Date: 12 November 2012

Precise Semantics of UML Composite Structures

Response to Request for Proposal XXXX

Initial submission

OMG Document Number:

Associated Machine Readable Files:

Submitted by:

Supported by:

Table of Contents

0 Submission Introduction

0.1 Preface

[TODO]

0.2 Resolution of Requirements

0.2.1 Mandatory Requirements

6.5.1 Precise Semantics	
6.5.1.a. Proposals shall provide precise semantics for metaclasses underlying UML composite structures, which includes at least Connector, ConnectorEnd, ConnectableElement, EncapsulatedClassifier, InvocationAction, Port, StructuredClassifier, Trigger and Interface. The semantic description shall establish explicit relationships with fUML, for example by specifying a precise formal model transformation from the metaclasses listed above to metaclasses which are part of the fUML subset, and/or by extending the fUML execution model, for example with appropriate visitor classes. Whatever the way the execution semantics is actually specified, proposals shall be readable as if they are additions to fUML semantics, rather than separate specifications.	[TODO]
6.5.1.b. Proposals shall extend the base semantics of fUML with specific axioms for UML composite structures only if necessary. These new axioms shall have explicit relationships with existing axioms of fUML base semantics. These axioms shall be expressed in Common Logic Interchange Format (as was done for fUML), and they shall be consistent with fUML axioms.	[TODO]
6.5.2 Semantic Variability	
6.5.2.a. Proposals shall precisely identify allowed semantic variabilities. These semantic variabilities shall be in the scope of semantic variabilities allowed by UML composite structures (which means that only a subset of allowed UML semantic variabilities could be reflected in the semantic description, as was done for fUML).	[TODO]
6.5.2.b. Proposals shall define rules for defining semantic variants, where a semantic variant is an internally consistent set of values for the different semantic variabilities allowed from requirement 6.5.2.a.	[TODO]
6.5.3 Alignment	

6.5.3.a. Proposals shall comply with the current version of the UML 2 metamodel and notation (Consideration should also be given to upcoming developments, such as the anticipated UML 2.5 revision).	[TODO]
6.5.3.b. Proposals shall comply with the current version of the fUML specification.	[TODO]
6.5.4 Test suite	
6.5.4.a Proposals shall provide a suite of test cases that can demonstrate conformance to this specification.	[TODO]

0.2.2 Optional Requirements

6.6.1 Proposals may associate execution semantics with metaclasses Collaboration, CollaborationUse, Component and ComponentRealization.	[TODO]
6.6.2 Proposals may use the Action language for Foundational UML [Alf] as a concrete syntax for specifying the execution semantics of composite structures.	[TODO]
6.6.3 Proposals may provide a non-normative Annex containing a specialization of the composite structure semantics, capturing semantics introduced by both the Generic Component Model (GCM) and the High Level Application Modeling (HLAM) subprofiles of MARTE.	[TODO]
6.6.4 Proposals may provide a non-normative Annex containing a specialization of the composite structure semantics, capturing semantics introduced by the Blocks, Ports and Flows, and Constraint Blocks packages of SysML.	[TODO]
6.6.5 Proposals may define extensions of Alf to support the concepts defined for UML composite structures.	[TODO]

0.3 Responses to Issues to be Discussed

6.7.1 Proposals shall describe the semantics of execution in situations where required information is missing and therefore might need to be dynamically injected by some manual or automated means.
[TODO]

6.7.2 Proposals shall discuss how an executable UML model begins its execution.
[TODO]

6.7.3 Proposals shall discuss a proof of concept implementation that can successfully execute tests from the conformance test suite.
[TODO]

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.

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.
[NOTE: I guess we should write something in relation with the test suite, as well as with existing fUML conformance criteria]

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.
[TODO: Put some statements about the invalid UML 2.4.1 constraints related to the type of the target input pin of a call operation action and the ownership of the called operation => Explain that this is addressed in UML 2.5]

6.2 Acknowledgments

Thank you

7 Abstract Syntax

7.1 Overview

The Abstract Syntax clause describes how the fUML syntax model is extended to include UML Composite Structures modeling. This extension refers to all UML metaclasses and relationships which enable a fUML Class to be both structured and encapsulated. Structured means that a class can have an internal structure comprising a network of parts linked through connectors. Encapsulated means that a class can have an external structure comprising a set of ports, exposing the features that a class provides to or requires from its environment. The Semantics clause 8 defines precise semantics of elements introduced in following sub-sections.

The fUML Abstract Syntax model mimics the package architecture of the unmerged UML 2.4.1 metamodel, keeping only packages (and elements defined in these packages) that are relevant to the foundational subset of UML. In order to be consistent with fUML syntax, the Composite Structures syntax model described in this specification also mimics the package architecture of the unmerged UML 2.4.1 metamodel. UML packages relevant to Composite structures modeling are imported as a copy by the proposed syntax model. Elements in copied packages which are not relevant for this specification are removed. When a relationship contained in an imported package (including package merge relationships) targets a UML metaclass or package which is already part of the fUML subset, this relationship is redirected towards the fUML equivalent metaclass.

As compared to the fUML Abstract Syntax model, the proposed model does not carry out package merges, avoiding duplication of the content of fUML syntax. However, package merge relationships included in this syntax model are specified in a way that, when merges are carried out, a flat model corresponding to the fUML subset extended with UML composite structures modeling is produced. In addition, the result of the merge is also that any former references from elements of the fUML Semantics model to elements of the fUML Abstract Syntax model now target elements of the fUML subset extended with UML Composite Structures modeling *[Note for the initial submission: This is the intent, but the merging process does not produces a valid flat model]*.

The following sub-clauses describe the packages copied from the UML 2.4.1 unmerged metamodel. Each sub-clause provides rationale for copying the corresponding package, as well as an overview of elements contained in these packages and how they relate to fUML syntax elements. Each sub-clause also highlights additional constraints introduced by this specification regarding Composite Structures modeling.

TODO: Diagram showing the architecture of the Abstract Syntax model

7.2 Classes

7.2.1 Overview

The Classes package is partially imported as a copy by the Composite Structures syntax model. This copy is motivated by the fact that it contains sub-packages Dependencies and Interfaces, which are both required for Composite Structures modeling. The copy of the Classes package is partial in the sense that only these two sub-packages are copied. Note that the Classes package is already part of the fUML Abstract Syntax model. However, Dependencies and Interfaces were explicitly removed from the foundational subset. Following sub-clauses provide details about the content of these two sub-packages.

7.2.2 Dependencies

7.2.2.1 Overview

The Dependencies package is copied for two main reasons. The first one is that it introduces metaclass Usage, which is potentially used to specify required interfaces of a port. The second one is that it introduces metaclass Realization, which is a general metaclass for InterfaceRealization (introduced in the Interfaces package). Note that Metaclasses

DirectedRelationship and Substitution, as well as association A_mapping_abstraction are removed, since they are not relevant to Composite Structures modeling.

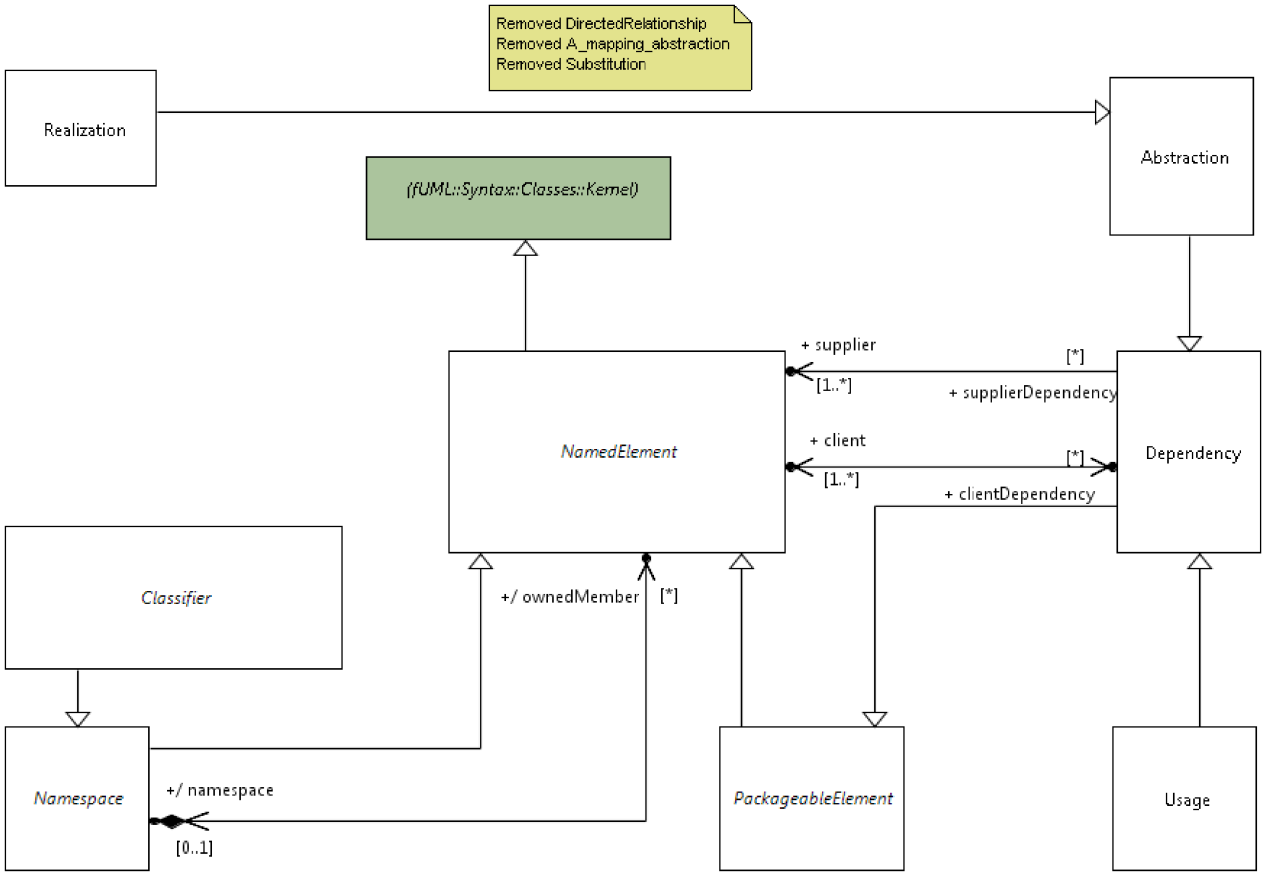

Figure 1: Dependencies diagram

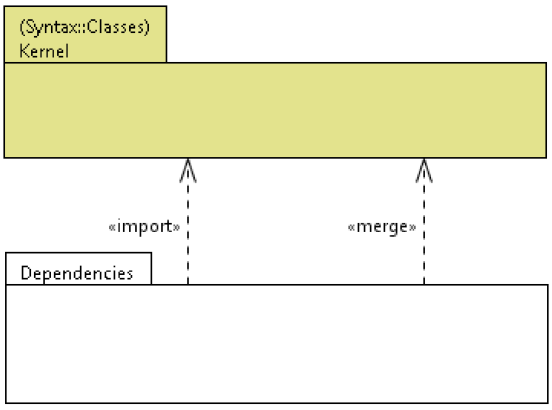

Figure 2: Dependencies package relationships diagram

7.2.2.2 Class descriptions

7.2.2.2.1 Abstraction

Abstraction is introduced to enable modeling of InterfaceRealization. This is an indirect general metaclass for InterfaceRealization.

Generalizations

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

Attributes

- None

Associations

- None

7.2.2.2.2 Classifier

Classes::Dependencies::Classifier does not extend in any way fUML Classifier. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- None

7.2.2.2.3 Dependency

Dependency is introduced to enable modeling of both Usage and InterfaceRealization relationships. This is a general metaclass for these two metaclasses.

Generalizations

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

Attributes

- None

Associations

- client : NamedElement [1..*]
- supplier : NamedElement [1..*]

7.2.2.2.4 NamedElement

Classes::Dependencies::NamedElement extends (merge increment) fUML NamedElement with the ability to indicate all the dependencies that reference this named element as a client (property clientDependency). [TODO: Check if this is useful, e.g., it is subseted somehow in the context of InterfaceRealization]

Generalizations

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

Attributes

- None

Associations

- clientDependency : Dependency [0..*]
- namespace : Namespace [0..1]

7.2.2.2.5 Namespace

Classes::Dependencies::Namespace does not extend in any way fUML Namespace. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- ownedMember : NamedElement [0..*]

7.2.2.2.6 PackageableElement

Classes::Dependencies::PackeageableElement does not extend in any way fUML PackageableElement. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- None

7.2.2.2.7 Realization

Realization is introduced to enable modeling of InterfaceRealization. This is a general metaclass for InterfaceRealization.

Generalizations

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

Attributes

- None

Associations

- None

7.2.2.2.8 Usage

Usage is introduced since it is potentially useful to specify required interfaces of a port.

Generalizations

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

Attributes

- None

Associations

- None

7.2.3 Interfaces

7.2.3.1 Overview

The Interfaces package is introduced to enable exposure of features that an encapsulated classifier provides to or requires from its environment, through interfaces associated with its ports. It also includes mechanisms which enable to specify that a given behaviored classifier realizes the features of an interface. At run time, this information is useful to determine if a given object (instance of a behaviored classifier) is a valid target for the invocation of a behavioral feature (where this feature belongs to an interface).

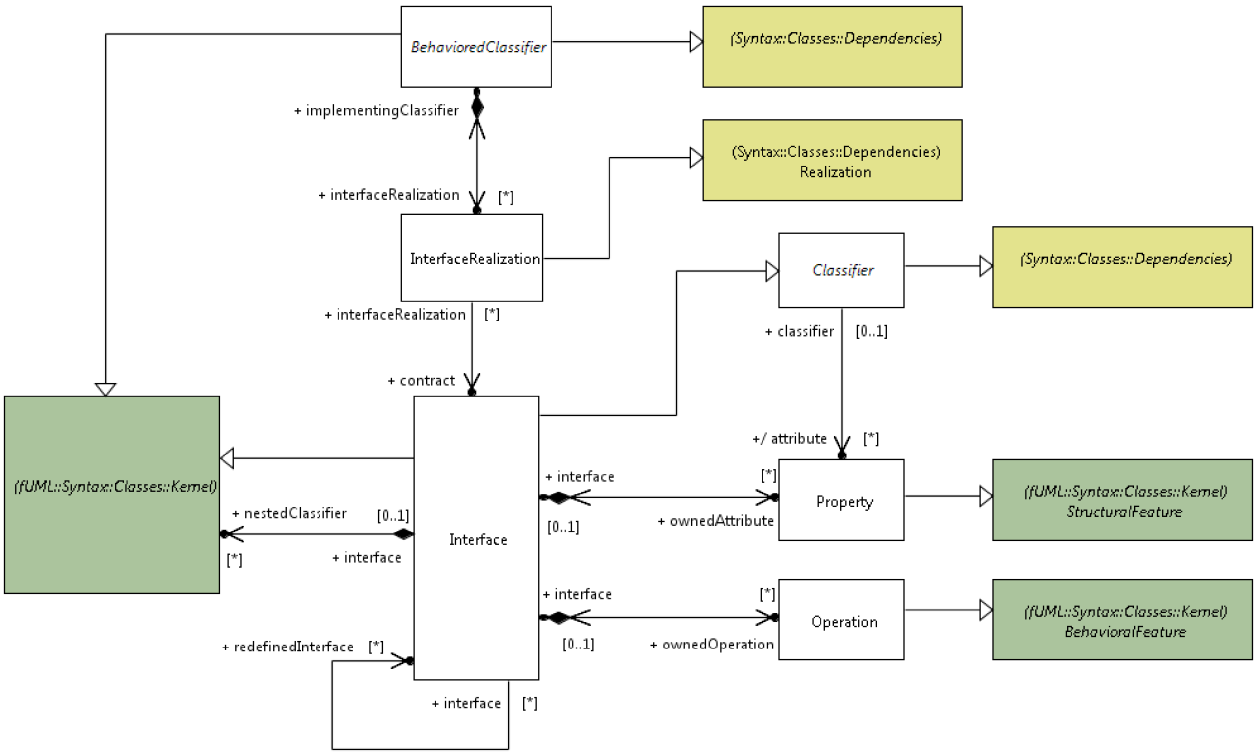

Figure 3: Interfaces diagram

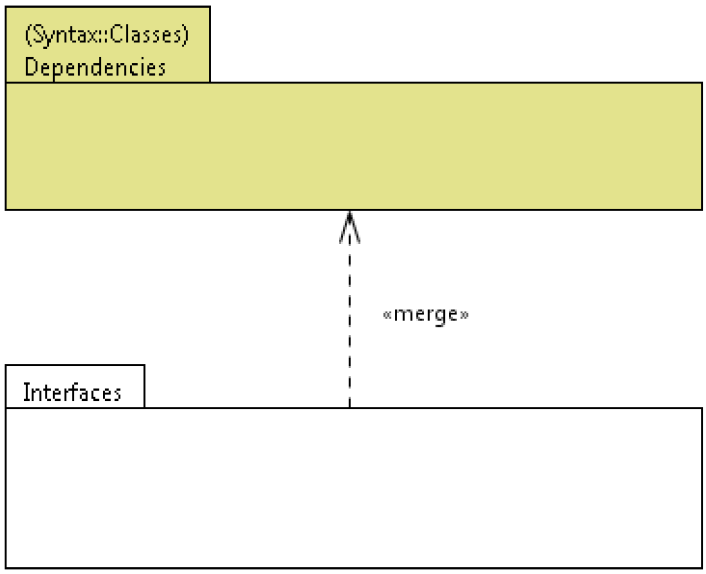

Figure 4: Interfaces package relationships diagram

7.2.3.2 Class descriptions

7.2.3.2.1 BehavioredClassifier

Classes::Interfaces::BehavioredClassifier extends (merge increment) fUML BehavioredClassifier with the ability to own a set of InterfaceRealization relationships.

Generalizations

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

Attributes

- None

Associations

- interfaceRealization : InterfaceRealization [0..*]

7.2.3.2.2 Classifier

Classes::Interfaces::Classifier does not extend in any way fUML Classifier. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- attribute : Property [0..*]

7.2.3.2.3 Interface

Interface is introduced to enable exposition of features provided or required on a Port. Through InterfaceRealization relationships, it also enable to specify features which are realized by a given BehavioredClassifier.

Generalizations

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

Attributes

- None

Associations

- nestedClassifier : Classifier [0..*]
- ownedAttribute : Property [0..*]
- ownedOperation : Operation [0..*]
- redefinedInterface : Interface [0..*]

7.2.3.2.4 InterfaceRealization

InterfaceRealization is introduced since it is potentially useful to specify provided interfaces of a Port. In addition, it can also be used to specify that features of the contract interface are realized by the implementingClassifier BehavioredClassifier.

Generalizations

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

Attributes

- None

Associations

- contract : Interface [1..1]
- implementingClassifier : BehavioredClassifier [1..1]

7.2.3.2.5 Operation

Classes::Interfaces::Operation extends (merge increment) fUML Operation with the capability to be owned by an Interface.

Generalizations

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

Attributes

- None

Associations

- interface : Interface [0..1]

7.2.3.2.6 Property

Classes::Interfaces::Property extends (merge increment) fUML Property with the capability to be owned by an Interface.

Generalizations

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

Attributes

- None

Associations

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

7.3 CommonBehaviors

7.3.1 Overview

The CommonBehaviors package is partially imported as a copy by the Composite Structures syntax model. This copy is motivated by the fact that it contains the sub-package Communications, which is required for Composite Structures modeling. More precisely, it allows receptions to be owned by interfaces, by the way enabling an encapsulated classifier to specify that it provides or requires receptions for signals at some of its ports. The copy of the CommonBehaviors package is partial in the sense that only this sub-package is copied. Note that the CommonBehaviors::Communications package is already part of the fUML Abstract Syntax model. However, the Interface metaclass was explicitly removed from the foundational subset. The following sub-clause provides details about the content of this sub-package.

7.3.2 Communications

7.3.2.1 Overview

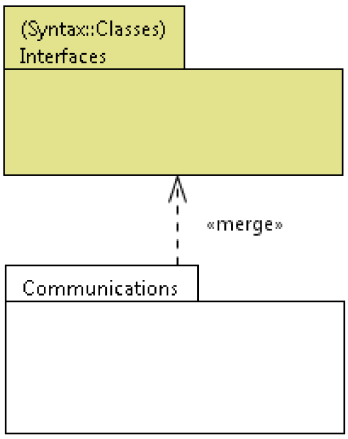

Figure 5: Communications package relationships diagram

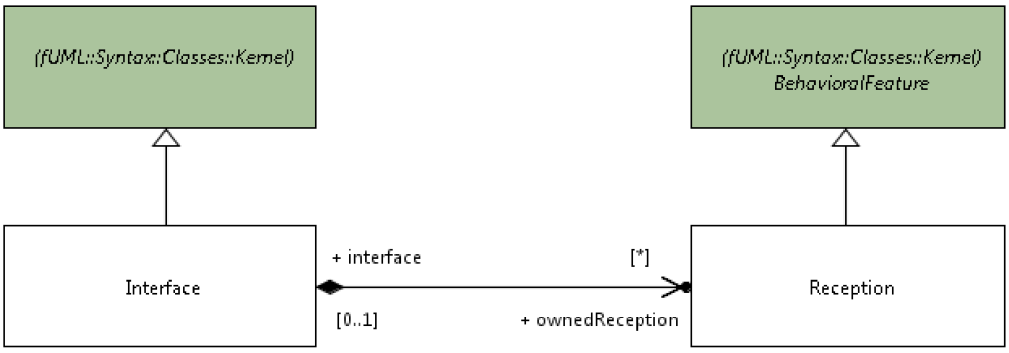

Figure 6: Communications diagram

7.3.2.2 Class descriptions

7.3.2.2.1 Interface

CommonBehaviors::Communications::Interface extends (merge increment) Classes::Interfaces::Interface with the ability to own receptions.

Generalizations

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

Attributes

- None

Associations

- ownedReception : Reception [0..*]

7.3.2.2.2 Reception

CommonBehaviors::Communications::Reception does not extend in any way fUML Reception. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- None

7.4 Components

7.4.1 Overview

The Components package is partially imported as a copy by the Composite Structures syntax model. This copy is motivated by the fact that it contains the sub-package BasicComponents, which is required for Composite Structures

modeling. More precisely, it extends (merge increment) CompositeStructures::StructuredClasses::Connector with the property kind, which enables to determine if a given connector is for delegation or assembly. Note that metaclass Component as well as package Components::PackagingComponents are excluded from this specification, since they do not introduce any concept impacting semantics of UML Composite Structures.

7.4.2 BasicComponents

7.4.2.1 Overview

Figure 7: BasicComponents diagram

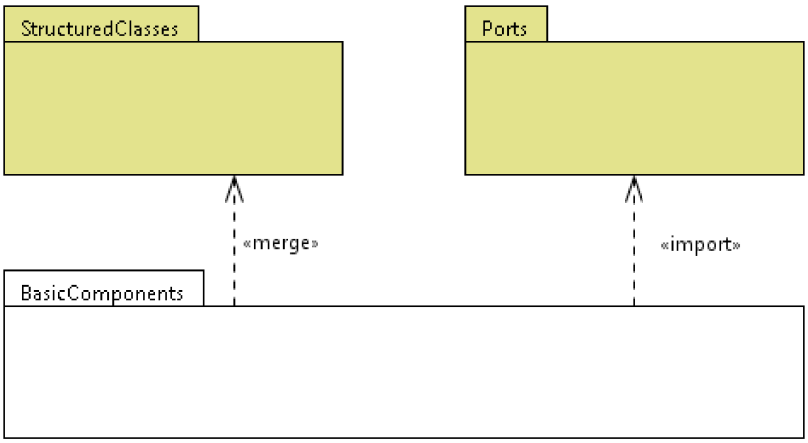

Figure 8: BasicComponents package relationships diagram

7.4.2.2 Class descriptions

7.4.2.2.1 Connector

Components::BasicComponents::Connector extends (merge increment) CompositeStructures::StructuredClasses::Connector with a property indicating the kind (delegation or assembly) of a connector.

Generalizations

- None

Attributes

- kind : ConnectorKind [1..1]

Associations

- None

7.4.2.2.2 ConnectorKind (Enumeration)

Generalizations

- None

Literals

- assembly
- delegation

7.5 CompositeStructures

7.5.1 Overview

The CompositeStructures package is totally imported as a copy by the Composite Structures syntax model. This copy is motivated by the fact that all its sub-packages introduce key modeling concepts underlying Composite Structures. This includes packages InternalStructures, InvocationActions, Ports and StructuredClasses. Following sub-clauses provide details about the content of these sub-packages.

7.5.2 InternalStructures

7.5.2.1 Overview

The InternalStructures package is introduced to enable modeling of the internal structure of a class as a network of parts, linked through connectors. The topology of parts and connectors places constraints on the runtime structure of a class instance.

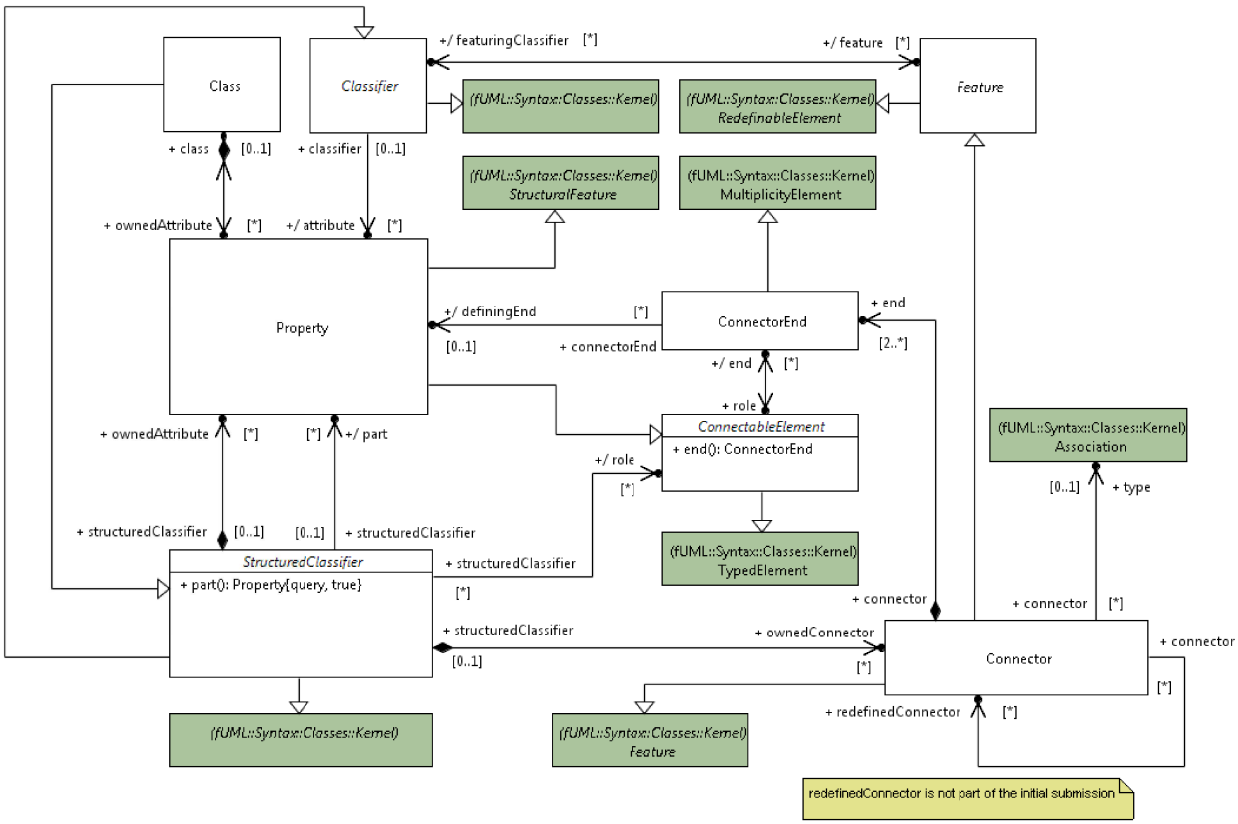

Figure 9: InternalStructures diagram

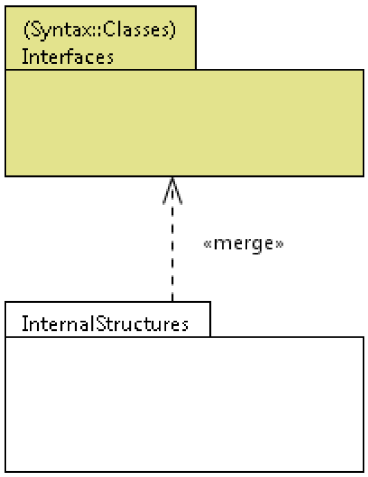

Figure 10: InternalStructures package relationship diagram

7.5.2.2 Class descriptions

7.5.2.2.1 Class

CompositeStructures::InternalStructures::Class extends (merge increment) fUML Class by being a StructuredClassifier.

Generalizations

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

Attributes

- None

Associations

- ownedAttribute : Property [0..*]

7.5.2.2.2 Classifier

CompositeStructures::InternalStructures::Classifier does not extend in any way fUML Classifier. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- attribute : Property [0..*]
- feature : Feature [0..*]

7.5.2.2.3 ConnectableElement

ConnectableElement is introduced as an abstract metaclass denoting elements than can be interconnected, by being attached to connector ends belonging to connectors.

Generalizations

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

Attributes

- None

Associations

- end : ConnectorEnd [0..*]

7.5.2.2.4 Connector

Connector is introduced to enable modeling of interconnections between parts and or roles, in the context of a structured classifier. This specification introduces the additional constraint that the type of a connector is not optional. Rationale for this constraint are given in clause 8.1.

Generalizations

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

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

Attributes

- None

Associations

- end : ConnectorEnd [2..*]
- redefinedConnector : Connector [0..*]
- type : Association [0..1]

Additional Constraints

- connector_is_typed
  A Connector must by typed by an association

```
inv: self.type->notEmpty()
```

7.5.2.2.5 ConnectorEnd

ConnectorEnd is introduced to allow modeling of Connectors.

Generalizations

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

Attributes

- None

Associations

- definingEnd : Property [0..1]
- role : ConnectableElement [1..1]

7.5.2.2.6 Feature

CompositeStructures::InternalStructures::Feature does not extend in any way fUML Feature. This metaclass is kept in the Composite Structures syntax model to make the merging process (described in the Overview sub-clause of the Abstract Syntax clause) meaningful.

Generalizations

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

Attributes

- None

Associations

- featuringClassifier : Classifier [0..*],

7.5.2.2.7 Property

CompositeStructures::InternalStructures::Property extends (merge increment) fUML Property. By being a ConnectableElement, a Property can be connected to other connectable elements, through connectors.

Generalizations

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

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

Attributes

- None

Associations

- class : Class [0..1]

7.5.2.2.8 StructuredClassifier

StructuredClassifier is introduced to enable modeling of internal structures comprising a network of parts and roles linked by connectors.

Generalizations

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

Attributes

- None

Associations

- ownedAttribute : Property [0..*]
- ownedConnector : Connector [0..*]
- part : Property [0..*]
- role : ConnectableElement [0..*]

7.5.3 InvocationActions

7.5.3.1 Overview

The InvocationActions package introduces extensions to fUML InvocationAction and Trigger to account for ports of an encapsulated classifier.

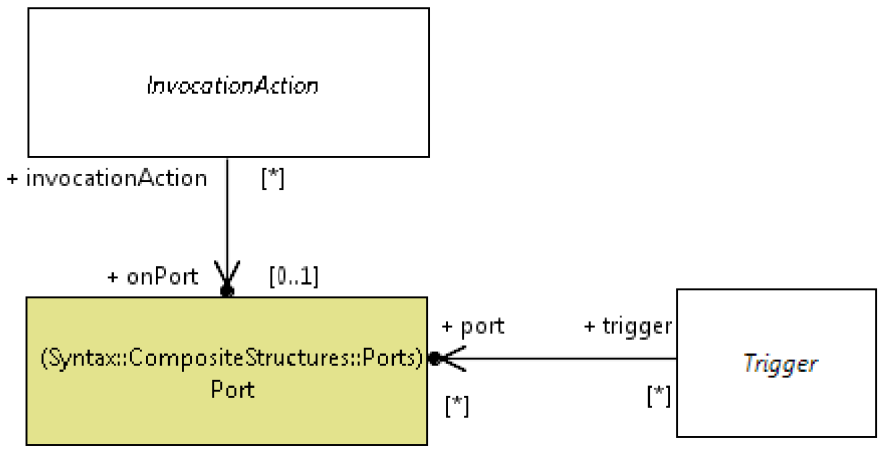

Figure 11: InvocationActions diagram

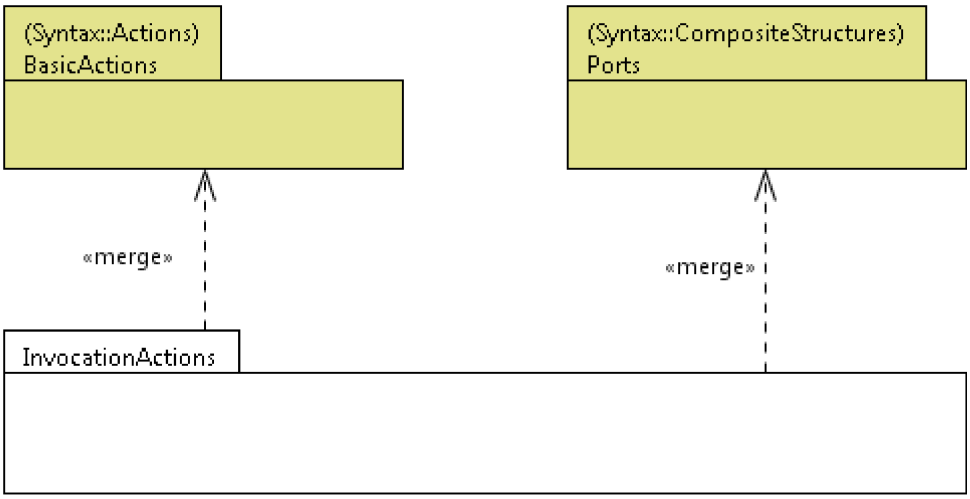

Figure 12: InvocationActions package relationships diagram

7.5.3.2 Class descriptions

7.5.3.2.1 InvocationAction

CompositeStructures::InvocationActions::InvocationAction extends (merge increment) fUML InvocationAction so that, in addition of targeting an object, the invocation (i.e., the call for an operation or the emission of a signal) can be made through a specific port of this object (onPort).

Generalizations

- None

Attributes

- None

Associations

- onPort : Port [0..1]

7.5.3.2.2 Trigger

CompositeStructures::InternalStructures::Trigger extends (merge increment) fUML Trigger so that a list of ports can be associated with a trigger.

Generalizations

- None

Attributes

- None

Associations

- port : Port [0..*]

7.5.4 Ports

7.5.4.1 Overview

The Ports package introduces mechanisms enabling a classifier to have an external structure. This external structure comprising a set of ports, where features provided to or required from the environment in which it will be instantiated are exposed.

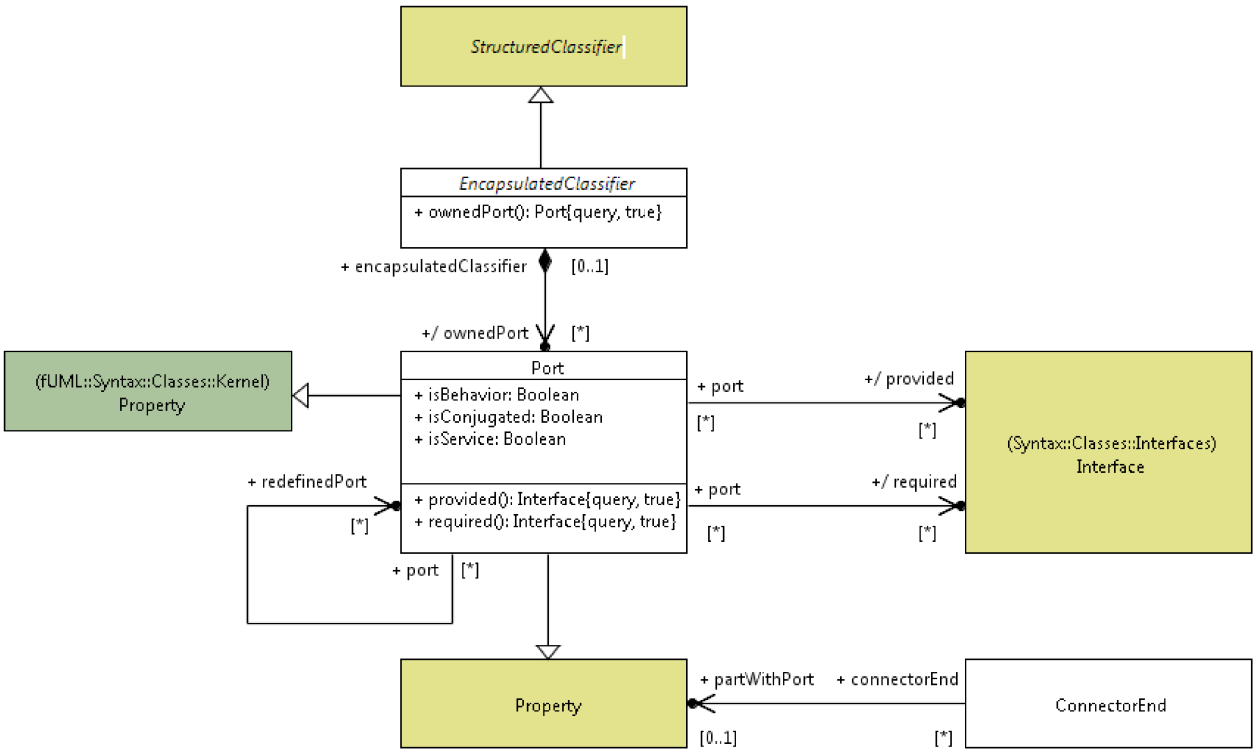

Figure 13: Ports diagram

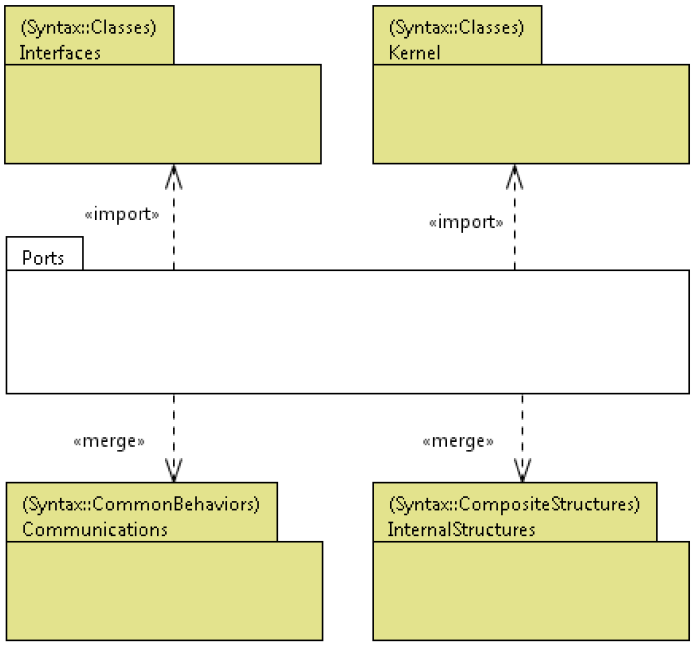

Figure 14: Ports package relationships diagram

7.5.4.2 Class descriptions

7.5.4.2.1 ConnectorEnd

CompositeStructures::Ports::ConnectorEnd extends (merge increment) CompositeStructures::InternalStructures::ConnectorEnd with an additional property (partWithPort) which enables to specify that, when the connected role is a Port, this port is connected in the context of a specific property of the context classifier.

Generalizations

- None

Attributes

- None

Associations

- partWithPort : Property [0..1]

7.5.4.2.2 EncapsulatedClassifier

EncapsulatedClassifier extends StructuredClassifier with the ability to own an external structure comprising a set of ports.

Generalizations

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

Attributes

- None

Associations

- ownedPort : Port [0..*]

7.5.4.2.3 Port

Port enables to specify interaction points through which instances of an encapsulated classifier can be connected and communicate with their environment. Port also enables to specify a set of features provided or required by the owning encapsulated classifier, that is to say features defined by provided or required interfaces associated with the port.

Generalizations

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

Attributes

- isBehavior : Boolean [1..1]
- isConjugated : Boolean [1..1]
- isService : Boolean [1..1]

Associations

- provided : Interface [0..*]
- redefinedPort : Port [0..*]
- required : Interface [0..*]

Additional constraints

- behavior_port_belongs_to_an_active_class
  A behavior port can only belong to an active class.

  ```
  inv: [TODO: Write OCL]
  ```

7.5.5 StructuredClasses

7.5.5.1 Overview

The StructuredClasses package is introduced to enable a Class to be both structured and encapsulated, which is the key aspects of UML Composite Structures.

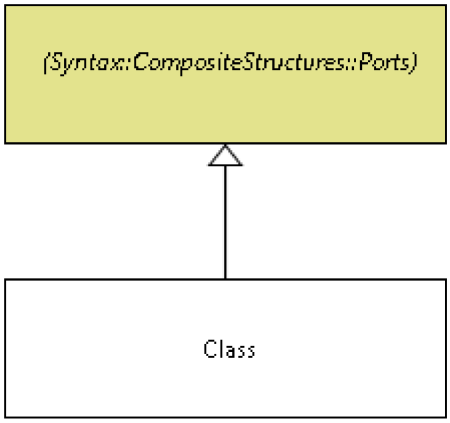

Figure 15: StructuredClasses diagram

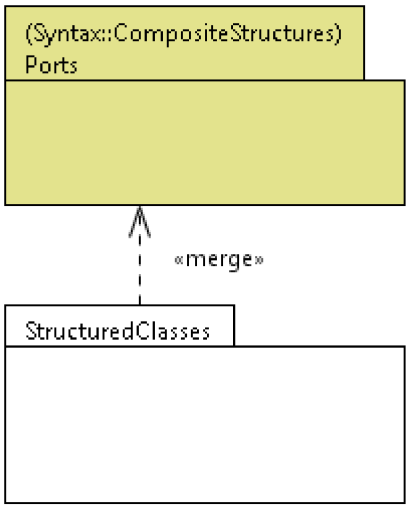

Figure 16: StructuredClasses package relationships diagram

7.5.5.2 Class descriptions

7.5.5.2.1 Class

CompositeStructures::StructuredClasses::Class extends (merge increment) fUML Class with the ability to be both structured and encapsulated. A class can have an internal structure comprising a network of linked parts, as well as an external structure comprising a set of ports.

Generalizations

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

Attributes

- None

Associations

- None

8 Semantics

8.1 Overview

The Semantics clause defines precise semantics for elements introduced in the Abstract Syntax clause. The semantic definition consists in extending the fUML execution model (defined in the Semantics clause of the "Semantics of a Foundational Subset for Executable UML Models" specification) with appropriate semantic visitors, semantic strategies and the definition of a semantic mapping between these elements and elements defined in the Abstract Syntax clause. By convention, all classes introduced in the extended execution model are prefixed by "CS_", which stands for "Composite Structures". Structural and behavioral semantics of UML composite structures are both covered by this extension.

Structural Semantics

Structural semantics concerns the run time manifestation of (structured/encapsulated) classes, which implies defining how ports, parts and connectors are represented at run time. [To be completed]

Behavioral Semantics

Behavioral semantics concerns the interpretation of Activities making statements (through appropriate activity nodes and edges) in relation with life cycle of objects (construction, destruction, observation and modification) and communication between those objects. [To be completed]

Semantic Strategies and Semantic Variants

The definition of semantic strategies (i.e., the places where this specification allows for semantic variability) and semantic variants (i.e., sets of consistent semantic choices in the scope of allowed variability) follows the same principles as in fUML. Each semantic strategy is defined by an abstract strategy class in the execution model. Defining a particular semantic choice consists in defining a concrete realization of this abstract strategy class. As in fUML, grouping concrete semantic strategy classes in order to define a consistent semantic variant is a tool implementation concern (cf. Clause 8.2.2.1 of "Semantics of a Foundational Subset for Executable UML Models", sub-clause on Configuring the Execution Environment at a Locus).

This specification introduces two new semantic strategies. CS_StructuralFeatureOfInterfaceAccessStrategy (clause 8.4.2.2.3) deals with reading and writing of structural features of an Interface, in connection with how these features are actually realized by a given behaviored classifier. CS_RequestPropagationStrategy (clause 8.5.2.2.4) deals with propagation of requests in the case where multiple propagation paths (e.g., multiple interaction points, multiple links) are valid. This specification also introduces CS_DispatchOperationOfInterfaceStrategy (clause 8.4.2.2.1), a new default realization of fUML DispatchStrategy. This realization deals with dispatching of operations of an Interface, in connection with how these operations are actually realized by a behaviored classifier.

Hypothesizes and constraints on the syntax

The execution model described in the following sub-clauses is written with the strong hypothesis that connectors are necessarily typed by associations. This constraint is introduced in clause 7.5.2.2.4 and motivated by the following reasons. Regarding Connectors in the context of StructuredClassifiers, the UML specification says *[Note for the initial submission: This sentence and the following in this paragraph are quoted from the UML 2.5 specification]*: *"A Connector specifies links (see 11.5 Associations) between two or more instances playing roles within a StructuredClassifier"*. For representing the runtime manifestation of Association instances, the fUML execution model already introduces a class called Link. In order to be consistent with fUML, this specification relies on fUML Link for managing the runtime manifestation of Connector instances (see CS_Link, clause 8.5.3.2.2). Since fUML Link has a non optional relationship with Association, it implies that Connectors must be typed by associations.

In practice, users rarely type connectors by associations, especially in cases where these connectors are used to connect elements through their ports. In order to take into account this common practice, this specification defines formal rules

(See normative Annex A – Model Transformations) for computing the default typing associations to be associated with untyped connectors. From a user standpoint, these rules enable to relax the constraints on connector typing, with the guaranty that a valid model (i.e., with connectors actually typed by associations computed according rules defined in Annex A) can be automatically derived before interpretation by the extended execution model.

Connector types are also required for managing semantics of object construction implied by UML Composite Structures. Regarding Connectors in the context of StructuredClassifiers, the UML specification also says: *"Links corresponding to Connectors may be created upon the creation of the instance of the containing StructuredClassifier"* and *"The topologies that result from matching the multiplicities of ConnectorEnds and those of ConnectableElements they interconnect cannot always be deduced from the model. Specific examples in which the topology can be determined from the multiplicities are shown in Figure 11-6 and Figure 11-7"*. The only action defined by UML for creating an object is CreateObjectAction, which has the following semantics: *"A CreateObjectAction is an Action that creates a direct instance of a given Classifier and places the new instance on its result OutputPin. [...] The new instance has no values for its StructuralFeatures and participates in no links"*. These semantics are formalized in the fUML execution model. In order to comply with fUML while taking into the semantics of UML Composite Structures about object creation (which implies the creation of a topology of instances and links, which contradicts semantics of CreateObjectAction where created objects are empty), Annex A of this specification also defines rules for generating construction behaviors simulating semantics of Composite Structures object creation. The generated behaviors typically includes CreateLinkActions for the construction of links corresponding to instances of connectors. CreateLinkActions rely on LinkEndData for identifying the actual end objects to be connected. As currently defined in UML, the identification of an end object by a LinkEndData requires the existence of an Association, which in turn requires that connectors are typed by associations to allow the creation of corresponding links.

The following sub-clauses provides details about how the fUML execution model is extended to address semantics of UML Composite Structures.

8.2 Actions

8.2.1 Overview

8.2.2 CompleteActions

8.2.2.1 Overview

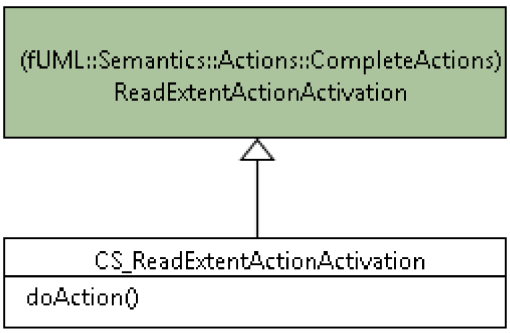

Figure 17:: CompleteActions diagram

8.2.2.2 Class descriptions

8.2.2.2.1 CS_ReadExtentActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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);
```

8.2.3 IntermediateActions

8.2.3.1 Overview

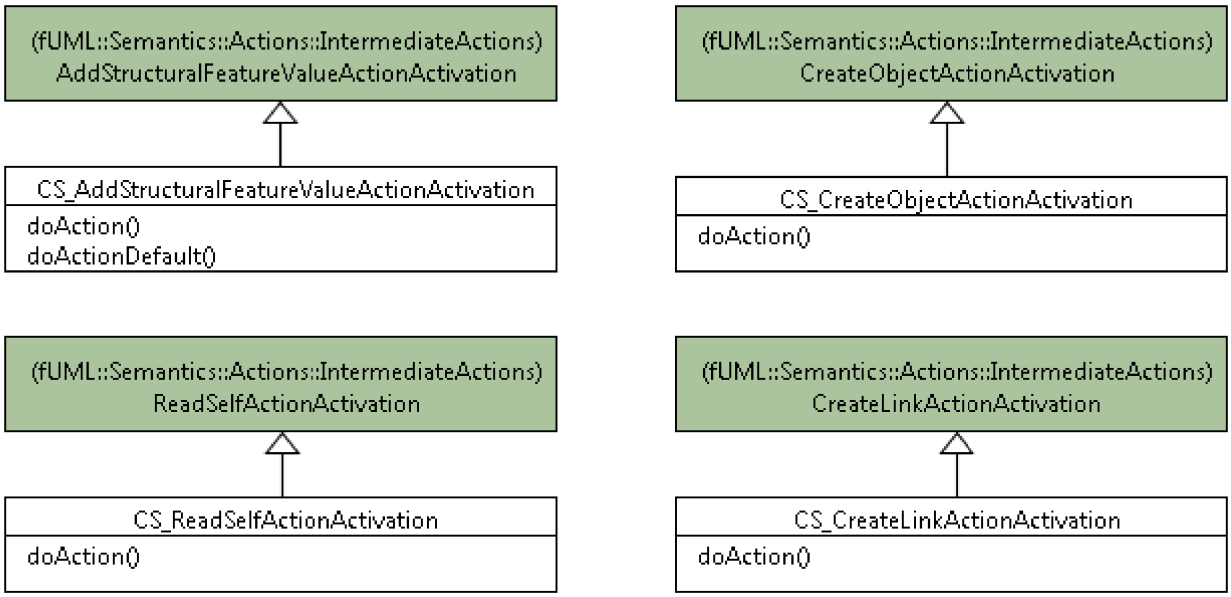

Figure 18:: IntermediateActions diagram

8.2.3.2 Class descriptions

8.2.3.2.1 CS_AddStructuralFeatureValueActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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(interactionPoint, 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(interactionPoint);
                        } else {
                        featureValue.values.addValue(insertAt - 1, interactionPoint);
                        }
                }
                if (action.result != null) {
                        this.putToken(action.result, owner);
                }
                }
                else {
                // behaves as usual
                this.doActionDefault() ;
                }
        }

[2] 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);
        }
```

8.2.3.2.2 CS_CreateLinkActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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);
```

8.2.3.2.3 CS_CreateObjectActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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);
```

8.2.3.2.4 CS_ReadSelfActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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();
        if (context.referent instanceof CS_Object) {// i.e. alternatively, it can be an execution
                context.compositeReferent = (CS_Object)context.referent ;
        }

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

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

8.3 Classes

8.3.1 Overview

8.3.2 Kernel

8.3.2.1 Overview

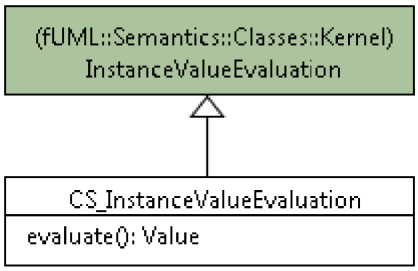

Figure 19:: Kernel diagram

8.3.2.2 Class descriptions

8.3.2.2.1 CS_InstanceValueEvaluation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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;
```

8.4 CommonBehaviors

8.4.1 Overview

8.4.2 Communications

8.4.2.1 Overview

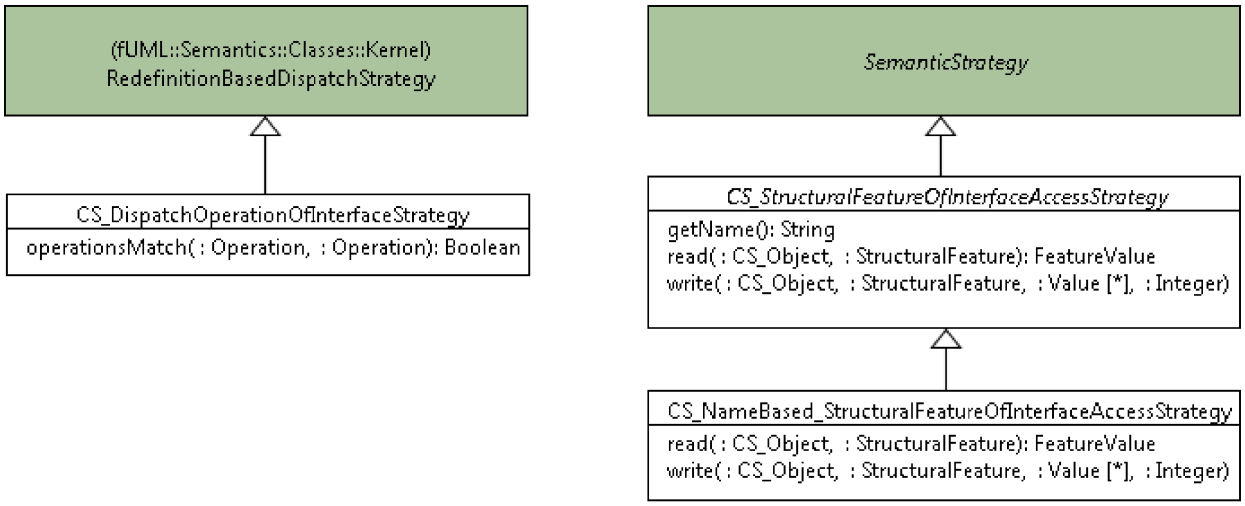

Figure 20:: Communications diagram

8.4.2.2 Class descriptions

8.4.2.2.1 CS_DispatchOperationOfInterfaceStrategy

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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 ;
```

8.4.2.2.2 CS_NameBased_StructuralFeatureOfInterfaceAccessStrategy

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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 ;


[2] 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) ;
        }
```

8.4.2.2.3 CS_StructuralFeatureOfInterfaceAccessStrategy

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] public getName() : String
        // StructuralFeatureAccessStrategy are always named "structuralFeature"
        return "structuralFeature";

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


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

8.5 CompositeStructures

8.5.1 Overview

8.5.2 InvocationActions

8.5.2.1 Overview

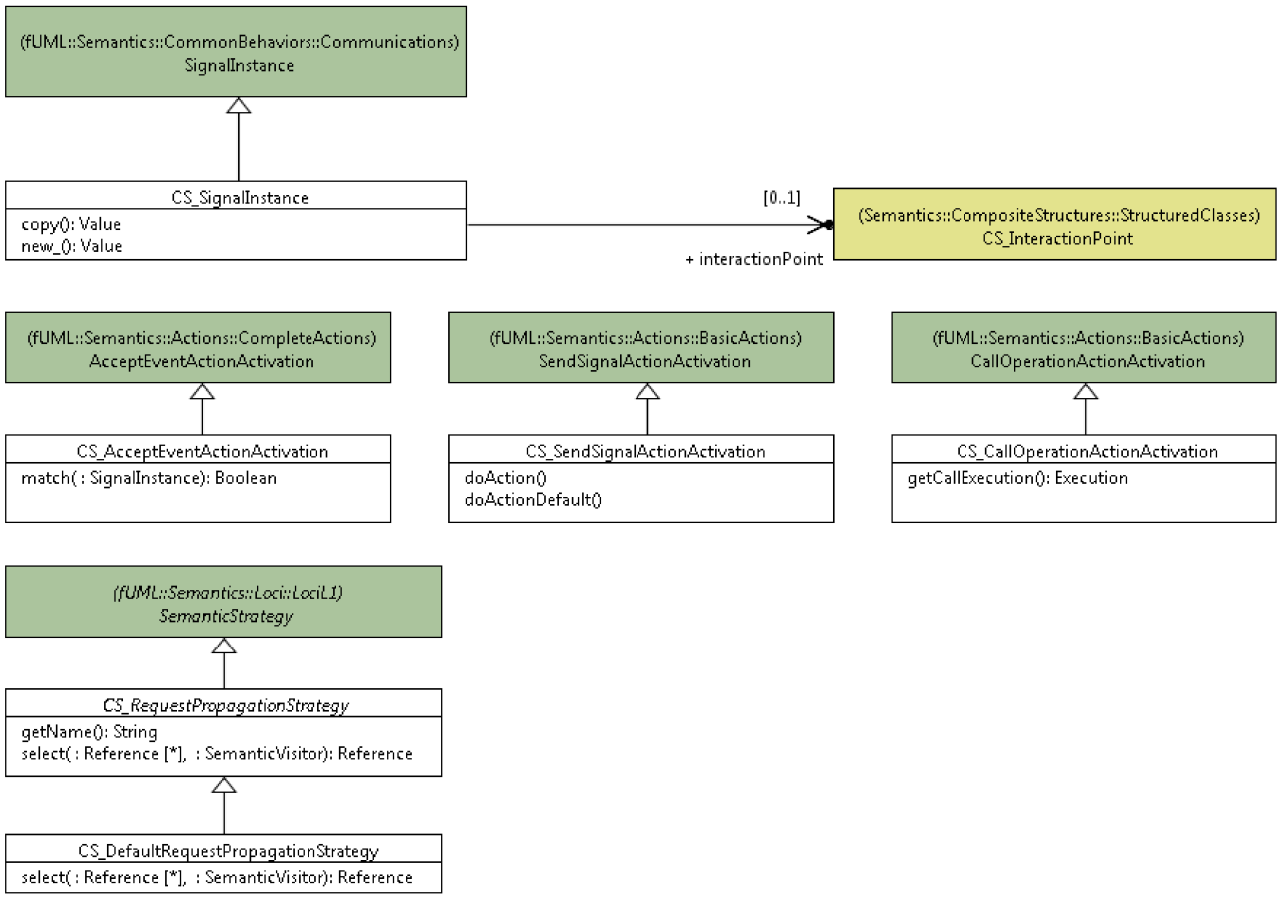

Figure 21:: InvocationActions diagram

8.5.2.2 Class descriptions

8.5.2.2.1 CS_AcceptEventActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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 && t.port.size()>0 ) {
                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;
```

8.5.2.2.2 CS_CallOperationActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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;
```

8.5.2.2.3 CS_DefaultRequestPropagationStrategy

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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;
```

8.5.2.2.4 CS_RequestPropagationStrategy

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] public getName() : String
      // a CS_RequestPropagationStrategy are always named "requestPropagation"
```

```
        return "requestPropagation";

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

8.5.2.2.5 CS_SendSignalActionActivation

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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 interface of 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

        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);
                }
                }
        }

[2] 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);
        }
```

8.5.2.2.6 CS_SignalInstance

Generalizations

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

Attributes

- None

Associations

- interactionPoint : CS_InteractionPoint[0..1], The InteractionPoint on which this signal instance occured.

Operations

```
[1] 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;

[2] public new_() : Value
        // Create a new signal instance with no type or feature values.
        return new CS_SignalInstance();
```

8.5.3 StructuredClasses

8.5.3.1 Overview

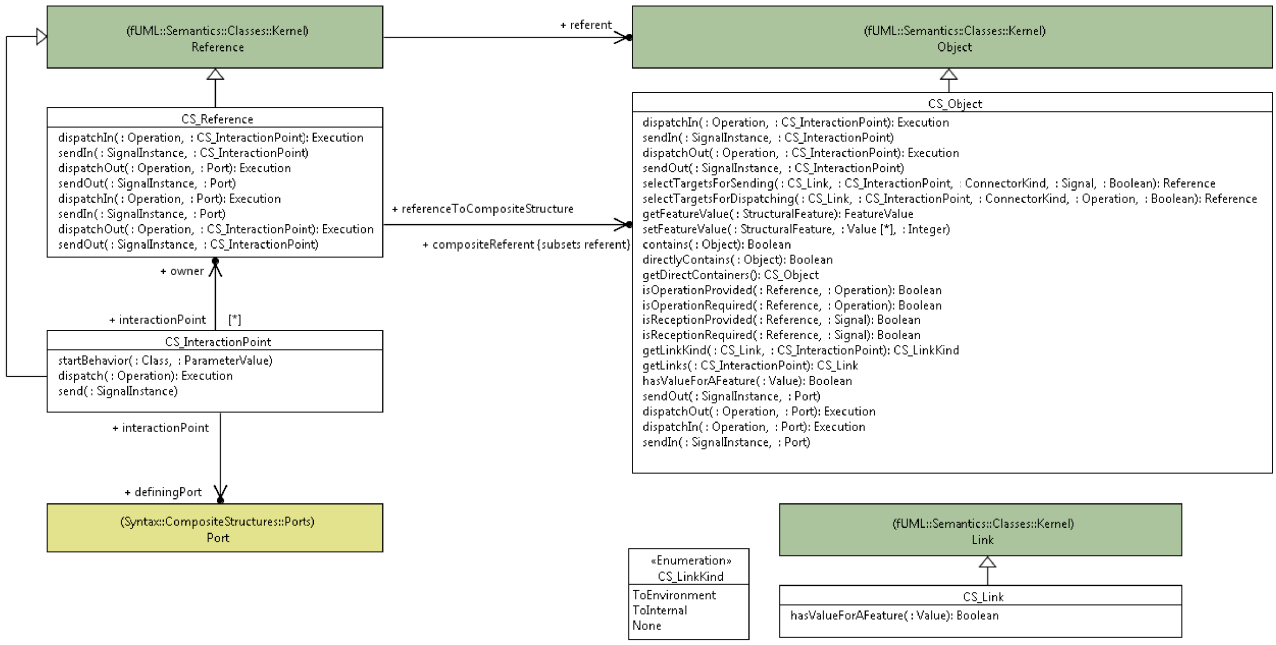

Figure 22:: StructuredClasses diagram

8.5.3.2 Class descriptions

8.5.3.2.1 CS_InteractionPoint

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

```
[1] public dispatch(operation:Operation) : Execution
       // Delegates dispatching to the owning object
       return this.owner.dispatchIn(operation, this) ;
```

```
[2] public send(signalInstance:SignalInstance)
        // Delegates sending to the owning object
        this.owner.sendIn(signalInstance, this) ;

[3] public startBehavior(classifier:Class, inputs:ParameterValue[*])
// Overriden to do nothing
```

8.5.3.2.2 CS_Link

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] public hasValueForAFeature(value:Value) : Boolean
        // Returns true if the given value object is used as a value for a FeatureValue of this link
        FeatureValueList allFeatureValues = this.getFeatureValues() ;
        Integer i = 1 ;
        boolean isAValue = false ;
        while (i <= allFeatureValues.size() && !isAValue) {
                FeatureValue featureValue = allFeatureValues.getValue(i-1);
                isAValue = !featureValue.values.isEmpty() &&
featureValue.values.getValue(0).equals(value) ;
                i = i + 1 ;
        }
        return isAValue ;
```

8.5.3.2.3 CS_LinkKind

Generalizations

- None

Enumeration literals

- ToEnvironment
- ToInternal
- None

8.5.3.2.4 CS_Object

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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;

[2] 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;

[3] 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 = new ReferenceList() ;
		CS_LinkList cddLinks = this.getLinks(interactionPoint) ;
		Integer linkIndex = 1 ;
		while (linkIndex <= cddLinks.size()) {
		ReferenceList validTargets =
this.selectTargetsForDispatching(cddLinks.getValue(linkIndex - 1),
			interactionPoint, ConnectorKind.delegation, operation, operationIsProvided) ;
		Integer targetIndex = 1 ;
		while(targetIndex <= validTargets.size()) {
			potentialTargets.add(validTargets.getValue(targetIndex-1)) ;
			targetIndex = targetIndex + 1 ;
		}
		linkIndex = linkIndex + 1 ;
		}
		// If potentialTargets 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 ;

[4] public dispatchIn(operation:Operation, onPort:Port) : Execution
        // delegates dispatching to composite referent
        // 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.locus.factory
                .getStrategy("choice"))
                .choose(featureValue.values.size()) - 1;
        CS_InteractionPoint interactionPoint = (CS_InteractionPoint)values.getValue(choice) ;
        return interactionPoint.dispatch(operation) ;


[5] public dispatchOut(operation:Operation, interactionPoint:CS_InteractionPoint) : Execution
        // Select appropriate delegation links from interactionPoint,
        // and propagates the operation call through these links
        // Appropriate links are either links which target elements
        // in the environment of this CS_Object.
        // These can be delegation links (i.e, the targeted elements must
        // require the operation) or assembly links (i.e., the target elements
        // must provide the operation)


        Execution execution = null ;

        boolean operationIsNotProvided = false ; // i.e. it is required
        ReferenceList allPotentialTargets = new ReferenceList() ;
        ReferenceList targetsForDispatchingIn = new ReferenceList() ;
        ReferenceList targetsForDispatchingOut = new ReferenceList() ;

        CS_LinkList cddLinks = this.getLinks(interactionPoint) ;
        Integer linkIndex = 1 ;
        while (linkIndex <= cddLinks.size()) {
                ReferenceList validAssemblyTargets =
this.selectTargetsForDispatching(cddLinks.getValue(linkIndex - 1),
                        interactionPoint, ConnectorKind.assembly, operation, operationIsNotProvided) ;
                Integer targetIndex = 1 ;
                while(targetIndex <= validAssemblyTargets.size()) {
                allPotentialTargets.addValue(validAssemblyTargets.getValue(targetIndex-1)) ;
                targetsForDispatchingIn.addValue(validAssemblyTargets.getValue(targetIndex-1)) ;
                targetIndex = targetIndex + 1 ;
                }
                ReferenceList validDelegationTargets =
this.selectTargetsForDispatching(cddLinks.getValue(linkIndex - 1),
                        interactionPoint, ConnectorKind.delegation, operation,
operationIsNotProvided) ;
                targetIndex = 1 ;
                while(targetIndex <= validDelegationTargets.size()) {
                allPotentialTargets.addValue(validDelegationTargets.getValue(targetIndex-1)) ;
                targetsForDispatchingOut.addValue(validDelegationTargets.getValue(targetIndex-1)) ;
                targetIndex = targetIndex + 1 ;
                }
                linkIndex = linkIndex + 1 ;
        }

        CS_RequestPropagationStrategy strategy =
(CS_RequestPropagationStrategy)this.locus.factory.getStrategy("requestPropagation") ;
        ReferenceList selectedTargets = strategy.select(allPotentialTargets, 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
                // i.e. a delegation connector for a required operation can only target a port
                CS_InteractionPoint cddTarget =
(CS_InteractionPoint)targetsForDispatchingOut.getValue(k) ;
                if (cddTarget == target) {
                        CS_Reference owner = cddTarget.owner ;
                        execution = owner.dispatchOut(operation, cddTarget) ;
                }
                }
        }
        return execution ;


[6] 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.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(0) ;
                execution = this.dispatchOut(operation, target) ;
        }
        return execution ;

[7] 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 ;

[8] 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);
        }

[9] public getLinkKind(link:CS_Link, interactionPoint:CS_InteractionPoint) : CS_LinkKind
        // If the given interaction point belongs to the given object,
        // and if the given interaction point is used as an end of the link,
        // then the links targets the environment of the object (enumeration literal ToEnvironment)
        // if all the feature values of the link
        // (but one for the interaction point) refer to values which are not themselves values
        // for features of the interaction point.
```

```
        // If all the feature values of the link refer to values which are themselves values for
        // features of the interaction point,
        // the link targets the internals of the object (enumeration literal ToInternal).
        // Otherwise, the link has no particular meaning
        // in the context defined by the object and the interaction point (enumeration literal None).
        if (! link.hasValueForAFeature(interactionPoint)) {
                return CS_LinkKind.None ;
        }
        CS_LinkKind kind = CS_LinkKind.ToInternal ;
        FeatureValueList featureValues = link.getFeatureValues() ;
        Integer i = 1 ;
        while (i <= featureValues.size() && kind != CS_LinkKind.None) {
                FeatureValue value = featureValues.getValue(i-1) ;
                if (value.values.isEmpty()) {
                kind = CS_LinkKind.None ;
                }
                else {
                Value v = value.values.getValue(0) ;
                boolean vIsAValueForAFeatureOfContext = false ;
                if (v.equals(interactionPoint)) {
                        vIsAValueForAFeatureOfContext = true ;
                }
                else if (v instanceof CS_InteractionPoint) {
                        v = ((CS_InteractionPoint)v).owner ;
                        vIsAValueForAFeatureOfContext = this.hasValueForAFeature(v) ;
                }
                else {
                        vIsAValueForAFeatureOfContext = this.hasValueForAFeature(v) ;
                }
                if (!vIsAValueForAFeatureOfContext) {
                        kind = CS_LinkKind.ToEnvironment ;
                }
                }
                i = i + 1 ;
        }
        return kind ;

[10] public getLinks(interactionPoint:CS_InteractionPoint) : CS_Link[*]
        // Get all links (available at the locus of this object) where the given
        // interaction point is used as a feature value
        // (i.e. the interaction is an end such links)
        ExtensionalValueList extensionalValues = this.locus.extensionalValues ;
        Integer i = 1 ;
        CS_LinkList connectorInstances = new CS_LinkList() ;
        while (i <= extensionalValues.size()) {
                ExtensionalValue value = extensionalValues.getValue(i-1) ;
                if (value instanceof CS_Link) {
                CS_Link link = (CS_Link)value ;
                if (this.getLinkKind(link, interactionPoint) != CS_LinkKind.None) {
                        connectorInstances.addValue(link) ;
                }
                }
                i = i + 1 ;
        }
        return connectorInstances ;

[11] public hasValueForAFeature(value:Value) : Boolean
        // Returns true if the given value object is used as a value for a feature
        // value of this object
        FeatureValueList allFeatureValues = this.getFeatureValues() ;
        Integer i = 1 ;
        boolean isAValue = false ;
        while (i <= allFeatureValues.size() && !isAValue) {
                FeatureValue featureValue = allFeatureValues.getValue(i-1);
                if (!featureValue.values.isEmpty()) {
                ValueList valuesForCurrentFeature = featureValue.values ;
                Integer j = 1 ;
                while (j <= valuesForCurrentFeature.size() && !isAValue) {
                        isAValue = featureValue.values.getValue(j-1).equals(value) ;
                        j = j + 1 ;
                }
                }
                i = i + 1 ;
        }
        return isAValue ;

[12] public isOperationProvided(reference:Reference, operation:Operation) : Boolean
```

```
	// Determines if the given reference provides the operation
	// If the reference is an interaction point, it provides the operation if this operation
	// is a member of one of its provided interfaces
	// If the reference is NOT an interactionPoint, it provides this operation if
	// this operation is an operation of one of its type, or one of its type provides a
	// realization for this operation (in the case
	// where the namespace of this Operation is an interface)
	boolean isProvided = false ;
	if (reference instanceof CS_InteractionPoint) {
		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 =
((CS_InteractionPoint)reference).definingPort.provided() ;
		while (interfaceIndex <= providedInterfaces.size() && !isProvided) {
			Interface interface_ = providedInterfaces.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 {
		// We have to look if one of the Classifiers typing this reference
		// directly or indirectly provides this operation
		ClassifierList types = reference.getTypes() ;
		Integer typeIndex = 1 ;
		while (typeIndex <= types.size() && !isProvided) {
		if (types.getValue(typeIndex - 1) instanceof Class_) {
			Integer memberIndex = 1 ;
			NamedElementList members = ((Class_)types.getValue(typeIndex - 1)).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 ;
			}
		}
		typeIndex = typeIndex + 1 ;
		}
	}
	return isProvided ;

[13] public isOperationRequired(reference:Reference, operation:Operation) : Boolean
	// Determines if the given reference requires the operation
	// If the reference is an interaction point, it requires the operation if this operation
	// is a member of one of its required interfaces
	// If the reference is not a interaction point, it cannot require an operation
	boolean matches = false ;
	if (reference instanceof CS_InteractionPoint) {
		Integer interfaceIndex = 1 ;
		// Iterates on provided interfaces of the port
		InterfaceList requiredInterfaces =
((CS_InteractionPoint)reference).definingPort.required() ;
		while (interfaceIndex <= requiredInterfaces.size() && !matches) {
		Interface interface_ = requiredInterfaces.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 ;

[14] public isReceptionProvided(reference:Reference, signal:Signal) : Boolean
        // Determines if the given reference provides a reception for signal
        // If the reference is an interaction point, it provides a reception if one of its
        // provided interfaces owns a reception for that signal
        // If the reference is NOT an interaction point, it provides a reception if one of its
        // types has a reception for that signal
        TypeList types = new TypeList() ;
        if (reference instanceof CS_InteractionPoint) {
                // types are interfaces provided by this port
                Integer interfaceIndex = 1 ;
                InterfaceList providedInterfaces =
((CS_InteractionPoint)reference).definingPort.provided() ;
                while (interfaceIndex <= providedInterfaces.size()) {
                Interface interface_ = providedInterfaces.getValue(interfaceIndex-1) ;
                types.addValue(interface_) ;
                interfaceIndex = interfaceIndex + 1 ;
                }
        }
        else {
                ClassifierList referenceTypes = reference.getTypes() ;
                Integer typeIndex = 1 ;
                while (typeIndex <= referenceTypes.size()) {
                types.addValue(referenceTypes.getValue(typeIndex - 1)) ;
                typeIndex = typeIndex + 1 ;
                }
        }
        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 - 1) ;
                if (cddReception instanceof Reception) {
                        if (((Reception)cddReception).signal == signal) {
                        matches = true ;
                        }
                }
                memberIndex = memberIndex + 1 ;
                }
                typeIndex = typeIndex + 1 ;
        }
        return matches ;

[15] public isReceptionRequired(reference:Reference, signal:Signal) : Boolean
        // Determines if the given reference requires a reception for signal
        // If the reference is an interaction point, it requires a reception if one of its
        // required interfaces owns a reception for that signal
        // If the reference is NOT an interaction point, it cannot require a reception
        boolean matches = false ;
        TypeList types = new TypeList() ;
        if (reference instanceof CS_InteractionPoint) {
                // types are interfaces provided by this port
                Integer interfaceIndex = 1 ;
                InterfaceList requiredInterfaces =
((CS_InteractionPoint)reference).definingPort.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-1) ;
                        if (cddReception instanceof Reception) {
                        if (((Reception)cddReception).signal == signal) {
                                matches = true ;
                        }
```

```
                        }
                        memberIndex = memberIndex + 1 ;
                }
                interfaceIndex = interfaceIndex + 1 ;
                }
        }
        return matches ;

[16] public selectTargetsForDispatching(link:CS_Link, interactionPoint:CS_InteractionPoint,
connectorKind:ConnectorKind, operation:Operation, isProvided:Boolean) : Reference[*]
        // From the given link, operation and interaction point, retrieves potential targets
        // (i.e. end values of link)
        // through which request can be propagated
        // These targets are attached to interaction point through the given link, and respect
        // the following rules:
        // - if isProvided is true, connectorKind must be Delegation, the given link has to target
        // the internals of this CS_Object,
        // and a valid target must provide the Operation
        // - if isProvided is false, the given link has to target the environment of this CS_Object.
        //      - if connectorKind is assembly, a valid target has to provide the operation
        //      - if connectorKind is delegation, a valid target has to require the operation
        ReferenceList potentialTargets = new ReferenceList() ;
        if (isProvided && connectorKind == ConnectorKind.delegation) {
                if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToInternal) {
                Integer i = 1 ;
                while(i <= link.getFeatureValues().size()) {
                        ValueList values = link.getFeatureValues().getValue(i-1).values ;
                        if (!values.isEmpty() && values.getValue(0) instanceof Reference) {
                        Reference cddTarget = (Reference)values.getValue(0) ;
                        if (this.isOperationProvided(cddTarget, operation)) {
                                potentialTargets.add(cddTarget) ;
                        }
                        }
                }
                }
        }
        else { // !isProvided
                if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToEnvironment) {
                Integer i = 1 ;
                while(i <= link.getFeatureValues().size()) {
                        ValueList values = link.getFeatureValues().getValue(i-1).values ;
                        if (!values.isEmpty() && values.getValue(0) instanceof Reference) {
                        Reference cddTarget = (Reference)values.getValue(0) ;
                        if (connectorKind == ConnectorKind.assembly) {
                                if (!cddTarget.equals(interactionPoint) &&
this.isOperationProvided(cddTarget, operation)) {
                                potentialTargets.add(cddTarget) ;
                                }
                        }
                        else { // delegation
                                if (!cddTarget.equals(interactionPoint) &&
this.isOperationRequired(cddTarget, operation)) {
                                potentialTargets.add(cddTarget) ;
                                }
                        }
                        }
                        i = i + 1 ;
                }
                }
        }
        return potentialTargets ;

[17] public selectTargetsForSending(link:CS_Link, interactionPoint:CS_InteractionPoint,
connectorKind:ConnectorKind, signal:Signal, isProvided:Boolean) : Reference[*]
        // From the given link, signal and interaction point, retrieves potential targets (
        // i.e. end values of link) through which request can be propagated
        // These targets are attached to interaction point through the given link,
        // and respect the following rules:
        // - if isProvided is true, connectorKind must be Delegation, the given link has
        // to target the internals of this CS_Object,
        // and a valid target must provide a reception for the signal
        // - if isProvided is false, the given link has to target the environment of this CS_Object.
        //      - if connectorKind is assembly, a valid target has to provide a reception
        //        for the signal
        //      - if connectorKind is delegation, a valid target has to require a reception for
        //        the signal
        ReferenceList potentialTargets = new ReferenceList() ;
```

```
        if (isProvided && connectorKind == ConnectorKind.delegation) {
                if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToInternal) {
                Integer i = 1 ;
                while(i <= link.getFeatureValues().size()) {
                        ValueList values = link.getFeatureValues().getValue(i-1).values ;
                        if (!values.isEmpty()) {
                        Integer j = 1 ;
                        while (j <= values.size()) {
                                Reference cddTarget = (Reference)values.getValue(j-1) ;
                                if (!cddTarget.equals(interactionPoint) &&
this.isReceptionProvided(cddTarget, signal)) {
                                potentialTargets.add(cddTarget) ;
                                }
                                j = j + 1 ;
                        }
                        }
                        i = i + 1 ;
                }
                }
        }
        else { // !isProvided
                if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToEnvironment) {
                Integer i = 1 ;
                while(i <= link.getFeatureValues().size()) {
                        ValueList values = link.getFeatureValues().getValue(i-1).values ;
                        if (!values.isEmpty() && values.getValue(0) instanceof Reference) {
                        Reference cddTarget = (Reference)values.getValue(0) ;
                        if (connectorKind == ConnectorKind.assembly) {
                                if (this.isReceptionProvided(cddTarget, signal)) {
                                potentialTargets.add(cddTarget) ;
                                }
                        }
                        else { // delegation
                                if (this.isReceptionRequired(cddTarget, signal)) {
                                potentialTargets.add(cddTarget) ;
                                }
                        }
                        }
                        i = i + 1 ;
                }
                }
        }
        return potentialTargets ;


[18] 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 = new ReferenceList() ;
                CS_LinkList cddLinks = this.getLinks(interactionPoint) ;
                Integer linkIndex = 1 ;
                while (linkIndex <= cddLinks.size()) {
                ReferenceList validTargets = this.selectTargetsForSending(cddLinks.getValue(linkIndex
- 1), interactionPoint, ConnectorKind.delegation, signalInstance.type, receptionIsProvided) ;
                Integer targetIndex = 1 ;
                while(targetIndex <= validTargets.size()) {
                        potentialTargets.add(validTargets.getValue(targetIndex-1)) ;
                        targetIndex = targetIndex + 1 ;
                }
                linkIndex = linkIndex + 1 ;
                }
                // If potential targets is empty, no delegation target has been found,
                // and the signal is lost
                // Otherwise, do the following concurrently
                for (int i = 0 ; i < potentialTargets.size() ; i++) {
```

```
                Reference target = potentialTargets.getValue(i) ;
                CS_SignalInstance newSignalInstance = (CS_SignalInstance)signalInstance.copy() ;
                newSignalInstance.interactionPoint = interactionPoint ;
                target.send(newSignalInstance) ;
                }
        }

[19] 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.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) ;
        }

[20] public sendOut(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)
        // Select appropriate delegation links from interactionPoint,
        // and propagates the signal instance through these links
        // Appropriate links are links which target elements
        // in the environment of this CS_Object.
        // These can be delegation links (i.e, the targeted elements must
        // require a reception for the signal) or assembly links (i.e., the target elements
        // must provide a reception for the signal)

        boolean receptionIsNotProvided = false ; // i.e. it is required
        ReferenceList allPotentialTargets = new ReferenceList() ;
        ReferenceList targetsForSendingIn = new ReferenceList() ;
        ReferenceList targetsForSendingOut = new ReferenceList() ;

        CS_LinkList cddLinks = this.getLinks(interactionPoint) ;
        Integer linkIndex = 1 ;
        while (linkIndex <= cddLinks.size()) {
                ReferenceList validAssemblyTargets =
this.selectTargetsForSending(cddLinks.getValue(linkIndex - 1),
                        interactionPoint, ConnectorKind.assembly, signalInstance.type,
receptionIsNotProvided) ;
                Integer targetIndex = 1 ;
                while(targetIndex <= validAssemblyTargets.size()) {
                allPotentialTargets.addValue(validAssemblyTargets.getValue(targetIndex-1)) ;
                targetsForSendingIn.addValue(validAssemblyTargets.getValue(targetIndex-1)) ;
                targetIndex = targetIndex + 1 ;
                }
                ReferenceList validDelegationTargets =
this.selectTargetsForSending(cddLinks.getValue(linkIndex - 1),
                        interactionPoint, ConnectorKind.delegation, signalInstance.type,
receptionIsNotProvided) ;
                targetIndex = 1 ;
                while(targetIndex <= validDelegationTargets.size()) {
                allPotentialTargets.addValue(validDelegationTargets.getValue(targetIndex-1)) ;
                targetsForSendingOut.addValue(validDelegationTargets.getValue(targetIndex-1)) ;
                targetIndex = targetIndex + 1 ;
                }
                linkIndex = linkIndex + 1 ;
        }

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

        for (int j = 0 ; j < selectedTargets.size() ; j++) {
                Reference target = selectedTargets.getValue(j) ;
                for (int k = 0 ; k < targetsForSendingIn.size() ; k++) {
                Reference cddTarget = targetsForSendingIn.getValue(k) ;
                if (cddTarget == target) {
                        target.send(signalInstance) ;
                }
                }
```

```
                for (int k = 0 ; k < targetsForSendingOut.size() ; k++) {
                // The target must be an interaction point
                // i.e. a delegation connector for a required reception can only target a port
                CS_InteractionPoint cddTarget =
(CS_InteractionPoint)targetsForSendingOut.getValue(k) ;
                if (cddTarget == target) {
                        CS_Reference owner = cddTarget.owner ;
                        owner.sendOut(signalInstance, cddTarget) ;
                }
                }
        }

[21] 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.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.sendOut(signalInstance, target) ;
        }

[22] 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);
        }
```

8.5.3.2.5 CS_Reference

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

```
[1] public dispatchIn(operation:Operation, interactionPoint:CS_InteractionPoint) : Execution
        //Delegates dispatching to composite referent
        return this.compositeReferent.dispatchIn(operation, interactionPoint) ;


[2] public dispatchIn(operation:Operation, onPort:Port) : Execution
        // delegates dispatching to composite referent
        return this.compositeReferent.dispatchIn(operation, onPort) ;

[3] public dispatchOut(operation:Operation, onPort:Port) : Execution
```

```
        // delegates dispatching to composite referent
        return this.compositeReferent.dispatchOut(operation, onPort) ;

[4] 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) ;

[5] public sendIn(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)
        // delegates sending to composite referent
        this.compositeReferent.sendIn(signalInstance, interactionPoint) ;

[6] public sendIn(signalInstance:SignalInstance, onPort:Port)
        // delegates sending to composite referent
        this.compositeReferent.sendIn(signalInstance, onPort) ;

[7] public sendOut(signalInstance:SignalInstance, onPort:Port)
        // delegates sending to composite referent
        this.compositeReferent.sendOut(signalInstance, onPort) ;

[8] public sendOut(signalInstance:SignalInstance, interactionPoint:CS_InteractionPoint)
        // Delegates sending (through the interaction point, to the environment)
        // to compositeReferent
        this.compositeReferent.sendOut(signalInstance, interactionPoint) ;
```

8.6 Loci

8.6.1 Overview

8.6.2 LociL3

8.6.2.1 Overview

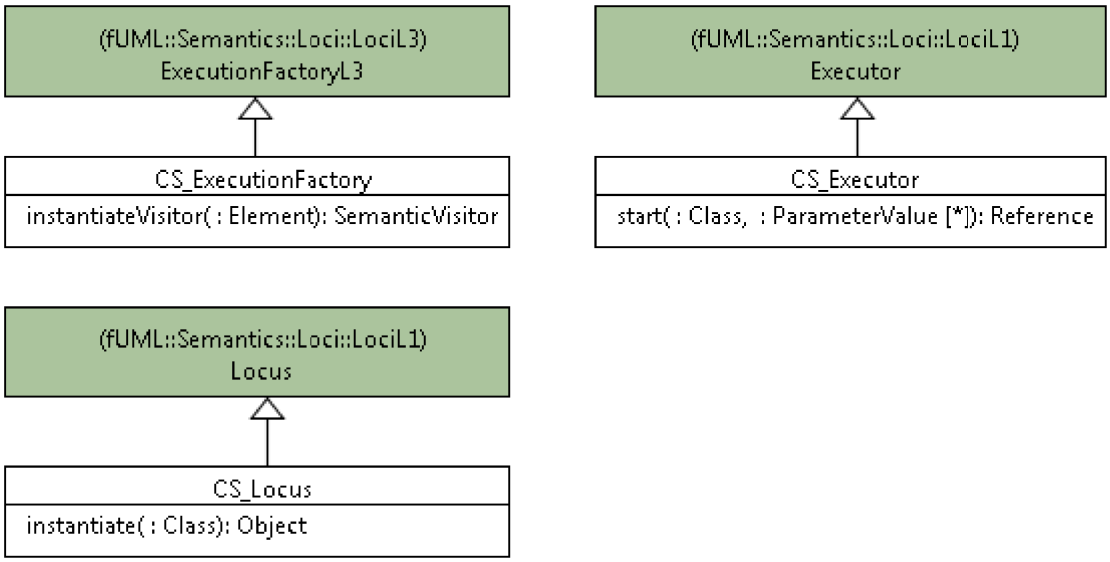

Figure 23:: LociL3 diagram

8.6.2.2 Class descriptions

8.6.2.2.1 CS_ExecutionFactory

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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 ;
```

8.6.2.2.2 CS_Executor

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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;
```

8.6.2.2.3 CS_Locus

Generalizations

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

Attributes

- None

Associations

- None

Operations

```
[1] 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;
```

9 Test Suite

[TODO: Some blabla on what we expect for the revised submission. Maybe include the current basic example?]

Annex A (normative) – Model Transformations

Rules for the generation of default typing associations for untyped connectors:
[TODO]

Rules for the generation of default typing classes for ports typed by interfaces:
[TODO]

Rules for the generation of default constructor operations and methods

This generalization rule requires the two previous ones.

[TODO:

The following cases should be addressed:

- A construction operation is already defined, with an InstanceSpecification specifying the resulting topology of links and instances]
- No construction operation is defined at all

]

Notes from the telco on 10th of October:

Structural semantics of composite structures (i.e. topologies resulting from instantiation):

- Handled with explicit constructor operations (and corresponding methods) to encapsulate construction of objects
- The activity method of each constructor is intended to produce a configuration of instances which complies with structural semantics of class instantiation
- => i.e., this is also something that we could consider as the result of a "pre-processing" transformation, as defined in the following UML 2.5 semantic clauses of Encapsulated Classifier, Structured Classifier, and Class:
    - EncapsulatedClassifier:
        - "When an instance of an EncapsulatedClassifier is created, instances corresponding to each of its Ports are created and held in the slots specified by each Port, in accordance with its type and multiplicity. These instances are referred to as "interaction points" and provide unique references."
        - => In the constructors, we have CreateObjectActions for creating values of port, and AddStructuralFeatureValueActions for setting the port with the created values
    - StructuredClassifier:
        - "The multiplicities on ConnectableElements constrain the number of objects that may be created within an instance of the containing StructuredClassifier, according to the semantics of MultiplicityElement"
        - => In the constructors, we have CreateObjectActions for creating values of properties, CallOperationActions on "constructors" of these objects, and AddStructuralFeatureValueActions for setting the properties with the created values
        - "Links corresponding to Connectors may be created upon the creation of the instance of the containing StructuredClassifier."
        - => In the constructors, we have CreateLinkActions for creating links between values (or interaction point on this values) of properties and ports, according to declared connectors.
    - Class:
        - "A Class may be designated by setting isActive to true as active (i.e., each of its instances is an active object). When isActive is false the Class is passive (i.e., each of its instances executes within the context of some other object). An active object is an object that, as a direct consequence of its creation, commences to execute its classifierBehavior, and does not cease until either the complete Behavior is executed or the object is terminated by some external object."
        - Note: the Alf to fUML mapping for constructor calls implements these semantics.

- => Alf semantics is “call constructor” and then “start classifier behavior”
  - startClassifierBehavior should not be part of the constructor method
- => use startObjectBehavior instead of startClassifierBehavior (which is deprecated)
- => In the constructors for active classes, we have StartClassifierBehaviorActions for starting the classifier behavior of the constructed object
- => Cf. Eldad example for alternative semantics (To be discussed in the next telco)

Annex B (non-normative) – Extensions to Alf

[TODO: Some blabla about why we will define extensions to Alf, in connection with the definition of examples, the test suite and coverage criteria. Some idea about what the solution could be]

Annex C (non-normative) – Semantics of MARTE GCM

[TODO: Some idea about what the solution could be, or remove this section from the initial submission]

Annex D (non-normative) – Semantics of SysML Blocks, Ports and Flows

[TODO: Some idea about what the solution could be, or remove this section from the initial submission]