The OOIS UML Profile for UI Capsule-Based Modeling

Dragan Milicev

University of Belgrade
Faculty of Electrical Engineering
Belgrade, Serbia
dmilicev@etf.rs

Introduction

This document contains a specification of the OOS UML profile for UI modeling based on capsules, wires, and pins. The specification follows the OMG's UML Superstructure Specification 2.2 [1]. The motivation and applicability of the concepts and features of the profile are explained elsewhere [2, 3].

Overview

This specification describes the contents of the OOIS IML UI profile that references the UML2 metamodel (Figure 1).

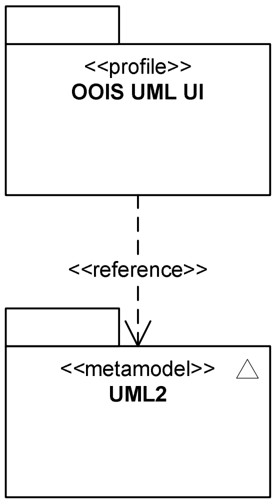


Figure 1

Capsule, Internal Structure, and Constructor

Capsule is an optional stereotype that can be applied to a Class (from `CompositeStructures::StructuredClasses`), as shown in Figure 2. A capsule has an optional presentation style (the `style` attribute), which is an implementation-specific style that defines the elements for formatting and appearance, such as colors, fonts, borders, padding, margins, etc. A capsule has an optional UI configuration context (the `guiContext` attribute), which is a GUI context as defined in OOIS UML [4]. If not specified in a capsule, each instance of the capsule inherits any of these from its owner capsule instance.

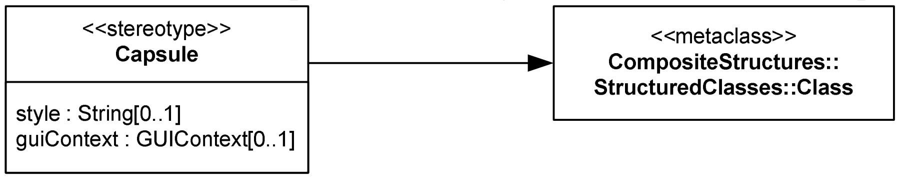


Figure 2

The internal structure of a capsule shown in a OOIS UML UI diagram, including parts (capsule references), ports, wires, conditional creation constraints, and the `parameters` compartment, is just a notational shorthand (i.e., a profiled diagrammatic representation) for the definition of a constructor operation with the given creational specification as in OOIS UML [4]. For example, the diagram in Figure 3 is a notational shorthand for the UML specification in Figure 4. Instance specifications, including links (wires), have the semantics of creational specifications in OOIS UML [4].

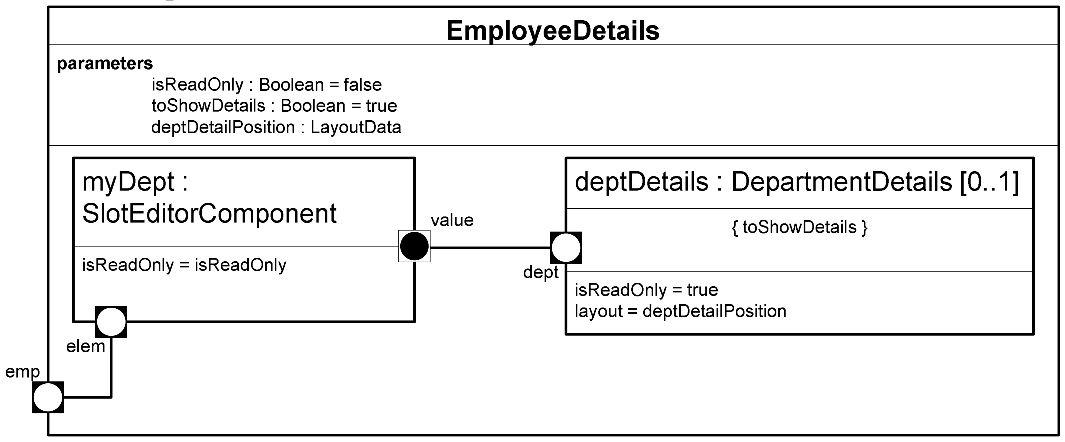

EmployeeDetails
parameters
isReadOnly : Boolean = false
toShowDetails : Boolean = true
deptDetailPosition : LayoutData
myDept : SlotEditorComponent
isReadOnly = isReadOnly
value
dept
deptDetails : DepartmentDetails [0..1]
{ toShowDetails }
isReadOnly = true
layout = deptDetailPosition
elem
emp

Figure 3

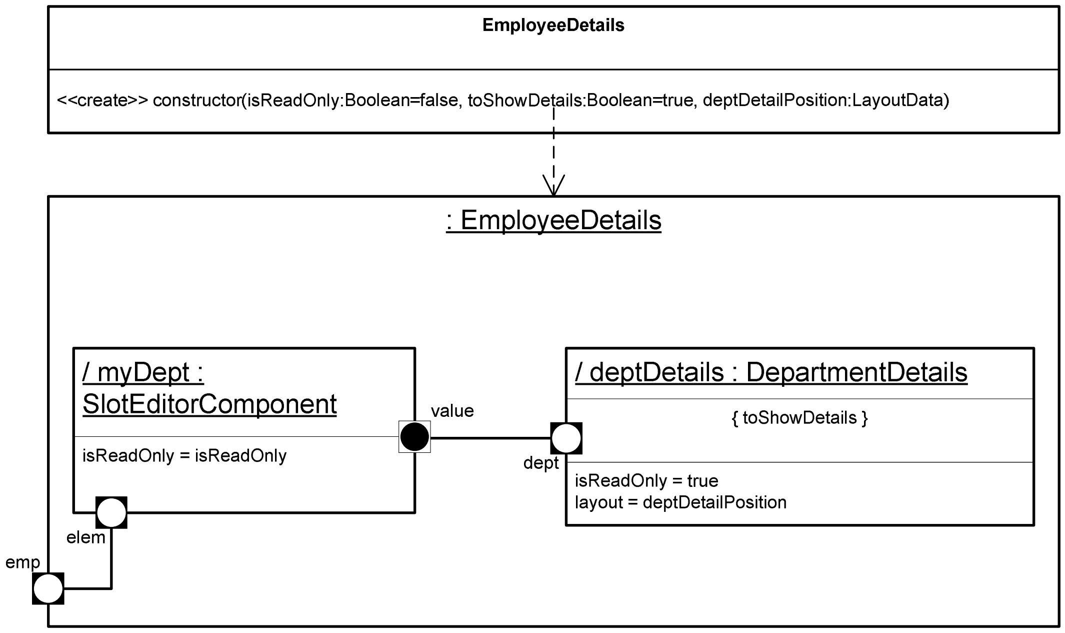

EmployeeDetails
<<create>> constructor(isReadOnly:Boolean=false, toShowDetails:Boolean=true, deptDetailPosition:LayoutData)
: EmployeeDetails
/ myDept : SlotEditorComponent
isReadOnly = isReadOnly
value
dept
/ deptDetails : DepartmentDetails
{ toShowDetails }
isReadOnly = true
layout = deptDetailPosition
elem
emp

Figure 4

Additional Constraints

1. All owned attributes of a capsule, except ports, must be either protected or private:
```
context Capsule:
  self.base_Class.ownedAttribute->
    forAll(m| not m.oclIsKindOf(Port) implies
      (m.visibility=protected or m.visibility=private))
```
2. All owned operations of a capsule, except constructors, must be either protected or private:
```
context Capsule:
```

```
self.base_Class.ownedOperation->
  forAll(m| m.extension$_create->size()=0 implies
    (m.visibility=protected or m.visibility=private))
```

Pins

Pin is a mandatory stereotype applied to UML ports (Figure 5).

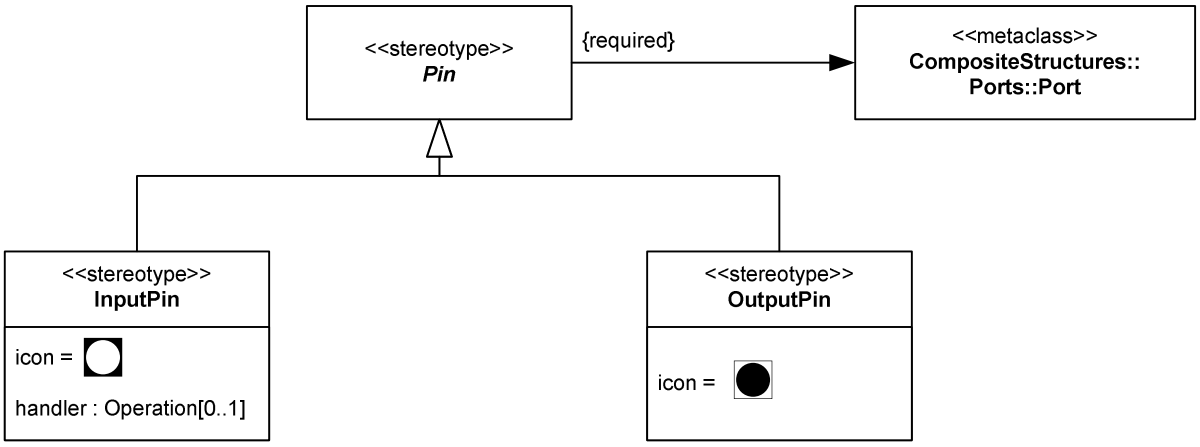
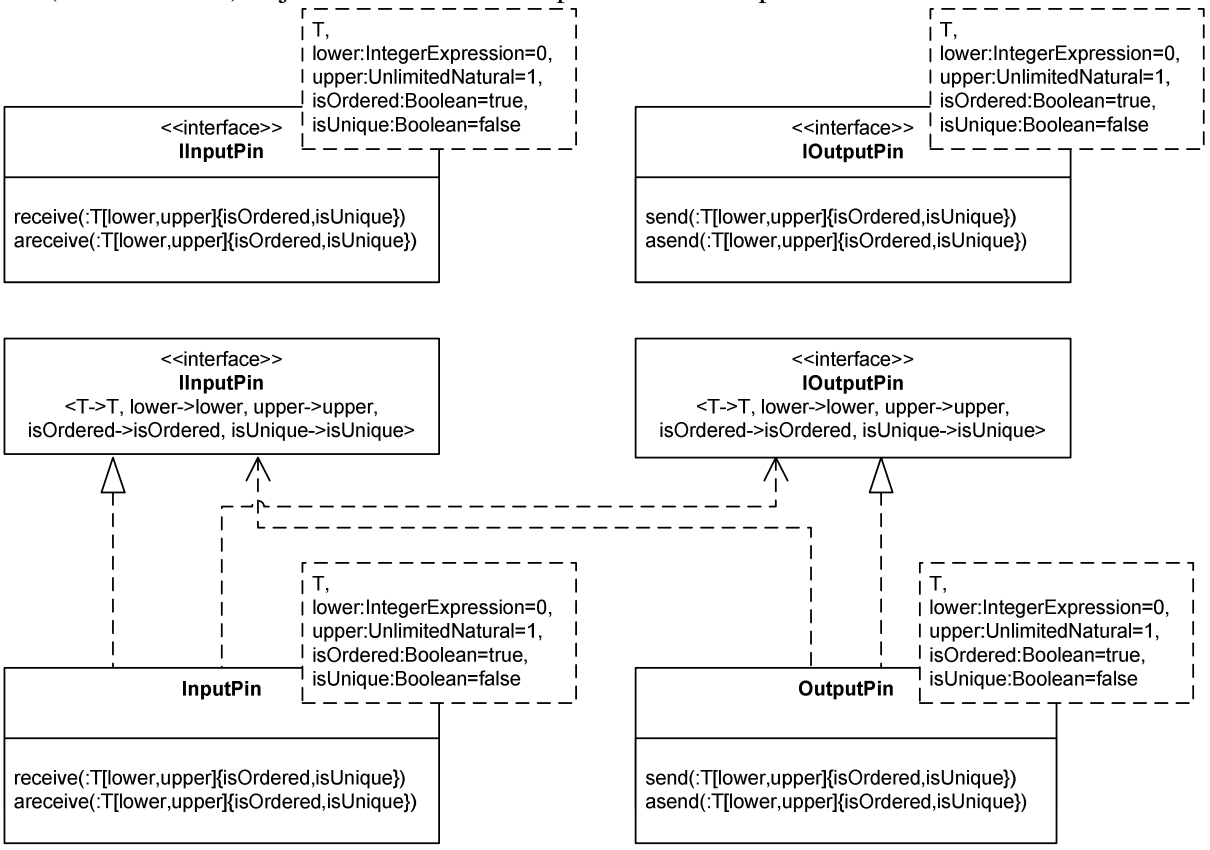


Figure 5

An input pin (`<<inputPin>>`), being a UML property, is typed with the paramaterized (template) class `InputPin`, and an output pin (`<<outputPin>>`) is typed with the paramaterized class `OutputPin` shown in Figure 6; these two classes are library classes of the profile. The template parameters define the type (`T`), multiplicity, ordering, and uniqueness of the (collections of) objects that can be transported over the pin.



T,
lower:IntegerExpression=0,
upper:UnlimitedNatural=1,
isOrdered:Boolean=true,
isUnique:Boolean=false
<<interface>>
IInputPin
receive(:T[lower,upper]{isOrdered,isUnique})
areceive(:T[lower,upper]{isOrdered,isUnique})
T,
lower:IntegerExpression=0,
upper:UnlimitedNatural=1,
isOrdered:Boolean=true,
isUnique:Boolean=false
<<interface>>
IOutputPin
send(:T[lower,upper]{isOrdered,isUnique})
asend(:T[lower,upper]{isOrdered,isUnique})
<<interface>>
IInputPin
<T->T, lower->lower, upper->upper,
isOrdered->isOrdered, isUnique->isUnique>
<<interface>>
IOutputPin
<T->T, lower->lower, upper->upper,
isOrdered->isOrdered, isUnique->isUnique>
T,
lower:IntegerExpression=0,
upper:UnlimitedNatural=1,
isOrdered:Boolean=true,
isUnique:Boolean=false
InputPin
receive(:T[lower,upper]{isOrdered,isUnique})
areceive(:T[lower,upper]{isOrdered,isUnique})
T,
lower:IntegerExpression=0,
upper:UnlimitedNatural=1,
isOrdered:Boolean=true,
isUnique:Boolean=false
OutputPin
send(:T[lower,upper]{isOrdered,isUnique})
asend(:T[lower,upper]{isOrdered,isUnique})

Figure 6

The class `OutputPin` realizes the parameterized interface `IOutputPin` (Figure 6). This is the (only) interface provided by an output pin. It requires the interface `IInputPin` provided by input pins. The operation `send` is available for sending (collections of) objects of type `T` over that output pin synchronously. The operation `asend` is available for sending (collections of) objects of type `T` over that output pin asynchronously.
Conversely, the class `InputPin` realizes the parameterized interface `IInputPin` (Figure 6). This is the (only) interface provided by an input pin. It requires the interface `IOutputPin` provided by output pins. The operation `receive` is called internally when (collections of) objects of type `T` are provided on that input pin synchronously. The operation `areceive` is called internally when (collections of) objects of type `T` are provided on that input pin asynchronously.
Interface pin is a synonym for a service port of a class. *Internal pin* is a synonym for a non-service port of a class.
The attribute `handler` of the stereotype `InputPin` is an optional specification of the operation of the class that owns the pin and that is called as the handler of the incoming message arriving on the pin.
A *behavior's pin* is a notational shorthand for an internal behavior pin. The notation in Figure 7a is a shorthand for the model in Figure 7b.

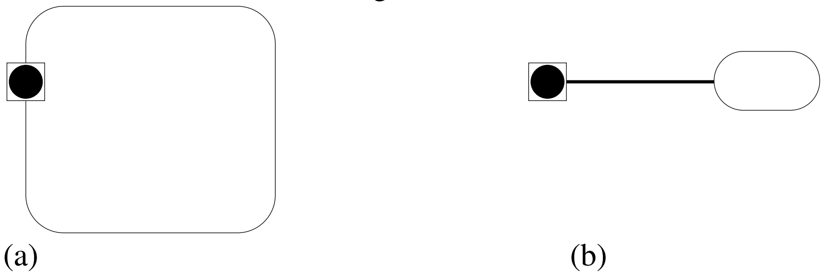


Figure 7

Additional Constraints

3. A pin must be a property of either a capsule or of a service provider (to be defined later), but not both:
```
context Pin:
  self.base_Port.class.extension$_capsule->size()=1 xor
  self.base_Port.class.extension$_serviceProvider->size()=1
```
4. An interface pin must have unspecified, public, or package visibility:
```
context Pin:
  self.base_Port.isService implies
  (self.base_Port.visibility.size()=0 or
   self.base_Port.visibility=public or
   self.base_Port.visibility=package)
```
5. An internal pin must have unspecified, private, or protected visibility:
```
context Pin:
  not self.base_Port.isService implies
  (self.base_Port.visibility.size()=0 or
   self.base_Port.visibility=private or
   self.base_Port.visibility=protected)
```
6. The type of an input pin must be `InputPin`:
```
context <<stereotype>>InputPin:
  self.base_Port.type=InputPin
```
7. The type of an output pin must be `OutputPin`:
```
context <<stereotype>>OutputPin:
  self.base_Port.type=OutputPin
```
8. Only a behavior pin can have a handler operation:
```
context Pin:
        self.handler->size()>0 implies
        self.base_Port.isBehavior
```

9. The handler operation of a pin must be a feature of the same class that owns the pin:

```
      context Pin:
        self.handler->size()>0 implies
        self.base_Port.class->allFeatures()->includes(self.handler)
```

Additional Operations

1. `Pin::conformsTo(other:Pin):Boolean`: Does this pin conform to the other given pin? The conformance is defined according to the type of the pin, that is, to the type `T`, multiplicity, ordering, and uniqueness of the parameterized class typing the port (`self.type`), as defined in OOIS UML [4].

SAP and SPP

Input SAP is a kind of input pin that is always internal (Figure 8). *Output SAP* is a kind of output pin that is always internal. An input or output SAP has a specified SPP (the attribute `spp`) it is bound to. The SPP has to be compatible with the SAP.

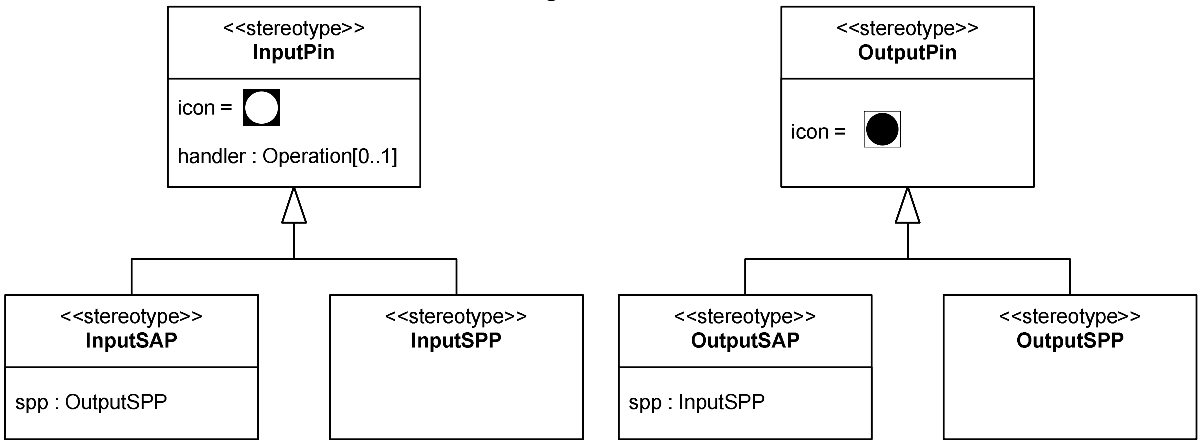
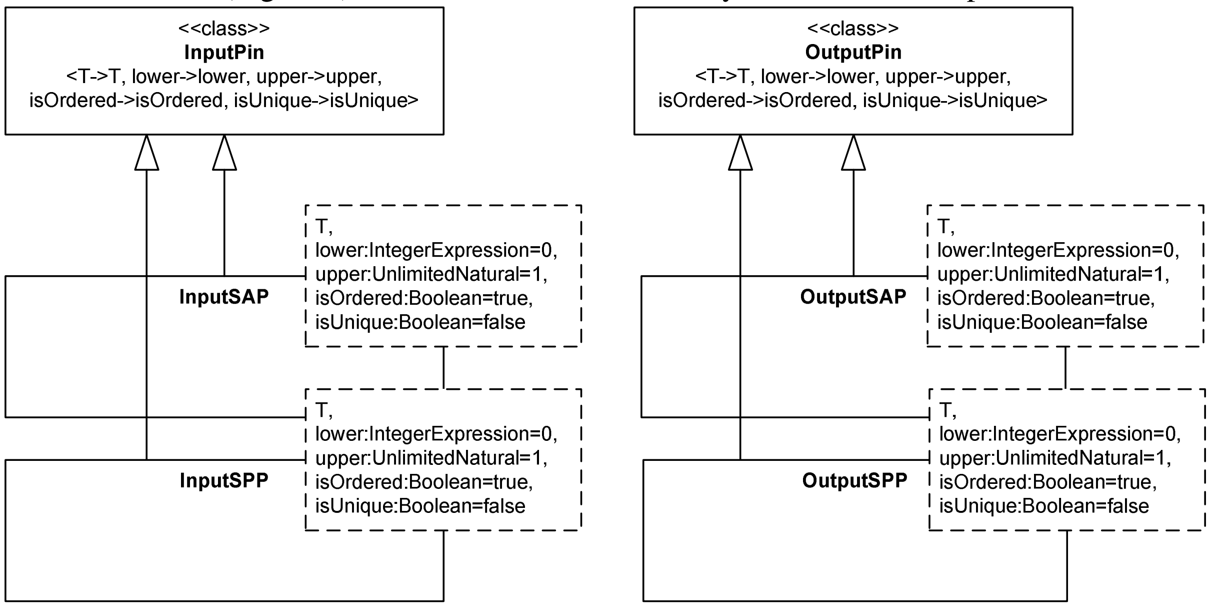


Figure 8

The type of an input SAP is the class `InputSAP` (Figure 9). The type of an output SAP is the class `OutputSAP` (Figure 9). These two classes are library classes from this profile.



<<class>>
InputPin
<T->T, lower->lower, upper->upper, isOrdered->isOrdered, isUnique->isUnique>
InputSAP
T, lower:IntegerExpression=0, upper:UnlimitedNatural=1, isOrdered:Boolean=true, isUnique:Boolean=false
InputSPP
T, lower:IntegerExpression=0, upper:UnlimitedNatural=1, isOrdered:Boolean=true, isUnique:Boolean=false
<<class>>
OutputPin
<T->T, lower->lower, upper->upper, isOrdered->isOrdered, isUnique->isUnique>
OutputSAP
T, lower:IntegerExpression=0, upper:UnlimitedNatural=1, isOrdered:Boolean=true, isUnique:Boolean=false
OutputSPP
T, lower:IntegerExpression=0, upper:UnlimitedNatural=1, isOrdered:Boolean=true, isUnique:Boolean=false

Figure 9

Service Provider is a stereotype that can be applied to a class (from `CompositeStructures::StructuredClasses`, Figure 10). A class that is a service provider provides access to its instance that serves requests from SAPs through its operation specified in `ServiceProvider::instanceOp`: for each incoming message sent via an SAP to an SPP that belongs to the service provider, the instance of the provider is obtained by calling the instance operation, and then the message is passed to the SAP of that instance.
Input SPP is a kind of an input port (Figure 8). The type of an input SPP is the class `InputSPP`, analoguous to `InputSAP`. *Output SPP* is a kind of an output port (Figure 8). The type of an output SPP is the class `OutputSPP`, analoguous to `OutputSAP`. An SPP serves as an interface port of its owner service provider. It serves the requests sent from SAPs.

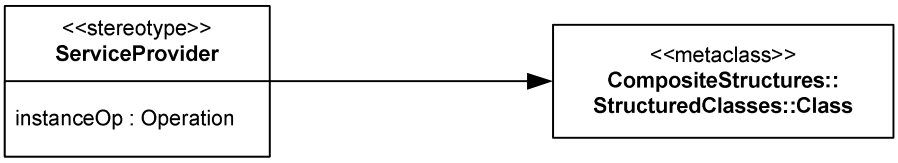


Figure 10

Additional Constraints

1. The owner of an input or output SAP must be a capsule:
```
context InputSAP:
  self.base_Port.class.extension$_capsule->size()=1
context OutputSAP:
  self.base_Port.class.extension$_capsule->size()=1
```
2. An input or output SAP is always internal:
```
context InputSAP:
  not self.base_Port.isService
context OutputSAP:
  not self.base_Port.isService
```
3. The type of an input SAP must be `InputSAP`:
```
context <<stereotype>>InputSAP:
  self.base_Port.type=InputSAP
```
4. The type of an output SAP must be `OutputSAP`:
```
context <<stereotype>>OutputSAP:
  self.base_Port.type=OutputSAP
```
5. The bound SPP and SAP have to conform one to the other:
```
context InputSAP:
  self.spp.conformsTo(self)
context OutputSAP:
  self.conformsTo(self.spp)
```
6. The instance operation of a service provider has to be a feature of the service provider's class, has to be static, and has to return an instance of that very class:
```
context InputSPP, OutputSPP:
  self.base_Port.class->allFeatures()->includes(self.instanceOp)
  and self.instanceOp.type=self.base_Port.class and
  self.instanceOp.isStatic
```
7. The owner of an SPP must be a service provider:
```
context InputSPP:
  self.base_Port.class.extension$_serviceProvider->size()=1
context OutputSPP:
  self.base_Port.class.extension$_serviceProvider->size()=1
```
8. An input or output SPP is always an interface port:
```
context InputSPP:
  self.base_Port.isService
context OutputSPP:
  self.base_Port.isService
```
9. The type of an input SPP must be `InputSPP`:

```
context <<stereotype>>InputSPP:
  self.base_Port.type=InputSPP
```

10. The type of an output SPP must be `OutputSPP`:

```
context <<stereotype>>OutputSPP:
  self.base_Port.type=OutputSPP
```

Wires

Wire is a required stereotype of UML connector (from `CompositeStructures::InternalStructures`, Figure 11).

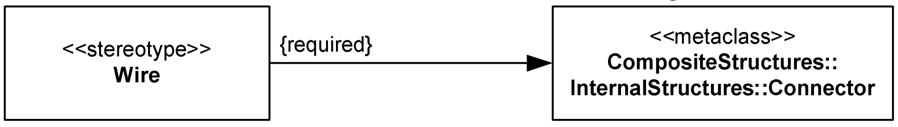


Figure 11

A wire can connect pins according to the basic UML rules, but also to additional rules defined in Table 1. This table specifies whether a pin *X* of a certain kind (input or output) can be connected with a wire to a pin *Y* of a certain kind. The label "Behavior" denotes an internal behavior pin, while the label "Part's" denotes a pin of the owner capsule's part (internal component). The label "N/A" means that the given connection is not allowed. The symbol "X→Y" means that the connection is allowed and that the messages flow through that wire from *X* to *Y*; in that case, *X* is called the *source*, and *Y* the *destination* pin. It can be concluded from this table that interface pins and SAPs behave the same (like pins for external communication), while internal behavior pins and part's pins behave the same (like pins for internal communication) in terms of connectivity and flow direction.

		Pin Y							
		Input				Output			
Pin X		Interface	SAP	Behavior	Part's	Interface	SAP	Behavior	Part's
Input	Interface	N/A	N/A	X→Y	X→Y	X→Y	X→Y	N/A	N/A
	SAP	N/A	N/A	X→Y	X→Y	X→Y	X→Y	N/A	N/A
	Behavior	Y→X	Y→X	N/A	N/A	N/A	N/A	Y→X	Y→X
	Part's	Y→X	Y→X	N/A	N/A	N/A	N/A	Y→X	Y→X
Output	Interface	Y→X	Y→X	N/A	N/A	N/A	N/A	Y→X	Y→X
	SAP	Y→X	Y→X	N/A	N/A	N/A	N/A	Y→X	Y→X
	Behavior	N/A	N/A	X→Y	X→Y	X→Y	X→Y	N/A	N/A
	Part's	N/A	N/A	X→Y	X→Y	X→Y	X→Y	N/A	N/A

Table 1

Additional Operations

1. `Wire::sourcePin():Pin[0..1]`, `Wire::destPin():Pin[0..1]`: Return the source and the destination pin of the wire according to the definitions given in Table 1 above.

Additional Constraints

1. A wire must have exactly two ends:

```
context Wire:
  self.base_Connector.end->size()=2
```

2. A wire can connect pins only:

```
context Wire:
  self.base_Connector.end->
  forEach(ce| ce.role.oclIsKindOf(Port) and
  ce.role.oclAsType(Port).extension$_pin->size()=1)
```

3. A wire can connect pins only according to the definitions in Table 1:

```
context Wire:
  self.sourcePin()->size()=1 and self.destPin()->size()=1
```

4. A wire can connect compatible pins only:

```
context Wire:
  self.sourcePin().conformsTo(self.destPin())
```

References

1. Object Management Group (OMG), *OMG Unified Modeling Language (OMG UML), Superstructure*, version 2.2, OMG Document Number: formal/2009-02-02, Standard document URL: http://www.omg.org/spec/UML/2.2/Superstructure, February 2009
2. Milicev, D., Mijailovic, Z., „Effective Modeling of User Interfaces for Large-Scale Applications,“ submitted for publication.
3. SOLoist framework, www.sol.rs
4. Milicev, D., *Model-Driven Development with Executable UML*, John Wiley & Sons, 2009