# UMO Profile for SoC Specification

**OMG Final Adopted Specification**
**January 2006**
**ptc/06-01-05**

**Date**: January 2006

# A UML Profile for SoC

OMG Adopted Specification

ptc/06-01-05

## OMG’s Issue Reporting Procedure

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

# Table of Contents

# 1 Scope

This specification defines a UML 2 profile for SoC (System on a Chip) design. The advantage of using this profile:

- This profile adds the SoC description capability to UML. This profile provides following representation capabilities :
  - Hierarchical representation of modules and channels, which are the fundamental elements of SoC.
  - Roles of modules.
  - Information transferred between modules using only one type of diagram.
  - Most diagrams used for SoC are internal structure diagrams extended by the profile.
  - Without the profile we had to represent them by description of class constructor function using sequence diagrams. This representation method lacks capability enough to explicitly show a module's role, which is fundamental information to analyze SoC.
- This profile does not require the use of any specific language such as SystemC.
  - SystemC is used for illustration purposes only.
- Semantics: for SoC behavior specified using *protocol*, which is a collaboration diagram, whose semantics is defined informally in UML. The other construct that requires clarification is the synchronocity semantics. At the PIM level we can maintain an asynchronous semantics, while on refinement to the PSM level, we can support synchronous semantics. Sometimes it can be globally asynchronous, while locally synchronous and vice-versa.

Our goal is to reuse existing UML as much as possible, and to define the profile as a minimal extension for modeling SoC designs.

# 2 Conformance

This specification defines five conformance points. Implementations must support at least one of these conformance points:

- Module description. Details are found in the specification
- Protocol
- Process
- Ports and channels including clock and reset
- connector

# 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.

- UML 2 Superstructure Specification ptc/04-10-02
- UML . Infrastructure Specification ptc/04-10-14
- MOF 2.0 Core Specification ptc/04-10-15
- UML Profile for Schedulability, Performance and Time formal/03-09-01
- UML Profile for System Engineering RFP ad/03-03-41

# 4 Terms and Definitions

None

# 5 Symbols

None

# 6 Additional Information

## 6.1 Changes to Adopted OMG Specifications

None.

## 6.2 How to Read this Specification

The rest of this document contains the technical content of this specification. Paragraphs marked as editorial comments are not normative; they are provided only to help explain design choices, trade-offs, and future work items.

## 6.3 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- Takashi Hasegawa, Fujitsu Limited
- Minoru Shoji, Fujitsu Limited
- Qiang Zhu, Fujitsu Laboratories LTD.
- Sreeranga P. Rajan, Fujitsu Laboratories of America, Inc.
- Tom Rutt, Fujitsu Corporation

- Masato Matsuoka, IBM Japan, Ltd. (Rational Software)
- Nobuaki Takahashi, IBM Japan, Ltd.
- Hisashi Suzuki, IBM Japan, Ltd.
- Takayuki Okada, IBM Japan, Ltd.
- Minoru Tomobe, NEC Corporation
- Toshiaki Minami, CANON INC.
- Tetsuji Saito, CANON INC.
- Yuichi Tsukada, CATS Co., Ltd.
- Koji Asari, Forte Design Systems.
- Masaki Yamada, Metabolics, Ltd.
- Tadayoshi Miyahara, RICOH COMPANY, LTD.
- Mutsumi Namba, RICOH COMPANY, LTD.
- Satoshi Okada, RICOH COMPANY, LTD.
- Masami Aihara, Toshiba Corporation
- Bran Selic, IBM Corporation
- Alan Moore, Artisan Software Tools LTD.
- Pete Rivett, Adaptive , Inc.
- Manfred R. Koethe, 88 Solutions Corporation
- Sumeet Malhotra, Unisys Corporation
- Sridhar Iyengar, IBM Corporation
- Stephen Mellor, Accelerated Technology (Mentor Graphics)
- Jean Hartmann, Siemens Corporate R&D

# 7 Profile for SoC

## 7.1 Overview

The term System-on-Chip (SoC) denotes the integration of the components of a computing or a communication system into a single chip. The components may be of digital, analog, or mixed-signal types. SoC system design is very similar to the well-known component-based design in software engineering. The main motivation for SoC is compactness and design reuse. Existing UML semantics have a strong focus on software requirements specification and therefore is inadequate for supporting modeling and specification of SoC designs. In order to support modeling and specification of SoC designs, we propose a UML profile for SoC. To augment the semantics for UML-based SoC design, stereotypes for each UML metaclass have to be defined by introducing a special notation and constraint for each SoC element. For this purpose we introduce *SoC structure diagrams*. SoC designers create both class diagrams and SoC structure diagrams for developing an *Executable* and *Realizable* system level model.

In the rest of this chapter and Chapter 8, we define stereotypes for SoC structure diagram, their notations and constraints. Additionally, in Annex B of this document, we show an example description of executable system level model corresponding to UML model. This example provides an insight into developing executable and realizable models and a method of modeling using template modules in SoC structure diagram. We have chosen SystemC as an executable system modeling language for illustrating the SoC design profile. However, another suitable system modeling language such as SystemVerilog could be used instead.

## 7.2 Stereotypes of SoC profile

Each element of the SoC model corresponds to the stereotyped UML metaclasses as shown in Table 7.1 and Figure 7.1. Constraints defined by these stereotypes are given using OCL in Chapter 8. A detailed description of each SoC Model element is provided in this section.

**Table 7.1 - Metaclasses for UML Profile for SoC**

| Section No. | SoC Model element | Stereotype | UML metaclass |
|---|---|---|---|
| 7.2.1 | Module | SoCModule | Class |
| 7.2.2 | Process | SoCProcess | Operation |
| 7.2.3 | Data | Data | Class |
| 7.2.4 | Controller | Controller | Class |
| 7.2.5 | Protocol Interface | SoCInterface | Interface |
| 7.2.6 | Channel | SoCChannel | Class |
| 7.2.7 | Protocol | SoCProtocol | Collaboration |
| 7.2.8 | Port | SoCPort | Port/Class |
| 7.2.9 | Module Part | SoCModuleProperty | Property |
| 7.2.10 | Channel Part | SoCChannelProperty | Property |
| 7.2.11 | Connector | SoCConnector | Connector |

Table 7.1 - Metaclasses for UML Profile for SoC

| | | | |
|---|---|---|---|
| 7.2.12 | Clock Port | SoCClock | Port |
| 7.2.13 | Reset Port | SoCReset | Port |
| 7.2.14 | Clock Channel | SoCClockChannel | Property |
| 7.2.15 | Reset Channel | SoCResetChannel | Property |
| 7.2.16 | Data Type | SoCDataType | Dependency |

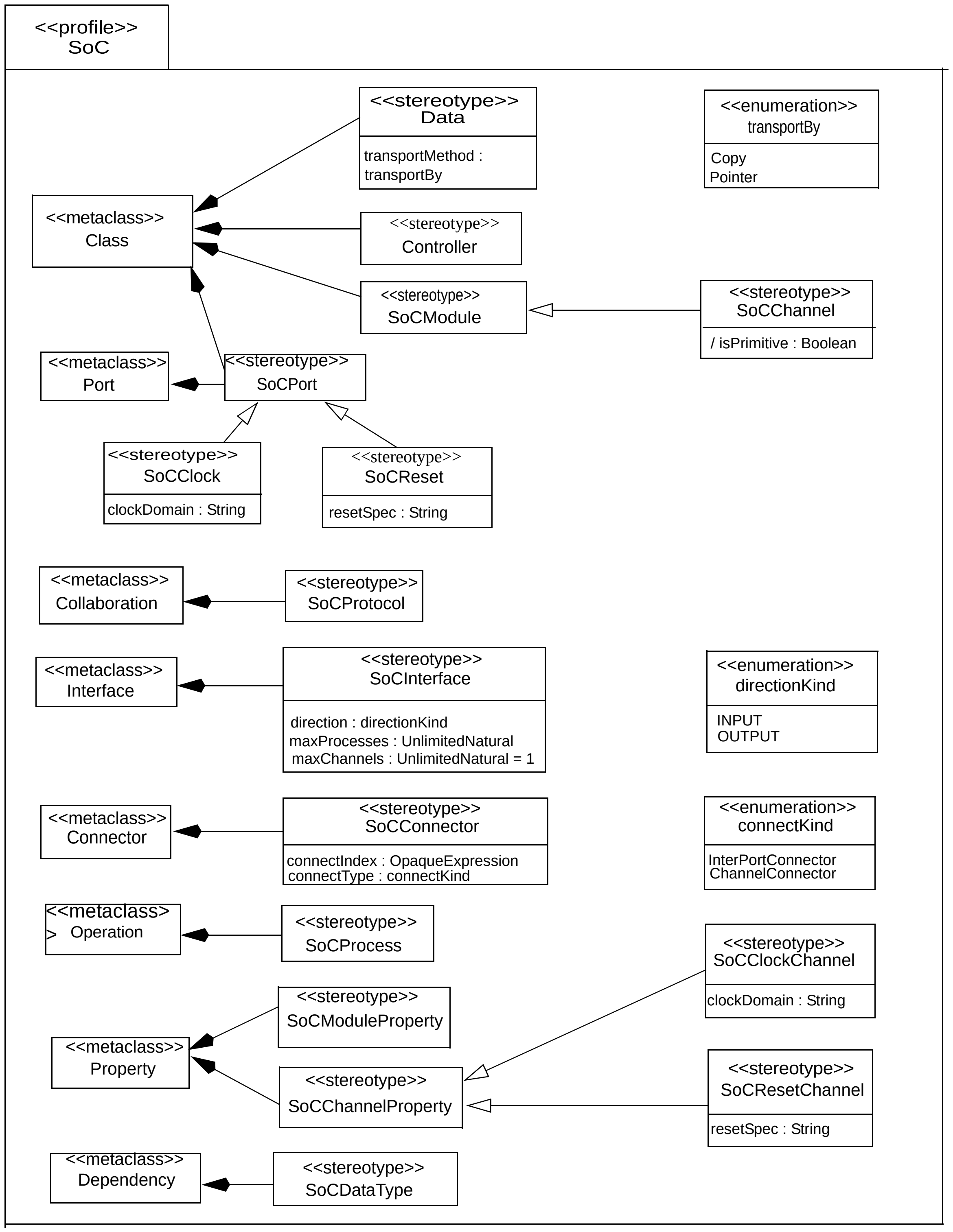

<<profile>>
SoC
<<stereotype>> Data
transportMethod : transportBy
<<enumeration>> transportBy
Copy
Pointer
<<metaclass>> Class
<<stereotype>> Controller
<<stereotype>> SoCModule
<<stereotype>> SoCChannel
/ isPrimitive : Boolean
<<metaclass>> Port
<<stereotype>> SoCPort
<<stereotype>> SoCClock
clockDomain : String
<<stereotype>> SoCReset
resetSpec : String
<<metaclass>> Collaboration
<<stereotype>> SoCProtocol
<<metaclass>> Interface
<<stereotype>> SoCInterface
direction : directionKind
maxProcesses : UnlimitedNatural
maxChannels : UnlimitedNatural = 1
<<enumeration>> directionKind
INPUT
OUTPUT
<<metaclass>> Connector
<<stereotype>> SoCConnector
connectIndex : OpaqueExpression
connectType : connectKind
<<enumeration>> connectKind
InterPortConnector
ChannelConnector
<<metaclass>> Operation
<<stereotype>> SoCProcess
<<stereotype>> SoCClockChannel
clockDomain : String
<<stereotype>> SoCModuleProperty
<<metaclass>> Property
<<stereotype>> SoCChannelProperty
<<stereotype>> SoCResetChannel
resetSpec : String
<<metaclass>> Dependency
<<stereotype>> SoCDataType

**Figure 7.1 - SoC Profile Stereotypes**

### 7.2.1 Module

Module is the base class for describing SoC hierarchical class structure.

**Tagged value added by this stereotype:** None

**Other restrictions:**

Only ports, constructors, and destructors are visible external to the module. Modules may consist of template modules and channels. Template parameter also can be used as sub-module, port and channel multiplicity, or as connector connection specifications. At least one member function of a module should not be a process.

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.2 Process

Process is a member function of the module that describes its behavior.

**Tagged value added by this stereotype:** None

**Other restrictions:**

There is no restriction on the number of processes contained in a module. There is restriction on the number of processes that can access a channel or port depending on the protocol interface realizing the channel.

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.3 Data

Only the Data class would be the target of communication between modules using the Channel Class.

**Tagged value added by this stereotype is:**

transportMethod : transportBy

Enumeration value that specifies the instance of the data transmission method between, namely Copy and Pointer. When the value is Copy then module will send the copy of data instance through the channel. When the value is Pointer then module transmits the pointer to the data instance. This pointer method is used when the data class that contains pointer (or reference) as its member and when the data instance should not be copied in order to avoid invalid data access.

**Other Restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.4 Controller

Only the member function of this class could be the module's process that is described in Section 7.2.1, "Module," on page 8 and Section 7.2.2, "Process," on page 8.

### 7.2.5 Protocol Interface

It defines the communication between modules, and provides an abstraction for the method of communication. All communication between modules must be realized through protocol interfaces. In many cases, the class that the Protocol Interface can transmit is restricted, and it shall be defined by the "data type" dependency described in Section 7.2.16, “Data Type,” on page 12, Section 7.6, “Protocol Interface,” on page 16, and Section 7.15, “Data Type,” on page 19. If a protocol interface is template protocol interface, then the dependency may have one template parameter as its target as described in Section 7.2.16, “Data Type and Section 7.15, “Data Type.

**Tagged values added by this stereotype:**

<u>Direction: directionKind</u>

directionKind is an enumeration type, whose elements are  INPUT and OUTPUT, that specify the incoming/outgoing direction of the protocol interface.

<u>maxProcesses: UnlimitedNatural</u>

Specifies the maximum number of processes that is accessible "through this protocol interface" to the channel realizing this protocol interface. Those processes may be owned by multiple modules (the number of processes must not exceed maxProcesses attribute value after instantiation of whole module hierarchy).

<u>maxChannels: UnlimitedNatural</u>

Specify the maximum number of channels that is connectable to a port instance having this protocol interface as its required interface (the number of channels must not exceed maxChannel attribute value after instantiation of whole module hierarchy).  The default value is 1.

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

## 7.2.6 Channel

All communication between modules shall be realized by the Channel through the Potocol Interfaces. Channel must be inherited from the Protocol Interface. Similar to the Module class, Channel may contain Process, Channel, Port and other modules. Its notation and specification of tagged value is also identical to that of module. Such Channels are called "Hierarchical Channels." Simple channels which provide primitive functionality such as hardware signals are called "Primitive Channels."

**Tagged value added by this stereotype:**

<u>isPrimitive : Boolean</u>

Specifies if the channel is primitive or not (hierarchical channel). This value is derived by the existence of other modules, processes, and channels contained in the channel. A design tool should be able to alter the channel notation using this value.

**Other restrictions:**

Each channel must be inherited from at least one INPUT direction protocol interface and one OUTPUT direction protocol interface.

(See Chapter 8, SoC Profile Constraints defined using OCL)

## 7.2.7 Protocol

Protocol describes the relationship between protocol interfaces and a channel. As with the UML diagrams, notation is same as that of collaboration diagram.

**Tagged value added by this stereotype:** None

**Other restrictions:**

Each protocol must have (a) at least one incoming protocol interface, (b) at least one outgoing protocol interface, and (c) at least one channel inherited directly or indirectly from protocol interfaces.

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.8 Port

It defines the protocol interface used for communication external to modules.

**Tagged value added by this stereotype:** None

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.9 Module Part

It is a kind of property (part) typed by the module class. It is usually called as "sub-module."

**Tagged value added by this stereotype:** None

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL).

### 7.2.10 Channel Part

It is a kind of property (part) typed by the channel class. Typically the channel must be shown as this property, and does not exist independently (channel must occur as property/part of a module).

**Tagged value added by this stereotype:** None

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.11 Connector

Connector connects ports or connects ports and a channel. If a connector is connected to either of port, module part, or channel part with multiplicity of 1, then the specification of connection between each instances may be defined by OCL. The specification rule is described in the document "SoC profile constraints defined by OCL.".

**Tagged value added by this stereotype:** None

**Other restrictions:**

In a module (here we call parent module), ports contained in the module can be connected only to the other ports contained in the module properties (parts). In this case, we call it as a sub-module contained in the parent module. In addition, channel property owned by parent module can be connected only to the ports of sub-modules. Ports of sub-modules can be connected to the ports or channels owned by the parent module. Connection without connectors is prohibited. In addition, when ports are connected directly, the type of protocol interfaces of these ports must be compatible with each other.

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.12 Clock Port

This class is inherited from port, and it specifies clock input or clock output port. It is typically used in or after the "structural design" phase, which requires investigation of synchronization schemes.

**Tagged values added by this stereotype:**

<u>clockDomain : domainName</u>

domainName specifies the name of "clock domain," which consists of clock cycle, phase, duty-ratio, and the method of controlling the gated-clock. There is no limitation on the name or the specification of clock domains. Connection between the ClockPorts, or connection between ClockPort and ClockChannel (explained later) with a different clockDomain attribute is prohibited. It is possible to connect different clock domains only if the connection is  through the process that converts the clock domain..

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.13 Reset Port

This class is inherited from port, and it specifies reset input or reset output port. It is typically used in or after the "structural design" phase, which requires investigation of reset schemes.

**Tagged value added by this stereotype:**

<u>resetSpec : resetSpecName</u>

resetSpecName specifies the name of "reset specification", which consists of reset timing specification (synchronous / asynchronous reset, polarity etc.). There is no restriction on the name or the specification of reset specification. Connection between the resetPorts, or connection between resetPort and resetChannnel (explained later) with a different resetSpec attribute is prohibited. It is possible to connect different reset specifications only if it is through a process that converts reset specifications..

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.14 Clock Channel

This is a property (part) defining the channel instance(s) that transmits the execution events to synchronous processes. It is typically used in or after  "structural design" phase, which requires consideration of synchronization schemes.

**Tagged value added by this stereotype :**

<u>clockDomain : domainName</u>

domainName specifies the name of "clock domain". Refer to the clockPort for the definition of clock domain. Connection to a clockPort is limited to that with same clockDomain attribute.

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.15 Reset Channel

This is a property(part) defining channel instance(s) that transmits the reset events to processes. It is typically used in or after the "structural design" phase, which requires the consideration of resetting schemes.

**Tagged value added by this stereotype:**

<u>resetSpec : resetSpecName</u>

ResetSpecName specifies the name of "reset specification." Refer to the resetPort for the definition of reset specification. Connection to a resetPort is limited to that with same resetSpec attribute.

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

### 7.2.16 Data Type

This is an extension of UML dependency and points at a data class which the protocol interfaces transmit. If the protocol interface is a template, then this dependency may point to one of the template parameter classes.

**Tagged value added by this stereotype:** None

**Other restrictions:**

(See Chapter 8, SoC Profile Constraints defined using OCL)

## 7.3 SoC Structure diagram

SoC structure diagram is used to describe the structure of SoC. The following elements are used:

1. Module and Module Part
2. Port (Single or Multiple)
3. Protocol Interface
4. Channel Part
5. Connector
6. Protocol
7. Process
8. Clock Port
9. Reset Port

10.Clock Channel

11.Reset Channel

12.Data Type

This section explains the notations in SoC structure diagrams.

## 7.4 Module and Module Part

Module hierarchy is explained by a module and other module parts contained in it as shown in Figure 7.2. When module part hierarchy of module is displayed using SoC structure diagram, the expanded form is typically used (in this diagram, elements other than property name and class name of module in the structure diagram are omitted). The method of expanding every module in a hierarchy is not defined in UML specification, although it is necessary for large-scale system (SoC) specification notation. However, it is also difficult for designers to operate whole hierarchy expressed in expanded form drawn on one diagram for large scale system design. There should be alternative method of suppressing the expansion of module for selected parts of a hierarchy. To express the module whose hierarchy is suppressed on a diagram, there should be an icon as shown in Figure 7.3.  This diagram shows sub-module C2 shown in Figure 7.2, without expanding the sub-module hierarchy.

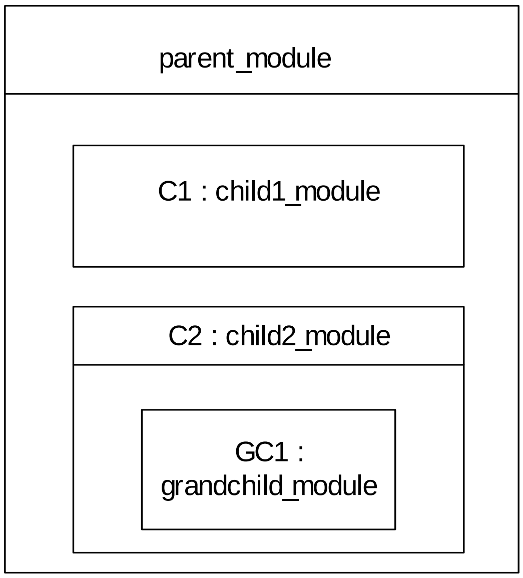


**Figure 7.2 - module hierarchy in structure diagram**

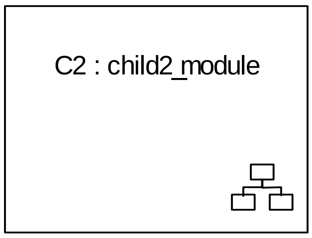


**Figure 7.3 - The module which abbreviates sub module hierarchy**

The initialization parameters of sub-modules and channels owned by a module must be passed as the module's parameter, and passed to the sub-modules and channels during the module's construction.

Module can also be parameterized (templated). The notation of these parameterized module instances follow the UML notation. Templated module is shown in uninstantiated form in structure diagram in the same way as templated class. Template parameter (which may be a constant) must be specified in some form by the designer (tools will not assume the parameter by the port types because it is too complex) when it is instantiated.

Furthermore, it is possible to configure the module hierarchy using the constructor parameter. In this case, uninstantiated form is used on SoC structure diagram. Constructor parameter is specified in SoC structure diagram as shown in the following figure.

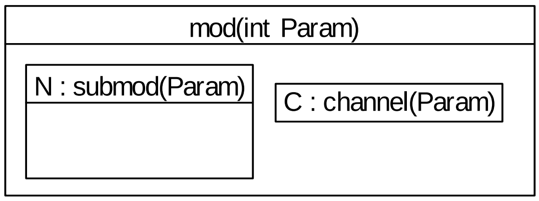


**Figure 7.4 - An example of constructor arguments usage**

Sequence diagrams may be used to define complex sequences during module construction as shown in figure 6. This diagram shows an example of a constructor sequence that reads a file for each module instance name. To construct module hierarchy with these sequences, the module may have pointers to module parts or channel parts instead of having instances as its members. These members are assigned the address of instances when instantiated in the constructor body. In that case, the module part or channel part must be marked by attributes. The attribute specification depends on the SoC design tools and therefore is not part of this profile.

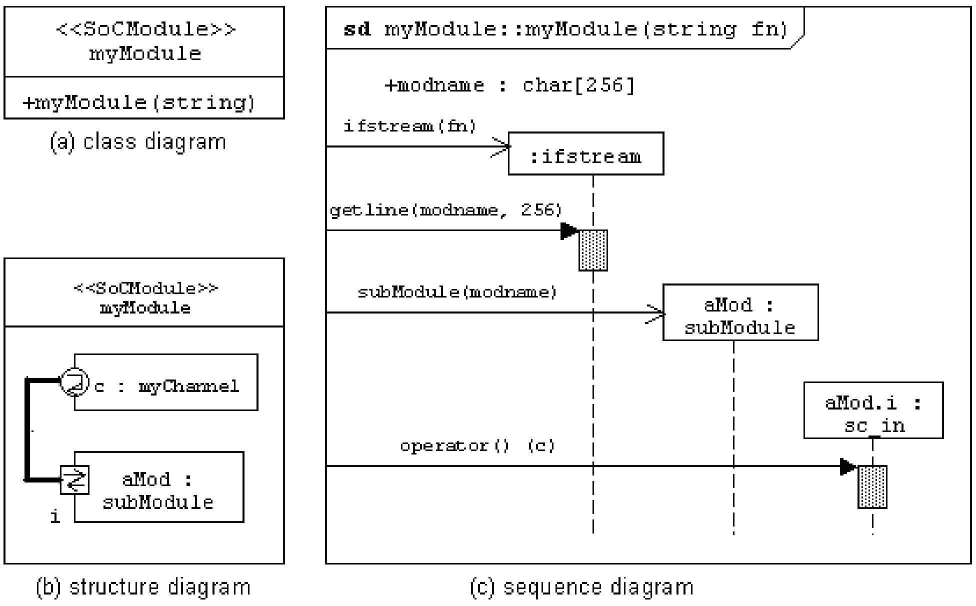


**Figure 7.5 - Complex constructor definition example**

## 7.5 Port and the Property

The module class has to consist of a process member function that defines a branch or a merge when a branch or a merge has to be realized to/from a port. This definition is similar to the definition of typical module.

Single port is the port with multiplicity equals one, and is shown as arectangle two opposite-facing arrows as shown in Figure 7.6.

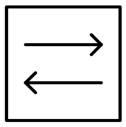


**Figure 7.6 - Icon of Port**

With the multi port that has multiplicity of 2 or more, another icon is used. The following icon is used for multi port.

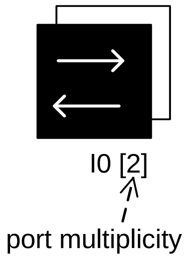


**Figure 7.7 - Icon of Multi Port**

## 7.6 Protocol Interface

Protocol interface on the structure diagram is expressed by the circle with U shaped arrow in it, written on a rectangle showing a channel as shown in Figure 7.8.

Protocol interface must have a dependency (Data Type dependency) to a data class. If a protocol interface is inherited from another protocol interface, either general interface or specialized interface must have the dependency. In other words, there should be exactly one data type dependency when referred to by ports or channels.

As the design changes, modification and implementation of protocol interface become necessary. For this purpose, modification and implementation of protocol are expressed in the generalization of protocol and protocol interface just as it is done in software modeling.

## 7.7 Channel, channel part and its property

In SoC structure diagram, channel parts are displayed as owned property (part) of a module, and are drawn in a rectangle. In the case of channel part typed by primitive channel, it is possible to omit the rectangle and express the channel only by a text string next to the connector consisting of the instance name and class name as shown in figure 9. In this case, a connector must be drawn using a solid line. For hierarchical channels, it is possible to specify the internal structure of the channel using same notation as modules as shown below (except that it is drawn in dotted line instead of continuous line).

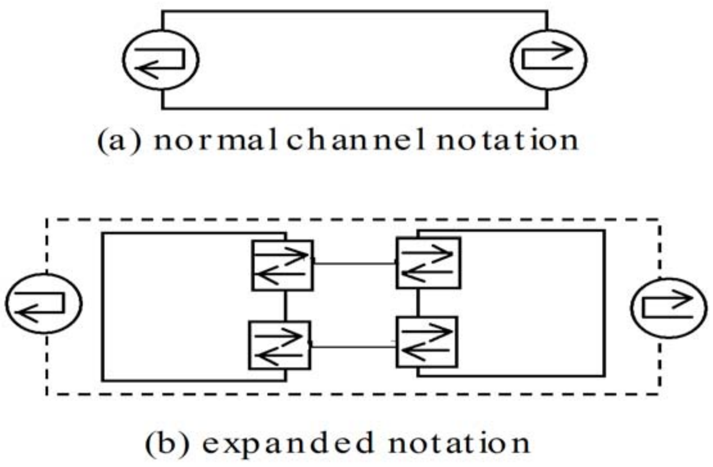
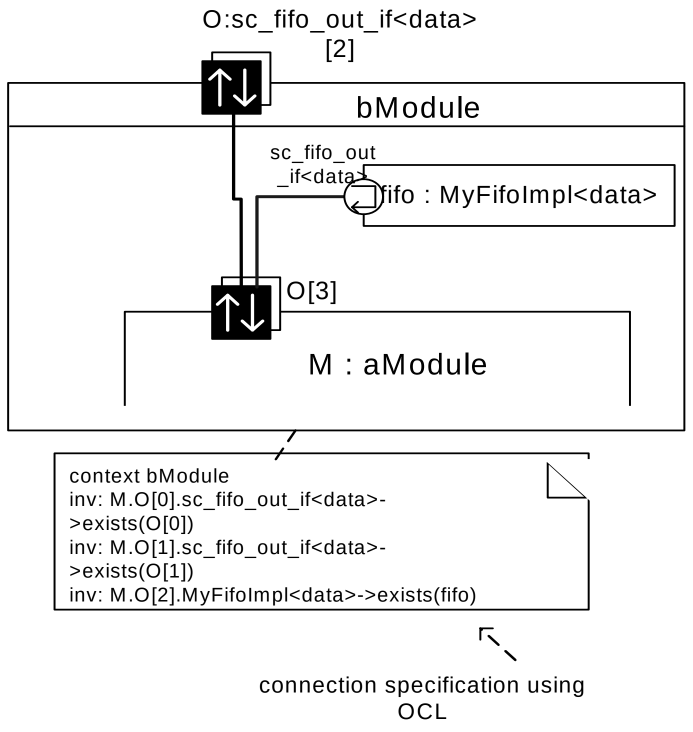


**Figure 7.8 - Hierarchical channel notation variations**

## 7.8 Connector

Connector is expressed as a line drawn between a channel and port, or between ports. As shown in Figure 7.9, connector between ports or connector omitting rectangle, which shows a channel, must be drawn in a solid line. As depicted in Figure 7.1, the attribute **connectType** associated with the **<<SoCConnector>>** stereotype has an <<**enumeration**>> stereotype value called **connectionKind**. <<**connectionKind**>> has two attributes namely **InterPortConnector** between two ports and **ChannelConnector** between a channel and a port. These attributes and values may be automatically extracted from the diagram. Connection index must be shown in a tagged value notation.



context bModule
inv: M.O[0].sc_fifo_out_if<data>-
>exists(O[0])
inv: M.O[1].sc_fifo_out_if<data>-
>exists(O[1])
inv: M.O[2].MyFifoImpl<data>->exists(fifo)

**Figure 7.9 - Specification of connecting instances**

## 7.9 Protocol

Protocol is a kind of Collaboration, expressed as in the following figure.

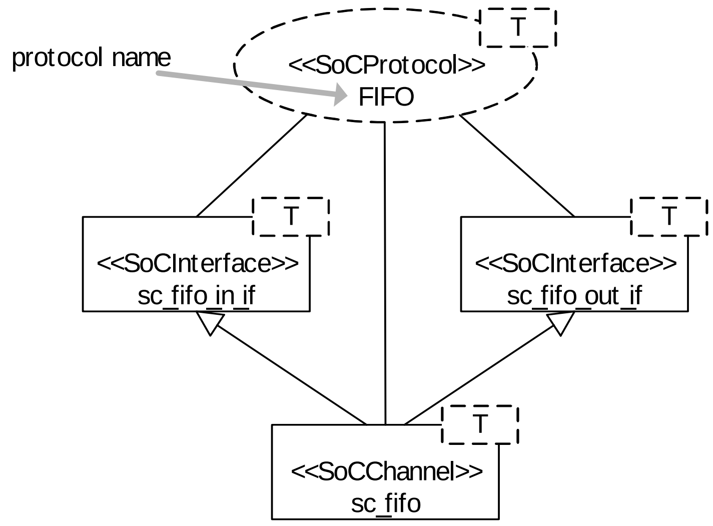


**Figure 7.10 - Protocol**

## 7.10 Process

Process is a kind of Operation described below.

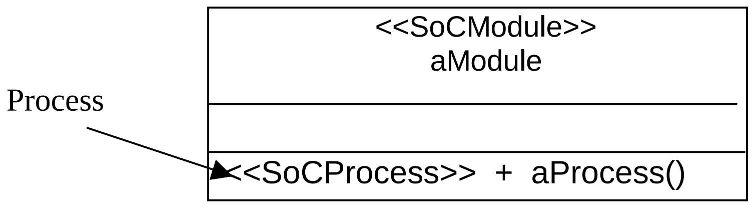


**Figure 7.11 - Process**

## 7.11 Clock Port

The clock port uses same notation as port, with a difference of having an attribute specific to clock port. Example of the clock port is shown in Figure 7.12.

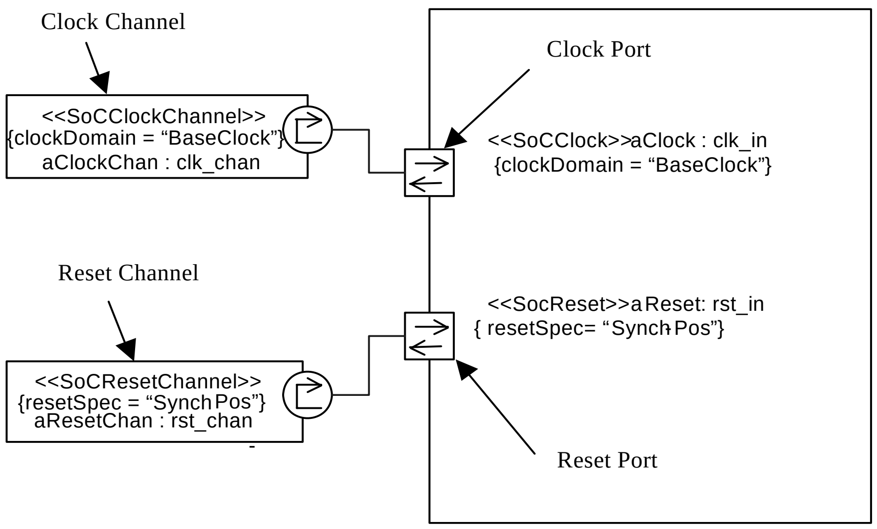

Clock Channel
<<SoCClockChannel>>
{clockDomain = "BaseClock"}
aClockChan : clk_chan
Clock Port
<<SoCClock>>aClock : clk_in
{clockDomain = "BaseClock"}
Reset Channel
<<SoCResetChannel>>
{resetSpec = "Synch Pos"}
aResetChan : rst_chan
<<SocReset>>aReset: rst_in
{ resetSpec= "Synch Pos"}
Reset Port

**Figure 7.12 - Clock and reset (channel/port)**

## 7.12 Reset Port

The reset port uses same notation as port except for having an attribute specific to reset port. Example of the reset port is shown in Figure 7.12.

## 7.13 Clock Channel

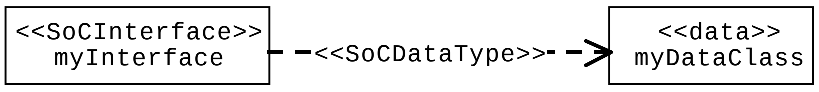
The clock channel uses same notation as channel part except for having an attribute specific to clock channel. Example of the clock channel is shown in Figure 7.12.

## 7.14 Reset Channel

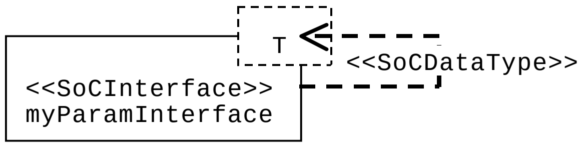
The reset channel uses same notation as channel part except for having an attribute specific to reset channel. Example of the reset channel is shown in Figure 7.12.

## 7.15 Data Type

Data Type dependency has the same notation as normal dependency as shown below.



(a) SoCDataType dependency



(b) SoCDataType dependency pointing at template parameter

**Figure 7.13 - Data Type dependency example**

# 8 SoC Profile Constraints Defined by OCL

This section describes the SoC profile constraints defined by OCL. Please note that this section defines only the constraints that must be defined by OCL. Other constraints are defined by class diagrams shown in Figure 7.1 and Figure 7.2.

```
-- Every <<SoCModule>>(<<SoCChannel>>) shall not own itself.
context SoCModule
inv
   let instances(sf : StructuralFeature) : Set(Class) =
        if (sf.type->notEmpty() and sf.type.oclIsKindOf(Class)) then
        Set { sf.type }->union(sf.type.oclAsKind(Class).allParents()->
         select(p | p.oclIsKindOf(Class))->
           union(instances(sf.type.feature->
         select(f | f.oclIsAKind(StructuralFeature))
        else
         Set {}
        endif
   in
       instances(feature->
        select(f | f.oclIsKindOf(StructuralFeature)))
         ->excludes(self)


-- <<SoCConnector>> is only allowed between <<SoCPort>> owned by parent module
-- and <<SoCPort>> owned by sub-module, or between <<SoCPort>> owned by
-- sub-module and <<SoCChannelProperty>>.

context SoCConnector
inv
    let portOwner(p : Port) : Set(Class) =
        if (p.socport->notEmpty()) then
            Set { p.featuringClassifier.oclAsKind(Class) }
        else
            Set {}
        endif

    let owningPort(ce : ConnectorEnd) : Boolean =
       ce.role->notEmpty() and ce.role.oclIsKindOf(Port) and
        portOwner(ce.role.oclAsKind(Port))->
          includes(featuringClassifier)

    let subModPort(ce : ConnectorEnd) : Boolean =
      ce.role->notEmpty() and ce.role.oclIsKindOf(Port) and
       not portOwner(ce.role.oclAsType(Port))->
                   includes(featuringClassifier))
    let channel(ce : ConnectorEnd) : Boolean =
      ce.role->notEmpty() and ce.role.oclIsKindOf(SoCChannelProperty)
    in
   -- one-end owning port -> sub-module-port
    owningPort(end->at(1)) implies subModPort(end->at(2)) and
    owningPort(end->at(2)) implies subModPort(end->at(1)) and
   -- or one-end channel->sub-module-port
    channel(end->at(1)) implies subModPort(end->at(2)) and
    channel(end->at(2)) implies subModPort(end->at(1))


-- <<SoCConnector>> shall have exactly two connector ends and role.
context SoCConnector
inv:
    end->size() = 2 and
    end->forAll(ce | ce.role->notEmpty())

-- <<SoCClock>> shall be connected to <<SoCClock>> or <<SoCClockChannel>>
-- having the same clockDomain attribute value.
context SoCConnector
inv:
    let portClkAttrs(p : Port) : Set(String) = p.clockDomain
    let channelClkAttrs(c : Property) : Set(String) = p.clockDomain
    let clkAttrs(ce : ConnectorEnd) : Set(String) =
        if (ce.role.oclIsKindOf(Port)) then
            portClkAttrs(ce.role)
        else if (ce.role.oclIsKindOf(Property)) then
            channelClkAttrs(ce.role)
        else
            Set { }
        endif
    in
      clkAttrs(end->at(1))->
         includesAll(clkAttrs(end->at(2)))
-- <<SoCReset>> shall be connected to <<SoCReset>> or <<SoCResetChannel>>
-- having the same resetSpec attribute value.
context SoCConnector
inv:
    let portResetAttrs(p : Port) : Set(String) = p.resetSpec
    let channelResetAttrs(c : Property) : Set(String) = p.resetSpec
    let resetAttrs(ce : ConnectorEnd) : Set(String) =
        if (ce.role.oclIsKindOf(Port)) then
            portResetAttrs(ce.role)
        else if (ce.role.oclIsKindOf(Property)) then
            channelResetAttrs(ce.role)
        else
            Set { }
        endif
    in
      resetAttrs(end->at(1))->
```

```
         includesAll(resetAttrs(end->at(2)))


-- All <<SoCPort>> instances shall be distinguishable (shall be ordered).
context SoCport
inv:isOrdered = true

-- All instances of <<SoCModuleProperty>> shall be distinguishable and shall be -- ordered.
context SoCModuleProperty
inv:isOrdered = true

-- All instance of <<SoCChannelProperty>> shall be distinguishable and shall be -- ordered.
context SoCChannelProperty
inv:isOrdered = true


-- All <<SoCInterface>> shall be inherited from other <<SoCInterface>> whose
-- attribute is either of INPUT or OUTPUT (shall not be inherited from both of --- them).
context SoCInterface
inv:
   let parentInterfaces(c : Classifier) : Set(Interface) =
       c.allParents()->union(Set{ c })->
       select(c | c.oclIsKindOf(SoCInterface))
   let isInput(c : Classifier) : Boolean =
       parentInterfaces(c)->exists(i|i.direction = directionKind::INPUT)
   let isOutput(c : Classifier) : Boolean =
       parentInterfaces(c)->exists(i|i.direction = directionKind::OUTPUT)
   in
       not (isInput(self) and isOutput(self))


-- All <<SoCChannel>> shall be inherited from both INPUT <<SoCInterface>>
-- and OUTPUT <<SoCInterface>>
context SoCChannel
inv:
  let parentInterfaces(c : Classifier) : Set(Interface) =
      c.allParents()->union(Set{c})->select(c | c.oclIsKindOf(SoCInterface))
  let isInput(c : Classifier) : Boolean =
      parentInterfaces(c)->one(i | i.direction = directionKind::INPUT)
  let isOutput(c : Classifier) : Boolean =
      parentInterfaces(c)->one(i | i.direction = directionKind::OUTPUT)
   in
      isInput(self) and isOutput(self)


-- <<SoCProtocol>> shall have at least one INPUT <<SoCInterface>> and one
-- OUTPUT <<SoCInterface>> and one <<SoCChannel>>.
-- <<SoCProtocol>> shall specify all of the inheritance relation between them.
context SoCProtocol
inv:
   let Channels : Set(Class) =
     part.type->collect(t | t.oclIsKindOf(SoCChannel))
   let Inputs : Set(Interface) =
     part.type->collect(t | t.oclIsKindOf(SoCInterface) and
       t.oclAsType(SoCInterface)->
                  exists(direction = directionKind::INPUT)
   let Outputs : Set(Interface) =
     part.type->collect(t | t.oclIsKindOf(SoCInterface) and
       t.oclAsType(SoCInterface)->
                   exists(direction = directionKind::OUTPUT)
   let parentClasses(c : Class) : Set(Class) =
     c.superClass->union(c.superClass->
        collect(cc | parentClasses(cc)))->asSet()
   in
     Channels->notEmpty() and Inputs->notEmpty() and Outputs->notEmpty() and
             Channels->collect(c|parentClasses(c))->
                       includesAll(Inputs->union(Outputs))

-- All <<SoCPort>> shall have exactly one <<SoCInterface>>.
context SoCPort
inv: required->size() = 1

-- All <<SoCPort>> shall not have a provided interface.
context SoCPort
inv:
    provided->isEmpty()


-- For all <<SoCInterface>> that a <<SoCChannel>> implements, classes pointed by
-- <<SoCDataType>> dependency from them shall be conforms to each other.
context SoCChannel
inv:
    let parentInterfaces(c : Classifier) : Set(Interface) =
         c.allParents->select(c | c.oclIsKindOf(Interface))
in
    parentInterfaces(self)->select(c | c.oclIsKindOf(SoCInterface).
        clientDependency->select(d | d.oclIsKindOf(SoCDataType).supplier
                                 ->asSet()->size() = 1

--  Following descriptions are the OCL examples for
--  connection specification between ports, channels,
--  sub-module ports with multiplicity.
--  OCL example for connection with multi-ports
context aModule
inv:
   -- portA[0] <-> submod[2].PortA[1]
   -- both portA and submod.portA has required interface of sc_fifo_out_if<int>
submod[2].PortA[1].sc_fifo_out_if<int>->includes(portA[0])
  and
   -- channelC[1] <-> submod[3].PortA[4]
```

```
    --    channelC    has    type    channelClass,    which    implements    sc_fifo_out_if<int>
 submod[3].portA[4].channelClass->includes(channelC[1])
   and
    -- channelC[I] <-> submod[I].PortB (where I = 0..N(parameter))
    -- submod.portB has required interface of sc_fifo_out_if<int>
   Sequence{ 0..N }->
         forAll(i | submod[i].portB.channelClass->includes(channelC[i]))
```

# Annex A
(informative)

## The SystemC Description

The SystemC description corresponding to the UML model using the profile for SoC is shown here. The SystemC description consists of "the module description file," which describes the declaration of module, instantiation of the sub module, the channel, and the port, and "the data class description file," which defines the data class. A detailed configuration of the files is not specified in this document

## A.1 Module Description File

A header file will be created for each module. The header file includes module definition, port declaration, channel declaration, member function (which becomes a process) declaration and module constructor definition. It does also have "#include" as part of the necessary header (for sub modules and channels).

In addition, initialization parameter is the list of parameters. The number of parameters depends on module.

### Module definitions

Module definitions are as shown below:

```
[ template < templetParameterList > ]
SC_MODULE (className) {
[ (port declaration) ]
[ (sub module declaration) ]
[ (channel declaration) ]
[ (process function declaration) ]
[ (declaration of user defined member variable) ]
[ (declaration and definition of user defined member function,_and so on) ]
SC_HAS_PROCESS (className);
className (sc_module_name name [, initializationParameters ]): Sc_module (name) [,
(port initialization) ] [, (initialization of user defined member variables) ] {
   [ (Definition of sub module generation) ]
   [ (definition of channel generation) ]
   [ (definition of connection) ]
   [ (process generation definition) ]
   [ (initialization of user defined member variable, and so on) ]
}
};

[ template < templetParameterList > ]
void className
   [ < templetParameterList > ]: : memberFunctionName () {
   // contents of process function
}
[ Definition of user defined member function, and so on]
```

### Port declaration

The port declaration is described as below:

```
sc_port< protocolInterfaceClassName
      [ < data_type_dataClassName > ] >
      instanceName [ ' [ ' portMultiplicity ' ] ' ];
```

Each port should have this description. When the port multiplicity is 1 (in case of the single port), the port multiplicity and the bracket [ ] shall be omitted. When the protocol interface is template interface, the dataClassName must be specified as a template parameter. If not parameterized, the data_type_dataClassName and <> shall be omitted.

### Clock port

The clock port is described as below.

```
sc_in_clk instanceName [ ' [ ' portMultiplicity ' ] ' ];
```

Declaration of the reset port is similar to normal port declaration.

### Sub-module declaration

The sub module declaration is described as below:

```
subModuleClassName * subModuleInstanceName
   [ ' [ 'multiplicity' ] ' ];
```

Each sub module should have this description. When the sub module  multiplicity is 1, the multiplicity and the bracket [ ] shall be omitted.

### Channel declaration

The channel declaration is described as below.

```
channelName
   [ < data_type_dataClassName > ] *
   channelInstanceName [ ' [ ' multiplicity ' ] ' ];
```

Each channel should have this description. When the channel multiplicity is 1, the multiplicity and the bracket [] shall be omitted.

### Clock channel and reset channel declarations

The clock channel and the reset channel are declared as below:

```
clockChannelName * clockChannelInstanceName;
```

### Process function declaration

The process function declaration is described as below.

```
void memberFunctionName ();
```

Also the sub modules, the channels and the processes are declared in the module constructor. The arguments for the constructor will be given from the initialization parameter for the constructor.

## Port initialization

The port initialization is described as below.

```
portInstanceName ("portInstanceName")
   [, portInstanceName ("portInstanceName") ]
```

## Sub-module instantiation

The sub module is instantiated as below:

```
submoduleInstanceName [ ' [ ' index ' ] ' ] =
   new submoduleClassName("instanceName",
   initialization parameters);
```

Each sub module should have this description. When the sub module multiplicity is 1, the index and the bracket [] shall be omitted.

## Channel instantiation

The channel is instantiated as below:

```
channelInstanceName [ ' [ ' index ' ] ' ] =
   new channelName [ < data_type_dataClassName > ] (initialization parameters);
```

Each channel should have this description. When the channel multiplicity is 1, the index and the bracket [ ] shall be omitted.

## Clock channel instantiation

The clock channel is instantiated as below. Clock parameter will be given from the "clock domain" specification shown as the clockDomain tagged value.

```
clockChannelInstanceName =
   new clockChannelName("clockChannelInstanceName",
   clock parameter);
```

## Connection between channels or ports

The connection between the channels or the ports is defined, with according to the connection specification of the module, as below:

```
submoduleInstanceName [ ' [ ' subModuleIndex ' ] ' ] -> portInstanceName [ ' [ '
subModulePortIndex ' ] ' ] (portInstanceName [ ' [ ' portIndex ' ] ' ], or *
channelInstanceName [ ' [ ' channelIndex ' ] ' ]);
```

Each connection should have this description. When the connection is to the sub module with multiplicity 1, the sub module index and the bracket [ ] shall be omitted. When the connection is to the sub-module port with multiplicity 1, the port index and the bracket [] shall be omitted. When the connection is to the module port with multiplicity 1, the port index and the bracket [ ] shall be omitted. When the connection is to the channel with multiplicity 1, the channel index and the bracket [ ] shall be omitted.

## Process instantiation

The process instantiation is described as below:

```
SC_THREAD (memberFunctionName);
```

## A.2 Data Class Description File

The header file is created in every class with the <<data>> stereotype. The description is a little different depending on the transport Method tagged value, as shown below.

```
typedef dataClassName* data_type_dataClassName; (When transportMethod is Pointer)
typedef dataClassName data_type_dataClassName; (When transportMethod is Copy)
```

# Annex B
(informative)

# The SystemC Description Example

This annex shows several examples of the actual UML diagrams for SoC designs and the generated SystemC codes from UML. The first example is the description of a module for sending and receiving data using a FIFO, the second example is a structured design model with "clock" and "reset."

## B.1 Description Example 1

Figure B-1 shows the UML class diagram. Here func() member function of user_class1 and func() member function of user_class2 are processes. And data1, data2, data3 are input/output data between the processes.

First, for this class diagram, <<Data>> <<Controller>> stereotypes are specified.

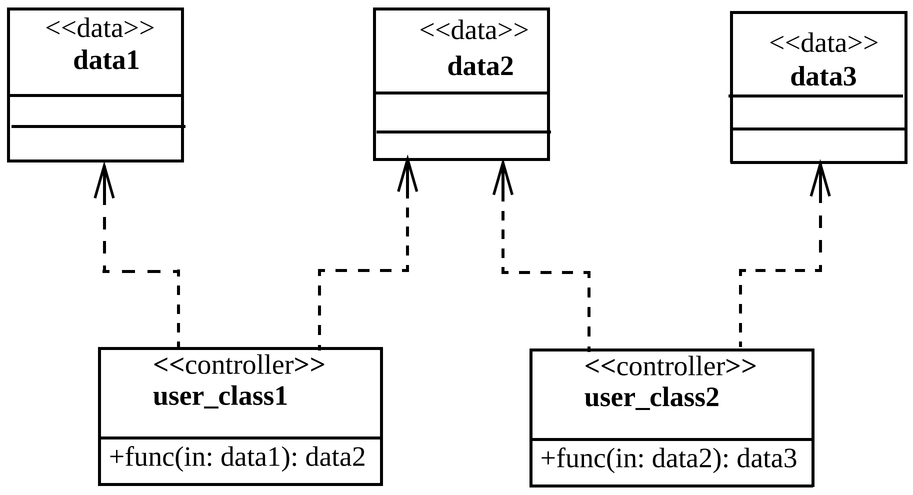


**Figure B.1 - Class Diagram after the Stereotype Specified**

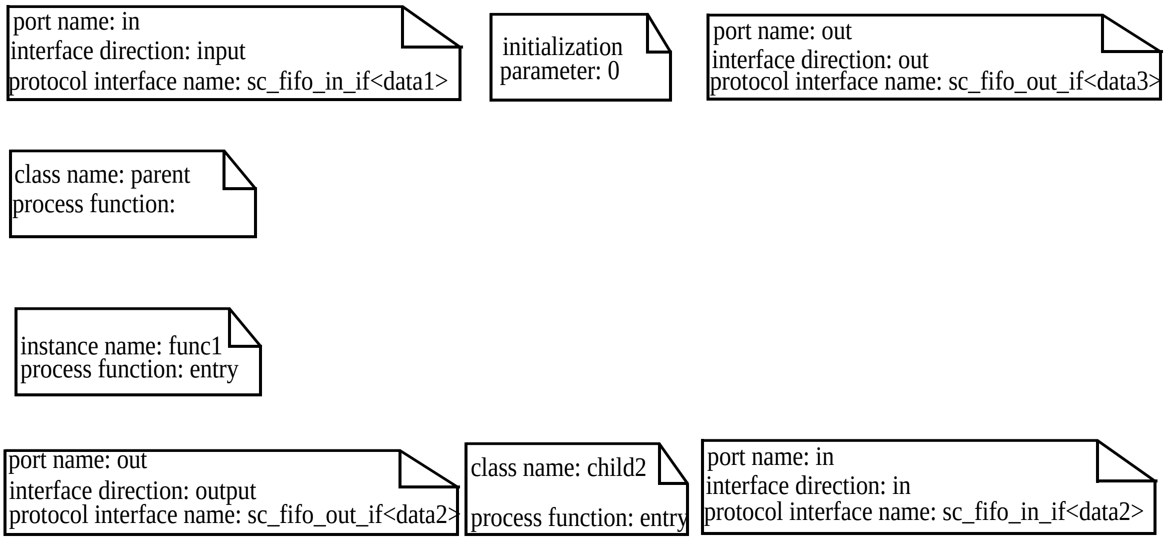

port name: in
interface direction: input
protocol interface name: sc_fifo_in_if<data1>
initialization parameter: 0
port name: out
interface direction: out
protocol interface name: sc_fifo_out_if<data3>
class name: parent
process function:
instance name: func1
process function: entry
port name: out
interface direction: output
protocol interface name: sc_fifo_out_if<data2>
class name: child2
process function: entry
port name: in
interface direction: in
protocol interface name: sc_fifo_in_if<data2>

**Figure B.2 - Structure Diagram and Attribute of Each Object, Tagged Value**

Figure B-2 is the structure diagram. All the necessary values and attributes are specified as notes.

The channel is described in the UML diagram as shown below. Here the FIFO channel of SystemC (sc_fifo) is used as a channel with buffer size = 16 (default value of sc_fifo class). The buffer size is given as an initialization parameter at the instantiation of the channel. (In the figure below, attributes, and/or template arguments of buffer size are not shown.)

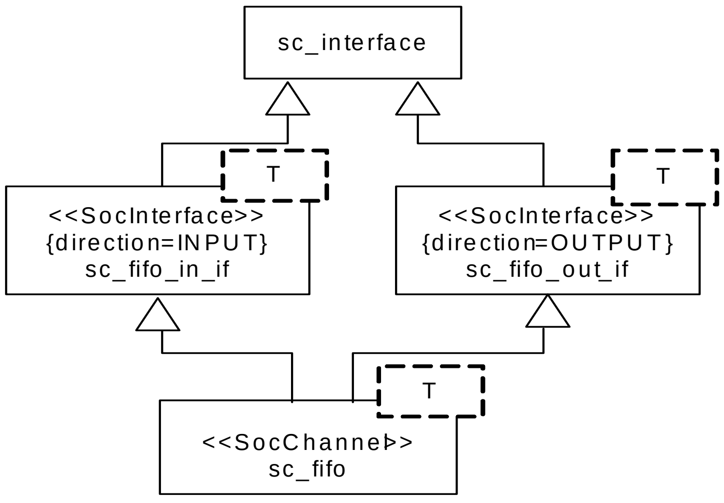


**Figure B.3 - FIFO Protocol Interface and Channel**

The generated SystemC source code from the UML diagram above is shown below. Please note that the user specified portion and the portion automatically generated using the tool are not distinguished here using comments. And also, the include statements which usually exists in the header file are ommited.

```
(parent.h)
#include "child1.h"
#include "child2.h"
#include "data_type_data1.h"
#include "data_type_data2.h"
#include "data_type_data3.h"
SC_MODULE(parent) {
   sc_port< sc_fifo_in_if< data_type_data1 > > in;
   sc_port< sc_fifo_out_if< data_type_data3 > > out;

   child1* func1;
   child2* func2;

   sc_fifo< data_type_data2 >  *fifo;

   SC_HAS_PROCESS(parent);
   parent(sc_module_name name)
   : sc_module(name), in("in"), out("out") {
      func1 = new child1("func1");
      func2 = new child2("func2");

      fifo = new sc_fifo< data_type_data2 >(0);

      func1->in(in);
      func1->out(*fifo);
      func2->in(*fifo);
      func2->out(out);
   }
};

(child1.h)
#include "data_type_data1.h"
#include "data_type_data2.h"
SC_MODULE(child1) {
   sc_port< sc_fifo_in_if< data_type_data1 > > in;
   sc_port< sc_fifo_out_if< data_type_data2 > > out;

   void entry();

   SC_HAS_PROCESS(child1);
   child1(sc_module_name name)
   : sc_module(name), in("in"), out("out") {
      SC_THREAD(entry);
   }
};
```

```
void child1::entry() {

}

(child2.h)
#include "data_type_data2.h"
#include "data_type_data3.h"
SC_MODULE(child2) {
   sc_port< sc_fifo_in_if< data_type_data2 > > in;
   sc_port< sc_fifo_out_if< data_type_data3 > > out;

   void entry();

   SC_HAS_PROCESS(child2);
   child2(sc_module_name name)
   : sc_module(name), in("in"), out("out") {
      SC_THREAD(entry);
   }
};

(data_type_data1.h)
typedef data1* data_type_data1;

(data_type_data2.h)
typedef data2* data_type_data2;

(data_type_data3.h)
typedef data3* data_type_data3;
```

## B.2 Description Example 2

Here is the SystemC description example of the clock and reset. In this example, the UML structure diagram and the corresponding SystemC description for the module (and its process), which has a synchronous reset and a clock, are shown. Figure B-4 depicts the structure diagram, and the clockDomain = "Clock" specifies the clock with the positive edge trigger, and the reset parameter "Synch" specifies the synchronous reset. Please note that the channels of data input/output are not shown in this figure.

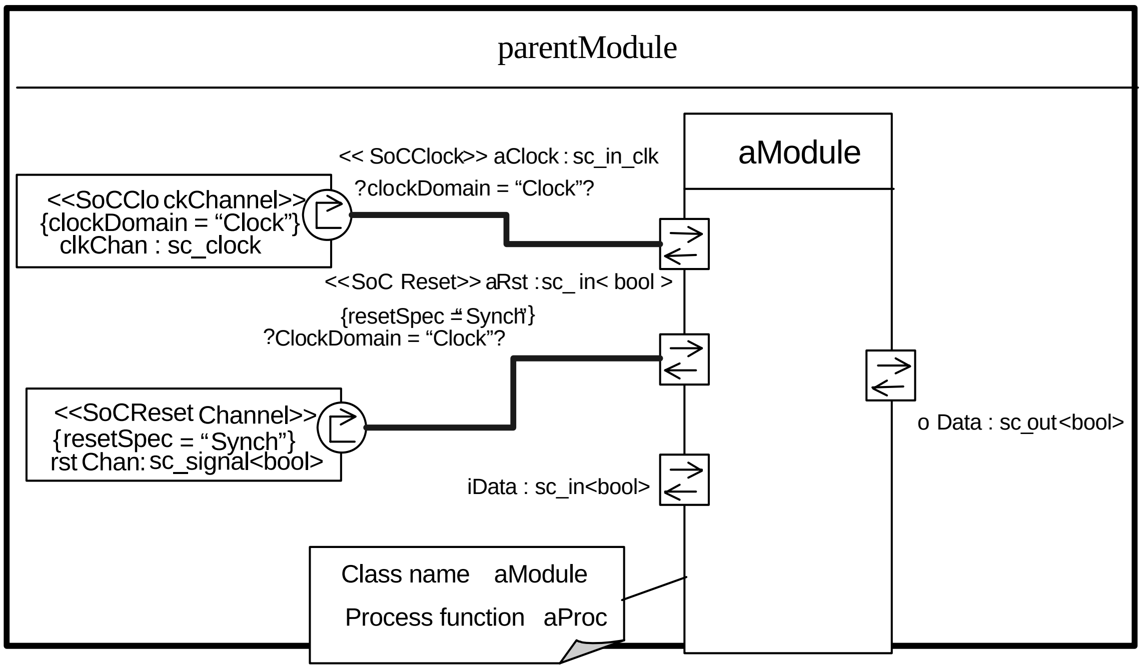


**Figure B.4 - Clock, Reset example**

The SystemC description for the structure diagram follows is shown in Figure B-5. Again, please note that the data channel is not shown here.

```
SC_MODULE(aModule) {
  sc_in_clk aClock;
  sc_in<bool> aRst;
  sc_in<bool> iData;
  SC_HAS_PROCESS(aModule);
  inline void aProc();
  aModule(sc_module_name name) :
   sc_module(name),
   aClock(“aClock”), aRst(“aRst”), iData(“iData”) {
     SC_CTHREAD(aProc, aClock.pos());
     watching(aRst.delayed() == true);
   }
};
void aModule::aProc() {
  while (true) {
    wait();
    bool const b = iData.read();
    oData.write(b);
  }
}
SC_MODULE(parentModule) {
  sc_signal *clkChan;
  sc_signal<bool> *rstChan;
  aModule *aMod;
  parentModule(sc_module_name name) : sc_module(name) {
    clkChan = new sc_clock(“clkChan”, 10, SC_NS);
    rstChan = new sc_signal<bool>(“rstChan”);
    aMod = new aModule(“aModule”);
    aMod->aClock(*clkChan);
    aMod->aRst(*rstChan);
    …// data channel instantiation and bindings here
  }
};
```

**Figure B.5 - Clock, Reset in SystemC description example**

# Annex C
(informative)

## Modeling Guidelines

This section describes the basic method of transmitting data instances via channels. As described in the data stereotype discussion, there are two modes in transmission of the data instances. In the first part, SystemC example of Copy mode is shown. Next example of Pointer mode is shown. These examples show how modules aCopyMod or aPointerMod receive the instance of Data type, then how these modules transfer the received instances to another module. The timing of creation and deletion of value is critical to pointer mode. Therefore we recommend that Copy mode is adopted. However, when pointers or references are mandatory because of implementation algorithms Pointer mode must be selected. When the pointer mode is used, application of following rules is recommended to avoid problems during model

```
typedef Data data_type_Data;
SC_MODULE(aCopyMod) {
  sc_port< sc_fifo_in_if< data_type_Data > > in;
  sc_port< sc_fifo_out_if< data_type_Data > > out;
  void entry();
 aCopyMod(sc_module_name name) :
  sc_module(name), in(“in”), out(“out”)
 {
    SC_THREAD(entry);
  }
};
void aCopyMod::entry() {
  while (true) {
    Data t = in.read();
    Data nt = // The value nt which based on the value t is created here.
    out.write(nt);
  }
}
```

**Figure C.1 - Example of the data transmission by copy mode**

creation: 1)sender of the data instance allocates the memory area for the data instance, initialize the area with values, and send the pointer pointing at the area, 2) receiver deallocates the area pointed by the pointer after its use of received data.

```
typedef Data* data_type_Data;
SC_MODULE(aPointerMod) {
  sc_port< sc_fifo_in_if< data_type_Data > > in;
  sc_port< sc_fifo_out_if< data_type_Data > > out;
  void entry();
  aPointerMod(sc_module_name name) : sc_module(name),
 in(“in”), out(“out”) {
    SC_THREAD(entry);
  }
};
void aPointerMod::entry() {
  while (true) {
    data_type_Data t = in.read();
    data_type_Data nt = new Data(); // Here, the Data type value nt
// is instantiated using the value t.
    delete t; // The read value t is deleted here.
    out.write(nt);
  }
}
```

**Figure C.2 - Example of the data transmission by Pointer mode**

## C.1 Notation of Module Hierarchy and Conversion to the SystemC Description

In SoC design it is essential that the configuration precisely conforms to a specification. This section explains how this configuration must be captured by the design entry tools.

As shown in the following diagram, implementing some functionality with sub modules, and channels connecting them is widely employed.

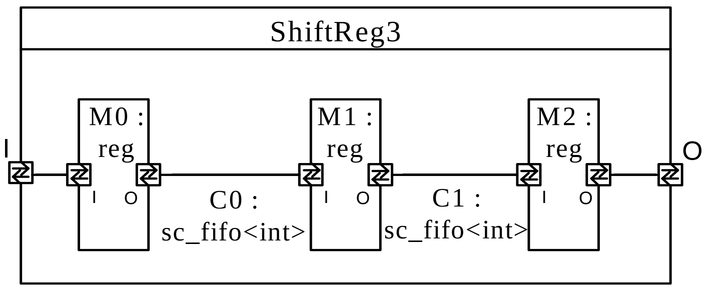
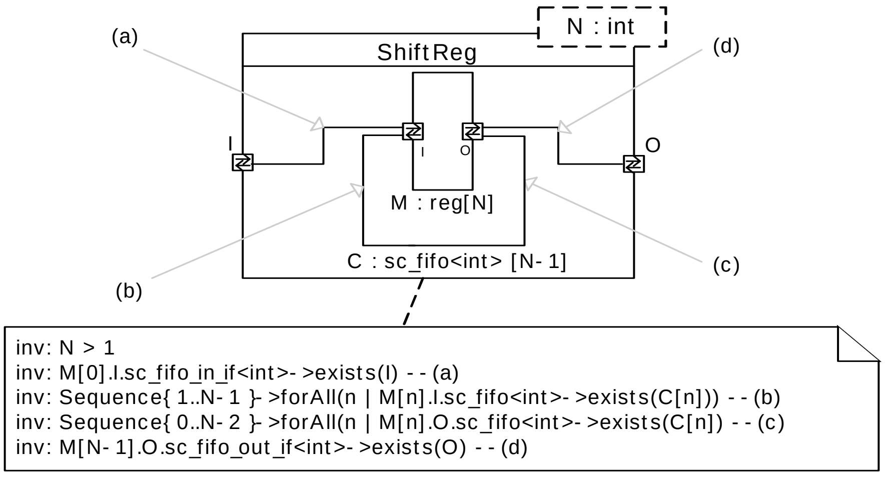


**Figure C.3 - Example of the module which has systematic structure.**

In SoC design, it is typical to construct a module structure using an algorithm parameterized by template parameters or constructor arguments. This section explains how this kind of structure is described using the structure diagram. Figure 24 shows the same module using a template parameter for specifying the number of owning modules. Connection specification between each instances are described using OCL as invariants of ShiftReg. By allowing this kind of description, designers are able to define the highly re-usable modules.



```
inv: N > 1
inv: M[0].I.sc_fifo_in_if<int>- >exists(I) - - (a)
inv: Sequence{ 1..N- 1 }- >forAll(n | M[n].I.sc_fifo<int>- >exists(C[n])) - - (b)
inv: Sequence{ 0..N- 2 }- >forAll(n | M[n].O.sc_fifo<int>- >exists(C[n]) - - (c)
inv: M[N- 1].O.sc_fifo_out_if<int>- >exists(O) - - (d)
```

**Figure C.4 - Example of template module**

The SystemC description corresponding to the diagram is shown below, (note that include statements or any declarations, that is required but not relevant to the explanation of the notation, are abbreviated).

```
template <int N>
struct ShiftReg : public sc_module {
    sc_port< sc_fifo_in_if<int> > I;
    sc_port< sc_fifo_out_if<int> > O;

    reg *M[N];
    sc_fifo<int> *C[N-1];

    ShiftReg(sc_module_name name) : sc_module(name) {
        assert(N>0);
        for (int i = 0; i < N; i++) {
            stringstream regname;
            regname << "reg[" << i << "]";
            M[i] = new reg(regname.str().c_str());
        }
        for (int i = 0; i < N-1; i++) {
            C[i] = new sc_fifo<int>;
        }
        for (int i = 0; i < N; i++) {
            if (i == 0) {
                M[i]->I(I);
            }
            if (i > 0) {
                M[i]->I(*C[i-1]);
            }
        for (int i = 0; i < N; i++) {
            if (i == N-1) {
                M[i]->O(O);
            }
```

```
                if (i < N-1) {
                    M[i]->O(*C[i]);
                }
            }
        }
    };
```

# Annex D
(informative)

## Glossary

For the purpose of this specification, the terms and definitions given in the normative references and this glossary apply.

| | |
|---|---|
| **Channel** | Realizes communication between modules. |
| **Connector** | Connector connects ports or connects ports and a channel. |
| **Member function** | Meaning identical to that of "Operation" of a UML class. |
| **Module** | A fundamental element for implementation of SoC functionalities. It may own channels or lower hierarchy modules, that constructing the module, as its member. These relationships can be represented hierarchically in the same way as composite states. |
| **Port** | Defines the protocol interface used for communication external to modules. |