Date: April 2012

Precise Semantics of UML Composite Structures *(DRAFT document)*

Draft Version 20120810

OMG Document Number: mars/2012-04-01

Normative reference: http://www.omg.org/spec/acronym/1.0/

Machine readable file(s): http://www.omg.org/acronym/20120401

Normative:	http://www.omg.org/spec/acronym/20120401/foo.xmi
Non-normative:	http://www.omg.org/spec/acronym/20120401/non_normative_foo.xmi

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.)

Table of Contents

1 Scope....1
2 Conformance....3
3 Normative References....3
4 Terms and Definitions....3
5 Symbols....3
6 Additional Information 3
- 6.1 Changes to Adopted OMG Specifications [optional]....3
- 6.2 Acknowledgements....3

7 Abstract Syntax....4
- 7.1 Overview....4
- 7.2 CompositeStructures....4
  - 7.2.1 Overview 4
  - 7.2.2 InvocationActions....4
  - 1.1.1 StructuredClasses 6
  - 1.1.2 InternalStructures 7
  - 1.1.3 Ports 11
- 1.1 Components....14
  - 1.1.1 Overview 14
  - 1.1.4 BasicComponents....14
- 1.2 Classes 16
  - 1.2.1 Overview 16
  - 1.1.5 Interfaces 16
  - 1.1.6 Dependencies 19
- 1.3 CommonBehaviors....25
  - 1.3.1 Overview 25
  - 1.1.7 Communications 25

8 Semantics....27
- 8.1 Overview....27
- 8.2 Loci 27
  - 8.2.1 Overview 27
  - 8.2.2 LociL3 27
- 1.1 Classes 28
  - 1.1.1 Overview 28
  - 1.1.1 Kernel 28
- 1.2 Actions 29
  - 1.2.1 Overview 29
  - 1.1.2 IntermediateActions....29
- 1.3 CompositeStructures....31
  - 1.3.1 Overview 31
  - 1.1.3 StructuredClasses 31
  - 1.1.4 InvocationActions....40

Preface

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable, and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at http://www.omg.org/.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:
http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
140 Kendrick Street
Building A, Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult http://www.iso.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier - 10 pt. Bold: Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to http://www.omg.org/report_issue.htm.

1 Scope

The Scope clause shall appear at the beginning of each specification and define, without ambiguity, the subject of the specification and the aspect(s) covered. It indicates the limits of applicability of the specification or particular parts of it. It shall not contain requirements.

The scope shall be succinct so that it can be used as a summary for bibliographic purposes.

It shall be worded as a series of statements of fact.

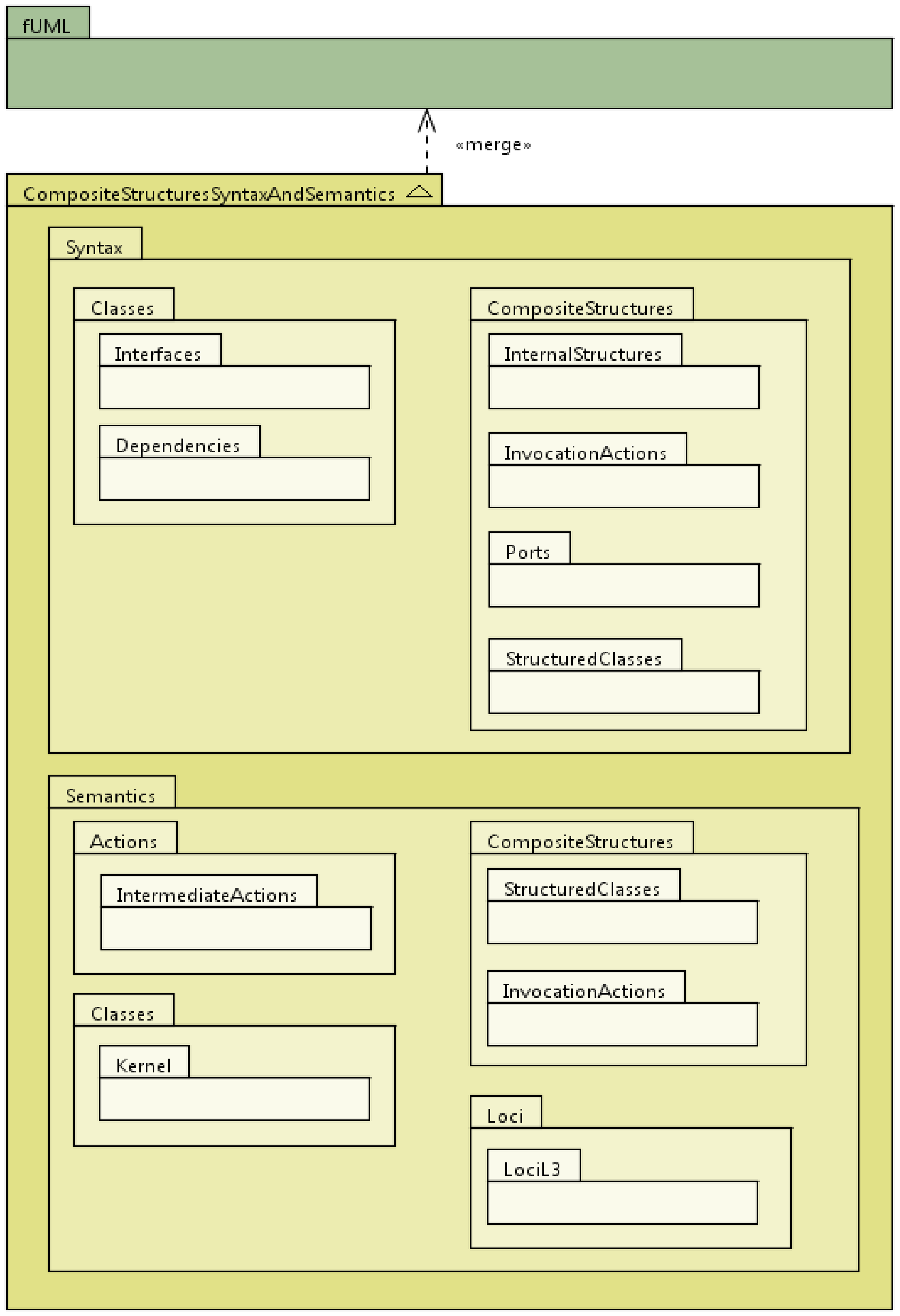


Figure 1: Architecture

2 Conformance

The Conformance clause identifies which clauses of the specification are mandatory (or conditionally mandatory) and which are optional in order for an implementation to claim conformance to the specification.

Note: For conditionally mandatory clauses, the conditions must, of course, be specified.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

List of normative references.

4 Terms and Definitions

For the purposes of this specification, the following terms and definitions apply.

Term

Definition

Term

Definition

Term

Definition

5 Symbols

List of symbols/abbreviations.

6 Additional Information

6.1 Changes to Adopted OMG Specifications [optional]

This specification completely replaces the xxx specification.

6.2 Acknowledgements

7 Abstract Syntax

7.1 Overview

TODO.

7.2 CompositeStructures

7.2.1 Overview

7.2.2 InvocationActions

7.2.2.1 Overview

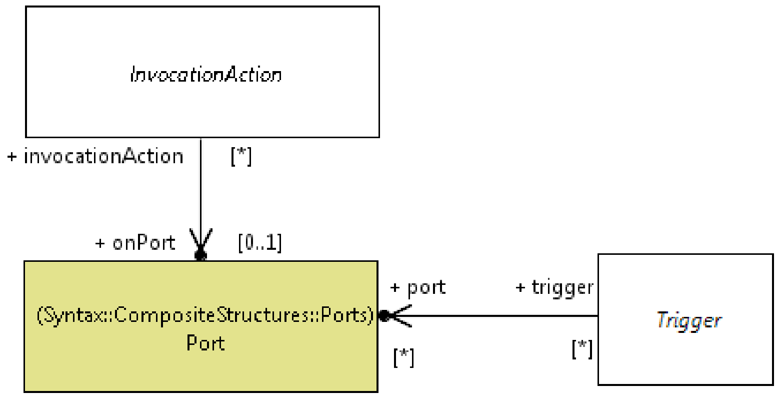


Figure 1: InvocationActions diagram

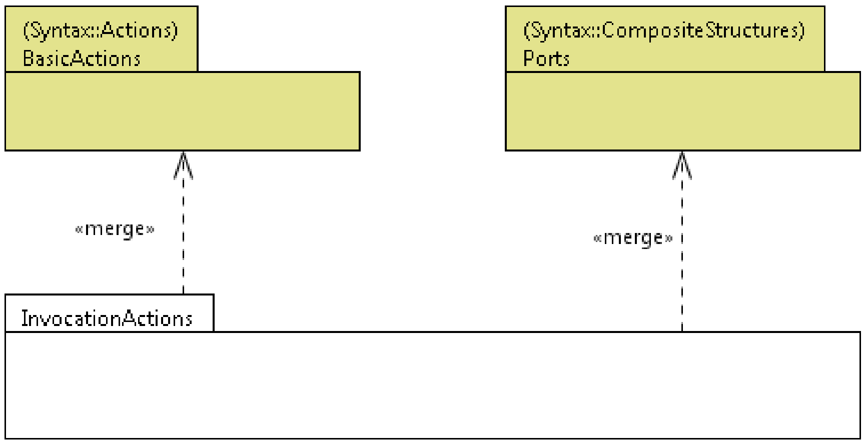


Figure 2: InvocationActions package relationships diagram

7.2.2.2 Class descriptions

Trigger

A trigger specification may be qualified by the port on which the event occurred.

Generalizations

- None

Attributes

- None

Associations

- port : Port [0..*], A optional port of the receiver object on which the behavioral feature is invoked.

InvocationAction

In addition to targeting an object, invocation actions can also invoke behavioral features on ports from where the invocation requests are routed onwards on links deriving from attached connectors. Invocation actions may also be sent to a target via a given port, either on the sending object or on another object.

Generalizations

- None

Attributes

- None

Associations

- onPort : Port [0..1], A optional port of the receiver object on which the behavioral feature is invoked.

1.1.1 StructuredClasses

1.1.1.1 Overview

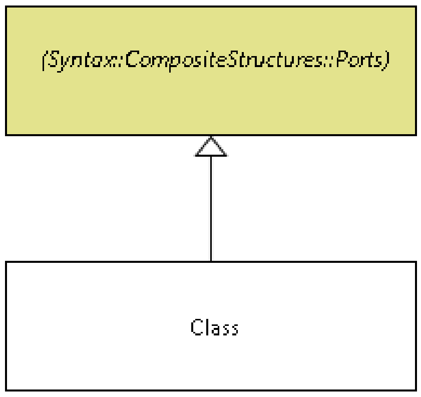


Figure 3: StructuredClasses diagram

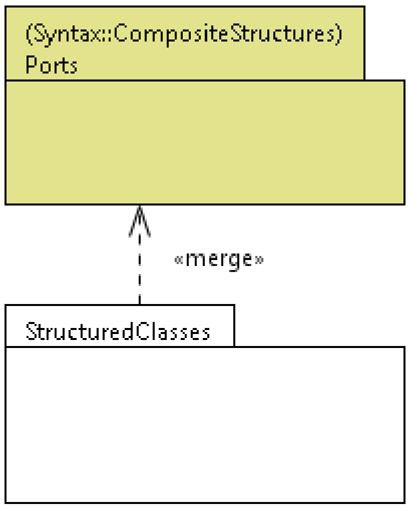


Figure 4: StructuredClasses package relationships diagram

1.1.1.1 Class descriptions

Class

A class has the capability to have an internal structure and ports.

Generalizations

- EncapsulatedClassifier (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::Ports)

Attributes

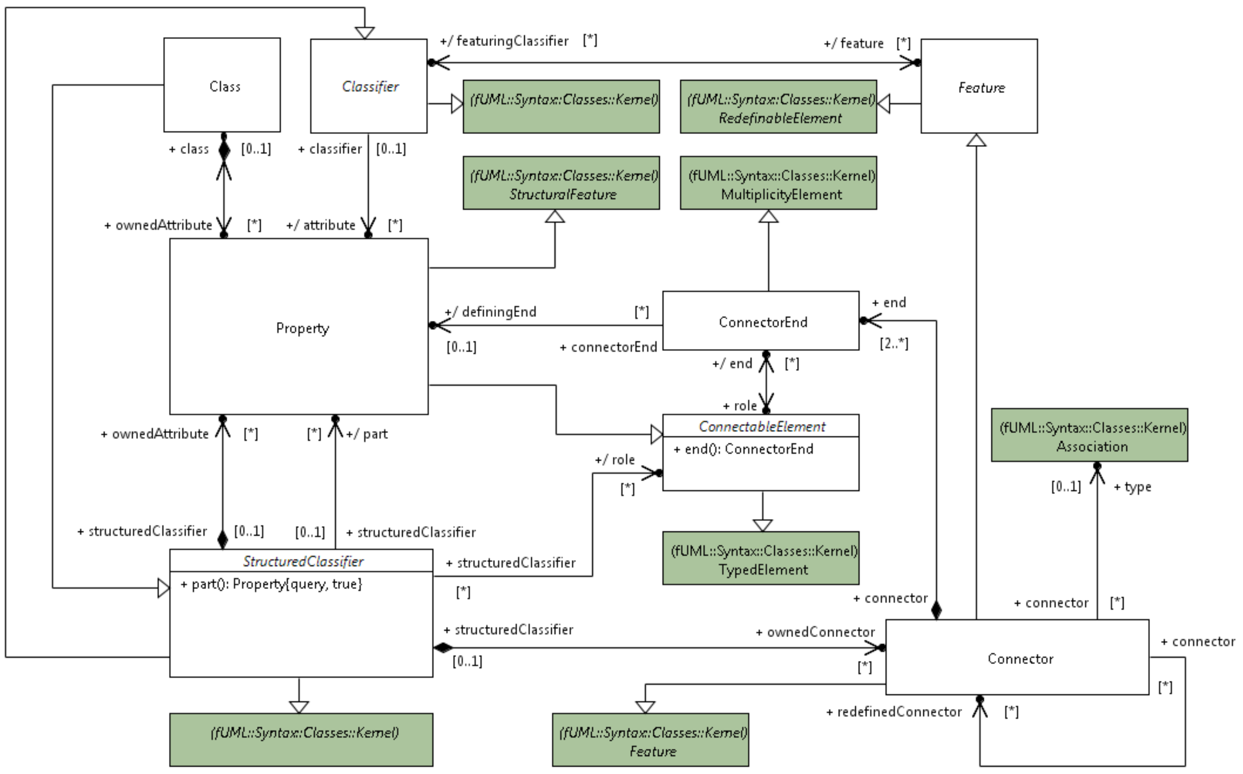
- None

Associations

- None

1.1.2 InternalStructures

1.1.2.1 Overview



Class
Classifier
Feature
+/ featuringClassifier [*]
+/ feature [*]
(fUML::Syntax::Classes::Kernel)
(fUML::Syntax::Classes::Kernel) RedefinableElement
+ class [0..1]
+ classifier [0..1]
(fUML::Syntax::Classes::Kernel) StructuralFeature
(fUML::Syntax::Classes::Kernel) MultiplicityElement
+ ownedAttribute [*]
+/ attribute [*]
Property
+/ definingEnd [*]
[0..1]
+ connectorEnd
ConnectorEnd
+ end
[2..*]
+/ end [*]
+ role
ConnectableElement
+ end(): ConnectorEnd
+/ role
[*]
(fUML::Syntax::Classes::Kernel) Association
[0..1]
+ type
+ ownedAttribute [*]
[*] +/ part
+ structuredClassifier [0..1]
[0..1] + structuredClassifier
(fUML::Syntax::Classes::Kernel) TypedElement
StructuredClassifier
+ part(): Property{query, true}
+ structuredClassifier
[*]
+ connector
+ connector [*]
+ structuredClassifier
+ ownedConnector
[0..1]
[*]
Connector
+ connector
[*]
+ redefinedConnector [*]
(fUML::Syntax::Classes::Kernel)
(fUML::Syntax::Classes::Kernel) Feature

Figure 5: InternalStructures diagram

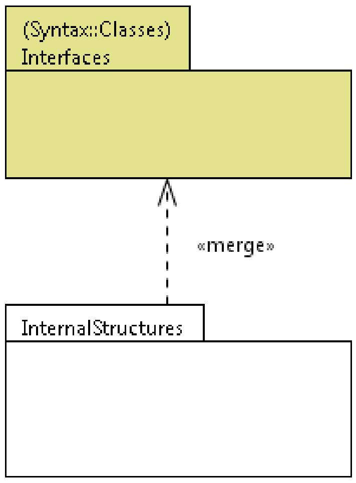


Figure 6: InternalStructures package relationship diagram

1.1.1.2 Class descriptions

ConnectorEnd

A connector end is an endpoint of a connector, which attaches the connector to a connectable element. Each connector end is part of one connector.

Generalizations

- MultiplicityElement (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- definingEnd : Property [0..1], A derived association referencing the corresponding association end on the association which types the connector owing this connector end. This association is derived by selecting the association end at the same place in the ordering of association ends as this connector end.
- role : ConnectableElement [1..1], The connectable element attached at this connector end. When an instance of the containing classifier is created, a link may (depending on the multiplicities) be created to an instance of the classifier that types this connectable element.

StructuredClassifier

A structured classifier is an abstract metaclass that represents any classifier whose behavior can be fully or partly described by the collaboration of owned or referenced instances.

Generalizations

- Classifier (from fUML::Syntax::Classes::Kernel)
- Classifier (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::InternalStructures)

Attributes

- None

Associations

- ownedAttribute : Property [0..*], References the properties owned by the classifier.
- ownedConnector : Connector [0..*], References the connectors owned by the classifier.
- part : Property [0..*], References the properties specifying instances that the classifier owns by composition. This association is derived, selecting those owned properties where isComposite is true.
- role : ConnectableElement [0..*], References the roles that instances may play in this classifier.

Class

Generalizations

- StructuredClassifier (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::InternalStructures)

Attributes

- None

Associations

- ownedAttribute : Property [0..*],

ConnectableElement

ConnectableElement is an abstract metaclass representing a set of instances that play roles of a classifier. Connectable elements may be joined by attached connectors and specify configurations of linked instances to be created within an instance of the containing classifier.

Generalizations

- TypedElement (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- end : ConnectorEnd [0..*], Denotes a set of connector ends that attaches to this connectable element.

Feature

Generalizations

- RedefinableElement (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- featuringClassifier : Classifier [0..*],

Classifier

A classifier has the capability to own collaboration uses. These collaboration uses link a collaboration with the classifier to give a description of the workings of the classifier.

Generalizations

- Namespace (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- attribute : Property [0..*], Refers to all of the Properties that are direct (i.e. not inherited or imported) attributes of the classifier.
- feature : Feature [0..*],

Property

A property represents a set of instances that are owned by a containing classifier instance.

Generalizations

- ConnectableElement (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::InternalStructures)
- StructuralFeature (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- class : Class [0..1],

Connector

Specifies a link that enables communication between two or more instances. This link may be an instance of an association, or it may represent the possibility of the instances being able to communicate because their identities are known by virtue of being passed in as parameters, held in variables or slots, or because the communicating instances are the same instance. The link may be realized by something as simple as a pointer or by something as complex as a network connection. In contrast to associations, which specify links between any instance of the associated classifiers, connectors specify links between instances playing the connected parts only.

Generalizations

- Feature (from fUML::Syntax::Classes::Kernel)
- Feature (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::InternalStructures)

Attributes

- None

Associations

- end : ConnectorEnd [2..*], A connector consists of at least two connector ends, each representing the participation of instances of the classifiers typing the connectable elements attached to this end. The set of connector ends is ordered.
- redefinedConnector : Connector [0..*], A connector may be redefined when its containing classifier is specialized. The redefining connector may have a type that specializes the type of the redefined connector. The types of the connector ends of the redefining connector may specialize the types of the connector ends of the redefined connector. The properties of the connector ends of the redefining connector may be replaced.
- type : Association [0..1], An optional association that specifies the link corresponding to this connector.

1.1.3 Ports

1.1.3.1 Overview

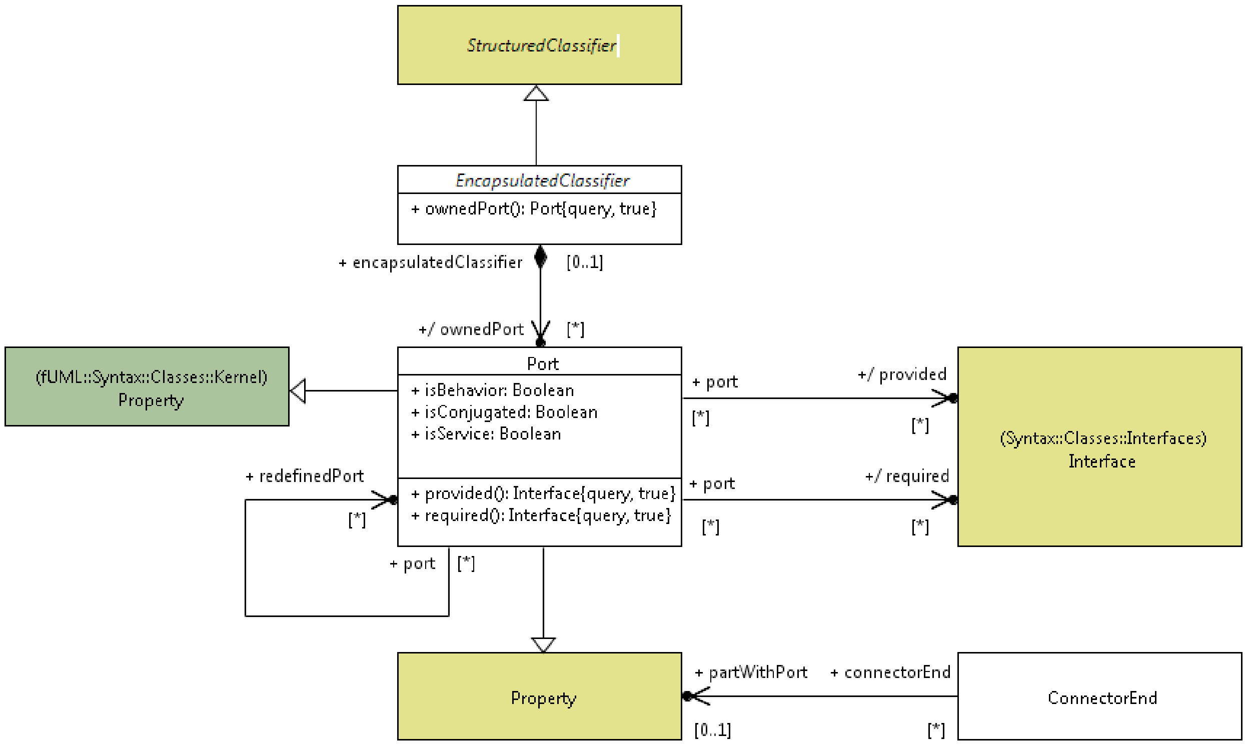


Figure 7: Ports diagram

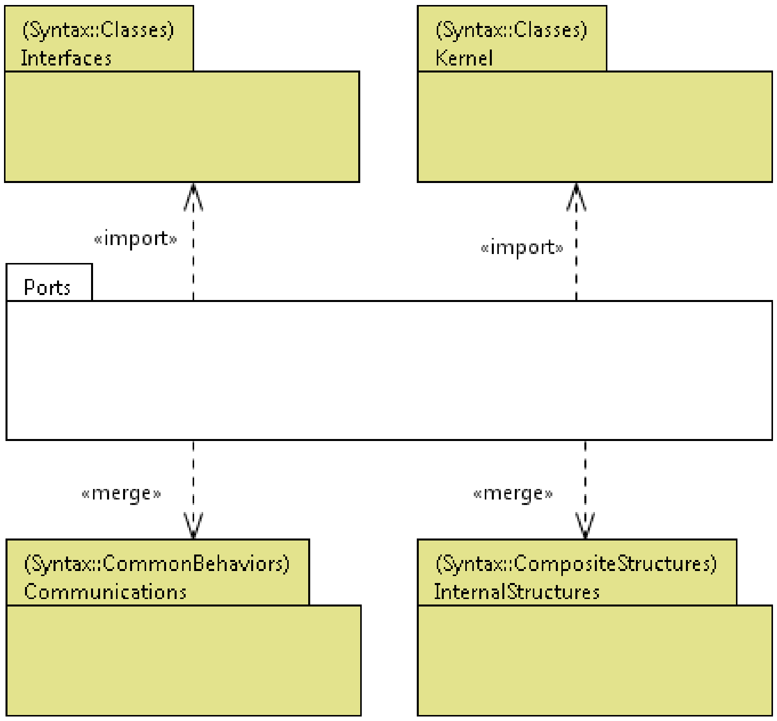

(Syntax::Classes) Interfaces
(Syntax::Classes) Kernel
«import»
«import»
Ports
«merge»
«merge»
(Syntax::CommonBehaviors) Communications
(Syntax::CompositeStructures) InternalStructures

Figure 8: Ports package relationships diagram

1.1.1.3 Class descriptions

Port

A port is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A Port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment.

Generalizations

- Property (from fUML::Syntax::Classes::Kernel)
- Property (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::InternalStructures)

Attributes

- isBehavior : Boolean [1..1], Specifies whether requests arriving at this port are sent to the classifier behavior of this classifier. Such ports are referred to as behavior port. Any invocation of a behavioral feature targeted at a behavior port will be handled by the instance of the owning classifier itself, rather than by any instances that this classifier may contain.
- isConjugated : Boolean [1..1], Specifies the way that the provided and required interfaces are derived from the Port's Type. The default value is false.
- isService : Boolean [1..1], If true indicates that this port is used to provide the published functionality of a classifier; if false, this port is used to implement the classifier but is not part of the essential externally-visible functionality of the classifier and can, therefore, be altered or deleted along with the internal implementation of the classifier and other properties that are considered part of its implementation.

Associations

- provided : Interface [0..*], References the interfaces specifying the set of operations and receptions that the classifier offers to its environment via this port, and which it will handle either directly or by forwarding it to a part of its internal structure. This association is derived according to the value of isConjugated. If isConjugated is false, provided is derived as the union of the sets of interfaces realized by the type of the port and its supertypes, or directly from the type of the port if the port is typed by an interface. If isConjugated is true, it is derived as the union of the sets of interfaces used by the type of the port and its supertypes.
- redefinedPort : Port [0..*], A port may be redefined when its containing classifier is specialized. The redefining port may have additional interfaces to those that are associated with the redefined port or it may replace an interface by one of its subtypes.
- required : Interface [0..*], References the interfaces specifying the set of operations and receptions that the classifier expects its environment to handle via this port. This association is derived according to the value of isConjugated. If isConjugated is false, required is derived as the union of the sets of interfaces used by the type of the port and its supertypes. If isConjugated is true, it is derived as the union of the sets of interfaces realized by the type of the port and its supertypes, or directly from the type of the port if the port is typed by an interface.

EncapsulatedClassifier

A classifier has the ability to own ports as specific and type checked interaction points.

Generalizations

- StructuredClassifier (from CompositeStructuresSyntaxAndSemantics::Syntax::CompositeStructures::InternalStructures)

Attributes

- None

Associations

- ownedPort : Port [0..*], References a set of ports that an encapsulated classifier owns.

ConnectorEnd

A connector end is an endpoint of a connector, which attaches the connector to a connectable element. Each connector end is part of one connector.

Generalizations

- None

Attributes

- None

Associations

- partWithPort : Property [0..1], Indicates the role of the internal structure of a classifier with the port to which the connector end is attached.

1.1 Components

1.1.1 Overview

1.1.4 BasicComponents

1.1.4.1 Overview

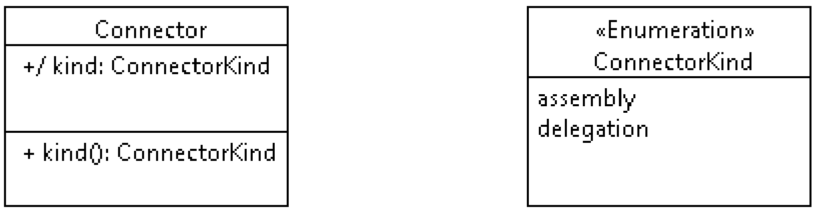


Figure 9: BasicComponents diagram

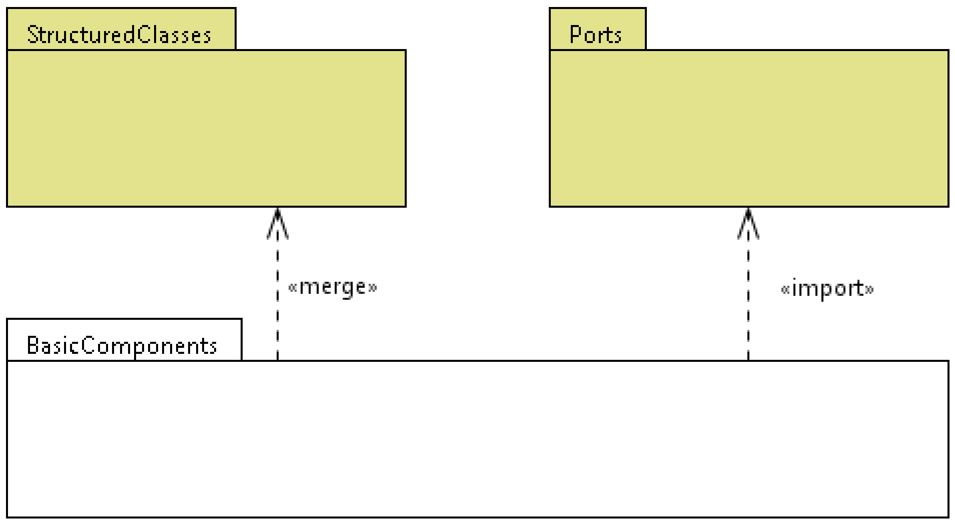


Figure 10: BasicComponents package relationships diagram

1.1.1.4 Class descriptions

Connector

A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the realization of that behavior. It represents the forwarding of events (operation requests and events): a signal that arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling. An assembly connector is a connector between two or more parts or ports on parts that defines that one or more parts provide the services that other parts use.

Generalizations

- None

Attributes

- kind : ConnectorKind [1..1], Indicates the kind of connector. This is derived: a connector with one or more ends connected to a Port which is not on a Part and which is not a behavior port is a delegation; otherwise it is an assembly.

Associations

- None

1.2 Classes

1.2.1 Overview

1.1.5 Interfaces

1.1.5.1 Overview

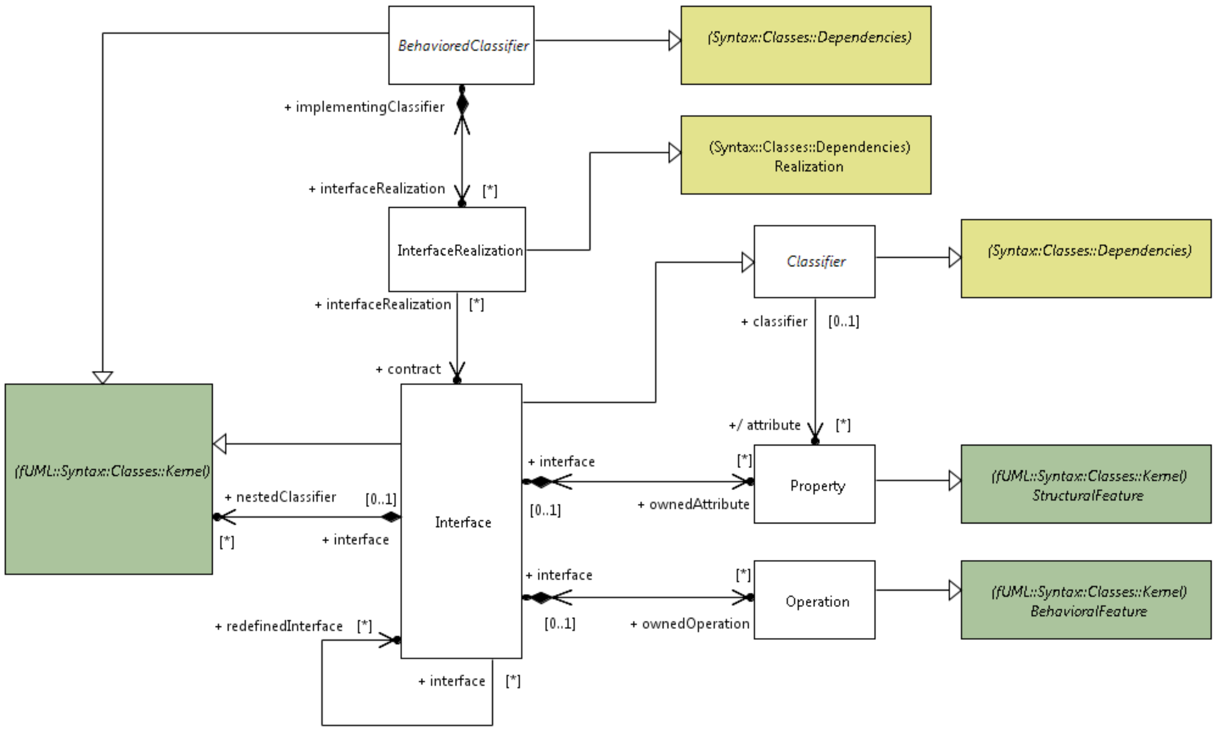

BehavioredClassifier
(Syntax::Classes::Dependencies)
+ implementingClassifier
(Syntax::Classes::Dependencies) Realization
+ interfaceRealization [*]
InterfaceRealization
Classifier
(Syntax::Classes::Dependencies)
+ interfaceRealization [*]
+ classifier [0..1]
+ contract
(fUML::Syntax::Classes::Kernel)
+/ attribute [*]
+ interface
[*]
Interface
Property
(fUML::Syntax::Classes::Kernel) StructuralFeature
+ nestedClassifier [0..1]
[0..1]
+ ownedAttribute
[*]
+ interface
+ interface
[*]
Operation
(fUML::Syntax::Classes::Kernel) BehavioralFeature
+ redefinedInterface [*]
[0..1]
+ ownedOperation
+ interface [*]

Figure 11: Interfaces diagram

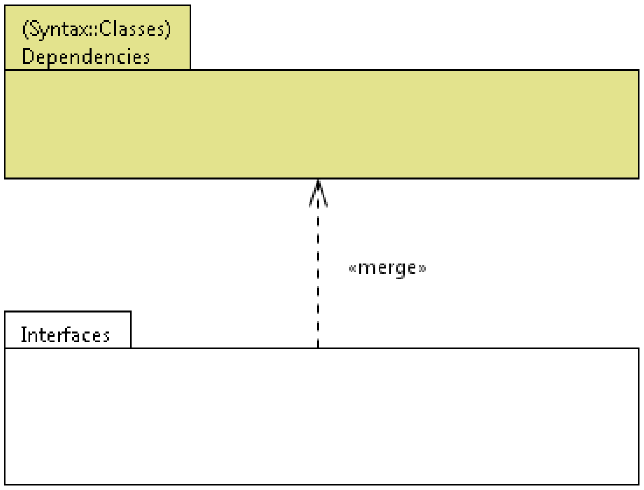


Figure 12: Interfaces package relationships diagram

1.1.1.5 Class descriptions

BehavioredClassifier

A behaviored classifier may have an interface realization.

Generalizations

- NamedElement (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)
- Classifier (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- interfaceRealization : InterfaceRealization [0..*], The set of InterfaceRealizations owned by the BehavioredClassifier. Interface realizations reference the Interfaces of which the BehavioredClassifier is an implementation.

Classifier

Generalizations

- Namespace (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- attribute : Property [0..*],

Operation

Generalizations

- BehavioralFeature (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- interface : Interface [0..1], The Interface that owns this Operation.

Interface

An interface is a kind of classifier that represents a declaration of a set of coherent public features and obligations. An interface specifies a contract; any instance of a classifier that realizes the interface must fulfill that contract. The obligations that may be associated with an interface are in the form of various kinds of constraints (such as pre- and post-conditions) or protocol specifications, which may impose ordering restrictions on interactions through the interface.

Generalizations

- Classifier (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Interfaces)
- Classifier (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- nestedClassifier : Classifier [0..*], References all the Classifiers that are defined (nested) within the Class.
- ownedAttribute : Property [0..*], The attributes (i.e. the properties) owned by the class.
- ownedOperation : Operation [0..*], The operations owned by the class.
- redefinedInterface : Interface [0..*], References all the Interfaces redefined by this Interface.

InterfaceRealization

An interface realization is a specialized realization relationship between a classifier and an interface. This relationship signifies that the realizing classifier conforms to the contract specified by the interface.

Generalizations

- Realization (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- contract : Interface [1..1], References the Interface specifying the conformance contract.
- implementingClassifier : BehavioredClassifier [1..1], References the BehavioredClassifier that owns this Interfacerealization (i.e., the classifier that realizes the Interface to which it points).

Property

Generalizations

- StructuralFeature (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- interface : Interface [0..1], References the Interface that owns the Property

1.1.6 Dependencies

1.1.6.1 Overview

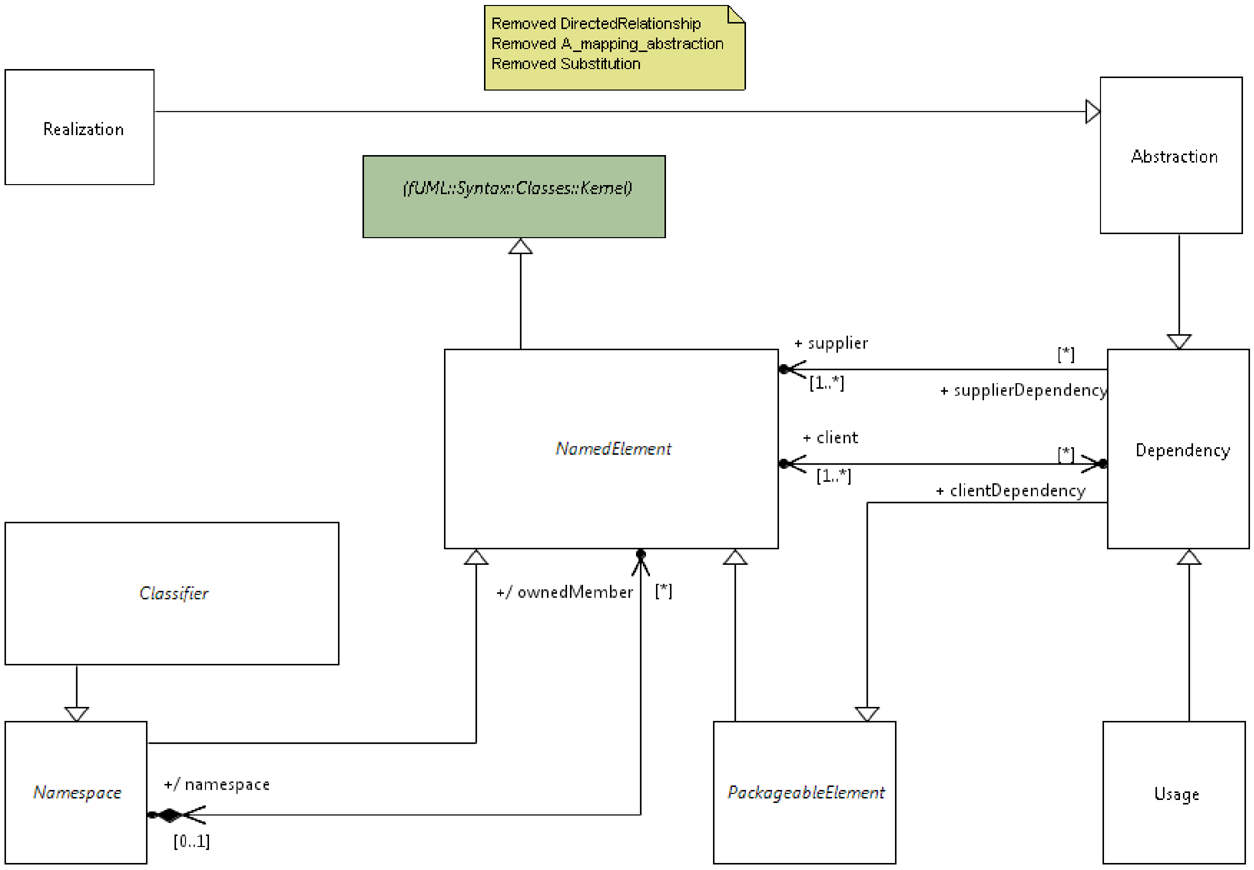


Figure 13: Dependencies diagram

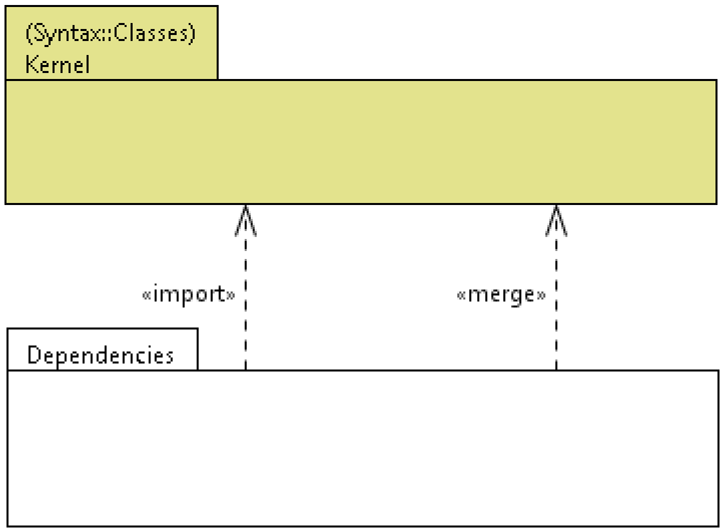


Figure 14: Dependencies package relationships diagram

1.1.1.6 Class descriptions

PackageableElement

Generalizations

- NamedElement (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- None

Classifier

Generalizations

- Namespace (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- None

Usage

A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. A usage is a dependency in which the client requires the presence of the supplier.

Generalizations

- Dependency (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- None

NamedElement

Generalizations

- Element (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- clientDependency : Dependency [0..*], Indicates the dependencies that reference the client.
- namespace : Namespace [0..1], Specifies the namespace that owns the NamedElement.

Namespace

Generalizations

- NamedElement (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- ownedMember : NamedElement [0..*], A collection of NamedElements owned by the Namespace.

Abstraction

An abstraction is a relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints.

Generalizations

- Dependency (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- None

Dependency

A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s).

Generalizations

- PackageableElement (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- client : NamedElement [1..*], The element(s) dependent on the supplier element(s). In some cases (such as a Trace Abstraction) the assignment of direction (that is, the designation of the client element) is at the discretion of the modeler, and is a stipulation.
- supplier : NamedElement [1..*], The element(s) independent of the client element(s), in the same respect and the same dependency relationship. In some directed dependency relationships (such as Refinement Abstractions), a common convention in the domain of class-based OO software is to put the more abstract element in this role. Despite this convention, users of UML may stipulate a sense of dependency suitable for their domain, which makes a more abstract element dependent on that which is more specific.

Realization

Realization is a specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client). Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.

Generalizations

- Abstraction (from CompositeStructuresSyntaxAndSemantics::Syntax::Classes::Dependencies)

Attributes

- None

Associations

- None

1.3 CommonBehaviors

1.3.1 Overview

1.1.7 Communications

1.1.7.1 Overview

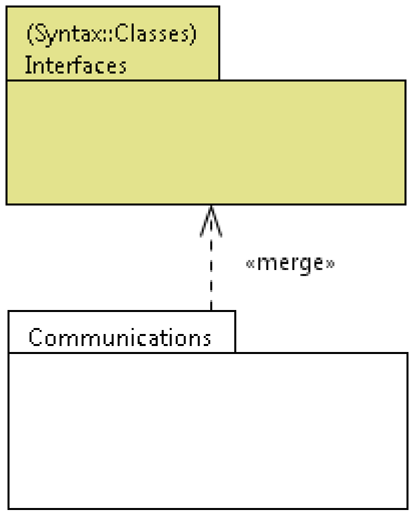


Figure 15: Communications package relationships diagram

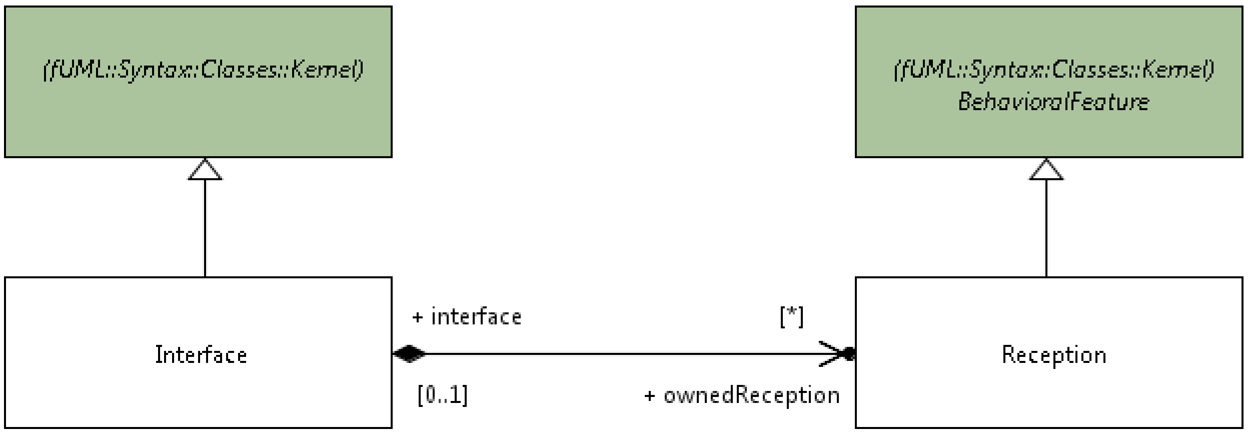


Figure 16: Communications diagram

1.1.1.7 Class descriptions

Interface

Interfaces may include receptions (in addition to operations).

Generalizations

- Classifier (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- ownedReception : Reception [0..*], Receptions that objects providing this interface are willing to accept.

Reception

A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal. A reception designates a signal and specifies the expected behavioral response. The details of handling a signal are specified by the behavior associated with the reception or the classifier itself.

Generalizations

- BehavioralFeature (from fUML::Syntax::Classes::Kernel)

Attributes

- None

Associations

- None

8 Semantics

8.1 Overview

TODO.

8.2 Loci

8.2.1 Overview

8.2.2 LociL3

8.2.2.1 Overview

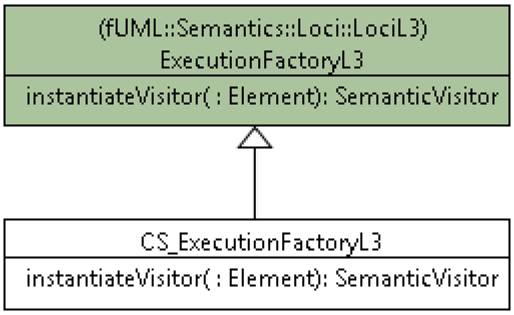

(fUML::Semantics::Loci::LociL3)
ExecutionFactoryL3
instantiateVisitor(: Element): SemanticVisitor
CS_ExecutionFactoryL3
instantiateVisitor(: Element): SemanticVisitor

Figure 17: LociL3 diagram

8.2.2.2 Class descriptions

CS_ExecutionFactoryL3

Generalizations

- ExecutionFactoryL3 (from fUML::Semantics::Loci::LociL3)

Attributes

- None

Associations

- None

Operations

```
instantiateVisitor (element:Element) : SemanticVisitor

  // TODO
  return super.instantiateVisitor(element) ;
```

1.1 Classes

1.1.1 Overview

1.1.1 Kernel

1.1.1.1 Overview

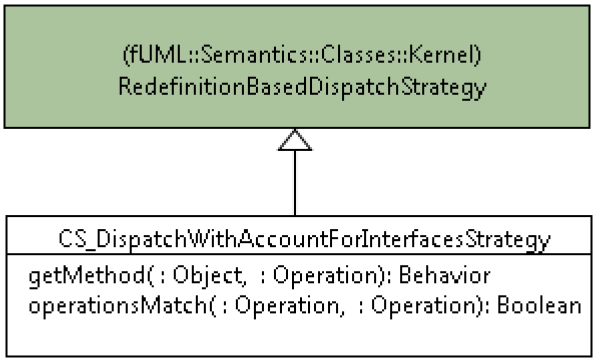

(fUML::Semantics::Classes::Kernel)
RedefinitionBasedDispatchStrategy
CS_DispatchWithAccountForInterfacesStrategy
getMethod(: Object, : Operation): Behavior
operationsMatch(: Operation, : Operation): Boolean

Figure 18: Kernel diagram

1.1.1.1 Class descriptions

CS_DispatchWithAccountForInterfacesStrategy

Extends fUML RedefinitionBasedDispatchStrategy to account for the fact that the invoked operation may belong to an interface, and not to one of the classifiers of the target object (NOTE: Not mandatory to have it defined as an extension. Could be defined as direct specialization of DispatchStrategy)

Generalizations

- RedefinitionBasedDispatchStrategy (from fUML::Semantics::Classes::Kernel)

Attributes

- None

Associations

- None

Operations

```
getMethod (object:Object, operation:Operation) : Behavior

  // Override getMethod so that it accounts for the fact that the Operation
  // may belong to an Interface (realized by one of the classifier of object),
  // and not directly to one of the classifier of object.
  return null ;


operationsMatch (ownedOperation:Operation, baseOperation:Operation) : Boolean

  // Override operationsMatch, in the case where baseOperation belongs
  // to an Interface
  return false ;
```

1.2 Actions

1.2.1 Overview

1.1.2 IntermediateActions

1.1.2.1 Overview

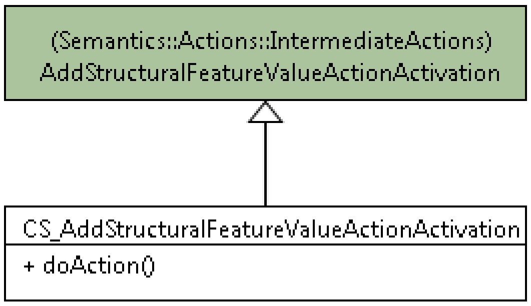

(Semantics::Actions::IntermediateActions)
AddStructuralFeatureValueActionActivation
CS_AddStructuralFeatureValueActionActivation
+ doAction()

Figure 19: IntermediateActions diagram

1.1.1.2 Class descriptions

CS_AddStructuralFeatureValueActionActivation

The behavior of fUML AddStructuralFeatureActionActivation::doAction() is overriden. In the case where the targeted structural feature is a port and the value to be added is a Reference, an InteractionPoint is created on the basis of the given Reference. It then behaves like in fUML, except that the execution continues using the created InteractionPoint instead of the given Reference.

Generalizations

- AddStructuralFeatureValueActionActivation (from fUML::Semantics::Actions::IntermediateActions)

Attributes

- None

Associations

- None

Operations

```
doAction ()


    // If the feature is a port and the input value to be added is a
    // Reference,
    // Replaces this Reference by an InteractionPoint, and then behaves
    // as usual.
    // If the feature is not a port, behaves as usual

    AddStructuralFeatureValueAction action = (AddStructuralFeatureValueAction) (this.node);
    StructuralFeature feature = action.structuralFeature;

    if (!(feature instanceof Port)) {
      // Behaves as usual
      super.doAction();
    } else {
      ValueList inputValues = this.takeTokens(action.value);
      // NOTE: Multiplicity of the value input pin is required to be 1..1.
      Value inputValue = inputValues.getValue(0);
      if (inputValue instanceof Reference) {
        // First constructs an InteractionPoint from the inputValue
        Reference reference = (Reference) inputValue;
        CS_InteractionPoint interactionPoint = new CS_InteractionPoint();
        interactionPoint.referent = reference.referent;
        interactionPoint.definingPort = (Port) feature;
        // The value on action.object is necessarily instanceof
        // ReferenceToCompositeStructure (otherwise, the feature cannot
        // be a port)
        CS_Reference owner = (CS_Reference) this.takeTokens(
            action.object).getValue(0);
        interactionPoint.owner = owner;
        // Then replaces the Reference by an InteractionPoint
        // in the inputValues
        inputValues.remove(0);
        inputValues.addValue(0, interactionPoint);
        // Finally concludes with usual fUML behavior of
        // AddStructuralFeatureValueAction (i.e., the usual behavior
        // when
        // the value on action.object pin is a StructuredValue)
        Integer insertAt = 0;
        if (action.insertAt != null) {
          insertAt = ((UnlimitedNaturalValue) this.takeTokens(
              action.insertAt).getValue(0)).value.naturalValue;
        }
        if (action.isReplaceAll) {
          owner.setFeatureValue(feature, inputValues, 0);
        } else {
          FeatureValue featureValue = owner.getFeatureValue(feature);

          if (featureValue.values.size() > 0 & insertAt == 0) {
            // If there is no insertAt pin, then the structural
            // feature must
            // be unordered, and the insertion position is
            // immaterial.
            insertAt = ((ChoiceStrategy) this.getExecutionLocus().factory
                .getStrategy("choice"))
                .choose(featureValue.values.size());
          }
          if (feature.multiplicityElement.isUnique) {
            // Remove any existing value that duplicates the input
            // value
            Integer j = position(inputValue, featureValue.values, 1);
            if (j > 0) {
              featureValue.values.remove(j - 1);
              if (insertAt > 0 & j < insertAt) {
                insertAt = insertAt - 1;
              }
            }
          }

          if (insertAt <= 0) {
            // Note: insertAt = -1 indicates an unlimited value of
            // "*"
            featureValue.values.addValue(inputValue);
          } else {
            featureValue.values.addValue(insertAt - 1, inputValue);
          }
        }
      } else {
        // behaves as usual
        super.doAction();
      }
    }
```

1.3 CompositeStructures

1.3.1 Overview

1.1.3 StructuredClasses

1.1.3.1 Overview

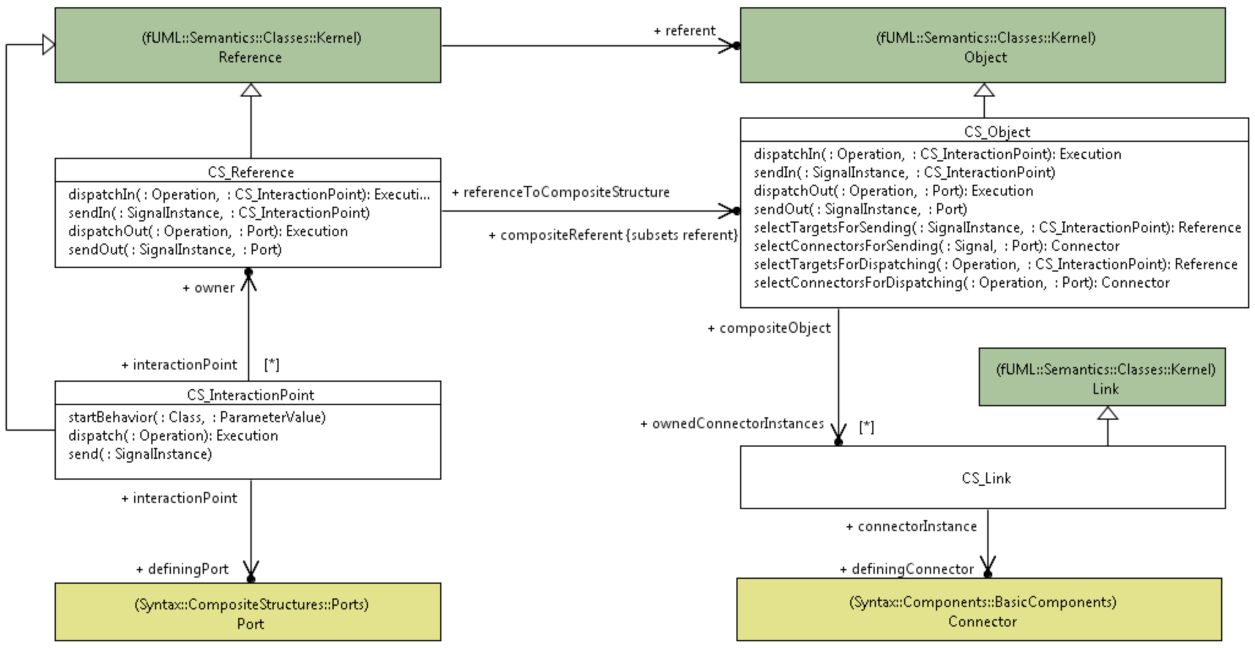

(fUML::Semantics::Classes::Kernel)
Reference
+ referent
(fUML::Semantics::Classes::Kernel)
Object
CS_Object
dispatchIn(: Operation, : CS_InteractionPoint): Execution
sendIn(: SignalInstance, : CS_InteractionPoint)
dispatchOut(: Operation, : Port): Execution
sendOut(: SignalInstance, : Port)
selectTargetsForSending(: SignalInstance, : CS_InteractionPoint): Reference
selectConnectorsForSending(: Signal, : Port): Connector
selectTargetsForDispatching(: Operation, : CS_InteractionPoint): Reference
selectConnectorsForDispatching(: Operation, : Port): Connector
CS_Reference
dispatchIn(: Operation, : CS_InteractionPoint): Executi...
sendIn(: SignalInstance, : CS_InteractionPoint)
dispatchOut(: Operation, : Port): Execution
sendOut(: SignalInstance, : Port)
+ referenceToCompositeStructure
+ compositeReferent {subsets referent}
+ owner
+ interactionPoint
[*]
+ compositeObject
CS_InteractionPoint
startBehavior(: Class, : ParameterValue)
dispatch(: Operation): Execution
send(: SignalInstance)
(fUML::Semantics::Classes::Kernel)
Link
+ ownedConnectorInstances
[*]
CS_Link
+ interactionPoint
+ connectorInstance
+ definingPort
+ definingConnector
(Syntax::CompositeStructures::Ports)
Port
(Syntax::Components::BasicComponents)
Connector

Figure 20: StructuredClasses diagram

1.1.1.3 Class descriptions

CS_InteractionPoint

An InteractionPoint represents the runtime manifestation of a Reference to an Object playing the role of a Port. More specifically, it overrides operation dispatching and signal receptions in order to capture the specific propagation semantics of requests targeting a port.
NOTE: This class is related to the following requirements:
- R1. The target value of an invocation action may also be a port. In this case, the invocation request is sent to the object owning this port as identified by the port identity, and is, upon arrival, handled as described in "Port" clause

Generalizations

- Reference (from fUML::Semantics::Classes::Kernel)

Attributes

- None

Associations

- owner : CS_Reference[1..1], Represents the Reference to the CompositeObject owning this InteractionPort. NOTE: This is introduced to address requirement R3 (It represents the "link from that instance to the instance of the owning classifier [...] through which communication is forwarded to the instance of the owning classifier or through which the owning classifier communicates)
- definingPort : Port[1..1], The Port for which this InteractionPoint is a runtime manifestation

Operations

```
startBehavior (classifier:Class, inputs:ParameterValue[ * ])

  // Overriden to do nothing


dispatch (operation:Operation) : Execution

  // Delegates dispatching to the owning object
  return this.owner.dispatchIn(operation, this) ;


send (signalInstance:SignalInstance)

  // Delegates sending to the owning object
  this.owner.sendIn(signalInstance, this) ;
```

CS_Link

ConnectorInstance extends Link with the ability to specify that this association instance plays a particular Connector.
NOTE: The execution model described in this specification makes the hypothesis that connectors are necessarily typed by an Association.

Generalizations

- Link (from fUML::Semantics::Classes::Kernel)

Attributes

- None

Associations

- definingConnector : Connector[1..1], The Connector played by this ConnectorInstance

Operations

```
None
```

CS_Reference

This class extends fuml Reference with specific operations for managing request propagation through ports, from the environment to the internals of the referent object, or from the referent objet to its environment. (NOTE: Addresses requirement R1 "The target value of an invocation action may also be a port. In this case, the invocation request is sent to the object owning this port as identified by the port identity, and is, upon arrival, handled as described in "Port" clause", and R2 "Invocation actions may also be sent to a target via a given port, either on the sending object or on another object.")

Generalizations

- Reference (from fUML::Semantics::Classes::Kernel)

Attributes

- None

Associations

- compositeReferent : CS_Object[1..1], The composite object referenced by this ReferenceToCompositeStructure. This property subsets Reference::referent.

Operations

```
dispatchIn (operation:Operation, interactionPoint:CS_InteractionPoint) : Execution

  //Delegates dispatching to composite referent
  return this.compositeReferent.dispatchIn(operation,
 interactionPoint) ;


sendIn (signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)

  // delegates sending to composite referent
  this.compositeReferent.sendIn(signalInstance, interactionPoint) ;


sendOut (signalInstance:SignalInstance, onPort:Port)

  // Delegates sending (through the port, to the environment)
  // to compositeReferent
  this.compositeReferent.sendOut(signalInstance, onPort) ;


dispatchOut (operation:Operation, onPort:Port) : Execution

  // Delegates dispatching (through the port, to the environment)
  // to compositeReferent
  return this.compositeReferent.dispatchOut(operation, onPort) ;
```

CS_Object

CompositeObject extends fUML Object with specific operations for managing propagations of requests through ports, from the environment to the internals of this object, or from the objet to its environment.
NOTE, this class addresses the following requirements:
- R4: If connectors are attached to both the port when used on a property within the internal structure of a classifier and the port on the container of an internal structure, the instance of the owning classifier will forward any requests arriving at this port along the link specified by those connectors.
- R5: If there is a connector attached to only one side of a port, any requests arriving at this port will terminante at this port [Non-behavior port]
- R6: For a behavior port, the instance of the owning classifier will handle requests arriving at this port (as specified in the behavior of the classifier), if this classifier has any behavior.
- R7: If there is no behavior defined for this classifier, any communication arriving at a behavior port is lost.
- R8: If several connectors are attached on one side of a port, then any request arriving at this port on a link derived from a connector on the other side of the port will be fowarded on links corresponding to these connectors. It is a semantic variation point whether these requests will be forwarded on all links, or on only one of those links.

Generalizations

- Object (from fUML::Semantics::Classes::Kernel)

Attributes

- None

Associations

- ownedConnectorInstances : CS_Link[0..*], The collection of ConnectorInstance owned by this CompositeObject. For each ConnectorInstance, definingConnector is a Connector belonging to one of the Class typing this CompositeObject

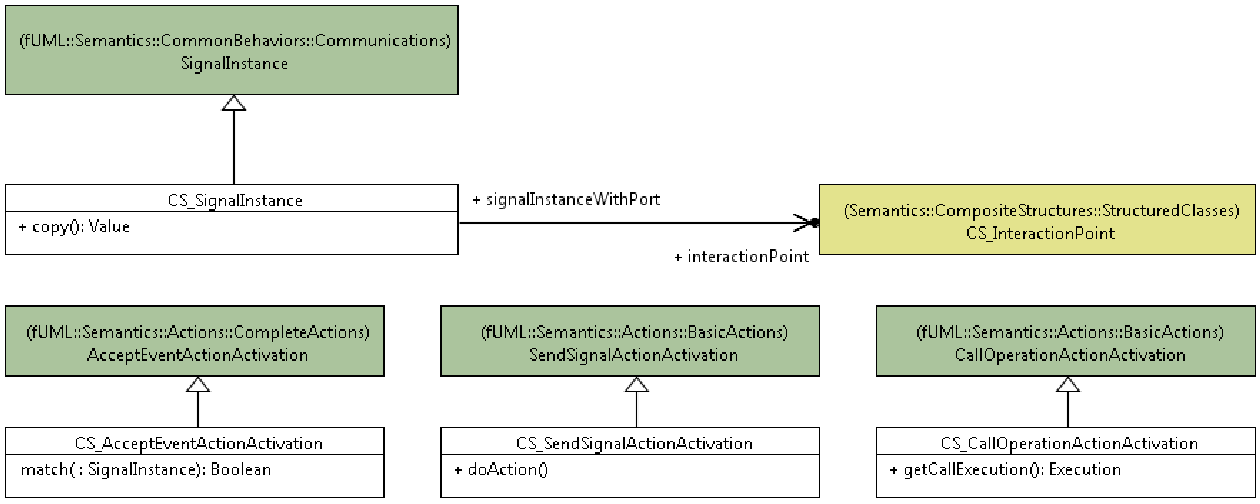
Operations

```
dispatchIn (operation:Operation, interactionPoint:CS_InteractionPoint) : Execution

  // If the interaction is a behavior port, does nothing [for the moment... ?],
  // since the only kind of event supported in fUML is SignalEvent
  // If this is not a behavior port, select appropriate delegation links
  // from interactionPoint, and propagates the operation call through
  // these links
  Execution execution = null ;
  if (interactionPoint.definingPort.isBehavior) {
    // Do nothing
  }
  else {
    ReferenceList targets =
      this.selectTargetsForDispatching(operation, interactionPoint) ;
    // If targets is empty, no delegation target have been found,
    // and the operation call will be lost
    if (! (targets.size()==0)) {
      // Choose one target non-deterministically
      Integer index =
            ((ChoiceStrategy)this.locus.factory.getStrategy("choice"))
            .choose(targets.size()) ;
      Reference target = targets.getValue(index - 1) ;
      execution = target.dispatch(operation) ;
    }
  }
  return execution ;
```

```
sendIn (signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)

  // If the interaction is a behavior port,
  // creates a SignalInstanceWithPort from the signal instance,
  // and sends it as usual using operation send
  // If this is not a behavior port,
  // select appropriate delegation targets from interactionPoint,
  // and propagates the signal to these targets
  if (interactionPoint.definingPort.isBehavior) {
    CS_SignalInstance newSignalInstance =
                          new CS_SignalInstance() ;
    SignalInstance copy = (SignalInstance)signalInstance.copy() ;
    newSignalInstance.featureValues = copy.featureValues ;
    newSignalInstance.type = copy.type ;
    newSignalInstance.interactionPoint = interactionPoint ;
    this.send(newSignalInstance) ;
  }
  else {
    ReferenceList targets =
          this.selectTargetsForSending(signalInstance,
                                     interactionPoint) ;
    // If targets is empty, no delegation target have been found,
    // and the signal is lost
    Integer i = 1 ;
    // Do the following concurrently
    while (i <= targets.size()) {
      Reference target = targets.getValue(i-1) ;
      target.send(signalInstance) ;
      i = i + 1 ;
    }
  }

selectTargetsForSending (signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint) :
Reference[*]

  // From the given signalInstance and interactionPoint,
  // retrieves potential connectors through which request can be delegated
  // These connectors are delegation connectors attached to
  // Port interactionPoint.definingPort, and whose target provide a
  // reception for Signal signalInstance.type
  ConnectorList connectors =
        this.selectConnectorsForSending(signalInstance.type,
                                      interactionPoint.definingPort) ;

  // Select links owned by the context object for which the
  // definingConnector is included in the list of matching connectors.
  Integer i = 1 ;
  CS_LinkList connectorInstances =
                                new CS_LinkList() ;
  while (i <= connectors.size()) {
    Integer j = 1 ;
    Connector connector = connectors.getValue(i-1) ;
    while (j <= this.ownedConnectorInstances.size()) {
      CS_Link connectorInstance =
                     this.ownedConnectorInstances.getValue(j-1) ;
      if (connectorInstance.definingConnector == connector) {
        connectorInstances.addValue(connectorInstance) ;
      }
      j=j+1 ;
    }
    i = i+1 ;
  }

  // For each matching link, retrieves the end value opposite
  // to interactionPoint.
  // If this value is a reference (which means that it is possible to send it
  // a signal), it is added in the list of potential targets.
  ReferenceList targets = new ReferenceList() ;
  i = 1 ;
  while (i <= connectorInstances.size()) {
    CS_Link link = connectorInstances.getValue(i-1) ;
    Association association = link.type ;
    Property oppositeEnd = association.memberEnd.getValue(0);
    if (oppositeEnd == interactionPoint.definingPort) {
       oppositeEnd = association.memberEnd.getValue(1);
    }
    Value value = link.getFeatureValue(oppositeEnd).values.getValue(0) ;
    if (value instanceof Reference) {
      targets.addValue((Reference)value) ;
    }
    i = i + 1;
  }

  // if targets is empty, no matching targets have been found,
  // and the signal instance will be lost
  return targets ;

selectConnectorsForSending (signal:Signal, port:Port) : Connector[*]

  // From the given signal and port, retrieves potential connectors
  // through which request can be delegated
  // These connectors are delegation connectors attached to Port
  // interactionPoint.definingPort,
  // and whose target provide a reception for Signal signalInstance.type
  ConnectorList connectors = new ConnectorList() ;
  Integer i = 1 ;
  // Iterates on types of this CompositeObject
  while (i <= this.types.size()) {
    Type t =this.types.getValue(i-1) ;
    if (t instanceof Class_) {
      Class_ class_ = (Class_) t ;
      Integer j = 1 ;
      // Iterates on Connectors of the current type
      while (j <= class_.encapsulatedClassifier.ownedConnector.size()) {
        Connector cddConnector =
                class_.encapsulatedClassifier.ownedConnector.getValue(j-1) ;
        if (cddConnector.kind == ConnectorKind.delegation) {
          Integer k = 1 ;
          Boolean matches = false ;
          // Iterates on ConnectorEnds of the current Connector
          while (k <= cddConnector.end.size() && !matches) {
            ConnectorEnd cddEnd = cddConnector.end.getValue(k-1) ;
            if (!(cddEnd.role.actualConnectableElement == port && cddEnd.partWithPort == null)) {
              if (cddEnd.role.actualConnectableElement instanceof Port) {
                // We have to look in provided interfaces of the port
                // if there is a reception defined for signal
                Integer l = 1 ;
                // Iterates on provided interfaces of the current end
                InterfaceList providedInterfaces =
                          ((Port)cddEnd.role.actualConnectableElement).provided() ;
                while (l <= providedInterfaces.size() && !matches) {
                  Interface interface_ =
                    ((Port)cddEnd.role.actualConnectableElement).provided().getValue(l-1) ;
                  // Iterates on Receptions of the current interface
                  Integer m = 1 ;
                  while (m <= interface_.ownedReception.size()
                        && !matches) {
                    if (interface_.ownedReception.getValue(m-1).signal
                      == signal) {
                      matches = true ;
                      connectors.addValue(cddConnector) ;
                    }
                    m = m + 1 ;
                  }
                  l = l + 1 ;
                }
                if (matches == false) {
                  // No matching reception has been found in reception
                  // directly owned by provided interfaces
                  // Need to check in inherited members of these
                  //provided interfaces.
                  l = 1 ;
                  // Iterates again on provided interfaces
                  while (l <= providedInterfaces.size() && ! matches) {
                    Interface interface_ =
                          providedInterfaces.getValue (l -1) ;
                    // Iterates on inherited members of the current Interface
                    Integer m = 1 ;
                    while (m <= interface_.inheritedMember.size()
                          && !matches) {
                      NamedElement cddReception =
                            interface_.inheritedMember.getValue(m-1) ;
                      if ((cddReception instanceof Reception)
                         &&
                        (((Reception)cddReception).signal == signal)) {
                        matches = true ;
                        connectors.addValue(cddConnector) ;
                      }
                      m = m + 1 ;
                    }
                    l = l + 1 ;
                  }
                }
              }
              else {
                // The connector does not target a Port.
                // We have to look if the Classifier typing this property
                // directly or indirectly provides a Reception for signal
                Integer l = 1 ;
                if (cddEnd.role.actualConnectableElement.typedElement.type instanceof Classifier) {
                  while (l <= ((Classifier)cddEnd.role
                                .actualConnectableElement.typedElement.type).member.size()
                        && !matches) {
                    NamedElement cddReception =
                        ((Classifier)cddEnd.role.actualConnectableElement.typedElement.type)
                                .member.getValue(l-1) ;
                    if ((cddReception instanceof Reception)
                      &&
                      (((Reception)cddReception).signal == signal)) {
                      matches = true ;
                      connectors.addValue(cddConnector) ;
                    }
                    l = l + 1 ;
                  }
                }
              }
            }
            k = k + 1 ;
          }
        }
        j = j + 1 ;
      }
      i = i + 1 ;
    }
  }

  return connectors ;

selectTargetsForDispatching (operation:Operation, interactionPoint:CS_InteractionPoint) :
Reference[*]

  // From the given operation and interactionPoint, retrieves potential
  // connectors through which request can be delegated
  // These connectors are delegation connectors attached to Port
  // interactionPoint.definingPort,
  // and whose target provides or realize operation
  ConnectorList connectors =
      this.selectConnectorsForDispatching(operation,
                                      interactionPoint.definingPort) ;

  // Select links owned by the context object for which the
  // definingConnector is included in the list of matching connectors.
  Integer i = 1 ;
  CS_LinkList connectorInstances =
                      new CS_LinkList() ;
  while (i <= connectors.size()) {
    Integer j = 1 ;
    Connector connector = connectors.getValue(i-1) ;
    while (j <= this.ownedConnectorInstances.size()) {
      CS_Link connectorInstance =
              this.ownedConnectorInstances.getValue(j-1) ;
      if (connectorInstance.definingConnector == connector) {
        connectorInstances.addValue(connectorInstance) ;
      }
      j=j+1 ;
    }
    i = i+1 ;
  }

  // For each matching link, retrieves the end value opposite
  // to interactionPoint.
  // If this value is a reference (which means that it is possible to dispatch
  // operation to it), it is added in the list of potential targets.
  ReferenceList targets = new ReferenceList() ;
  i = 1 ;
  while (i <= connectorInstances.size()) {
    CS_Link link = connectorInstances.getValue(i-1) ;
    Association association = link.type ;
    Property oppositeEnd = association.memberEnd.getValue(0);
    if (oppositeEnd == interactionPoint.definingPort) {
       oppositeEnd = association.memberEnd.getValue(1);
    }
    Value value = link.getFeatureValue(oppositeEnd).values.getValue(0) ;
    if (value instanceof Reference) {
      targets.addValue((Reference)value) ;
    }
    i = i + 1;
  }

  // if targets is empty, no matching targets have been found,
  // and the operation call will be lost
  return targets ;

selectConnectorsForDispatching (operation:Operation, port:Port) : Connector[*]

  // From the given signal and port, retrieves potential connectors through
  // which request can be delegated. These connectors are delegation
  // connectors attached to Port interactionPoint.definingPort, and whose
  //target provides the requested operation

  ConnectorList connectors = new ConnectorList() ;
  Integer i = 1 ;
  // Iterates on types of this CompositeObject
  while (i <= this.types.size()) {
    Type t =this.types.getValue(i-1) ;
    if (t instanceof Class_) {
      Class_ class_ = (Class_) t ;
      Integer j = 1 ;
      // Iterates on Connectors of the current type
      while (j <= class_.encapsulatedClassifier.ownedConnector.size()) {
        Connector cddConnector = class_.encapsulatedClassifier.ownedConnector.getValue(j-1) ;
        if (cddConnector.kind == ConnectorKind.delegation) {
          Integer k = 1 ;
          Boolean matches = false ;
          // Iterates on ConnectorEnds of the current Connector
          while (k <= cddConnector.end.size() && !matches) {
            ConnectorEnd cddEnd = cddConnector.end.getValue(k-1) ;
            if (!(cddEnd.role.actualConnectableElement == port && cddEnd.partWithPort == null)) {
              if (cddEnd.role.actualConnectableElement instanceof Port) {
                if (operation.owner instanceof Interface) {
                  // We have to look in provided interfaces of the port if
                  // they define directly or indirectly the Operation
                  Integer l = 1 ;
                  // Iterates on provided interfaces of the current end
                  InterfaceList providedInterfaces =
                            ((Port)cddEnd.role.actualConnectableElement).provided() ;
                  while (l <= providedInterfaces.size() && !matches) {
                    Interface interface_ = ((Port)cddEnd.role.actualConnectableElement)
                                      .provided()
                                      .getValue(l-1) ;
                    // Iterates on members of the current Interface
                    Integer m = 1 ;
                    while (m <= interface_.member.size() && !matches) {
                      NamedElement cddOperation =
                                interface_.member
                                       .getValue(m-1) ;
                      if (cddOperation instanceof Operation) {
                        DispatchWithAccountForInterfacesStrategy strategy =
                          new DispatchWithAccountForInterfacesStrategy() ;
                        matches =
                          strategy.operationsMatch(
                                      (Operation)cddOperation,
                                      operation) ;
                        if (matches) {
                          connectors.addValue(cddConnector) ;
                        }
                      }
                      m = m + 1 ;
                    }
                    l = l + 1 ;
                  }
                }
              }
              else {
                // The connector does not target a Port.
                // We have to look if the Classifier typing this property
                // directly or indirectly provides a Reception for signal
                Integer l = 1 ;
                if (cddEnd.role.actualConnectableElement.typedElement.type instanceof Class_) {
                  while (l <= ((Class_)cddEnd.role.actualConnectableElement.typedElement.type)
                                .member.size()
                        && !matches) {
                    NamedElement cddOperation =
                            ((Class_)cddEnd.role.actualConnectableElement.typedElement.type)
                            .member.getValue(l-1) ;
                    if (cddOperation instanceof Operation) {
                      DispatchWithAccountForInterfacesStrategy strategy =
                        new DispatchWithAccountForInterfacesStrategy() ;
                      matches =
                        strategy.operationsMatch(
                                      (Operation)cddOperation,
                                      operation) ;
                      if (matches) {
                        connectors.addValue(cddConnector) ;
                      }
                    }
                    l = l + 1 ;
                  }
                }
              }
            }
            k = k + 1 ;
          }
        }
        j = j + 1 ;
      }
      i = i + 1 ;
    }
  }

  return connectors ;

sendOut (signalInstance:SignalInstance, onPort:Port)

  // TODO:
  // Propagate the signal instance through interaction points
  // corresponding to onPorts, following appropriate links.
  // This will result in calling send(signalInstance) on
  // oppositeEnd found from the links


dispatchOut (operation:Operation, onPort:Port) : Execution

  // TODO:
  // Propagate the operation call through interaction points corresponding
  // to onPorts, following appropriate links,
  // This will result in calling dispatch(operation) on oppositeEnd found
  // from the links

  return null ;
```

1.1.4 InvocationActions

1.1.4.1 Overview



(fUML::Semantics::CommonBehaviors::Communications)
SignalInstance
CS_SignalInstance
+ copy(): Value
+ signalInstanceWithPort
+ interactionPoint
(Semantics::CompositeStructures::StructuredClasses)
CS_InteractionPoint
(fUML::Semantics::Actions::CompleteActions)
AcceptEventActionActivation
(fUML::Semantics::Actions::BasicActions)
SendSignalActionActivation
(fUML::Semantics::Actions::BasicActions)
CallOperationActionActivation
CS_AcceptEventActionActivation
match(: SignalInstance): Boolean
CS_SendSignalActionActivation
+ doAction()
CS_CallOperationActionActivation
+ getCallExecution(): Execution

Figure 21: InvocationActions diagram

1.1.1.4 Class descriptions

CS_SendSignalActionActivation

Extends behavior of fUML SendSignalActionActivation::doAction(). If onPort is specified, instead of sending directly to target reference by calling operation send, sendOut (cf. ReferenceToCompositeStructure) is called, so that the constructed signal instance will be finally sent to the environment. (Note: Addresses requirement R2 "Invocation actions may also be sent to a target via a given port, either on the sending object or on another object.")

Generalizations

- SendSignalActionActivation (from fUML::Semantics::Actions::BasicActions)

Attributes

- None

Associations

- None

Operations

```
doAction ()

  // If onPort is not specified, behaves like in fUML
  // If onPort is specified,
  // Get the value from the target pin. If the value is not a reference,
  // then do nothing.
  // Otherwise, construct a signal using the values from the argument pins
  // As compared to fUML, instead of sending directly to target reference
  // by calling operation send,
  // sendOut is called, so that the constructed signal will be finally sent
  // to the environment.

  SendSignalAction action = (SendSignalAction)(this.node);

  if (action.onPort == null) {
    super.doAction() ;
  }
  else {
    Value target = this.takeTokens(action.target).getValue(0) ;

    if (target instanceof CS_Reference) {
     Signal signal = action.signal;
     SignalInstance signalInstance = new SignalInstance();
     signalInstance.type = signal;

     PropertyList attributes = signal.ownedAttribute;
     InputPinList argumentPins = action.argument;
      Integer i = 0 ;
     while ( i < attributes.size()) {
     Property attribute = attributes.getValue(i);
     InputPin argumentPin = argumentPins.getValue(i);
     ValueList values = this.takeTokens(argumentPin);
     signalInstance.setFeatureValue(attribute, values, 0);
     }

      CS_Reference targetReference =
          (CS_Reference)target ;
      Port onPort = action.onPort ;
      targetReference.sendOut(signalInstance, onPort) ;
    }
  }
```

CS_SignalInstance

SignalInstanceWithPort extends fUML SignalInstance with the ability to reference the specific InteractionPoint on which it occured. This is introduced to address the following requirements R9 ("Specifying one or more ports for an event implies that the event triggers the execution of an associated behavior only if the event was received via one of the specified ports.").

Generalizations

- SignalInstance (from fUML::Semantics::CommonBehaviors::Communications)

Attributes

- None

Associations

- interactionPoint : CS_InteractionPoint[1..1], The InteractionPoint on which this signal instance occured.

Operations

```
copy () : Value

  // Create a new signal instance with the same type, interaction point and feature values as this
signal instance.
  CS_SignalInstance newValue = (CS_SignalInstance) super.copy();
  newValue.type = this.type ;
  newValue.interactionPoint = this.interactionPoint ;
  return newValue;
```

CS_AcceptEventActionActivation

The behavior of fUML CallOperationActionActivation::match() is overriden, in order to account for the fact that a given signal instance may need to be matched with triggers where a list of ports is given. (NOTE: Addresses requirement R9 "Specifying one or more ports for an event implies that the event triggers the execution of an associated behavior only if the event was received via one of the specified ports.")

Generalizations

- AcceptEventActionActivation (from fUML::Semantics::Actions::CompleteActions)

Attributes

- None

Associations

- None

Operations

```
match (signalInstance:SignalInstance) : Boolean

  // Return true if the given signal instance matches a trigger of the accept
  // action of this activation.
  // Matching implies that the type of the signalInstance matches the Signal
  // of one of the triggers.
  // When the type matches with the Signal, and if the trigger specifies a
  // list of ports,
  // the signalInstance matches the trigger only if it occurred on a port
  // identified in the list.

  AcceptEventAction action = (AcceptEventAction)(this.node) ;
  TriggerList triggers = action.trigger ;
  Signal signal = signalInstance.type ;

  Boolean matches = false;
  Integer i = 1;
  while (!matches & i <= triggers.size()) {
    Trigger t = triggers.getValue(i-1) ;
    matches = ((SignalEvent)t.event).signal == signal ;
    if (matches) {
      if (! (signalInstance instanceof CS_SignalInstance)) {
        matches = false ;
      }
      else {
        PortList portsOfTrigger = t.port ;
        Port onPort =
          ((CS_SignalInstance)signalInstance).interactionPoint
                                            .definingPort ;
        Boolean portMatches = false ;
        Integer j = 1 ;
        while (! portMatches & j <= portsOfTrigger.size() ) {
          portMatches = onPort == portsOfTrigger.getValue(j-1) ;
          j = j + 1 ;
        }
        matches = portMatches ;
      }
    }
   i = i + 1;
  }

  return matches;
```

CS_CallOperationActionActivation

Extends fUML CallOperationActionActivation::getCallExecution(). If onPort is specified, instead of dispatching directly to target reference by calling operation dispatch, dispatchOut (cf. ReferenceToCompositeStructure) is called, so that the operation call will be finally dispatched to the environment (from where the execution will be taken). (Note: Adresses requirement R2 "Invocation actions may also be sent to a target via a given port, either on the sending object or on another object.")

Generalizations

- CallOperationActionActivation (from fUML::Semantics::Actions::BasicActions)

Attributes

- None

Associations

- None

Operations

```
getCallExecution () : Execution

  // If onPort is not specified, behaves like in fUML
  // If onPort is specified, and if the value on the target input pin is a
  // reference, dispatch the operation
  // to it and return the resulting execution object.
  // As compared to fUML, instead of dispatching directly to target reference
  // by calling operation dispatch,
 // dispatchOut is called, so that the operation call will be finally
  // dispatched to the environment (from where the execution will be taken).

  CallOperationAction action = (CallOperationAction)(this.node);
  Execution execution = null ;
  if (action.onPort == null ) {
    execution = super.getCallExecution() ;
  }
  else {
    Value target = this.takeTokens(action.target).getValue(0);
    if (target instanceof CS_Reference) {
     execution = ((CS_Reference)target)
                     .dispatchOut(action.operation, action.onPort);
    }
  }
  return execution;
```