Date: April 2012

This is the OMG submission template in Word and ODT. The OMG document number is pas/2012-04-08. The template was updated with a new Cover and Preface and supersedes OMG document pas/2008-10-01.

Precise Semantics of UML Composite Structures

Version 1.x, 2.x.x, etc.

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

Sample: The machine readable file(s) URL 20120401 would be used when the schema file document numbers are ptc/12-04-14, ptc/2012-04-15, ptc/2012-04-16. If the machine readable files are in a .zip file, please list each file and URL in your Inventory report.

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

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

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.

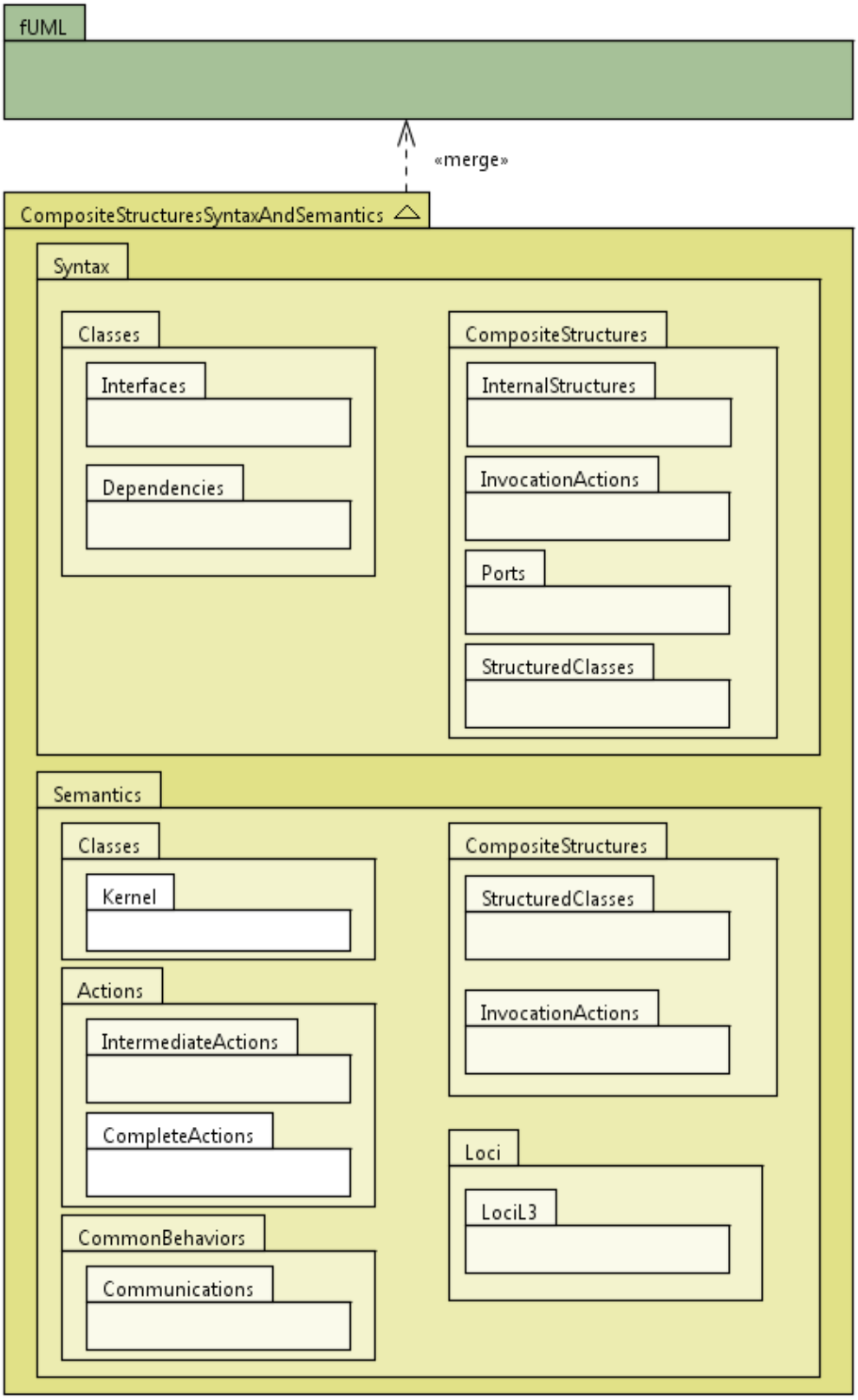

Figure 1: Package architecture

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 CommonBehaviors

7.2.1 Overview

7.2.2 Communications

7.2.2.1 Overview

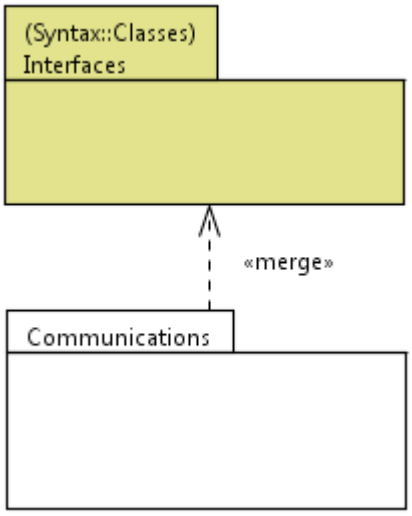

Figure 1: Communications package relationships diagram

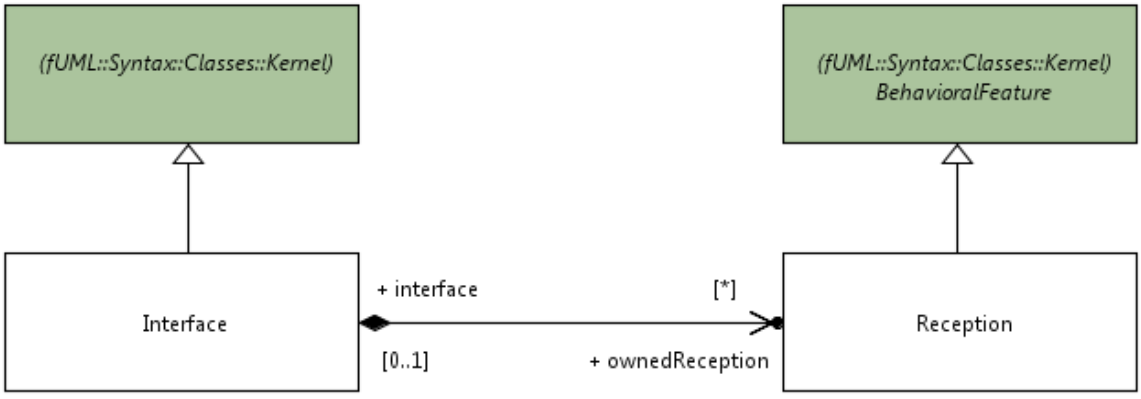

Figure 2: Communications diagram

TODO.

7.2.2.2 Class descriptions

1.1.1.1.1 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

1.1.1.1.2 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.

1.1 CompositeStructures

1.1.1 Overview

1.1.1 Ports

1.1.1.1 Overview

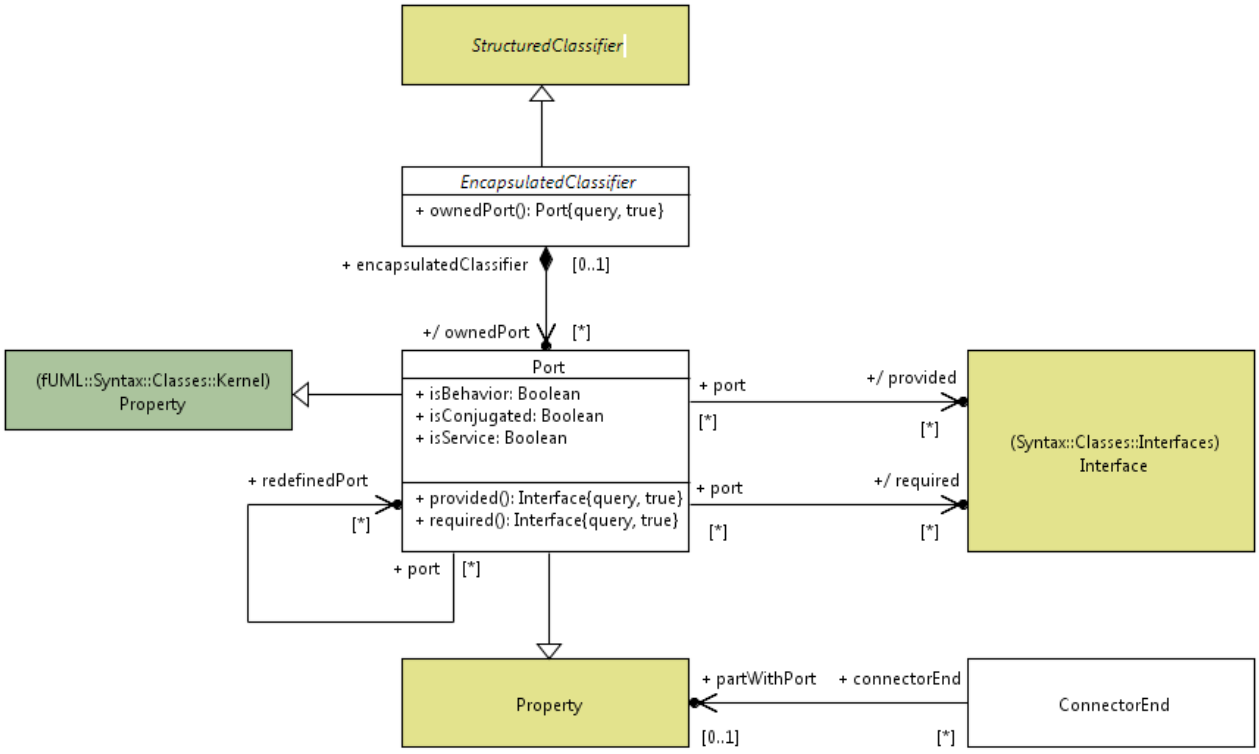

Figure 3: Ports diagram

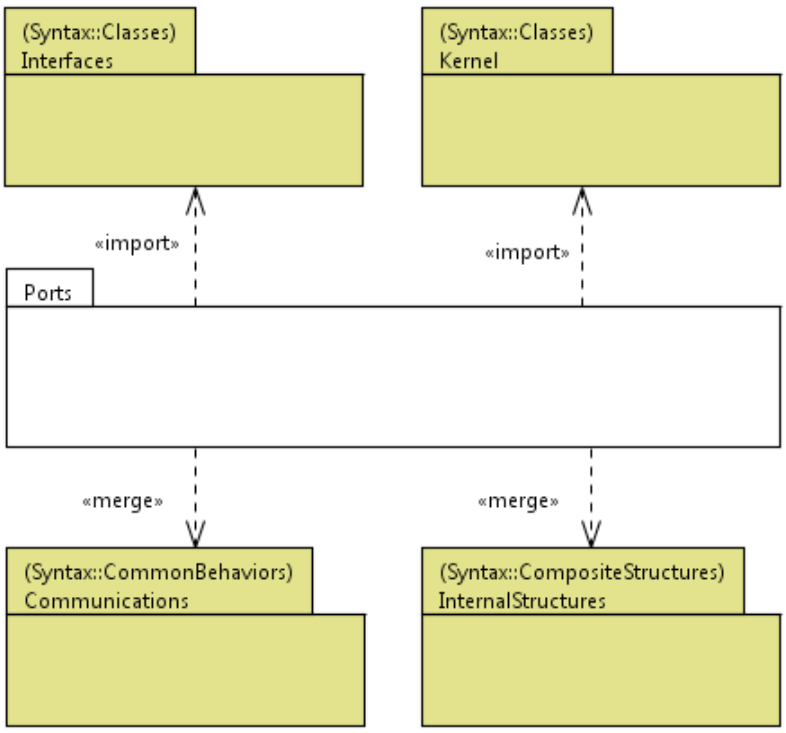

Figure 4: Ports package relationships diagram

TODO.

1.1.1.1 Class descriptions

1.1.1.1.3 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.

1.1.1.1.4 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.

1.1.1.1.5 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.

1.1.2 StructuredClasses

1.1.2.1 Overview

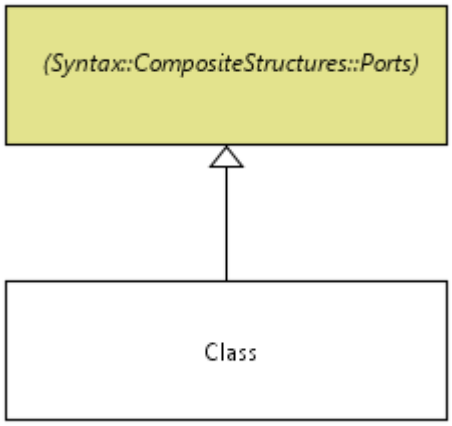

Figure 5: StructuredClasses diagram

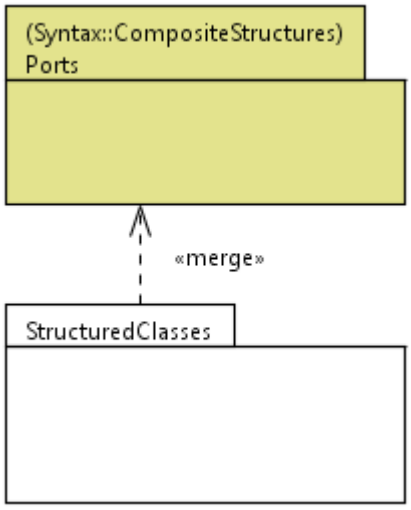

Figure 6: StructuredClasses package relationships diagram

TODO.

1.1.1.2 Class descriptions

1.1.1.1.6 Class

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

Generalizations

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

Attributes

- None

Associations

- None

1.1.3 InternalStructures

1.1.3.1 Overview

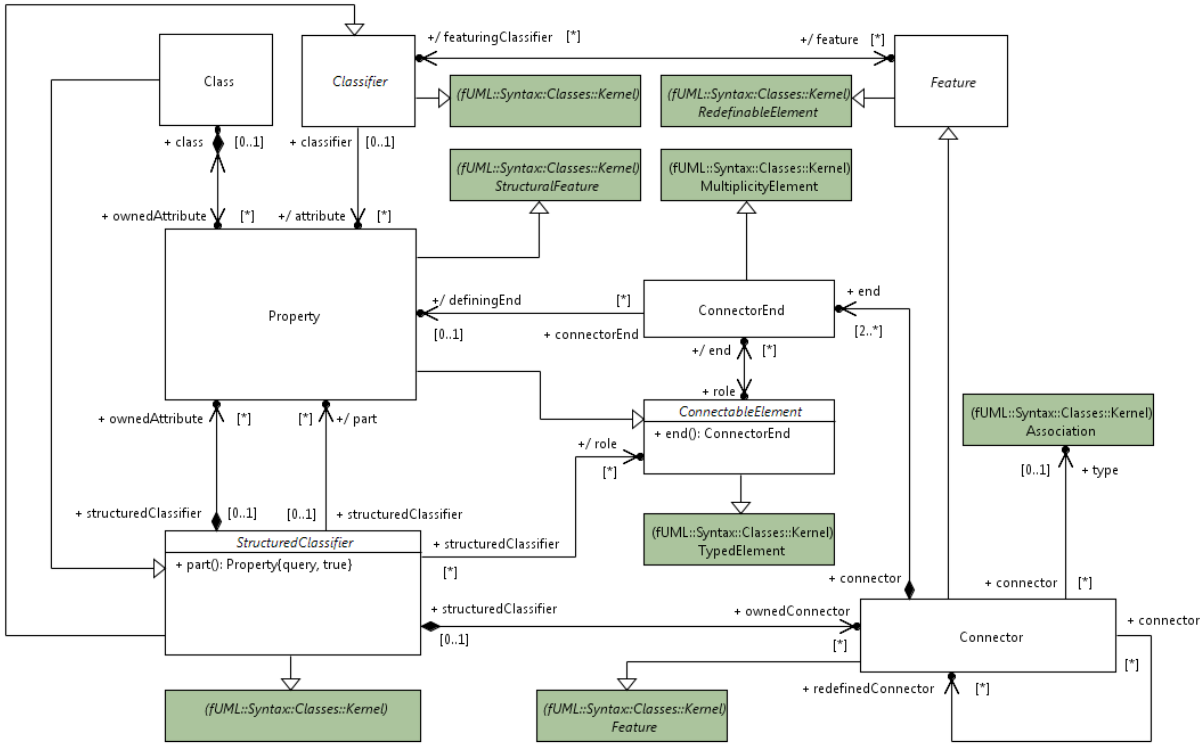

Figure 7: InternalStructures diagram

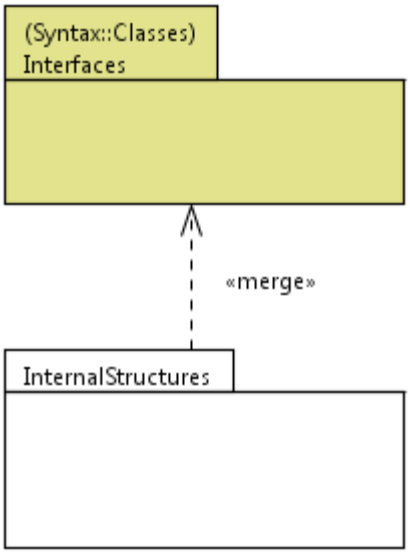

Figure 8: InternalStructures package relationship diagram

TODO.

1.1.1.3 Class descriptions

1.1.1.1.7 Feature

Generalizations

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

Attributes

- None

Associations

- featuringClassifier : Classifier [0..*],

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

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.

1.1.1.1.9 Class

Generalizations

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

Attributes

- None

Associations

- ownedAttribute : Property [0..*],

1.1.1.1.10 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..*],

1.1.1.1.11 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.

1.1.1.1.12 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.

1.1.1.1.13 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.

1.1.1.1.14 Property

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

Generalizations

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

Attributes

- None

Associations

- class : Class [0..1],

1.1.4 InvocationActions

1.1.4.1 Overview

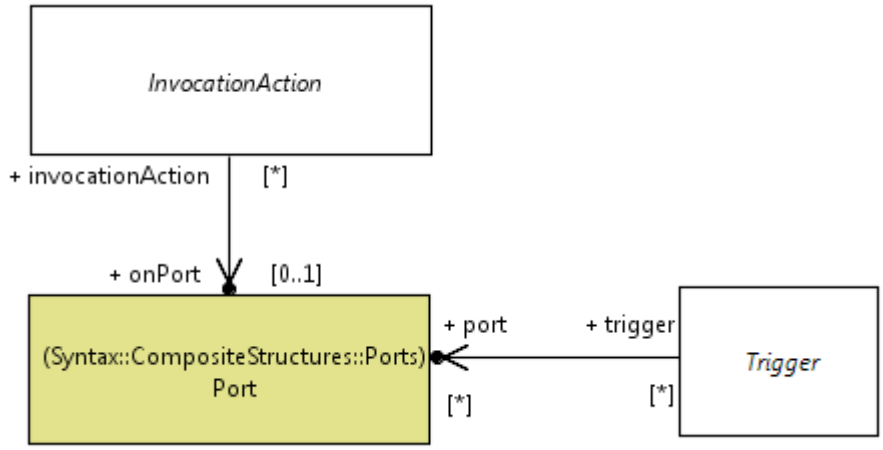

Figure 9: InvocationActions diagram

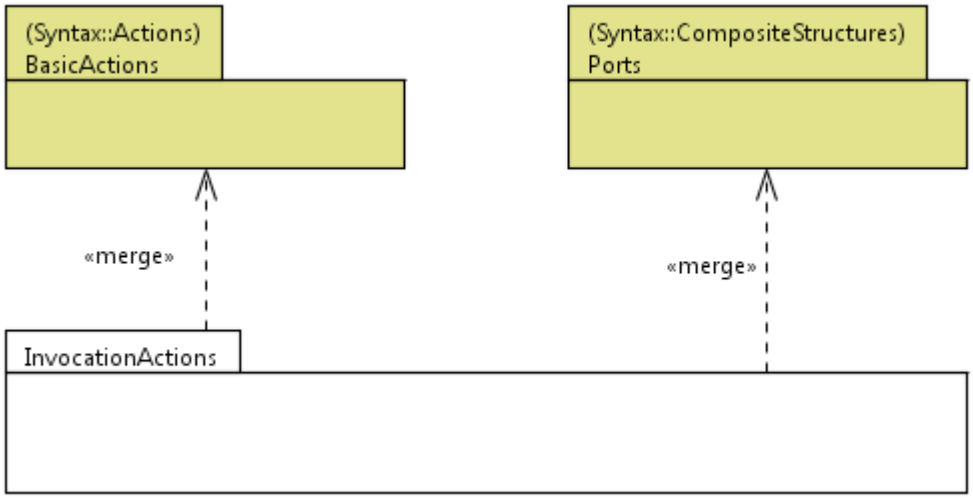

Figure 10: InvocationActions package relationships diagram

TODO.

1.1.1.4 Class descriptions

1.1.1.1.15 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.

1.1.1.1.16 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.

1.2 Components

1.2.1 Overview

1.1.5 BasicComponents

1.1.5.1 Overview

Figure 11: BasicComponents diagram

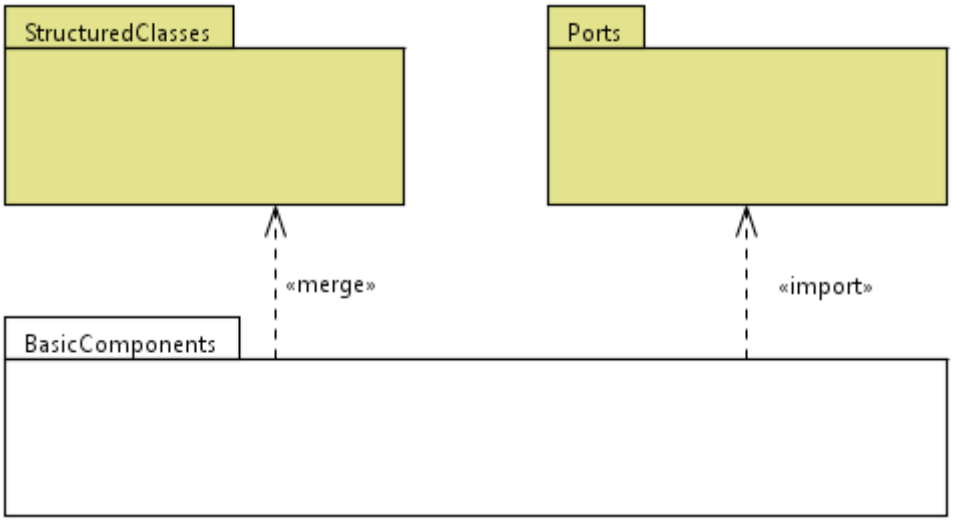

Figure 12: BasicComponents package relationships diagram

TODO.

1.1.1.5 Class descriptions

1.1.1.1.17 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

1.3 Classes

1.3.1 Overview

1.1.6 Dependencies

1.1.6.1 Overview

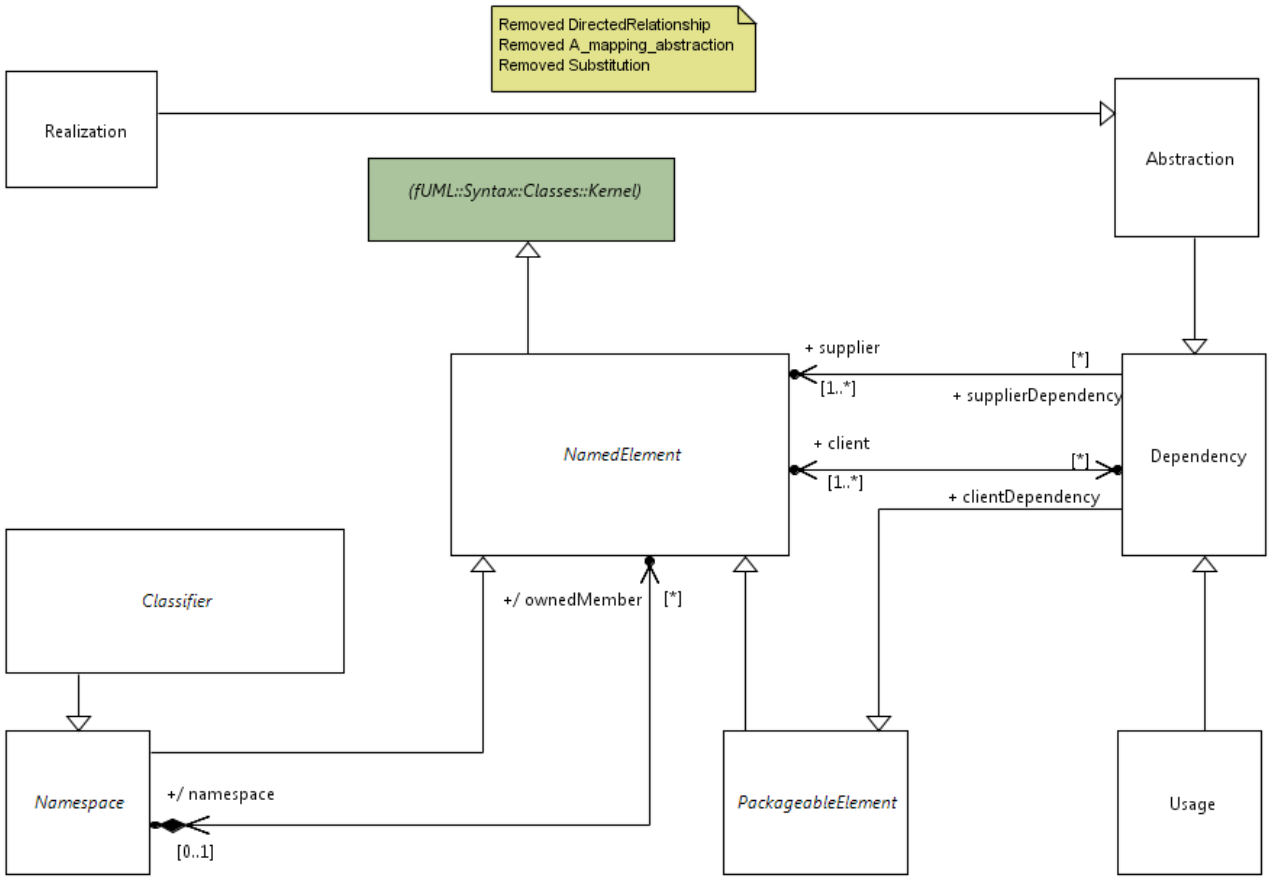

Figure 13: Dependencies diagram

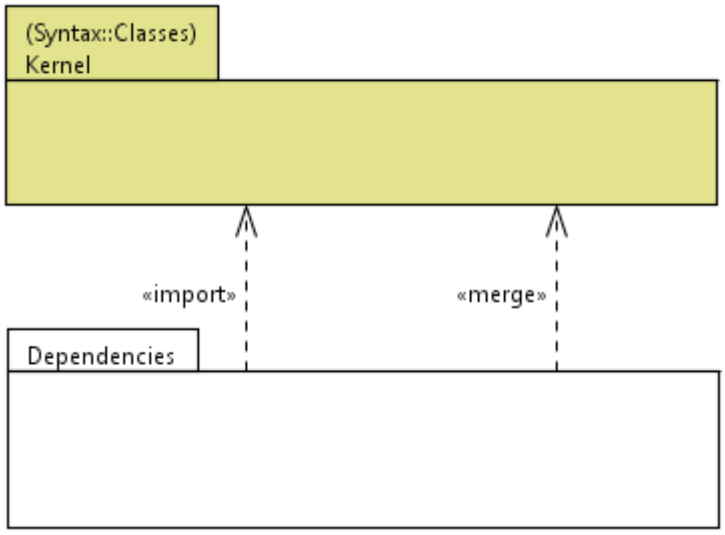

Figure 14: Dependencies package relationships diagram

TODO.

1.1.1.6 Class descriptions

1.1.1.1.18 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

1.1.1.1.19 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

1.1.1.1.20 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.

1.1.1.1.21 Namespace

Generalizations

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

Attributes

- None

Associations

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

1.1.1.1.22 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

1.1.1.1.23 PackageableElement

Generalizations

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

Attributes

- None

Associations

- None

1.1.1.1.24 Classifier

Generalizations

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

Attributes

- None

Associations

- None

1.1.1.1.25 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.

1.1.7 Interfaces

1.1.7.1 Overview

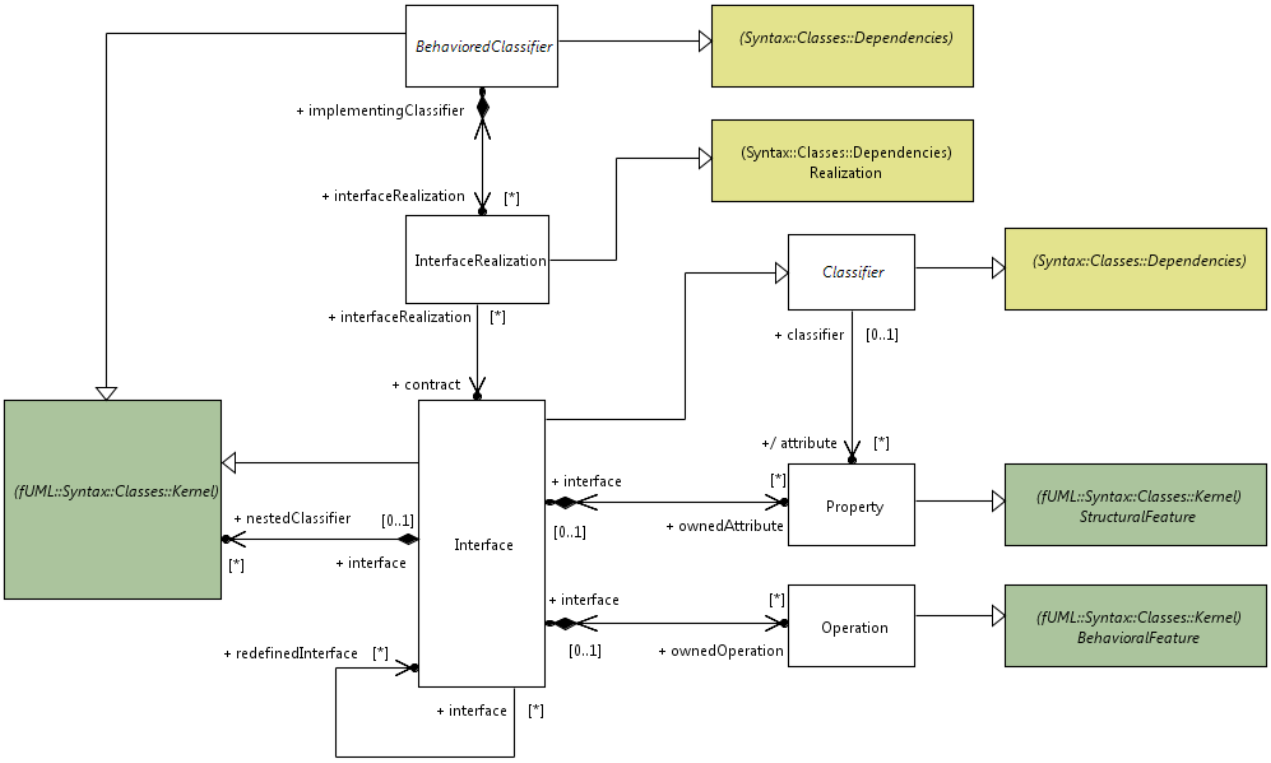

Figure 15: Interfaces diagram

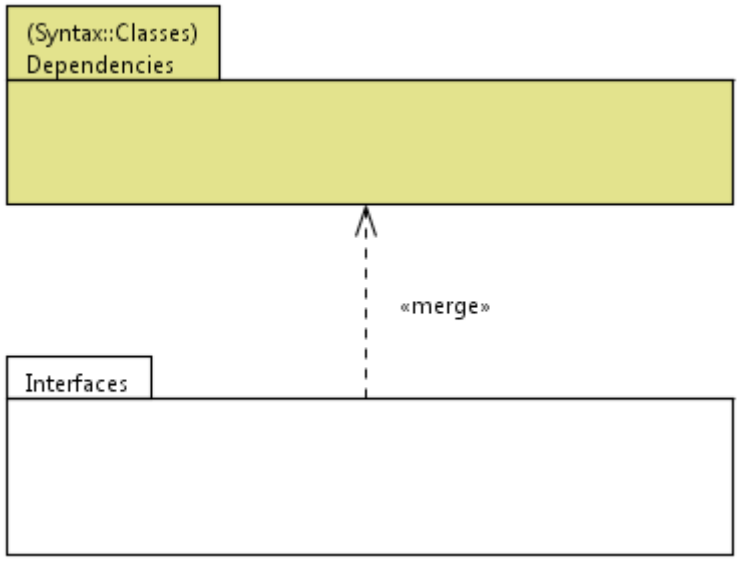

Figure 16: Interfaces package relationships diagram

TODO.

1.1.1.7 Class descriptions

1.1.1.1.26 BehavioredClassifier

A behaviored classifier may have an interface realization.

Generalizations

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

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.

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

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.

1.1.1.1.28 Property

Generalizations

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

Attributes

- None

Associations

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

1.1.1.1.29 Classifier

Generalizations

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

Attributes

- None

Associations

- attribute : Property [0..*],

1.1.1.1.30 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).

1.1.1.1.31 Operation

Generalizations

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

Attributes

- None

Associations

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

8 Semantics

8.1 Overview

TODO.

8.2 Loci

8.2.1 Overview

TODO.

8.2.2 LociL3

8.2.2.1 Overview

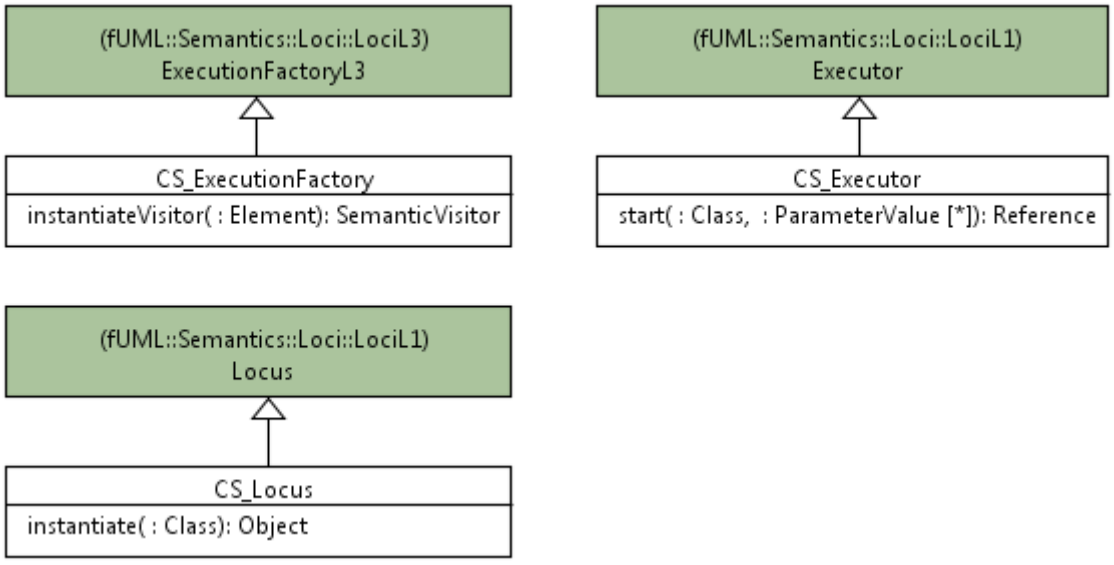

Figure 17: LociL3 diagram

TODO.

8.2.2.2 Class descriptions

1.1.1.1.32 CS_Locus

Extends fUML semantics by instantiating a CS_Object in the case where type is not a Behavior. Otherwise behaves like in fUML.

Generalizations

- Locus (from fUML::Semantics::Loci::LociL1)

Attributes

- None

Associations

- None

Operations

```
public instantiate(type:Class) : Object

    // Extends fUML semantics by instantiating a CS_Object
    // in the case where type is not a Behavior.
    // Otherwise behaves like in fUML

    Object_ object = null;

    if (type instanceof Behavior) {
      object = super.instantiate(type);
    } else {
      object = new CS_Object() ;
      object.types.addValue(type);
      object.createFeatureValues();
      this.add(object);
    }

    return object;
```

1.1.1.1.33 CS_ExecutionFactory

Extends fUML semantics in the sense that newly introduced semantic visitors are instantiated instead of fUML visitors.

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public instantiateVisitor(element:Element) : SemanticVisitor

  // Extends fUML semantics in the sense that newly introduced
  // semantic visitors are instantiated instead of fUML visitors

  SemanticVisitor visitor = null ;
  if (element instanceof ReadExtentAction) {
    visitor =new CS_ReadExtentActionActivation() ;
  }
  else if (element instanceof AddStructuralFeatureValueAction) {
```

```
    visitor = new CS_AddStructuralFeatureValueActionActivation() ;
  }
  else if (element instanceof CreateLinkAction) {
    visitor = new CS_CreateLinkActionActivation() ;
  }
  else if (element instanceof CreateObjectAction) {
    visitor = new CS_CreateObjectActionActivation() ;
  }
  else if (element instanceof ReadSelfAction) {
    visitor = new CS_ReadSelfActionActivation() ;
  }
  else if (element instanceof InstanceValue) {
    visitor = new CS_InstanceValueEvaluation() ;
  }
  else if (element instanceof AcceptEventAction) {
    visitor = new CS_AcceptEventActionActivation() ;
  }
  else if (element instanceof CallOperationAction) {
    visitor = new CS_CallOperationActionActivation() ;
  }
  else if (element instanceof SendSignalAction) {
    visitor = new CS_SendSignalActionActivation() ;
  }
  else {
    visitor = super.instantiateVisitor(element) ;
  }
  return visitor ;
```

1.1.1.1.34 CS_Executor

fUML semantics is extended in the sense that when the instantiated object is a CS_Object, a CS_Reference is returned (instead of a Reference). [Note: this can be avoided if fUML introduces a factory for Reference]

Generalizations

- Executor (from fUML::Semantics::Loci::LociL1)

Attributes

- None

Associations

- None

Operations

```
public start(type:Class, inputs:ParameterValue[*]) : Reference

    // Instantiate the given class and start any behavior of the resulting
    // object.
    // (The behavior of an object includes any classifier behaviors for an
    // active object or the class of the object itself, if that is a
    // behavior.)
    // fUML semantics is extended in the sense that when the instantiated object
    // is a CS_Object, a CS_Reference is returned (instead of a Reference)

    Debug.println("[start] Starting " + type.name + "...");

    Object_ object = this.locus.instantiate(type);

    Debug.println("[start] Object = " + object);
    object.startBehavior(type, inputs);

    Reference reference ;
    if (object instanceof CS_Object) {
      reference = new CS_Reference();
```

```
    ((CS_Reference)reference).compositeReferent = (CS_Object)object ;
  }
  else {
    reference = new Reference() ;
  }
  reference.referent = object;

  return reference;
```

1.1 Classes

1.1.1 Overview

TODO.

1.1.1 Kernel

1.1.1.1 Overview

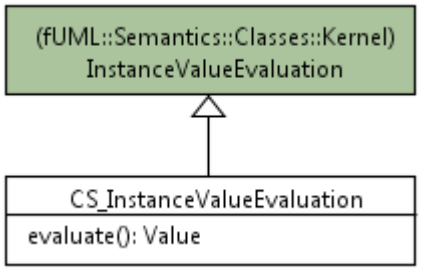

Figure 18: Kernel diagram

TODO.

1.1.1.1 Class descriptions

1.1.1.1.35 CS_InstanceValueEvaluation

Extends fUML semantics in the sense that when the instance specification is for an object which is not typed by a Behavior, a CS_Reference (to a CS_Object) is produced instead of a Reference (to an Object)

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public evaluate() : Value

    // If the instance specification is for an enumeration, then return the
    // identified enumeration literal.
    // If the instance specification is for a data type (but not a primitive
    // value or an enumeration), then create a data value of the given data
    // type.
    // If the instance specification is for an object, then create an object
    // at the current locus with the specified types.
    // Set each feature of the created value to the result of evaluating the
    // value specifications for the specified slot for the feature.
    // Extends fUML semantics in the sense that when the instance specification
    // is for an object which is not typed by a Behaviore, A CS_Reference (to a
    // CS_Object) is produced instead of a Reference (to an Object)

    // Debug.println("[evaluate] InstanceValueEvaluation...");

    InstanceSpecification instance = ((InstanceValue) this.specification).instance;
    ClassifierList types = instance.classifier;
    Classifier myType = types.getValue(0);

    Debug.println("[evaluate] type = " + myType.name);

    Value value;
    if (instance instanceof EnumerationLiteral) {
      // Debug.println("[evaluate] Type is an enumeration.");
      EnumerationValue enumerationValue = new EnumerationValue();
      enumerationValue.type = (Enumeration) myType;
      enumerationValue.literal = (EnumerationLiteral) instance;
      value = enumerationValue;
    }
    else {
      StructuredValue structuredValue = null;

      if (myType instanceof DataType) {
        // Debug.println("[evaluate] Type is a data type.");
        DataValue dataValue = new DataValue();
        dataValue.type = (DataType) myType;
        structuredValue = dataValue;
      }
      else {
        Object_ object = null;
        if (myType instanceof Behavior) {
          // Debug.println("[evaluate] Type is a behavior.");
          object = this.locus.factory.createExecution(
              (Behavior) myType, null);
        }
        else {
          // Debug.println("[evaluate] Type is a class.");
          object = new CS_Object();
          for (int i = 0; i < types.size(); i++) {
            Classifier type = types.getValue(i);
            object.types.addValue((Class_) type);
          }
        }

        this.locus.add(object);

        Reference reference ;
        if (object instanceof CS_Object) {
          reference = new CS_Reference();
          ((CS_Reference)reference).compositeReferent = (CS_Object)object ;
        }
        else {
          reference = new Reference() ;
        }
        reference.referent = object;
        structuredValue = reference;
      }

      structuredValue.createFeatureValues();

      // Debug.println("[evaluate] " + instance.slot.size() +
      // " slot(s).");
```

```
    SlotList instanceSlots = instance.slot;
    for (int i = 0; i < instanceSlots.size(); i++) {
      Slot slot = instanceSlots.getValue(i);
      ValueList values = new ValueList();

      // Debug.println("[evaluate] feature = " +
      // slot.definingFeature.name + ", " + slot.value.size() +
      // " value(s).");
      ValueSpecificationList slotValues = slot.value;
      for (int j = 0; j < slotValues.size(); j++) {
        ValueSpecification slotValue = slotValues.getValue(j);
        // Debug.println("[evaluate] Value = " +
        // slotValue.getClass().getName());
        values.addValue(this.locus.executor.evaluate(slotValue));
      }
      structuredValue.setFeatureValue(slot.definingFeature, values, 0);
    }

    value = structuredValue;
  }

  return value;
```

1.2 Actions

1.2.1 Overview

TODO.

1.1.2 CompleteActions

1.1.2.1 Overview

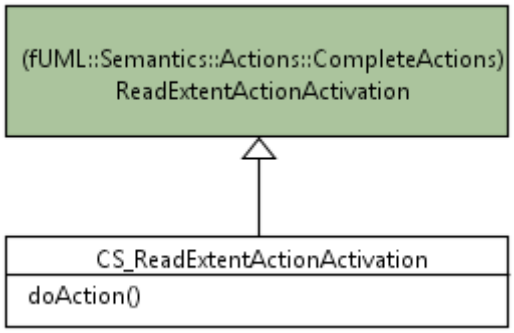

Figure 19: CompleteActions diagram

TODO.

1.1.1.2 Class descriptions

1.1.1.1.36 CS_ReadExtentActionActivation

Extends default fUML semantics in the sense that produced tokens contain CS_References instead of References, in the case where the object is a CS_Object. [Note: This extension can be avoided if fUML introduces a factory for Reference]

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public doAction()

    // Get the extent, at the current execution locus, of the classifier
    // (which must be a class) identified in the action.
    // Place references to the resulting set of objects on the result pin.
    // Extends default fUML semantics in the sense that produced tokens contain
    // CS_References instead of References, in the case where the object is a
    // CS_Object

    ReadExtentAction action = (ReadExtentAction) (this.node);
    ExtensionalValueList objects = this.getExecutionLocus().getExtent(
        action.classifier);

    // Debug.println("[doAction] " + action.classifier.name + " has " +
    // objects.size() + " instance(s).");

    ValueList references = new ValueList();
    for (int i = 0; i < objects.size(); i++) {
      Value object = objects.getValue(i);
      Reference reference = null ;
      if (object instanceof CS_Object) {
        reference = new CS_Reference() ;
        ((CS_Reference)reference).compositeReferent = (CS_Object)object ;
      }
      else {
        reference = new Reference() ;
      }
      reference.referent = (Object_) object;
      references.addValue(reference);
    }

    this.putTokens(action.result, references);
```

1.1.3 IntermediateActions

1.1.3.1 Overview

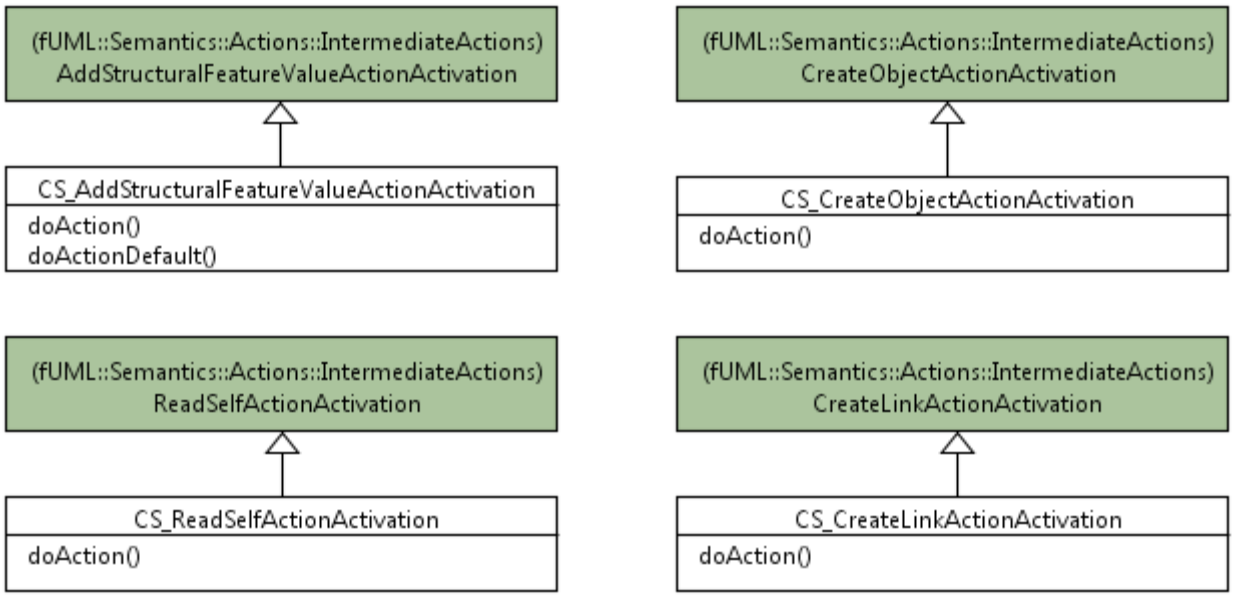

Figure 20: IntermediateActions diagram

TODO.

1.1.1.3 Class descriptions

1.1.1.1.37 CS_ReadSelfActionActivation

Extends fUML semantics in the sense that the reference placed on the result pin is a CS_Reference, not a Reference. [Note: this extension can be avoided if fUML introduces a factory for instantiating Reference]

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public doAction()

    // Get the context object of the activity execution containing this
    // action activation and place a reference to it on the result output
    // pin.
    // Extends fUML semantics in the sense that the reference placed on
```

```
    // the result pin is a CS_Reference, not a Reference

    // Debug.println("[ReadSelfActionActivation] Start...");

    CS_Reference context = new CS_Reference();
    context.referent = this.getExecutionContext();
    context.compositeReferent = (CS_Object)context.referent ;

    // Debug.println("[ReadSelfActionActivation] context object = " +
    // context.referent);

    OutputPin resultPin = ((ReadSelfAction) (this.node)).result;
    this.putToken(resultPin, context);
```

1.1.1.1.38 CS_CreateLinkActionActivation

fUML semantics is extended in the sense that a CS_Link is created instead of a Link. [Note: This extension can be avoided if fUML introduces a factory for Link]

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public doAction()

    // Get the extent at the current execution locus of the association for
    // which a link is being created.
    // Destroy all links that have a value for any end for which
    // isReplaceAll is true.
    // Create a new link for the association, at the current locus, with the
    // given end data values,
    // inserted at the given insertAt position (for ordered ends).
    // fUML semantics is extended in the sense that a CS_Link is created instead of
    // a Link

    CreateLinkAction action = (CreateLinkAction) (this.node);
    LinkEndCreationDataList endDataList = action.endData;

    Association linkAssociation = this.getAssociation();
    ExtensionalValueList extent = this.getExecutionLocus().getExtent(
        linkAssociation);

    Link oldLink = null;
    for (int i = 0; i < extent.size(); i++) {
      ExtensionalValue value = extent.getValue(i);
      Link link = (Link) value;

      boolean noMatch = true;
      int j = 1;
      while (noMatch & j <= endDataList.size()) {
        LinkEndCreationData endData = endDataList.getValue(j - 1);
        if (endData.isReplaceAll
            & this.endMatchesEndData(link, endData)) {
          oldLink = link;
          link.destroy();
          noMatch = false;
        }
```

```
      j = j + 1;
    }
  }

  CS_Link newLink = new CS_Link();
  newLink.type = linkAssociation;

  // This necessary when setting a feature value with an insertAt position
  newLink.locus = this.getExecutionLocus();

  for (int i = 0; i < endDataList.size(); i++) {
    LinkEndCreationData endData = endDataList.getValue(i);

    int insertAt;
    if (endData.insertAt == null) {
      insertAt = 0;
    } else {
      insertAt = ((UnlimitedNaturalValue) (this
          .takeTokens(endData.insertAt).getValue(0))).value.naturalValue;
      if (oldLink != null) {
        if (oldLink.getFeatureValue(endData.end).position < insertAt) {
          insertAt = insertAt - 1;
        }
      }
    }
    newLink.setFeatureValue(endData.end,
        this.takeTokens(endData.value), insertAt);
  }

  this.getExecutionLocus().add(newLink);
```

1.1.1.1.39 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 interaction point is created on the basis of the given Reference. It then behaves like in fUML, except that the execution continues using the created interaction point instead of the given Reference.

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public 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
    this.doActionDefault() ;
```

```
    }
    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
        this.doActionDefault() ;
      }
    }


public doActionDefault()

    // Get the values of the object and value input pins.
    // If the given feature is an association end, then create a link
```

```
// between the object and value inputs.
// Otherwise, if the object input is a structural value, then add a
// value to the values for the feature.
// If isReplaceAll is true, first remove all current matching links or
// feature values.
// If isReplaceAll is false and there is an insertAt pin, insert the
// value at the appropriate position.
// This operation captures same semantics as fUML
// AddStructuralFeatureValueActionActivation.doAction(), except that
// when the feature is an association end, a CS_Link will be created instead
// of a Link

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

Value value = this.takeTokens(action.object).getValue(0);
ValueList inputValues = this.takeTokens(action.value);

// NOTE: Multiplicity of the value input pin is required to be 1..1.
Value inputValue = inputValues.getValue(0);

int insertAt = 0;
if (action.insertAt != null) {
  insertAt = ((UnlimitedNaturalValue) this
      .takeTokens(action.insertAt).getValue(0)).value.naturalValue;
}

if (association != null) {
  LinkList links = this.getMatchingLinks(association, feature, value);

  Property oppositeEnd = this.getOppositeEnd(association, feature);
  int position = 0;
  if (oppositeEnd.multiplicityElement.isOrdered) {
    position = -1;
  }

  if (action.isReplaceAll) {
    for (int i = 0; i < links.size(); i++) {
      Link link = links.getValue(i);
      link.destroy();
    }
  } else if (feature.multiplicityElement.isUnique) {
    for (int i = 0; i < links.size(); i++) {
      Link link = links.getValue(i);
      FeatureValue featureValue = link.getFeatureValue(feature);
      if (featureValue.values.getValue(0).equals(inputValue)) {
        position = link.getFeatureValue(oppositeEnd).position;
        if (insertAt > 0 & featureValue.position < insertAt) {
          insertAt = insertAt - 1;
        }
        link.destroy();
      }
    }
  }

  CS_Link newLink = new CS_Link();
  newLink.type = association;

  // This necessary when setting a feature value with an insertAt
  // position
  newLink.locus = this.getExecutionLocus();

  newLink.setFeatureValue(feature, inputValues, insertAt);

  ValueList oppositeValues = new ValueList();
  oppositeValues.addValue(value);
  newLink.setFeatureValue(oppositeEnd, oppositeValues, position);

  newLink.locus.add(newLink);

} else if (value instanceof StructuredValue) {
  StructuredValue structuredValue = (StructuredValue) value;

  if (action.isReplaceAll) {
    structuredValue.setFeatureValue(feature, inputValues, 0);
  } else {
```

```
      FeatureValue featureValue = structuredValue
          .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
        int 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);
      }
    }
  }

  if (action.result != null) {
    this.putToken(action.result, value);
  }
```

1.1.1.1.40 CS_CreateObjectActionActivation

Extends fUML semantics in the sense that the reference placed on the result pin is a CS_Reference (in the case where the instantiated object is a CS_Object) not a Reference. [Note: this extension can be avoided if fUML introduces a factory for Reference]

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
public doAction()

    // Create an object with the given classifier (which must be a class) as
    // its type, at the same locus as the action activation.
    // Place a reference to the object on the result pin of the action.
    // Extends fUML semantics in the sense that the reference placed
    // on the result pin is a CS_Reference (in the case where the instantiated object
    // is a CS_Object) not a Reference
    // Note that Locus.instantiate(Class) is extended in this specification
    // to produce a CS_Object instead of an Object in the case where the class
    // to be instantiated is not a behavior
```

```
CreateObjectAction action = (CreateObjectAction) (this.node);

Reference reference ;
Object_ referent = this.getExecutionLocus().instantiate((Class_) (action.classifier));
if (referent instanceof CS_Object) {
  reference = new CS_Reference() ;
  ((CS_Reference)reference).compositeReferent = (CS_Object)referent ;
}
else {
  reference = new Reference() ;
}
reference.referent = referent ;

this.putToken(action.result, reference);
```

1.3 CompositeStructures

1.3.1 Overview

TODO.

1.1.4 StructuredClasses

1.1.4.1 Overview

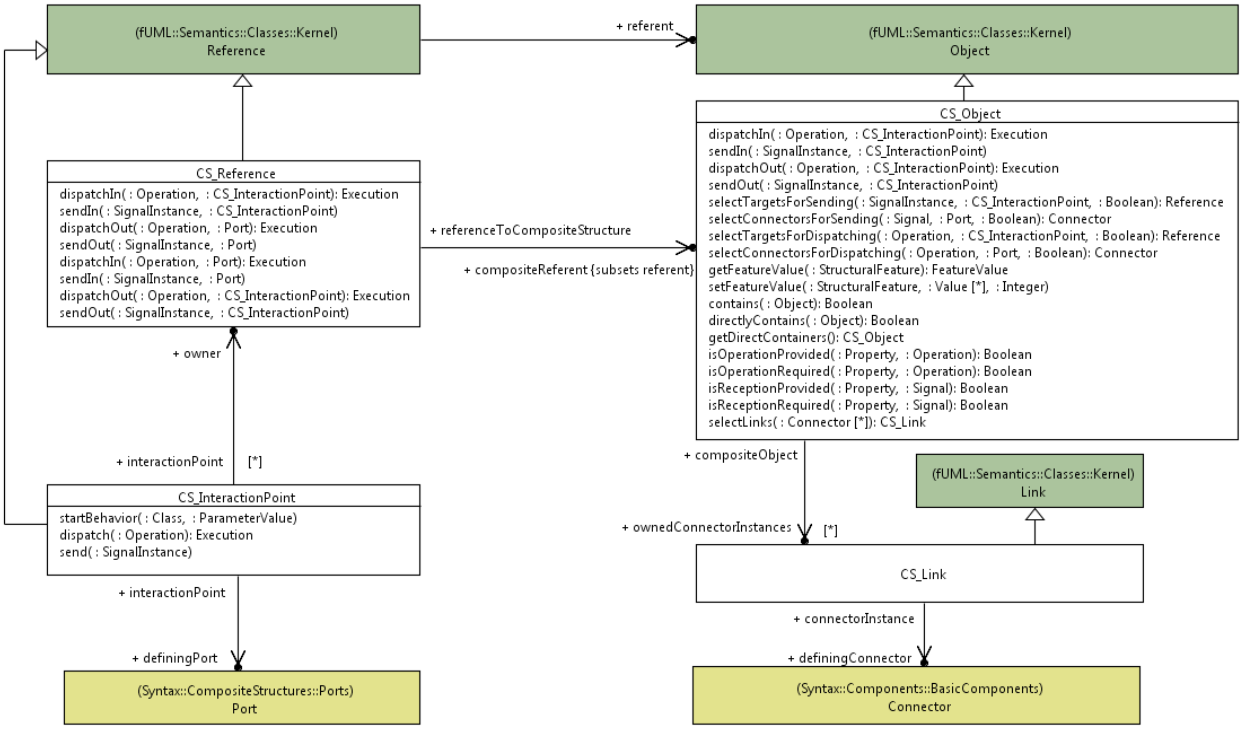

Figure 21: StructuredClasses diagram

TODO.

1.1.1.4 Class descriptions

1.1.1.1.41 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

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

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


public sendIn(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)

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


public sendOut(signalInstance:SignalInstance, onPort:Port)

  // Select a CS_InteractionPoint value playing onPort,
  // and send the signal instance to this interaction point
  FeatureValue featureValue = this.getFeatureValue(onPort) ;
  ValueList values = featureValue.values ;
  ReferenceList potentialTargets = new ReferenceList() ;
  for (int i = 0 ; i < values.size() ; i++) {
    potentialTargets.addValue((Reference)values.getValue(i)) ;
  }
  CS_RequestPropagationStrategy strategy =
      (CS_RequestPropagationStrategy)this.referent.locus.factory.getStrategy("requestPropagation") ;
  ReferenceList targets = strategy.select(potentialTargets, new SendSignalActionActivation()) ;
  for (int i = 0 ; i < targets.size() ; i++) {
    CS_InteractionPoint target = (CS_InteractionPoint)targets.getValue(i) ;
    this.compositeReferent.sendOut(signalInstance, target) ;
  }


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

  // Select a CS_InteractionPoint value playing onPort,
  // and dispatches the operation to this interaction point
  Execution execution = null ;
  FeatureValue featureValue = this.getFeatureValue(onPort) ;
```

```
  ValueList values = featureValue.values ;
  ReferenceList potentialTargets = new ReferenceList() ;
  for (int i = 0 ; i < values.size() ; i++) {
    potentialTargets.addValue((Reference)values.getValue(i)) ;
  }
  CS_RequestPropagationStrategy strategy =
      (CS_RequestPropagationStrategy)this.referent.locus.factory.getStrategy("requestPropagation") ;
  ReferenceList targets = strategy.select(potentialTargets, new CallOperationActionActivation()) ;
  // if targets is empty, no dispatch target has been found,
  // and the operation call is lost
  if (targets.size() >= 1) {
    CS_InteractionPoint target = (CS_InteractionPoint)targets.getValue(1) ;
    execution = this.compositeReferent.dispatchOut(operation, target) ;
  }
  return execution ;


public dispatchIn(operation:Operation, onPort:Port) : Execution

  // Select a CS_InteractionPoint value playing onPort,
  // and dispatches the operation call to this interaction point
  FeatureValue featureValue = this.getFeatureValue(onPort) ;
  ValueList values = featureValue.values ;
  Integer choice = ((ChoiceStrategy) this.referent.locus.factory
      .getStrategy("choice"))
      .choose(featureValue.values.size()) - 1;
  CS_InteractionPoint interactionPoint = (CS_InteractionPoint)values.getValue(choice) ;
  return interactionPoint.dispatch(operation) ;


public sendIn(signalInstance:SignalInstance, onPort:Port)

  // Select a Reference value playing onPort,
  // and send the signal instance to this interaction point
  FeatureValue featureValue = this.getFeatureValue(onPort) ;
  ValueList values = featureValue.values ;
  ReferenceList potentialTargets = new ReferenceList() ;
  for (int i = 0 ; i < values.size() ; i++) {
    potentialTargets.addValue((Reference)values.getValue(i)) ;
  }
  CS_RequestPropagationStrategy strategy =
      (CS_RequestPropagationStrategy)this.referent.locus.factory.getStrategy("requestPropagation") ;
  ReferenceList targets = strategy.select(potentialTargets, new SendSignalActionActivation()) ;
  for (int i = 0 ; i < targets.size() ; i++) {
    Reference target = targets.getValue(i) ;
    target.send(signalInstance) ;
  }


public dispatchOut(operation:Operation, interactionPoint:CS_InteractionPoint) : Execution

  // Delegates dispatching (through the interaction point, to the environment)
  // to compositeReferent
  return this.compositeReferent.dispatchOut(operation, interactionPoint) ;


public sendOut(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)

  // Delegates sending (through the interaction point, to the environment)
  // to compositeReferent
  this.compositeReferent.sendOut(signalInstance, interactionPoint) ;
```

1.1.1.1.42 CS_InteractionPoint

A CS_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

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

  // Overriden to do nothing

public dispatch(operation:Operation) : Execution

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

public send(signalInstance:SignalInstance)

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

1.1.1.1.43 CS_Object

CS_Object 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

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

    // If the interaction point refers to a behavior port, does nothing [for the moment... ?],
    // since the only kind of event supported in fUML is SignalEvent
    // If it does not refer to 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 {
      boolean operationIsProvided = true ;
      ReferenceList potentialTargets =
          this.selectTargetsForDispatching(operation, interactionPoint, operationIsProvided) ;
      // If targets is empty, no delegation target have been found,
      // and the operation call will be lost
      if (! (potentialTargets.size()==0)) {
        CS_RequestPropagationStrategy strategy =
            (CS_RequestPropagationStrategy)this.locus.factory.getStrategy("requestPropagation") ;
        // Choose one target non-deterministically
        ReferenceList targets = strategy.select(potentialTargets, new
CallOperationActionActivation()) ;
        Reference target = targets.getValue(0) ;
        execution = target.dispatch(operation) ;
      }
    }
    return execution ;


public sendIn(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)

    // If the interaction is a behavior port,
    // creates a CS_SignalInstance from the signal instance,
    // sets its interaction point,
    // and sends it to the target object 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 = (CS_SignalInstance)signalInstance.copy() ;
      newSignalInstance.interactionPoint = interactionPoint ;
      this.send(newSignalInstance) ;
    }
    else {
      boolean receptionIsProvided = true ;
      ReferenceList potentialTargets =
          this.selectTargetsForSending(signalInstance, interactionPoint, receptionIsProvided) ;
      CS_RequestPropagationStrategy strategy =
          (CS_RequestPropagationStrategy)this.locus.factory.getStrategy("requestPropagation");
      ReferenceList targets = strategy.select(potentialTargets, new SendSignalActionActivation()) ;
      // If targets is empty, no delegation target has been found,
      // and the signal is lost
      // Otherwise, do the following concurrently
      for (int i = 0 ; i < targets.size() ; i++) {
        Reference target = targets.getValue(i) ;
        CS_SignalInstance newSignalInstance = (CS_SignalInstance)signalInstance.copy() ;
        newSignalInstance.interactionPoint = interactionPoint ;
        target.send(newSignalInstance) ;
```

```
      }
    }


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

    // From the given signalInstance and interactionPoint,
    // retrieves potential targets through which request can be delegated.
    // These targets are attached to interactionPoint through CS_Links,
    // defined by connectors respecting the following rule:
    // - if isProvided is true, such a connector can be a delegation connector,
    // where the opposite of interactionPoint.definingPort provides a reception for Signal
signalInstance.type,
    // of it can be an assembly connector, where the opposite of interactionPoint.definingPort
    // requires a reception for Signal signalInstance.type
    // - if isProvided is false, such a connector can be a delegation connector,
    // where the opposite of interactionPoint.definingPort requires a reception for Signal
signalInstance.type,
    // of it can be an assembly connector, where the opposite of interactionPoint.definingPort
    // provides a reception for Signal signalInstance.type
    ConnectorList connectors =
        this.selectConnectorsForSending(signalInstance.type, interactionPoint.definingPort,
isProvided) ;

    // Select links owned by the context object for which the
    // definingConnector is included in the list of matching connectors.
    CS_LinkList connectorInstances = this.selectLinks(connectors) ;

    // 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() ;
    Integer 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 ;


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

    // From the given signal and port, retrieves potential connectors
    // through which request can be delegated
    // These connectors are attached to port,and respect the following rules:
    // - if isProvided is true, such a connector can be a delegation connector,
    // where the opposite of port provides a reception for signal,
    // of it can be an assembly connector, where the opposite of port
    // requires a reception for signal
    // - if isProvided is false, such a connector can be a delegation connector,
    // where the opposite of port requires a reception for signal,
    // of it can be an assembly connector, where the opposite of port
    // provides a reception for signal
    ConnectorList connectors = new ConnectorList() ;
    Integer typeIndex = 1 ;
    // Iterates on types of this CompositeObject
    while (typeIndex <= this.types.size()) {
      Type t =this.types.getValue(typeIndex-1) ;
      if (t instanceof Class_) {
        Class_ class_ = (Class_) t ;
        Integer connectorIndex = 1 ;
        // Iterates on Connectors of the current type
        while (connectorIndex <= class_.encapsulatedClassifier.ownedConnector.size()) {
```

```
          Connector cddConnector =
class_.encapsulatedClassifier.ownedConnector.getValue(connectorIndex-1) ;
          Integer connectorEndIndex = 1 ;
          Boolean matches = false ;
          // Iterates on ConnectorEnds of the current Connector
          while (connectorEndIndex <= cddConnector.end.size() && !matches) {
            ConnectorEnd cddEnd = cddConnector.end.getValue(connectorEndIndex-1) ;
            if (!(cddEnd.role.actualConnectableElement == port)) {
              if (cddConnector.kind == ConnectorKind.delegation) {
                if (isProvided) {
                  if (this.isReceptionProvided(cddEnd.role.actualConnectableElement, signal)) {
                    matches = true ;
                  }
                }
                else if (this.isReceptionRequired(cddEnd.role.actualConnectableElement, signal)) {
                  matches = true ;
                }
              }
              else { // this is an assembly connector
                if (isProvided) {
                  if (this.isReceptionRequired(cddEnd.role.actualConnectableElement, signal)) {
                    matches = true ;
                  }
                }
                else if (this.isReceptionProvided(cddEnd.role.actualConnectableElement, signal)) {
                  matches = true ;
                }
              }
            }
            if (matches = true) {
              connectors.addValue(cddConnector) ;
            }
            connectorEndIndex = connectorEndIndex + 1 ;
          }
          connectorIndex = connectorIndex + 1 ;
        }
        typeIndex = typeIndex + 1 ;
      }
    }

    return connectors ;


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

    // From the given operation and interactionPoint,
    // retrieves potential targets through which request can be delegated.
    // These targets are attached to interactionPoint through CS_Links,
    // defined by connectors respecting the following rules:
    // - if isProvided is true, such a connector can be a delegation connector,
    // where the opposite of interactionPoint.definingPort provides the operation,
    // of it can be an assembly connector, where the opposite of interactionPoint.definingPort
    // requires the operation
    // - if isProvided is false, such a connector can be a delegation connector,
    // where the opposite of interactionPoint.definingPort requires the operation,
    // of it can be an assembly connector, where the opposite of interactionPoint.definingPort
    // provides the operation
    ConnectorList connectors =
        this.selectConnectorsForDispatching(operation, interactionPoint.definingPort, isProvided) ;
    // Select links owned by the context object for which the
    // definingConnector is included in the list of matching connectors.
    CS_LinkList connectorInstances = this.selectLinks(connectors) ;
    // 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() ;
    Integer 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 ;


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

    // From the given signal and port, retrieves potential connectors
    // through which request can be delegated
    // These targets are attached to port, and respect the following rules:
    // - if isProvided is true, such a connector can be a delegation connector,
    // where the opposite of port provides the operation,
    // of it can be an assembly connector, where the opposite of port
    // requires the operation
    // - if isProvided is false, such a connector can be a delegation connector,
    // where the opposite of port requires the operation,
    // of it can be an assembly connector, where the opposite of port
    // provides the operation

    ConnectorList connectors = new ConnectorList() ;
    Integer typeIndex = 1 ;
    // Iterates on types of this CompositeObject
    while (typeIndex <= this.types.size()) {
      Type t =this.types.getValue(typeIndex-1) ;
      if (t instanceof Class_) {
        Class_ class_ = (Class_) t ;
        Integer connectorIndex = 1 ;
        // Iterates on Connectors of the current type
        while (connectorIndex <= class_.encapsulatedClassifier.ownedConnector.size()) {
          Connector cddConnector =
class_.encapsulatedClassifier.ownedConnector.getValue(connectorIndex-1) ;
          if (cddConnector.kind == ConnectorKind.delegation) {
            Integer connectorEndIndex = 1 ;
            Boolean matches = false ;
            // Iterates on ConnectorEnds of the current Connector
            while (connectorEndIndex <= cddConnector.end.size() && !matches) {
              ConnectorEnd cddEnd = cddConnector.end.getValue(connectorEndIndex-1) ;
              if (!(cddEnd.role.actualConnectableElement == port)) {
                if (cddConnector.kind == ConnectorKind.delegation) {
                  if (isProvided) {
                    if (this.isOperationProvided(cddEnd.role.actualConnectableElement, operation)) {
                      matches = true ;
                    }
                  }
                  else if (this.isOperationRequired(cddEnd.role.actualConnectableElement,operation)){
                    matches = true ;
                  }
                }
                else { // this is an assembly connector
                  if (isProvided) {
                    if (this.isOperationRequired(cddEnd.role.actualConnectableElement, operation)) {
                      matches = true ;
                    }
                  }
                  else if (this.isOperationProvided(cddEnd.role.actualConnectableElement,operation)){
                    matches = true ;
                  }
                }
              }
              if (matches == true) {
                connectors.addValue(cddConnector) ;
              }
              connectorEndIndex = connectorEndIndex + 1 ;
            }
          }
          connectorIndex = connectorIndex + 1 ;
        }
        typeIndex = typeIndex + 1 ;
      }
    }
```

```
    return connectors ;


public sendOut(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)

    // select appropriate delegation targets from interactionPoint,
    // and propagates the signal to these targets

    // Retrieves containers for this CS_Object
    CS_ObjectList containers = this.getDirectContainers() ;

    boolean receptionIsNotProvided = false ;
    CS_LinkList connectorInstances = new CS_LinkList() ;
    for (int i = 0 ; i < containers.size() ; i++) {
      CS_Object container = containers.getValue(i) ;
      ConnectorList selectedConnectors =
          container.selectConnectorsForSending(signalInstance.type, interactionPoint.definingPort,
receptionIsNotProvided) ;
      CS_LinkList selectedLinks = container.selectLinks(selectedConnectors) ;
      for (int j = 0 ; j < selectedLinks.size() ; j++) {
        connectorInstances.addValue(selectedLinks.getValue(j)) ;
      }
    }

    // For each matching link, retrieves the end value opposite
    // to interactionPoint.
    // If values opposite to this interaction point are references
    // (which means that it is possible to send them a signal),
    // they are added in the list of potential targets.
    ReferenceList targetsForSendingIn = new ReferenceList() ;
    ReferenceList targetsForSendingOut = new ReferenceList() ;
    ReferenceList targets = new ReferenceList() ;
    Integer i = 1 ;
    while (i <= connectorInstances.size()) {
      CS_Link connectorInstance = connectorInstances.getValue(i-1) ;
      Association association = connectorInstance.type ;
      Property oppositeEnd = association.memberEnd.getValue(0);
      if (oppositeEnd == interactionPoint.definingPort) {
        oppositeEnd = association.memberEnd.getValue(1);
      }
      Value value = connectorInstance.getFeatureValue(oppositeEnd).values.getValue(0) ;
      if (value instanceof Reference) {
        if (connectorInstance.definingConnector.kind == ConnectorKind.assembly) {
          targetsForSendingIn.addValue((Reference)value) ;
        }
        else {
          targetsForSendingOut.addValue((Reference)value) ;
        }
      }
      targets.add((Reference)value) ;
      i = i + 1;
    }

    CS_RequestPropagationStrategy strategy =
(CS_RequestPropagationStrategy)this.locus.factory.getStrategy("propagationRequest") ;
    ReferenceList selectedTargets = strategy.select(targets, new SendSignalActionActivation()) ;

    for (int j = 0 ; j < selectedTargets.size() ; j++) {
      Reference target = selectedTargets.getValue(j) ;
      boolean signalSent = false ;
      for (int k = 0 ; k < targetsForSendingIn.size() && !signalSent ; k++) {
        Reference cddTarget = targetsForSendingIn.getValue(k) ;
        if (cddTarget == target) {
          target.send(signalInstance) ;
          signalSent = true ;
        }
      }
      for (int k = 0 ; k < targetsForSendingOut.size() && !signalSent ; k++) {
        // The target must be an interaction point
        CS_InteractionPoint cddTarget = (CS_InteractionPoint)targetsForSendingOut.getValue(k) ;
        if (cddTarget == target) {
          CS_Reference owner = cddTarget.owner ;
          owner.sendOut(signalInstance, cddTarget) ;
        }
      }
    }
```

```
public dispatchOut(operation:Operation, interactionPoint:CS_InteractionPoint) : Execution

    // Propagate the operation call through interactionPoint,
    // following appropriate links.
    // These links are owned be the CS_Objects containing this CS_Object,
    // and they are attached to the interactionPoint given as a parameter.
    // If such a connector is a delegation connector,
    // it must be connected to a port which also requires operation,
    // If such a connector is an assembly connector,
    // it must be connected to a connectable element which provides operation

    Execution execution = null ;

    // Retrieves containers for this CS_Object
    CS_ObjectList containers = this.getDirectContainers() ;

    boolean operationIsNotProvided = false ; // i.e. it is required
    CS_LinkList connectorInstances = new CS_LinkList() ;
    for (int i = 0 ; i < containers.size() ; i++) {
      CS_Object container = containers.getValue(i) ;
      ConnectorList selectedConnectors =
          container.selectConnectorsForDispatching(operation, interactionPoint.definingPort,
operationIsNotProvided) ;
      CS_LinkList selectedLinks = container.selectLinks(selectedConnectors) ;
      for (int j = 0 ; j < selectedLinks.size() ; j++) {
        connectorInstances.addValue(selectedLinks.getValue(j)) ;
      }
    }

    // For each matching link, retrieves the end value opposite
    // to interactionPoint.
    // If values opposite to this interaction point are references
    // (which means that it is possible to dispatch the operation),
    // they are added in the list of potential targets.
    ReferenceList targetsForDispatchingIn = new ReferenceList() ;
    ReferenceList targetsForDispatchingOut = new ReferenceList() ;
    ReferenceList targets = new ReferenceList() ;
    Integer i = 1 ;
    while (i <= connectorInstances.size()) {
      CS_Link connectorInstance = connectorInstances.getValue(i-1) ;
      Association association = connectorInstance.type ;
      Property oppositeEnd = association.memberEnd.getValue(0);
      if (oppositeEnd == interactionPoint.definingPort) {
        oppositeEnd = association.memberEnd.getValue(1);
      }
      Value value = connectorInstance.getFeatureValue(oppositeEnd).values.getValue(0) ;
      if (value instanceof Reference) {
        if (connectorInstance.definingConnector.kind == ConnectorKind.assembly) {
          targetsForDispatchingIn.addValue((Reference)value) ;
        }
        else {
          targetsForDispatchingOut.addValue((Reference)value) ;
        }
      }
      targets.add((Reference)value) ;
      i = i + 1;
    }

    CS_RequestPropagationStrategy strategy =
(CS_RequestPropagationStrategy)this.locus.factory.getStrategy("propagationRequest") ;
    ReferenceList selectedTargets = strategy.select(targets, new SendSignalActionActivation()) ;

    for (int j = 0 ; j < selectedTargets.size() ; j++) {
      Reference target = selectedTargets.getValue(j) ;
      for (int k = 0 ; k < targetsForDispatchingIn.size() && execution == null ; k++) {
        Reference cddTarget = targetsForDispatchingIn.getValue(k) ;
        if (cddTarget == target) {
          execution = target.dispatch(operation) ;
        }
      }
      for (int k = 0 ; k < targetsForDispatchingOut.size() && execution == null ; k++) {
        // The target must be an interaction point
        CS_InteractionPoint cddTarget = (CS_InteractionPoint)targetsForDispatchingOut.getValue(k) ;
        if (cddTarget == target) {
          CS_Reference owner = cddTarget.owner ;
          execution = owner.dispatchOut(operation, cddTarget) ;
        }
```

```
      }
    }
    return execution ;


public getFeatureValue(feature:StructuralFeature) : FeatureValue

    // In the case where the feature belongs to an Interface,
    // fUML semantics is extended in the sense that reading is
    // delegated to a CS_StructuralFeatureOfInterfaceAccessStrategy
    if (feature.namespace instanceof Interface) {
      CS_StructuralFeatureOfInterfaceAccessStrategy readStrategy =

(CS_StructuralFeatureOfInterfaceAccessStrategy)this.locus.factory.getStrategy("structuralFeature") ;
      return readStrategy.read(this, feature) ;
    }
    else {
      return super.getFeatureValue(feature);
    }


public setFeatureValue(feature:StructuralFeature, values:Value[*], position:Integer)

    // In the case where the feature belongs to an Interface,
    // fUML semantics is extended in the sense that writing is
    // delegated to a CS_StructuralFeatureOfInterfaceAccessStrategy
    if (feature.namespace instanceof Interface) {
      CS_StructuralFeatureOfInterfaceAccessStrategy writeStrategy =

(CS_StructuralFeatureOfInterfaceAccessStrategy)this.locus.factory.getStrategy("structuralFeature") ;
      writeStrategy.write(this, feature, values, position) ;
    }
    else {
      super.setFeatureValue(feature, values, position);
    }


public contains(object:Object) : Boolean

    // Determines if the object given as a parameter is directly
    // or indirectly contained by this CS_Object
    boolean objectIsContained = this.directlyContains(object) ;
    // if object is not directly contained, restart the research
    // recursively on the objects owned by this CS_Object
    for (int i = 0 ; i < this.featureValues.size() && !objectIsContained ; i++) {
      FeatureValue featureValue = this.featureValues.getValue(i) ;
      ValueList values = featureValue.values ;
      for (int j = 0 ; j < values.size() && !objectIsContained ; j++) {
        Value value = values.getValue(j) ;
        if (value instanceof CS_Object) {
          objectIsContained = ((CS_Object)value).contains(object) ;
        }
        else if (value instanceof CS_Reference) {
          CS_Object referent = ((CS_Reference)value).compositeReferent ;
          objectIsContained = referent.contains(object) ;
        }
      }
    }
    return objectIsContained;


public directlyContains(object:Object) : Boolean

    // Determines if the object given as a parameter is directly
    // contained by this CS_Object
    boolean objectIsContained = false ;
    for (int i = 0 ; i < this.featureValues.size() && !objectIsContained ; i++) {
      FeatureValue featureValue = this.featureValues.getValue(i) ;
      ValueList values = featureValue.values ;
      for (int j = 0 ; j < values.size() && !objectIsContained ; j++) {
        Value value = values.getValue(j) ;
        if (value == object) {
          objectIsContained = true ;
        }
        else if (value instanceof CS_Reference) {
          objectIsContained = (((CS_Reference)value).referent == object) ;
        }
```

```
      }
    }
    return objectIsContained;

public getDirectContainers() : CS_Object[*]

    // Retrieves all the extensional values at this locus which are direct
    // containers for this CS_Object
    // An extensional value is a direct container for an object if:
    // - it is a CS_Object
    // - it directly contains this object (i.e. CS_Object.directlyContains(Object)==true)
    CS_ObjectList containers = new CS_ObjectList() ;
    for (int i = 0 ; i < this.locus.extensionalValues.size() ; i++) {
      ExtensionalValue extensionalValue = this.locus.extensionalValues.getValue(i) ;
      if (extensionalValue != this && extensionalValue instanceof CS_Object) {
        CS_Object cddContainer = (CS_Object)extensionalValue ;
        if (cddContainer.directlyContains(this)) {
          containers.add(cddContainer) ;
        }
      }
    }
    return containers ;


public isOperationProvided(actualConnectableElement:Property, operation:Operation) : Boolean

    // Determines if the given connectable element provides the operation
    // If the connectable element is a port, it provides the operation if this operation
    // is a member of one of its provided interfaces
    // If the connectable eleemnt is NOT a port, it provides this operation if this operation is
    // an operation of this type, or this type provides a realization for this operation (in the case
    // where the namespace of this Operation is an interface)
    boolean isProvided = false ;
    if (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 interfaceIndex = 1 ;
        // Iterates on provided interfaces of the port
        InterfaceList providedInterfaces = ((Port)actualConnectableElement).provided() ;
        while (interfaceIndex <= providedInterfaces.size() && !isProvided) {
          Interface interface_ = ((Port)actualConnectableElement)
              .provided()
              .getValue(interfaceIndex-1) ;
          // Iterates on members of the current Interface
          Integer memberIndex = 1 ;
          while (memberIndex <= interface_.member.size() && !isProvided) {
            NamedElement cddOperation = interface_.member.getValue(memberIndex-1) ;
            if (cddOperation instanceof Operation) {
              isProvided = operation == cddOperation ;
            }
            memberIndex = memberIndex + 1 ;
          }
          interfaceIndex = interfaceIndex + 1 ;
        }
      }
    }
    else {
      // The connectable element is not a Port.
      // We have to look if the Classifier typing this connectable element
      // directly or indirectly provides this operation
      if (actualConnectableElement.typedElement.type instanceof Class_) {
        Integer memberIndex = 1 ;
        NamedElementList members = ((Class_)actualConnectableElement.typedElement.type).member ;
        while (memberIndex <= members.size() && !isProvided) {
          NamedElement cddOperation = members.getValue(memberIndex-1) ;
          if (cddOperation instanceof Operation) {
            CS_DispatchOperationOfInterfaceStrategy strategy = new
CS_DispatchOperationOfInterfaceStrategy() ;
            isProvided = strategy.operationsMatch((Operation)cddOperation, operation) ;
          }
          memberIndex = memberIndex + 1 ;
        }
      }
    }
    return isProvided ;
```

```
public isOperationRequired(actualConnectableElement:Property, operation:Operation) : Boolean

    // Determines if the given connectable element requires the operation
    // If the connectable element is a port, it requires the operation if this operation
    // is a member of one of its provided interfaces
    // If the connectable element is NOT a port, it cannot require an operation
    boolean matches = false ;
    if (actualConnectableElement instanceof Port) {
      Integer interfaceIndex = 1 ;
      // Iterates on provided interfaces of the port
      InterfaceList requiredInterfaces = ((Port)actualConnectableElement).required() ;
      while (interfaceIndex <= requiredInterfaces.size() && !matches) {
        Interface interface_ = ((Port)actualConnectableElement)
            .provided()
            .getValue(interfaceIndex-1) ;
        // Iterates on members of the current Interface
        Integer memberIndex = 1 ;
        while (memberIndex <= interface_.member.size() && !matches) {
          NamedElement cddOperation = interface_.member.getValue(memberIndex-1) ;
          if (cddOperation instanceof Operation) {
            matches = operation == cddOperation ;
          }
          memberIndex = memberIndex + 1 ;
        }
        interfaceIndex = interfaceIndex + 1 ;
      }
    }
    return matches ;


public isReceptionProvided(actualConnectableElement:Property, signal:Signal) : Boolean

    // Determines if the given connectable element provides a reception for signal
    // If the connectable element is a port, it provides a reception if one of its
    // provided interfaces owns a reception for that signal
    // If the connectable eleemnt is NOT a port, it provides a reception if its
    // type has a reception for that signal
    TypeList types = new TypeList() ;
    if (actualConnectableElement instanceof Port) {
      // types are interfaces provided by this port
      Integer interfaceIndex = 1 ;
      InterfaceList providedInterfaces = ((Port)actualConnectableElement).provided() ;
      while (interfaceIndex <= providedInterfaces.size()) {
        Interface interface_ = providedInterfaces.getValue(interfaceIndex-1) ;
        types.addValue(interface_) ;
        interfaceIndex = interfaceIndex + 1 ;
      }
    }
    else {
      types.addValue(actualConnectableElement.typedElement.type) ;
    }
    boolean matches = false ;
    // Iterates on types
    Integer typeIndex = 1 ;
    while (typeIndex <= types.size() && !matches) {
      Type type_ = types.getValue(typeIndex-1) ;
      // Iterates on members of the current type
      Integer memberIndex = 1 ;
      NamedElementList members = type_.member ;
      while (memberIndex <= members.size() && !matches) {
        NamedElement cddReception = members.getValue(memberIndex) ;
        if (cddReception instanceof Reception) {
          if (((Reception)cddReception).signal == signal) {
            matches = true ;
          }
        }
        memberIndex = memberIndex + 1 ;
      }
      typeIndex = typeIndex + 1 ;
    }
    return matches ;


public isReceptionRequired(actualConnectableElement:Property, signal:Signal) : Boolean

    // Determines if the given connectable element requires a reception for signal
    // If the connectable element is a port, it requires a reception if one of its
```

```
    // required interfaces owns a reception for that signal
    // If the connectable eleemnt is NOT a port, it cannot require a reception
    boolean matches = false ;
    TypeList types = new TypeList() ;
    if (actualConnectableElement instanceof Port) {
      // types are interfaces provided by this port
      Integer interfaceIndex = 1 ;
      InterfaceList requiredInterfaces = ((Port)actualConnectableElement).required() ;
      while (interfaceIndex <= requiredInterfaces.size() && !matches) {
        Interface interface_ = requiredInterfaces.getValue(interfaceIndex-1) ;
        types.addValue(interface_) ;
        // Iterates on members of the current type
        Integer memberIndex = 1 ;
        NamedElementList members = interface_.member ;
        while (memberIndex <= members.size() && !matches) {
          NamedElement cddReception = members.getValue(memberIndex) ;
          if (cddReception instanceof Reception) {
            if (((Reception)cddReception).signal == signal) {
              matches = true ;
            }
          }
          memberIndex = memberIndex + 1 ;
        }
        interfaceIndex = interfaceIndex + 1 ;
      }
    }
    return matches ;


public selectLinks(connectors:Connector[*]) : CS_Link[*]

    // Select links owned by the context object for which the
    // definingConnector is included in the list of connectors
    // given as a parameter.
    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 ;
    }
    return connectorInstances ;
```

1.1.1.1.44 CS_Link

CS_Link 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

1.1.5 InvocationActions

1.1.5.1 Overview

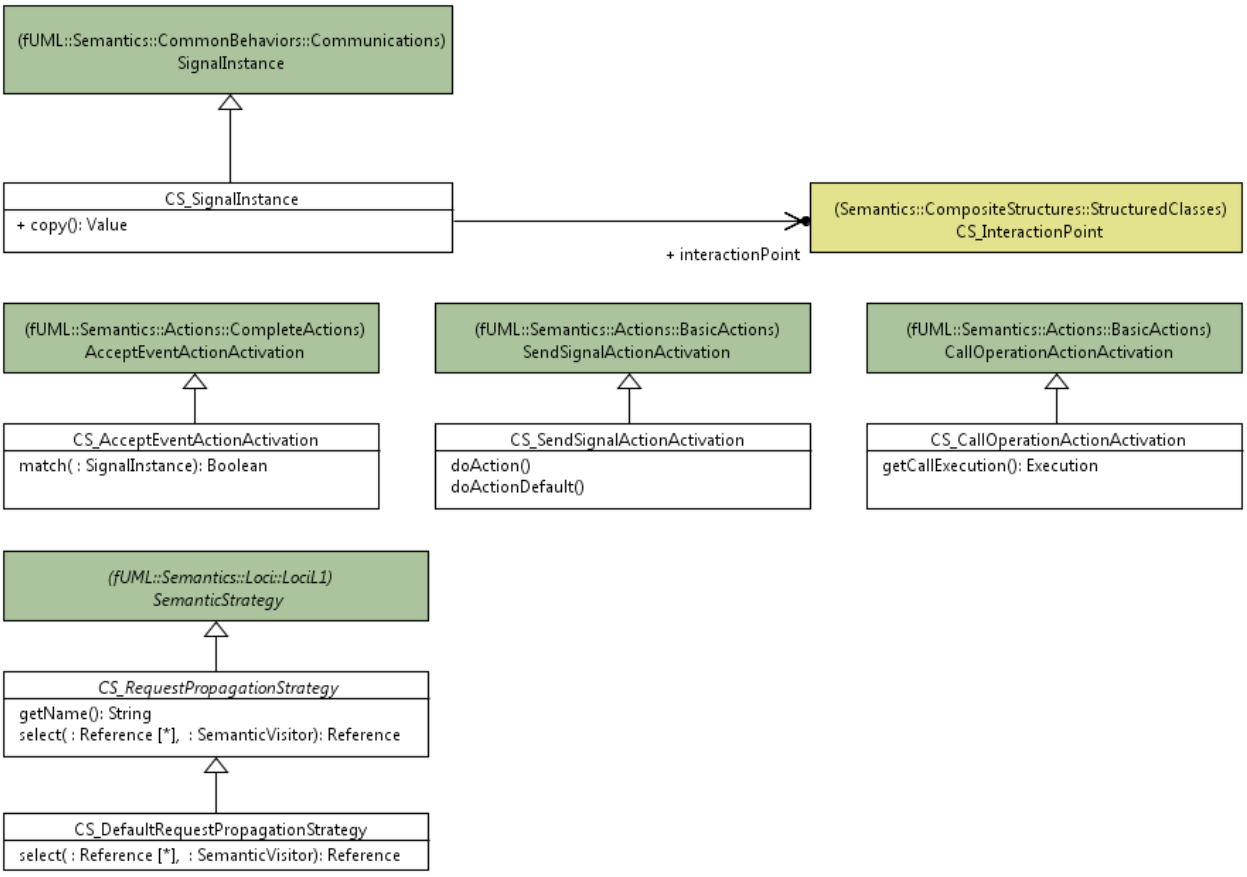

Figure 22: InvocationActions diagram

TODO.

1.1.1.5 Class descriptions

1.1.1.1.45 CS_SendSignalActionActivation

Extends semantics of fUML SendSignalActionActivation::doAction() to capture propagation of signals across ports. (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

```
public 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:
 // - if the target is to be the same as or a container of (directly or indirectly)
  // the object executing the Action, the Signal shall be related to a Reception belonging
  // to a required onPort, and sendOut is called on the target reference
  // so that the signal will be sent to the environment
  // - if the target is NOT to be the same as or a container of (directly or indirectly)
  // the object executing the Action, the Signal shall be related to a Reception belonging
  // to a provided Interface of onPort, and operation sendIn is called so that the signal
  // will be sent to the internals of the target object (from where the execution
  // will be taken).

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

  if (action.onPort == null) {
    // Behaves like in fUML
    this.doActionDefault() ;
  }
  else {
    Value target = this.takeTokens(action.target).getValue(0) ;

    if (target instanceof CS_Reference) {
      // Constructs the signal instance
     Signal signal = action.signal;
     CS_SignalInstance signalInstance = new CS_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);
     }

     // Tries to determine if the signal has to be
     // sent to the environment or to the internals of
     // target, through onPort
      CS_Reference targetReference = (CS_Reference)target ;
      Port onPort = action.onPort ;
      Object_ executionContext = this.group.activityExecution.context ;
      if (executionContext == targetReference.referent
          || targetReference.compositeReferent.contains(executionContext)) {
        targetReference.sendOut(signalInstance, action.onPort);
      }
      else {
        targetReference.sendIn(signalInstance, action.onPort);
      }
      targetReference.sendOut(signalInstance, onPort) ;
    }
  }
```

```
public doActionDefault()

  // 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
  // and send it to the referent object.
  // This operation captures same semantics as fUML
  // SendSignalActionActivation.doAction() except that it constructs
  // a CS_SignalInstance instead of a SignalInstance

  SendSignalAction action = (SendSignalAction) (this.node);
  Value target = this.takeTokens(action.target).getValue(0);

  if (target instanceof Reference) {
    Signal signal = action.signal;

    CS_SignalInstance signalInstance = new CS_SignalInstance();
    signalInstance.type = signal;

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

    ((Reference) target).send(signalInstance);
  }
```

1.1.1.1.46 CS_CallOperationActionActivation

Extends fUML CallOperationActionActivation::getCallExecution() to capture dispatching semantics of requests across ports. (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

```
public 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:
    // - if the target is to be the same as or a container of (directly or indirectly)
    // the object executing the Action, the called Operation shall belong to a required
    // Interface of onPort, and dispatchOut is called on the target reference
    // so that the operation call will be dispatched to the environment
    // (from where the execution will be taken)
    // - if the target is NOT to be the same as or a container of (directly or indirectly)
    // the object executing the Action, the called Operation shall belong to a provided
    // Interface of onPort, and operation dispatchIn is called so that the operation call
```

```
    // will be dispatch to the internals of the target object (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) {
        // Tries to determine if the operation call has to be
        // dispatched to the environment or to the internals of
        // target, through onPort
        CS_Reference targetReference = (CS_Reference)target ;
        Object_ executionContext = this.group.activityExecution.context ;
        if (executionContext == targetReference.referent
            || targetReference.compositeReferent.contains(executionContext)) {
          execution = targetReference.dispatchOut(action.operation, action.onPort);
        }
        else {
          execution = targetReference.dispatchIn(action.operation, action.onPort);
        }
      }
    }
    return execution;
```

1.1.1.1.47 CS_RequestPropagationStrategy

This semantic strategy is introduced to enable semantic variants related to propagation of requests through connectors, in the case where multiple propagation paths are possible. Concrete strategy classes shall provide a behavior for abstract operation select.

Generalizations

- SemanticStrategy (from fUML::Semantics::Loci::LociL1)

Attributes

- None

Associations

- None

Operations

```
public getName() : String

    // a CS_RequestPropagationStrategy are always named "requestPropagation"
    return "requestPropagation";


public abstract select(potentialTargets:Reference[*], context:SemanticVisitor) : Reference[*]
```

1.1.1.1.48 CS_DefaultRequestPropagationStrategy

This class proposes a basic semantic variant for the semantic variation point captured by CS_RequestPropagationSrategy. This semantic variants consists in broadcasting the request to all possible targets in the case where the request concerns a signal sending. In the case where the request concerns an operation call, only the first target is kept.

Generalizations

- CS_RequestPropagationStrategy (from CompositeStructuresSyntaxAndSemantics::Semantics::CompositeStructures::InvocationActions)

Attributes

- None

Associations

- None

Operations

```
public select(potentialTargets:Reference[*], context:SemanticVisitor) : Reference[*]

    // returns all potential targets in the case where the context is a SendSignalActionActivation
    // returns the first potential target in the case where the context is anything else
    ReferenceList selectedTargets = new ReferenceList() ;
    if (context instanceof SendSignalActionActivation) {
      for (int i = 0 ; i < potentialTargets.size() ; i++) {
        selectedTargets.addValue(potentialTargets.getValue(i)) ;
      }
    }
    else {
      if (potentialTargets.size() >= 1) {
        selectedTargets.addValue(potentialTargets.get(0)) ;
      }
    }
    return selectedTargets;
```

1.1.1.1.49 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

```
public match(signalInstance:SignalInstance) : Boolean

  // Return true if the given signal instance matches a trigger of the accept
  // event 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;
```

1.1.1.1.50 CS_SignalInstance

CS_SignalInstance extends fUML SignalInstance with the ability to reference the specific interaction point 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

```
public 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;
```

1.4 CommonBehaviors

1.4.1 Overview

TODO.

1.1.6 Communications

1.1.6.1 Overview

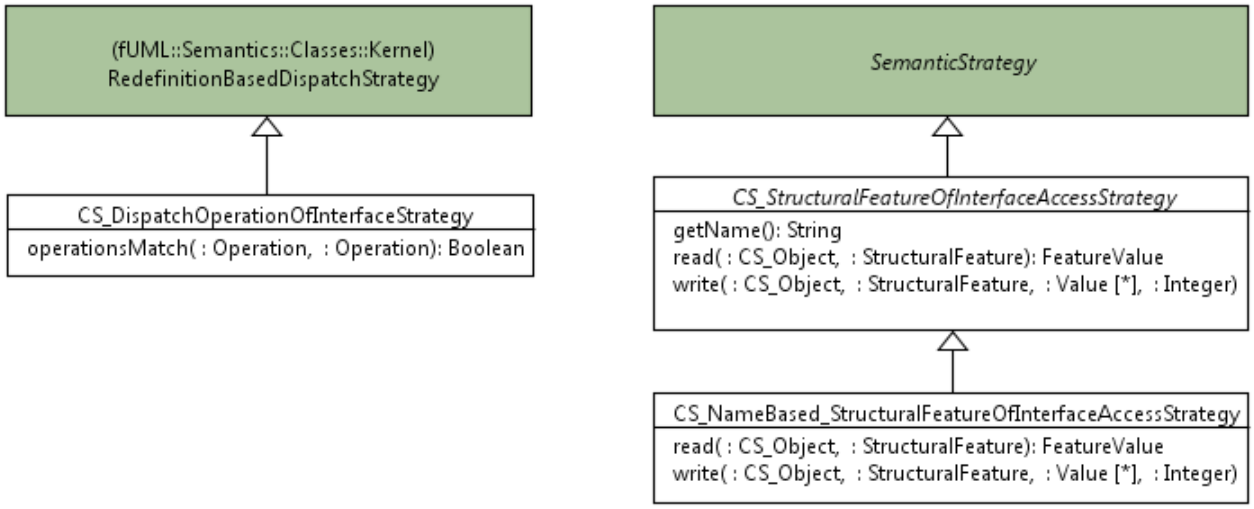

Figure 23: Communications diagram

TODO.

1.1.1.6 Class descriptions

1.1.1.1.51 CS_StructuralFeatureOfInterfaceAccessStrategy

This abstract strategy class enables to deal with realization of structural features by a behaviored classifier, in the case where these structural features belong to an interface realized by this classifier. Concrete realization of this abstract class shall specify behaviors for abstract operations read and write.

Generalizations

- SemanticStrategy (from fUML::Semantics::Loci::LociL1)

Attributes

- None

Associations

- None

Operations

```
public getName() : String

    // StructuralFeatureAccessStrategy are always named "structuralFeature"
    return "structuralFeature";

public abstract read(cs_Object:CS_Object, feature:StructuralFeature) : FeatureValue

public abstract write(cs_Object:CS_Object, feature:StructuralFeature, values:Value[*],
position:Integer)
```

1.1.1.1.52 CS_NameBased_StructuralFeatureOfInterfaceAccessStrategy

This class provides a realization of semantic strategy CS_StructuralFeatureOfInterfaceAccessStrategy. This realization makes the hypothesis that, for a given behaviored classifier to actually realize an interface, it shall have structural features which exactly match structural features of the realized interface. Realizing features shall have same name, type and multiplicity as structural features of the interface.

Generalizations

- CS_StructuralFeatureOfInterfaceAccessStrategy (from CompositeStructuresSyntaxAndSemantics::Semantics::CommonBehaviors::Communications)

Attributes

- None

Associations

- None

Operations

```
public read(cs_Object:CS_Object, feature:StructuralFeature) : FeatureValue

    // returns the a copy of the first feature value of cs_Object where the name of the corresponding
feature
    // matches the name of the feature given as a parameter
    // Otherwise, returns an empty feature value
    FeatureValueList featureValues = cs_Object.featureValues ;
    FeatureValue matchingFeatureValue = null ;
    for (int i = 0 ; i < featureValues.size() && matchingFeatureValue == null ; i++) {
      FeatureValue featureValue = featureValues.getValue(i) ;
      if (featureValue.feature.name.equals(feature.name)) {
        matchingFeatureValue = featureValue ;
      }
    }
    if (matchingFeatureValue != null) {
      matchingFeatureValue = matchingFeatureValue.copy() ;
      matchingFeatureValue.feature = feature ;
    }
    else {
      matchingFeatureValue = new FeatureValue() ;
      matchingFeatureValue.feature = feature ;
      matchingFeatureValue.values = new ValueList() ;
      matchingFeatureValue.position = 0 ;
    }

    return matchingFeatureValue ;
```

```
public write(cs_Object:CS_Object, feature:StructuralFeature, values:Value[*], position:Integer)

  // Retrieves the first feature value of cs_Object where the name of the corresponding feature
  // matches the name of the feature given as a parameter
  // Then updates the values for this feature value
  FeatureValueList featureValues = cs_Object.featureValues ;
  FeatureValue matchingFeatureValue = null ;
  for (int i = 0 ; i < featureValues.size() && matchingFeatureValue == null ; i++) {
    FeatureValue featureValue = featureValues.getValue(i) ;
    if (featureValue.feature.name.equals(feature.name)) {
      matchingFeatureValue = featureValue ;
    }
  }
  if (matchingFeatureValue != null) {
    cs_Object.setFeatureValue(matchingFeatureValue.feature, values, position) ;
  }
```

1.1.1.1.53 CS_DispatchOperationOfInterfaceStrategy

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 of RedefinitionBasedDispatchStrategy. Could be defined as direct specialization of DispatchStrategy)

Generalizations

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

Attributes

- None

Associations

- None

Operations

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

  // Override operationsMatch, in the case where baseOperation belongs
  // to an Interface.
  // In this case, ownedOperation matches baseOperation if it has the same name and signature
  // Otherwise, behaves like fUML RedefinitionBasedDispatchStrategy
  boolean matches = true ;
  if (baseOperation.namespace instanceof Interface) {
    matches = (baseOperation.name == ownedOperation.name) ;
    matches = matches && (baseOperation.ownedParameter.size() ==
ownedOperation.ownedParameter.size()) ;
    ParameterList ownedOperationParameters = ownedOperation.ownedParameter ;
    ParameterList baseOperationParameters = baseOperation.ownedParameter ;
    for (int i = 0 ; matches == true && i < ownedOperationParameters.size() ; i++) {
      Parameter ownedParameter = ownedOperationParameters.getValue(i) ;
      Parameter baseParameter = baseOperationParameters.getValue(i) ;
      matches = (ownedParameter.type == baseParameter.type) ;
      matches = matches && (ownedParameter.multiplicityElement.lower ==
ownedParameter.multiplicityElement.lower) ;
      matches = matches && (ownedParameter.multiplicityElement.upper ==
ownedParameter.multiplicityElement.upper) ;
      matches = matches && (ownedParameter.direction == ownedParameter.direction) ;
    }
  }
  else {
    matches = super.operationsMatch(ownedOperation, baseOperation) ;
  }
```

```
return matches ;
```