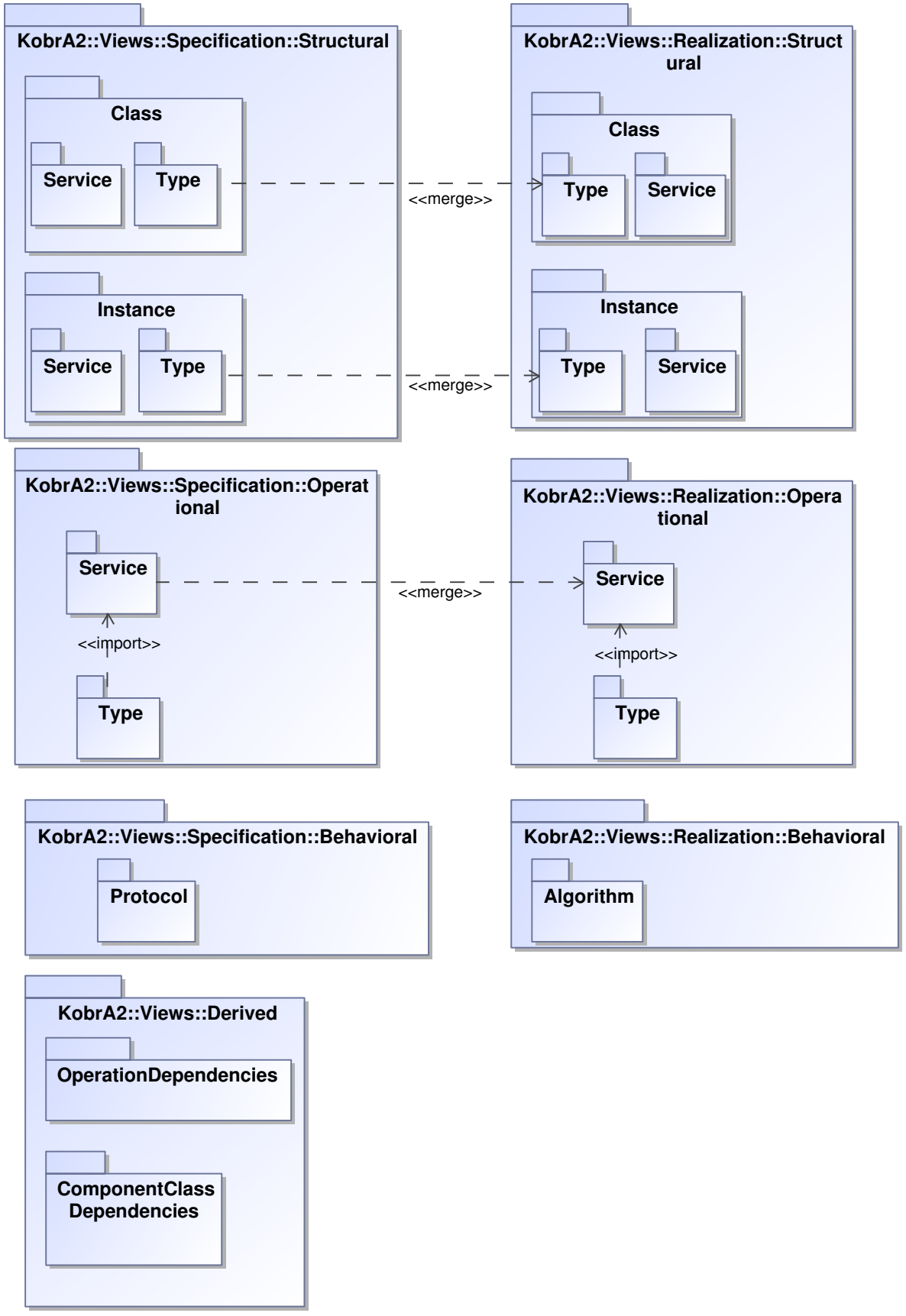

Figure 1.20: Package Dependencies between Views

In future K2 Versions, the views Communication, Sequence and Operation Dependencies will be included.

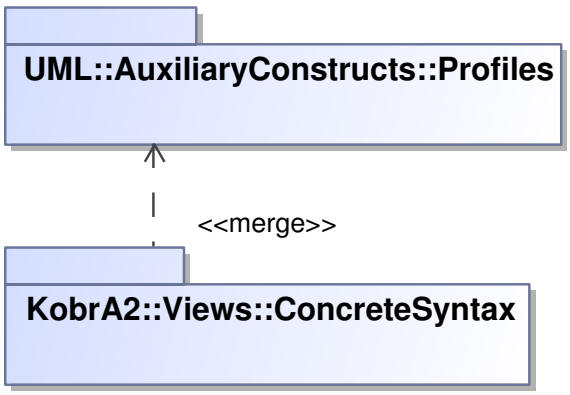

Figure 1.21: Package Dependencies between Views and UML

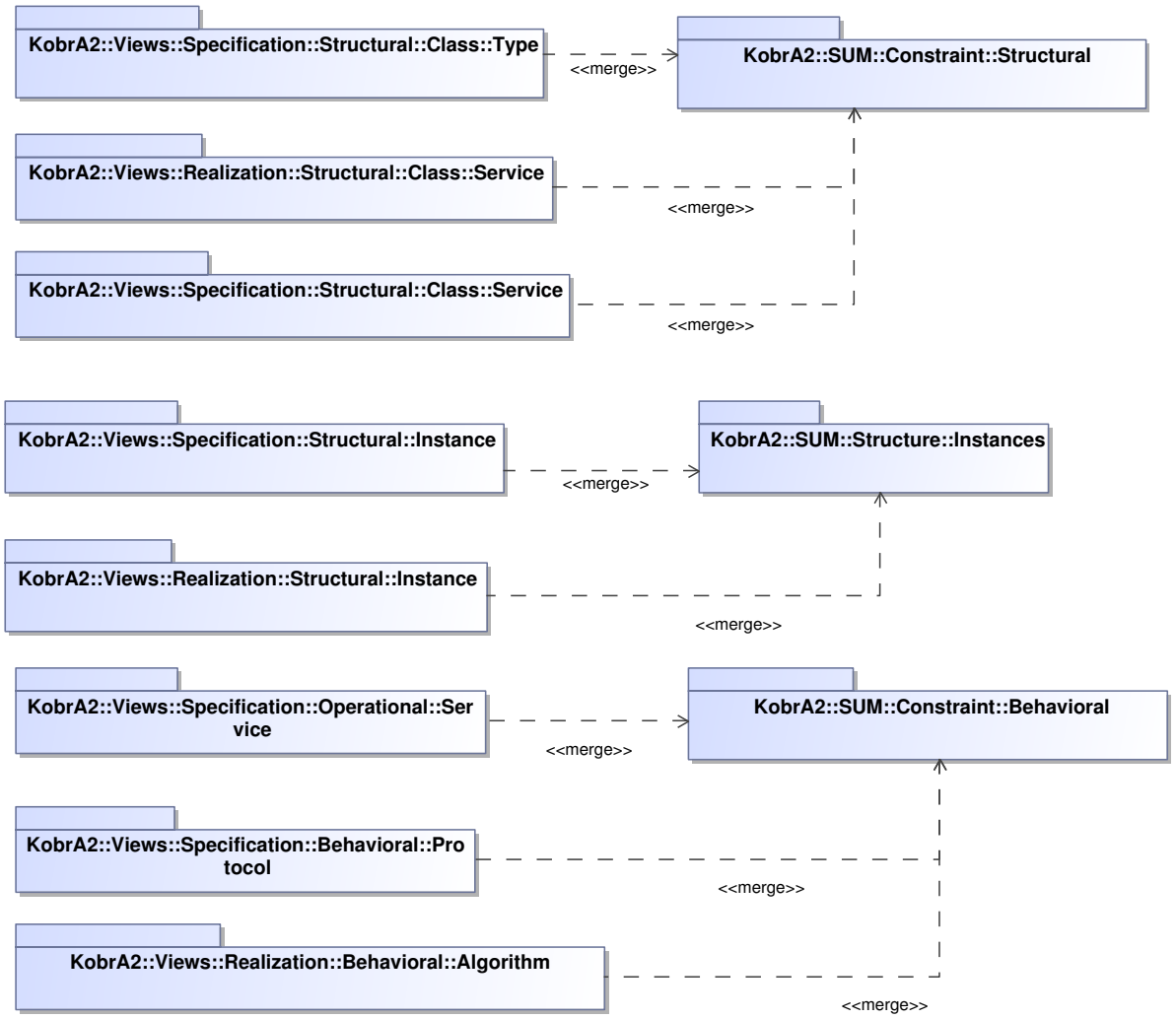

Figure 1.22: Package Dependencies between Views and SUM

1.5.2 Common Packages

Concrete Syntax

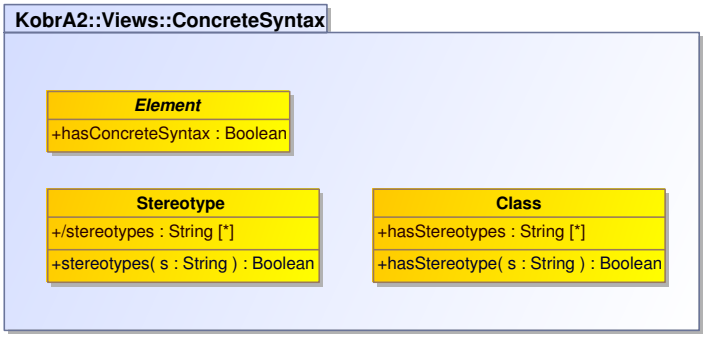

Figure 1.23: Auxiliary Methods for concrete syntax

context Class::hasStereotype(s: **String**): **Boolean**
−− navigate in the profile metamodel from Class to Stereotype
body: extension.ownedEnd.type.name = s

context Stereotype::stereotypes(s: **String**): **Boolean**
−− the other way round
body: extensionEnd.extension.metaclass.name = s

context Class::hasStereotypes : **Set**(**String**)
derived: extension.ownedEnd.type.name

context Stereotype::stereotypes : **Set**(**String**)
derived: extensionEnd.extension.metaclass.name

Subject

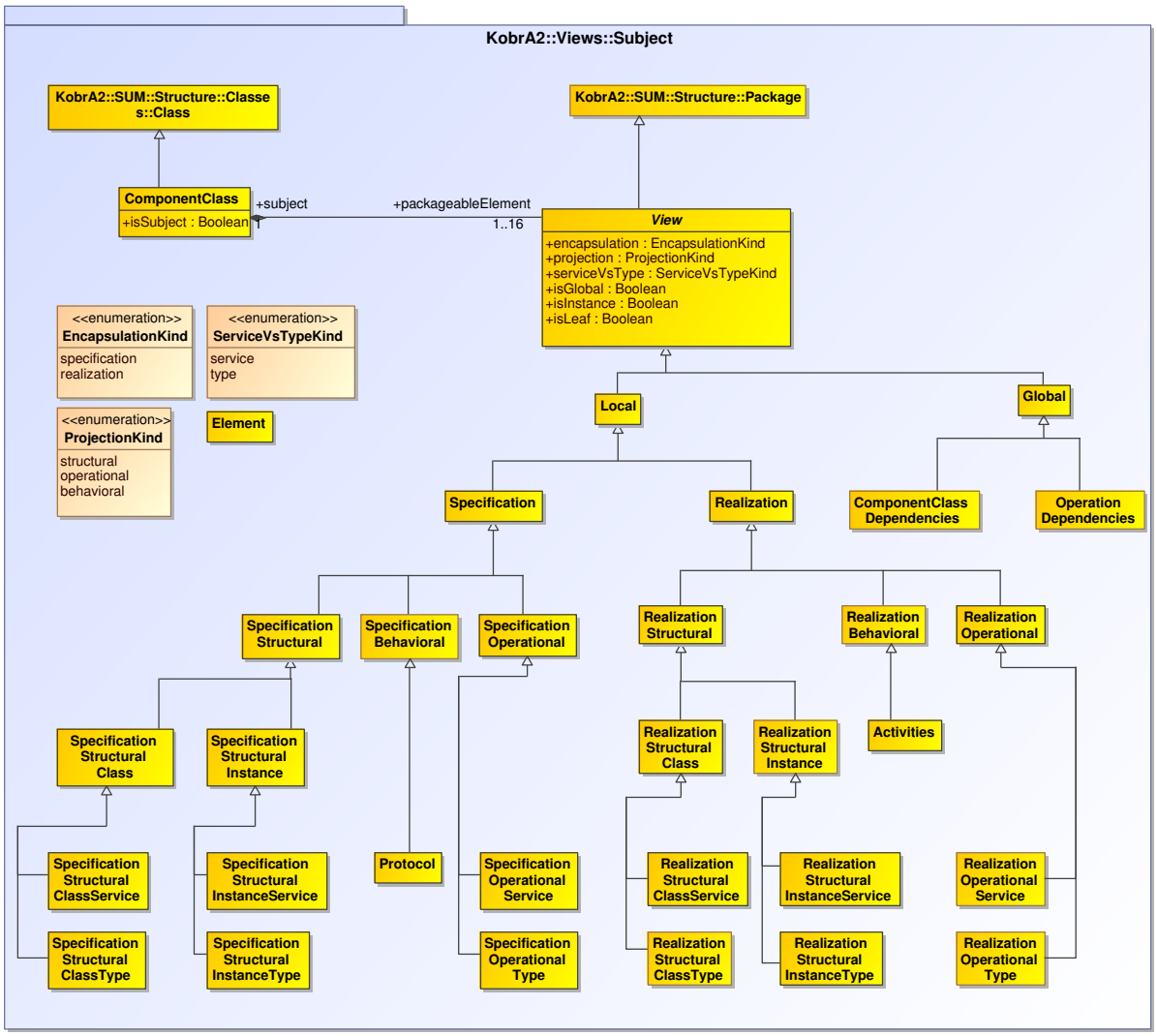

Figure 1.24: "Subject" component and its View-SUM relationship

context SpecificationStructuralService
inv: encapsulation = #specification
inv: projection = #structural
inv: serviceVsType = #service
inv: isInstance = #false
inv: isDerived = #false

context View
-- *a non-leaf view can only package other views that are specializations of itself*
inv: **not** isLeaf **implies** packagedElement = nestedPackaged->select(oclIsKindOf(self))

context ComponentClass
-- *all nested classifiers of the SUM ComponentClass must have at least one abstraction that appears in one of the views of the ComponentClass, except for Stereotypes and derived Views*
-- *packagedElement ... a particular view*
-- *packagedElement.packagedElement ... its contents*

-- *nestedClassifier ... contents (nested elements) of the ComponentClass in the SUM*
-- *nestedClassifier.abstraction ... their abstraction*
-- *nestedClassifier.abstraction.ve ... counterparts in the views of the contents (nested elements) of the ComponentClass of the SUM*
inv: packagedElement->reject(isDerived).packagedElement->reject(oclIsKindOf(Stereotype)) = nestedClassifier.abstraction.ve

-- *all directly or indirectly owned elements of all nested classifiers of the SUM ComponentClass must have at least one abstraction that appears in one of the views of the ComponentClass, and vice versa, except for stereotypes and derived Views*
inv: packagedElement->reject(isDerived).packagedElement->reject(oclIsKindOf(Stereotype)).ownedClosure = nestedClassifier.abstraction.ve.ownedClosure

def: associationEnds : **Set**(Property) = ownedAttribute->select(association->notEmpty())
-- *get all outgoing associations (even n-ary)*
def: outgoingAssociations : **Set**(Association) =
association->select(navigableOwnedEnd->intersection(associationEnds)->isEmpty())
->reject(oclIsKindOf(Nests))->asSet()
-- *source elements of all outgoing associations*
def: outgoingAssociationSources : **Set**(Property) = outgoingAssociations().memberEnd->intersection(associationEnds)

context Element
-- *all directly or indirectly owned elements*
def: ownedClosure : **Set**(Element) =
if oclIsKindOf(Activity)
then ownedElement->union(ownedElement.ownedClosure)->union(
subactivityClosure)->asSet()
else ownedElement->union(ownedElement.ownedClosure)->asSet()
endif

1.5.3 Specification - Structural - Class

Service View

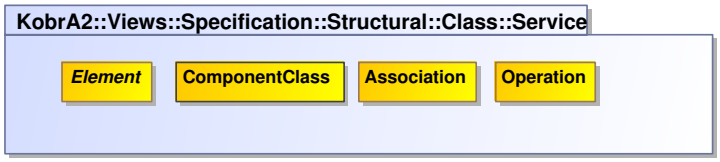

Figure 1.25: ServiceView

In comparison to the SUM, we don't need InstanceSpecification, Slot, AssociationClass. Also, we don't need any behavior at all. We only have Acquires, there's no need for Association.

context Element
-- *Elements (and its subclasses) that are allowed*
inv allowedElements:
let kindAllowed = **Set**{TypeElement, StructuralBehavioralElement, StructuralConstraintElement, CommonConstraintElement} − **Set**{EnumerationLiteral, Class, Usage, AssociationClass, Nests}

context ComponentClass
−− *only public attributes in this view, no nested classifiers, no protocol*
inv: ownedAttribute−>forAll(visibility=#public)
−− *only public Operations are allowed in the specification*
inv: ownedOperation−>forAll(visibility=#public)
inv: nestedClassifier −>isEmpty()
inv: protocol−>isEmpty()
−−

context Association
−− *only allow Aquires associations or ones that are associtated with a Powertype*
−− *TBD: add oclIsKindOf(Creates)?*
inv: oclIsKindOf(Acquires) **or** endType−>exists(powertypeExtent−>notEmpty())

context Operation
−− *only the signature of the Operation is shown, not its behavior (role name "method" refers to the Activities of the operation), or dependencies*
inv: method−>union(precondition)−>union(body)−>union(postcondition)−>union(using) −>union(used)−>isEmpty()

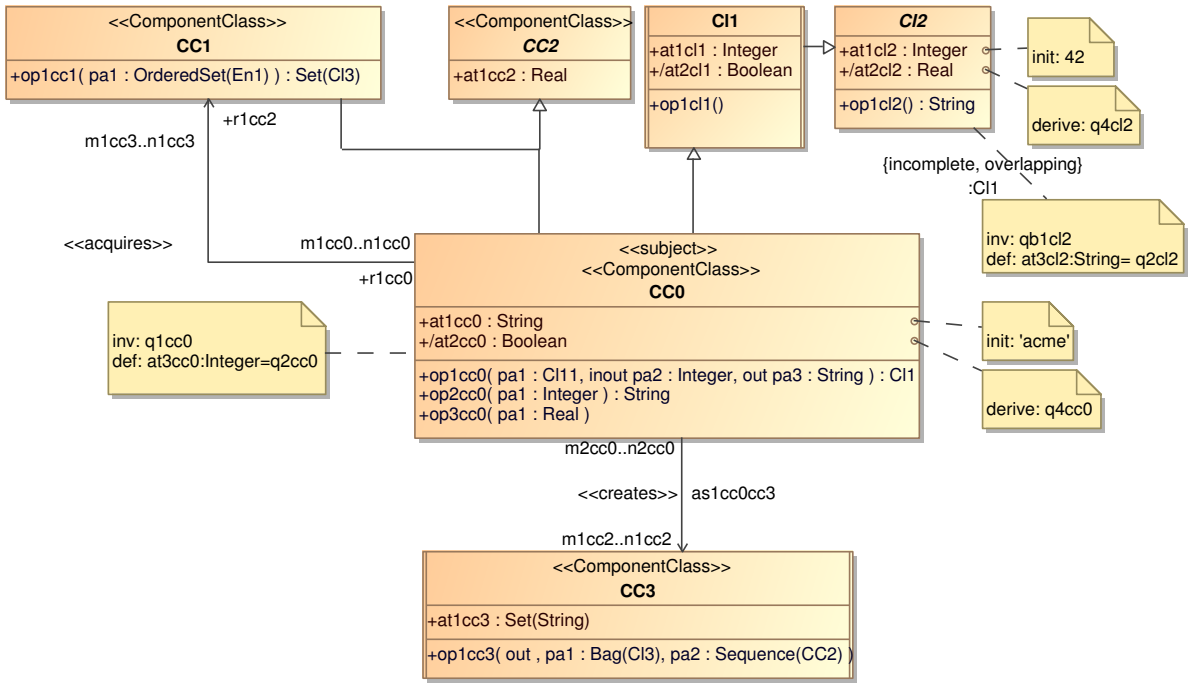

Figure 1.26: Prototypical Specification Structural Class Service View

Type View

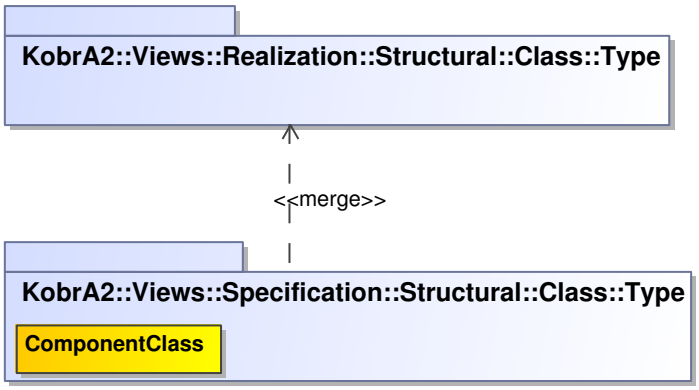

Figure 1.27: Type View

context ComponentClass
inv: oclIsTypeOf(ComponentClass) **implies** hasStereotypes−>includes('subject')
−− *only public attributes in this view, no nested classifiers, no protocol*
inv: ownedAttribute−>forAll(visibility=#public)
−− *only public Operations are allowed in the specification*
inv: ownedOperation−>forAll(visibility=#public)

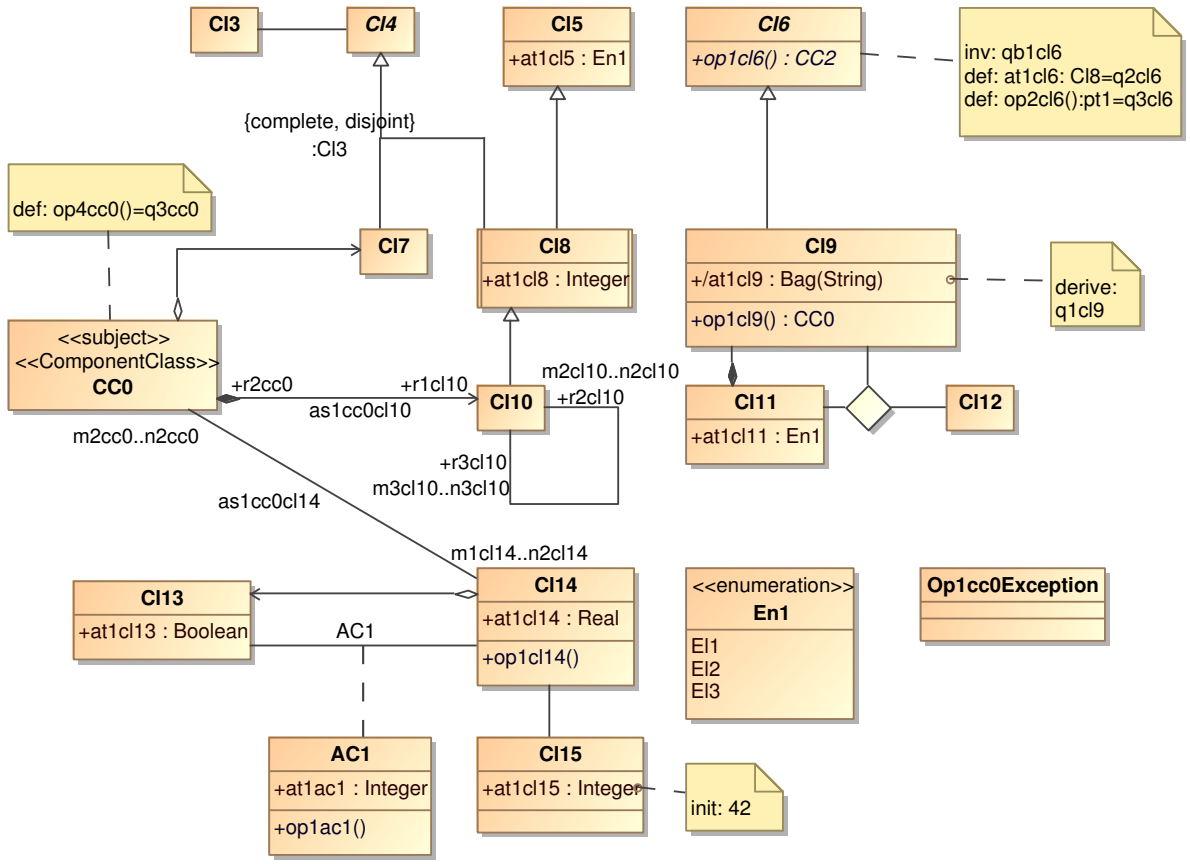

Figure 1.28: Prototypical Specification Structural Class Type View

1.5.4 Specification - Structural - Instance

Service View

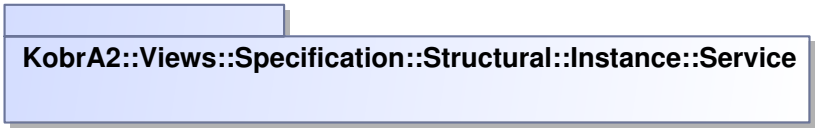

Figure 1.29: Specification Service Instance View

context Element
−− Elements (and its subclasses) that are allowed
inv allowedElements:
let kindAllowed = **Set**{InstanceElement, StructuralBehaviorElement} − **Set**{Operation, Parameter, Object, NestsLink, ObjectLink}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context KobrA2::Specification::Structural::Instance::Service
inv: packagedElements−>select(oclIsKindOf(ComponentObject) **and** hasStereotypes−>includes('subject'))−>size() = 1

context ComponentObject
inv: hasStereotypes−>includes('componentObject')

context Link
inv: oclIsKindOf(AcquiresLink) **or** oclIsKindOf(CreatesLink)

context Slot
inv: definingFeature. visibility = #public

context AcquiresLink
inv: hasStereotypes−>includes('acquiresLink')

context CreatesLink
inv: hasStereotypes−>includes('createsLink')

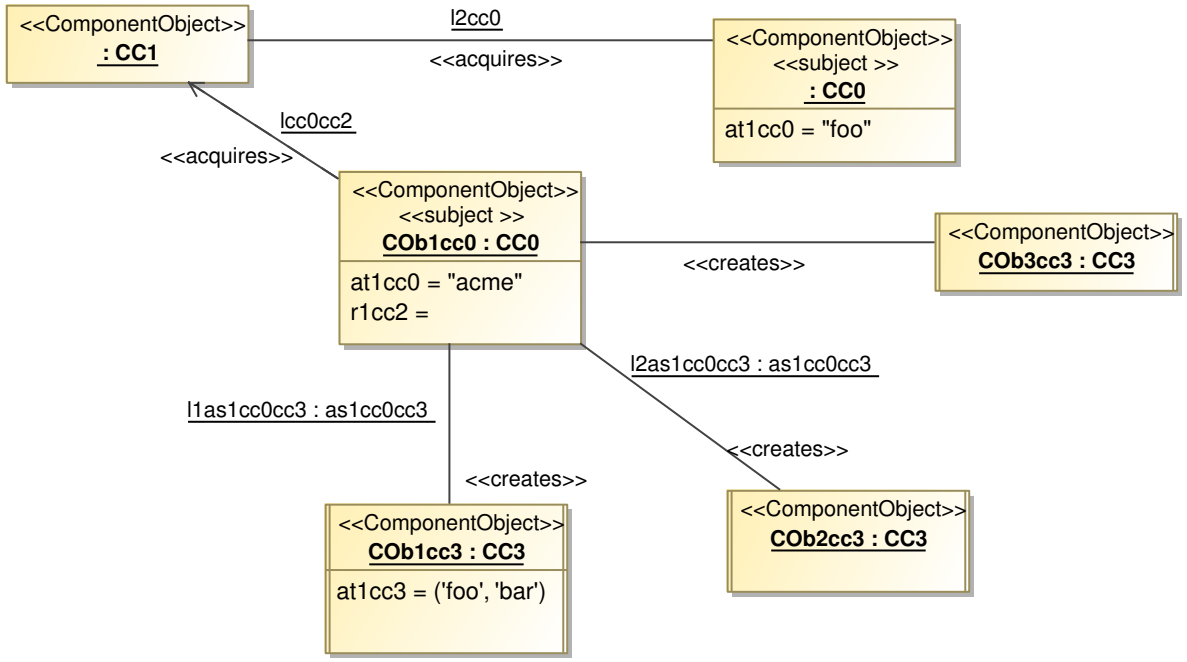

Figure 1.30: Prototypical Specification Structural Instance Service View

Type View

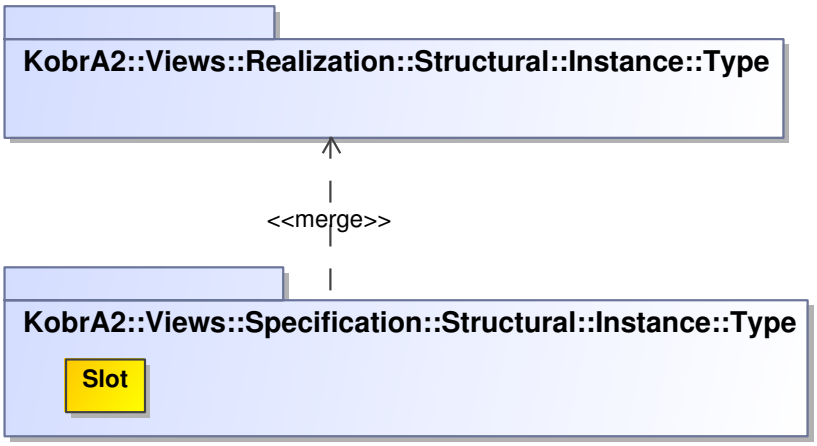

Figure 1.31: Specification Structural Type Instance View

context Element
−− Elements (and its subclasses) that are allowed
inv allowedElements:
let kindAllowed = **Set**{InstanceElement, StructuralBehaviorElement} − **Set**{Operation, Parameter, AcquiresLink, NestsLink}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context KobrA2::Specification::Structural::Instance::Type
inv: packagedElements−>select(oclIsKindOf(ComponentObject) **and** hasStereotypes−>includes('subject'))−>size() = 1

context Slot
inv: definingFeature. visibility = #public

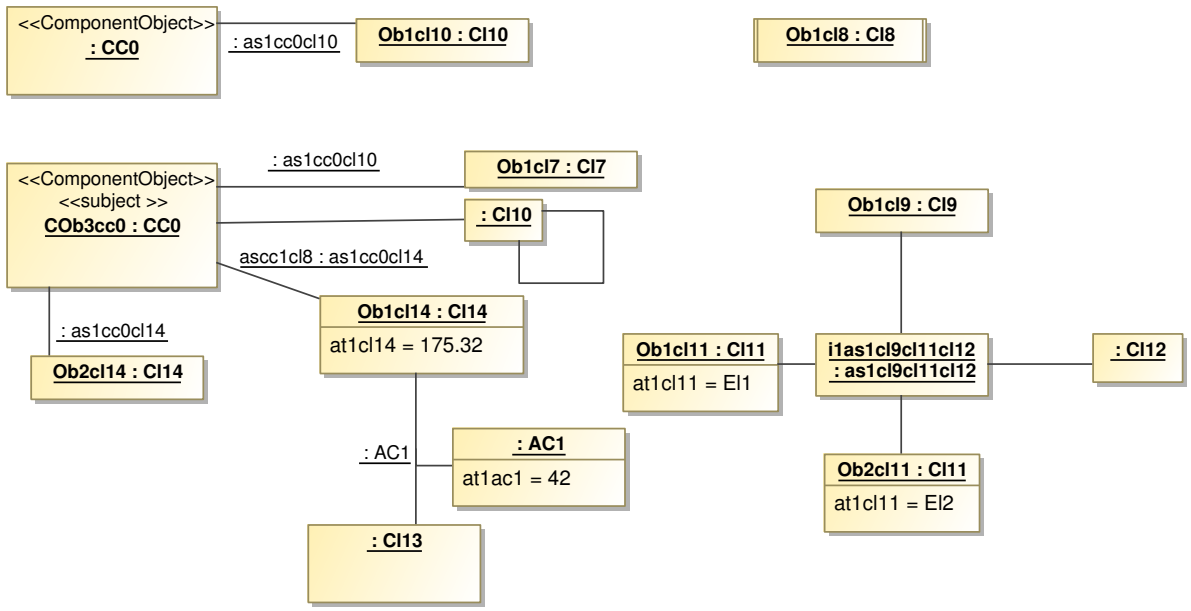

Figure 1.32: Prototypical Specification Structural Instance Type View

1.5.5 Specification - Operational

Service Operation View

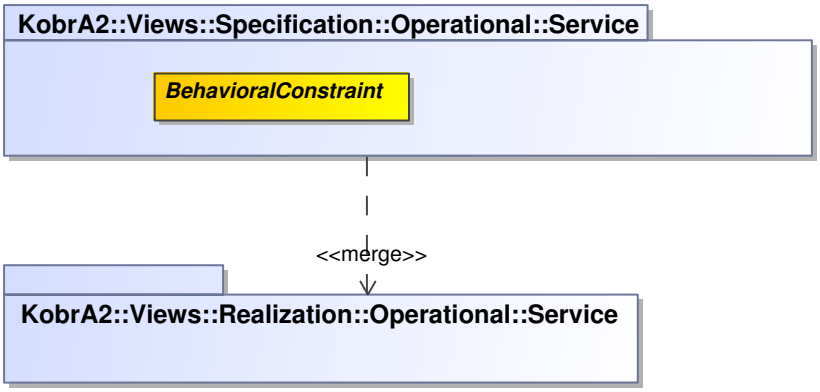

Figure 1.33: Service Operation View

context BehavioralConstraint
inv: constrainedElement. visibility = #public

This view contains a textual list of Operations of the subject ComponentClass and the Classes in the Structrual Class Service view with their pre- and postconditions, e.g.:

context CC0::op1cc0(pa1 : Cl1, inout pa2 : **Integer**, out pa3 : **String**) : Cl1
pre: qb1op1cc0
post: –– *neither post nor body,*
body: –– *because we have an activity diagram for the behavior*

context CC0::op2cc0(pa1 : **Integer**) : **String**
pre: −−
post: p1op2cc0
body: −−

context CC0::op3cc0(pa1 : **Real**)
pre: −−
post: −−
body: q1op3cc0

context CC1::op1cc1(pa1: OrderedSet(En1)): **Set**(Cl3)
pre: q1op1cc1
post: −−
body: 12op1cc1

context Cl1::op1cl1()
pre: q1op1cl1
post: p1op2cl22
body: −−

context Cl2::op1cl2() : **String**
pre: q1op1cl2
post: −−
body: q2op1cl2

def: op2cl2() = q3cl2

Type Operation View

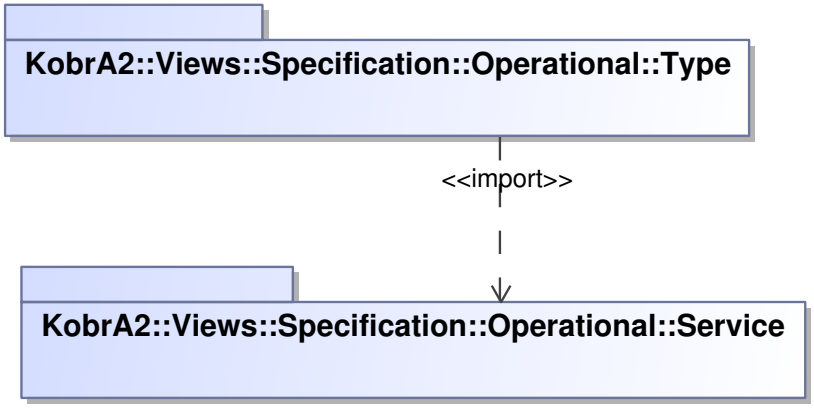

Figure 1.34: Type Operation View

The elements in the Type Operation view are the same as in the Service Operation view. A prototypical Specification Operational Type view follows.

context Cl6::op1cl6() : CC2
pre: qb1op1cl6
post: p1op1cl6
body: −−

context Cl9::op1cl9() : CC0
pre: qb1op1cl9
post: −−
body: q1op1cl9

context Cl14:op1cl14()
pre: −−
post: −−
body: q1op1cl14

context AC1::op1ac1()
pre: −−
post: b1op2ac1
body: −−

1.5.6 Specification - Behavioral

Protocol View

Figure 1.35: Protocol View

context Element
−− *Elements (and its subclasses) that are allowed*
inv allowedElements:
let kindAllowed = **Set**{TypeElement, StructuralBehaviorElement,
BehavioralConstraintElement, CommonConstraintElement, ProtocolElement} − **Set**{
OpDef, Property, Generalization, GeneralizationSet, Usage}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context Association
−− *only allow associations that are AssociationClass to type parameters of operations*
inv: oclIsKindOf(AssociationClass)

We don't need the concrete syntax of Region as we only allow a single region in our Protocol-StateMachine.

context Operation
−− *avoid Activity specifying the Algorithm of the Operation to appear in the Protocol View*
inv: ComponentClass−>union(method)−>union(postcondition)−>isEmpty()

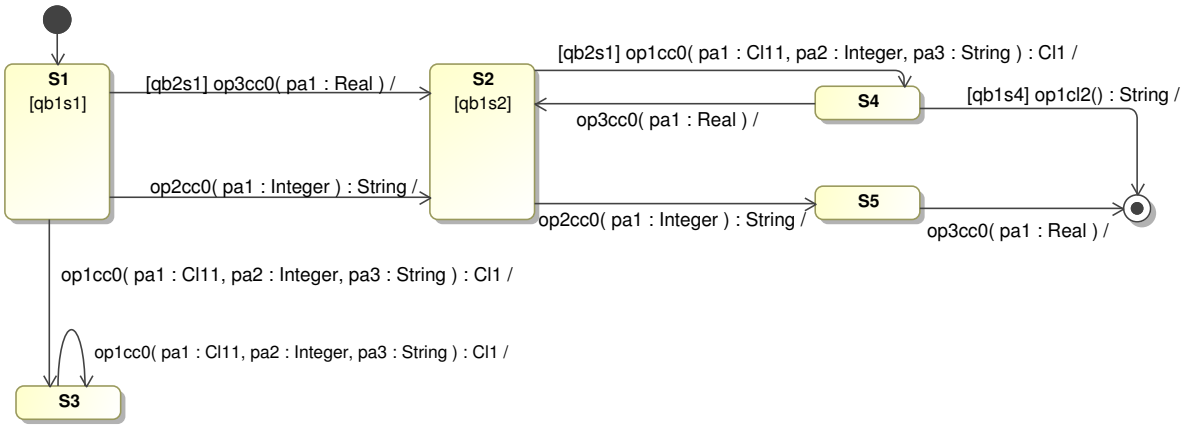

Figure 1.36: Prototypical Protocol View

1.5.7 Realization - Structural - Class

In general, in the realization, only new information that is not already in the specification is displayed - however, a tool might allow the user to toggle between viewing all information (including the specification) and only the new information not in the specification.

Service View

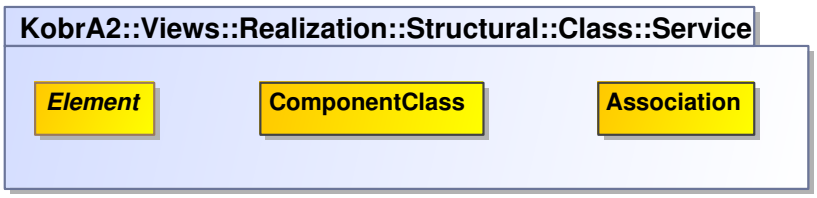

Figure 1.37: Realization Structural Class Service View

context Element
-- *Elements (and its subclasses) that are allowed*
inv allowedElements:
let kindAllowed = **Set**{TypeElement, StructuralBehavioralElement, StructuralConstraintElement, CommonConstraintElement} − **Set**{EnumerationLiteral, Class, Usage, AssociationClass}

context ComponentClass
inv: hasStereotypes−>includes('componentClass')
-- *suppress associations from the subject to unnested ComponentClasses or Classes*
inv: isSubject **implies** ownedAttribute−>reject(outgoingAssociationSources)
inv: protocol−>isEmpty()
-- *nested ComponentClasses and Classes only have public features*
inv: componentClass.isSubject **implies**
ownedAttribute−>union(ownedOperation)−>forAll(visibility=#public)

context Association
inv: oclIsKindOf(Acquires) **or** oclIsKind(Nests) **or** oclIsKindOf(Creates)

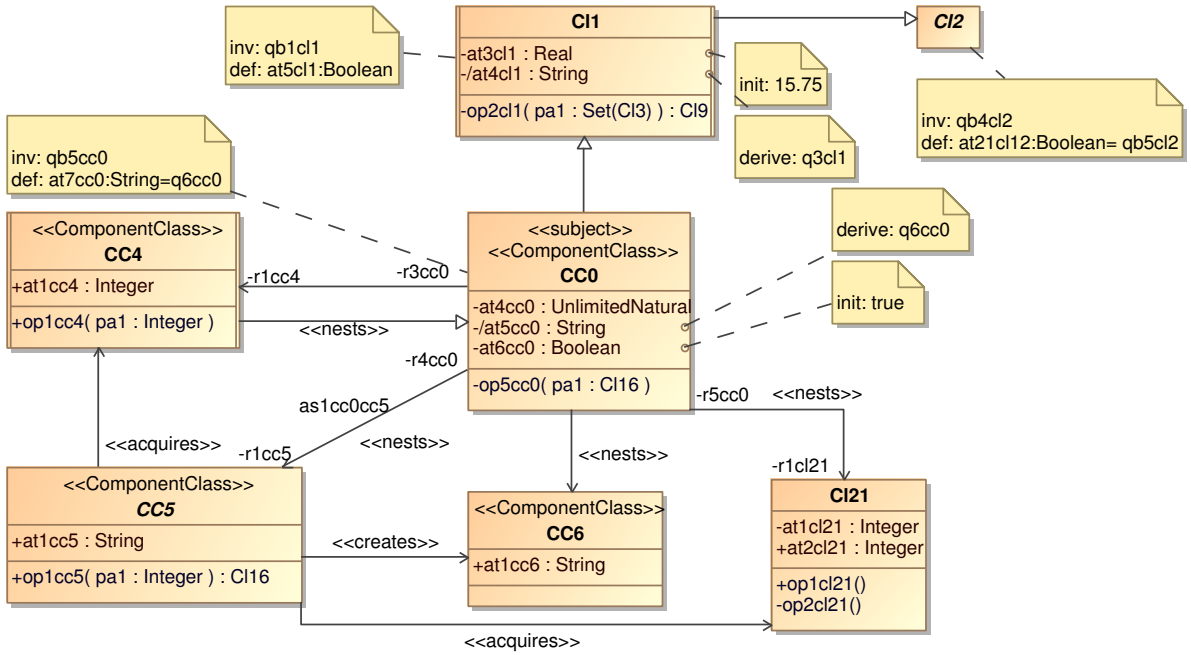

Figure 1.38: Prototypical Realization Structural Class Service View

Type View

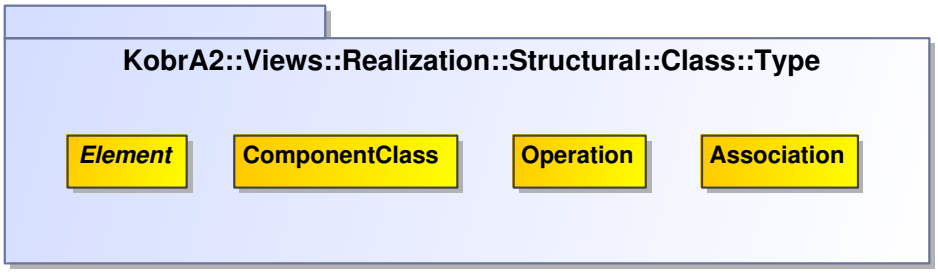

Figure 1.39: Realization Structural Class Type View

context Element
-- *Elements (and its subclasses) that are allowed*
inv allowedElements:
let kindAllowed = **Set**{TypeElement, StructuralBehavioralElement, StructuralConstraintElement, CommonConstraintElement} − **Set**{Acquires, Usage, Nests }

context ComponentClass
inv: oclIsTypeOf(ComponentClass) **implies** hasStereotypes−>includes('subject')

context Association
-- *prohibit Acquires and Nests associations*
inv: **not Set**{Acquires, Nests}−>exists(e: Element | self.oclIsKindOf(e))

context Operation
-- *only the signature of the Operation is shown, not its behavior (role name "method" refers to the Activities of the operation), or dependencies*

inv: method−>union(precondition)−>union(body)−>union(postcondition)−>union(using) −>union(used)−>isEmpty()

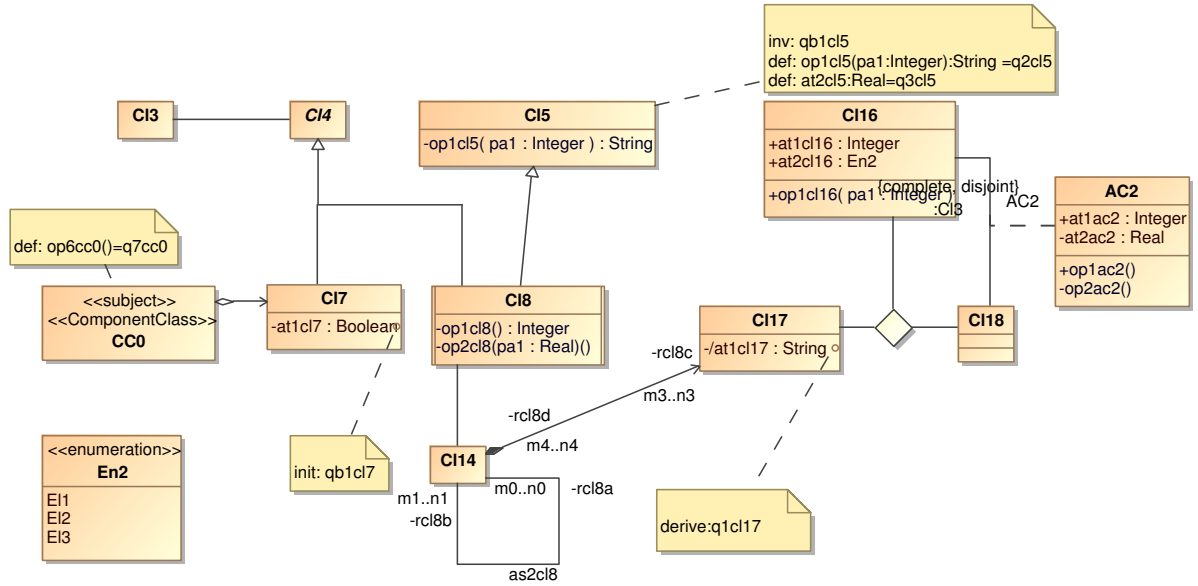

Figure 1.40: Prototypical Realization Structural Class Type View

1.5.8 Realization - Structural - Instance

Service View

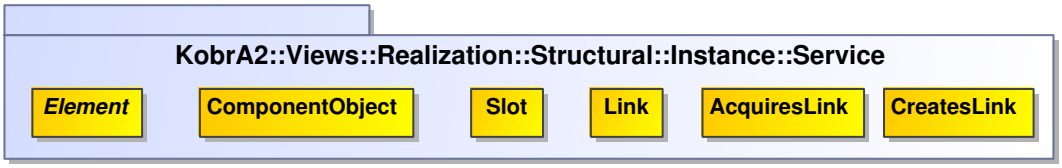

Figure 1.41: Realization Structural Instance Service View

context Element
−− *Elements (and its subclasses) that are allowed*
inv allowedElements:
let kindAllowed = **Set**{InstanceElement, StructuralBehaviorElement} − **Set**{Operation, Parameter, Object, ObjectLink}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context KobrA2::Realization::Structural::Instance::Service
inv: packagedElements−>select(oclIsKindOf(ComponentObject) **and** hasStereotypes−> includes('subject'))−>size() = 1

context ComponentObject
inv: hasStereotypes−>includes('componentObject')
−− *outgoing links of the subject component are prohibited*
−− *only AcquiresLinks between nested component objects*
−− *aes: slots that are "instances" of association ends*
−− *aesc: outgoing links*

−− ccoas: outgoing association ends of the component that is the classifier of the component object
inv: let cc = classifier , ccoas = cc.outgoingAssociationSources, s = slot, aes = s−>select(definingFeature.association−>notEmpty()), aesc = aes.classifier
if isSubject **then** aesc−>asSet()−>intersection(ccoas−>asSet())−>isEmpty()
else aesc−>forAll(oclIsKindOf(AcquiresLink))
endif

context Slot
−− don't show private features of the nested components
inv: **not** isSubject **implies** definingFeature. visibility = #public

context AcquiresLink
inv: hasStereotypes−>includes('acquiresLink')

context CreatesLink
inv: hasStereotypes−>includes('createsLink')

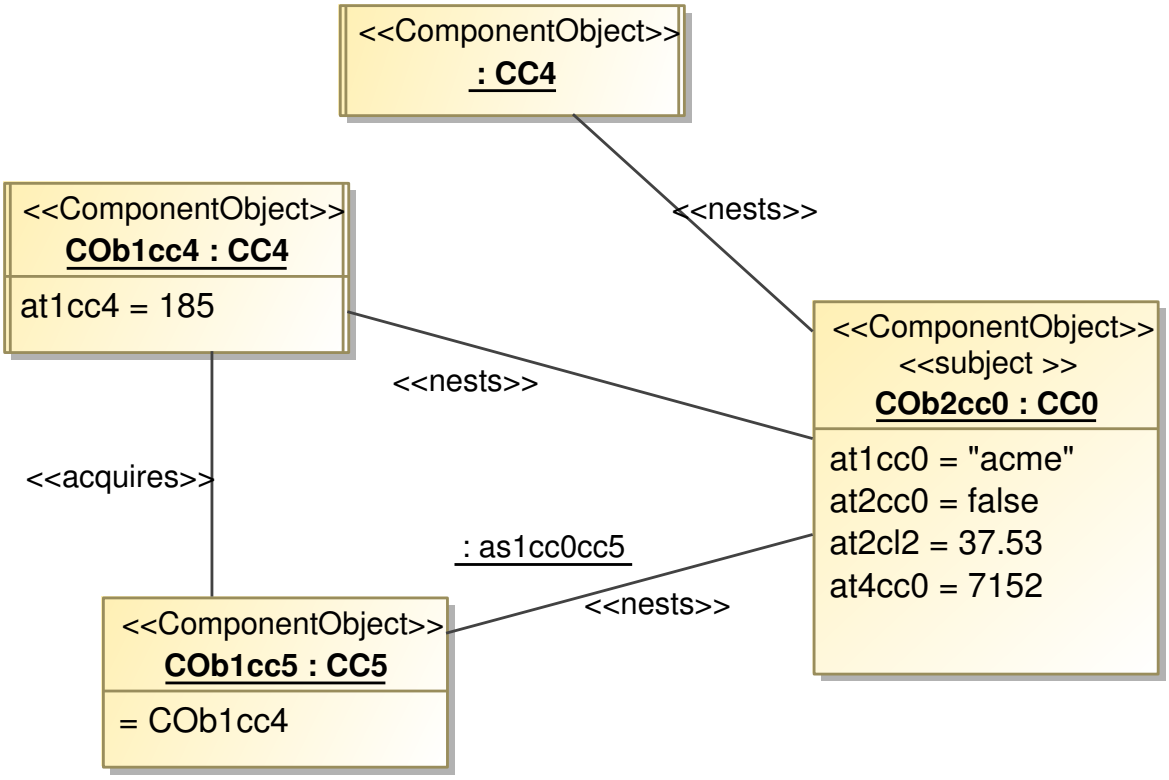

Figure 1.42: Prototypical Realization Structural Instance Service View

Type View

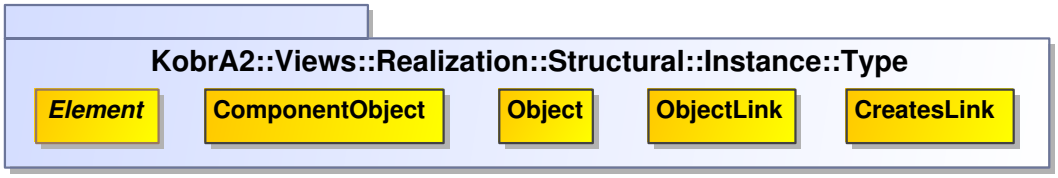

Figure 1.43: Realization Structural Instance Type View

context Element
–– Elements (and its subclasses) that are allowed
inv allowedElements:
let kindAllowed = **Set**{InstanceElement, StructuralBehaviorElement} − **Set**{Operation, Parameter, AcquiresLink, NestsLink}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context KobrA2::Realization::Structural::Instance::Type
inv: packagedElements−>select(oclIsKindOf(ComponentObject) **and** hasStereotypes−>includes('subject'))−>size() = 1

context ComponentObject
inv: hasStereotypes−>includes('componentObject')

context Object
inv: hasStereotypes−>includes('object')

context ObjectLink
inv: hasStereotypes−>includes('objectLink')

context CreatesLink
inv: hasStereotypes−>includes('createsLink')

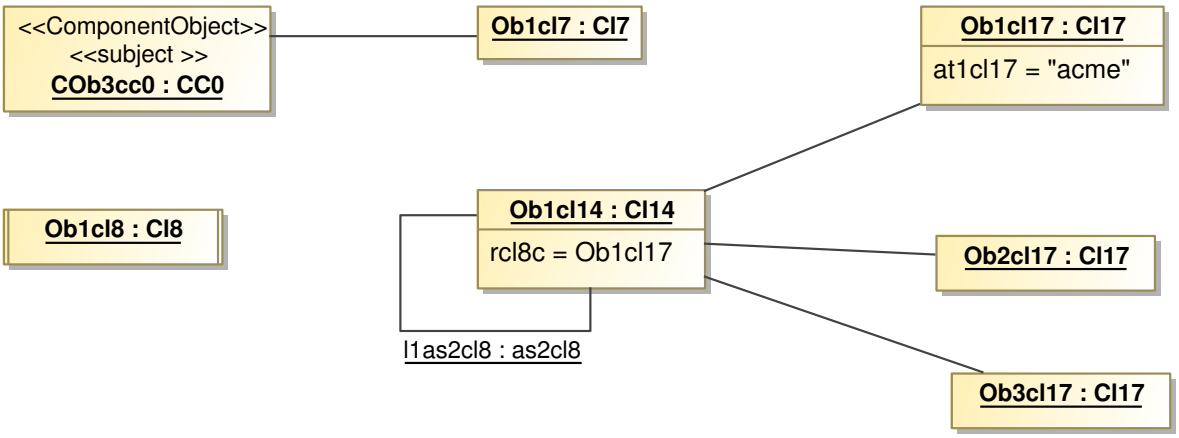

Figure 1.44: Prototypical Realization Structural Instance Type View

1.5.9 Realization - Operational

Service View

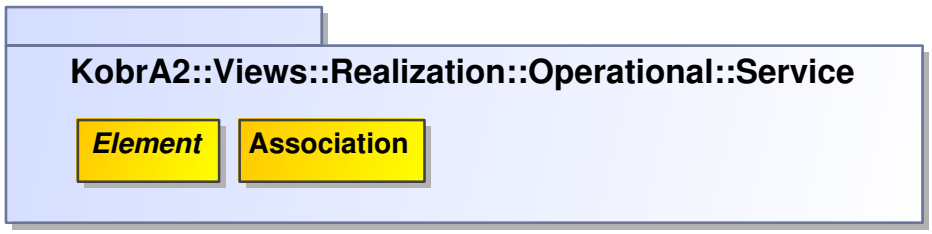

Figure 1.45: Realization Operational Service View

context Element
−− *Elements (and its subclasses) that are allowed*
inv allowedElements:
let kindAllowed = {TypeElement, StructuralBehavioralElement, BehavioralConstraintElement, CommonConstraintElement} − **Set** {Property, Generalization, GeneralizationSet, Usage, State, ProtocolTransition, ActionInOcl, ActivityEdge}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context Association
−− *only allow associations that are AssociationClass to type parameters of*
inv: oclIsKindOf(AssociationClass)

A prototypical Realization Operational Service View follows.

context CC0::op5cc0(pa1 : Cl16)
pre: qb1op5cc0
post: −−
body: q2op5cc0

context CC4::op1cc4(pa1 : **Integer**)
pre: −−
post: −−
body: q1op1cc4

context CC5::op1cc5(pa1 : **Integer**) : Cl16
pre: qb1op1cc5
post: −−
body: −−

context Cl1::op2cl1(pa1: **Set**(Cl3)) : Cl9
pre: qb1op2cl1
post: p1op2cl1
body: −−

context Cl21:op1cl21()
pre: −−
post: p1op1cl21
body: −−

Type View

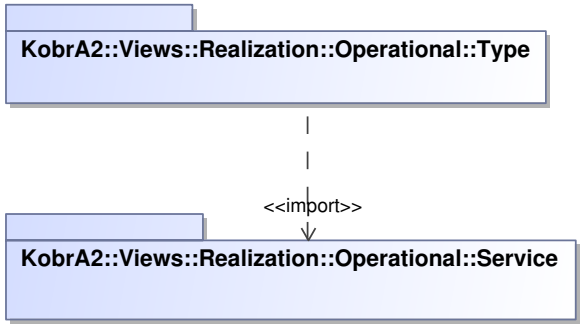

Figure 1.46: Realization Operational Type View

A prototypical Realization Operational Type view follows.

context Cl5::op1cl5(pa1 : **Integer**) : **String**
pre: qb1op1cl5
post: −−
body: ab2op1cl5

context Cl8::op1cl8() : **Integer**
pre: qb1op1cl8
post: p1op1cl8
body: −−

context Cl8::op2cl8(pa1 : **Real**)
pre: −−
post: −−
body: qb2op2cl8

context Cl16::op1cl16(pa1 : **Integer**)
pre: −−
post: p1op1cl16
body: −−

context AC2::op1ac2()
pre: −−
post: p1op1ac2
body: −−

context AC2::op2ac2()
pre: −−
post: −−
body: qb1op2ac2

1.5.10 Realization - Behavioral

Algorithmic View

Figure 1.47: Realization Behavioral Algorithmic View

context Element
−− Elements (and its subclasses) that are allowed
inv allowedElements:
let kindAllowed = **Set**{TypeElement, StructuralBehavioralElement,
CommonConstraintElement, BehavioralConstraintElement, ActivityElement} − **Set**{
OpDef, Property, Generalization, GeneralizationSet, Usage}
in kindAllowed−>exists(e: Element | self.oclIsKindOf(e))

context Association
−− only allow associations that are AssociationClass to type parameters of operations
inv: oclIsKindOf(AssociationClass)

PostExp and BodyExp are the localPostcondition, whereas PreExp is the guard.

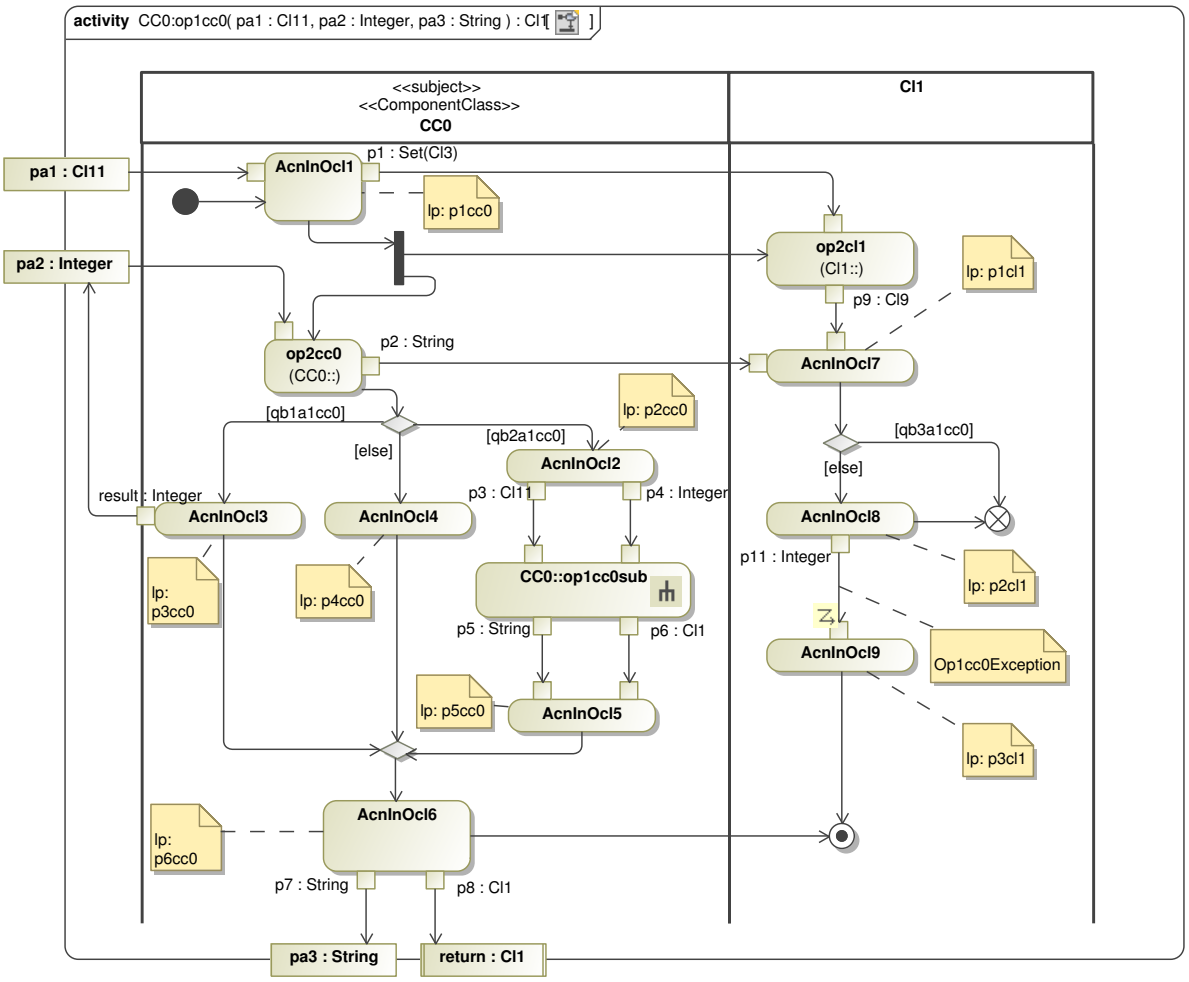

Figure 1.48: Prototypical Realization Behavioral Algorithmic View

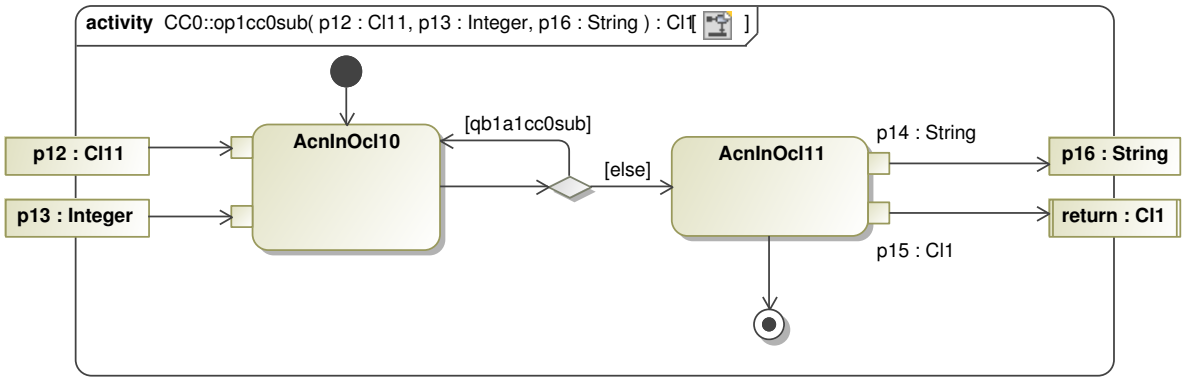

Figure 1.49: Prototypical Realization Behavioral Algorithmic View (sub activity)

1.6 Transformation

This section explains how elements from the SUM are mapped to Views (and vice versa).

1.6.1 Package Dependencies

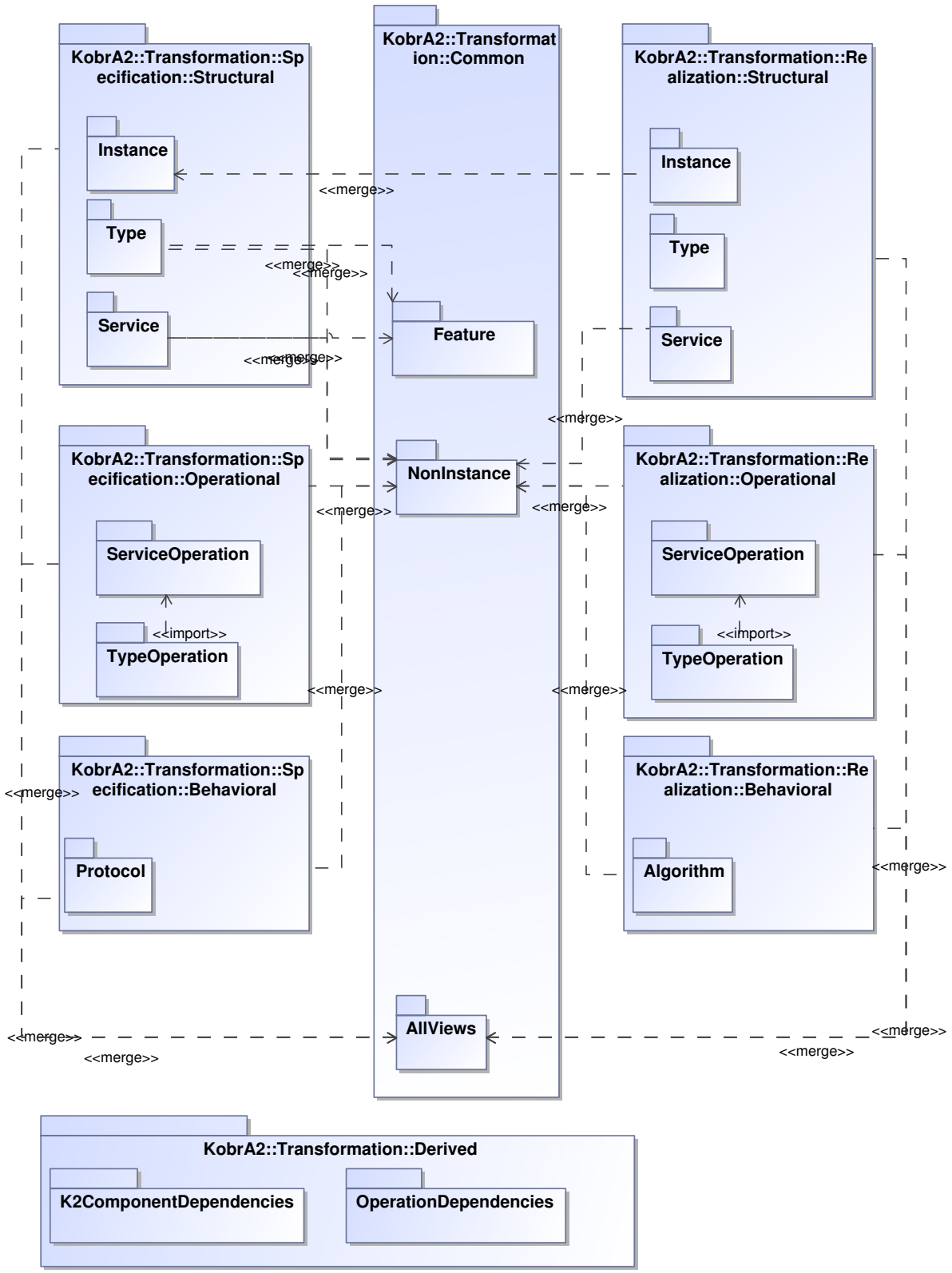

Figure 1.50: Package Dependencies between Transformations

1.6.2 Common Packages

All Views

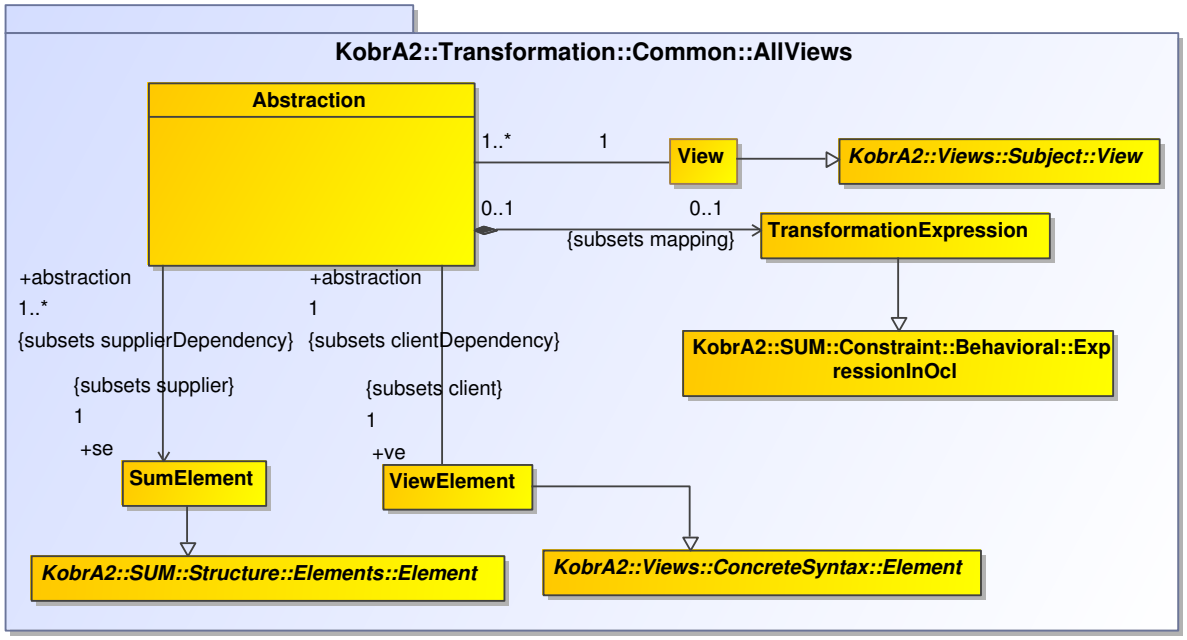

Figure 1.51: Transformation Abstractions

The role name "ve" stands for "ViewElement". The following constraints assume that there is a one to one correspondence between an element in the SUM and in a view. This might change in later versions of KobrA.

context Abstraction
−− only elements of the same type can be related by an abstraction
inv: se.type = ve.type
def: mappedElementType : Type = se.type

context View
−− the subject component must have the stereotype 'subject'
inv: let comp = packagedElement−>select(oclIsKindOf(ComponentClass) **and not** oclIsKindOf(Class))
in comp−>notEmpty()
implies forAll(c: ComponentClass |
c.abstraction.sum = subject
implies c.hasStereotypes−>includes('subject'))

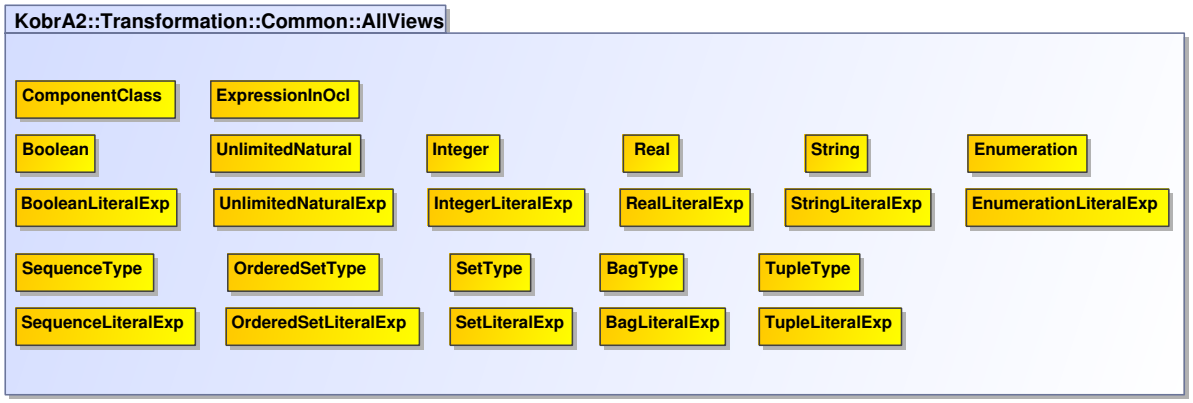

Figure 1.52: Types and Expressions for Transformations

Note: In the UML metamodel, the structural composition of the concrete syntax elements in Attributes and Operation signatures is not isomorphic to the structural composition of the corresponding abstract syntax elements. For example, the type of a parameter is given by a type-expression which mapping to the subelements of the type is not specified. This mapping is implemented fairly uniformly across UML case tools by using the name attribute of the non-primitive types. For primitive types, there is no abstract syntax in UML, only a generic PrimitiveType that specializes DataType but that is not itself specialized by Boolean, String, Integer, UnlimitedNatural. Therefore, in KobrA 2, we don't use UML PrimitiveType but we use the OCL 2 type metamodel. Making appear the concrete syntax of these OCL 2 types as ¡type-expression¿ in UML requires a fairly complex set of OCL metaconstraints that are to be specified. Therefore, in the current version, we wrongly copy typed UML model elements inside our transformations from the SUM to the views instead of transforming types into their type signatures.

The same problem occurs for the defaultValue in attributes and parameter signatures. If such default values are Objects, the common convention in UML case tools is to show only the name of the Object and not an embedded rectangle inside the signature (as would be the result of deriving the concrete syntax by recursively following the abstract syntax structural decomposition and just applying the concrete syntax of the leaf elements of this composition).

context ComponentClass def: stringInSignature:**String** = name

context **Enumeration** def: stringInSignature:**String** = name

context **Boolean** def: stringInSignature:**String** = 'Boolean'

context UnlimitedNatural def: stringInSignature:**String** = 'UnlimitedNatural'

context **Integer** def: stringInSignature:**String** = 'Integer'

context **Real** def: stringInSignature:**String** = 'Real'

context **String** def: stringInSignature:**String** = 'Boolean'

context SetType
def: stringInSignature : **String** = 'Set('.concat(type.stringInSignature).concat(')')

context OrderedSetType
def: stringInSignature : **String** = 'OrderedSet('.concat(type.stringInSignature).concat(')')

```
contex BagType
def: stringInSignature : String = 'Bag('.concat(type.stringInSignature).concat(')')

contex SequenceType
def: stringInSignature : String = 'Sequence('.concat(type.stringInSignature).concat(')')

context TupleType
def: stringInSignature : String
= iterate(p: TupleLiteralPart; result : String = 'TupleType(' |
                let pa = p->attribute
                in concat(pa.name).concat(pa.type.stringInSignature).concat(','))
   .concat(')'))

context ExpressionInOcl def: stringInSignature: String = bodyExpression.stringInSignature

context EnumLiteralExp def: stringInSignature : String = EnumerationLiteral->name

context RealLiteralExp def: :stringInSignature : String = realSymbol.oclAsType(String)

context IntegerLiteralExp
def: stringInSignature : String = intergerSymbol.oclAsType(String)

context UnlimitedNaturalExp def: stringInSignature : String = symbol.oclAsType(String)

context BooleanLiteralExp
def: stringInSignature : String = booleanSymbol.oclAsType(String)

context CollectionLiteralExp
def: stringInSignature : String =
  part->iterate(p: CollectionItem; result : String = kind.oclAsType(String).concat('{') |
                        .concat(item.stringInSignature).concat(','))
            .concat('}')

context TupleLiteralExp
def: stringInSignature : String =
  part->iterate(p: TupleLiteralPart; result : String = 'Tuple(' |
                        let pa = p->attribute
                        in concat(pa.name).concat('␣=␣').concat(slot.value.stringInSignature
                            )
                              .concat(','))
             .concat(')')
```

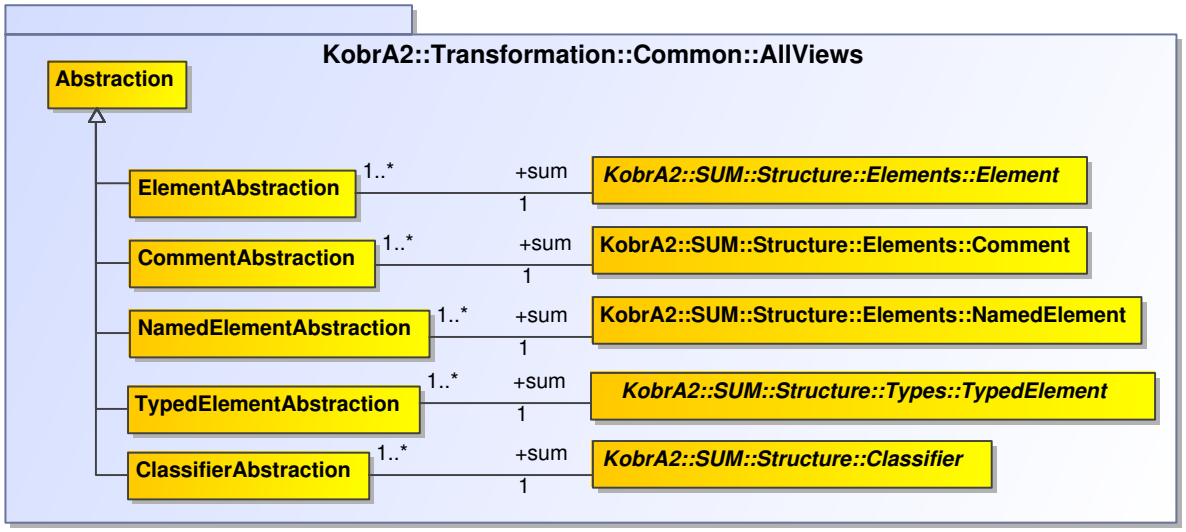

Figure 1.53: Transformations common to all views

context ElementAbstraction
inv: ve.ownedComment = se.ownedComment

context Comment
inv: ve.annotatedElement = se.annotatedElement
inv: ve.body = se.body

context NamedElementAbstraction
inv: ve.name = se.name
inv: ve. visibility = se. visibility
−− fully qualified name is only useful for our derived views
inv: **if** oclIsKindOf(OperationDependencyAbstraction) **or** oclIsKindOf(
ComponentClassDependency)
then ve.qualifiedName = se.qualifiedName
else ve.qualifiedName−>isEmpty() **endif**

context TypedElementAbstraction
inv: ve.type = se.type.stringInSignature

context ClassifierAbstraction
inv: ve.isAbstract = se.isAbstract
inv: ve. generalization = se. generalization

Non-instance Views

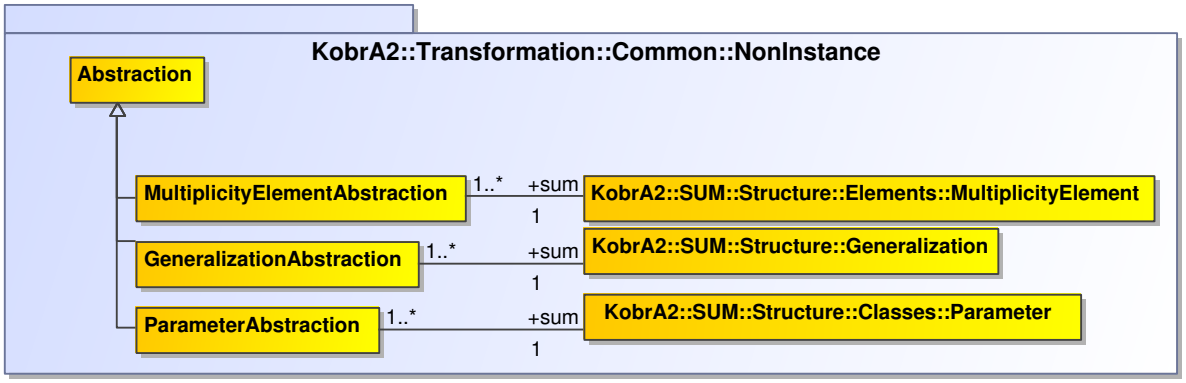

Figure 1.54: Transformations common to all non-instance views

context MultiplicityElementAbstraction
inv: ve.isOrdered = se.isOrdered
inv: ve.isUnique = se.isUnique
-- *these two attributes are derived from upperValue and lowerValue (of type ValueSpecification)*
inv: ve.upper = se.upper
inv: ve.lower = se.lower

context Generalization
inv: ve.general = se.general
inv: ve. specific = se. specific
-- *used in KobrA 2 for conformance of ProtocolStateMachines*
inv: ve. isSubstitutable = se. isSubstitutable

context ParameterAbstraction
inv: ve. direction = se. direction
inv: ve.defaultValue = se.defaultValue.stringInSignature

Feature

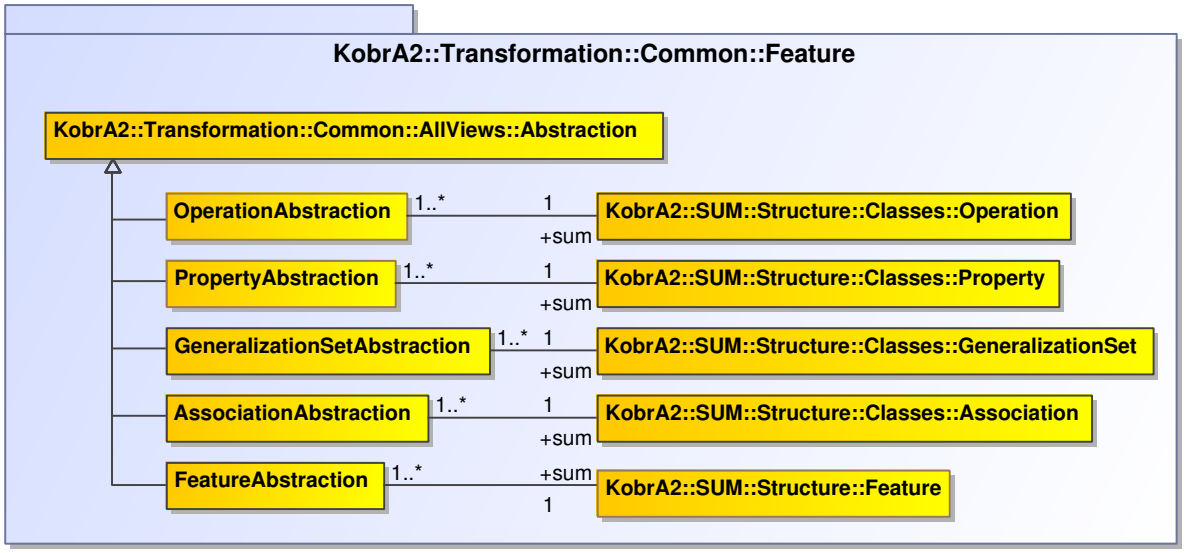

Figure 1.55: Feature Transformations

context PropertyAbstraction
inv: ve. isStatic = se. isStatic
inv: ve.isReadOnly = se.isReadOnly
inv: ve.isDerived = se.isDerived
inv: ve.isDerivedUnion = se.isDerivedUnion
inv: ve. default = se.defaultValue.stringInSignature
inv: ve.aggregation = se.aggregation
inv: ve.isComposite = se.isComposite
inv: ve.opposite=se.opposite
inv: ve.subsettedProperty = se.subsettedProperty
inv: ve.redefinedProperty = se.redefinedProperty
inv: ve.componentClass = se.componentClass
inv: ve.derived = se.derived
inv: ve. init = se. init
inv: ve.propDef = se.propDef

context OperationAbstraction
inv: ve. isStatic = se. isStatic
inv: ve.isReadOnly = se.isReadOnly
inv: ve.ownedParameter = se.ownedParameter
inv: ve.isQuery = se.isQuery
inv: ve.redefinedOperation = se.redefinedOperation

context GeneralizationSetAbstraction
inv: ve.isCovering = se.isCovering
inv: ve. isDisjoint = se. isDisjoint
inv: ve. generalization = se. generalization
inv: ve.powerType = se.powerType

context AssociationAbstraction
inv: ve.memberEnd = se.memberEnd
inv: ve.navigableOwnedEnd = se.navigableOwnedEnd

−− not copying the "ownedEnd / owningAssociation" Metassocation might cause problems when using existing UML tools
inv: ve.oclIsKindOf(Acquires) **implies** ve.hasStereotypes−>includes('acquires')
inv: ve.oclIsKindOf(Nests) **implies** ve.hasStereotypes−>includes('nests')
−− only Acquires associations are allowed in specification service views
inv: (ve.encapsulation = #specification **and** ve.serviceVsType = #service) **implies** ve.oclIsKindOf(Acquires)
−− only Acquires and Nests associations are allowed in realization service views
inv: (ve.encapsulation = #realization **and** ve.serviceVsType = #service) **implies** (ve.oclIsKindOf(Acquires) **or** ve.oclIsKindOf(Nests))
−− only regular associations (neither Acquires nor Nests) are allowed in type views
inv: ve.serviceVsType = #type **implies not** ve.oclIsKindOf(Acquires) **and not** ve.oclIsKindOf(Nests)

context FeatureAbstraction
inv: ve. isStatic = se. isStatic

1.6.3 Specification - Structural

Service View

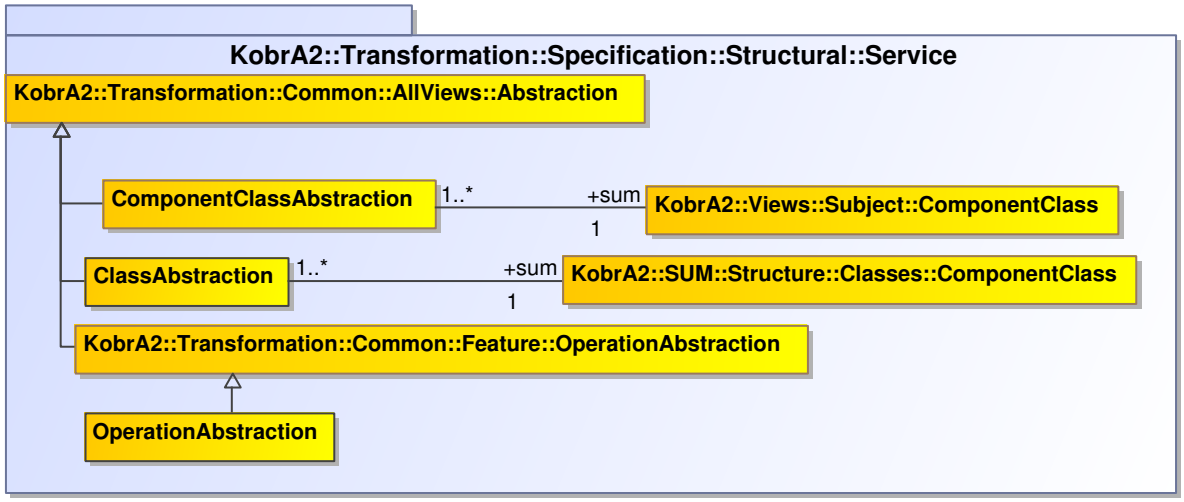

Figure 1.56: Transformation to Specification Service View

context ComponentClassAbstraction
inv: ve.superClass = se.superClass
inv: ve.ownedAttribute = se.ownedAttribute−>select(visibility=#public)
inv: ve.ownedOperation= se.ownedOperation−>select(visibility=#public)
inv: ve.**inv** = se.**inv**
inv: ve.hasStereotypes−>includes('componentClass')

−− just needed to copy powertypeExtent that is only allowed for class
context ClassAbstraction
inv: ve.powertypeExtent = se.powertypeExtent

context OperationAbstraction
inv: ve.componentClass = se.componentClass

In transformations of Operation and Property, isStatic of Feature has to be copied. For Property, also isReadOnly has to be copied.

Type View

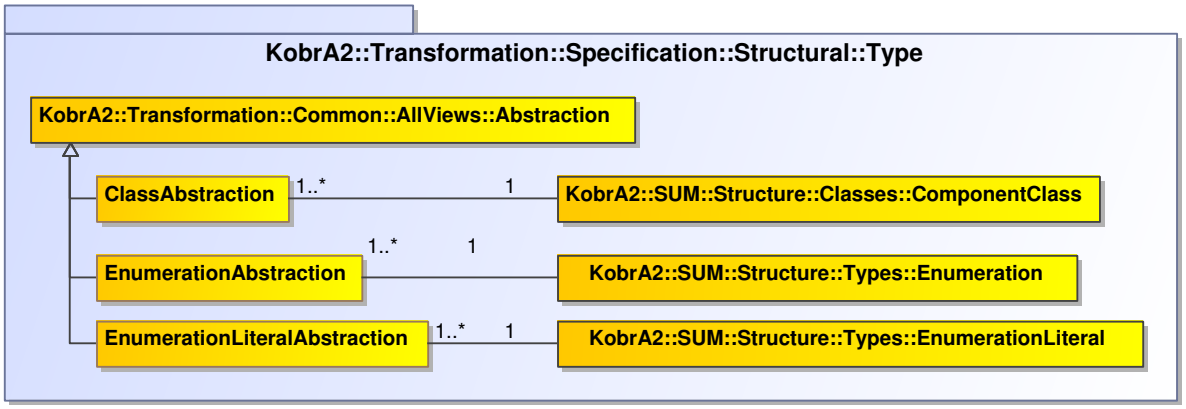

Figure 1.57: Transformation to Specification Type View

context ClassAbstraction
inv: ve.ownedAttribute = se.ownedAttribute−>select(visibility=#public)
inv: ve.ownedOperation = se.ownedOperation−>select(visibility=#public)
inv: ve.powertypeExtent = se.powertypeExtent
inv: ve.**inv** = se.**inv**

context EnumerationAbstraction
inv: ve.ownedLiteral = se.ownedLiteral

context EnumerationLiteralAbstraction
inv: ve. specification = se. specification .stringInSignature

Instance View

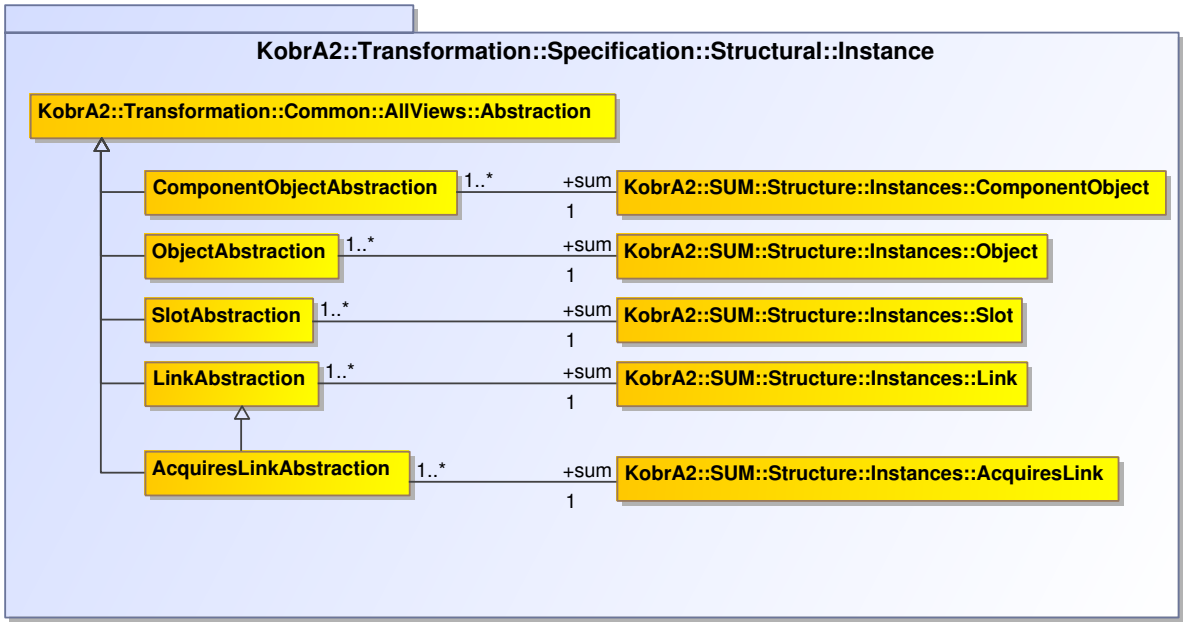

Figure 1.58: Transformations to Specification Instance View

context ComponentObjectAbstraction
inv: ve.hasStereotypes−>includes('componentObject')
inv: ve.slot = se.slot
inv: ve. classifier = se. classifier

context ObjectAbstraction
inv: ve.hasStereotypes−>includes('object')
inv: ve.slot = se.slot
inv: ve. classifier = se. classifier

context SlotAbstraction
inv: ve.owningInstance = se.owningInstance
−− *only show the slots that correspond to public attributes*
inv: ve.definingFeature = se.definingFeature−>select(visibility =#public)
inv: ve.value = se.value.stringInSignature

context LinkAbstraction
inv: ve. classifier = se. classifier

context AcquiresLinkAbstraction
inv: ve.hasStereotypes−>includes('acquiresLink')

1.6.4 Specification - Operational

Service View

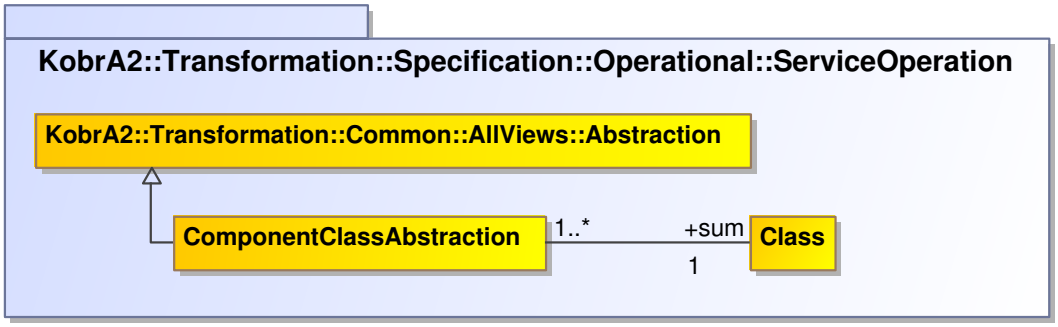

Figure 1.59: Transformation to Specification Service Operation View

context ComponentClassAbstraction
inv: ve.operation.precondition = se.operation−>select(visibility =#public).precondition
inv: ve.operation.postcondition = se.operation−>select(visibility =#public).postcondition
inv: ve.operation.bodycondition = se.operation−>select(visibility=#public).bodycondition
inv: ve.operation.opDef = se.operation−>select(visibility=#public).opDef

Type View

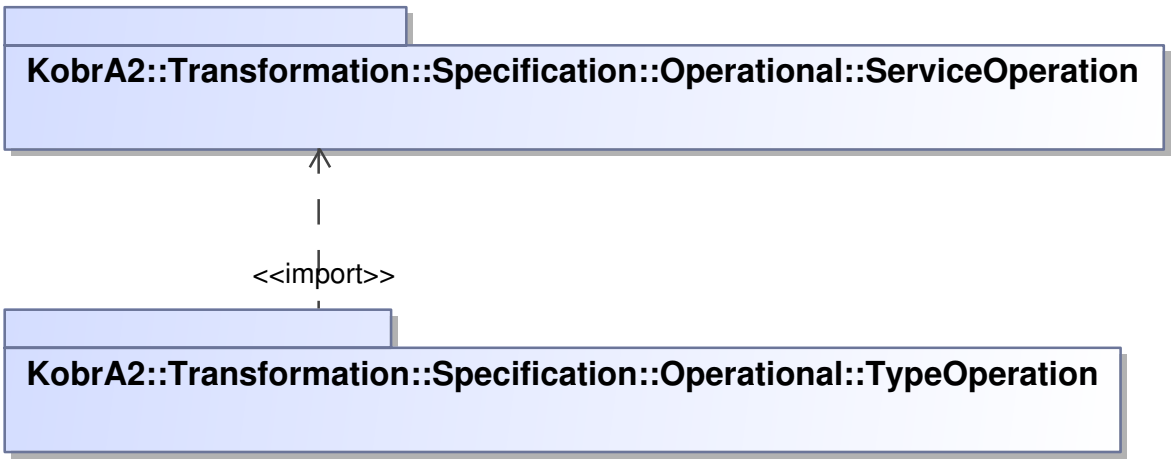

Figure 1.60: Transformation to Specification Type Operation View

1.6.5 Specification - Behavioral

Protocol View

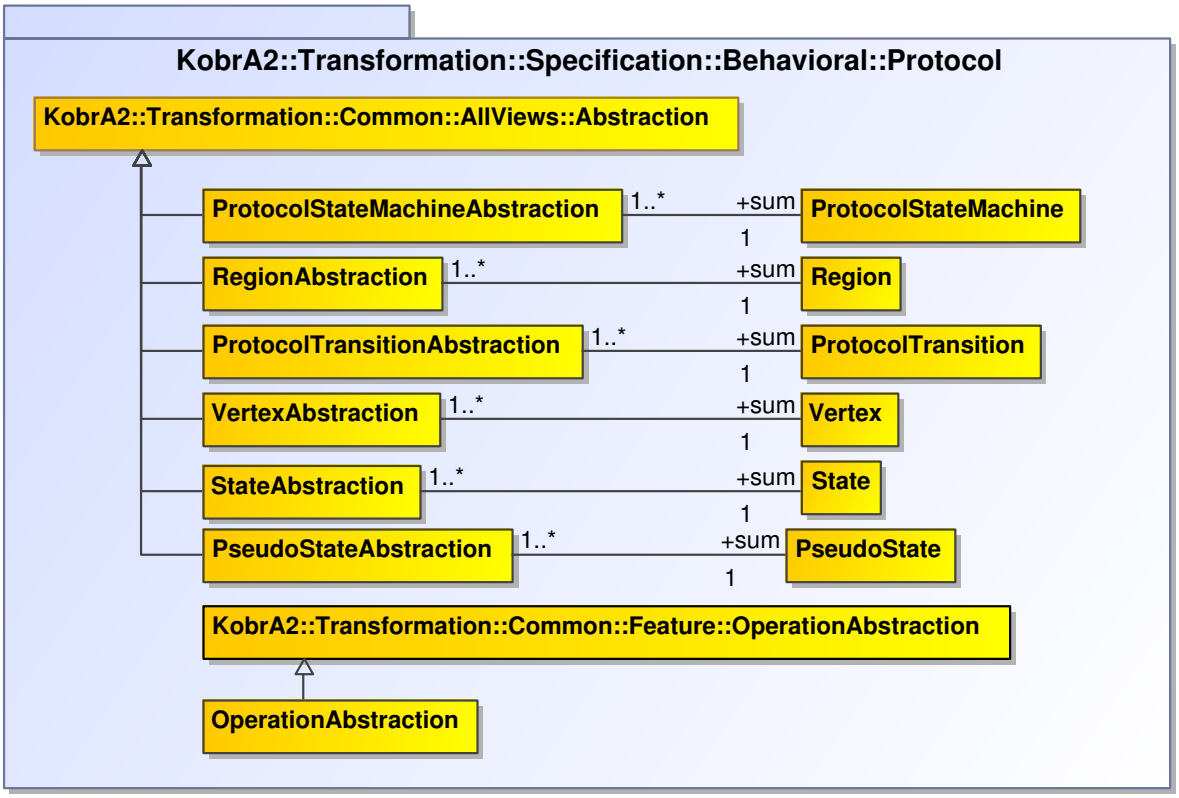

Figure 1.61: Transformation to Protocol View

context ProtocolStateMachineAbstraction
inv: ve.region = se.region

context RegionAbstraction
inv: ve.transition = se.transition
inv: ve.subvertex = se.subvertex

context ProtocolTransitionAbstraction
inv: ve.referred = se.referred
inv: ve.preCondition = se.preCondition
-- *postcondition is empty in the SUM, so no need to transform it*
inv: ve.container = se.container
inv: ve.source = se.source
inv: ve.target = se.target
inv: ve.kind = se.kind

context StateAbstraction
inv: ve.stateInvariant = se.stateInvariant
inv: ve.region = se.region
inv: ve.container = se.container
-- *note: must be false in the SUM*
inv: ve.isComposite = se.isComposite

inv: ve.isSimple = se.isSimple
inv: ve.isOrthogonal = se.isOrthogonal
inv: ve.isSubmachineState = se.isSubmachineState

context VertexAbstraction
inv: ve.container = se.container
inv: ve.incoming = se.incoming
inv: ve.outgoing = se.outgoing

context PseudoStateAbstraction
inv: ve.kind = se.kind
inv: ve.stateMachine = se.stateMachine

context OperationAbstraction
inv: ve.protocolTransition = se.protocolTransition
−− we 'cut' the method and ComponentClass by not copying them

1.6.6 Realization - Structural

Service View

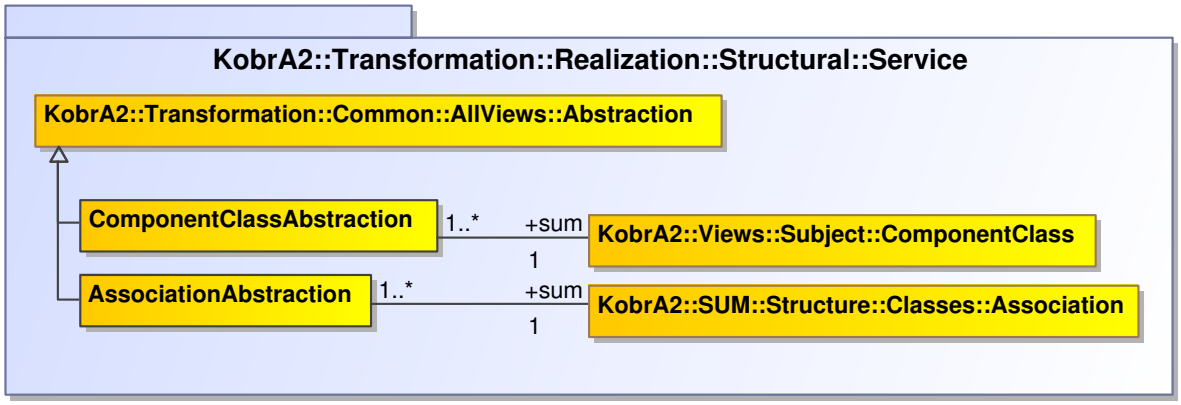

Figure 1.62: Transformation to Realization Service View

context ComponentClassAbstraction
inv: ve.hasStereotypes−>includes('componentClass')
inv: se.isSubject **implies**
−− suppress outgoing associations
(ve.ownedAttribute = se.ownedAttribute−>reject(outgoingAssociationSources)
and ve.ownedOperation = se.ownedOperation
and ve.nestedClassifier = se. nestedClassifier)
inv: se.componentClass.isSubject **implies**
(ve.ownedAttribute = se.ownedAttribute−>select(visibility=#public)
and ve.ownedOperation= se.ownedOperation−>select(visibility=#public)
and ve.superClass = se.superClass
and ve.**inv** = se.**inv**)

context NestedClassAbstraction
inv: se.componentClass.isSubject **implies**
(ve.superClass = se.superClass

and ve.ownedAttribute = se.ownedAttribute−>select(visibility=#public)
and ve.ownedOperation= se.ownedOperatio−>select(visibility=#public)
and ve.**inv** = se.**inv**)

context AssociationAbstraction
inv: ((oclIsKindOf(Acquires) **or** oclIsKindOf(Nests)) **and** se.componentClass.isSubject)
implies
(ve.memberEnd = se.memberEnd
and ve.navigableOwnedEnd = se.navigableOwnedEnd)

Type View

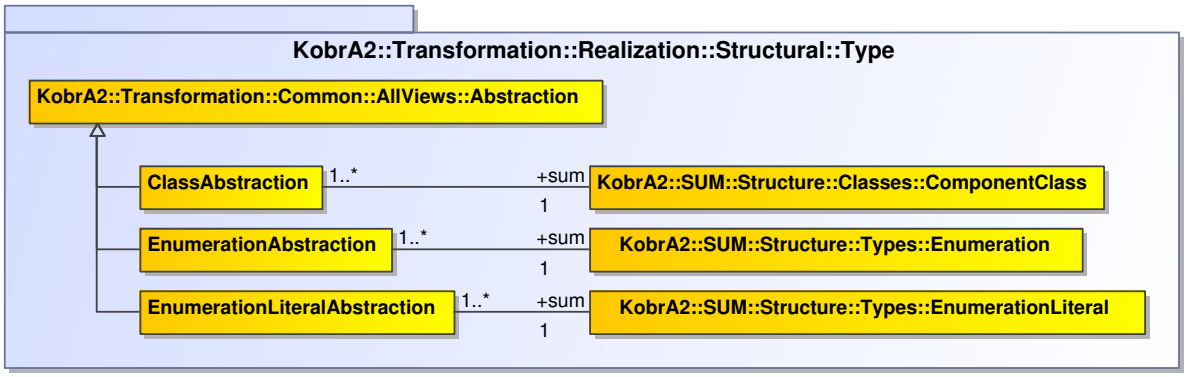

Figure 1.63: Transformation to Realization Type View

context ClassAbstraction
inv: ve.ownedAttribute = se.ownedAttribute
inv: ve.ownedOperation = se.ownedOperation
inv: ve.powertypeExtent = se.powertypeExtent
inv: ve.**inv** = se.**inv**

context EnumerationAbstraction
inv: ve.ownedLiteral = se.ownedLiteral

context EnumerationLiteralAbstraction
inv: ve. specification = se. specification .stringInSignature

Instance View

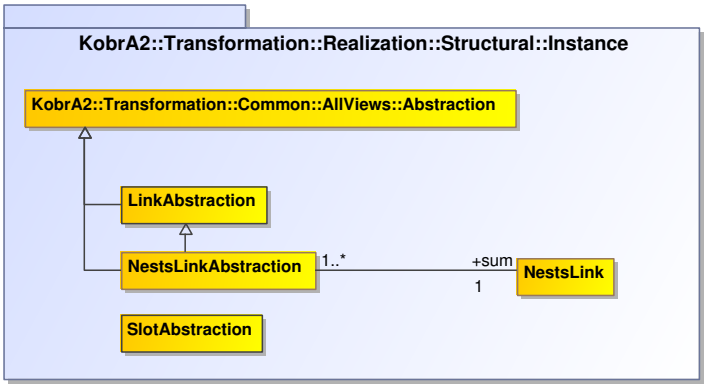

Figure 1.64: Transformation to Realization Instance View

context ComponentObject
−− in the realization instance view, we show the nested instances of the subject component
inv: se. classifier .isSubject **implies**

context SlotAbstraction
inv: ve.owningInstance = se.owningInstance
inv: ve.definingFeature = se.definingFeature
inv: ve.value = se.value

context NestsLinkAbstraction
inv: ve.hasStereotypes−>includes('nestsLink')

1.6.7 Realization - Operational

Service Operation View

Figure 1.65: Transformation to Realization Service Operation View

context ComponentClassAbstraction
inv: ve.operation.precondition = se.operation.precondition
inv: ve.operation.postcondition = se.operation.postcondition
inv: ve.operation.bodycondition = se.operation.bodycondition
inv: ve.operation.opDef = se.operation.opDef

Type Operation View

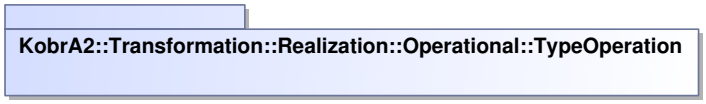

Figure 1.66: Transformation to Realization Type Operation View

1.6.8 Realization - Behavioral

Algorithmic View

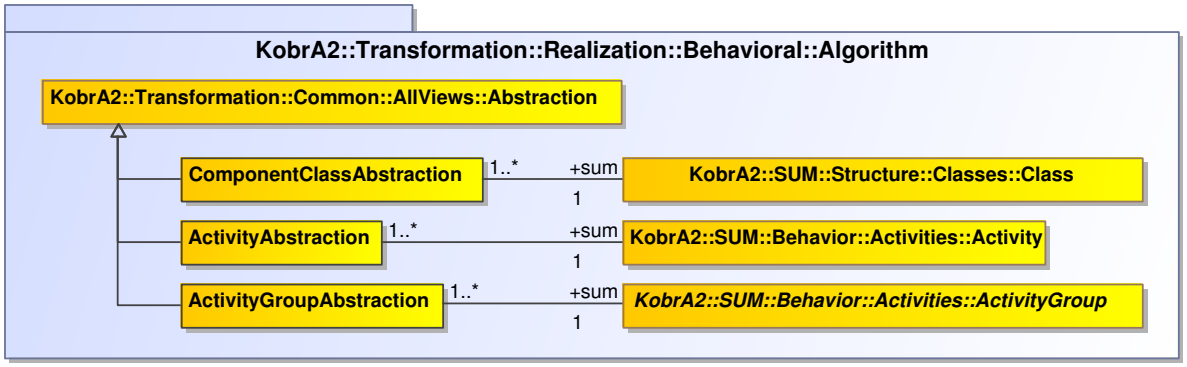

Figure 1.67: Transformation to Algorithmic View

context ComponentClassAbstraction
−− the activities of the component in this view are all the methods of all its operations
inv: ve. activity = se.ownedOperation.method−>asSet()

context ActivityAbstraction
inv: ve.group = se.group
inv: ve.edge = se.edge

context ActivtyGroupAbstraction
inv: ve.containedNode = se.containedNode