Precise Semantics of UML Composite Structure (PSCS)

Version 1.2

Table of Contents

Preface

OMG

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

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

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

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Specifications are available from the OMG website at:

http://www.omg.org/spec

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

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

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

Issues

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

1 Scope

This specification defines an extension of fUML syntax and semantics to enable modeling and execution of UML composite structures. The term "composite structures" refers to the ability of UML classes to be structured as defined in Clause 9 of the UML specification.

The syntactic extensions add the capability for a fUML class to have an internal structure that may include public, private, and protected ports as well as a network of parts linked by connectors.

The semantic extensions supplement the fUML execution model with appropriate visitor classes, strategy classes, and semantic mappings to account for structural and behavioral semantics implied by the syntactic extensions.

The overall purpose of this specification is illustrated in Figure 1.1, with extension of fUML syntax on the left-hand side, extension of fUML semantics on the right-hand side and extension of the semantic mapping in the middle.
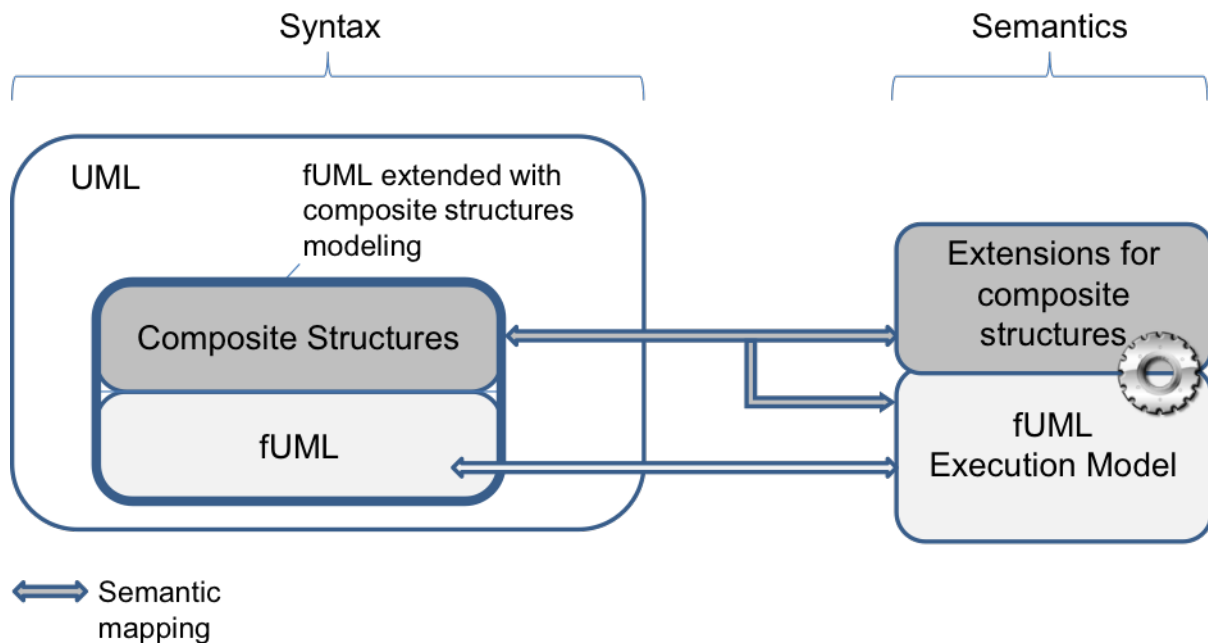
Figure 1.1 - Scope of this Specification

2 Conformance

2.1 General

This specification defines a precise semantics for a subset of UML 2 concepts associated with the notion of a structured class. This definition is based on the Foundational UML Subset (henceforth referred to as "fUML" in the remainder of this clause), which is specified in the OMG recommendation "Semantics of a Foundational Subset for Executable UML Models". Hence, *except where explicitly noted in this clause*, the definitions, interpretations (meaning), and types of conformance and related terms in this specification fully match their corresponding definitions, interpretations, and types in fUML. Thus, as in fUML, conformance to this specification has two aspects:

1. *Syntactic Conformance.* A conforming model must be restricted to the abstract syntax subset defined in Clause 7 of this document.

2. *Semantic Conformance.* A conforming execution tool must provide execution semantics for a conforming model consistent with the semantics specified in Clause 8 of this document. Passing all the tests of the test suites in Clause 9 are sufficient to demonstrate conformance with the semantics specified in Clause 8.

2.2 Genericity of the Execution Model

To support a variety of different execution paradigms and environments, the specification of the execution model incorporates a degree of genericity. This is achieved in two ways: (1) by leaving some key semantic elements unconstrained, and (2) by defining explicit semantic variation points. A particular execution tool can then realize specific semantics by suitably constraining the unconstrained semantic aspects and providing specifications for any desired variation at semantic variation points.

The semantic areas that are not explicitly constrained by the execution model in this specification are the same as the ones defined in subclause 2.3 of the fUML specification. Different execution tools may semantically vary in the above areas in executing the same model, while still being conformant to the semantics specified by the execution model in this specification. Additional semantic specifications or constraints may be provided for a specific execution tool in these areas, so long as it remains, overall, conformant to the execution model. For instance, a particular tool may be limited to a single centralized time source such that all time measurements can be fully ordered.

In contrast to the above areas, the items below are explicit semantic variation points. That is, the execution model as given in this specification by default fully specifies the semantics of these items. However, it is allowable for a conforming execution tool to define alternate semantics for them, so long as this alternative is fully specified as part of the conformance statement for the tool.

- The semantic variation points defined in subclause 2.3 of the fUML specification. Note, however, that default operation call dispatching strategy defined for fUML is replaced by a new default strategy specific to this specification (see 8.4.2.2).

- In addition, this specification introduces three new semantic variation points and corresponding default strategies for them:

    o The method of reading and writing of the structural features of an Interface, depending on how they are realized by the corresponding behavior classifier (see 8.4.2.10)

    o The method of propagating requests in situations where there are multiple possible paths (see 8.4.2.9).

    o The method for constructing an object, which includes creation of a topology of objects, interaction points and links, as well as initialization of corresponding properties according to any specified default values (see 8.6.2.6)

If a conforming execution tool wishes to implement a semantic variation in one of the above areas, then a specification most be provided for this variation via a specialization of the appropriate execution model class as identified above. This specification must be provided as a fUML model in the "base UML" subset interpretable by the base semantics of Clause 10 of the fUML specification. Further, it must be defined in what cases the variation is used and, if different variants may be used in different cases, when each variant applies, and/or how what variant to use, is to be specified in a conforming model accepted by the execution tool.

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.

1. OMG Unified Modeling Language (OMG UML), Version 2.5.1, http://www.omg.org/spec/UML/2.5.1.

2. Semantics of a Foundational Subset for Executable UML Models (FUML), Version 1.4, http://www.omg.org/spec/FUML/1.4.

4 Terms and Definitions

There are no formal definitions of terms in this specification.

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Changes to Adopted OMG Specifications

This specification does not require any changes to adopted OMG specifications.

6.2 Acknowledgments

This specification was originally authored by:

- Conrad Bock (NIST)
- Yves Bernard (EADS)
- Michael Chonoles (Lockheed Martin)
- Arnaud Cuccuru (Commissariat à l'Energie Atomique)
- Peter Denno (NIST)
- Sébastien Gérard (Commissariat à l'Energie Atomique)
- Sanford Friedenthal (INCOSE)
- Nerijus Jankevicius (No Magic)
- Julio Medina (Universidad de Cantabria)
- Eldad Palachi (International Business Machines)
- Laurent Rioux (THALES)
- Nicolas Rouquette (JPL/NASA)
- Ed Seidewitz (Model Driven Solutions)
- Bran Selic (Simula Research Laboratory)
- Ed Shaw (NexJ Systems)
- Daniel Siegl (LieberLieber)

This page intentionally left blank

7 Abstract Syntax

7.1 Overview

This clause describes how the fUML syntax model is extended to include modeling of UML Composite Structures. This extension refers to the additional UML metaclasses, constraints, and relationships that enable a fUML Class to be structured. The term "structured" here means that a Class may have public, private, and protected Ports as well as a network of parts linked through Connectors. The Semantics Clause 8 defines precise semantics for the elements introduced in this section.

The PSCS subset definition is formally captured in the package PSCS_Syntax::Syntax. This package includes the subpackages shown in Figure 7.1, each of which imports into its namespace all the elements from the corresponding fUML abstract syntax subset package. In addition, each of these subpackages imports those additional elements from the corresponding UML abstract syntax package that are in the PSCS subset but not the fUML subset. All the elements in the subpackages are then re-imported into the top-level Syntax package, which allows them to be uniformly referenced by qualified name directly from the top-level package (similarly to the namespace structure used in the UML abstract syntax metamodel). Finally, the top-level Syntax package directly imports the fUML subset packages for Classification, Packages and Activities, since the PSCS subset does not add any additional metaclasses in these areas.
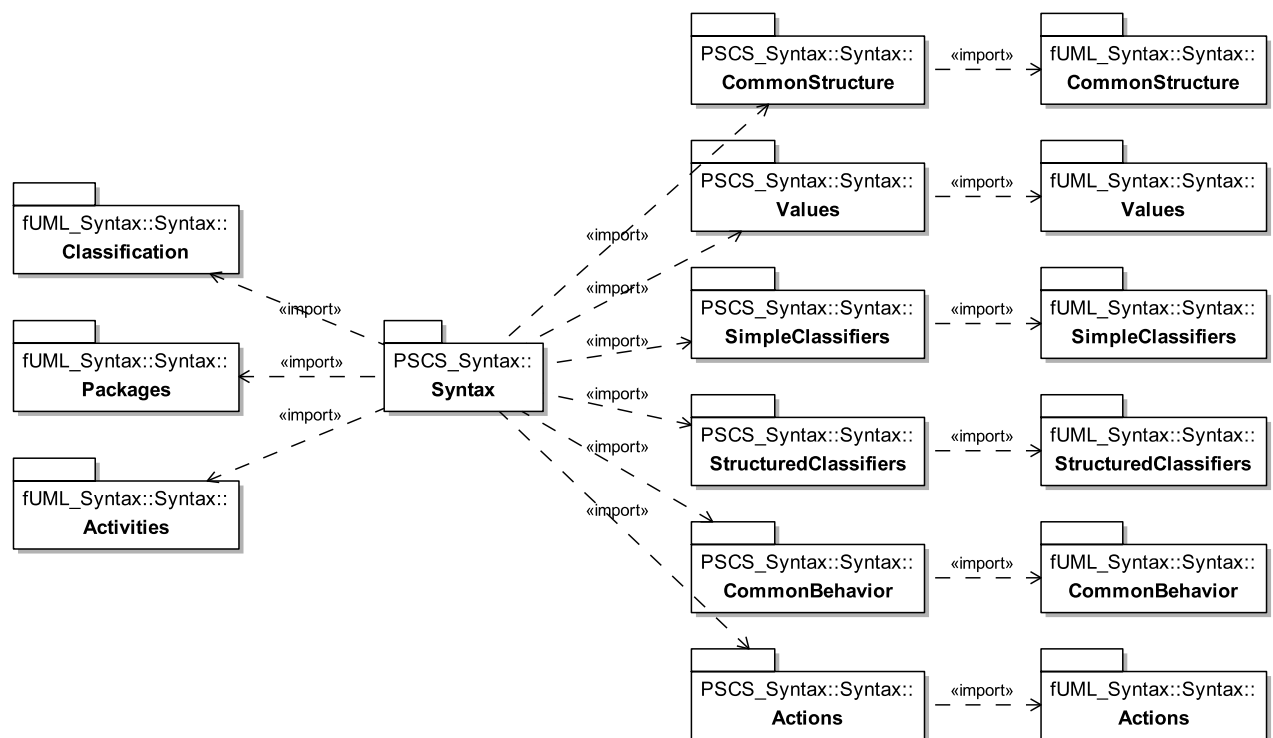
Figure 7.1 - PSCS Syntax Package

A UML model that syntactically conforms to this subset shall have an abstract syntax representation that consists solely of instances of metaclasses that are (imported) members of the PSCS_Syntax::Syntax package. For simplicity, meta-associations from the UML abstract syntax metamodel are *not* explicitly imported into the PSCS_Syntax::Syntax package, but it is, nevertheless, permissible for the model elements of a conforming model, within the PSCS subset, to be involved in any meta-associations consistent with both the UML metamodel and any further constraints as defined in this specification.

Note. This approach for defining a subset of the UML abstract syntax is consistent with the approach used for fUML (see the fUML Specification, 7.2). It is similar to the approach used for defining the metamodel subset covered by a UML profile, in which specially identified package imports (metamodelReferences) and element imports

(metaclassReferences) are used to import the metaclasses from the subset into the namespace of the Profile (see the UML 2 Specification, 12.3).

In addition to being representable within the PSCS abstract syntax subset, as described above, a UML model that syntactically conforms to PSCS shall also satisfy all relevant constraints defined in the UML abstract syntax metamodel *and* the additional syntactic constraints specified here for PSCS. The PSCS semantics specified in Clause 8 are only defined for well-formed PSCS models that meet all the necessary constraints.

The constraints specified for PSCS are all those that are imported members of the PSCS_Syntax::Constraints package (see Figure 7.2). Each of these constraints has as its single constrained element the UML abstract syntax metaclass to which the constraint applies. Almost all of the constraints in the PSCS_Syntax::Constraints package are imported from the fUML_Syntax::Constraints package. PSCS only adds two further constraints, in the StructuredClassifiers package (as defined in 7.5.2).

The following subclauses describe the subpackages of PSCS_Syntax::Syntax shown in Figure 7.1. Each subclause provides rationale for the PSCS extensions related to the corresponding package, as well as an overview of the additional elements imported into these packages and how they relate to fUML syntax elements. Subclause 7.5 is the only subclause that also defines additional constraints introduced by this specification related to Composite Structures modeling.
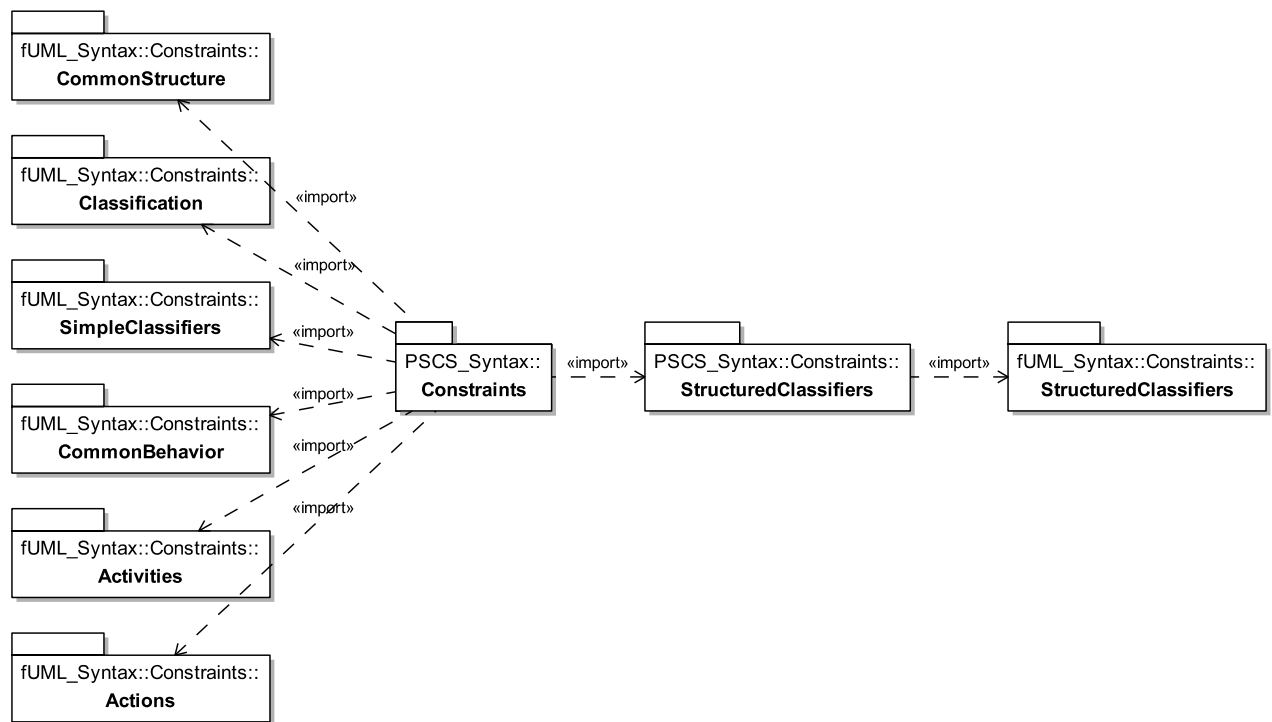
Figure 7.2 - PSCS Constraints Package

7.2 Common Structure

7.2.1 Overview

Dependencies are included from the CommonStructure package (see Figure 7.3) for two reasons. First, the metaclass Usage is needed to specify the required interfaces of a port. Second, the metaclass Realization is a generalization of InterfaceRealization defined in the Interfaces package. Note that the association A_mapping_abstraction is not included, since it is not relevant to Composite Structures modeling.
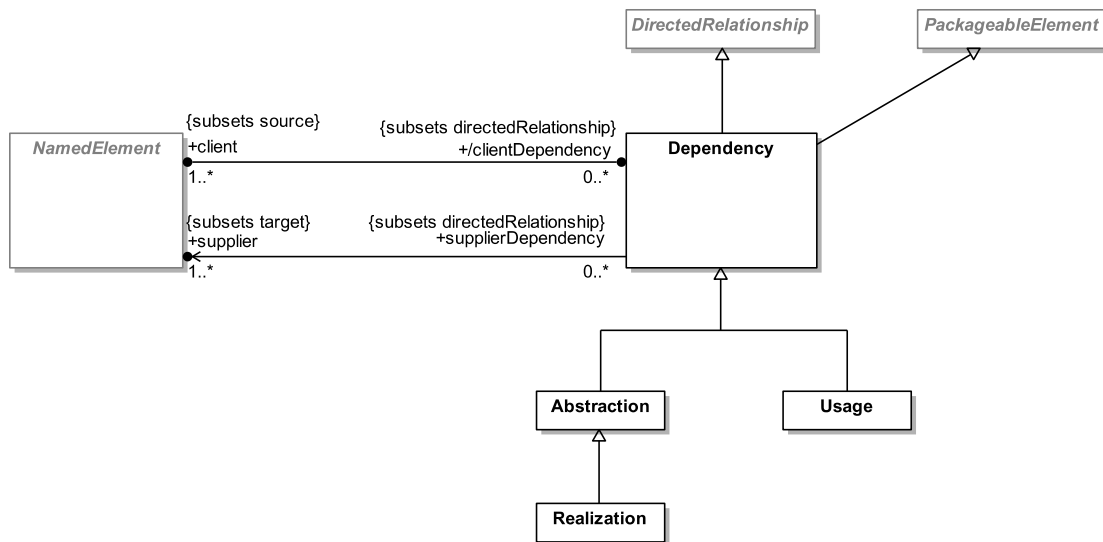
Figure 7.3 - Dependencies

7.2.2 Constraints

None.

7.3 Values

7.3.1 Overview

OpaqueExpression from the Values package is included in the Composite Structures syntax model, though it was explicitly excluded from fUML (see Figure 7.4). In this specification, OpaqueExpressions are typically used to associate default values with properties of composite structures, in the case where other kinds of ValueSpecification supported by fUML cannot be used easily. The main use case concerns situations where properties represent collections of elements.
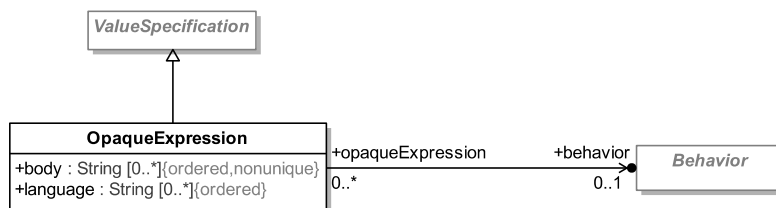
Figure 7.4 - Expressions

7.3.2 Constraints

None.

7.4 Simple Classifiers

7.4.1 Overview

Interface from the SimpleClassifiers package is included (see Figure 7.5) to enable the definition of required and/or provided features associated with ports. It also includes a mechanism for specifying that a given BehavioredClassifier realizes the features of an Interface. At run time, this information is used to determine if a given object (instance of a BehavioredClassifier) is a valid target for the invocation of a BehavioralFeature (where this feature is specified by an Interface).

In addition to Operations, such BehavioralFeatures may include Receptions, as a way of enabling a Classifier to specify that it provides or requires Receptions for Signals for some of its Ports.

Note. The Reception metaclass is already included from the SimpleClassifiers package in the fUML subset. However, the Interface metaclass and its association with Reception are explicitly excluded from fUML.
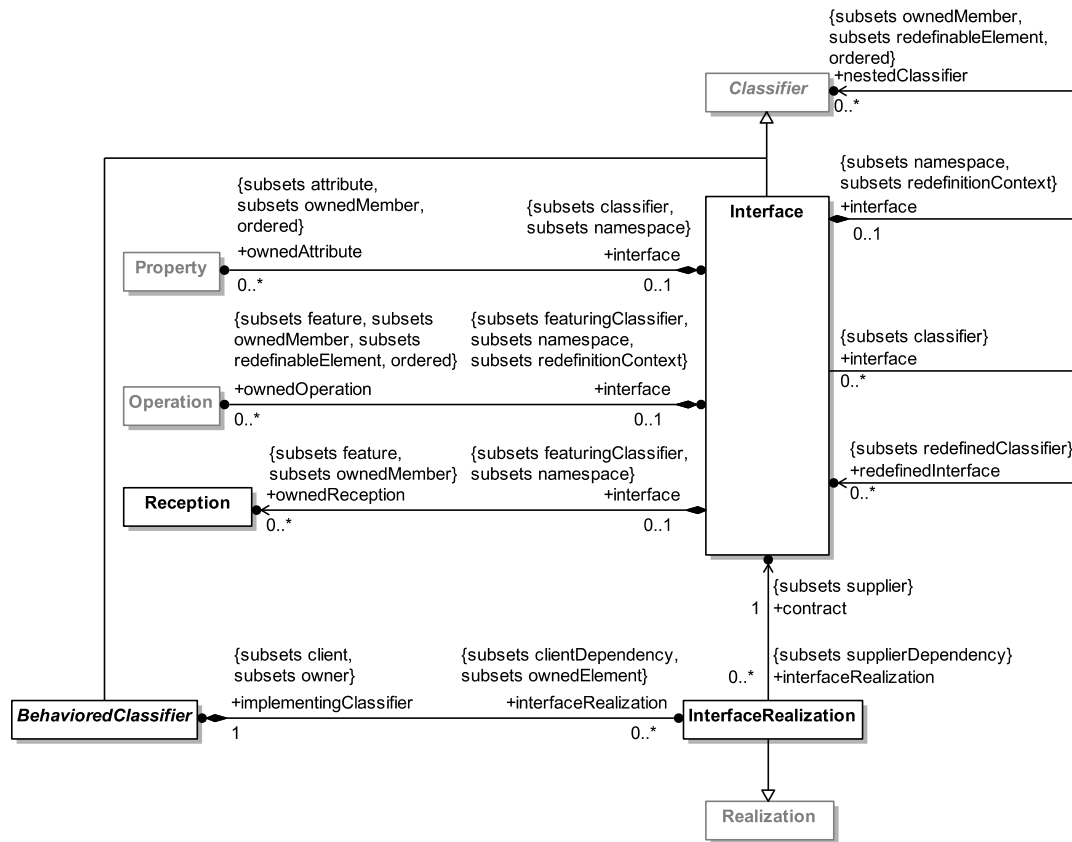
Figure 7.5 - Interfaces

7.4.2 Constraints

None.

7.5 Structured Classifiers

7.5.1 Overview

The ability of a class to be structured is a key aspect of UML Composite Structures. The internal structure of a Class is modeled as a network of parts, linked through Connectors (see Figure 7.6). The topology of parts and Connectors places constraints on the runtime structure of a Class instance. Classes may also have Ports (see Figure 7.7).

Note. The metaclasses StructuredClassifier, EncapsulatedClassifier and Class are already included in the fUML abstract syntax subset. However, the metaclass ConnectableElement, Connector, ConnectorEnd and Port, along with their related meta-associations, are not.
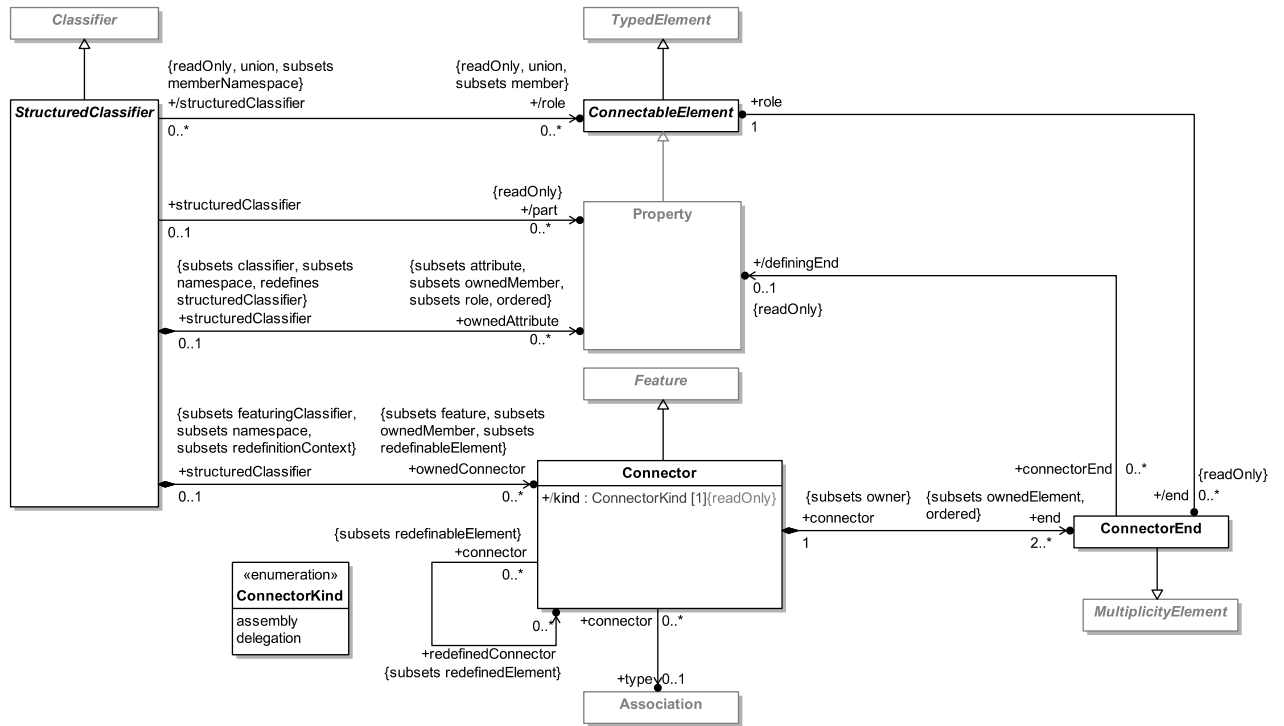
Figure 7.6 - Structured Classifiers
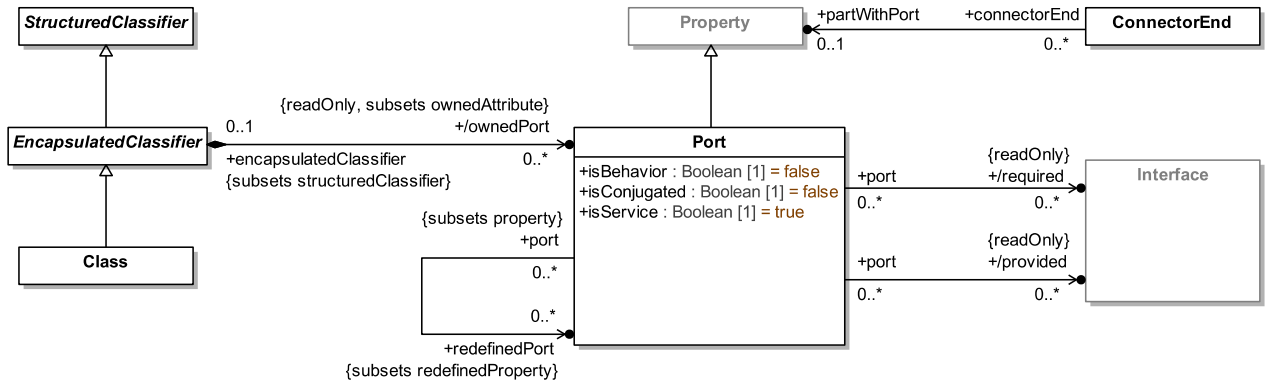
Figure 7.7 - Encapsulated Classifiers

7.5.2 Constraints

7.5.2.1 Connector

pscs_connector_is_typed
type <> null

7.5.2.2 Port

pscs_behavior_port_belongs_to_an_active_class
isBehavior implies
    (encapsulatedClassifier<>null and encapsulatedClassifier.oclIsKindOf(Class) and encapsulatedClassifier.isActive)

7.6 Common Behavior

7.6.1 Overview

The fUML subset already includes the metaclass Trigger, but it is extended here so that a list of Ports can be associated with a Trigger (see Figure 7.8).
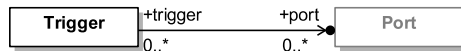
Figure 7.8 - Triggers

7.6.2 Constraints

None.

7.7 Actions

7.7.1 Overview

The fUML subset already includes the metaclass InvocationAction, but it is extended here to account for Ports of an EncapsulatedClassifier (see Figure 7.9). In addition to 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).
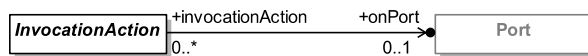
Figure 7.9 - Invocation Actions

7.7.2 Constraints

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.

8.1.1 Structural Semantics

In this specification, structural semantics concerns the run time manifestation of (structured/encapsulated) classes, which implies defining how ports, parts and connectors are represented at run time (see fUML Terms and Definitions for a more general definition).

8.1.2 Behavioral Semantics

In this specification, behavioral semantics concerns the interpretation of Activities making statements (through appropriate activity nodes and edges) related to the life cycle of objects (construction, destruction, observation and modification) and to communications between those objects (see fUML Terms and Definitions for a more general definition).

8.1.3 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 of 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. 8.2.2.1 of "Semantics of a Foundational Subset for Executable UML Models", subclause on Configuring the Execution Environment at a Locus).

This specification introduces three new semantic strategies. CS_StructuralFeatureOfInterfaceAccessStrategy (8.4.2.10) deals with reading and writing of structural features of an Interface, specifying how these features are actually realized by a given behaviored classifier. CS_RequestPropagationStrategy (8.4.2.9) deals with propagation of requests in the case where multiple possible propagation paths exist (e.g., multiple interaction points, multiple links). CS_ConstructStrategy (8.6.2.6) deals with instantiation of composite structures, by constructing topologies of objects, interaction points and links, as well as dealing with default values of corresponding properties. In addition, this specification introduces CS_DispatchOperationOfInterfaceStrategy (8.4.2.2), a new default realization of fUML DispatchStrategy. This realization deals with dispatching of operations of an Interface, specifying how these operations are actually realized by a behaviored classifier.

8.1.4 Instantiation of Composite Structures

The UML specification says: *"Links corresponding to Connectors may be created upon the creation of the instance of the containing StructuredClassifier"* (subclause 11.2.3.3) 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"* (subclause 11.2.3.4). 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"* (subclause 16.4.3.1). These semantics are formalized in the fUML execution model.

In order to comply with fUML while taking into account 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), this specification overrides semantics of CallOperationAction in the case where the standard stereotype Create is applied on the called Operation, and this operation has no associated method. In this case, a construction strategy is applied (see CS_ConstructStrategy in 8.6.2.6). Since Profiles and Stereotypes were not included in fUML, this specification provides some extensions, described in the following paragraph.

8.1.5 Dealing with Profiles and Stereotypes

The UML abstract syntax for profiles and stereotypes is not part of the fUML subset. However, for the purposes of fUML execution semantics, a UML profile can be interpreted exactly as its equivalent CMOF model, which can be represented in fUML (with the same considerations used when representing the rest of the UML abstract syntax in fUML). The application of a stereotype can then be treated as an object of the equivalent CMOF class for the stereotype tied to the appropriate UML metaobject by a link of the equivalent CMOF association for the extension.

Since the bUML subset used for the execution model does not include associations, when the UML abstract syntax is represented in bUML, associations are essentially ignored and all navigation is via the meta-class-owned end properties defined by the associations. Since the equivalent CMOF association for an extension is navigable only from the equivalent CMOF class for the stereotype to the extended UML metaclass, the usual UML abstract syntax conventions imply that the CMOF class for the stereotype owns the "base_metaclassName" property, but that there is not a corresponding "extension_stereotypeName" property added to the UML metaclass. This means that, in order to determine whether a metaobject has a specific stereotype applied, one needs to search the extent of the equivalent CMOF class for the stereotype to see if there is one that extends the given metaobject. This can be done using bUML code of the following form:

```
ExtensionalValueList extent = locus.getExtent(stereotypeClass);
ExtensionalValue extensionObject = null;
int i = 1;
while (i <= extent.size() && extensionObject == null) {
      ExtensionalValue object = extent.getValue(i - 1);
      if(object.getFeatureValue(baseEnd).values.getValue(0).equals(baseObject)) {
            extensionObject = object;
      }
      i = i + 1;
}
```

The result of the above code is that extensionObject contains the equivalent CMOF object for the stereotype applied to the model element represented by baseObject or, if the stereotype is not applied, it is null.

Note that doing the above requires that the equivalent CMOF class for the stereotype is actually instantiated at the execution locus, in order for its extent to be looked up. This means that it would be possible for a stereotype to be applied to a model at one execution, but not applied to that same model at a different execution locus. No fUML semantics have previously put any such requirement on the instantiation of abstract syntax metaclasses.

It also requires that the specific metaobject representing a stereotypeClass be known. The root package of the equivalent CMOF model of a profile has to be registered, rather than the individual stereotypes within the profile, so that a stereotype could be looked up by its qualified name, to avoid possible name conflicts. In this specification, this mechanism is added to the execution model by the specialization of ExecutionFactory (see 8.7.2.1). Then the equivalent class for a stereotype could be looked up using a call such as the following:

```
Class_ createStereotypeClass = locus.factory.getStereotypeClass("StandardProfile", "Create");
```

8.1.6 Semantics Packages

The following subclauses provide details about how the fUML execution model is extended to address semantics of UML Composite Structures. Figure 8.1 shows the relationships of the PSCS semantics packages with each other, the corresponding PSCS syntax packages and the fUML execution model packages being extended. The packaging of the PSCS semantic model parallels that of the PSCS abstract syntax (see Clause 7), except that there are no semantics packages for CommonStructure and SimpleClassifiers. PSCS includes only the subset of CommonStructure and SimpleClassifiers syntax necessary for Interfaces and InterfaceRealization (see 7.4 and 7.5), which do not have

independent semantics, but which are important in the semantic model of Ports on StructuredClassifiers. The PSCS semantic model also includes an additional Loci package, which includes extensions to fUML Locus, ExecutionFactory and Executor classes, in order to account for new semantic visitors introduced by this specification.
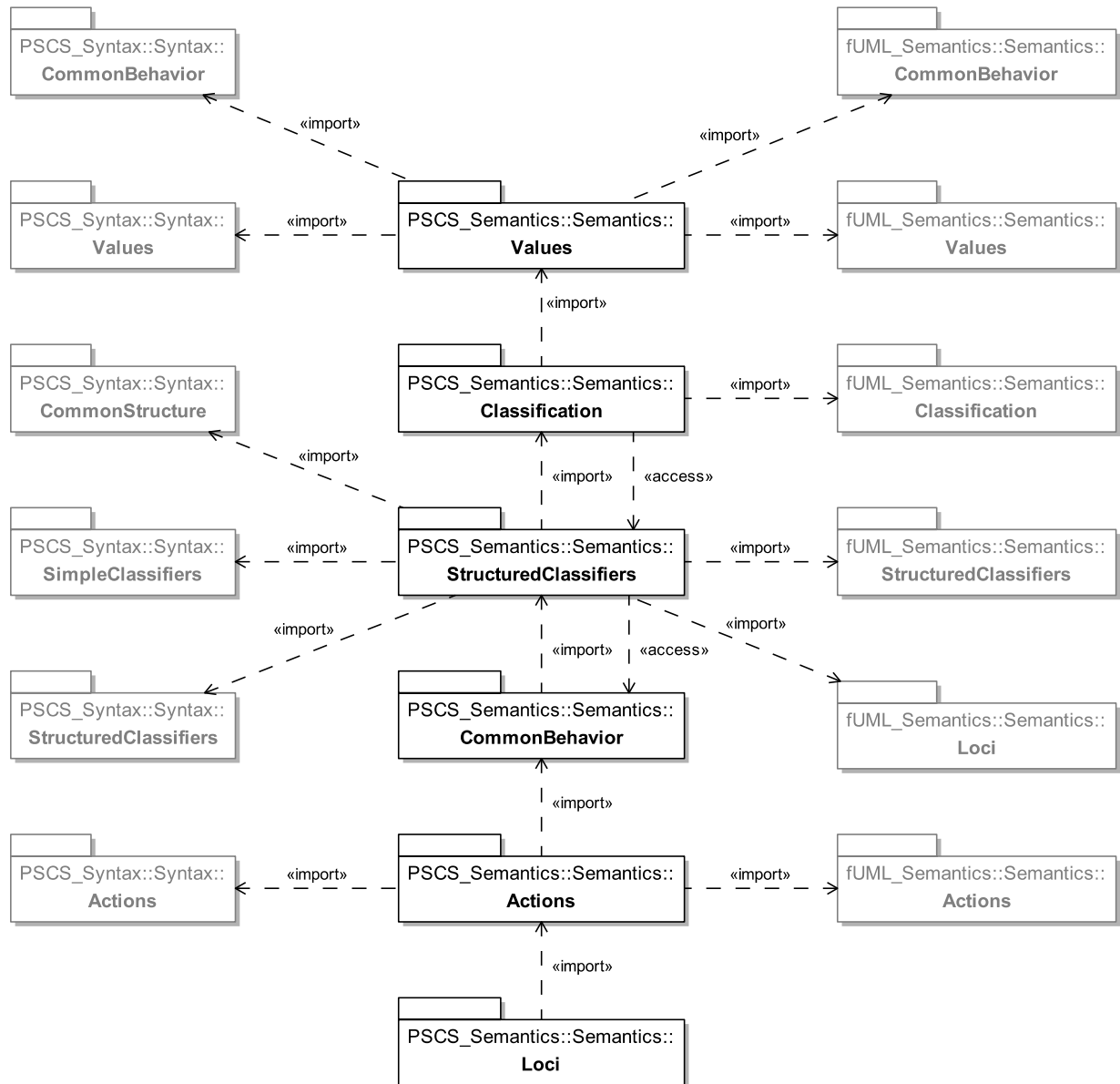
Figure 8.1 - PSCS Semantics Package

8.2 Values

8.2.1 Overview

The Values package contains a subclass of the fUML Evaluation class that defines the semantics for the evaluation of OpaqueExpressions (see Figure 8.2).

```
fUML_Semantics::Semantics::Values::
            Evaluation
```

```
CS_OpaqueExpressionEvaluation
+evaluate() : Value [0..1]
+executeExpressionBehavior() : Value [0..*]
```
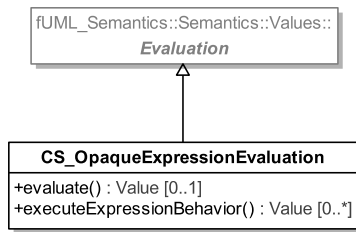
Figure 8.2 - Evaluations

8.2.2 Class Descriptions

8.2.2.1 CS_OpaqueExpressionEvaluation

CS_OpaqueExpressionEvaluation defines semantics for the evaluation of OpaqueExpressions. The evaluation consists in executing the Behavior associated with the evaluated OpaqueExpression.

The fUML semantics for the evaluation of ValueSpecification is based on the assumption that at most a single Value is returned. In CS_OpaqueExpressionEvaluation, this assumption is reflected in operation evaluate (described below). There are however cases where one would expect that the evaluation of ValueSpecification returns a collection of Values. A typical use case concerns a default value for a property with a lower bound greater than 1.

To support this use case, CS_OpaqueExpressionEvaluation introduces the operation executeExpressionBehavior, which returns a collection of values. This operation is used by CS_DefaultConstructStrategy, when instantiating objects corresponding to parts with default values specified by OpaqueExpressions. Other construction strategies might consider using this operation as well.

Generalizations

- Evaluation (from fUML_Semantics::Semantics::Values)

Attributes

- None

Associations

- None

Operations

```
[1] public evaluate() : Value[0..1]
        // Execute the behavior associated with the context OpaqueExpression, if any.
        // If multiple return values are computed, then return the first one.
        // If no values are computed, return null
        ValueList evaluation = this.executeExpressionBehavior() ;
        if (evaluation.size() > 0) {
                return evaluation.get(0) ;
        }
        else {
                return null ;
        }

[2] public executeExpressionBehavior() : Value[*]
        // If a behavior is associated with the context OpaqueExpression,
        // then execute this behavior, and return computed values.
        // Otherwise, return an empty list of values.
        ValueList evaluation = new ValueList() ;
        OpaqueExpression expression = (OpaqueExpression)this.specification ;
        Behavior behavior = expression.behavior ;
        if (behavior != null) {
                ParameterValueList inputs = new ParameterValueList() ;
                ParameterValueList results = this.locus.executor.execute(behavior, null, inputs) ;
```

```
                for (int i = 0 ; i < results.size() ; i++) { // results.size should be 1
                        ParameterValue parameterValue = results.get(i) ;
                        ValueList values = parameterValue.values ;
                        for (int j = 0 ; j < values.size() ; j++) {
                                evaluation.add(values.get(j)) ;
                        }
                }
        }
        return evaluation ;
```

8.3 Classification

8.3.1 Overview

The Classification package introduces an extension to the fUML InstanceValueEvaluation class (see Figure 8.3).
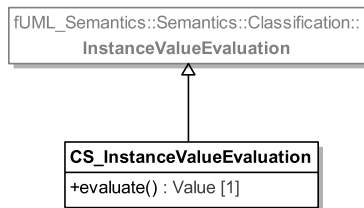
Figure 8.3 - Instance Values

8.3.2 Class Descriptions

8.3.2.1 CS_InstanceValueEvaluation

FUML semantics are extended so that, if the evaluated instance specification is for an object which is not typed by a Behavior, a CS_Reference (to a CS_Object) is returned instead of Reference (to a fUML Object).

Generalizations

- InstanceValueEvaluation (from fUML_Semantics::Semantics::Classification)

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 Behavior, 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 Structured Classifiers

8.4.1 Overview

The StructuredClassifiers package introduces extensions to fUML in order to support the runtime manifestation of parts, ports and connectors (see Figure 8.4) and extensions to semantic strategies (see Figure 8.5).
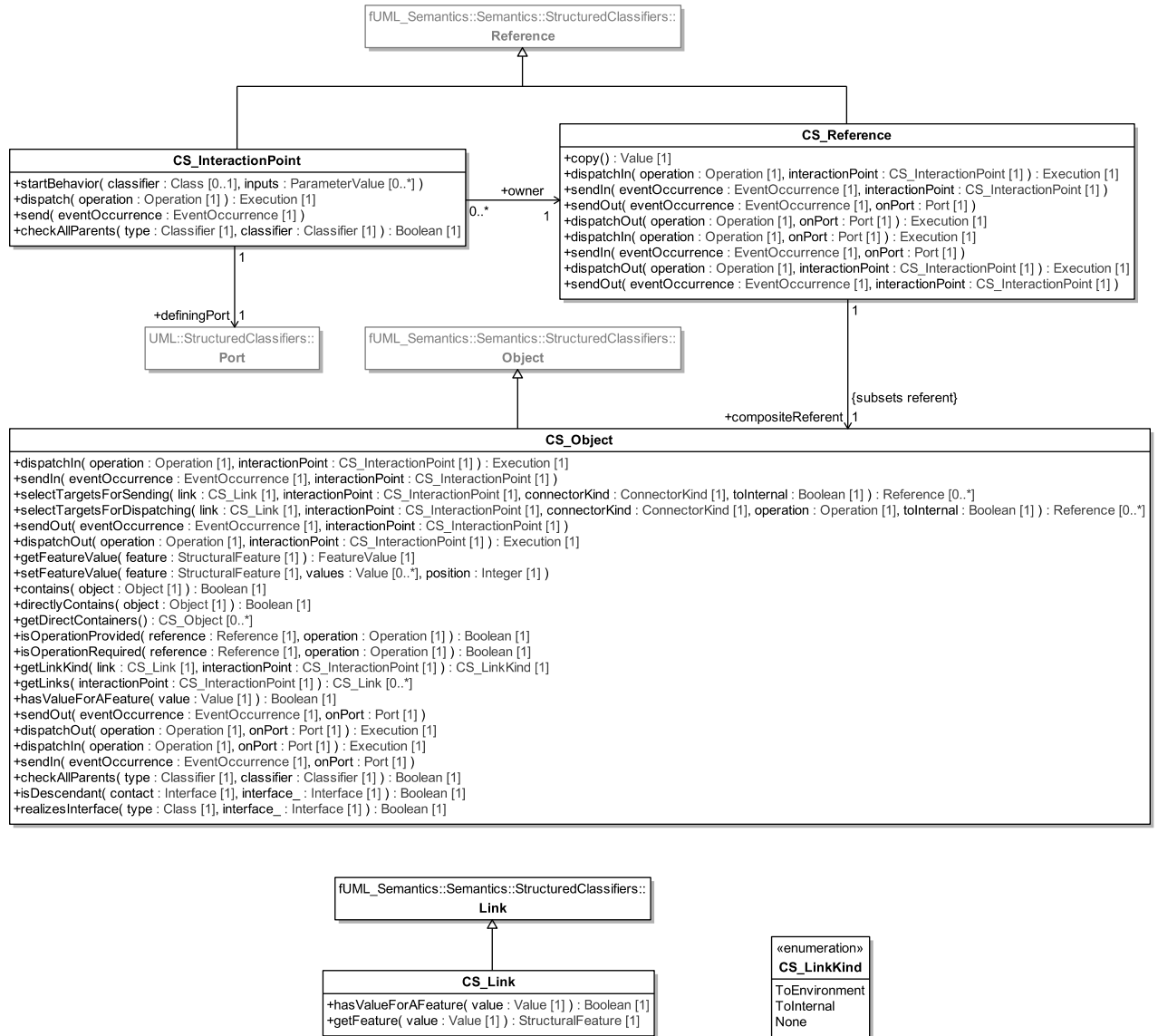
Figure 8.4 - Extensional Values
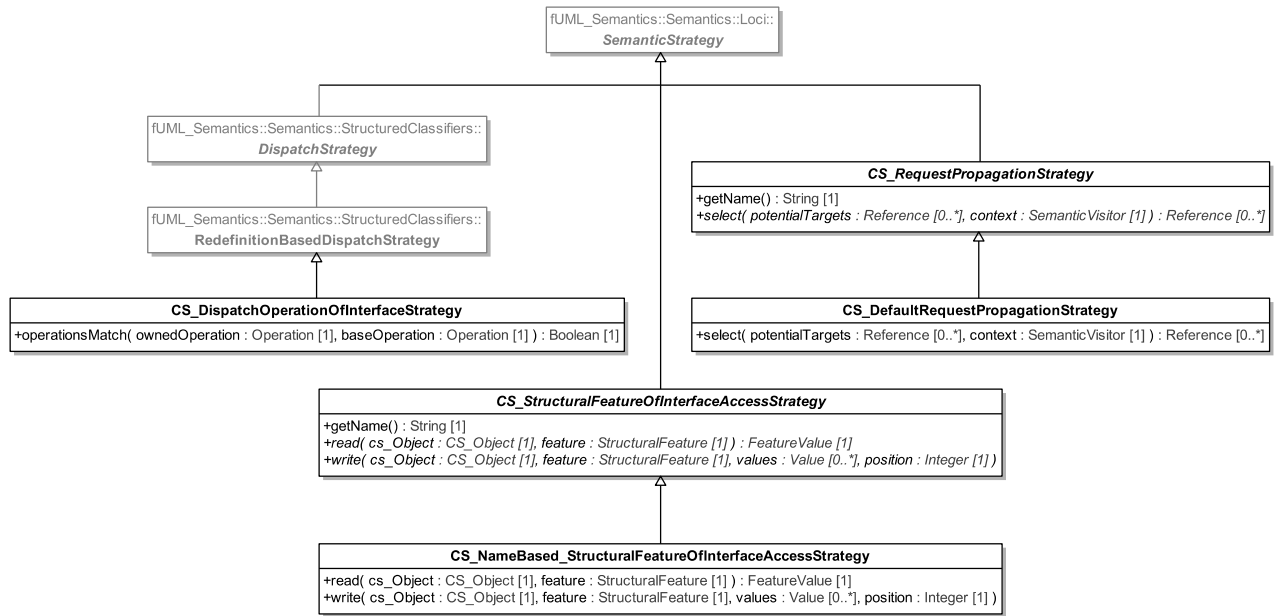
Figure 8.5 - Strategies

8.4.2 Class Descriptions

8.4.2.1 CS_DefaultRequestPropagationStrategy

CS_DefaultRequestPropagationStrategy is the default realization provided by this specification for semantic strategy CS_RequestPropagationStrategy. If the request concerns the emission of a Signal and there are multiple possible targets, the signal is broadcasted to all the targets. If the request concerns an Operation call and there are multiple possible targets, the call is propagated to the first target.

Generalizations

- CS_RequestPropagationStrategy (from PSCS_Semantics::Semantics::StructuredClassifiers)

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;
```

8.4.2.2 CS_DispatchOperationOfInterfaceStrategy

CS_DispatchOperationOfInterfaceStrategy extends RedefinitionBasedDispatchStrategy, which is the default semantic strategy provided by fUML for operation call dispatching. The extension consists in taking into account the introduction of Interfaces (cf. 7.4.1). If the called Operation belongs to an Interface (cf. 8.6.2.4, CS_CallOperationActionActivation), this strategy is used to determine if an Operation of the target Object (i.e., an Operation that belongs to a Class typing this Object) matches the Operation of the Interface. It matches if it has the same name and signature.

Generalizations

- RedefinitionBasedDispatchStrategy (from fUML_Semantics::Semantics::StructuredClassifiers)

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 ;
```

8.4.2.3 CS_InteractionPoint

CS_InteractionPoint provides support for the runtime manifestation of Ports. It is a Reference to an Object which is a value for a Port, in the context of a CS_Object. Figure 8.6 illustrates the relationships between CS_Object, CS_Reference, FeatureValue and CS_InteractionPoint, when instantiating a composite structure with a Port typed by a Class.
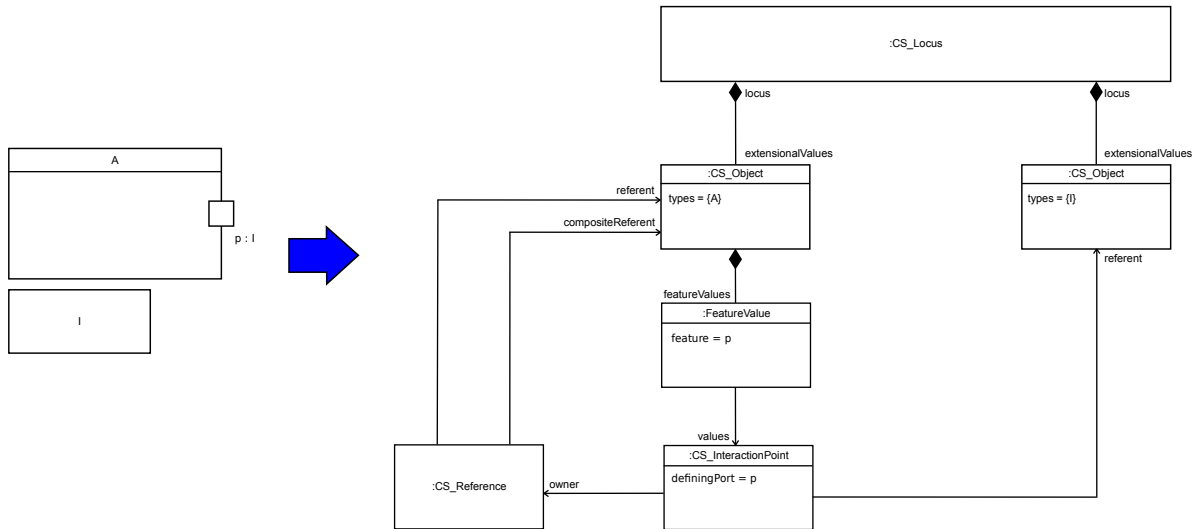
Figure 8.6 - Runtime relationships between CS_Object, CS_Reference, FeatureValue and CS_InteractionPoint resulting from the instantiation of a Composite Structure

Generalizations

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

Attributes

- None

Associations

- owner : CS_Reference[1], Represents the reference to the CS_Object owning this CS_InteractionPort.
- definingPort : Port[1], The Port for which this CS_InteractionPoint is a runtime manifestation.

Operations

```
[1] public checkAllParents ( type : Classifier, classifier : Classifier ) : Boolean
// Delegates the type checking to the reference
return this.referent.checkAllParents(type, classifier) ;

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

[3] public send(eventOccurrence:EventOccurrence)
// An event occurrence that passes through a CS_InteractionPoint is
//(if necessary) wrapped in a CS_EventOccurrence. This event occurrence
// is then sent to the owning object
CS_EventOccurrence wrappingEventOccurrence = null;
if(eventOccurrence instanceof CS_EventOccurrence){
       wrappingEventOccurrence = (CS_EventOccurrence) eventOccurrence;
}else{
       wrappingEventOccurrence = new CS_EventOccurrence();
       wrappingEventOccurrence.wrappedEventOccurrence = eventOccurrence;
}
wrappingEventOccurrence.interactionPoint = this;
this.owner.sendIn(wrappingEventOccurrence, this) ;

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

8.4.2.4 CS_Link

CS_Link extends fUML Link with helper Operations, used by CS_Object to determine links through which requests can be propagated. As in fUML, a CS_Link is the runtime manifestation of an Association instance. In this specification, it is also used to represent connector instances, though there is no explicit relationship between CS_Link and Connector. Subclause 8.6.2.9 (CS_DefaultConstructStrategy) specifies how CS_Links are instantiated from Connectors.

A CS_Link can also be created using a CreateLinkAction. Note that this kind of Action relies 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, and has no consideration for Connectors (this is the reason why there is no explicit relationships between CS_Link and Connector). The manual instantiation of a Connector is thereby specified with a CreateLinkAction, where elements to be linked act as values for roles identified by the Connector. If the Connector is typed by an Association, the LinkEndData of the CreateLinkAction relies on this Association. In the case where the Connector is not typed, a generic Association (with untyped ends) can be used to specify the LinkEndData of the CreateLinkAction. Such a generic Association is specified in informative Annex C.

Generalizations

- Link (from fUML_Semantics::Semantics::StructuredClassifiers)

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 ;

[2] public getFeature(value:Value) : StructuralFeature
  FeatureValueList allFeatureValues = this.getFeatureValues() ;
  Integer i = 1 ;
  StructuralFeature feature = null ;
  while (i <= allFeatureValues.size() && feature == null) {
    FeatureValue featureValue = allFeatureValues.get(i-1);
    if (!featureValue.values.isEmpty() && featureValue.values.get(0).equals(value)) {
      feature = featureValue.feature ;
    }
    i = i + 1 ;
  }
  return feature ;
```

8.4.2.5 CS_LinkKind

CS_LinkKind is an enumeration that characterizes a CS_Link, in the context of a CS_Object, with respect to a CS_InteractionPoint, if the interaction point belongs to this object, and if it is used as an end of the link. The link kind of a link is determined as follows:

- 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 this object.

- If all the feature values of the link refer to values which are themselves values for features of this object, 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**).

These rules are formalized in 8.4.2.7, CS_Object, operation getLinkKind.

Generalizations

- None

Enumeration literals

- ToEnvironment
- ToInternal
- None

8.4.2.6   CS_NameBased_StructuralFeatureOfInterfaceAccessStrategy

CS_NameBased_StructuralFeatureOfInterfaceAccessStrategy is the default realization provided by this specification for the semantic strategy CS_StructuralFeatureOfInterfaceAccessStrategy. This default realization requires that, for a given Class to actually realize the StructuralFeatures of an Interface, it must define or inherit StructuralFeatures that match the StructuralFeatures of the Interface. StructuralFeatures match if they have same name, compatible type, and same multiplicity. Note that UML does not impose that the realizing Classes have StructuralFeatures matching the structural features of the realized Interfaces. For example, alternative strategies may require that the realizing Classes provide "get" and "set" Operations matching the StructuralFeatures of the realized Interfaces.

Generalizations

- CS_StructuralFeatureOfInterfaceAccessStrategy (from PSCS_Semantics::Semantics::StructuredClassifiers)

Attributes

- None

Associations

- None

Operations

```
[1] public read(cs_Object:CS_Object, feature:StructuralFeature) : FeatureValue
        // returns 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) ;
        }
```

8.4.2.7   CS_Object

CS_Object extends fUML Object to specify the runtime manifestation of a composite structure. FUML Object owns a single operation for sending event occurrences (Operation send) and a single operation for dispatching an operation call (Operation dispatch). These are used to specify the semantics of SendSignalAction and CallOperationAction. In fUML, these actions always directly target an Object. With composite structures, these InvocationActions can also be made through ports, with invocations being propagated either to the environment or the internals of the composite structure instance. The rules for determining if an invocation is propagated inside or outside of a composite structure, through a port, are specified in 8.6.2.13, CS_SendSignalActionActivation, and 8.6.2.4, CS_CallOperationActionActivation.

To support propagation of invocations through ports, either inside or outside of the composite structure instance, CS_Object introduces variants of send and dispatch Operations of fUML Object: sendIn, sendOut, dispatchIn, dispatchOut, which are specified below. Propagation of invocations is made following existing CS_Link, which represent connector instances between CS_InteractionPoints and/or CS_References (which represent values for parts). Sending and dispatching inside of a CS_Object through a CS_InteractionPoint accounts for the fact that the defining port is behavior or not. Note that the PSCS semantics for dispatchIn through a behavior port uses the usual fUML dispatch mechanism to determine how an incoming operation call is handled. This means that an operation call through a behavior port can be handled either by a method defined for the operation on the owning behaviored classifier of the port or through a call event handled by the classifier behavior of the owning behaviored classifier, just as is the case for an operation invoked directly on an object, rather than through a port (using the default fUML dispatching strategy).

CS_Object also accounts for the realization of StructuralFeatures of Interfaces, by overriding Operations getFeatureValue and setFeatureValue. Their behavior depends on the CS_StructuralFeatureOfInterfaceAccessStrategy registered at the execution locus.

Generalizations

- Object (from fUML_Semantics::Semantics::StructuredClassifiers)

Attributes

- None

Associations

- None

Operations

```
[1] public checkAllParents ( type:Classifier, classifier:Classifier ) : Boolean
// If the given classifier is not an Interface, behaves like in fUML
// Otherwise, check if the given type (or one of its direct or indirect ancestors) has an
// InterfaceRealization relationships with the given classifier.
boolean matched = false;
if (!(classifier instanceof Interface)) {
```

```
            matched = super.checkAllParents(type, classifier);
    } else if (!(type instanceof Class)) {
            matched = false;
    } else if (this.realizesInterface((Class) type, (Interface) classifier)) {
            matched = true;
    } else {
            ClassifierListdirectParents = type.general;
            int i = 1;
            while (!matched & i <= directParents.size()) {
                    Classifier directParent = directParents.getValue(i - 1);
                    matched = this.checkAllParents(directParent, classifier);
                    i = i + 1;
            }
    }
    return matched;

[2] 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;

[3] 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;

[4] public dispatchIn(operation:Operation, interactionPoint:CS_InteractionPoint) : Execution
  // If the interaction point refers to a behavior port,
  // the operation call is dispatched to the object owning the behavior port.
  // This may result in the method being handled by the method defined for the operation
  // at the object or through a call event handled by the classifier behavior of the owning
  // object. The latter case only occurs if the dispatched operation has no implementation.
  // 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) {
    execution = this.dispatch(operation);
    if (execution instanceof CS_CallEventExecution){
      ((CS_CallEventExecution)execution).interactionPoint = interactionPoint;
    }
  }
```

```
      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 ;

[5] 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) ;

[6] 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 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 ;

[7] 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 ;

[8] 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 ;

[9] 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);
    }

[10] 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 ;

[11] 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 ;

[12] 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 ;

[13] public isDescendant ( contract:Interface, interface:Interface ) : Boolean
  // Checks if the given contract is a descendant of the given interface
  boolean matched = false;
  ClassifierList descendants = contract.general;
  int i = 1;
  while (i <= descendants.size() & !matched) {
    if (descendants.getValue(i - 1) instanceof Interface) {
      Interface descendant = (Interface) descendants.getValue(i - 1);
      if (descendant == interface_) {
        matched = true;
      } else {
        matched = this.isDescendant(descendant, interface_);
      }
    }
    i = i + 1;
  }
  return matched;

[14] 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 ;

[15] 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 ;

[16] public realizesInterface (type : Class, interface_ :Interface) : Boolean
// Checks if the given type has an InterfaceRealization relationship
// with the given interface or a descendant of the interface
InterfaceRealizationList realizations = type.interfaceRealization;
boolean realized = false;
int i = 1;
while (i <= realizations.size() & !realized) {
        InterfaceRealization realization = realizations.getValue(i - 1);
        Interface contract = realization.contract;
        if (contract == interface_) {
                realized = true;
        } else if (this.isDescendant(contract, interface_)) {
                realized = true;
        }
        i = i + 1;
}
return realized ;

[17] public selectTargetsForDispatching(link:CS_Link, interactionPoint:CS_InteractionPoint,
connectorKind:ConnectorKind, operation:Operation, toInternal: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 toInternal 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 toInternal 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 (toInternal && connectorKind == ConnectorKind.delegation) {
  if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToInternal) {
    Integer i = 1 ;
    while(i <= link.getFeatureValues().size()) {
      ValueList values = link.getFeatureValues().get(i-1).values ;
      if (!values.isEmpty() && values.get(0) instanceof Reference) {
        Reference cddTarget = (Reference)values.get(0) ;
        if (cddTarget != interactionPoint && this.isOperationProvided(cddTarget, operation)) {
          potentialTargets.add(cddTarget) ;
        }
      }
      i = i + 1 ;
    }
  }
}
else { // to environment
  if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToEnvironment) {
    Integer i = 1 ;
    while(i <= link.getFeatureValues().size()) {
      ValueList values = link.getFeatureValues().get(i-1).values ;
      if (!values.isEmpty() && values.get(0) instanceof Reference) {
        Reference cddTarget = (Reference)values.get(0) ;
        if (connectorKind == ConnectorKind.assembly) {
          if (! (cddTarget instanceof CS_InteractionPoint)) { // This is an assembly link
            if (this.isOperationProvided(cddTarget, operation)) {
              potentialTargets.add(cddTarget) ;
            }
          }
          else {
            // This is an assembly if the interaction point is not a feature value
            // for a container of this CS_Object
            CS_ObjectList directContainers = this.getDirectContainers() ;
            boolean isAssembly = true ;
            Integer j = 1 ;
            if (! this.hasValueForAFeature(cddTarget)) {
              while (isAssembly && j <= directContainers.size()) {
                CS_Object container = directContainers.get(j - 1) ;
                if (container.hasValueForAFeature(cddTarget)) {
                  isAssembly = false ;
                }
                j++ ;
              }
            }
            else {
              isAssembly = false ;
            }
            if (isAssembly) {
              if (this.isOperationProvided(cddTarget, operation)) {
                potentialTargets.add(cddTarget) ;
              }
            }
          }
        }
        else { // delegation
          // This is a delegation if the target is an interaction point
          // and if this interaction is a feature value for a container of this CS_Object
          if (cddTarget instanceof CS_InteractionPoint) {
            CS_ObjectList directContainers = this.getDirectContainers() ;
            boolean isDelegation = false ;
            Integer j = 1 ;
            while (!isDelegation && j <= directContainers.size()) {
```

```
                  CS_Object container = directContainers.get(j - 1) ;
                  if (container.hasValueForAFeature(cddTarget)) {
                    isDelegation = true ;
                  }
                  j++ ;
                }
                if (isDelegation) {
                  if (this.isOperationRequired(cddTarget, operation)) {
                    potentialTargets.add(cddTarget) ;
                  }
                }
              }
            }
          }
          i = i + 1 ;
        }
      }
    }
    return potentialTargets ;

[18] public selectTargetsForSending( link : CS_Link, interactionPoint : CS_InteractionPoint,
connectorKind : ConnectorKind , toInternal : 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 toInternal is true, connectorKind must be Delegation,
  // the given link has to target the internals of this CS_Object
  // - if toInternal is false, the given link has to target the environment of this CS_Object.
  ReferenceList potentialTargets = new ReferenceList() ;
  if (toInternal && connectorKind == ConnectorKind.delegation) {
    if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToInternal) {
      Integer i = 1 ;
      while(i <= link.getFeatureValues().size()) {
        ValueList values = link.getFeatureValues().get(i-1).values ;
        if (!values.isEmpty()) {
          Integer j = 1 ;
          while (j <= values.size()) {
            Reference cddTarget = (Reference)values.get(j-1) ;
            if (!cddTarget.equals(interactionPoint)) {
              potentialTargets.add(cddTarget) ;
            }
            j = j + 1 ;
          }
        }
        i = i + 1 ;
      }
    }
  }
  else { // to Environment
    if (this.getLinkKind(link, interactionPoint) == CS_LinkKind.ToEnvironment) {
      Integer i = 1 ;
      while(i <= link.getFeatureValues().size()) {
        ValueList values = link.getFeatureValues().get(i-1).values ;
        if (!values.isEmpty() && values.get(0) instanceof Reference) {
          Reference cddTarget = (Reference)values.get(0) ;
          if (connectorKind == ConnectorKind.assembly) {
            if (! (cddTarget instanceof CS_InteractionPoint)) { // This is an assembly link
              potentialTargets.add(cddTarget) ;
            }
            else {
              // This is an assembly if the interaction point is not a feature value
              // for a container of this CS_Object
              CS_ObjectList directContainers = this.getDirectContainers() ;
              boolean isAssembly = true ;
              Integer j = 1 ;
              if (! this.hasValueForAFeature(cddTarget)) {
                while (isAssembly && j <= directContainers.size()) {
                  CS_Object container = directContainers.get(j - 1) ;
                  if (container.hasValueForAFeature(cddTarget)) {
                    isAssembly = false ;
                  }
```

```
                j++ ;
              }
            }
            else {
              isAssembly = false ;
            }
            if (isAssembly) {
              potentialTargets.add(cddTarget) ;
            }
          }
        }
        else { // delegation
          // This is a delegation if the target is an interaction point
          // and if this interaction is a feature value for a container of this CS_Object
          if (cddTarget instanceof CS_InteractionPoint) {
            CS_ObjectList directContainers = this.getDirectContainers() ;
            boolean isDelegation = false ;
            Integer j = 1 ;
            while (!isDelegation && j <= directContainers.size()) {
              CS_Object container = directContainers.get(j - 1) ;
              if (container.hasValueForAFeature(cddTarget)) {
                isDelegation = true ;
              }
              j++ ;
            }
            if (isDelegation) {
              potentialTargets.add(cddTarget) ;
            }
          }
        }
      }
      i = i + 1 ;
    }
  }
}
return potentialTargets ;

[19] public sendIn ( eventOccurrence : EventOccurrence, interactionPoint : CS_InteractionPoint)
  // 1] If the interaction is a behavior port,
  //    then sends the event occurrence to the target object using operation send
  // 2] If this is not a behavior port,
  //    select appropriate delegation targets from interactionPoint
  //    and propagates the event occurrence to these targets
  if (interactionPoint.definingPort.isBehavior) {

    this.send(eventOccurrence) ;
  }
  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,
                                    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 event occurrence is lost
    // Otherwise, do the following concurrently
    for (int i = 0 ; i < potentialTargets.size() ; i++) {
      Reference target = potentialTargets.getValue(i) ;
      target.send(eventOccurrence) ;
```

```
        }
    }

[20] public sendIn (eventOccurrence:EventOccurrence, onPort:Port)
        // Select a Reference value playing onPort,
        // and send the event occurrence 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(eventOccurrence) ;
        }

[21] public sendOut(eventOccurrence:EventOccurrence, interactionPoint:CS_InteractionPoint)
    // Select appropriate delegation links from interactionPoint,
    // and propagates the event occurrence 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,
                                            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,
                                            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(eventOccurrence) ;
       }
     }
     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(eventOccurrence, cddTarget) ;
       }
     }
   }

[22] public sendOut(eventOccurrence: EventOccurrence, onPort:Port)
   // Select a CS_InteractionPoint value playing onPort,
   // and send the event occurrence 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(eventOccurrence, target) ;
   }

[23] 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);
   }
```

8.4.2.8   CS_Reference

CS_Reference extends fUML Reference to account for new capabilities introduced by CS_Object (i.e., variants of send and dispatch operations introduced to deal with sending and dispatching inside or outside of a composite structure instance, through a port). CS_Reference is a reference to a CS_Object (property compositeReferent, which subsets inherited property Object::referent). All operations of CS_Reference are realized by delegation to its referent CS_Object.

Generalizations

- Reference (from fUML_Semantics::Semantics::StructuredClassifiers)

Attributes

- None

Associations

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

Operations

```
[1] public copy() : Value
// Create a new reference with the same referent and composite referent
// as this reference.
```

```
CS_Reference newValue = new CS_Reference();
newValue.referent = this.referent;
newValue.compositeReferent = this.compositeReferent;
return newValue;

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

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

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

[5] 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) ;

[6] public sendIn ( eventOccurrence:EventOccurrence, interactionPoint:CS_InteractionPoint)
      // delegates sending to composite referent
      this.compositeReferent.sendIn(eventOccurrence, interactionPoint);

[7] public sendIn ( eventOccurrence:EventOccurrence, onPort:Port)
      // delegates sending to composite referent
      this.compositeReferent.sendIn(eventOccurrence, onPort);

[8] public sendOut ( eventOccurrence:EventOccurrence, onPort:Port)
      // delegates sending to composite referent
      this.compositeReferent.sendOut(eventOccurrence, onPort);

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

8.4.2.9   CS_RequestPropagationStrategy

CS_RequestPropagationStrategy is a semantic strategy which deals with propagation of requests in the case where multiple possible propagation paths exist (e.g., multiple interaction points, multiple links. An example of such a situation is depicted in Figure 9.4). Its purpose is to select, among a list of potential targets, the actual targets to which the request will be propagated. It is used by the various send and dispatch operations provided by class CS_Object. In this specification, the default realization for this strategy is CS_DefaultRequestPropagationStrategy.

Generalizations

- SemanticStrategy (from UML_Semantics::Semantics::Loci)

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[*]
```

8.4.2.10 CS_StructuralFeatureOfInterfaceAccessStrategy

CS_StructuralFeatureOfInterfaceAccessStrategy is a new semantic strategy. It deals with reading and writing of structural features of an Interface, specifying how these features are actually realized by a given behaviored classifier. It is involved in the case where a WriteStructuralFeatureAction (or, more precisely, its concrete descendant metaclasses) or ReadStructuralFeatureValueAction concerns a StructuralFeature which belongs to an Interface. It is effectively used by CS_Object (cf. CS_Object, operations getFeatureValue and setFeatureValue). This specification provides a default realization for this strategy: CS_NameBased_StructuralFeatureOfInterfaceAccessStrategy.

Generalizations

- SemanticStrategy (from UML_Semantics::Semantics::Loci)

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 Common Behavior

8.5.1 Overview

The CommonBehavior package introduces specializations of the fUML concepts of event occurrence, representing the occurrence of an event received through a port, and of call event executions, corresponding to the dispatching of an operation on a behavior port (see Figure 8.7).
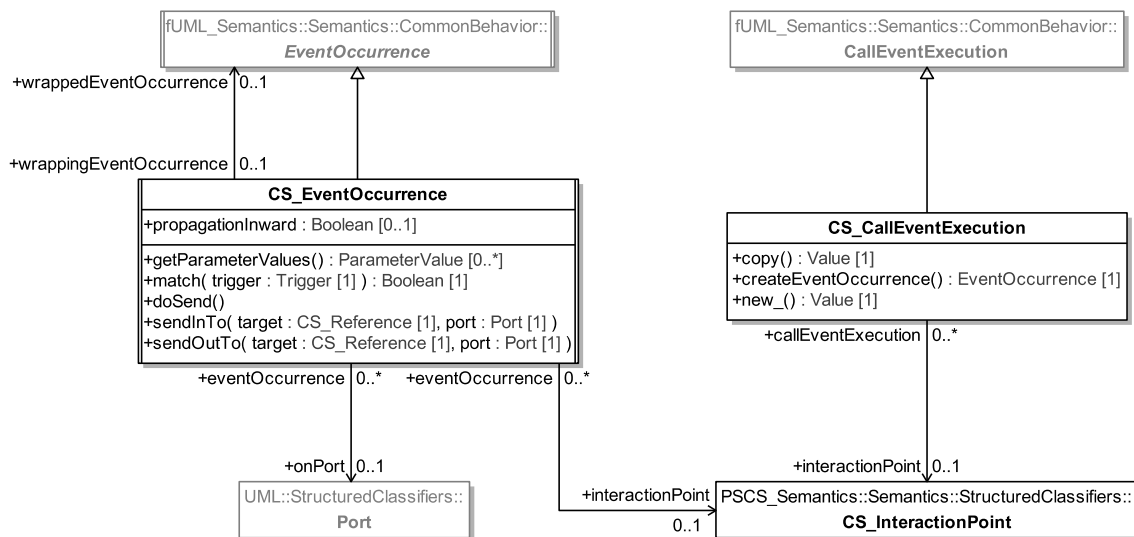
Figure 8.7 - Event Occurrences

8.5.2 Class Descriptions

8.5.2.1 CS_CallEventExecution

A CS_CallEventExecution is the Execution corresponding to the dispatching of an operation on a behavior port of an active class. When a CS_CallEventExecution is executed, it sends a CS_EventOccurence wrapping a CallEventOccurrence. This enables the possibility to designate the CS_InteractionPoint (i.e., the manifestation of the behavior port) from which the event occurrence has arrived.

Generalizations

- CallEventExecution (from fUML_Semantics::Semantics::CommonBehavior)

Attributes

- None

Associations

- interactionPoint : CS_InteractionPoint [0..1], The interaction point representing the behavior port manifestation to which the operation call was dispatched.

Operations

```
[1] public copy ( ) : Value
// Create a new call event execution that is a copy of this execution
// with no referenced interaction point.
CS_CallEventExecution copy = (CS_CallEventExecution) super.copy();
copy.interactionPoint = null;
return copy;

[2] public createEventOccurrence ( ) : EventOccurrence
// Wrap the created event occurrence within a CS_EventOccurrence which
// references the behavior port on which the call was dispatched.
CS_EventOccurrence wrappingEventOccurrence = new CS_EventOccurrence();
wrappingEventOccurrence.interactionPoint = this.interactionPoint;
wrappingEventOccurrence.wrappedEventOccurrence = super.createEventOccurrence();
return wrappingEventOccurrence;

[3] public new_ ( ) : Value
// Create a new call event execution.
return new CS_CallEventExecution();
```

8.5.2.2 CS_EventOccurence

A CS_EventOccurrence is a specialization of the concept of event occurrence provided by fUML (see subclause 8.8.2.6 of fUML). It represents a single occurrence of an event that was received by an object through a specific port.

Generalizations

- EventOccurrence (from fUML_Semantics::Semantics::CommonBehavior)

Attributes

- propagationInward : Boolean [0..1], The direction followed to propagate the event occurrence

Associations

- wrappedEventOccurrence : EventOccurrence [0..1], The event occurrence that was initially sent to the target object.
- interactionPoint : CS_InteractionPoint [0..1], The interaction point through which the event occurrence arrived at the target object.
- onPort : Port [0..1], The port specification through which the event occurrence shall be propagated.

Operations

```
[1] public doSend ( )
// If the specified target is a CS_Reference and the propagation must be done to the
// environment, then the operation sendOut(EventOccurrence, Port) is called and
// make the propagation to continue. If the specified target is a CS_Reference but
// the propagation must be done to the internals
// then the operation sendIn(EventOccurrence, Port) is called.
// In the case were the target is not a CS_EventOccurrence
// then send(EventOccurence)is called.
if(this.target instanceof CS_Reference){
        if(this.propagationInward == true){
                ((CS_Reference)this.target).sendIn(this, this.onPort);
        }else{
                        ((CS_Reference)this.target).sendOut(this, this.onPort);
        }
}else{
        super.doSend();
}

[2] public getParameterValues ( ) : ParameterValue [*]
// Return the parametric data of the referenced event occurrence
return this.wrappedEventOccurrence.getParameterValues();

[3] public match ( trigger:Trigger ) : Boolean
// If the trigger references ports then to match this latter it is required that
// 1] the interaction point is instance of port referenced by the trigger
// 2] the referenced event occurrence match operation returns true.
// If the trigger does not reference ports then the behavior is the same than for fUML.
boolean matches = false;
if(trigger.port.size() > 0){
        int i = 1;
        while(!matches & i <= trigger.port.size()){
                if(this.interactionPoint.definingPort==trigger.port.get(i-1)){
                        matches = true;
                }
                i++;
        }
        if(matches){
                matches = this.wrappedEventOccurrence.match(trigger);
        }
}else{
        matches = this.wrappedEventOccurrence.match(trigger);
}
return matches;

[4] public sendInTo ( target : CS_Reference, port: Port )
// Propagate the sending of the event occurrence to the internals
this.onPort = port;
this.propagationInward = true;
this.sendTo(target);

[5] public sendOutTo ( target : CS_Reference, port: Port )
// Propagate the sending of the event occurrence to the environment
this.onPort = port;
this.propagationInward = false;
this.sendTo(target);
```

8.6 Actions

8.6.1 Overview

The Actions package introduces extensions to various fUML action activation classes, to deal with actions that have additional semantics when executed in the context of composite structures. This includes extensions to invocation actions (see Figure 8.8), object and link actions (see Figure 8.9), structural feature actions (see Figure 8.10) and accept actions (see Figure 8.11).
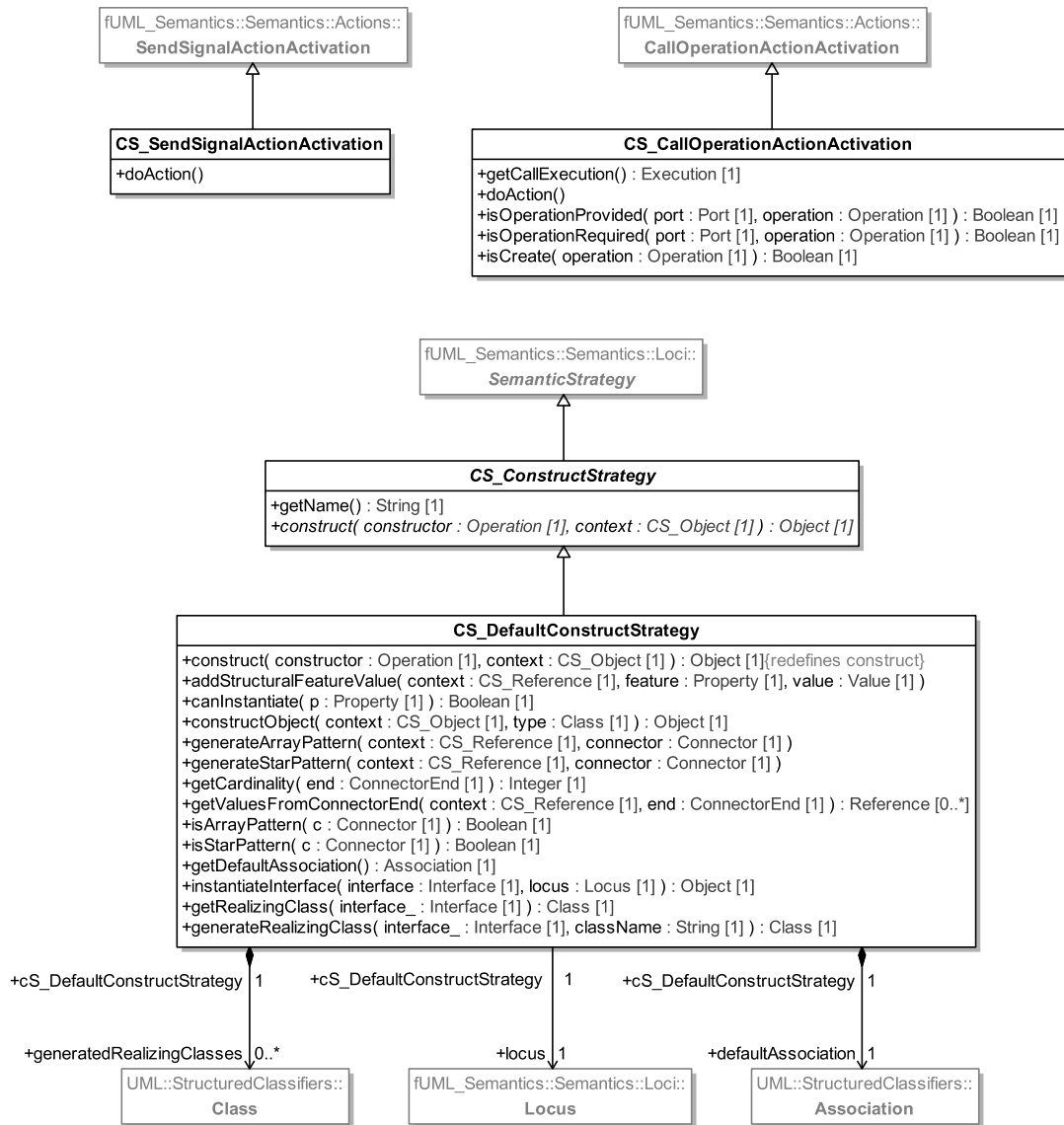
Figure 8.8 - Invocation Action Activations
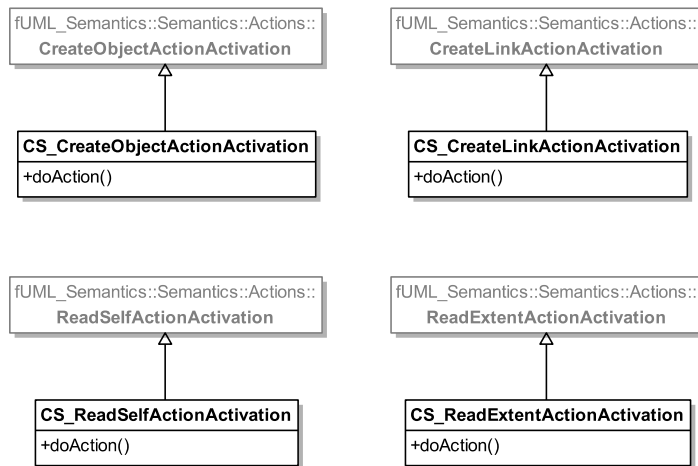
Figure 8.9 - Object and Link Action Activations
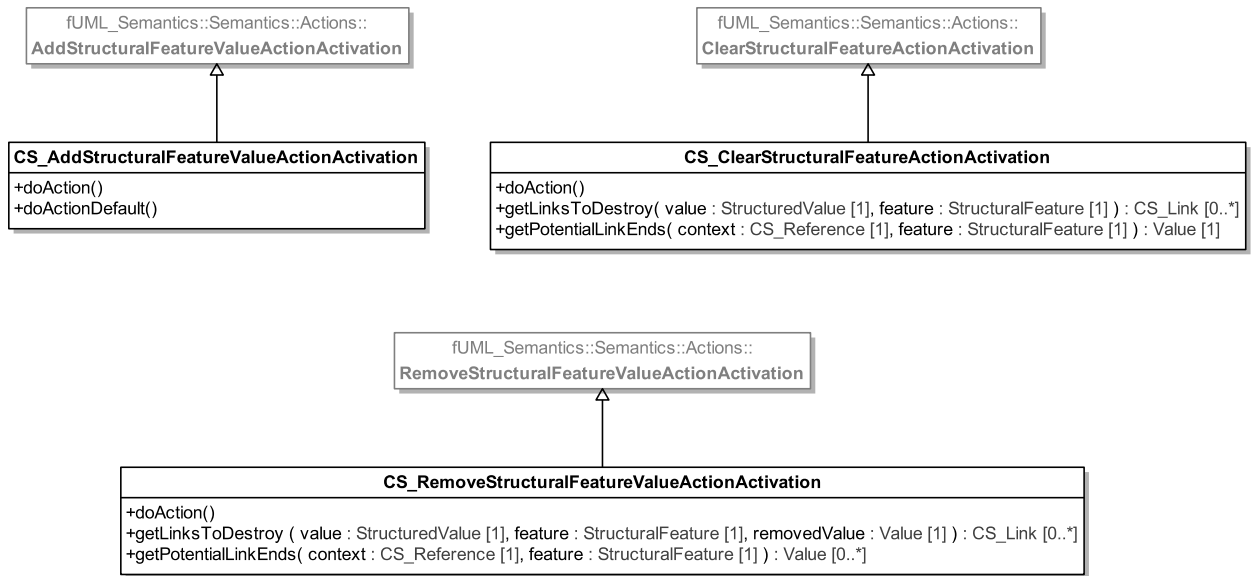
Figure 8.10 - Structural Feature Action Activations
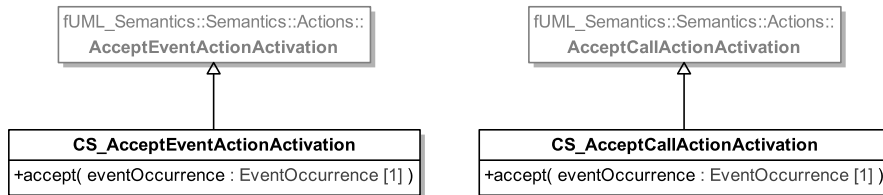
Figure 8.11 - Accept Action Activations

8.6.2 Class Descriptions

8.6.2.1 CS_AcceptCallActionActivation

In the PSCS context, an accepted event occurrence might be a CS_EventOccurence (i.e., an event occurrence that arrived through an instance of a Port). CS_AcceptCallActionActivation provides the required extension to AcceptCallActionActivation in order to deal with the acceptance of a CS_EventOccurence.

Generalizations

- AcceptCallActionActivation (from fUML_Semantics::Semantics::Actions)

Attributes

- None

Associations

- None

Operations

```
[1] public accept ( eventOccurrence:EventOccurrence )
// If the accepted event occurrence is a CS_EventOccurrence
// then the wrapped event occurrence is extracted.
// The acceptance process is the one define by AcceptCallActionActivation
// defined in fUML.

if(eventOccurrence instanceof CS_EventOccurrence){
```

```
        super.accept(((CS_EventOccurrence) eventOccurrence).wrappedEventOccurrence);
}else{
super.accept(eventOccurrence);
};
```

8.6.2.2 CS_AcceptEventActionActivation

In the PSCS context, an accepted event occurrence might be a CS_EventOccurence (i.e., an event occurrence that arrived through an instance of a Port). CS_AcceptEventActionActivation provides the required extension to AcceptEventActionActivation in order to deal with the acceptance of a CS_EventOccurence.

Generalizations

- AcceptEventActionActivation (from fUML_Semantics::Semantics::Actions)

Attributes

- None

Associations

- None

Operations

```
[1] public accept ( eventOccurrence:EventOccurrence )
// If the accepted event occurrence is a CS_EventOccurrence
// then the wrapped event occurrence is extracted.
// The acceptance process is the one define by AcceptEventActionActivation
// defined in fUML.

if(eventOccurrence instanceof CS_EventOccurrence){
super.accept(((CS_EventOccurrence) eventOccurrence).wrappedEventOccurrence);
}else{
super.accept(eventOccurrence);
};
```

8.6.2.3 CS_AddStructuralFeatureValueActionActivation

CS_AddStructuralFeatureValueActionActivation extends fUML AddStructuralFeatureValueActionActivation so that, if the StructuralFeature identified by the Action is a Port, a CS_InteractionPoint is inserted as a Value for the Port instead of a CS_Reference. In the case where the StructuralFeature is not a Port but identifies an association end, fUML semantics are also extended so that a CS_Link is created instead of a fUML Link.

Generalizations

- AddStructuralFeatureValueActionActivation (from fUML_Semantics::Semantics::Actions)

Attributes

- None

Associations

- None

Operations

```
[1] public doAction()
        // If the feature is a port and the input value to be added is a CS_Reference,
        // Replaces this CS_Reference by a CS_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
            // CS_Reference (otherwise, the feature cannot be a port)
            CS_Reference owner = (CS_Reference) this.takeTokens(
                    action.object).getValue(0);
            interactionPoint.owner = owner;
            // Then replaces the CS_Reference by a CS_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);
        }
```

8.6.2.4   CS_CallOperationActionActivation

FUML semantics are extended from two aspects. The first aspects accounts for property onPort of the CallOperationAction. If onPort is not specified, fUML semantics are unchanged. If onPort is specified, instead of dispatching directly to the target reference by calling operation dispatch (as in fUML):

- If the invoked BehavioralFeature is on a provided Interface but not on any required Interface, then, when the InvocationAction is executed, the invocation is made into the object given on the target InputPin through the given Port.

- If the invoked BehavioralFeature is on a required Interface but not on any provided Interface, then, if the InvocationAction is being executed inside the object given on the target InputPin, the invocation is forwarded out of the target object through the given Port.

- If the invoked BehavioralFeature is on both a provided and a required Interface, then, if the InvocationAction is being executed inside the object given on the target InputPin, the invocation is made out of the target object through the given Port.

Otherwise the invocation is made into the target object through the given Port.

The second aspects deals with instantiation semantics of composite structures. If the CallOperationAction concerns a constructor (i.e., the Operation has stereotype «Create» applied) and if this Operation has no method, the construction strategy registered at the execution locus (cf. 8.6.2.6, CS_ConstructStrategy) is executed to construct the target Object.

Generalizations

- CallOperationActionActivation (from fUML_Semantics::Semantics::Actions)

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 invoked BehavioralFeature is on a provided Interface but not on any required Interface,
  // then, when the InvocationAction is executed, the invocation is made into the object given on
  // the target InputPin through the given Port
  // - If the invoked BehavioralFeature is on a required Interface but not on any provided Interface,
  // then, if the InvocationAction is being executed inside the object given on the target InputPin,
  // the invocation is forwarded out of the target object through the given Port.
  // - If the invoked BehavioralFeature is on both a provided and a required Interface,
  // then, if the InvocationAction is being executed inside the object given on the target InputPin,
  // the invocation is made out of the target object through the given Port.
  // Otherwise the invocation is made into the target object through the given Port.
  CallOperationAction action = (CallOperationAction)(this.node);
  Execution execution = null ;
  if (action.onPort == null ) {
    execution = super.getCallExecution() ;
  }
  else {
    Value target = this.takeTokens(action.target).get(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.getActivityExecution().context ;
      boolean operationIsOnProvidedInterface =
        this.isOperationProvided(action.onPort, action.operation) ;
      boolean operationIsOnRequiredInterface =
        this.isOperationRequired(action.onPort, action.operation) ;
      // Operation on a provided interface only
      if (operationIsOnProvidedInterface && !operationIsOnRequiredInterface) {
        execution = targetReference.dispatchIn(action.operation, action.onPort);
      }
      // Operation is on a required interface only
      else if (!operationIsOnProvidedInterface && operationIsOnRequiredInterface){
        // If not executing in the context of the target,
        // Semantics are undefined.
        // Otherwise, dispatch outside.
        if (executionContext == targetReference.referent ||
              targetReference.compositeReferent.contains(executionContext)) {
          execution = targetReference.dispatchOut(action.operation, action.onPort);
        }
      }
      // Operation is both on a provided and a required interface
      else if (operationIsOnProvidedInterface && operationIsOnRequiredInterface) {
        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;

[2] public doAction()
  CallOperationAction action = (CallOperationAction)(this.node);
  // First determines if this is a call to a constructor and if a default
  // construction strategy needs to be applied.
  // This is a call to a constructor if the called operation has
  // stereotype <<Create>> applied.
  // The default construction strategy is used if no method is associated with the
  // <<Create>> operation.
  // Otherwise, behaves like in fUML.
  if (action.onPort == null && this.isCreate(action.operation)
      && action.operation.method.size() == 0) {
    Locus locus = this.getExecutionLocus() ;
    CS_ConstructStrategy strategy = ((CS_ConstructStrategy)locus.
                              factory.getStrategy("constructStrategy")) ;
    Value target = this.takeTokens(action.target).get(0);
    if (target instanceof CS_Reference) {
      strategy.construct(action.operation, ((CS_Reference)target).compositeReferent) ;
      ParameterList parameters = action.operation.ownedParameter ;
      OutputPinList resultPins = action.result ;
      ValueList values = new ValueList() ;
      values.add(target) ;
      int i = 1 ;
      while(i <= parameters.size()) {
        Parameter parameter = parameters.get(i - 1);
        if(parameter.direction == ParameterDirectionKind.return_) {
          OutputPin resultPin = resultPins.get(0);
          this.putTokens(resultPin, values);
        }
        i = i + 1;
      }
    }
  }
  else {
    super.doAction();
  }

[3] public isOperationProvided()
  boolean isProvided = false ;
  if (operation.owner instanceof Interface) {
    // We have to look in provided interfaces of the port if
    // they define directly or indirectly the Operation
    Integer interfaceIndex = 1 ;
    // Iterates on provided interfaces of the port
    InterfaceList providedInterfaces = port.provided ;
    while (interfaceIndex <= providedInterfaces.size() && !isProvided) {
      Interface interface_ = providedInterfaces.get(interfaceIndex-1) ;
      // Iterates on members of the current Interface
      Integer memberIndex = 1 ;
      while (memberIndex <= interface_.member.size() && !isProvided) {
        NamedElement cddOperation = interface_.member.get(memberIndex-1) ;
        if (cddOperation instanceof Operation) {
          isProvided = operation == cddOperation ;
        }
        memberIndex = memberIndex + 1 ;
      }
      interfaceIndex = interfaceIndex + 1 ;
    }
  }
  return isProvided ;

[4] public isOperationProvided()
  boolean isRequired = false ;
  Integer interfaceIndex = 1 ;
  // Iterates on provided interfaces of the port
  InterfaceList requiredInterfaces = port.required ;
  while (interfaceIndex <= requiredInterfaces.size() && !isRequired) {
    Interface interface_ = requiredInterfaces.get(interfaceIndex-1) ;
    // Iterates on members of the current Interface
```

```
    Integer memberIndex = 1 ;
    while (memberIndex <= interface_.member.size() && !isRequired) {
      NamedElement cddOperation = interface_.member.get(memberIndex-1) ;
      if (cddOperation instanceof Operation) {
        isRequired = operation == cddOperation ;
      }
      memberIndex = memberIndex + 1 ;
    }
    interfaceIndex = interfaceIndex + 1 ;
  }
  return isRequired ;

[5] public isCreate()
  CS_ExecutionFactory executionFactory = (CS_ExecutionFactory)this.getExecutionLocus().factory ;
  Class_ stereotypeCreate = executionFactory.getStereotypeClass("StandardProfile", "Create") ;
  if (stereotypeCreate == null) {
    // standard profile is not applied
    return false ;
  }
  return executionFactory.getStereotypeApplication(stereotypeCreate, o) != null ;
```

8.6.2.5   CS_ClearStructuralFeatureActionActivation

FUML semantics are extended so that, when a StructuralFeature is cleared, any links representing an instance of a Connector in which the StructuralFeature is involved (i.e., it is a role or a partWithPort for a ConnectorEnd of this Connector) are destroyed.

Generalizations

- ClearStructuralFeatureActionActivation (from fUML_Semantics::Semantics::Actions)

Attributes

- None

Associations

- None

Operations

```
[1] public doAction()
  // Get the value of the object input pin.
  // If the given feature is an association end, then
  // destroy all links that have the object input on the opposite end.
  // Otherwise, if the object input is a structured value, then
  // set the appropriate feature of the input value to be empty.
  ClearStructuralFeatureAction action = (ClearStructuralFeatureAction)(this.node);
  StructuralFeature feature = action.structuralFeature;
  Association association = this.getAssociation(feature);
  Value value = this.takeTokens(action.object).get(0);
  if(association != null) {
    LinkList links = this.getMatchingLinks(association, feature, value);
    for(int i = 0; i < links.size(); i++) {
      Link link = links.get(i);
      link.destroy();
    }
  } else if(value instanceof StructuredValue) {
    // If the value is a data value, then it must be copied before
    // any change is made.
    if(!(value instanceof Reference)) {
      value = value.copy();
    }
    else {
      // extension to fUML
      CS_LinkList linksToDestroy = this.getLinksToDestroy((StructuredValue)value, feature) ;
      for (int i = 0 ; i < linksToDestroy.size() ; i++) {
        linksToDestroy.get(i).destroy() ;
      }
      //
```

Precise Semantics of UML Composite Structures, v1.2 **47**

```
    }
    ((StructuredValue)value).setFeatureValue(action.structuralFeature, new ValueList(), 0);
  }
  if(action.result != null) {
    this.putToken(action.result, value);
  }

[2] public getLinksToDestroy(value: StructuredValue, feature:StructuralFeature) : CS_Link[*]
  // Retrieves links that must be destroyed when the given feature
  // is cleared in the context of the given value
  CS_LinkList linksToDestroy = new CS_LinkList() ;
  if (value instanceof CS_Reference) {
    CS_Reference context = (CS_Reference)value ;
    // Retrieves the feature values for the structural feature associated with this action,
    // in the context of this reference
    FeatureValue featureValue = context.getFeatureValue(feature) ;
    if (feature instanceof Port) {
      // all values are interaction points
      // any link targeting this interaction point must be destroyed
      for (int i = 0 ; i < featureValue.values.size() ; i++) {
        CS_InteractionPoint interactionPoint = (CS_InteractionPoint)featureValue.values.get(i) ;
        CS_LinkList connectorInstances = context.compositeReferent.getLinks(interactionPoint) ;
        for (int j = 0 ; j < connectorInstances.size() ; j++) {
          CS_Link link = connectorInstances.get(j) ;
          linksToDestroy.add(link) ;
        }
      }
    }
  }
  else { // feature is an attribute
    // Retrieve all potential link ends,
    // separating potential link ends corresponding to the given feature,
    // and potential link ends corresponding to other features.
    // By "potential link ends", we refer to the values of a given feature,
    // as well as interaction points associated with this value, if any.
    ValueList allValuesForFeature = new ValueList() ;
    ValueList allOtherValues = new ValueList() ;
    for (int i = 0 ; i < context.referent.featureValues.size() ; i++) {
      StructuralFeature currentFeature = context.referent.featureValues.get(i).feature ;
      ValueList values = this.getPotentialLinkEnds(context, currentFeature) ;
      for (int j = 0 ; j < values.size() ; j++) {
        Value v = values.get(j) ;
        if (currentFeature != feature) {
          allOtherValues.add(v) ;
        }
        else {
          allValuesForFeature.add(v) ;
        }
      }
    }
    // Retrieves all links available at the locus
    ExtensionalValueList extensionalValues = this.getExecutionLocus().extensionalValues ;
    CS_LinkList allLinks = new CS_LinkList() ;
    for (int i = 0 ; i < extensionalValues.size() ; i ++) {
      ExtensionalValue extensionalValue = extensionalValues.get(i) ;
      if (extensionalValue instanceof CS_Link) {
        allLinks.add((CS_Link)extensionalValue) ;
      }
    }
    // Retrieves links representing connector instances in the context object
    for (int i = 0 ; i < allLinks.size() ; i++) {
      CS_Link link = allLinks.get(i) ;
      boolean linkHasToBeDestroyed = false ;
      for (int j = 0 ; j < allValuesForFeature.size() && !linkHasToBeDestroyed; j++) {
        Value v = allValuesForFeature.get(j) ;
        StructuralFeature featureForV = link.getFeature(v) ;
        if (featureForV != null) {
          // Check if feature values of this link for other features
          // contains elements identified in allOtherValue
          for (int k = 0 ; k < link.featureValues.size() && !linkHasToBeDestroyed ; k++) {
            FeatureValue otherFeatureValue = link.featureValues.get(k) ;
            if (otherFeatureValue.feature != featureForV) {
              for (int l = 0; l < otherFeatureValue.values.size() && !linkHasToBeDestroyed ; l++) {
                for (int m = 0 ; m < allOtherValues.size() && !linkHasToBeDestroyed ; m++) {
```

```
                     if (otherFeatureValue.values.get(l) == allOtherValues.get(m)) {
                       linkHasToBeDestroyed = true ;
                     }
                   }
                 }
               }
             }
           }
           if (linkHasToBeDestroyed) {
             linksToDestroy.add(link) ;
           }
         }
       }
     }
   return linksToDestroy ;

[3] public getPotentialLinkEnds(context : CS_Reference, feature : StructuralFeature) : Value[*]
   // Retrieves all feature values for the context object for the given feature,
   // as well as all interaction point for these values
   ValueList potentialLinkEnds = new ValueList() ;
   FeatureValue featureValue = context.getFeatureValue(feature) ;
   for (int i = 0 ; i < featureValue.values.size() ; i++) {
     Value v = featureValue.values.get(i) ;
     potentialLinkEnds.add(v) ;
     if (v instanceof CS_Reference) {
       // add all interaction points associated with v
       for (int j = 0 ; j < ((CS_Reference)v).referent.featureValues.size() ; j++) {
         if (((CS_Reference)v).referent.featureValues.get(j).feature instanceof Port) {
           ValueList interactionPoints = (((CS_Reference)v).referent.featureValues.get(j)).values ;
           for (int k = 0 ; k < interactionPoints.size() ; k++) {
             potentialLinkEnds.add(interactionPoints.get(k)) ;
           }
         }
       }
     }
   }
   return potentialLinkEnds ;
```

8.6.2.6 CS_ConstructStrategy

CS_ConstructStrategy is a new semantic strategy. It deals with instantiation semantics of composite structures. The context in which this strategy is involved is described in 8.6.2.4, CS_CallOperationActionActivation. This specification defines a default realization for this strategy: CS_DefaultConstructStrategy.

Generalizations

- SemanticStrategy (from fUML_Semantics::Semantics::Loci)

Attributes

- None

Associations

- None

Operations

```
[1] public getName() : String
   // a CS_ConstructionStrategy is always named "constructStrategy"
   return "constructStrategy";

[2] public abstract construct(constructor:Operation, context:CS_Object) : Object
```

8.6.2.7 CS_CreateLinkActionActivation

FUML semantics are extended so that a CS_Link is created instead of a fUML Link.

Generalizations

- CreateLinkActionActivation (from fUML_Semantics::Semantics::Actions)

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 is 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);
```

8.6.2.8   CS_CreateObjectActionActivation

FUML semantics are extended so that a CS_Reference is produced instead of a fUML Reference, in the case where the execution locus instantiates a CS_Object instead of a fUML Object. With the extensions defined in this specification, a fUML Object is instantiated only in the case where the classifier identified by the Action is a Behavior.

Generalizations

- CreateObjectActionActivation (from fUML_Semantics::Semantics::Actions)

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);
```

8.6.2.9   CS_DefaultConstructStrategy

CS_DefaultConstructStrategy is the default realization provided by this specification for semantic strategy CS_ConstructStrategy. This strategy deals with instantiation of composite structures as follows:

- **Instantiation of parts and ports:** Parts (i.e., composite Properties) are instantiated according to their multiplicity lower bound. The same rule applies for Ports. It means that, if the lower bound of parts is 0, the topology resulting from the instantiation of a composite structure is empty. This case is illustrated in Figure 8.12. Instantiation of a value for a Port results in the creation of a CS_InteractionPoint, which itself refers to a CS_Object typed by the type of the Port. If the Port is typed by an Interface, the CS_Object is typed by a Class which realizes this Interface. This Class is dynamically generated, following rules defined in Operation getRealizingClass specified below.
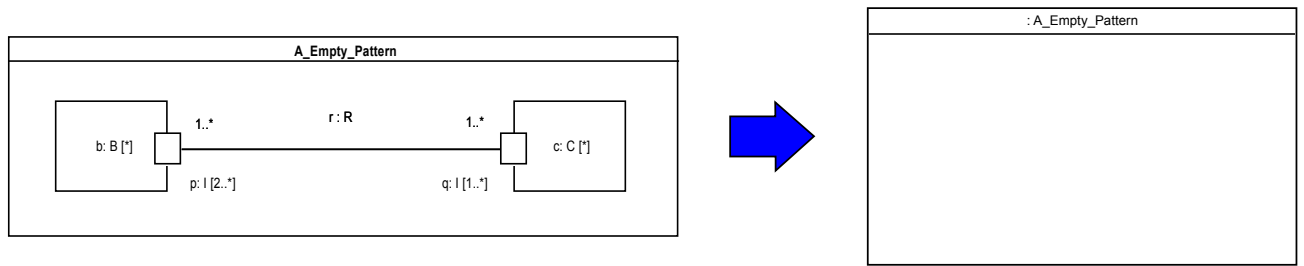
Figure 8.12 - Instantiation of a composite structure resulting in an empty topology

- **Instantiation of connectors:** Instantiation of Connectors occurs only for binary Connectors. The number of Connectors to be instantiated depends on the multiplicity lower bound of their ConnectorEnds, as well as on the number of parts/ports (identified by these ends) that have been instantiated according to rules previously mentioned. It means that, if no parts/ports have been instantiated, the instantiation of connectors will not occur, as illustrated in Figure 8.12. It also means that, if the lower bound on ends of a Connector is 0, the instantiation will not occur as well. This case is illustrated in Figure 8.13. Instantiation of a Connector results in the creation of a CS_Link. This link is typed by the type of the Connector, which is an Association. If the Connector is not typed, the created link is typed by an Association which is dynamically generated, following rules defined in Operation getDefaultAssociation specified below. The elements which act as values for the parts/ports identified by the ConnectorEnds also act as values for the ends of the created CS_Link (i.e., FeatureValues corresponding to the Association ends).
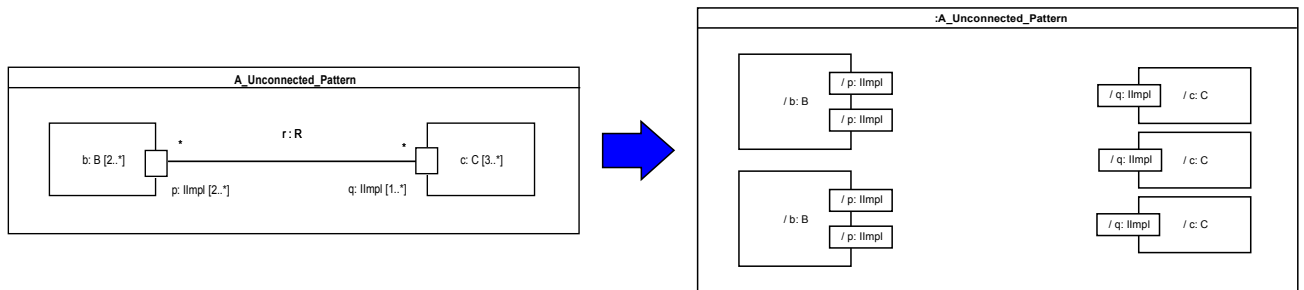
Figure 8.13 - Instantiation of a composite structure resulting in an unconnected topology

It is not always possible to unambiguously compute the topology of links that results from the instantiation of a composite structure. The two cases supported by this default strategy are the Array topology and the Star topology. The Array topology is the result of a situation where: the multiplicity lower bound of ConnectorEnds is 1, and the number of elements identified by each ConnectorEnd is the same for all the ConnectorEnds of the Connector. The number of ends identified by a ConnectorEnd is computed by multiplying the multiplicity lower bound of the ConnectorEnd::role by the multiplicity lower bound of the ConnectorEnd::partWithPort. If the ConnectorEnd has no partWithPort, the number of ends it identifies is simply the multiplicity lower bound of its role. An example of Array topology is illustrated in Figure 8.14.

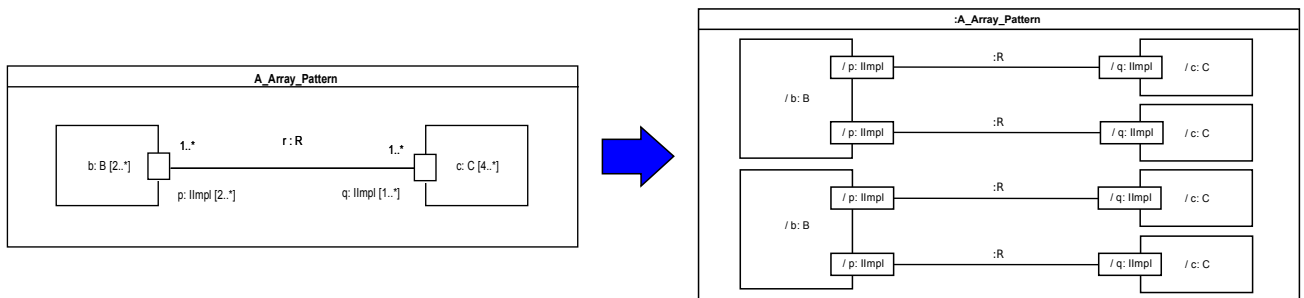**Figure 8.14 - Instantiation of a composite structure resulting in an array topology**

The Star topology is the result of a situation where the multiplicity lower bound of each ConnectorEnd is equal to the number of ends it identifies. An example of Star topology is illustrated in Figure 8.15. For other

combinations of part/port/connector end multiplicity lower bound, this default strategy does not instantiate any link.
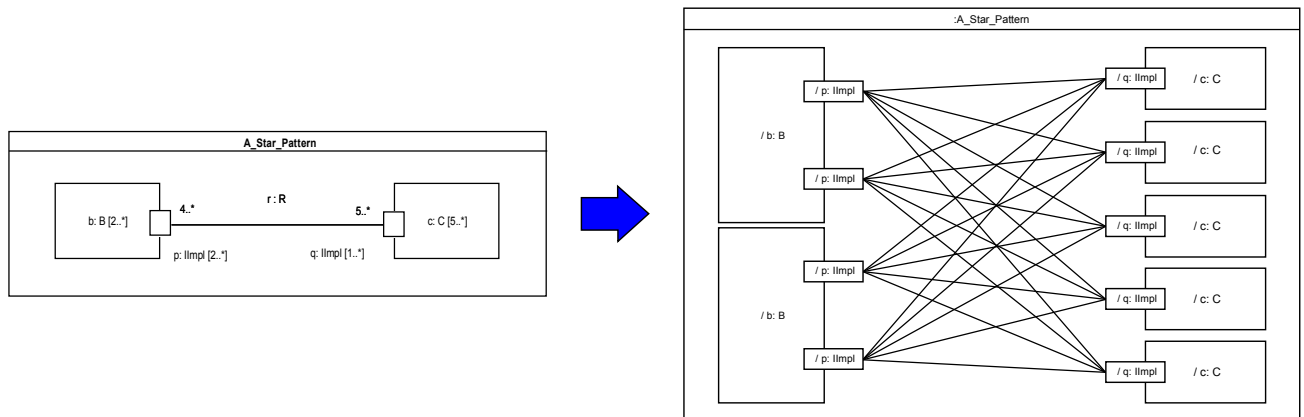
Figure 8.15 - Instantiation of a composite structure in a star topology

- **Assignment of default values**: If a default value is specified for a part, this construction strategy evaluates the ValueSpecification, and assigns the result of the evaluation to the value of this part.

Generalizations

- CS_ConstructStrategy (from PSCS_Semantics::Semantics::Actions)

Attributes

- None

Associations

- defaultAssociation : Association [0..1], the Association used to type CS_Links derived from untyped Connectors. This Association is generated once and reused for all CS_Links derived from untyped Connectors.
- generatedRealizingClasses : Class [*], the set of Classes generated to type Port objects, in the case of Ports typed by Interfaces.
- locus : Locus [1], the locus in which elements will be instantiated. It is obtained from a context CS_Object, which is passed as an argument to operation construct.

Operations

```
[1] public addStructuralFeatureValue(context:CS_Reference, feature:Property, value:Value)
      FeatureValue featureValue = context.getFeatureValue(feature) ;
      if (featureValue != null) {
            ValueList values = featureValue.values ;
            if (feature instanceof Port) {
                  // insert an interaction point
                  CS_InteractionPoint interactionPoint = new CS_InteractionPoint() ;
                  interactionPoint.definingPort = (Port)feature ;
                  interactionPoint.referent = (CS_Object)value ;
                  interactionPoint.owner = context ;
                  values.add(interactionPoint) ;
            }
            else if (value instanceof CS_Object) {
                  // insert a reference
                  CS_Reference reference = new CS_Reference() ;
                  reference.compositeReferent = (CS_Object)value ;
                  reference.referent = (CS_Object)value ;
                  values.add(reference) ;
            }
            else {
                  values.add(value) ;
            }
```

```
        }

[2] public canInstantiate(p:Property) : Boolean
        // Instantiate is possible if:
        // - p is composite
        // - p is typed
        // - This type is a Class and it is not abstract
        // - Or p is a Port and the type is an Interface
        if (p.isComposite) {
                if (p.typedElement.type != null) {
                        if (p.typedElement.type instanceof Class_) {
                                return ! ((Class_)p.typedElement.type).isAbstract ;
                        }
                }
                else if (p.typedElement.type instanceof Interface) {
                        return p instanceof Port ;
                }
        }
        return false ;

[3] public construct(constructor:Operation, context:CS_Object) : Object
        this.locus = context.locus ;
        return this.constructObject(context, (Class_)constructor.type) ;

[4] public constructObject(context:CS_Object, type:Class) : Object
  CS_Reference referenceToContext = new CS_Reference() ;
  referenceToContext.referent = context ;
  referenceToContext.compositeReferent = (CS_Object)context ;
  // FIXME detect infinite recursive instantiation
  PropertyList allAttributes = type.attribute ;
  int i = 1 ;
  // Instantiate ports and parts
  while (i <= allAttributes.size()) {
    Property p = allAttributes.get(i - 1) ;
    if (p.default_ != null) {
      ValueSpecification defaultValueSpecification = p.default_ ;
      Evaluation evaluation =
              (Evaluation)context.locus.factory.instantiateVisitor(defaultValueSpecification) ;
      evaluation.specification = defaultValueSpecification ;
      evaluation.locus = context.locus ;
      if (evaluation instanceof CS_OpaqueExpressionEvaluation) {
        ValueList evaluations =
                ((CS_OpaqueExpressionEvaluation)evaluation).executeExpressionBehavior() ;
        for (int j = 0 ; j < evaluations.size() ; j++) {
          this.addStructuralFeatureValue(referenceToContext, p, evaluations.get(j)) ;
        }
      }
      else {
        Value defaultValue = evaluation.evaluate() ;
        this.addStructuralFeatureValue(referenceToContext, p, defaultValue) ;
      }
    }
    else if (this.canInstantiate(p)) {
      int j = 1 ;
      while (j <= p.multiplicityElement.lower) {
        Object_ value ;
        // if p is a Port typed by an Interface
        // creates an Object without type, but with FeatureValues corresponding to
        // structural features of the interface.
        if (p instanceof Port && p.typedElement.type instanceof Interface) {
          value = this.instantiateInterface((Interface)p.typedElement.type, this.locus) ;
          this.addStructuralFeatureValue(referenceToContext, p, value);
        }
        else {
          value = context.locus.instantiate((Class_)p.typedElement.type) ;
          // TODO account for existing constructors
          value = this.constructObject((CS_Object)value, (Class_)p.typedElement.type) ;
          this.addStructuralFeatureValue(referenceToContext, p, (CS_Object)value) ;
          if (((Class_)p.typedElement.type).isActive) {
            value.startBehavior((Class_)p.typedElement.type, new ParameterValueList()) ;
          }
        }
        j = j + 1 ;
```

```
          }
        }
        i = i + 1 ;
    }
    // Instantiate connectors
    NamedElementList allMembers = type.member ;
    i = 1 ;
    while (i <= allMembers.size()) {
        NamedElement member = allMembers.get(i - 1) ;
        if (member instanceof Connector) {
            Connector connector = (Connector)member ;
            if (this.isArrayPattern(connector)) {
                this.generateArrayPattern(referenceToContext, connector) ;
            }
            else if (this.isStarPattern(connector)) {
                this.generateStarPattern(referenceToContext, connector) ;
            }
        }
        i = i + 1 ;
    }
    return referenceToContext.referent ;

[5] public generateArrayPattern(context:CS_Reference, connector:Connector)
        ConnectorEnd end1 = connector.end.getValue(0) ;
        ConnectorEnd end2 = connector.end.getValue(1) ;
        ReferenceList end1Values = this.getValuesFromConnectorEnd(context, end1) ;
        ReferenceList end2Values = this.getValuesFromConnectorEnd(context, end2) ;
        for (int i = 0 ; i < end1Values.size() ; i++) {
                CS_Link link = new CS_Link() ;
                if (connector.type == null) {
                        link.type = this.getDefaultAssociation() ;
                }
                else {
                        link.type = connector.type ;
                }
                ValueList valuesForEnd1 = new ValueList() ;
                valuesForEnd1.add(end1Values.get(i)) ;
                ValueList valuesForEnd2 = new ValueList() ;
                valuesForEnd2.add(end2Values.get(i)) ;
                link.setFeatureValue(link.type.ownedEnd.getValue(0), valuesForEnd1, -1) ;
                link.setFeatureValue(link.type.ownedEnd.getValue(1), valuesForEnd2, -1) ;
                link.addTo(context.referent.locus) ;
        }

[6] public generateRealizingClass(interface_:Interface, className:String) : Class
        Class_ realizingClass = new Class_() ;
        realizingClass.setName(className) ;
        InterfaceRealization realization = new InterfaceRealization() ;
        realization.contract = interface_ ;
        realization.implementingClassifier = realizingClass ;
        realizingClass.interfaceRealization.addValue(realization);
        // TODO Deal with structural features of the interface
        // TODO Make a test case for reading/writing structural features of an interface
        return realizingClass ;

[7] public generateStarPattern(context:CS_Reference, connector:Connector)
        ConnectorEnd end1 = connector.getEnds().get(0) ;
        ConnectorEnd end2 = connector.getEnds().get(1) ;
        List<Reference> end1Values = this.getValuesFromConnectorEnd(context, end1) ;
        List<Reference> end2Values = this.getValuesFromConnectorEnd(context, end2) ;
        for (int i = 0 ; i < end1Values.size() ; i++) {
                for (int j = 0 ; j < end2Values.size(); j++) {
                        CS_Link link = new CS_Link() ;
                        if (connector.type == null) {
                                link.type = this.getDefaultAssociation() ;
                        }
                        else {
                                link.type = connector.type ;
                        }
                        List<Value> valuesForEnd1 = new ArrayList<Value>() ;
                        valuesForEnd1.add(end1Values.get(i)) ;
                        List<Value> valuesForEnd2 = new ArrayList<Value>() ;
                        valuesForEnd2.add(end2Values.get(j)) ;
```

```
                    link.setFeatureValue(link.type.getOwnedEnds().get(0), valuesForEnd1, -1) ;
                    link.setFeatureValue(link.type.getOwnedEnds().get(1), valuesForEnd2, -1) ;
                    link.addTo(context.referent.locus) ;
                }
        }

[8] public getCardinality(end:ConnectorEnd) : Integer
        int lowerOfRole = end.role.actualConnectableElement.multiplicityElement.lower ;
        if (lowerOfRole == 0) {
                return 0 ;
        }
        else if (end.partWithPort == null) {
                return lowerOfRole ;
        }
        else {
                int lowerOfPart = end.partWithPort.multiplicityElement.lower ;
                return lowerOfRole * lowerOfPart ;
        }

[9] public getDefaultAssociation() : Association
        // Computes an returns an Association with two untyped owned ends,
        // with multiplicity [*].
        // This association can be used to type links instantiated from untyped connectors
        if (defaultAssociation == null) {
          defaultAssociation = new Association() ;
          defaultAssociation.name = "DefaultGeneratedAssociation";
          Property end1 = new Property() ;
          end1.setName("x"); ;
          end1.setLower(0);
          end1.setUpper(-1);
          end1.setIsOrdered(true);
          end1.setIsUnique(true);
          defaultAssociation.addOwnedEnd(end1);
          Property end2 = new Property() ;
          end2.setName("y");
          end2.setLower(0);
          end2.setUpper(-1);
          end2.setIsOrdered(true);
          end2.setIsUnique(true);
          defaultAssociation.addOwnedEnd(end2);
        }
        return defaultAssociation ;

[10] public getRealizingClass(interface_:Interface) : Class
        Class_ realizingClass = null ;
        // TODO For cached RealizingClasses, search based on InterfaceRealizations rather than name
        String realizingClassName = interface_.qualifiedName + "GeneratedRealizingClass" ;
        int i = 1 ;
        while (i <= generatedRealizingClasses.size() && realizingClass == null) {
          Class_ cddRealizingClass = generatedRealizingClasses.getValue(i - 1) ;
          if (cddRealizingClass.name.equals(realizingClassName)) {
            realizingClass = cddRealizingClass ;
          }
          i = i + 1 ;
        }
        if (realizingClass == null) {
          realizingClass = this.generateRealizingClass(interface_, realizingClassName) ;
          generatedRealizingClasses.addValue(realizingClass) ;
        }
        return realizingClass ;

[11] public getValuesFromConnectorEnd(context:CS_Reference, end:ConnectorEnd) : Value[*]
  ReferenceList endValues = new ReferenceList() ;
  if (end.partWithPort != null) {
    FeatureValue valueForPart = context.getFeatureValue(end.partWithPort) ;
    if (valueForPart != null) {
      for (int i = 0 ; i < valueForPart.values.size() ; i++) {
        Reference reference = (Reference)valueForPart.values.get(i) ;
        FeatureValue valueForPort =
                    reference.getFeatureValue((Port)end.role.actualConnectableElement) ;
        if (valueForPort != null) {
          for (int j = 0 ; j < valueForPort.values.size() ; j++) {
            endValues.add((Reference)valueForPort.values.get(j)) ;
```

```
            }
          }
        }
      }
    }
    else {
      FeatureValue valueForRole= context.getFeatureValue((Property)end.role.actualConnectableElement) ;
      if (valueForRole != null) {
        for (int i = 0 ; i < valueForRole.values.size() ; i++) {
          endValues.add((Reference)valueForRole.values.get(i)) ;
        }
      }
    }
  }
  return endValues ;

[12] public instantiateInterface(interface_:Interface, locus:Locus) : Object
      Class_ realizingClass = this.getRealizingClass(interface_) ;
      Object_ object = locus.instantiate(realizingClass) ;
      return object ;

[13] public isArrayPattern(c:Connector) : Boolean
  // This is an array pattern if:
  // - c is typed by an association FIXME this may no longer be required in UML 2.5
  // - c is binary
  // - lower bound of the two connector ends is 1
  // - Cardinality of ends are equals
  if (c.end.size() == 2) {
    if (c.end.get(0).role.actualConnectableElement.multiplicityElement.lower == 1) {
      if (c.end.get(1).role.actualConnectableElement.multiplicityElement.lower == 1) {
        if (this.canInstantiate(c.end.get(0).role.actualConnectableElement) &&
            this.canInstantiate(c.end.get(1).role.actualConnectableElement)) {
          int cardinality1 = this.getCardinality(c.end.get(0)) ;
          int cardinality2 = this.getCardinality(c.end.get(1)) ;
          return cardinality1 == cardinality2 ;
        }
      }
    }
  }
  return false ;

[14] public isStarPattern(c:Connector) : Boolean
  // This is a star pattern if:
  // - c is binary
  // - lower bound of end1 equals cardinality of end1
  // - lower bound of end2 equals cardinality of end2
  if (c.end.size() == 2) {
    if (this.canInstantiate(c.end.get(0).role.actualConnectableElement) &&
        this.canInstantiate(c.end.get(1).role.actualConnectableElement)) {
      int cardinalityOfEnd1 = this.getCardinality(c.end.get(0)) ;
      int lowerBoundofEnd1 = c.end.get(0).role.actualConnectableElement.multiplicityElement.lower ;
      if (cardinalityOfEnd1 == lowerBoundofEnd1) {
        int cardinalityOfEnd2 = this.getCardinality(c.end.get(1)) ;
        int lowerBoundofEnd2 = c.end.get(1).role.actualConnectableElement.multiplicityElement.lower ;
        return cardinalityOfEnd2 == lowerBoundofEnd2 ;
      }
    }
  }
  return false ;
```

8.6.2.10 CS_ReadExtentActionActivation

CS_ReadExtentActionActivation extends fUML ReadExtentActionActivation to account for the introduction of classes CS_Object (8.4.2.7) and CS_Reference (8.4.2.8). As compared to fUML semantics (which produces a collection of References to Objects available at the execution Locus), for each CS_Object (classified by the given ReadExtentAction classifier) that is available at the execution Locus, a CS_Reference is produced instead of a fUML Reference.

Generalizations

- ReadExtentActionActivation (from fUML_Semantics::Semantics::Actions)

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);
      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);
```

8.6.2.11 CS_ReadSelfActionActivation

FUML semantics are extended so that a CS_Reference is produced instead of a fUML reference.

Generalizations

- ReadSelfActionActivation (from fUML_Semantics::Semantics::Actions)

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);
```

8.6.2.12 CS_RemoveStructuralFeatureValueActionActivation

In PSCS context, when a value is removed from a structural feature, all links in which this value is involved must be destroyed. The link destruction must also be propagated to links in which values referenced in feature values of the removed value are involved. The rule applies recursively.

Generalizations

- RemoveStructuralFeatureValueActionActivation (from fUML_Semantics::Semantics::Actions)

Attributes

- None

Associations

- None

Operations

```
[1] public doAction()

// Get the values of the object and value input pins.
// If the given feature is an association end, then destroy any matching links.
// Otherwise, if the object input is a structural value, remove values from the given feature
// and destroy all links in which the removed values are involved.
// If isRemoveDuplicates is true, then destroy all current matching links
// or remove all values equal to the input value.
// If isRemoveDuplicates is false and there is no removeAt input pin,
// remove any one feature value equal to the input value (if there are any that are equal).
// If isRemoveDuplicates is false, and there is a removeAt input pin
// remove the feature value at that position.

RemoveStructuralFeatureValueAction action = (RemoveStructuralFeatureValueAction) (this.node);
StructuralFeature feature = action.getStructuralFeature();
Association association = this.getAssociation(feature);
Value value = this.takeTokens(action.getObject()).get(0);
Value inputValue = null;
if (action.getValue() != null) {
        // NOTE: Multiplicity of the value input pin is required to be 1..1.
        inputValue = this.takeTokens(action.getValue()).get(0);
}
int removeAt = 0;
if (action.getRemoveAt() != null) {
        removeAt = ((UnlimitedNaturalValue) this.takeTokens(action.getRemoveAt()).get(0)).value;
}
if (association != null) {
        List<Link> links = this.getMatchingLinksForEndValue(association, feature, value, inputValue);
        if (action.isRemoveDuplicates()) {
                for (int i = 0; i < links.size(); i++) {
                        Link link = links.get(i);
                        link.destroy();
                }
        } else if (action.getRemoveAt() == null) {
                // *** If there is more than one matching link,
                // non-deterministically choose one. ***
                if (links.size() > 0) {
                        int i = ((ChoiceStrategy)
this.getExecutionLocus().factory.getStrategy("choice")).choose(links.size());
                        links.get(i - 1).destroy();
                }
        } else {
```

```
                boolean notFound = true;
                int i = 1;
                while (notFound & i <= links.size()) {
                        Link link = links.get(i - 1);
                        if (link.getFeatureValue(feature).position == removeAt) {
                                notFound = false;
                                link.destroy();
                        }
                }
        }
} else if (value instanceof StructuredValue) {
        // If the value is a data value, then it must be copied before
        // any change is made.
        if (!(value instanceof Reference)) {
                value = value.copy();
        }
        FeatureValue featureValue = ((StructuredValue)
value).getFeatureValue(action.getStructuralFeature());
        List<Value> removedValues = new ArrayList<Value>();
        if (action.isRemoveDuplicates()) {
                int j = this.position(inputValue, featureValue.values, 1);
                while (j > 0) {
                        removedValues.add(featureValue.values.get(j - 1));
                        featureValue.values.remove(j - 1);
                        j = this.position(inputValue, featureValue.values, j);
                }
        } else if (action.getRemoveAt() == null) {
                List<Integer> positions = new ArrayList<Integer>();
                int j = this.position(inputValue, featureValue.values, 1);
                while (j > 0) {
                        positions.add(j);
                        j = this.position(inputValue, featureValue.values, j);
                }
                if (positions.size() > 0) {
                        // *** Nondeterministically choose which value to remove.
                        // ***
                        int k = ((ChoiceStrategy)
this.getExecutionLocus().factory.getStrategy("choice")).choose(positions.size());
                        removedValues.add(featureValue.values.get(positions.get(k - 1) - 1));
                        featureValue.values.remove(positions.get(k - 1) - 1);
                }
        } else {
                if (featureValue.values.size() >= removeAt) {
                        removedValues.add(featureValue.values.get(removeAt - 1));
                        featureValue.values.remove(removeAt - 1);
                }
        }
        // When values are removed from the list of values associated to the feature
        // (in the context of the target), these latter may be involved in links representing
        // instance of connectors. If this is the case, links in which the removed values are
        // involved are destroyed.
        for(int i = 0; i < removedValues.size(); i++){
                List<CS_Link> linkToDestroy = this.getLinksToDestroy((StructuredValue)value, feature,
removedValues.get(i));
                for(int j = 0; j < linkToDestroy.size(); j++){
                        linkToDestroy.get(j).destroy();
                }
        }
}
if (action.getResult() != null) {
        this.putToken(action.getResult(), value);
}

[2] public public getLinksToDestroy (value : StructuredValue, feature : StructuralFeature,
removedValue : Value) : CS_Link[*]

// Get all links that are required to be destroyed due to the removal of the removedValue.
List<CS_Link> linksToDestroy = new ArrayList<CS_Link>();
if (value instanceof CS_Reference) {
        CS_Reference context = (CS_Reference) value;
        // Retrieves the feature values for the structural feature associated with this action,
        // in the context of this reference
        if (feature instanceof Port) {
```

```
                    // The removed value is an interaction point.
                    // All links in which this interaction is involved must be destroyed.
                    CS_InteractionPoint interactionPoint = (CS_InteractionPoint) removedValue;
                    List<CS_Link> connectorInstances =
context.compositeReferent.getLinks(interactionPoint);
                    for (int j = 0; j < connectorInstances.size(); j++) {
                            CS_Link link = connectorInstances.get(j);
                            linksToDestroy.add(link);
                    }
            } else {
                    // Feature is not a Port. Search for all potential link
                    // ends existing in the context of this object.
                    List<Value> allValuesForFeature = new ArrayList<Value>();
                    for (int i = 0; i < context.referent.featureValues.size(); i++) {
                            StructuralFeature currentFeature =
context.referent.featureValues.get(i).feature;
                            if(feature != currentFeature){
                                    List<Value> values = this.getPotentialLinkEnds(context,
currentFeature);
                                    for (int j = 0; j < values.size(); j++) {
                                            allValuesForFeature.add(values.get(j));
                                    }
                            }
                    }
                    // Retrieves all links available at the locus
                    List<ExtensionalValue> extensionalValues = this.getExecutionLocus().extensionalValues;
                    List<CS_Link> allLinks = new ArrayList<CS_Link>();
                    for (int i = 0; i < extensionalValues.size(); i++) {
                            ExtensionalValue extensionalValue = extensionalValues.get(i);
                            if (extensionalValue instanceof CS_Link) {
                                    allLinks.add((CS_Link) extensionalValue);
                            }
                    }
                    // In the set of links involving potential link ends. Search for all
                    // links that involve the removed value in other end. Any link in that
                    // fulfill this condition is registered in the set of link to be destroyed.
                    for (int i = 0; i < allLinks.size(); i++) {
                            CS_Link link = allLinks.get(i);
                            boolean linkHasToBeDestroyed = false;
                            for (int j = 0; j < allValuesForFeature.size() && !linkHasToBeDestroyed; j++) {
                                    Value v = allValuesForFeature.get(j);
                                    StructuralFeature featureForV = link.getFeature(v);
                                    if (featureForV != null) {
                                            for (int k = 0; k < link.featureValues.size() && !
linkHasToBeDestroyed; k++) {
                                                    FeatureValue otherFeatureValue =
link.featureValues.get(k);
                                                    if (otherFeatureValue.feature != featureForV) {
                                                            for (int l = 0; l <
otherFeatureValue.values.size() && !linkHasToBeDestroyed; l++) {
                                                                    if(otherFeatureValue.values.get(l) ==
removedValue){
                                                                            linkHasToBeDestroyed = true;
                                                                    }
                                                            }
                                                    }
                                            }
                                    }
                            }
                            if (linkHasToBeDestroyed) {
                                    linksToDestroy.add(link);
                            }
                    }
            }
}
return linksToDestroy;

[3] public getPotentialLinkEnds ( context : CS_Reference, feature : StructuralFeature): Value[*]

// Get all links that are required to be destroyed due to the removal of the removedValue.
List<Value> potentialLinkEnds = new ArrayList<Value>();
FeatureValue featureValue = context.getFeatureValue(feature);
for (int i = 0; i < featureValue.values.size(); i++) {
```

```
      Value v = featureValue.values.get(i);
potentialLinkEnds.add(v);
        if (v instanceof CS_Reference) {
                // add all interaction points associated with v
                for (int j = 0; j < ((CS_Reference) v).referent.featureValues.size(); j++) {
                        if (((CS_Reference) v).referent.featureValues.get(j).feature instanceof Port) {
                                List<Value> interactionPoints = (((CS_Reference)
v).referent.featureValues.get(j)).values;
                                for (int k = 0; k < interactionPoints.size(); k++) {
                                        potentialLinkEnds.add(interactionPoints.get(k));
                                }
                        }
                }
        }
}
return potentialLinkEnds;
```

8.6.2.13 CS_SendSignalActionActivation

FUML semantics are extended to account for property onPort of the SendSignalAction. If onPort is not specified, fUML semantics are unchanged. If onPort is specified, instead of sending directly to the target reference by calling operation send (as in fUML):

- If the SendSignalAction is being executed inside the object given on the target InputPin, the invocation is made out of the target object through the given Port.

- Otherwise, the invocation is made into the target object through the given Port.

Generalizations

- SendSignalActionActivation (from fUML_Semantics::Semantics::Actions)

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 InvocationAction is being executed inside the object given
  // on the target InputPin, the invocation is made out of the target object
  // through the given Port.
  // - Otherwise the invocation is made into the target object through the
  // given Port.
  SendSignalAction action = (SendSignalAction)(this.node);
  if (action.onPort == null) {
    // Behaves like in fUML

    super.doAction () ;
  }
  else {
    Value target = this.takeTokens(action.target).get(0) ;
    if (target instanceof CS_Reference) {
      // Constructs the signal instance
      Signal signal = action.signal;

      SignalInstance signalInstance = new SignalInstance();
      signalInstance.type = signal;
```

```
      PropertyList attributes = signal.ownedAttribute ;
      InputPinList argumentPins = action.argument ;
      Integer i = 0 ;
      while ( i < attributes.size()) {
        Property attribute = attributes.get(i);
        InputPin argumentPin = argumentPins.get(i);
        ValueList values = this.takeTokens(argumentPin);
        signalInstance.setFeatureValue(attribute, values, 0);
        i = i + 1 ;
      }

      // Construct the signal event occurrence
      SignalEventOccurrence signalEventOccurrence = new SignalEventOccurrence();
      signalEventOccurrence.signalInstance = (SignalInstance) signalInstance.copy();

      CS_EventOccurrence wrappingEventOccurrence = new CS_EventOccurrence();
      wrappingEventOccurrence.wrappedEventOccurrence = signalEventOccurrence;
      // 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 ;
     Object_ executionContext = this.getActivityExecution().context ;
     if (executionContext == targetReference.referent ||
         targetReference.compositeReferent.contains(executionContext)) {
         wrappingEventOccurrence.sendOutTo(targetReference, action.onPort);
     }
     else {
         wrappingEventOccurrence.sendInTo(targetReference, action.onPort);
     }
    }
  }
}
```

8.7 Loci

8.7.1 Overview

The Loci package includes extensions to fUML Locus, ExecutionFactory and Executor, in order to account for new semantic visitors introduced by this specification (see Figure 8.16).
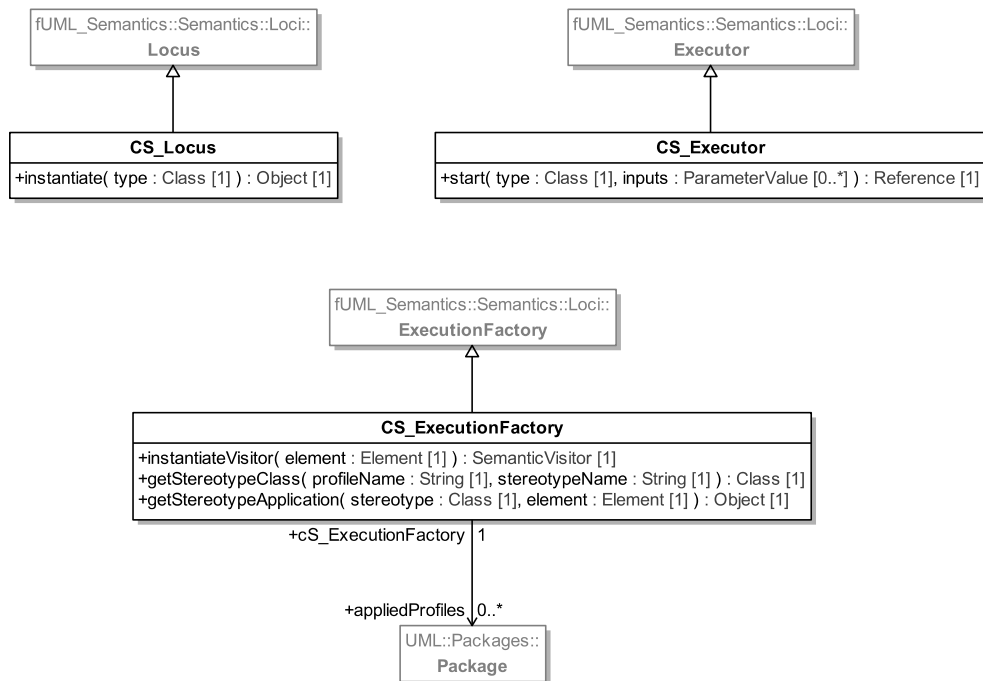
Figure 8.16 - Loci

8.7.2 Class Descriptions

8.7.2.1 CS_ExecutionFactory

CS_ExecutionFactory extends fUML ExecutionFactory to account for new semantic visitors introduced by this specification. It also includes mechanisms to deal with Profiles and Stereotype applications (property appliedProfiles, and operations getStereotypeClass and getStereotypeApplication). The principles underlying these mechanisms are described in 8.1, Dealing with Profiles and Stereotypes.

Generalizations

- ExecutionFactory (from fUML_Semantics::Semantics::Loci)

Attributes

- None

Associations

- appliedProfiles : Package [*], profiles containing any stereotype applied on elements being executed. Implementations shall initialize this collection before starting any execution.

Operations

```
[1] public getStereotypeClass(profileName:String, stereotypeName:String) : Classifier

[2] public getStereotypeApplication(stereotype:Class, element:Element) : Object
  ExtensionalValueList extent = locus.getExtent(stereotype);
  Object extensionObject = null;
  int i = 1;
  while (i <= extent.size() && extensionObject == null) {
    ExtensionalValue object = extent.getValue(i - 1);
    if(object.getFeatureValue(baseEnd).values.getValue(0).equals(element)) {
      extensionObject = (Object)object;
    }
    i = i + 1;
  }
  return extensionObject ;

[3] 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 ClearStructuralFeatureAction) {
            visitor = new CS_ClearStructuralFeatureValueActionActivation() ;
      }
      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 AcceptCallAction) {
            visitor = new CS_AcceptCallActionActivation();
}
      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 if (element instanceof OpaqueExpression) {
              visitor = new CS_OpaqueExpressionEvaluation() ;
        }
        else if (element instanceof RemoveStructuralFeatureValueAction) {
              visitor = new CS_RemoveStructuralFeatureValueActionActivation();
        }
        else if (element instanceof CallEventBehavior) {
              visitor = new CS_CallEventExecution();
        }
        else {
              visitor = super.instantiateVisitor(element) ;
        }
        return visitor ;
```

8.7.2.2 CS_Executor

FUML semantics are extended so that, when the start operation produces a CS_Object, a CS_Reference is returned instead of a fUML Reference.

Generalizations

- Executor (from ffUML_Semantics::Semantics::Loci)

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;
```

8.7.2.3 CS_Locus

FUML semantics are extended so that the instantiate operation produces a CS_Object instead of an Object, in the case where the Class to be instantiated is not a Behavior.

Generalizations

- Locus (from fUML_Semantics::Semantics::Loci)

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 Suites

9.1 Overview

This clause describes suites of test cases that can be used to demonstrate conformance to this specification. All test cases are included in a single UML model, complying with syntax defined in Clause 7 of this specification. This clause is decomposed into four test suites, each one focusing on a specific aspect of UML composite structures semantics.

Test Suite 1 focuses on semantics of instantiation, and validates topologies resulting from the instantiation of composite structures, as well as management of default values for properties. Test Suite 2 focuses on communication aspects, such as propagation of requests across ports, through connectors. Test Suite 3 also focuses on communication aspects, in the specific cases where property onPort of InvocationAction is used, and when reactions may occur only if a message was received on a specific port. Test Suite 4 finally focuses on destruction semantics of UML Composite Structures, such as the recursive destruction of parts and ports, and automatic deletion of links (representing connector instances) where connected roles are removed.

Test cases consist in combinations of structural and behavioral elements, whose executions shall verify some assertions. Assertions are specified using basic functions specified in a library, which is itself an executable UML model complying syntax defined in Clause 7 of this specification. A tool may demonstrate compliance with this specification if no assertion fails when executing the test cases. In the following subclauses, structural aspects are depicted with appropriate diagrams, while behavioral aspects are usually depicted in Alf. Exceptions include cases where extensions to Alf would be required to specify behavioral aspects (such as in Test Suite 3). In these cases, Activity diagrams are used to specify behaviors.

The architecture of the Test Suite is depicted in Figure 9.1. Subclause 9.2 describes the assertion library used across all test cases (package AssertionLibrary in the upper part of Figure 9.1). Subclauses 9.3, 9.4, 9.5 and 9.6 describe the various test suites (packages Test Suite 1, Test Suite 2, Test Suite 3, and Test Suite 4, on the left part of Figure 9.1), as well as variants of these Test Suites (all packages with suffix bis, ter, and quater, in Figure 9.1).
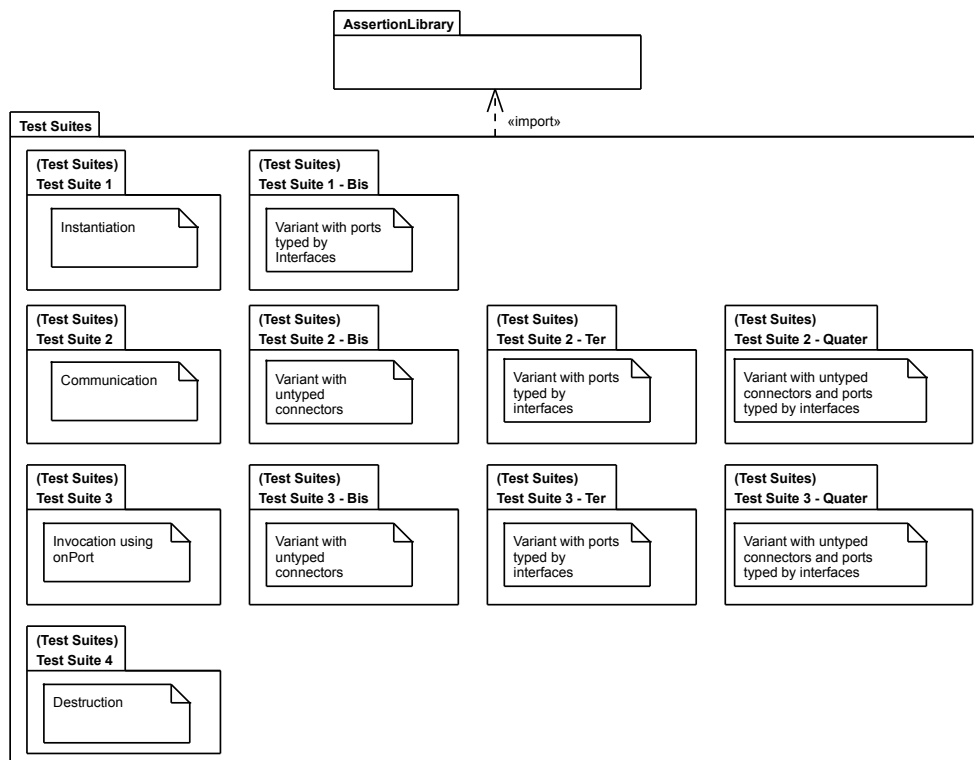
Figure 9.1 - Test Suites Architecture Diagram

9.2 Assertion library

The assertion library includes five activities, specified in Alf as follows. These activities are the basis for all test cases included in this specification.

```
[1] activity Write(in value: any) {
  // Writes value on the standard output channel
  // StandardOutputChannel is part of the foundational
  // model library, which is part of the fUML specification
  StandardOutputChannel.allInstances()[1].write(value);
}

[2] activity AssertTrue(in label: String, in condition: Boolean) {
  // Asserts that the given condition is true
  WriteLine(label + ": " + ToString(condition));
}

[3] activity AssertFalse(in label: String, in condition: Boolean) {
  // Asserts that the given condition is false
  WriteLine("! " + label + ": " + ToString(!condition));
}

[4] activity AssertEqual(in label: String, in value1: any, in value2: any) {
  // Asserts equality between value1 and value2
  Write(label + "==");
  Write(value2 instanceof String? "\"" + (String)value2 + "\"": value2);
  AssertTrue("", value1==value2);
}

[5] activity AssertList(in label: String, in list: any[*] sequence, in expected: any[*] sequence) {
  // Asserts that lists list and expected are equals
  // (i.e., same size, and same content)
  AssertEquals(label + "->size()", list->size(), expected->size());
  for (i in 1..IntegerFunctions::Min(list->size(),expected->size())) {
    AssertEquals(label + "[" + IntegerFunctions::ToString(i) + "]", list[i], expected[i]);
  }
}
```

9.3 Test Suite 1: Instantiation

This test suite focuses on instantiation semantics of UML Composite Structures. It validates topologies resulting from the instantiation of composite structures, as well as management of default values for properties. Subclause 9.3.1 describes a utility class and a couple of utility activities used by test cases defined in this subclause. Subclauses 9.3.2, 9.3.3 and 9.3.4 include test cases implying assembly connectors. Subclauses 9.3.5 and 9.3.6 deal with management of default values. Subclauses 9.3.7 and 9.3.8 include test cases implying delegation connectors. Subclause 9.3.9 finally deals with instantiation of a structure with multiple hierarchy levels. Subclause 9.3.10 describes a variant of Test Suite 1 with ports typed by interfaces.

9.3.1 Utilities

Utilities includes four assertion activities, each one corresponding to a topology supported by construction strategy CS_DefaultConstructStrategy, specified in 8.6.2.9. The Alf specification for these activities is as follows:

```
[1] activity AssertEmptyPattern(in ends1 : any[], in ends2 : any[]) {
  // Asserts empty topology
  // ends1 and ends2 shall be empty
  AssertEquals("EmptyPattern - ends1->size()", ends1->size(), 0) ;
  AssertEquals("EmptyPattern - ends2->size()", ends2->size(), 0) ;
}

[2] activity AssertUnconnectedPattern(in ends1 : any[], in ends2 : any[],
                                      in n1 : Integer, in n2 : Integer,
                                      in helper : AbstractAssertCompositeHelper) {
  // Asserts unconnected topology
  // Check cardinalities
  // Size of ends1 must be equal to expected size n1
  // Size of ends2 must be equal to expected size n2
  AssertEquals("UnconnectedPattern - ends1->size()", ends1->size(), n1) ;
```

```
      AssertEquals("UnconnectedPattern - ends2->size()", ends2->size(), n2) ;
      // Checks connections
      // There shall not be any connection between elements of ends1 and elements of ends2
      for (i in 1..ends1->size()) {
        for (j in 1..ends2->size()) {
          helper.assertNotConnected("UnconnectedPattern - ends1["
                                    + ToString(i)
                                    + "] is not connected to ends2["
                                    + ToString(j) + "]",
                                    ends1[i], ends2[j]) ;
        }
      }
    }

[3] activity AssertArrayPattern(in ends1 : any[], in ends2 : any[], in n : Integer,
                                  in helper : AbstractAssertCompositeHelper) {
      // Asserts array topology
      // Check cardinalities
      // Size of ends1 must be equal to expected size n
      // Size of ends2 must be equal to expected size n
      AssertEquals("ArrayPattern - ends1->size()", ends1->size(), n) ;
      AssertEquals("ArrayPattern - ends2->size()", ends2->size(), n) ;
      upper = Min(ends1->size(), ends2->size()) ;
      // Check connections
      // Each element of ends1 must be connected to one and only
      // one element of ends2, with same index.
      for (i in 1..upper) {
        for (j in 1..upper) {
          if (i == j) {
            helper.assertConnected("ArrayPattern - ends1["
                                   + ToString(i)
                                   + "] is connected to ends2["
                                   + ToString(j) + "]",
                                   ends1[i], ends2[j]) ;
          }
          else {
            helper.assertNotConnected("ArrayPattern - ends1["
                                      + ToString(i)
                                      + "] is not connected to ends2["
                                      + ToString(j) + "]",
                                      ends1[i], ends2[j]) ;
          }
        }
      }
    }

[4]activity AssertStarPattern(in ends1 : any[], in ends2 : any[],
                                  in n1 : Integer, in n2 : Integer,
                                  in helper : AbstractAssertCompositeHelper) {
      // Asserts star topology
      // Check cardinalities
      // Size of ends1 must be equal to expected size n1
      // Size of ends2 must be equal to expected size n2
      AssertEquals("StarPattern - ends1->size()", ends1->size(), n1) ;
      AssertEquals("StarPattern - ends2->size()", ends2->size(), n2) ;
      // Checks connections
      // Each element of ends1 must be connected to
      // all elements of ends2, and vice-versa.
      for (i in 1..ends1->size()) {
        for (j in 1..ends2->size()) {
          helper.assertConnected("StarPattern - ends1["
                                 + ToString(i)
                                 + "] is connected to ends2["
                                 + ToString(j) + "]",
                                 ends1[i], ends2[j]) ;
        }
      }
    }
```

All these activities have an input parameter typed by the abstract class AbstractAssertCompositeHelper. This abstract class is extended in following subclauses, to determine if connections exist between two elements. These specific realizations imply Associations defined in corresponding test cases. Figure 9.2 illustrates the specification of this class

where, for both operations assertConnected and assertNotConnected, parameter message is typed by String, and parameters end1 and end2 are not typed.

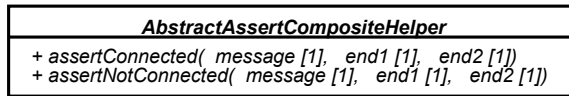**Figure 9.2 - AbstractAssertCompositeHelper**

9.3.2 Assembly connector between parts

This test case addresses instantiation semantics in the case of assembly connectors between parts. Structural aspects of this test case are depicted in Figure 9.3 and Figure 9.4.
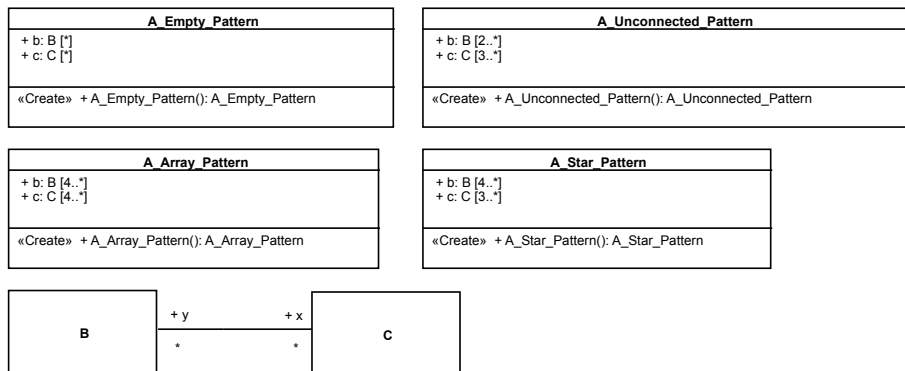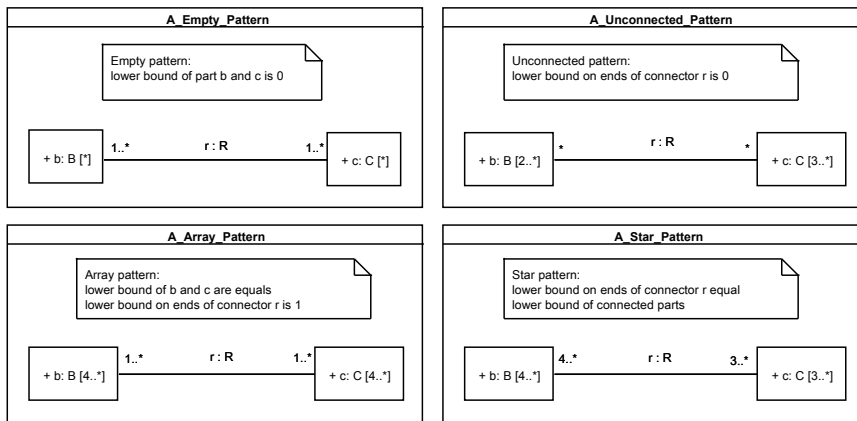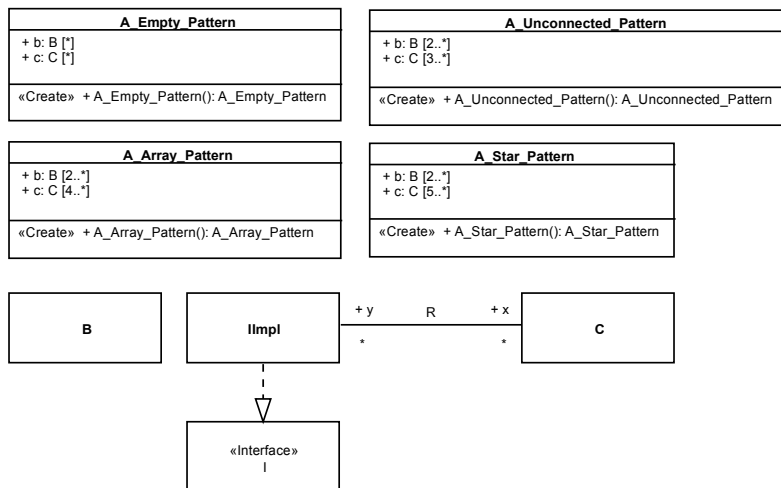

Figure 9.3 - Assembly Connector between parts – Classes
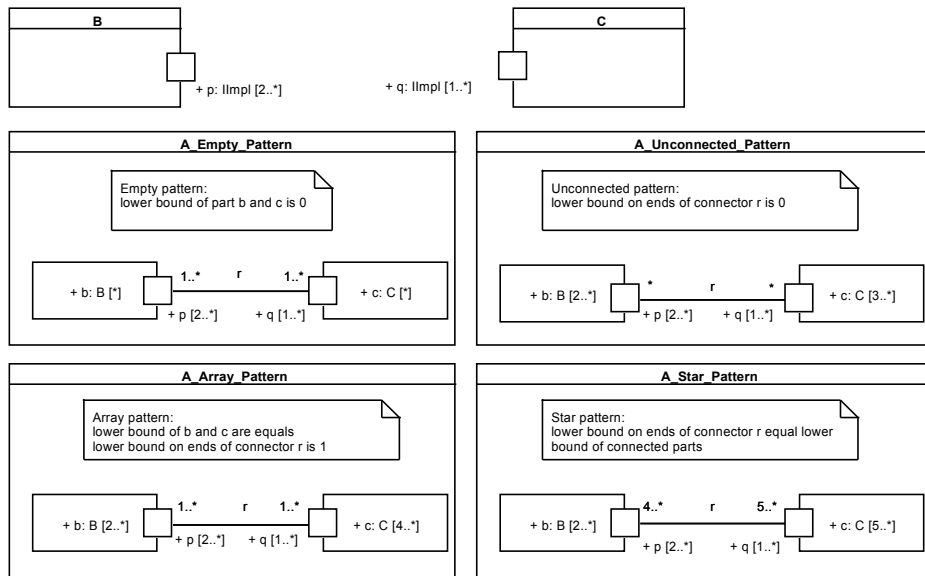

Figure 9.4 - Assembly Connector between parts – Internal structures

Corresponding test case behavior is:

```
activity TestCase_Assembly_P_P() {
  helper = new AssertCompositeHelper() ;
  WriteLine("-- Running test case: Assembly connector between two parts --") ;

  // Testing instantiation of A_Empty_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Empty_Pattern -") ;
  a_empty = new A_Empty_Pattern();
  AssertEmptyPattern(a_empty.b, a_empty.c) ;

  // Testing instantiation of A_Unconnected_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Unconnected_Pattern -") ;
```

```
  a_unconnected = new A_Unconnected_Pattern();
  AssertUnconnectedPattern(a_unconnected.b, a_unconnected.c, 2, 3, helper) ;

  // Testing instantiation of A_Array_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Array_Pattern") ;
  a_array = new A_Array_Pattern();
  AssertArrayPattern(a_array.b, a_array.c, 4, helper) ;

  // Testing instantiation of A_Star_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Star_Pattern") ;
  a_star = new A_Star_Pattern();
  AssertStarPattern(a_star.b, a_star.c, 4, 3, helper) ;

  WriteLine("-- End of test case --") ;
}
```

9.3.3 Assembly connector between a part with port and a part

This test case addresses instantiation semantics in the case of an assembly connector between a part with port and a part. Structural aspects of this test case are depicted in Figure 9.5 and Figure 9.6.

Figure 9.5 - Assembly Connector between part with port and part – Classes
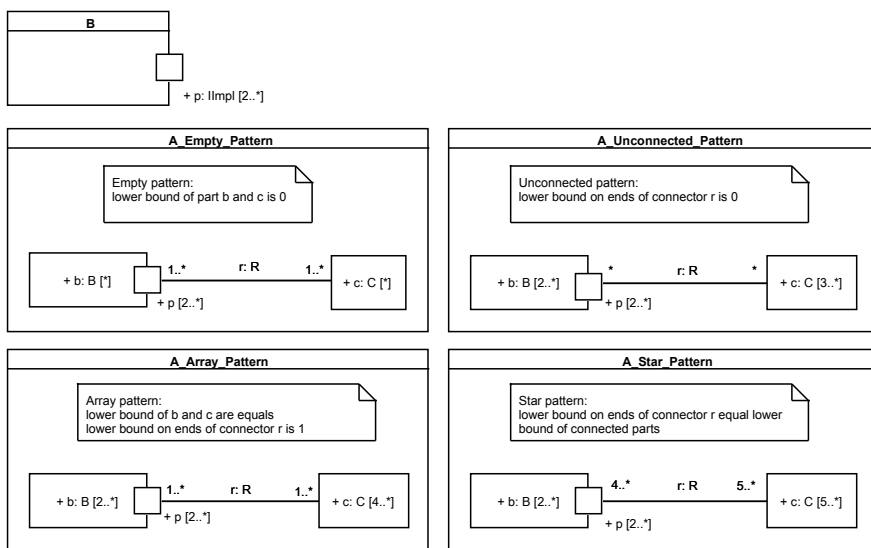
Figure 9.6 - Assembly Connector between part with port and part – Internal structures

Corresponding test case behavior is:

```
activity TestCase_Assembly_PWP_P() {
  helper = new AssertCompositeHelper() ;

  WriteLine("-- Running test case: Assembly connector between a part with port and a part --") ;

  // Testing instantiation of A_Empty_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Empty_Pattern -") ;
  a_empty = new A_Empty_Pattern() ;
  AssertEmptyPattern(a_empty.b, a_empty.c) ;

  // Testing instantiation of A_Unconnected_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Unconnected_Pattern -") ;
  a_unconnected = new A_Unconnected_Pattern();
  AssertUnconnectedPattern(a_unconnected.b.p, a_unconnected.c, 4, 3, helper) ;

  // Testing instantiation of A_Array_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Array_Pattern -") ;
  a_array = new A_Array_Pattern();
  AssertArrayPattern(a_array.b.p, a_array.c, 4, helper) ;

  // Testing instantiation of A_Star_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Star_Pattern -") ;
  a_star = new A_Star_Pattern();
  AssertStarPattern(a_star.b.p, a_star.c, 4, 5, helper) ;
}
```

9.3.4 Assembly connector between a part with port and a part with port

This test case addresses instantiation semantics in the case of an assembly connector between a part with port and a part with port. Structural aspects of this test case are depicted in Figure 9.7 and Figure 9.8.
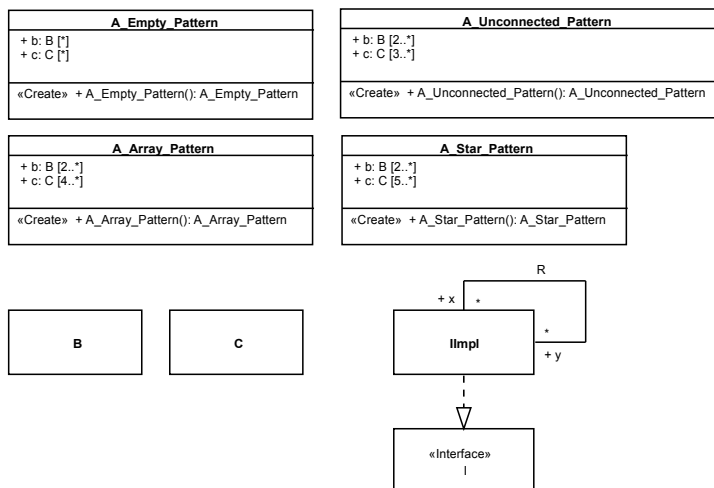
Figure 9.7 - Assembly Connector between part with port and part with port – Classes

B

+ p: IImpl [2..*]

C

+ q: IImpl [1..*]

A_Empty_Pattern

Empty pattern:
lower bound of part b and c is 0

+ b: B [*]   1..*   r   1..*   + c: C [*]
+ p [2..*]   + q [1..*]

A_Unconnected_Pattern

Unconnected pattern:
lower bound on ends of connector r is 0

+ b: B [2..*]   *   r   *   + c: C [3..*]
+ p [2..*]   + q [1..*]

A_Array_Pattern

Array pattern:
lower bound of b and c are equals
lower bound on ends of connector r is 1

+ b: B [2..*]   1..*   r   1..*   + c: C [4..*]
+ p [2..*]   + q [1..*]

A_Star_Pattern

Star pattern:
lower bound on ends of connector r equal lower
bound of connected parts

+ b: B [2..*]   4..*   r   5..*   + c: C [5..*]
+ p [2..*]   + q [1..*]

Figure 9.8 - Assembly Connector between part with port and part with port – Internal structures

Corresponding test case behavior is:

```
activity TestCase_Assembly_PWP_PWP () {
  helper = new AssertCompositeHelper() ;

  WriteLine("-- Running test case: Assembly connector between a part with port
            and a part with port --") ;

  // Testing instantiation of A_Empty_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Empty_Pattern -") ;
  a_empty = new A_Empty_Pattern() ;
  AssertEmptyPattern(a_empty.b, a_empty.c) ;

  // Testing instantiation of A_Unconnected_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Unconnected_Pattern -") ;
  a_unconnected = new A_Unconnected_Pattern();
  AssertUnconnectedPattern(a_unconnected.b.p, a_unconnected.c.q, 4, 3, helper) ;

  // Testing instantiation of A_Array_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Array_Pattern -") ;
  a_array = new A_Array_Pattern();
  AssertArrayPattern(a_array.b.p, a_array.c.q, 4, helper) ;

  // Testing instantiation of A_Star_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Star_Pattern -") ;
  a_star = new A_Star_Pattern();
  AssertStarPattern(a_star.b.p, a_star.c.q, 4, 5, helper) ;
}
```

9.3.5 Default values for basic types

This test case addresses instantiation of default values for properties typed by primitive types and enumerations.
Structural aspects of this test case are depicted in Figure 9.9.
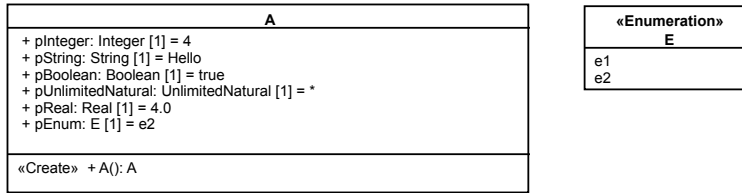
Figure 9.9 - Default values for basic types

Corresponding test case behavior is:

```
activity 'Test Case - Default Values'() {
  WriteLine("-- Running test case - Instantiation of Default values - basic types --") ;
  a = new A() ;
  AssertEquals("Default value of a.pInteger", a.pInteger, 4) ;
  AssertEquals("Default value of a.pString", a.pString, "Hello") ;
  AssertEquals("Default value of a.pBoolean", a.pBoolean, true) ;
  AssertEquals("Default value of a.pUnlimitedNatural", a.pUnlimitedNatural, *) ;
  AssertEquals("Default value of a.pEnum", a.pEnum, E::e2) ;
  WriteLine("-- End of test case --") ;
}
```

9.3.6 Default values for structures

This test case addresses instantiation of default values for properties typed by composite structures. Structural aspects of this test case are depicted in Figure 9.10.
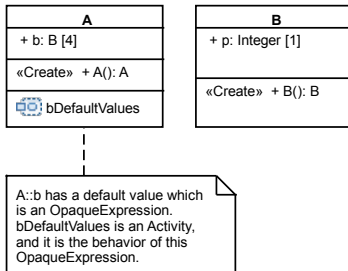
Figure 9.10 - Default values for structures

Detailed behavior of Activity bDefaultValues is:

```
activity bDefaultValues() : B[]{
        b1 = new B() ; b1.p = 1 ;
        b2 = new B() ; b2.p = 2 ;
        b3 = new B() ; b3.p = 3 ;
        b4 = new B() ; b4.p = 4 ;
        return new B[]{b1, b2, b3, b4} ;
}
```

Corresponding test case behavior is:

```
activity 'Test Case - Default Values - Structures'() {
  WriteLine("-- Running test case - Instantiation of Default values - Structures --") ;
  a = new A() ;
  AssertEquals("Default value of a.b[1].p", a.b[1].p, 1) ;
  AssertEquals("Default value of a.b[2].p", a.b[2].p, 2) ;
  AssertEquals("Default value of a.b[3].p", a.b[3].p, 3) ;
  AssertEquals("Default value of a.b[4].p", a.b[4].p, 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.3.7 Delegation between a port and a part

This test case addresses instantiation semantics in the case of a delegation connector between a port and a part. Structural aspects of this test case are depicted in Figure 9.11 and Figure 9.12.
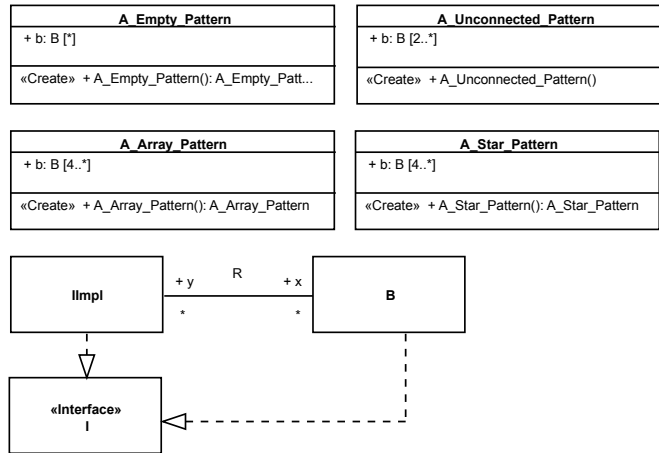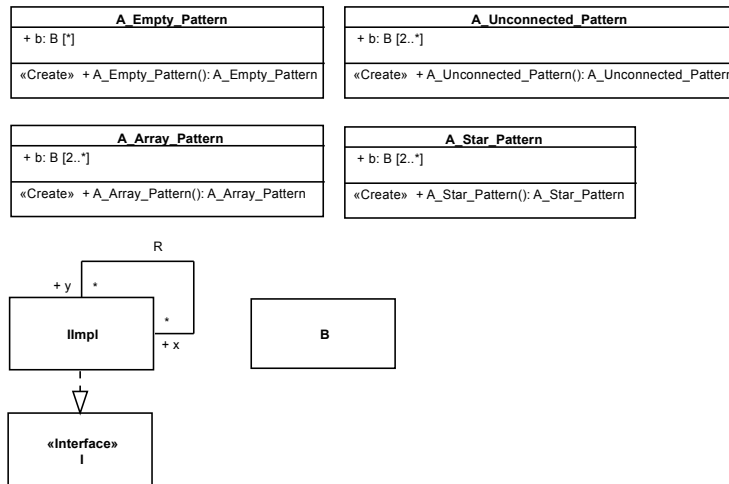

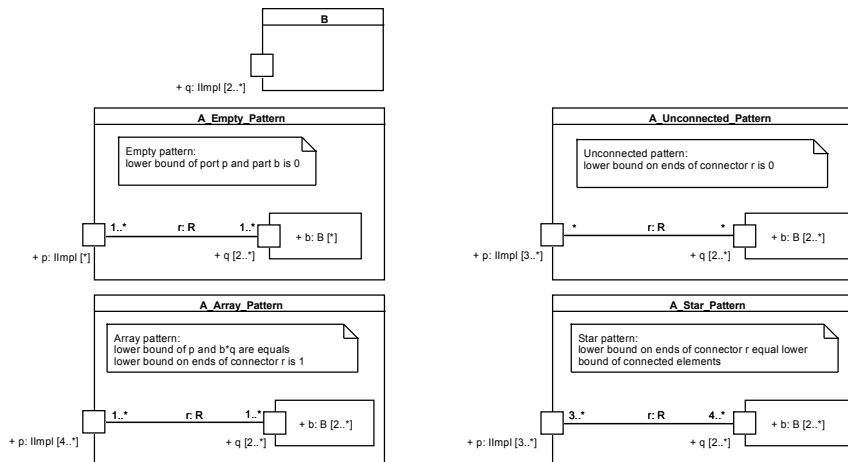
Figure 9.11 - Delegation between port and part – Classes
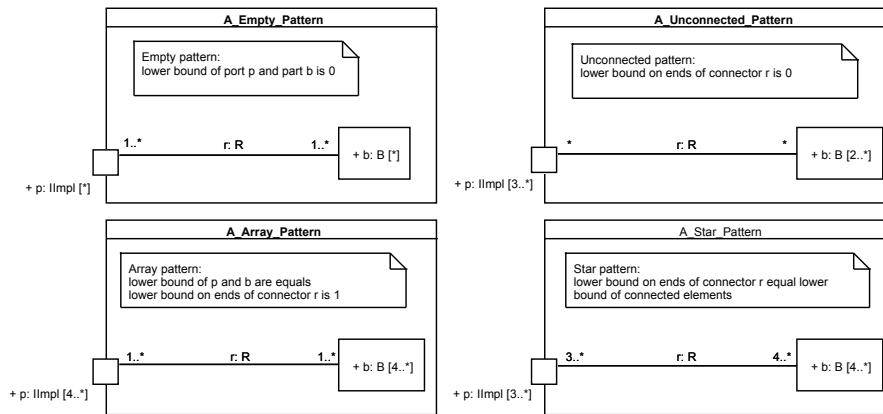
Figure 9.12 - Delegation between port and part – Internal structures

Corresponding test case behavior is:

```
activity TestCase_Delegation_Port_P() {
  helper = new AssertCompositeHelper() ;

  WriteLine("-- Running test case: Delegation connector between a port and a part --") ;

  a = new A_Array_Pattern() ;

  // Testing instantiation of A_Empty_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Empty_Pattern -") ;
  a_empty = new A_Empty_Pattern();
  AssertEmptyPattern(a_empty.p, a_empty.b) ;

  // Testing instantiation of A_Unconnected_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Unconnected_Pattern -") ;
  a_unconnected = new A_Unconnected_Pattern();
  AssertUnconnectedPattern(a_unconnected.p, a_unconnected.b, 3, 2, helper) ;

  // Testing instantiation of A_Array_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Array_Pattern") ;
  a_array = new A_Array_Pattern();
```

```
    AssertArrayPattern(a_array.p, a_array.b, 4, helper) ;

    // Testing instantiation of A_Star_Pattern
    WriteLine("") ;
    WriteLine("- Testing instantiation of A_Star_Pattern") ;
    a_star = new A_Star_Pattern();
    AssertStarPattern(a_star.p, a_star.b, 3, 4, helper) ;

}
```

9.3.8 Delegation between a port and a part with port

This test case addresses instantiation semantics in the case of a delegation connector between a port and a part with port.
Structural aspects of this test case are depicted in Figure 9.13 and Figure 9.14.

Figure 9.13 - Delegation between a port and a part with port – Classes

Figure 9.14 - Delegation between a port and part with port – Internal structures

Corresponding test case behavior is:

```
activity TestCase_Assembly_Port_PWP() {
  helper = new AssertCompositeHelper() ;

  WriteLine("-- Running test case: Delegation connector between a port and a part with port --") ;

  // Testing instantiation of A_Empty_Pattern
  WriteLine("") ;
  WriteLine("- Testing instantiation of A_Empty_Pattern -") ;
  a_empty = new A_Empty_Pattern();
```

```
    AssertEmptyPattern(a_empty.p, a_empty.b) ;

    // Testing instantiation of A_Unconnected_Pattern
    WriteLine("") ;
    WriteLine("- Testing instantiation of A_Unconnected_Pattern -") ;
    a_unconnected = new A_Unconnected_Pattern();
    AssertUnconnectedPattern(a_unconnected.p, a_unconnected.b.q, 3, 4, helper) ;

    // Testing instantiation of A_Array_Pattern
    WriteLine("") ;
    WriteLine("- Testing instantiation of A_Array_Pattern -") ;
    a_array = new A_Array_Pattern();
    AssertArrayPattern(a_array.p, a_array.b.q, 4, helper) ;

    // Testing instantiation of A_Star_Pattern
    WriteLine("") ;
    WriteLine("- Testing instantiation of A_Star_Pattern -") ;
    a_star = new A_Star_Pattern();
    AssertStarPattern(a_star.p, a_star.b.q, 3, 4, helper) ;

}
```

9.3.9 Hierarchy

This test case addresses instantiation a composite structure with multiple hierarchy levels. Structural aspects of this test case are depicted in Figure 9.15.
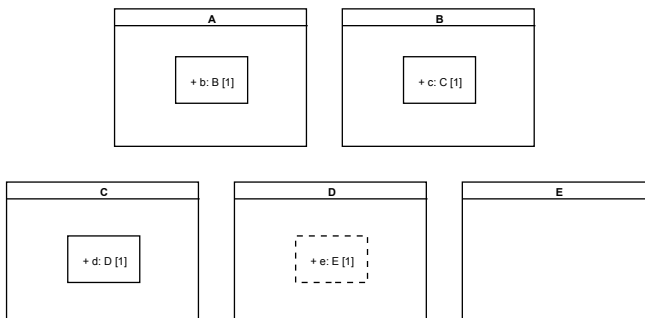
Figure 9.15 - Hierarchy

Corresponding test case behavior is:

```
activity TestCase_Hierarchy() {
  WriteLine("-- Running test case: Hierarchical instantiation --") ;
  a = new A() ;
  // a.b is not empty
  WriteLine("") ;
  AssertTrue("a.b is not empty", ListSize(a.b) > 0) ;
  // a.b.c is not empty
  AssertTrue("a.b.c is not empty", ListSize(a.b.c) > 0) ;
  // a.b.c.d is not empty
  AssertTrue("a.b.c.d is not empty", ListSize(a.b.c.d) > 0) ;
  // a.b.c.d.e is empty (e is shared, not composite)
  AssertTrue("a.b.c.d.e is empty", ListSize(a.b.c.d.e) == 0) ;
}
```

9.3.10 Variants of Test Suite 1

The normative Test Suite model contains a variant of Test Suite 1 (package Test Suite 1 – Bis, depicted in Figure 9.1), which contains variants of test cases 9.3.3, 9.3.4, 9.3.7, and 9.3.8. In these variants, ports are typed by Interfaces.

9.4 Test Suite 2: Communication

This test suite focuses on communication semantics of UML Composite Structures. Subclause 9.4.1 addresses reception of signals on behavior ports. Subclause 9.4.3 addresses loss of messages in case of operation calls received on a non-behavior port without delegation connectors. Subclauses 9.4.4, 9.4.5, 9.4.6, and 9.4.7 address forwarding of messages in

case of a single delegation link. Subclauses 9.4.8, 9.4.9, 9.4.10, and 9.4.11 address forwarding of messages across multiple links, corresponding to a single connector. Finally, subclauses 9.4.12, 9.4.13, 9.4.14, and 9.4.15 address forwarding of messages across multiple links, each link corresponding to a particular connector. Subclause 9.4.16 describes variants of Test Suite 2 where connectors are not typed and/or ports are typed by interfaces.

9.4.1 BehaviorPort – Signal

This test case addresses reception of a signal on a behavior port. If the signal is correctly dispatched to the classifier behavior of the receiving object, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.16 and Figure 9.17.
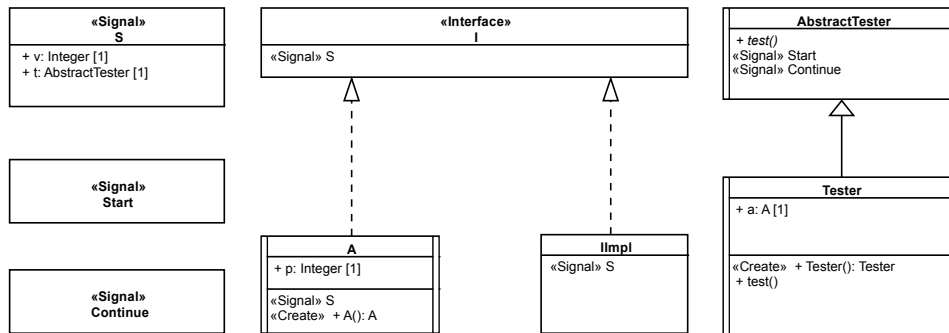
Figure 9.16 - Behavior port Signal – Classes, Signals and Interfaces
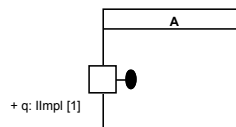
Figure 9.17 - Behavior port Signal – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - BehaviorPort - Signal'() {
  // Instantiate Tester
  t = new Tester() ;
  t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
  accept(Start) ;
  this.a.q.S(4, this) ;
  accept(Continue) ;
  this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
  WriteLine("-- Running test case: Reception of a Signal on a behavior port --") ;
  AssertTrue("Signal delegated to classifier behavior", this.a.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

Classifier behavior of A is:

```
activity AClassifierBehavior() {
  accept (s : S) {
    this.p = s.v ;
    s.t.Continue() ;
  }
}
```

9.4.2 BehaviorPort – Operation

This test case addresses the reception of an operation call on a behavior port. If the operation call is correctly dispatched and the operation that is called has no implementation a call event occurrence is placed in the event pool of the active object which owns the behavior port. The caller is suspended until the receiver replies to the call. The receiver reply is the only possibility of the caller to return from the call.

Figure 9.18 shows the classes involved in the case.
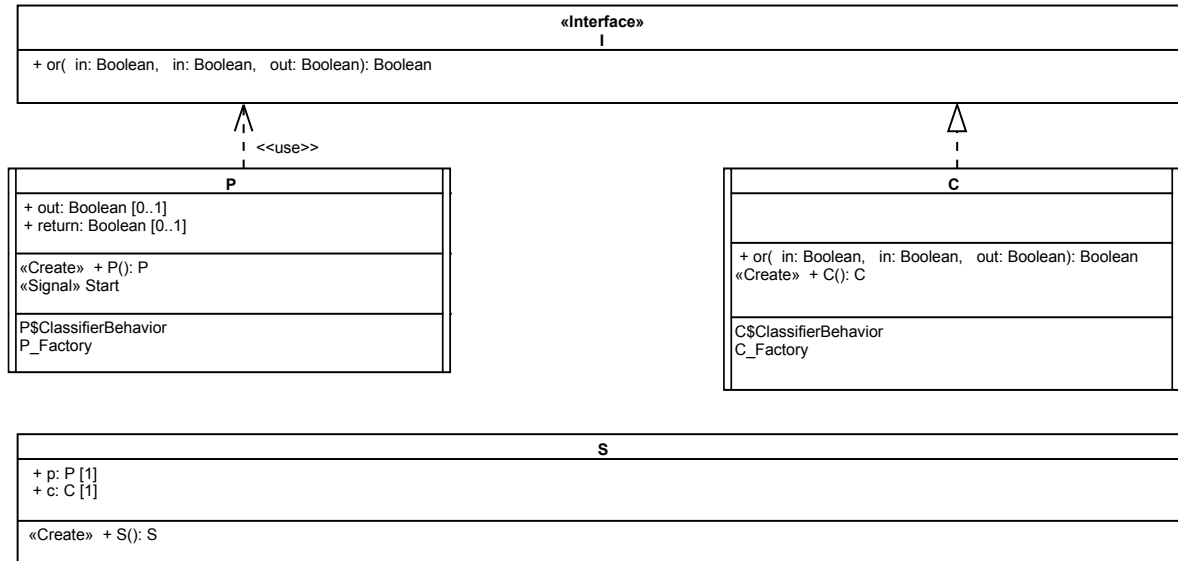
Figure 9.18 - Classes involved in test case BehaviorPort – Operation

Figure 9.19 shows the internal structure of class S.
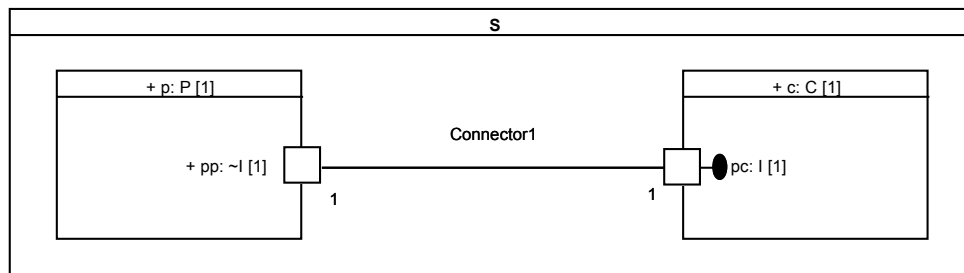
Figure 9.19 - Internal structure of Class S

Figure 9.20 shows the classifier behavior of class P.
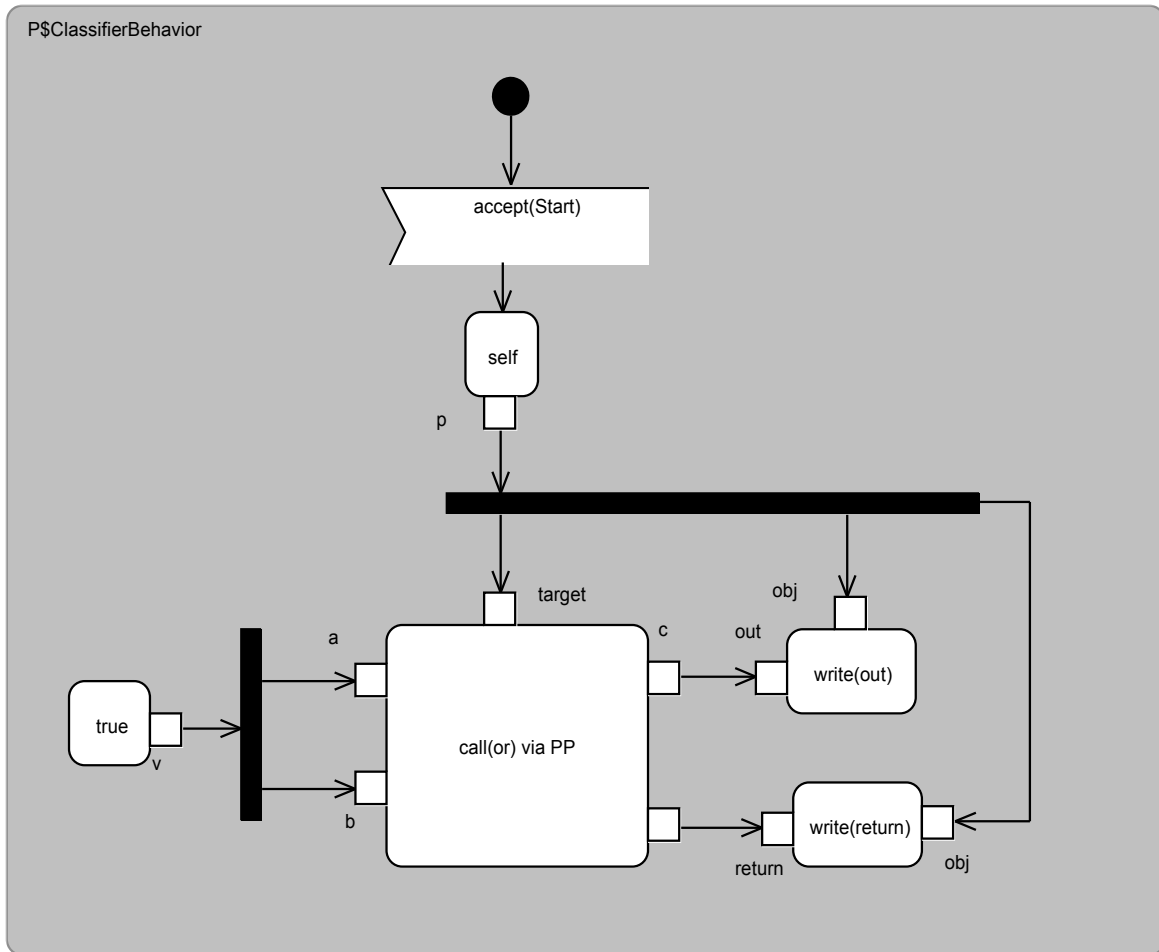
Figure 9.20 - Classifier behavior of Class P

The principle of the test case is the following: S is instantiated which implies that both P and C are also instantiated and their classifier behaviors are started. At this point P and C are respectively waiting for a Start event occurrence and a call event occurrence for the operation or. When then Start event occurrence is accepted by P, an operation or is called in synchronous manner through the port pp. When the operation call is dispatched through the behavior port pc, it results in call event occurrence for or operation to be added in C event pool. At this point P is suspended until the receiver returns from the call. When C accepts the call event occurrence for or it executes the behavior corresponding to the xor operation and replies to the caller using reply action. This enables the caller to continue its execution and to assign attributes out and return with the values produced by the operation call. If both attribute have the value false after the assignment then the test succeeds.

This test case is captured by the following specification:

```
activity 'Test case - Behavior Port - Operation'() {
      WriteLine("-- Running Test Case: Behavior Port - Operation --") ;
      S s = new S();
      s.p.Start();
      AssertTrue("sp.out == false", s.p.'out' == false);
      AssertTrue("sp.return == false", s.p.'return' == false);
      WriteLine("-- End of test case --") ;
}
```

9.4.3 Loss of Messages – Operation

This test case addresses loss of an operation call on non-behavior port with no delegation connector. If the operation call is not forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.21 and Figure 9.22.
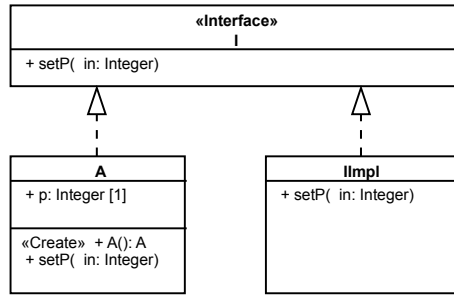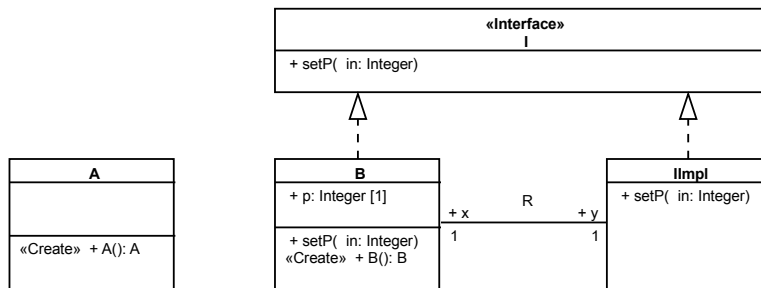
Figure 9.21 - Loss of messages – Classes and Interfaces
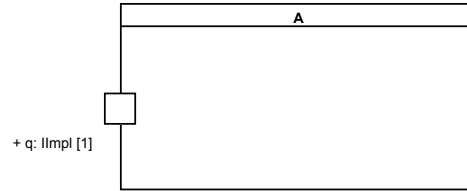
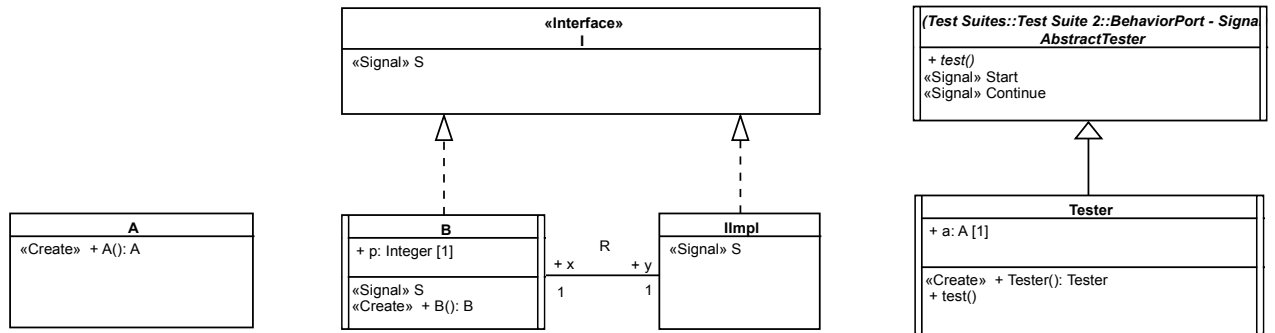**Figure 9.22 - Loss of messages – Internal structures**

Corresponding test case behavior is:

```
activity 'Test case - Loss of Messages - Operation'() {
  WriteLine("-- Running test case: Loss of Operation call on
            non-behavior ports with no delegation connector --") ;
  a = new A() ;
  a.q.setP(4) ;
  AssertFalse("Operation call received", a.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

Method of A::setP is:

```
activity setP(in v : Integer) {
      this.p = v ;
}
```

9.4.4 Single Delegation – PortToPart – Operation

This test case addresses forwarding of an operation through a single delegation link, corresponding to a single connector. If the operation call is forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.23 and Figure 9.24.

Figure 9.23 - Single Delegation from Port to Part (Operation) – Classes and Interfaces
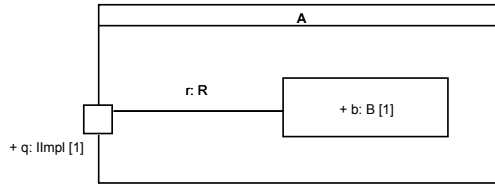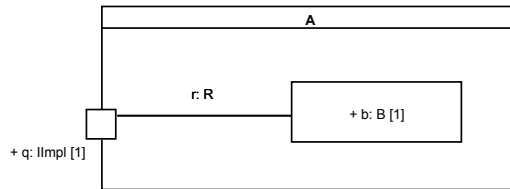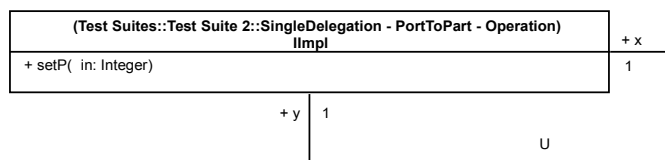
Figure 9.24 - Single Delegation from Port to Part (Operation) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - SingleDelegation - PortToPart - Operation'() {
  WriteLine("-- Running test case: Single delegation connector -
          Operation delegated from port to part --") ;
  a = new A() ;
  a.q.setP(4) ;
  AssertTrue("Operation call delegated", a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

Method of B::setP is:

```
activity setP(in v : Integer) {
  this.p = v ;
}
```

9.4.5 Single Delegation – PortToPart – Signal

This test case addresses forwarding of a signal through a single delegation link, corresponding to a single connector. If the signal is correctly forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.25 and Figure 9.26.

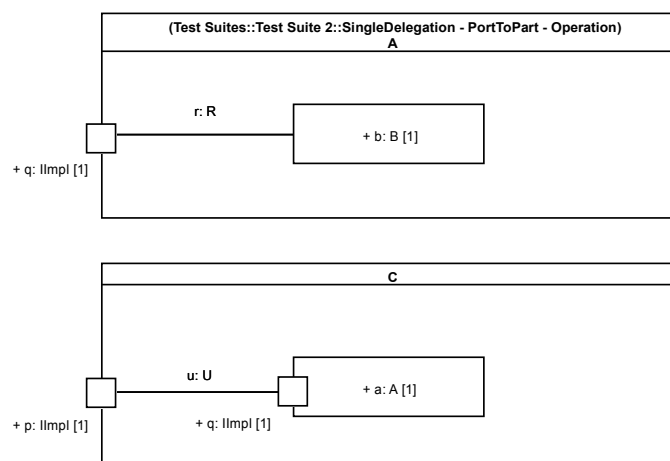**Figure 9.25 - Single Delegation from Port to Part (Signal) – Classes, Signals and Interfaces**

Figure 9.26 - Single Delegation from Port to Part (Signal) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - BehaviorPort - Signal'() {
  // Instantiate Tester
  t = new Tester() ;
  t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
  accept(Start) ;
  this.a.q.S(4, this) ;
  accept(Continue) ;
  this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
  WriteLine("-- Running test case: Single delegation connector -
            Signal delegated from port to part --") ;
  AssertTrue("Signal delegated to classifier behavior", this.a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

Classifier behavior of B is:

```
activity BClassifierBehavior() {
  accept (s : S) {
    this.p = s.v ;
    s.t.Continue() ;
  }
}
```

9.4.6 Single Delegation – PortToPartWithPort – Operation

This test case addresses forwarding of an operation call through a single delegation link, corresponding to a single connector, expressed between a port and a part with port. If the operation call is forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.27 and Figure 9.28. Note that classes A and B are reused from 9.4.4.
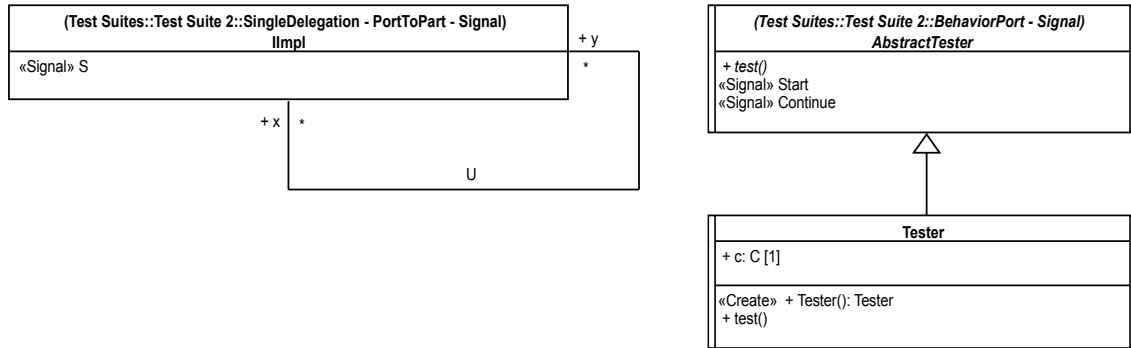

Figure 9.27 - Single Delegation from Port to Part with Port (Operation) – Classes and Interfaces
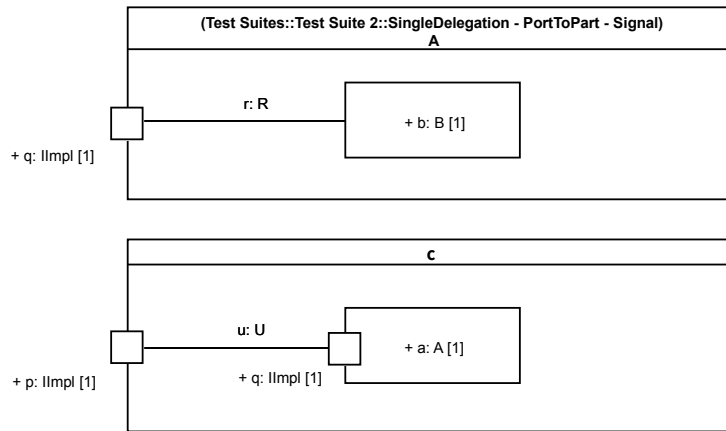

Figure 9.28 - Single Delegation from Port to Part with Port (Operation) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - SingleDelegation - PortToPWP - Operation'() {
```

```
   WriteLine("-- Running test case: Single delegation connector -
            Operation delegated from port to part with port --") ;
   c = new C() ;
   c.p.setP(4) ;
   AssertTrue("Operation call delegated", c.a.b.p == 4) ;
   WriteLine("-- End of test case --") ;
}
```

9.4.7 Single Delegation – PortToPartWithPort – Signal

This test case addresses forwarding of a signal through a single delegation link, corresponding to a single connector, expressed between a port and a part with port. If the signal is correctly forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.29 and Figure 9.30. Note that classes A and B are reused from 9.4.5.
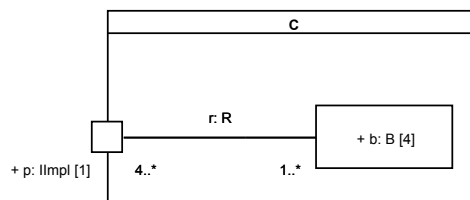

Figure 9.29 - Single Delegation from Port to Part with Port (Signal) – Classes, Signals and Interfaces

Figure 9.30 - Single Delegation from Port to Part with Port (Signal) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - SingleDelegation - PortToPWP - Signal'() {
   t = new Tester() ;
   t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
   accept(Start) ;
   this.c.p.S(4, this) ;
   accept(Continue) ;
   this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
  WriteLine("-- Running test case: Single delegation connector -
            Signal delegated from port to part with port --") ;
  AssertTrue("Signal delegated", this.c.a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.8 Multiple Delegation – SameConnector – PortToPart – Operation

This test case addresses forwarding of an operation call in presence of multiple delegation links, corresponding to a single connector, expressed between a port and a part. If the operation call is forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.31. Note that classes B and IImpl are reused from 9.4.4.

Figure 9.31 - Multiple Delegation with Single Connector from Port to Part (Operation) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - SameConnector - PortToPart - Operation'() {
  WriteLine("-- Running test case: Single delegation connector -
            Multiple links - Operation delegated from port to part --") ;
  c = new C() ;
  c.p.setP(4) ;
  AssertTrue("Operation call delegated", c.b[1].p == 4 || c.b[2].p == 4
                                         || c.b[3].p == 4 || c.b[4].p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.9 Multiple Delegation – SameConnector – PortToPart – Signal

This test case addresses forwarding of a signal through multiple delegation links, corresponding to a single connector, expressed between a port and a part. If the signal is correctly forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.32 and Figure 9.33. Note that classes A and B are reused from 9.4.5.
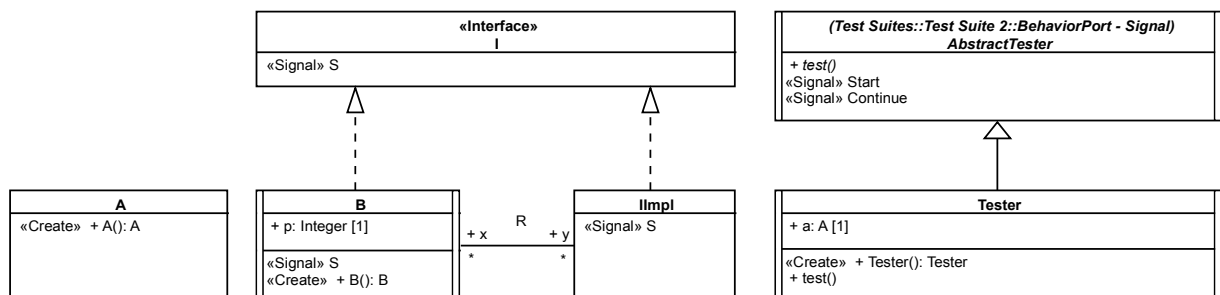

Figure 9.32 - Multiple Delegations with Single Connector from Port to Part (Signal) – Classes, Signals and Interfaces
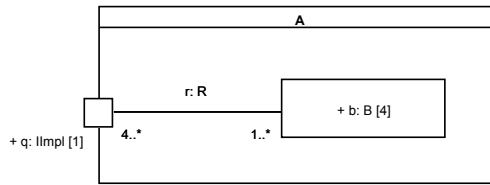
Figure 9.33 - Multiple Delegations with Single Connector from Port to Part (Signal) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - SameConnector - PortToPart - Signal'() {
  t = new Tester() ;
  t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
  accept(Start) ;
  this.a.q.S(4, this) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
  WriteLine("-- Running test case: Single delegation connector -
          Multiple links - Signal delegated from port to part --") ;
  AssertTrue("Signal delegated on all links (shall be true for default strategy)",
          this.a.b[1].p == 4 && this.a.b[2].p == 4
        && this.a.b[3].p == 4 && this.a.b[4].p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.10 Multiple Delegation – SameConnector – PortToPartWithPort – Operation

This test case addresses forwarding of an operation call in presence of multiple delegation links, corresponding to a single connector, expressed between a port and a part with port. If the operation call is forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.34. Note that classes A and IImpl are reused from 9.4.4.
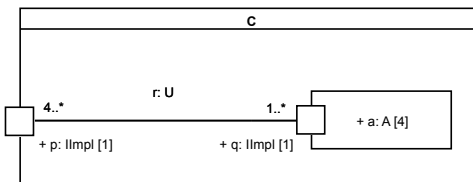

Figure 9.34 - Multiple Delegation with Single Connector from Port to Part with Port (Operation) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - SameConnector - PortToPWP - Operation'() {
  WriteLine("-- Running test case: Single delegation connector -
          Multiple links - Operation delegated from port to part with port--") ;
  c = new C() ;
  c.p.setP(4) ;
  AssertTrue("Operation call delegated", c.a[1].b.p == 4 || c.a[2].b.p == 4
                                      || c.a[3].b.p == 4 || c.a[4].b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.11 Multiple Delegation – SameConnector – PortToPartWithPort – Signal

This test case addresses forwarding of a signal through multiple delegation links, corresponding to a single connector, expressed between a port and a part with port. If the signal is correctly forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.35 and Figure 9.36. Note that classes A and IImpl are reused from 9.4.5.
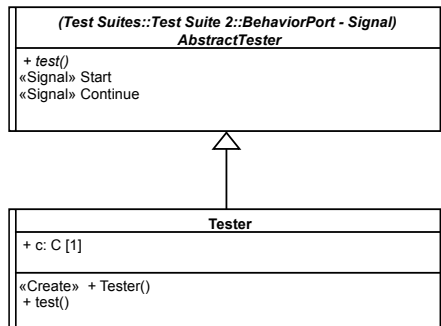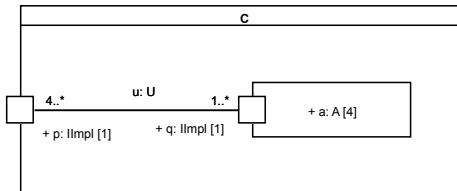
Figure 9.35 - Multiple Delegations with Single Connector from Port to Part with Port (Signal) – Classes, Signals and Interfaces

Figure 9.36 - Multiple Delegations with Single Connector from Port to Part with Port (Signal) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - SameConnector - PortToPWP - Signal'() {
  t = new Tester() ;
  t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
  accept(Start) ;
  this.c.p.S(4, this) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
  WriteLine("-- Running test case: Single delegation connector -
            Multiple links - Signal delegated from port to part with port --") ;
  AssertTrue("Signal delegated on all links (shall be true for default strategy)",
            this.c.a[1].b.p == 4 && this.c.a[2].b.p == 4
          && this.c.a[3].b.p == 4 && this.c.a[4].b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.12 Multiple Delegation – MultipleConnector – PortToPart – Operation

This test case addresses forwarding of an operation call in presence of multiple delegation links, each link corresponding to a particular connector, expressed between a port and a part. If the operation call is forwarded, some assertions can be

validated. Structural aspects of this test case are depicted in Figure 9.37 and Figure 9.38. Note that classes B and IImpl are reused from 9.4.4.
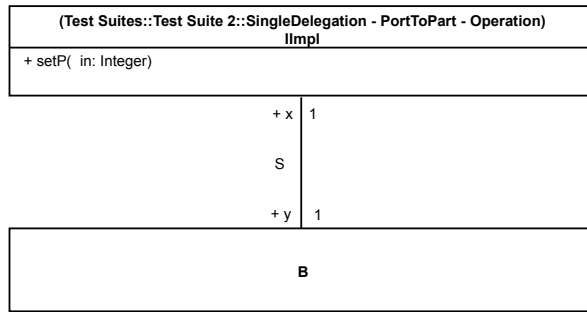

Figure 9.37 - Multiple Delegations with Multiple Connectors from Port to Part (Operation) – Classes
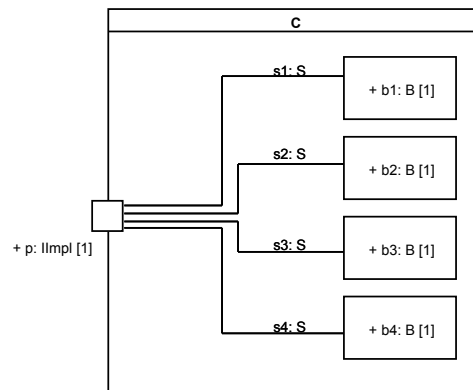

Figure 9.38 - Multiple Delegations with Multiple Connectors from Port to Part (Operation) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - MultipleConnectors - PortToPart - Operation'() {
  WriteLine("-- Running test case: Multiple delegation connectors -
            Multiple links - Operation delegated from port to part --") ;
  c = new C() ;
  c.p.setP(4) ;
  AssertTrue("Operation call delegated", c.b1.p == 4 || c.b2.p == 4 || c.b3.p == 4 || c.b4.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.13 Multiple Delegation – MultipleConnector – PortToPart – Signal

This test case addresses forwarding of a signal through multiple delegation links, each link corresponding to a particular connector, expressed between a port and a part. If the signal is correctly forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.39 and Figure 9.40. Note that classes A and B are reused from 9.4.5.
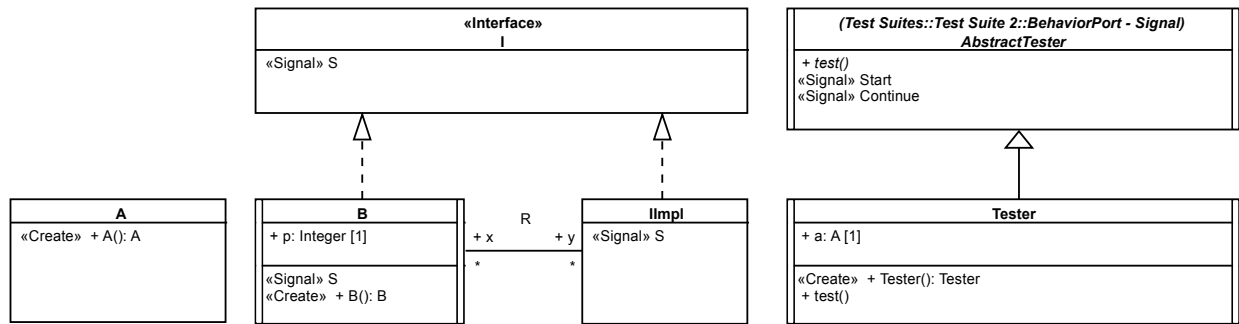
Figure 9.39 - Multiple Delegations with Multiple Connectors from Port to Part (Signal) – Classes, Signals and Interfaces
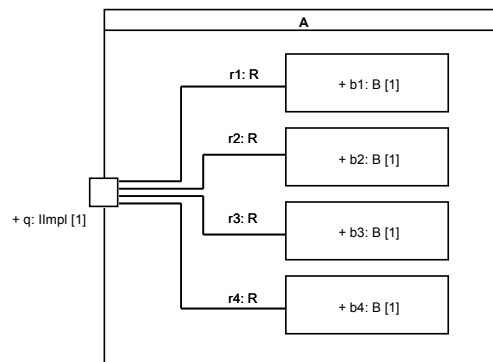

Figure 9.40 - Multiple Delegations with Multiple Connectors from Port to Part (Signal) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - MultipleConnectors - PortToPart - Signal'() {
  t = new Tester() ;
  t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
  accept(Start) ;
  this.a.q.S(4, this) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
  WriteLine("-- Running test case: Multiple delegation connectors -
            Multiple links - Signal delegated from port to part --") ;
  AssertTrue("Signal delegated on all links (shall be true for default strategy)",
            this.a.b1.p == 4 && this.a.b2.p == 4
         && this.a.b3.p == 4 && this.a.b4.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.14 Multiple Delegation – MultipleConnector – PortToPartWithPort – Operation

This test case addresses forwarding of an operation call in presence of multiple delegation links, each link corresponding to a particular connector, expressed between a port and a part with port. If the operation call is forwarded, some

assertions can be validated. Structural aspects of this test case are depicted in Figure 9.41. Note that classes A and IImpl are reused from 9.4.4.
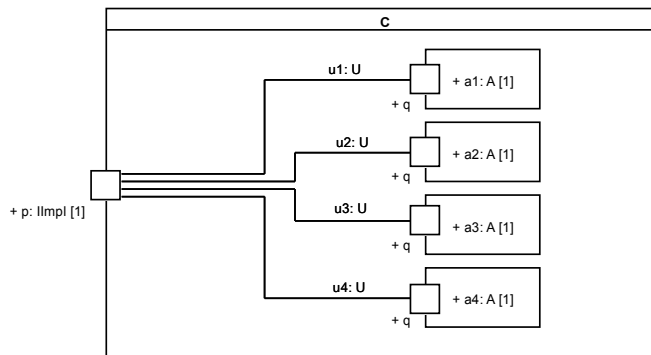

Figure 9.41 - Multiple Delegation with Multiple Connectors from Port to Part with Port (Operation) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - MultipleConnectors - PortToPWP - Operation'() {
  WriteLine("-- Running test case: Multiple delegation connectors -
            Multiple links - Operation delegated from port to part with port --") ;
  c = new C() ;
  c.p.setP(4) ;
  AssertTrue("Operation call delegated", c.a1.b.p == 4 || c.a2.b.p == 4
                                     || c.a3.b.p == 4 || c.a4.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.15 Multiple Delegation – MultipleConnector – PortToPartWithPort – Signal

This test case addresses forwarding of a signal through multiple delegation links, each link corresponding to a particular connector, expressed between a port and a part with port. If the signal is correctly forwarded, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.42 and Figure 9.43. Note that classes A and IImpl are reused from 9.4.5.
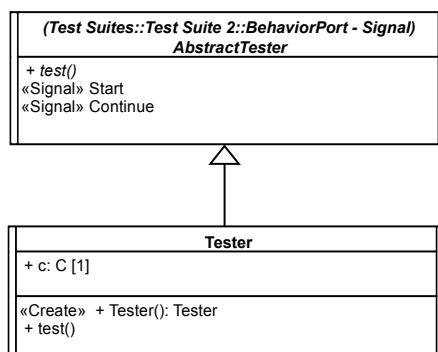

Figure 9.42 - Multiple Delegations with Multiple Connectors from Port to Part with Port (Signal) – Classes
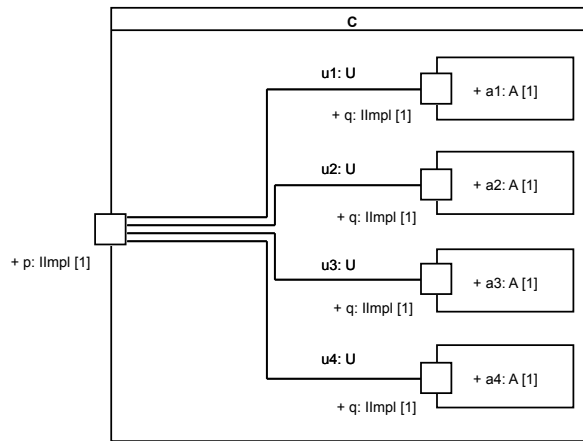
Figure 9.43 - Multiple Delegations with Multiple Connectors from Port to Part with Port (Signal) – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - MultipleDelegation - MultipleConnectors - PortToPWP - Signal'() {
  t = new Tester() ;
  t.Start() ;
}
```

Classifier behavior of Tester is:

```
activity TesterClassifierBehavior() {
  accept(Start) ;
  this.c.p.S(4, this) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  accept(Continue) ;
  this.test() ;
}
```

Method of Tester::test is:

```
activity test() {
      WriteLine("-- Running test case: Multiple delegation connectors -
              Multiple links - Signal delegated from port to part with port --") ;
  AssertTrue("Signal delegated on all links (shall be true for default strategy)",
            this.c.a1.b.p == 4 && this.c.a2.b.p == 4
        && this.c.a3.b.p == 4 && this.c.a4.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.4.16 Variants of Test Suite 2

The normative Test Suite model contains 3 variants of Test Suite 2 (packages Tests Suite 2 - Bis, - Ter, - Quater, depicted in Figure 9.1), which contain variants of all Test Suite 2 test cases. Test Suite 2 - Bis contains variants where connectors are not typed. Test Suite 2 - Ter contains variants where ports are typed by interfaces. Test Suite 2 - Quater contains variants where connectors are not typed and ports are typed by interfaces.

9.5 Test Suite 3: Communication (onPort)

This test suite focuses on communication semantics of UML Composite Structures, in the cases where property onPort of InvocationAction is used. Subclause 9.5.1 introduces Classes, Interfaces and Associations which are shared amongst the various test cases involving operation calls. The test cases are described in subclauses 9.5.2, 9.5.3, 9.5.4, and 9.5.5. Subclause 9.5.6 introduces Classes, Signals, Interfaces and Associations which are shared amongst the various test cases involving signals. These test cases are described in subclauses 9.5.7 and 9.5.8. Subclause 9.5.9 describes variants of Test Suite 3 where connectors are not typed and/or ports are typed by interfaces.

9.5.1 Operation common

Classes, Interfaces and Associations shared among test cases of Test Suite 3 involving operation calls are depicted in Figure 9.44.
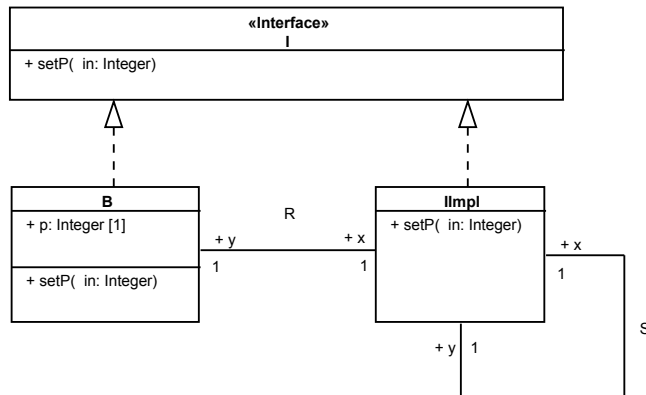

Figure 9.44 - Classes, Interfaces and Associations shared among test cases of Test Suite 3 involving operation calls

Method of B::setP is:

```
activity setP(in v : Integer) {
  this.p = v ;
}
```

9.5.2 Operation on Provided Interface

This test case addresses invocation of an Operation on a provided interface of a Port. If the operation call is forwarded into the port's owner, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.45 and Figure 9.46.
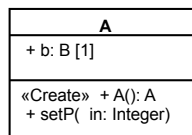

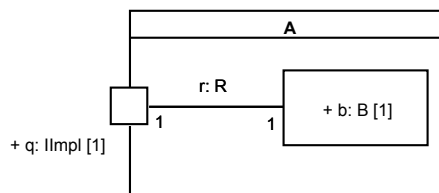**Figure 9.45 - Operation on a Provided Interface – Classes**

Figure 9.46 - Operation on a Provided Interface – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - Feature on Provided Interface'() {
  WriteLine("-- Running test case: Feature on Provided Interface --") ;
  a = new A() ;
  a.setP(4) ;
  AssertTrue("Invocation made into a through q", a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

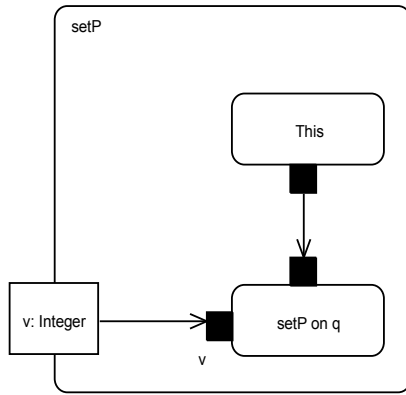Method of A::setP is depicted in the Activity diagram of Figure 9.47.

Figure 9.47 - Method of A::setP

9.5.3 Operation on Required Interface

This test case addresses invocation of an Operation on a required interface of a Port. If the operation call is forwarded outside of the port's owner, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.48 and Figure 9.49. Note that class A is reused from 9.5.2.

Figure 9.48 - Operation on a Required Interface – Classes

Figure 9.49 - Operation on a Required Interface – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - Feature on Required Interface'() {
  WriteLine("-- Running test case: Feature on Required Interface --") ;
  d = new D() ;
  d.c.setP(4) ;
  AssertTrue("Invocation forwarded out of c through q", d.a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

Method of C::setP is depicted in the Activity diagram of Figure 9.50.
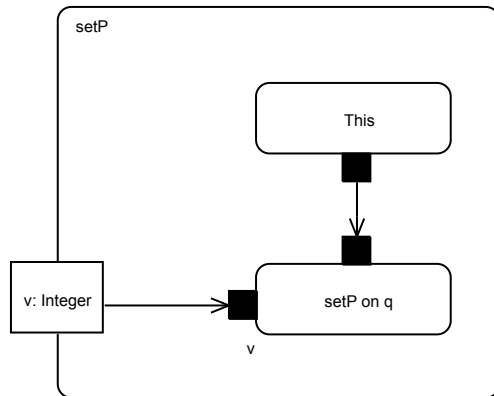
Figure 9.50 - Method of C::setP

9.5.4 Operation on Both Provided and Required Interface

This test case addresses invocation of an Operation which is both on a provided and a required interface of a Port. If the operation call is forwarded outside of the port's owner (i.e., in the test case, the invocation is made inside of the port's owner), some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.51 and Figure 9.52. Note that classes A and C are reused from 9.5.3.
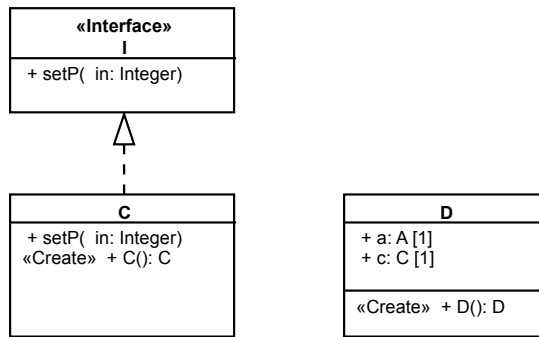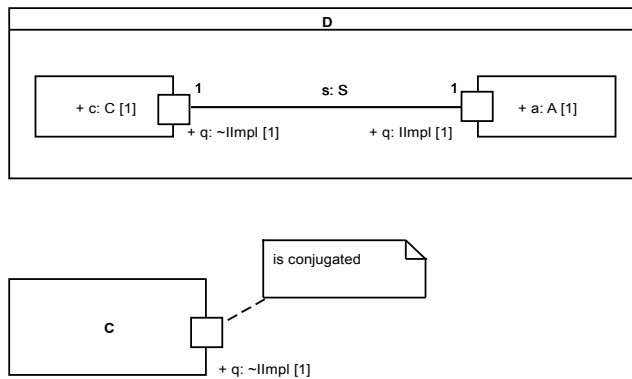

Figure 9.51 - Operation on both Provided and Required Interface – Classes

Figure 9.52 - Operation on Both Provided and Required Interface – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - Feature on both Required and Provided Interface'() {
  WriteLine("-- Running test case: Feature on both Required and Provided Interface --") ;
  d = new D() ;
  d.e.c.setP(4) ;
  AssertTrue("Invocation forwarded out of e through q", d.a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```
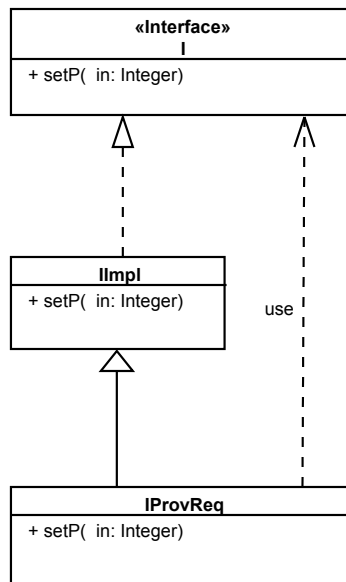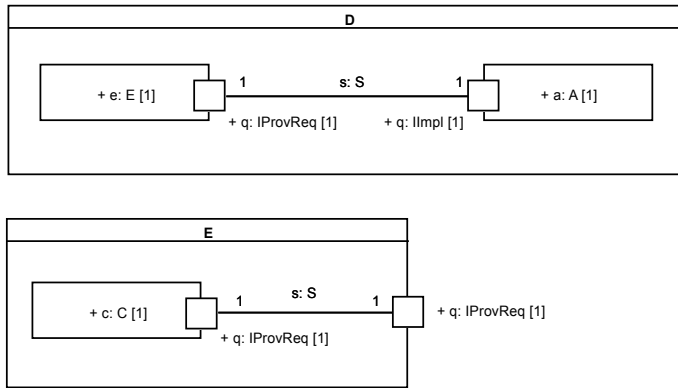
9.5.5 Operation on Required Interface with Delegation Chain

This test case addresses invocation of an Operation on a required interface of a Port, with a chain of delegation and assembly connectors. If the operation call is forwarded outside of the port's owner following appropriate connectors, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.53 and Figure 9.54. Note that classes A and C is reused from 9.5.3.
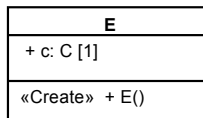
Figure 9.53 - Operation on a Required Interface with Delegation Chain – Classes
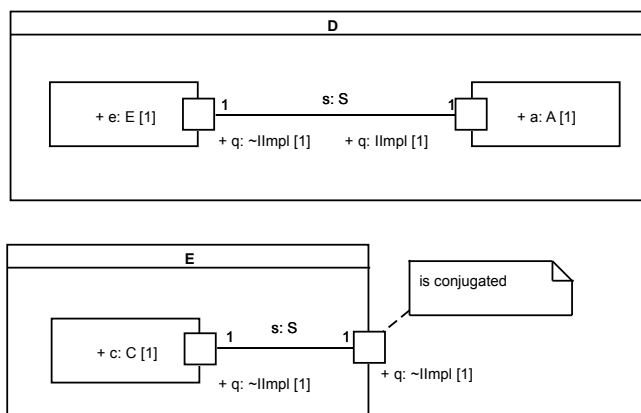
Figure 9.54 - Operation on a Required Interface with Delegation Chain – Internal structures

Corresponding test case behavior is:

```
activity 'Test case - Feature on Required Interface with Delegation Chain'() {
  WriteLine("-- Running test case: Feature on Required Interface with Delegation Chain--") ;
  d = new D() ;
  d.e.c.setP(4) ;
```

```
  AssertTrue("Invocation forwarded out of e through q", d.a.b.p == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.5.6 Signal common

Classes, Signals, Interfaces and Associations shared amongst test cases of Test Suite 3 involving signals are depicted in Figure 9.55 and Figure 9.56.
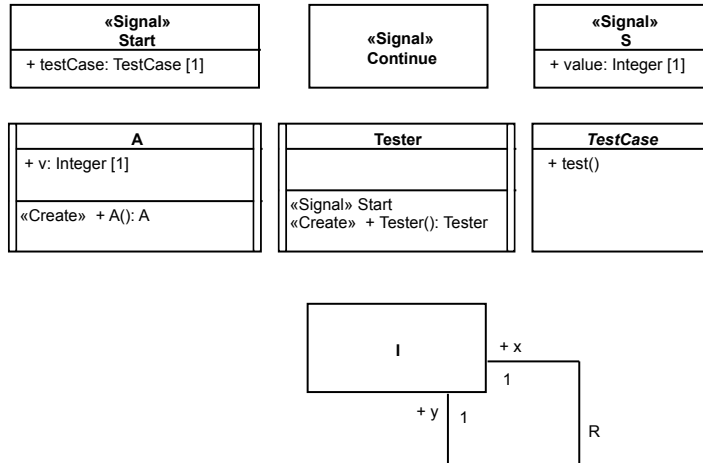
Figure 9.55 - Classes, Interfaces and Associations shared among test cases of Test Suite 3 involving signals
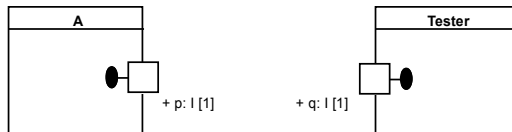
Figure 9.56 - Composite structures shared among test cases of Test Suite 3 involving signals

Classifier behaviors of A and Tester are depicted in activity diagrams of Figure 9.57 and Figure 9.58 respectively.
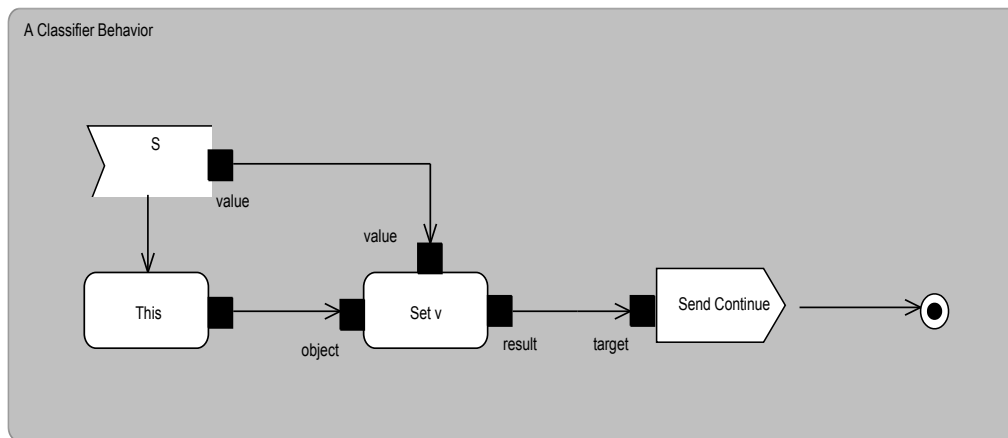
Figure 9.57 - Classifier behavior of A
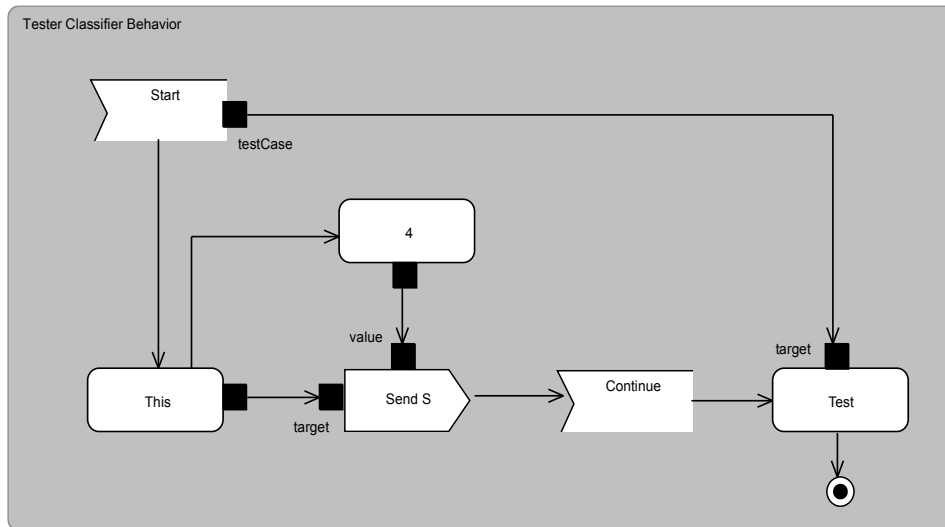
Figure 9.58 - Classifier behavior of Tester

9.5.7 Assembly

This test case addresses sending of a signal on a Port. If the signal is forwarded outside of the port's owner, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.59 and Figure 9.60.
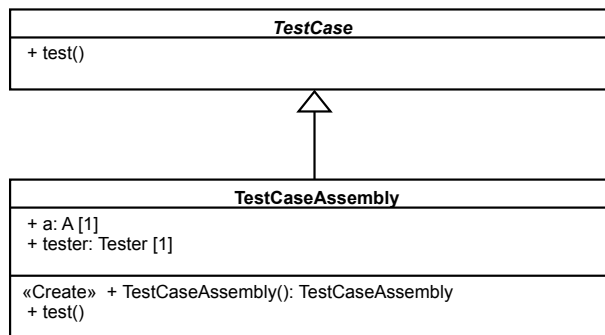
Figure 9.59 - Assembly – Classes
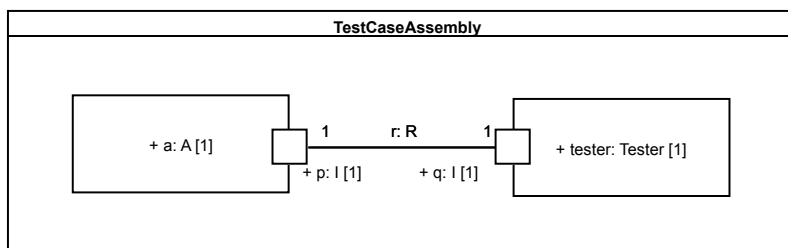
Figure 9.60 - Assembly – Internal structures

Corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Running test case - SendSignalAction using onPort -
           Single assembly connector --") ;
  testCase = new TestCaseAssembly() ;
  testCase.tester.Start(testCase) ;
}
```

Method of TestCaseAssembly::test is:

```
activity test() {
  AssertTrue("Signal correctly sent and received", this.a.v == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.5.8 Assembly and Delegation

This test case addresses sending of a signal on a Port. As compared to test case of 9.5.7, this test case implies a chain of delegation and assembly connectors. If the signal is forwarded outside of the port's owner, some assertions can be validated. Structural aspects of this test case are depicted in Figure 9.61 and Figure 9.62.
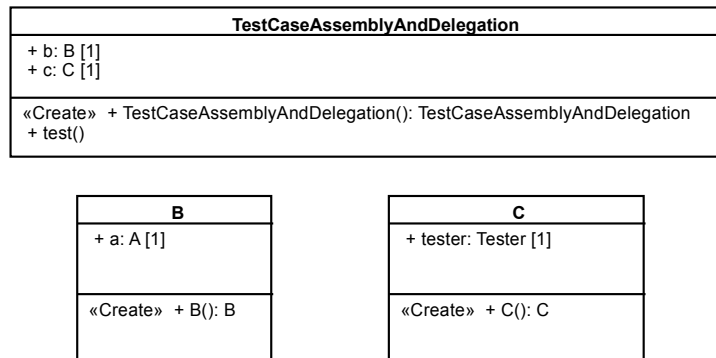
Figure 9.61 - Assembly and Delegation – Classes
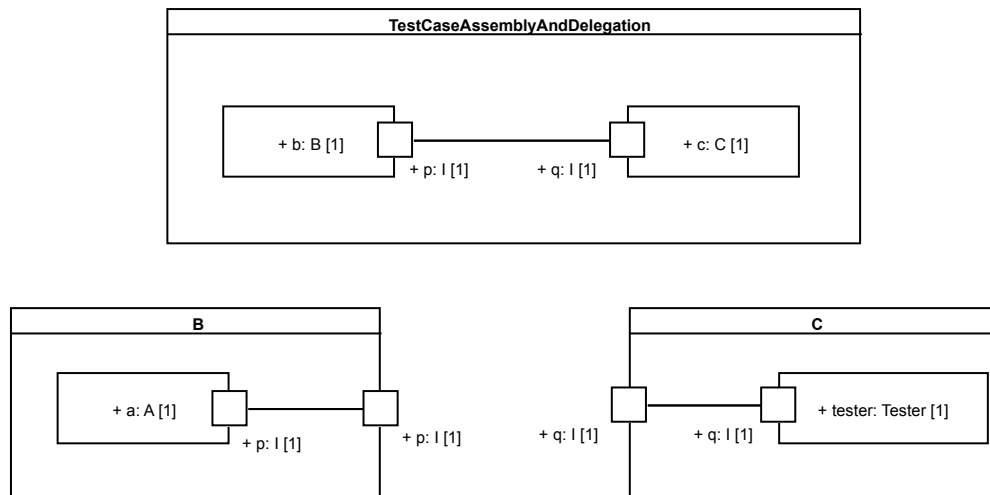
Figure 9.62 - Assembly and Delegation – Internal structures

Corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Running test case - SendSignalAction using onPort -
            Delegation/Assembly connector chain--") ;
  testCase = new TestCaseAssemblyAndDelegation() ;
  testCase.c.tester.Start(testCase) ;
}
```

Method of TestCaseAssemblyAndDelegation::test is:

```
activity test() {
  AssertTrue("Signal correctly sent and received", this.b.a.v == 4) ;
  WriteLine("-- End of test case --") ;
}
```

9.5.9 Variants of Test Suite 3

The normative Test Suite model contains 3 variants of Test Suite 3 (packages Tests Suite 3 - Bis, - Ter, - Quater, depicted in Figure 9.1), which contain variants of all Test Suite 3 test cases. Test Suite 3 - Bis contains variants where connectors are not typed. Test Suite 3 - Ter contains variants where ports are typed by interfaces. Test Suite 3 - Quater contains variants where connectors are not typed and ports are typed by interfaces.

9.6 Test Suite 4: Destruction

This test suite focuses on destruction semantics of UML Composite Structures. Subclause 9.6.1 concerns recursive destruction of parts and ports, when a composite object is destroyed. Subclauses 9.6.2 and 9.6.3 deal with situations in which structural features are cleared (via the usage of a ClearStructuralFeatureValueAction). These situations imply that links in which cleared values are involved shall be destroyed. Subclause 9.6.4 deals with situations in which an identified value is removed from a structural feature (via the usage of RemoveStructuralFeatureValueAction). These situations imply that links in which the removed value is involved shall be destroyed.

9.6.1 Recursive destruction of parts and ports

This test case validates recursive destruction of parts and ports in the case where a composite object is destroyed. Note that all classes are reused from 9.3.4. Corresponding test case behavior is:

```
activity TestCaseBehavior() {
  a = new A_Star_Pattern() ;
  WriteLine("-- Running Test Case: Recursive destruction of parts and ports --") ;
  Write("# Instances of B: ") ; Write(B.allInstances()->size()) ; WriteLine("") ;
  Write("# Instances of C: ") ; Write(C.allInstances()->size()) ; WriteLine("") ;
  Write("# Instances of IImpl: ") ; Write(IImpl.allInstances()->size()) ; WriteLine("") ;
  WriteLine("... Destruction of a ...") ;
  a.destroy() ;
  AssertTrue("All instances of B recursively destroyed", B.allInstances()->size() == 0) ;
  AssertTrue("All instances of C recursively destroyed", C.allInstances()->size() == 0) ;
  AssertTrue("All instances of IImpl recursively destroyed", IImpl.allInstances()->size() == 0) ;
  WriteLine("-- End of Test Case --") ;
}
```

9.6.2 Removing instance from part

This test case validates that the removal of an instance from a part implies destruction of related links. Note that all classes and associations are reused from 9.4.4. Corresponding test case behavior is:

```
activity 'Removing instance from part'() {
  WriteLine("-- Running test case: Removing instance from a part
            implies destruction of related links --") ;
  a = new A() ;
  b = a.b ;
  q = a.q ;
  a.b = null ;
  AssertTrue("Link has been destroyed", R.x(y => q)->size() == 0 && R.y(x => b)->size() == 0) ;
  WriteLine("-- End of test case --") ;
}
```

9.6.3 Removing instance from port

This test case validates that the removal of an instance from a port implies destruction of related links. Note that all classes and associations are reused from 9.4.4. Corresponding test case behavior is:

```
activity 'Removing instance from port'() {
  WriteLine("-- Running test case: Removing instance from a port
            implies destruction of related links --") ;
  a = new A() ;
  b = a.b ;
  q = a.q ;
  a.q = null ;
  AssertTrue("Link has been destroyed", R.x(y => q)->size() == 0 && R.y(x => b)->size() == 0) ;
  WriteLine("-- End of test case --") ;
```

```
}
```

9.6.4 Removing specific instances from part and port

Test cases included in this subclause validate that if a value is removed from a part or a port, then the links in which the removed value was involved are destroyed.

Figure 9.63 shows the internal structure of a class S.

Figure 9.63 - Internal Structure of Class S

When this composite structure is instantiated, objects and links depicted in Figure 9.64 are created.
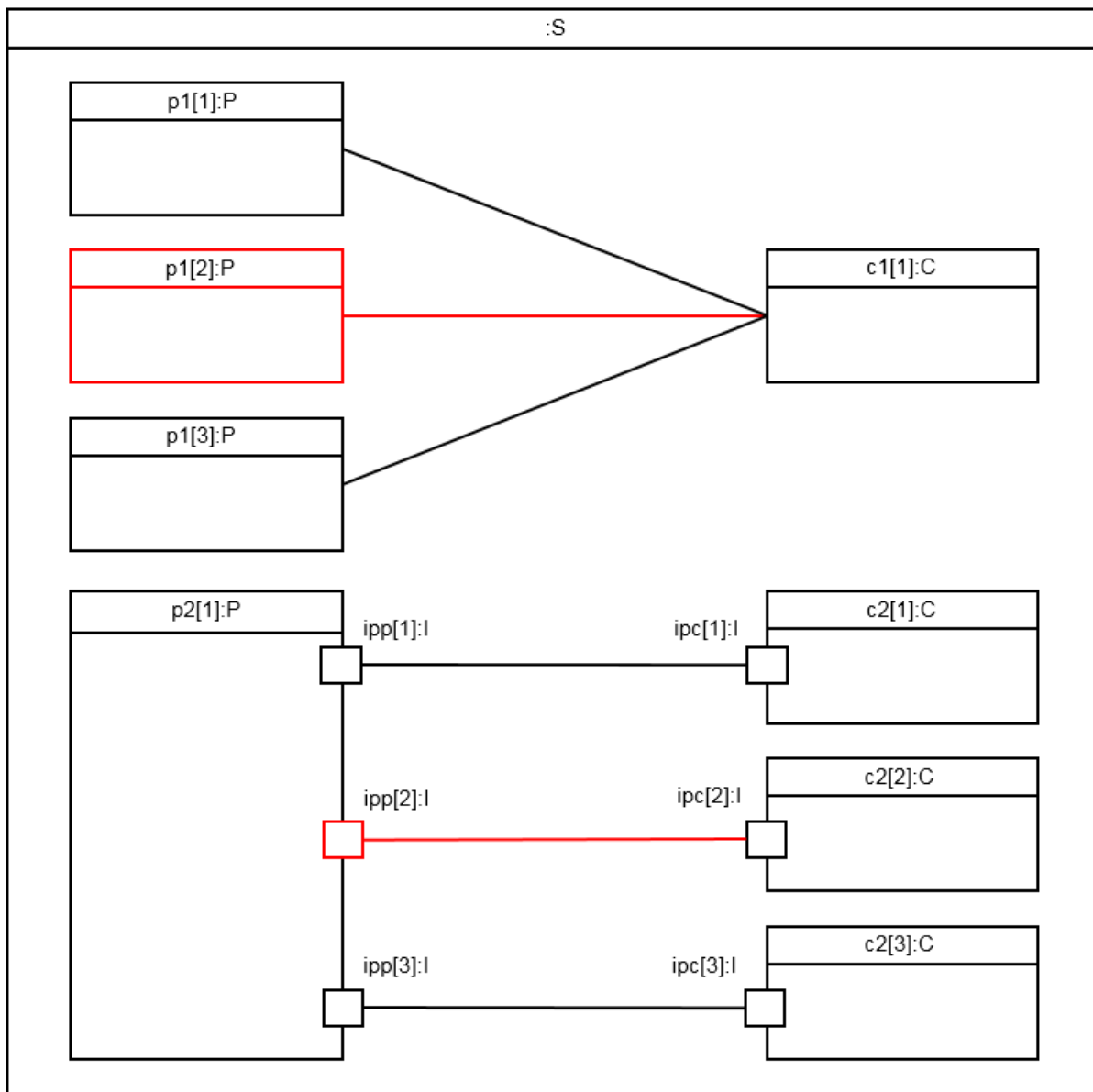
Figure 9.64 - Objects and Links Resulting from the Instantiation of S

If a RemoveStructuralFeatureValueAction is used to remove the second interaction point (i.e., p2[1].ipp[2] in Figure 9.64) from the list of values for the structural feature ipp of object p2[1], then it implies that:

1. p2[1].ipp[2] interaction point is removed from the list of values for ipp structural feature owned by p2[1].

2. The link in which p2[1].ipp[2] is involved is destroyed (see outgoing link of p2[1].ipp[2] in Figure 9.64).

Such behavior is validated by the test case specified below:

```
activity 'Remove instance P from S.p1 part - Test Case'() {
  WriteLine("-- Running Test Case: Remove instance P from S.p1 part --") ;
  S s = new S();
  P p = s.p1->at(2);
  removeP(s, p, 2);
  AssertTrue("Link involving s.p1[2] and s.c1[1] was not deleted", R1.c(p=>p)->size() == 0);
  AssertTrue("Links s.p1[1]-s.c1[1] and s.p1[3]-s.c1[1]", R1.p(c=>s.c1->at(1))->size() == 2);
  WriteLine("-- End of test case --") ;
}
```

If a RemoveStructuralFeatureValueAction is used to remove the second interaction point (i.e., p2[1].ipp[2] in Figure 9.64 from the list of values for the structural feature ipp of object p2[1], then it implies that:

1. p2[1].ipp[2] interaction point is removed from the list of values for ipp structural feature owned by p2[1].

2. The link in which p2[1].ipp[2] is involved is destroyed (see outgoing link of p2[1].ipp[2] in Figure 9.64).

Such behavior is validated by the test case specified below:

```
activity 'Remove port instance I from S.p2.ipp - Test Case'() {
  WriteLine("-- Running Test Case: Remove port instance I from S.p2.ipp --") ;
  S s = new S();
  P p = s.p2->at(1);
  I i = p.ipp->at(2);
  removeIpp(p, i, 2);
  AssertTrue("Link involving s.p2[1].ipp[2] and s.c2[2].ipc[1] was destroyed", R2.x(y=>i)->size() ==
0);
  AssertTrue("Link s.p2[1].ipp[1]-s.c2[1].ipc[1] remains in the locus", R2.y(x=>s.c2->at(1).ipc-
>at(1))->size() == 1);
  AssertTrue("Link s.p2[1].ipp[3]-s.c2[3].ipc[1] remains in the locus", R2.y(x=>s.c2->at(3).ipc-
>at(1))->size() == 1);
  WriteLine("-- End of test case --") ;
}
```

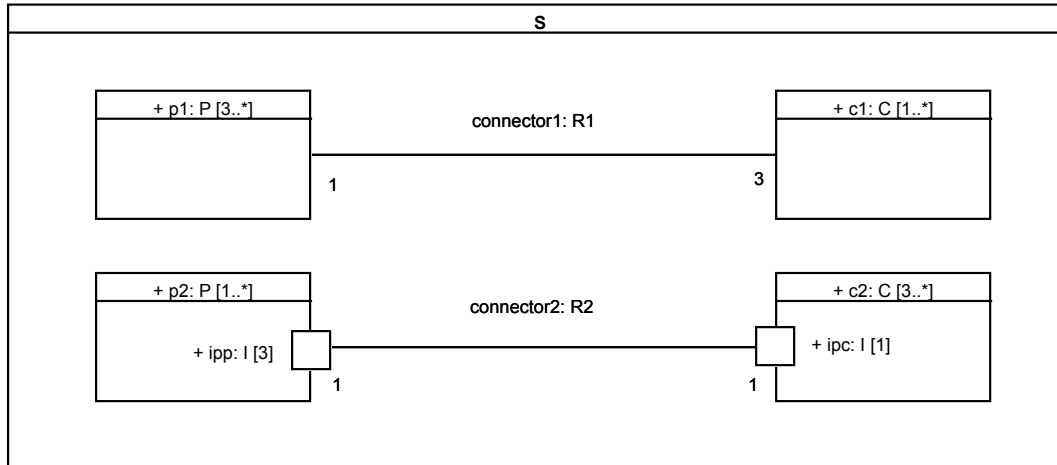Figure 9.65 shows the internal structure of class T.
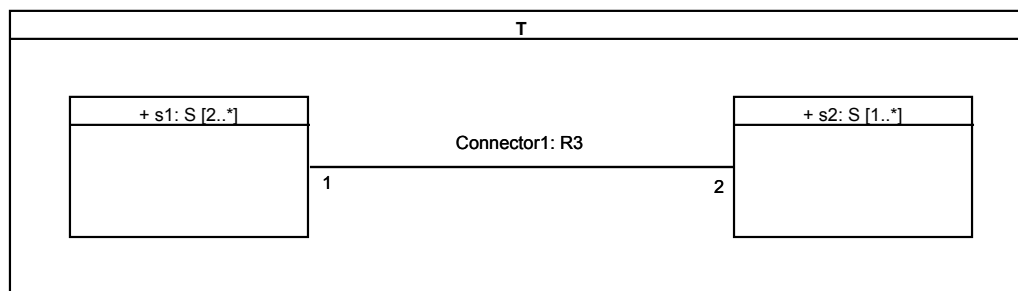
Figure 9.65 - Internal Structure of Class T

When the composite structure is instantiated, objects and links depicted in Figure 9.66 are created.
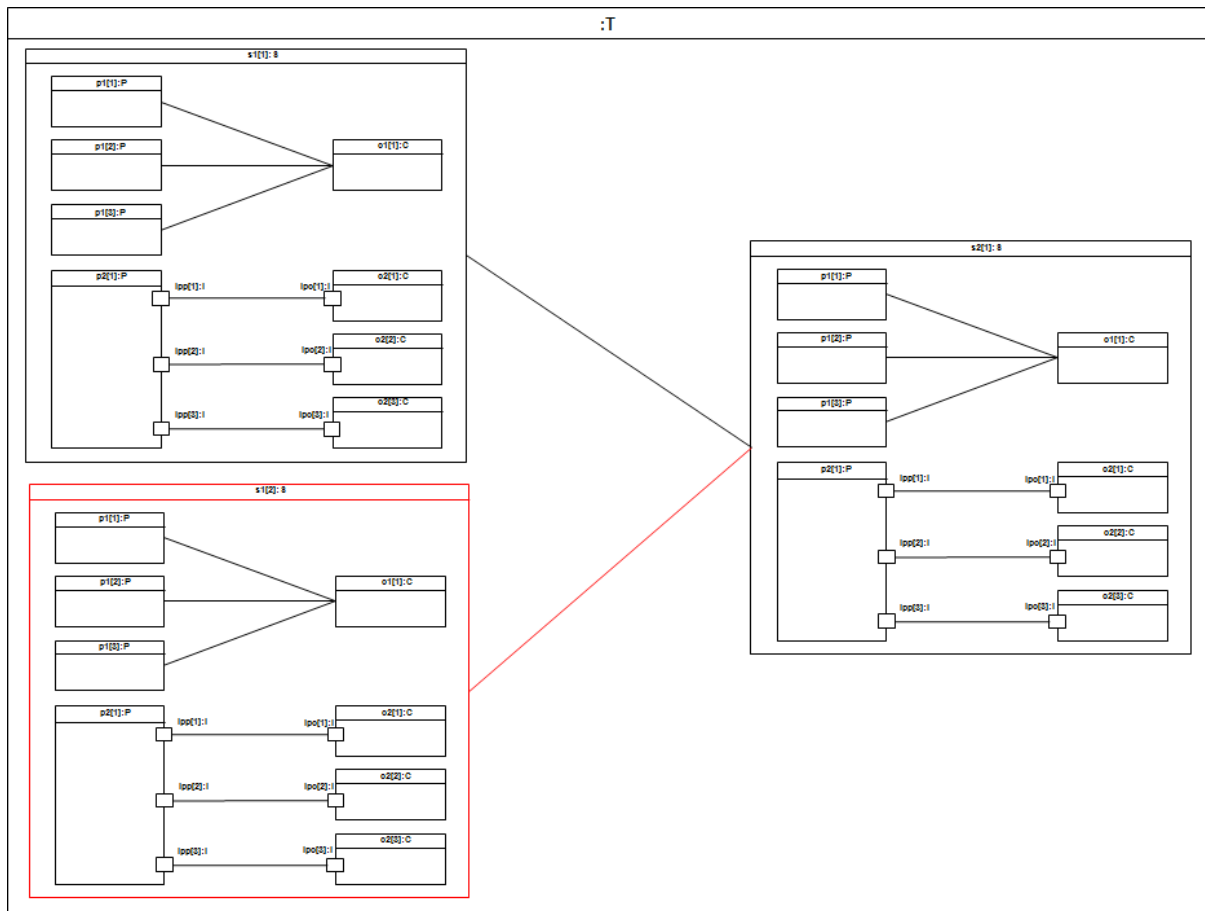
Figure 9.66 - Objects and Links Resulting from the Instantiation of T

If a RemoveStructuralFeatureValueAction is used to remove the second instance of S (see s1[2] in Figure 9.66) referenced the s1 the feature value of object T, then it implies that:

1. s1[2] object is removed from s1 feature value of object T.

2. The link in which s1[2] object is involved is destroyed.

Note: All links in which values that are referenced in feature values of the removed S object *are not required* to be destroyed. Hence after the execution of the RemoveStructuralFeatureValueAction the following links still exists at the current locus:

1. s1[2].p1[1] to s1[2].c1[1]

2. s1[2].p1[2] to s1[2].c1[1]

3. s1[2].p1[3] to s1[2].c1[1]

4. s1[2].p2[1].ipp[1] to s1[2].c1[1].ipc[1]

5. s1[2].p2[1].ipp[2] to s1[2].c1[2].ipc[1]

1. s1[2].p2[1].ipp[3] to s1[2].c1[3].ipc[1]

Such a behavior is validated by the test case specified below:

```
activity 'Remove instance S from T.s1 part - Test Case'() {
  WriteLine("-- Running Test Case: Remove instance S from T.s1 part --") ;
```

```
  T t = new T();
  S s = t.s1->at(1);
  removeS(t, s, 2);
  AssertTrue("Link involving t.s1[2] and t.s2[1] was destroyed", R3.x(y=>s)->size() == 0);
  AssertTrue("Link t.s2[2].p1[1]-t.s2.c1[1] remains in the locus", R1.c(p=>s.p1->at(1))->size() ==
1);
  AssertTrue("Link t.s2[2].p1[2]-t.s2.c1[1] remains in the locus", R1.c(p=>s.p1->at(2))->size() ==
1);
  AssertTrue("Link t.s2[2].p1[3]-t.s2.c1[1] remains in the locus", R1.c(p=>s.p1->at(3))->size() ==
1);
  AssertTrue("Link t.s2[2].p2[1].ipp[1]-t.s2[2].c2[1].ipc[1] remains in the locus", R2.y(x=>s.c2-
>at(1).ipc->at(1))->size() == 1);
  AssertTrue("Link t.s2[2].p2[1].ipp[2]-t.s2[2].c2[2].ipc[1] remains in the locus", R2.y(x=>s.c2-
>at(2).ipc->at(1))->size() == 1);
  AssertTrue("Link t.s2[2].p2[1].ipp[3]-t.s2[2].c2[3].ipc[1] remains in the locus", R2.y(x=>s.c2-
>at(3).ipc->at(1))->size() == 1);
  WriteLine("-- End of test case --") ;
}
```

Annex A
Semantics of MARTE PpUnits
(informative)

A.1 Introduction

The broad field of embedded systems, to which MARTE is dedicated, relies directly in the capacity of specifying, predicting, and/or assessing the usage that the system does of the available resources. For real-time systems this is strongly related to the scheduling of the platform execution capacity and the arbitration protocols needed in the mutually exclusive accesses to shared resources. For this reason models of Composite Structures subject to real-time constraints, or targeted to embedded platforms, need the semantics of the MARTE elements used on them to be clearly described. This should be done in such a way that restricts the "Genericity of the Execution Model" claimed in section 2.3 of fUML specification. This implies directly the need to specify the semantics of time, concurrency, and inter-object communication mechanisms, which were explicitly left open in fUML.

Considering the large amount of potential modeling intents for which MARTE is able to be applied, the adequate strategy to describe semantics here should follow an approach similar to the one used in MARTE section 2.4 (Conformance with MARTE). According to that approach potential usages of the specification (and consequently the tools provided by tool vendors for them) are categorized in a series of "compliance cases".

The description of the corresponding "Semantics Variation Cases" would encompass aspects like: the modeling elements in play (stereotypes, base UML elements, and concrete values for the attributes that condition the desired semantics), the additional rules for their usage (expressed either with OCL or by means of textually described constraints), and the transformations or equivalent models in the target sub-domains or languages.

But an exhaustive search for all coherent sets of semantics of all elements in MARTE for all those potential users and compliance cases is an overwhelming effort not targeted in this specification. Instead, this annex fixes the semantics of a concrete set of elements and modeling patterns. The actors to whom this semantics is addressed are those named as "Software Architect" in MARTE subclause 6.2.3 (How to use this specification), in particular those who design real-time systems with the additional need of having their final artifacts suitable to be validated by means of schedulability analysis.

Then, according to this modeling intent, the semantics of the combination of specific MARTE elements in its Generic Resource Modeling (GRM) and High-Level Application Modeling (HLAM) sections need to be provided. The two basic structural elements in MARTE for which the formalization of semantics is needed are the Real-time Unit (RtUnit), and the Passive Protected Unit (PpUnit). These concepts are described in the HLAM sub-profile of MARTE, and are the fundamental building blocks for programming real-time applications with high level languages from the analytical point of view [1] [2] [3]. The construction of analysis models to validate these design intent models is possible [4] under the assumption of a generic execution platform and a behavioral semantics that respects best practices and widely known models of interaction taken from the real-time systems community [2] [5].

An initial set of modeling rules for the semantics of RtUnits, which included those for describing the platform, was proposed in [4] as a set of rules for the modeling of tasks that may be (a) Independent, (b) Related by means of shared resources, or (c) by control flow dependencies. This annex expresses the semantics for PpUnits as stated in clause 1 (Scope) of this specification, namely by defining semantic visitors that specify the needed semantics as extension of the PSCS execution model.

The basis for modeling with schedulability analysis in mind is the specification of three basic models, the platform, the logic of the application and the stimuli workload the system is expected to support. Over the initial set of modeling rules, proposed in [4], here we enhance and extend it to address also execution semantics of the PpUnits used in high level application models.

Then besides the semantics that is formalized in the next clause, when tasks share passive data the PpUnit modeling construct is used, and in this case these additional rules apply:

- The ExecKind of PpUnit services is ImmediatRemote

- All services of the PpUnit use the same protection protocol: ImmediateCeiling or PriorityInheritance

- The ConcurrencyPolicy of PpUnit is Guarded.

The concurrencyPolicy of the kind Concurrent might be enabled in order to have the writer/reader ConcurrencyKind available, but this behavior requires additional capabilities from the analysis techniques to really take advantage of it, so in principle it is discouraged.

A.2 Semantics

A.2.1 Overview

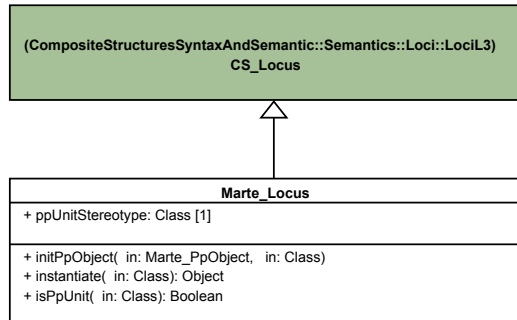Extensions to the normative composite structures execution model are depicted in Figure A.1 and Figure A.2.

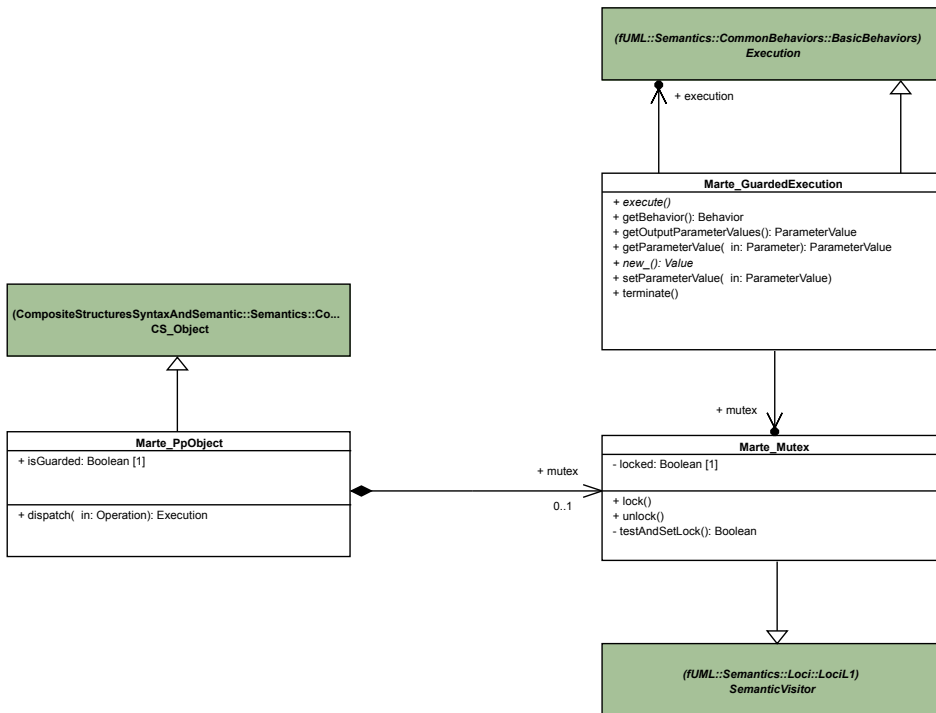**Figure A.1 - Locus extension diagram**

Figure A.2 - Object and Execution extension diagram

A.2.2 Class descriptions

A.2.2.1 Marte_GuardedExecution

Marte_GuardedExecution extends fUML Execution so that only one execution issued by a Marte_PpObject (i.e., an Object classified by a Marte PpUnit) can execute at a time. Mutual exclusion is guaranteed by the usage of a Marte_Mutex.

Generalizations

- Execution (from fUML_Semantics::Semantics::CommonBehavior)

Attributes

- None

Associations

- execution : Execution, the actual encapsulated Execution, whose call to execute is guarded by the lock of a mutex (see operation execute below).
- mutex : Marte_Mutex, a reference to the mutex that will be used to guard the execution of the encapsulated Execution. This mutex is owned by the Marte_PpObject that produced this Marte_GuardedExecution.

Operations

```
[1] public execute()
        // Ensures mutual exclusion of executions
        // issued by a same Marte_PpObject, due to a dispatch.
        // The execution of each encapsulated execution is protected
        // by locking mutex prior to execution.
        // This mutex belongs to a Marte_PpObject,
        // and it is shared by all Marte_GuardedExecution
        // that has been issued by this Marte_PpObject.
        this.mutex.lock();
        this.execution.execute() ;
        this.mutex.unlock();

[2] public getBehavior() : Behavior
        // delegates to the encapsulated execution
        return this.execution.getBehavior();

[3] public getOutputParameterValues() : ParameterValue [*]
        // delegates to the encapsulated execution
        return this.execution.getOutputParameterValues();

[4] public getParameterValue(parameter:Parameter) : ParameterValue
        // delegates to the encapsulated execution
        return this.execution.getParameterValue(parameter);

[5] public new_() : Value
        return new Marte_GuardedExecution();

[6] public setParameterValue(parameterValue:ParameterValue)
        // delegates to the encapsulated execution
        this.execution.setParameterValue(parameterValue);

[7] public terminate()
        // delegates to the encapsulated execution
        this.execution.terminate();
```

A.2.2.2 Marte_Locus

Marte_Locus extends CS_Locus so that, when a Class with stereotype PpUnit stereotype has to be instantiated, a Marte_PpObject is returned instead of a CS_Object.

Generalizations

- CS_Locus

Attributes

- ppUnitStereotype : Class, the Marte PpUnit stereotype

Associations

- None

Operations

```
[1] public initPpObject(object:Marte_PpObject, stereotypedClass:Class)
        // Initializes properties of the given Marte_PpObject, according to the
        // given stereotypedType. This Class is a classifier of the given object,
        // and it has stereotype PpUnit applied. The property values associated
        // with the stereotype application are used to initialize properties of the
        // Marte_PpObject.

[2] public instantiate(class:Class) : Object
        // Extends CS_Locus so that, if the given type has
        // stereotype PpUnit applied, a Marte_PpObject is
        // instantiated instead of a CS_Object.
        // Otherwise, behaves like in CS_Locus.
        Object_ object ;
        if (this.isPpUnit(type)) {
                object = new Marte_PpObject();
                object.types.add(type);
                object.createFeatureValues();
                this.add(object);
                this.initPpObjet((Marte_PpObject)object, type) ;
        }
        else {
                object = super.instantiate(type) ;
        }
        return object ;

[3] public isPpUnit(type:Class) : Boolean
        // Determines if the given type has stereotype PpUnit applied
```

A.2.2.3 Marte_Mutex

Marte_Mutex is a new kind of SemanticVisitor, introduced to guarantee mutual exclusion of multiple Marte_GuardedExecutions issued by a same Marte_PpObject.

Generalizations

- SemanticVisitor (from fUML_Semantics::Semantics::Loci)

Attributes

- locked : Boolean, indicates whether this Marte_Mutex is locked

Associations

- None

Operations

```
[1] public lock()
        // Loops until the calling object
        // has been able to lock this mutex
        while (this.testAndSetLock()) { }

[2] public unlock()
        // Unlocks the mutex
```

```
            _beginIsolation();
            this.locked = false;
            _endIsolation();

[3] public testAndSetLock() : Boolean
            // If the context mutex is not locked,
            // locks it and returns false.
            // Returns true otherwise
            boolean wait = true;
            _beginIsolation();
            if (!this.locked) {
                    this.locked = true;
                    wait = false;
            }
            _endIsolation();
            return wait;
```

A.2.2.4   Marte_PpObject

Marte_PpObject extends CS_Object so that, when operation dispatch is called, a Marte_GuardedExecution is returned instead of a fUML Execution, if this Marte_PpObject is guarded.

Generalizations

- CS_Object

Attributes

- isGuarded : Boolean, determines whether this Marte_PpObject is guarded. Its value is based on the value of property concPolicy of stereotype PpUnit, which is applied on a Class typing this object.

Associations

- mutex : Marte_Mutex, the mutex owned by this object. It is used to guarantee mutual exclusion of executions produced by this object.

Operations

```
[1] public dispatch(operation:Operation) : Execution
            // First construct an Execution for the given operation,
            // as specified by CS_Object.
            // If this Marte_PpObject is guarded,
            // the constructed execution is encapsulated in a
            // Marte_GuardedExecution, which is returned.
            // This execution keeps a reference to the mutex
            // owned by this Object.
            // Otherwise, simply returns the constructed execution
            Execution execution = super.dispatch(operation);
            if (this.isGuarded) {
                    Marte_GuardedExecution guarded = new Marte_GuardedExecution() ;
                    guarded.execution = execution ;
                    guarded.mutex = this.mutex ;
                    execution = guarded ;
            }
            return execution ;
```

A.3 PpUnit Test suite

This test suite encompasses two test cases, described in A.3.1 and A.3.2. Its purpose is to demonstrate that concurrent calls to operations of a guarded PpUnit are serialized.

A.3.1 No Ports – Globally Guarded

The structural aspects of this test case are depicted in Figure A.3 and Figure A.4. Four active objects (classified by A) share a single PpUnit object (classified by B). The active objects concurrently call operation B::compute() on the shared object. Since PpUnit B is guarded (property concPolicy == guarded), these calls shall be serialized.

Figure A.3 - Class diagram

Figure A.4 - Composite structure diagram

The classifier behavior of A is:

```
activity AClassifierBehavior() {
    accept(Start) ;
    b = R.b(this) ;
    b.compute() ;
}
```

The method of operation B::compute() is:

```
activity compute() {
    WriteLine("-------------------------------------") ;
    WriteLine("- Incrementing p:") ;
    this.p = this.p + 10 ;
    WriteLine("     - value of p: " + ToString(this.p)) ;
    WriteLine("- Decrementing p:") ;
    this.p = this.p - 10 ;
    WriteLine("     - value of p: " + ToString(this.p)) ;
    WriteLine("-------------------------------------") ;
    WriteLine("") ;
}
```

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
    c = new C() ;
    for (a in c.a) {
        a.Start() ;
    }
}
```

In a valid execution trace, the various executions of B::compute() shall be serialized, so that no interleaving can be observed:

```
-------------------------------------
- Incrementing p:
     - value of p: 10
- Decrementing p:
     - value of p: 0
-------------------------------------
-------------------------------------
```

```
- Incrementing p:
      - value of p: 10
- Decrementing p:
      - value of p: 0
------------------------------------
------------------------------------
- Incrementing p:
      - value of p: 10
- Decrementing p:
      - value of p: 0
------------------------------------
------------------------------------
- Incrementing p:
      - value of p: 10
- Decrementing p:
      - value of p: 0
------------------------------------
```

A.3.2 Ports – Globally Guarded

The structural aspects of this test case are depicted in Figure A.5 and Figure A.6. This is a variant of the test case depicted in A.3.1, where instances of A communicate with the shared instance of B through ports and connectors. The active objects concurrently call operation B::compute() on the shared object. Since PpUnit B is guarded (property concPolicy == guarded), these calls shall be serialized.
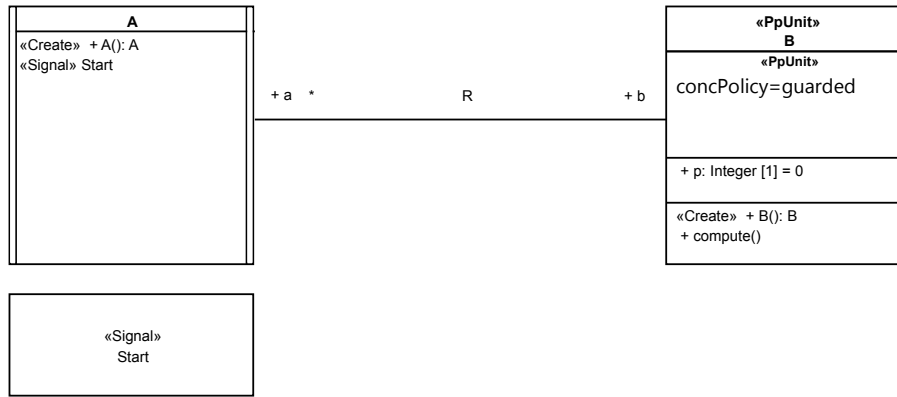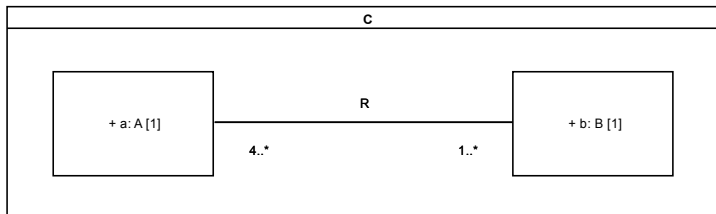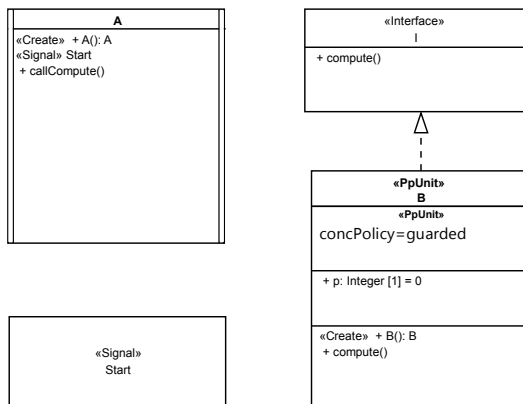

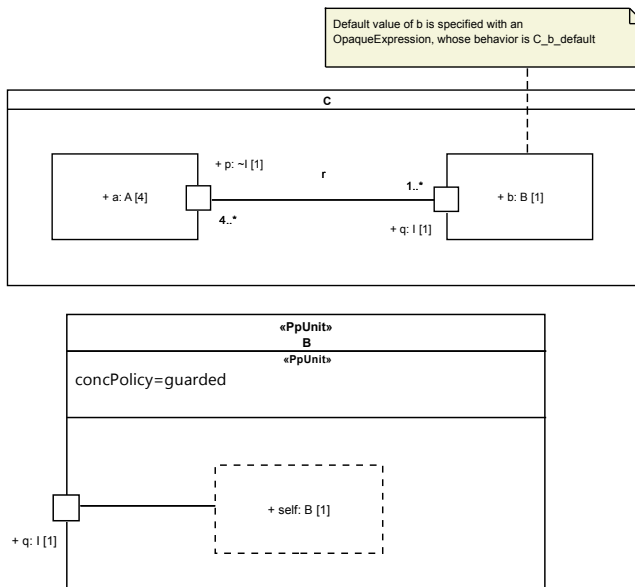**Figure A.5 - Class diagram**

Figure A.6 - Composite structure diagram

The classifier behavior of A is:

```
activity AClassifierBehavior() {
    accept(Start) ;
    this.callCompute() ;
}
```

The method of operation A::callCompute() is specified in the activity diagram of Figure A.7. Action "This" is a ReadSelfAction, and "Call compute on port p" is a CallOperationAction for operation I::compute(), with property onPort set to A::p.
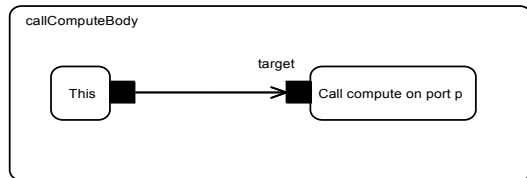
Figure A.7 - Activity diagram of A::callCompute() method

The method of operation B::compute() is:

```
activity compute() {
    WriteLine("-------------------------------------") ;
    WriteLine("- Incrementing p:") ;
    this.p = this.p + 10 ;
    WriteLine("      - value of p: " + ToString(this.p)) ;
    WriteLine("- Decrementing p:") ;
    this.p = this.p - 10 ;
    WriteLine("      - value of p: " + ToString(this.p)) ;
    WriteLine("-------------------------------------") ;
    WriteLine("") ;
}
```

Activity C_b_default (which is the behavior associated with the OpaqueExpression default value of C::b) is:

```
activity C_b_default() : B {
    b = new B() ;
    b.self = b ;
    // Default is a generic association,
    // specified in the non-normative library GenericAssociation
    // Cf. clause Annex C
    Default.createLink(x => b.q, y => b.self) ;
    return b ;
}
```

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
    c = new C() ;
    for (a in c.a) {
        a.Start() ;
    }
}
```

In a valid execution trace, the various executions of B::compute() shall be serialized, so that no interleaving can be observed:

```
-------------------------------------
- Incrementing p:
      - value of p: 10
- Decrementing p:
      - value of p: 0
-------------------------------------

-------------------------------------
- Incrementing p:
      - value of p: 10
- Decrementing p:
```

```
    - value of p: 0
-----------------------------------
-----------------------------------
- Incrementing p:
    - value of p: 10
- Decrementing p:
    - value of p: 0
-----------------------------------
-----------------------------------
- Incrementing p:
    - value of p: 10
- Decrementing p:
    - value of p: 0
-----------------------------------
```

A.4 References

[1] A. Burns, B. Dobbing and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. University of York Technical Report YCS-2003-348 January 2003.

[2] A. Burns and A. Wellings. HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. Elsevier Science, Amsterdam, NL, 1995. ISBN 0-444-82164-3.

[3] S. Mazzini, M. D'Alessandro, M. Di Natale, G. Lipari, and T. Vardanega. Issues in Mapping HRT- HOOD to UML. In G. Buttazzo, editor, Proceedings 15th Euromicro Conference on Real-Time Systems (ECRTS), pages 221-228. IEEE, July 2003.

[4] J. Medina and Á. García Cuesta. Model-Based Analysis and Design of Real-Time Distributed Systems with Ada and the UML Profile for MARTE. 16th Int. Conf. On Reliable Software Technologies, Ada-Europe'2011, Edinburg (UK), in Lecture Notes in Computer Science, LNCS Vol. 6652, pp. 89-102, June 2011.

[5] J. Medina and A. Pérez Ruiz. High Level Modeling for Real-Time Applications with UML & MARTE. Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13) WiP Session, pp. 13-16, July 2013.

This page intentionally left blank

Annex B
Semantics of SysML Blocks, ProxyPorts, and FlowProperties
(informative)

B.1 Introduction

SysML is a general purpose graphical modeling language that is used to describe systems in terms of their requirements, structure, behavior, and constraints. The core constructs used to model structure include blocks and their properties (e.g., value properties, part properties, constraint properties, and ports), associations, and connectors. These core structural constructs can be integrated with various behavioral constructs to describe how the system behaves.

SysML is intended to support many different types of behavior modeling paradigms. SysML, as with UML, includes activities, interactions and state machines for modeling behavior. Other forms of behavior, such as those described by differential equations, can be captured using SysML parametrics. The core structural elements can be integrated with other domain specific models and associated modeling paradigms to specify behavior in a way that is appropriate for the application.

The execution semantics for any particular type of behavior modeling approach must be specified in order to execute a model that is described in SysML. For example, parametrics can be executed by mapping them to various solvers such as Modelica, Mathematica, or Simulink, which each contain their own semantics.

The Semantics of a Foundational Subset for Executable UML (fUML) specifies the semantics for executing a subset of UML/SysML activities. The Precise Semantics of UML Composite Structures further extends these semantics to specify how the structural elements such as structured classes with ports impact the execution of activities. SysML requires further interpretation of these semantics for those areas of SysML that are different from UML.

Semantics specified in this clause only address a subset of SysML. This subset includes use cases where Blocks can have in or out FlowProperties (i.e., no inout). Blocks can also have ProxyPorts, typed by InterfaceBlocks with FlowProperties.

Subclause B.2 specifies the semantics for this subset. They describe how values written on out FlowProperties propagate through links existing between block instances, following links specified by connectors (either directly between parts, or between proxy ports on these parts). Subclause B.3 specifies a suite of test cases that can be used to demonstrate conformance to these semantics.

B.2 Semantics

B.2.1 Overview

Extensions to the normative Composite Structure execution model are depicted in Figure B.1, Figure B.2, and Figure B.3.
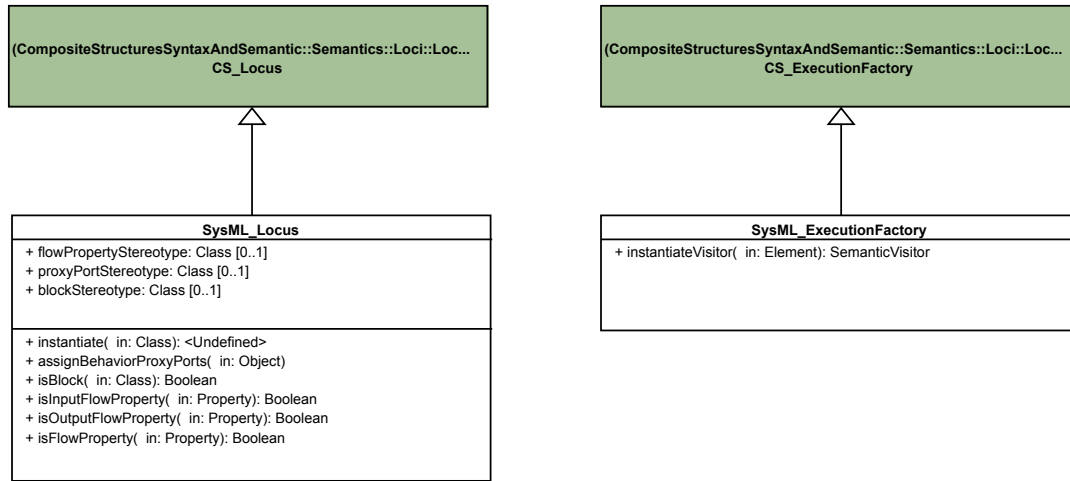
Figure B.1 - SysML Loci Extensions Diagram

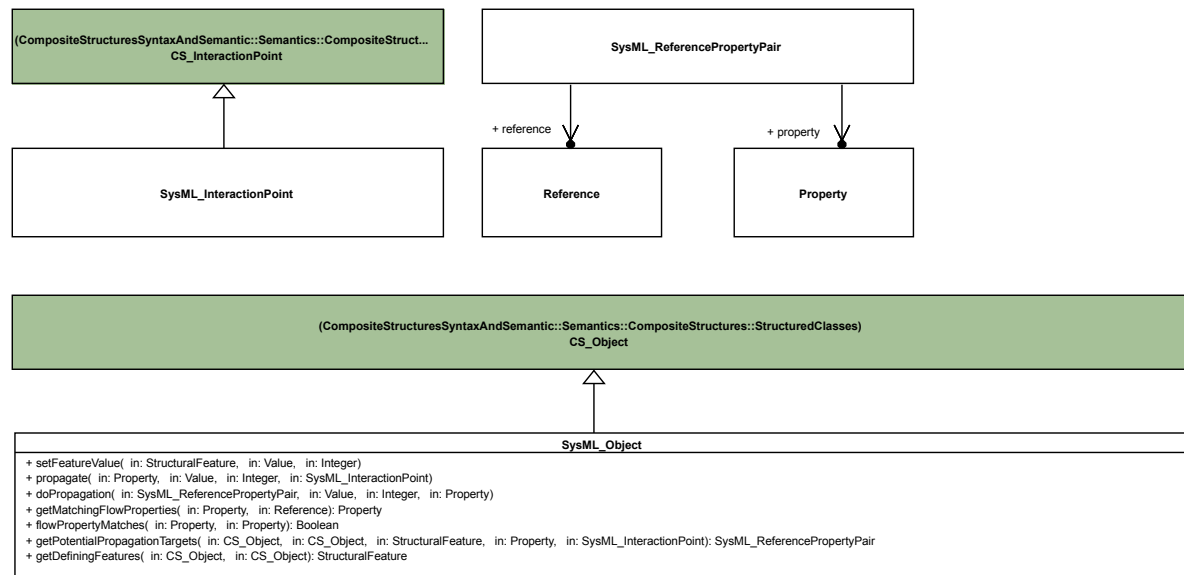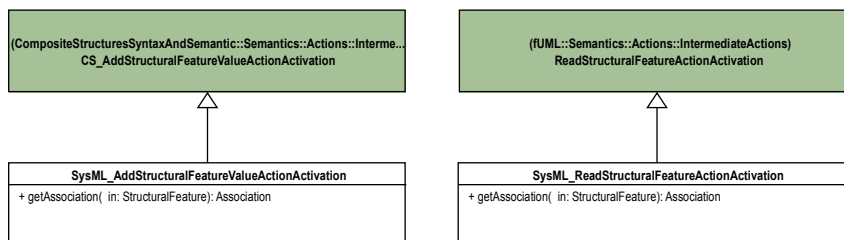**Figure B.2 - SysML Values Extensions Diagram**

Figure B.3 - SysML Actions Extensions Diagram

B.2.2 Class descriptions

B.2.2.1 SysML_AddStructuralFeatureValueActionActivation

SysML_AddStructuralFeatureValueAction extends CS_AddStructuralFeatureValueActionActivation to deal with
Associations that do not own all their ends. In SysML, properties typed by Blocks are defined with an Association

between the containing Block and the type of the Properties. As compared to fUML which requires these ends to be owned by the Association, SysML requires these ends to be owned by the containing Block.

Generalizations

- CS_AddStructuralFeatureValueActionActivation

Attributes

- None

Associations

- None

Operations

```
[1] public getAssociation(feature:StructuralFeature) : Association [0..1]
      // If the structural feature for the action of this activation is an
      // association end,
      // then get the associated association,
      // ONLY IF THIS FEATURE IS OWNED BY THE ASSOCIATION (SysML extension)
      Association association = null;
      if(feature instanceof Property) {
            association = ((Property)feature).association;
      }
      if (association != null) {
            int i = 1 ;
            boolean ownedByAssocation = false ;
            while (i <= association.ownedEnd.size() && !ownedByAssocation) {
                  Property current = association.ownedEnd.getValue(i - 1) ;
                  if (current == feature) {
                        ownedByAssocation = true ;
                  }
                  i = i + 1 ;
            }
            if (!ownedByAssocation) {
                  association = null ;
            }
      }
      return association;
```

B.2.2.2   SysML_ExecutionFactory

SysML_ExecutionFactory extends CS_ExecutionFactory to deal with instantiation of semantic visitors introduced by this extension.

Generalizations

- CS_ExecutionFactory

Attributes

- None

Associations

- None

Operations

```
[1] public instantiateVisitor(element:Element) : SemanticVisitor
      // Extends CS_ExecutionFactory to instantiate
      // SysML semantic visitors
      SemanticVisitor visitor = null ;
      if (element instanceof AddStructuralFeatureValueAction) {
            visitor = new SysML_AddStructuralFeatureValueActionActivation() ;
```

```
        }
        else if (element instanceof ReadStructuralFeatureAction) {
                visitor = new SysML_ReadStructuralFeatureActionActivation() ;
        }
        else {
                visitor = super.instantiateVisitor(element) ;
        }
        return visitor ;
```

B.2.2.3   SysML_InteractionPoint

SysML_InteractionPoint extends CS_InteractionPoint. The only kind of SysML port supported in this annex is behavior proxy ports. This kind of port does not require extensions to CS_InteractionPoint. However, since the other kind of SysML ports would require some extensions, this class is introduced to be used as a common ancestor to these potential extensions, that could be defined in extensions to the proposed execution model.

Generalizations

- CS_InteractionPoint

Attributes

- None

Associations

- None

Operations

- None

B.2.2.4   SysML_Locus

SysML_Locus extends CS_Locus by instantiating SysML_Objects instead of CS_Objects, in the case where the Type (from which instances are produced) is a Block. When instantiating a SysML_Object, it also deals with the automatic assignment of values for the behavior ProxyPorts of the Block typing the instantiated object. These values are SysML_InteractionPoint whose referents are the instantiated object itself. In other words, the instantiated object and the values for the behavior ProxyPorts of the typing Block are the same object. This mechanism is implemented in Operation assignBehaviorProxyPorts specified below.

SysML_Locus also owns utility properties and operations. All the *Stereotype properties below are references to Stereotype definitions from the SysML profile. These references are used to determine if model elements from a model being executed carry applications of the corresponding Stereotypes (see isBlock, isProxyPort, isFlowProperty, isInputFlowProperty, isOutputFlowProperty operations specified below).

Generalizations

- CS_Locus

Attributes

- blockStereotype : Class [0..1], the Block stereotype definition
- flowPropertyStereotype : Class [0..1], the FlowProperty stereotype definition
- proxyPortStereotype : Class [0..1], the ProxyPort stereotype definition

Associations

- None

Operations

```
[1] public assignBehaviorProxyPorts(object:Object)
  // For each FeatureValue of the given object,
  // if the definingFeature is a behavior ProxyPort,
  // assigns a value to this FeatureValue that is a
  // SysML_InteractionPoint, whose referent is the
  // given object itself
  int featureValueIndex = 1 ;
  while (featureValueIndex <= object.featureValues.size()) {
    FeatureValue currentFeatureValue = object.featureValues.get(featureValueIndex - 1) ;
    if (currentFeatureValue.feature instanceof Port) {
      Port p = (Port)currentFeatureValue.feature ;
      if (p.isBehavior && this.isProxyPort(p)) {
        ValueList values = new ValueList() ;
        CS_Reference owner = new CS_Reference() ;
        owner.referent = object ;
        owner.compositeReferent = (CS_Object)object ;
        SysML_InteractionPoint interactionPoint = new SysML_InteractionPoint() ;
        interactionPoint.definingPort = p ;
        interactionPoint.referent = object ;
        interactionPoint.owner = owner ;
        values.add(interactionPoint) ;
        currentFeatureValue.values = values ;
      }
    }
    featureValueIndex = featureValueIndex + 1 ;
  }

[2] public instantiate(type:Class):Object
      // If the type is a Block, instantiate a SysML_Object.
      // Otherwise behaves like in CS_Locus
      if (isBlock(type)) {
            Object_ object = null;
            object = new SysML_Object() ;
            object.types.add(type);
            this.add(object);
            object.createFeatureValues();
            this.assignBehaviorProxyPorts(object);
            return object;
      }
      else {
            return super.instantiate(type);
      }

[3] public isBlock(type:Class):Boolean

[4] public isFlowProperty(property:Property):Boolean

[5] public isInputFlowProperty(property:Property):Boolean

[6] public isOutputFlowProperty(property:Property):Boolean

[7] public isProxyPort(port:Port):Boolean
```

B.2.2.5 SysML_Object

SysML_Object extends CS_Object to deal with propagation of values when writing a value on an out FlowProperty of this object.

Generalizations

- CS_Object

Attributes

- None

Associations

- None

Operations

```
[1] public doPropagation(potentialTargets:SysML_ReferencePropertyPair,
                         values:Value[*],
                         position:Integer,
                         from:Property)
  // Performs the actual propagation for the given potentialTargets.
  // First, the potentialTargets list is partitionned by target objects.
  // For each partition (which is represented as a list of SysML_ReferencePropertyPair),
  // in the case where the list contains a single element, the propagation is done.
  // If the list contains more than 1 element,
  // tries to filer it by name, and repeats the operation recursively if the size
  // of the filtered list is 1.
  // The given values are written on objects identified by each SysML_ReferencePropertyPair.
  // If the given from property is typed by a block and has aggregation kind composite,
  // and if the property identified by SysML_ReferencePropertyPair is also composite,
  // then the value is moved rather than being simply copied (thereby dealing with
  // transfer of ownership).
  SysML_ReferencePropertyPairListList potentialTargetsByObject = new
SysML_ReferencePropertyPairListList() ;
  IntegerList added = new IntegerList() ;
  int i = 1 ;
  while (i <= potentialTargets.size()) {
    if (! added.contains) {
      added.addValue ;
      SysML_ReferencePropertyPair pair = potentialTargets.getValue(i - 1);
      SysML_ReferencePropertyPairList newList = new SysML_ReferencePropertyPairList() ;
      newList.addValue(pair) ;
      potentialTargetsByObject.addValue(newList) ;
      Object_ targetObject = pair.reference.referent ;
      int j = i + 1;
      while (j <= potentialTargets.size()) {
        SysML_ReferencePropertyPair pair2 = potentialTargets.getValue(j - 1) ;
        if (pair2.reference.referent == targetObject){
          newList.addValue(pair2) ; added.addValue(j - 1) ;
        }
        j = j + 1 ;
      }
    }
    i = i + 1 ;
  }
  FeatureValueList featureValuesToClean = new FeatureValueList() ;
  i = 1 ;
  while (i <= potentialTargetsByObject.size()) {
    SysML_ReferencePropertyPairList targets = potentialTargetsByObject.getValue(i - 1) ;
    if (targets.size() == 0){
      // No matching flow properties were found
    }
    else if (targets.size() == 1) {
      // Exactly one matching flow property was found
      ValueList valuesCopy = new ValueList() ;
      int valueIndex = 1 ;
      while (valueIndex <= values.size()){
        valuesCopy.add(values.getValue(valueIndex - 1).copy()) ;
        valueIndex = valueIndex + 1 ;
      }
      SysML_ReferencePropertyPair pair = targets.getValue(0) ;
      pair.reference.setFeatureValue(pair.property, valuesCopy, position);
      // if the type of from is a Class,
      // and if both properties from and pair.property are composite,
      // the value is moved, not copied.
      // the propagated values must be removed from property from
      if (from.getType() instanceof Class) {
        if (from.getAggregation() == AggregationKind.COMPOSITE_LITERAL &&
            pair.property.getAggregation() == AggregationKind.COMPOSITE_LITERAL) {
          FeatureValue fromFeatureValue = this.getFeatureValue(from) ;
          if (!featureValuesToClean.contains(fromFeatureValue)) {
            featureValuesToClean.addValue(fromFeatureValue) ;
          }
```

```
          }
        }
      }
      else {
        // Multiple matching flow properties
        // Tries to match by name
        SysML_ReferencePropertyPairList filteredByName = new SysML_ReferencePropertyPairList() ;
        int pairIndex = 1 ;
        while (pairIndex <= targets.size()) {
          SysML_ReferencePropertyPair currentPair = targets.getValue(pairIndex - 1) ;
          if (from.getName() != null && currentPair.property.getName() != null) {
            if (from.getName().equals(currentPair.property.getName())) {
              filteredByName.addValue(currentPair) ;
            }
          }
          pairIndex = pairIndex + 1 ;
        }
        if (filteredByName.size() == 1) {
          this.doPropagation(filteredByName, values, position, from);
        }
      }
      i = i + 1 ;
    }
    // cleaning features values of composite properties
    i = 1 ;
    while (i <= featureValuesToClean.size()){
      FeatureValue featureValue = featureValuesToClean.get(i - 1) ;
      featureValue.values.removeAll(values); i = i + 1 ;
    }

[2] public flowPropertyMatches(from:Property, to:Property):Boolean
    // Determines if the to FlowProperty matches the from FlowProperty
    // Property to matches Property from if it is an in FlowProperty and
    // if it is type compatible
    // Note that, by construction, Property from is always an out
    // FlowProperty
    // Not also that this solution does not account for propagation across
    // delegation connectors, only through assembly connectors.
    SysML_Locus sysML_Locus = (SysML_Locus)this.locus ;
    if (sysML_Locus.isInputFlowProperty(to)) {
      if (to.typedElement.type == null ||
          to.typedElement.type.isCompatibleWith(from.typedElement.type)) {
        return true ;
      }
    }
    return false ;
[3] public getDefiningFeatures(container:CS_Object, contained:CS_Object):StructuralFeature[*]
    // Retrieves the defining features for the given contained object,
    // in the context of the given container
    StructuralFeatureList definingFeatures = new StructuralFeatureList() ;
    FeatureValueList featureValues = container.getFeatureValues() ;
    int featureValueIndex = 1 ;
    while (featureValueIndex <= featureValues.size()) {
      FeatureValue currentFeatureValue = featureValues.getValue(featureValueIndex - 1) ;
      int valueIndex = 1 ;
      boolean found = false ;
      while (valueIndex <= currentFeatureValue.values.size() && !found) {
        Value currentValue = currentFeatureValue.values.getValue(valueIndex - 1) ;
        if (currentValue instanceof Reference) {
          if (((Reference)currentValue).referent == contained) {
            definingFeatures.addValue(currentFeatureValue.feature) ;
            found = true ;
          }
        }
        valueIndex = valueIndex + 1 ;
      }
      featureValueIndex = featureValueIndex + 1 ;
    }
    return definingFeatures ;

[4] public getMatchingFlowProperties(from:Property, context:Reference):Property[*]
    // Retries FlowProperties that match the given from property,
    // in the given context Reference.
```

```
    PropertyList matchingFlowProperties = new PropertyList() ;
    int featureValueIndex = 1 ;
    FeatureValueList featureValues = context.getFeatureValues() ;
    while (featureValueIndex <= featureValues.size()) {
      FeatureValue currentFeatureValue = featureValues.getValue(featureValueIndex - 1) ;
      StructuralFeature feature = currentFeatureValue.feature ;
      if (this.flowPropertyMatches(from, (Property)feature)) {
        matchingFlowProperties.addValue((Property)feature) ;
      }
      featureValueIndex = featureValueIndex + 1 ;
    }
    return matchingFlowProperties ;

[5] public getPotentialPropagationTargets(container:CS_Object,
                                          contained:CS_Object,
                                          feature:StructuralFeature,
                                          from:Property,
                                          sourceInteractionPoint:SysML_InteractionPoint
                                          ) : SysML_ReferencePropertyPair[*]
  // Retrieves potential propagation targets, in the case where:
  // - Values are written on the given contained object, on out FlowProperty from,
  //   through the given sourceInteractionPoint (can be null)
  // - Object contained is contained by the given container, acting as a value for
  //   the given StructuralFeature feature
  SysML_ReferencePropertyPairList potentialPropagationTargets = new SysML_ReferencePropertyPairList();
  FeatureValue featureValue = container.getFeatureValue(feature) ;
  Value referenceToContained = featureValue.values.getValue(0) ;
  ExtensionalValueList extensionalValues = this.locus.extensionalValues ;
  int linkIndex = 1 ;
  // Retrieves all links at the execution locus, and determines if these links
  // should be followed to propagate the values
  while (linkIndex <= extensionalValues.size()) {
    ExtensionalValue extensionalValue = extensionalValues.getValue(linkIndex-1) ;
    if (extensionalValue instanceof CS_Link) {
      CS_Link link = (CS_Link)extensionalValue ;
      StructuralFeature linkFeature = link.getFeature(referenceToContained) ;
      if (linkFeature != null) {
        // This link connect the given contained object to something
        FeatureValueList featureValues = link.getFeatureValues() ;
        FeatureValue currentFeatureValue = featureValues.getValue(0) ;
        Reference potentialTarget ;
        Reference source ;
        // retrieves the actual source and target for this link
        if (currentFeatureValue.feature != linkFeature) {
          potentialTarget =((Reference)featureValues.getValue(0).values.getValue(0)) ;
          source =((Reference)featureValues.getValue(1).values.getValue(0)) ;
        }
        else {
          potentialTarget =((Reference)featureValues.getValue(1).values.getValue(0)) ;
          source =((Reference)featureValues.getValue(0).values.getValue(0)) ;
        }
        boolean select = true ;
        // if a sourceInteractionPoint has been passed as an argument of this operation,
        // and if the source of the link is not this sourceInteractionPoint,
        // this link shall not be retained for propagation
        if (sourceInteractionPoint != null && ! (sourceInteractionPoint == source)) {
          select = false ;
        }
        if (select) {
          // if the potential target has flow properties that match the given from property,
          // it is added in the list of potential targets
          PropertyList matchingFlowProperties =
                            this.getMatchingFlowProperties(from, potentialTarget) ;
          int matchingFlowPropertyIndex = 1 ;
          while (matchingFlowPropertyIndex <= matchingFlowProperties.size()) {
            Property currentProperty =
                            matchingFlowProperties.getValue(matchingFlowPropertyIndex - 1) ;
            SysML_ReferencePropertyPair pair = new SysML_ReferencePropertyPair() ;
            pair.property = currentProperty ;
            pair.reference = potentialTarget ;
            potentialPropagationTargets.addValue(pair) ;
            matchingFlowPropertyIndex = matchingFlowPropertyIndex + 1 ;
          }
```

```
        }
      }
    }
    linkIndex = linkIndex + 1 ;
  }
  return potentialPropagationTargets ;

[6] public propagate(from:Property,
                     values:Value[*],
                     position:Integer,
                     interactionPoint:SysML_InteractionPoint)
  // Propagate values written on property from,
  // from the given interactionPoint, or through all
  // valid links if no interaction point is given
  CS_ObjectList containers = this.getDirectContainers() ;
  SysML_ReferencePropertyPairList potentialTargets = new SysML_ReferencePropertyPairList() ;
  int containerIndex = 1 ;
  while (containerIndex <= containers.size()) {
    CS_Object currentContainer =  containers.get(containerIndex - 1) ;
    StructuralFeatureList definingFeatures = this.getDefiningFeatures(currentContainer, this) ;
    int featureIndex = 1 ;
    while (featureIndex <= definingFeatures.size()) {
      StructuralFeature currentFeature = definingFeatures.getValue(featureIndex - 1) ;
      SysML_ReferencePropertyPairList targets =
                     this.getPotentialPropagationTargets(currentContainer, this,
                                                         currentFeature, from, interactionPoint) ;
      int targetIndex = 1 ;
      while (targetIndex <= targets.size()) {
        SysML_ReferencePropertyPair currentTarget = targets.getValue(targetIndex - 1) ;
        potentialTargets.addValue(currentTarget) ;
        targetIndex = targetIndex + 1 ;
      }
      featureIndex = featureIndex + 1 ;
    }
    containerIndex = containerIndex + 1 ;
  }
  this.doPropagation(potentialTargets, values, position, from);
[7] public setFeatureValue( feature:StructuralFeature, values:Value[*], position:Integer)
  // If the feature is a Port, behaves like in the composite structure execution model
  // If the feature is a an out FlowProperty,
  // behaves like in the composite structure execution model,
  // and then deal with the propagation of the written value.
  SysML_Locus sysML_Locus = (SysML_Locus)this.locus ;
  if (feature instanceof Port) {
    super.setFeatureValue(feature, values, position);
  }
  else if (feature instanceof Property) {
    Property p = (Property)feature ;
    super.setFeatureValue(feature, values, position);
    if (sysML_Locus.isOutputFlowProperty(p)) {
      this.propagate(p, values, position, null);
    }
  }
  else {
    super.setFeatureValue(feature, values, position);
  }
```

B.2.2.6   SysML_ReadStructuralFeatureActionActivation

SysML_AddStructuralFeatureValueAction extends fUML ReadStructuralFeatureActionActivation to deal with Associations that do not own all their ends (see SysML_AddStructuralFeatureValueActionActivation, B.2.2.1, for a description of cases where Associations do not own all their ends).

Generalizations

- ReadStructuralFeatureActionActivation (from fUML_Semantics::Semantics::Actions)

Attributes

- None

Associations

- None

Operations

```
[1] public getAssociation(feature:StructuralFeature) : Association [0..1]
        // If the structural feature for the action of this activation is an
        // association end,
        // then get the associated association,
        // ONLY IF THIS FEATURE IS OWNED BY THE ASSOCIATION (SysML extension)
        Association association = null;
        if(feature instanceof Property) {
                association = ((Property)feature).association;
        }
        if (association != null) {
                int i = 1 ;
                boolean ownedByAssocation = false ;
                while (i <= association.ownedEnd.size() && !ownedByAssocation) {
                        Property current = association.ownedEnd.getValue(i - 1) ;
                        if (current == feature) {
                                ownedByAssocation = true ;
                        }
                        i = i + 1 ;
                }
                if (!ownedByAssocation) {
                        association = null ;
                }
        }
        return association;
```

B.2.2.7   SysML_ReferencePropertyPair

SysML_ReferencePropertyPair enables to represent a pair composed of a reference to a Property, and a reference to a Reference. It is used to represent a target for the propagation of a value, when a value is written on an out FlowProperty of a SysML_Object (see B.2.2.5, Operations setFeatureValue and setFeatureValueOnInteractionPoint).

Generalizations

- None

Attributes

- None

Associations

- property : Property
- reference : Reference

Operations

- None

B.3 Test suites

This subclause describes suites of test cases that can be used to demonstrate conformance to the semantics specified in B.2. Test Suite 1 focuses on propagation of values when writing a value on an out flow property, in the context of an object classified by a Block without Ports. Test Suite 2 introduces variants of Test Suite 1 where objects are classified by Blocks with behavior ProxyPorts.

In the test cases specified below, the CS_DefaultConstructStrategy is not used. All systems are constructed manually, by instantiating and assigning values corresponding to parts. In addition, statements like `Default.createLink(x =>` `source, y => target)` specify the creation of a link typed by Association *Default*. These statements are required to establish connections between source and target objects, which can be values for parts (i.e., CS_References) or

ProxyPorts (i.e., SysML_InteractionPoints). *Default* is the generic Association defined in Annex C of this specification (i.e., two untyped owned ends, with ordered/unique multiplicity *).

B.3.1 Test Suite 1: Parts Directly Connected

B.3.1.1   Writing on FlowProperties Typed by ValueTypes

This test case addresses propagation of values when writing a value on an out FlowProperty typed by a ValueType. The object on which the value is written has no interaction points (i.e., it is classified by a Block without Ports). This source object is connected directly to a target object which has a single matching FlowProperty. When the value is written on the FlowProperty of the source object, this value is copied on the matching FlowProperty of the target object. Structural aspects of this test case are depicted in Figure B.4 and Figure B.5.
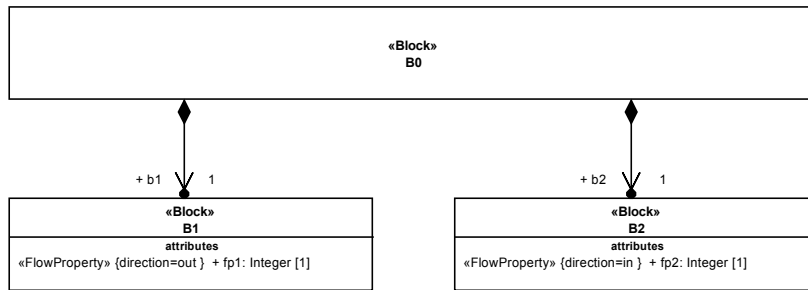
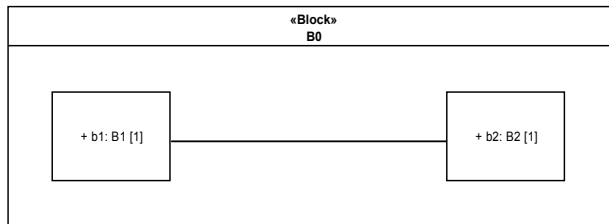**Figure B.4 - Block Definition Diagram**

Figure B.5 - Internal Block Diagram

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Untyped connector between parts with flow properties --") ;
  WriteLine("-- Multiplicity 1 on parts, connector ends, and flow properties --") ;
  WriteLine("-- Flow properties typed by a ValueType --") ;
  // Constructing the system
  let b0 : B0 = new B0() ;
  b0.b1 = new B1() ;
  b0.b2 = new B2() ;
  Default.createLink(x => b0.b1, y => b0.b2) ;
  // Testing
  b0.b1.fp1 = 4 ;
  WriteLine("") ;
  AssertTrue("Value successfully propagated", b0.b2.fp2 == 4) ;
  AssertTrue("Value has been copied (i.e., not moved)", b0.b1.fp1 == 4 && b0.b2.fp2 == 4) ;
  WriteLine("") ;
  WriteLine("-- End of test case --") ;
}
```

B.3.1.2   Writing on FlowProperties Typed by Blocks

This test case addresses propagation of values when writing a value on an out FlowProperty typed by a Block. The object on which the value is written has no interaction points (i.e., it is classified by a Block without Ports). This source object is connected directly to a target object which has a single matching FlowProperty. The two matching FlowProperties are composite Properties. The propagation of the written value thereby requires a transfer of ownership. It means that, when the value is written on the FlowProperty of the source object, this value is moved (i.e., not copied as for FlowProperties

typed by ValueTypes) on the matching FlowProperty of the target object. Structural aspects of this test case are depicted in Figure B.6 and Figure B.7.
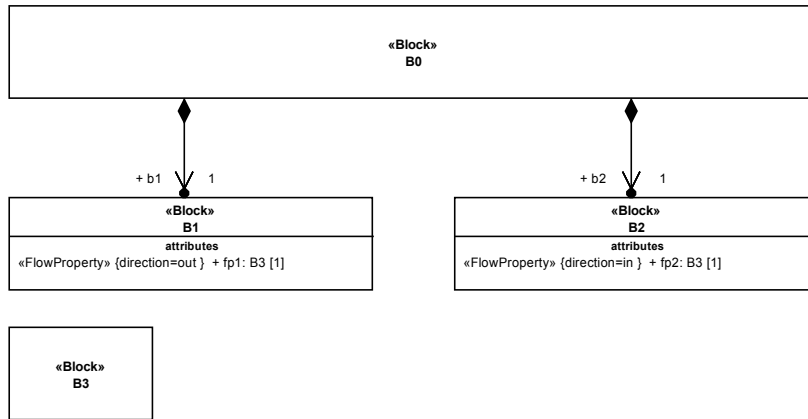

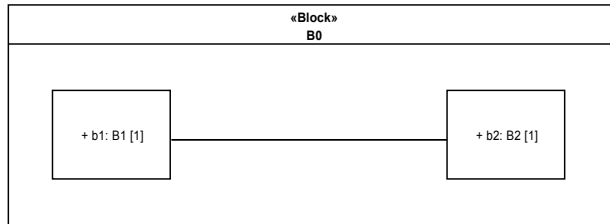**Figure B.6 - Block Definition Diagram**

Figure B.7 - Internal Block Diagram

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Untyped connector between parts with flow properties --") ;
  WriteLine("-- Multiplicity 1 on parts, connector ends, and flow properties --") ;
  WriteLine("-- Flow properties typed by a Block --") ;
  // Constructing the system
  let b3 : B3 = new B3() ;
  let b0 : B0 = new B0() ;
  b0.b1 = new B1() ;
  b0.b2 = new B2() ;
  Default.createLink(x => b0.b1, y => b0.b2) ;
  // Testing
  b0.b1.fp1 =b3  ;
  WriteLine("") ;
  AssertTrue("Value successfully propagated", b0.b2.fp2 == b3) ;
  AssertTrue("Value has been moved (i.e., not copied)", b0.b1.fp1 == null && b0.b2.fp2 == b3) ;
  WriteLine("") ;
  WriteLine("-- End of test case --") ;
}
```

B.3.2 Test Suite 2: Connectors Between Behavior ProxyPorts

B.3.2.1   Writing on FlowProperties Typed by ValueTypes

This test case addresses instantiation of Blocks with behavior ProxyPorts, as well as propagation of values when writing a value on an out FlowProperty typed by a ValueType. The object on which the value is written has an interaction point (i.e., it is classified by a Block with a ProxyPort). Since the defining port for this interaction point is behavior, the value of the interaction point is a reference to the object itself. The interaction point of this source object is connected to the interaction point of a target object, which has a single matching FlowProperty. When the value is written on the FlowProperty of the source object, this value is copied on the matching FlowProperty of the target object. Structural aspects of this test case are depicted in Figure B.8 and Figure B.9.
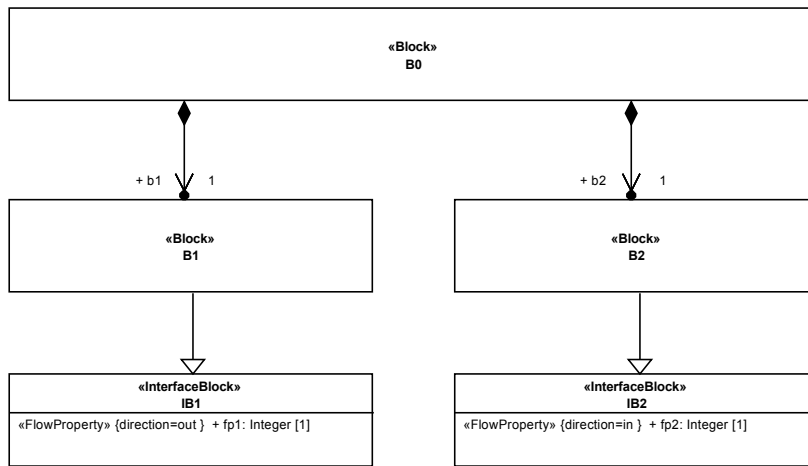
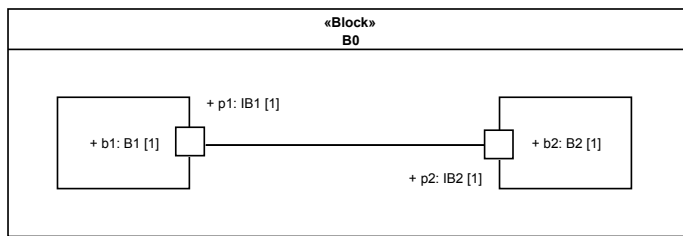**Figure B.8 - Block Definition Diagram**

Figure B.9 - Internal Block Diagram

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Untyped connector between behavior proxy ports with flow properties --") ;
  WriteLine("-- Multiplicity 1 on parts, ports, connector ends, and flow properties --") ;
  WriteLine("-- Flow properties typed by a ValueType --") ;
  // Constructing the system
  let b0 : B0 = new B0() ;
  b0.b1 = new B1() ;
  b0.b2 = new B2() ;
  Default.createLink(x => b0.b1.p1, y => b0.b2.p2) ;
  // Testing
  WriteLine("") ;
  AssertTrue("b0.b1 and b0.b1.p1 are the same object", b0.b1 == b0.b1.p1) ;
  AssertTrue("b0.b2 and b0.b2.p2 are the same object", b0.b2 == b0.b2.p2) ;
  b0.b1.p1.fp1 = 4 ;
  AssertTrue("Value successfully propagated", b0.b2.p2.fp2 == 4) ;
  AssertTrue("Value has been copied (i.e., not moved)", b0.b1.p1.fp1 == 4 && b0.b2.p2.fp2 == 4) ;
  WriteLine("") ;
  WriteLine("-- End of test case --") ;
}
```

B.3.2.2   Writing on FlowProperties Typed by Blocks

This test case addresses instantiation of Blocks with behavior ProxyPorts, as well as propagation of values when writing a value on an out FlowProperty typed by a Block. The object on which the value is written has an interaction point (i.e., it is classified by a Block with a ProxyPort). Since the defining port for this interaction point is behavior, the value of the interaction point is a reference to the object itself. The interaction point of this source object is connected to the interaction point of a target object, which has a single matching FlowProperty. The two matching FlowProperties are composite Properties. The propagation of the written value thereby requires a transfer of ownership. It means that, when the value is written on the FlowProperty of the source object, this value is moved (i.e., not copied as for FlowProperties typed by ValueTypes) on the matching FlowProperty of the target object. Structural aspects of this test case are depicted in Figure B.10 and Figure B.11.
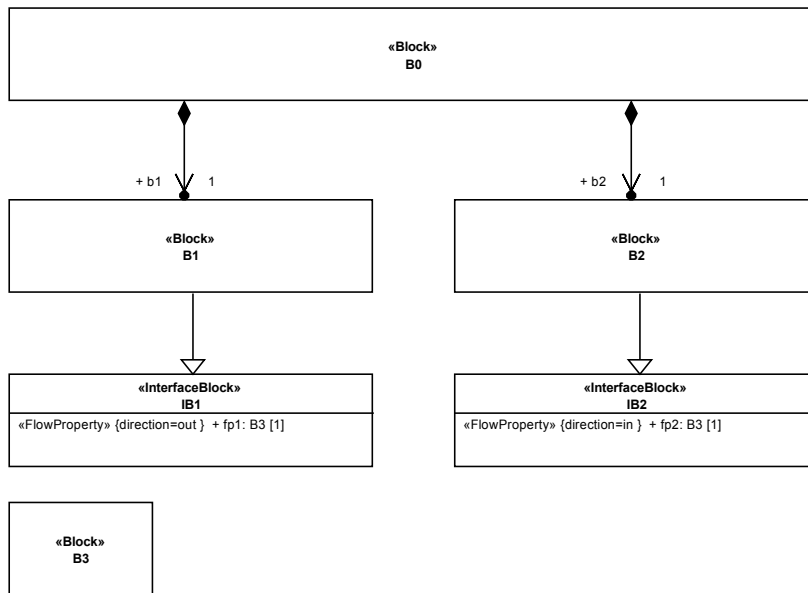
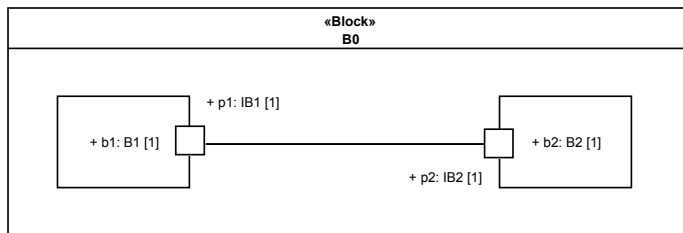**Figure B.10 - Block Definition Diagram**

Figure B.11 - Internal Block Diagram

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Untyped connector between behavior proxy ports with flow properties --") ;
  WriteLine("-- Multiplicity 1 on parts, ports, connector ends, and flow properties --") ;
  WriteLine("-- Flow properties typed by a Block --") ;
  // Constructing the system
  let b0 : B0 = new B0() ;
  b0.b1 = new B1() ;
  b0.b2 = new B2() ;
  Default.createLink(x => b0.b1.p1, y => b0.b2.p2) ;
  // Testing
  WriteLine("") ;
  AssertTrue("b0.b1 and b0.b1.p1 are the same object", b0.b1 == b0.b1.p1) ;
  AssertTrue("b0.b2 and b0.b2.p2 are the same object", b0.b2 == b0.b2.p2) ;
  let b3 : B3 = new B3() ;
  b0.b1.p1.fp1 = b3 ;
  AssertTrue("Value successfully propagated", b0.b2.p2.fp2 == b3) ;
  AssertTrue("Value has been moved (i.e., not copied)", b0.b1.p1.fp1 == null && b0.b2.p2.fp2 == b3) ;
  WriteLine("") ;
  WriteLine("-- End of test case --") ;
}
```

B.3.2.3   Block with Multiple Behavior ProxyPorts

This test case is a variant of the two previous test cases. The object on which the value is written has two interaction points (i.e., it is classified by a Block with two behavior ProxyPort). When the value is written on the FlowProperty of the source object, though a given interaction point, this value is propagated only through links where this interaction point is involved. Structural aspects of this test case are depicted in Figure B.12 and Figure B.13.
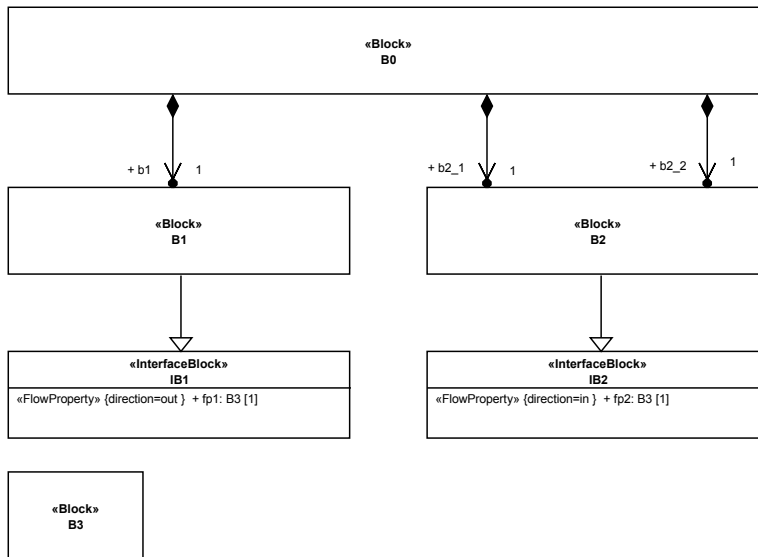
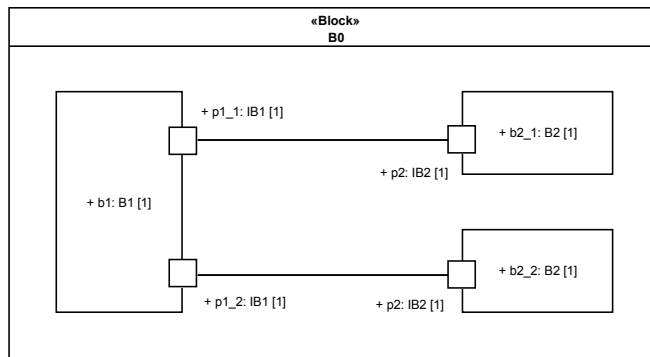**Figure B.12 - Block Definition Diagram**

Figure B.13 - Internal Block Diagram

The corresponding test case behavior is:

```
activity TestCaseBehavior() {
  WriteLine("-- Untyped connector between behavior proxy ports with flow properties --") ;
  WriteLine("-- Multiple Behavior Proxy Ports") ;
  WriteLine("-- Multiplicity 1 on parts, ports, connector ends, and flow properties --") ;
  WriteLine("-- Flow properties typed by a Block --") ;
  // Constructing the system
  b0 = new B0() ;
  b0.b1 = new B1() ;
  b0.b2_1 = new B2() ;
  b0.b2_2 = new B2() ;
  Default.createLink(x => b0.b1.p1_1, y => b0.b2_1.p2) ;
  Default.createLink(x => b0.b1.p1_2, y => b0.b2_2.p2) ;
  // Testing
  WriteLine("") ;
  let b3 : B3 = new B3() ;
  b0.b1.p1_1.fp1 = b3 ;
  AssertTrue("Value successfully propagated to b0.b2_1.p2", b0.b2_1.p2.fp2 == b3) ;
  AssertTrue("Value successfully propagated to b0.b2_2.p2", b0.b2_2.p2.fp2 == b3) ;
  WriteLine("") ;
  WriteLine("Writing directly on b0.b1.fp1...") ;
  let b4 : B3 = new B3() ;
  b0.b1.fp1 = b4 ;
  AssertTrue("Value successfully propagated to b0.b2_1.p2", b0.b2_1.p2.fp2 == b4) ;
  AssertTrue("Value successfully propagated to b0.b2_2.p2", b0.b2_2.p2.fp2 == b4) ;
  WriteLine("") ;
  WriteLine("-- End of test case --") ;
}
```

This page intentionally left blank

Annex C
A Generic Association for Instantiation of Untyped Connectors
(informative)

In this specification, CS_Link is used to represent connector instances, though there is no explicit relationship between CS_Link and Connector. Subclause 8.6.2.9 (CS_DefaultConstructStrategy) specifies how CS_Links are instantiated from Connectors.

A CS_Link can also be created using a CreateLinkAction. Note that this kind of Action relies 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, and has no consideration for Connectors (this is the reason why there is no explicit relationships between CS_Link and Connector). The manual instantiation of a Connector is thereby specified with a CreateLinkAction, where elements to be linked act as values for roles identified by the Connector. If the Connector is typed by an Association, the LinkEndData of the CreateLinkAction relies on this Association. In the case where the Connector is not typed, a generic Association (with untyped ends) can be used to specify the LinkEndData of the CreateLinkAction.

Machine readable file GenericAssociation.xmi contains such an Association called *Default*. Examples of how this association can be used are given in the SysML test suites, B.3.

This page intentionally left blank