SDL RT

specification & description language - real time

Object	Graphical language to specify and design real time and embedded software
Date	September 20th, 2021
Version	2.4
Reference	http://www.sdl-rt.org

Introduction - - - - - 6

Architecture - - - - - 8
System 8
Agents 8

Communication - - - - - 10

Behavior - - - - - 13
Start 13
State 13
Stop 14
Message input 15
Message output 16
To a queue Id 17
To a process name 17
To the environment 18
Via a channel or a gate 19
As a broadcast 21
Message save 23
Continuous signal 24
Action 25
Decision 25
Semaphore take 26
Semaphore give 27
Timer start 27
Timer stop 27
Task creation 27
Procedure call 28
Connectors 28
Transition option 29
Comment 30
Extension 31
Procedure start 31
Procedure return 32
Text symbol 32
Additional heading symbol 32
Object creation symbol 33
Super class transition symbol 34
Super class next state symbol 34
Composite state 35
Composite state definition 35
Composite state usage 36
Symbols ordering 38

Declarations - - - - - 39
Process 39
Procedure declaration 40

SDL-RT defined procedure 40
C defined procedure 41
Messages 41
Timers........ 42
Semaphores 42

MSC - - - - - - - - 43
Agent instance 43
Semaphore representation 44
Semaphore manipulations 44
Message exchange........ 46
Synchronous calls........ 48
State 49
Timers........ 51
Time interval 53
Coregion 55
MSC reference........ 56
Inline expressions 58
Text symbol........ 59
Comment 59
Action 60
Property Sequence Charts (PSC)........ 60
Component instance 61
Normal, required and fail messages 61
Parallel, alternative and loop operator 61
Strict operator 61
Relative time constraint 61
Unwanted/wanted message or chain constraints 61
High-level MSC (HMSC) 64

Data types - - - - - - - - 66
Type definitions and headers........ 66
Variables........ 66
C functions 66
External functions 66

Object orientation - - - - - - - - 67
Block class........ 67
Process class 68
Adding a transition 71
Overload a transition 71
Abstract transition 72
Reference to the super class 74
Example 77
Class diagram 78
Class 78
Specialisation 81
Association 81

Aggregation 82
Composition 83
Package 83
Usage in an agent 84
Usage in a class diagram 84

Deployment diagram - - - - 85
Node 85
Component 85
Connection 86
Dependency 87
Aggregation 88
Node and components identifiers 88

Symbols contained in diagrams - - - - 89

Textual representation - - - - 90
Common XML definitions 90
XML elements for standard diagrams 91
Principles 91
XML elements 92
Explanations 94
Behavioral diagrams 94
Principles 94
XML elements 95
Explanations 97
MSC diagram DTD 99
Principles 99
DTD text 99
Explanations 100

Example systems - - - - 104
Ping Pong 104
A global variable manipulation 107
Access Control System 110
Requirements 110
Analysis 111
Architecture 112
pCentral process 112
getCardNCode procedure 114
pLocal process 115
Display procedure 119
DisplayStar procedure 119
Deployment 120

Differences with classical SDL - - - - 121
Data types 121
Semaphores 121
Inputs 121

Names 121
Object orientation 121

Memory management - - - - 123
Global variables 123
Message parameters 123

Keywords - - - - 124

Syntax - - - - 125

Naming convention - - - - 126

Lexical rules - - - - 127

Glossary - - - - 128

Modifications from previous releases - - - - 129
V1.0 to V1.1 129
V1.1 to V1.2 129
V1.2 to V2.0 129
V2.0 to V2.1 129
V2.1 to V2.2 129
V2.2 to V2.3 130
V2.3 to V2.4 130

Index - - - - 131

1 - Introduction

As its name states, SDL-RT is based on SDL standard from ITU extended with real time concepts. V2.0 has introduced support of UML from OMG in order to extend SDL-RT usage to static part of the embedded software and distributed systems.

SDL has been developed in the first place to specify telecommunication protocols but experience showed some of its basic principles could be used in a wide variety of real time and embedded systems. Its main benefits are:

- architecture definition,
- graphical finite state machine,
- object orientation.

But SDL was not meant to design real time systems and some major drawbacks prevented it to be widely used in the industry:

- obsolete data types,
- old fashioned syntax,
- no pointer concept,
- no semaphore concept.

SDL being a graphical language it is obviously not suited for any type of coding. Some parts of the application still need to be written in C or assembly language. Furthermore legacy code or off the shelf libraries such as RTOS, protocol stacks, drivers have C APIs. Last but not least there is no SDL compilers so SDL need to be translated into C code to get down to target. So all SDL benefits are lost when it comes to real coding and integration with real hardware and software.

Considering the above considerations a real time extension to SDL needed to be defined that would keep the benefits of SDL and solve its weaknesses. The simpler the better ! SDL-RT was born based on 2 basic principles:

- Replace SDL data types by C,
- Add semaphore support in the behavior diagrams.

UML diagrams have been added to SDL-RT V2.0 to extend SDL-RT application field:

- When it comes to object orientation, UML class diagram brings a perfect graphical representation of the classes organisation and relations. Dynamic classes represent SDL agents and static classes represent C++ classes.
- To handle distributed systems, UML deployment diagram offers a graphical representation of the physical architecture and how the different nodes communicate with each other.

The result, SDL-RT, is a:

- simpler,
- object oriented,
- graphical language,
- combining dynamic and static representations,
- supporting classical real time concepts,
- extended to distributed systems,
- based on standard languages.

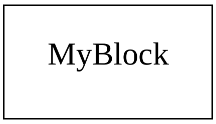
2 - Architecture

2.1 - System

The overall design is called the **system** and everything that is outside the **system** is called the **environment**. There is no specific graphical representation for the **system** but the **block** representation can be used if needed.

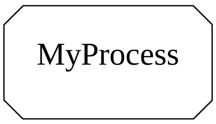
2.2 - Agents

An **agent** is an element in the system structure. There are two kinds of agents: **blocks** and **processes**. A system is the outermost block.
A **block** is a structuring element that does not imply any physical implementation on the target. A block can be further decomposed in blocks and so on allowing to handle large systems. A block symbol is a solid rectangle with its name in it:



A simple block example.

When the SDL-RT system is decomposed down to the simplest block, the way the block fulfils its functionality is described with processes. A lowest level block can be composed of one or several processes. To avoid having blocks with only one process it is allowed to mix together blocks and processes at the same level e.g. in the same block.
A process symbol is a rectangle with cut corners with its name in it:

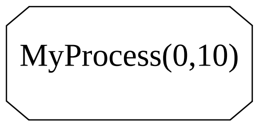


A simple process example.

A **process** is basically the code that will be executed. It is a finite state machine based task (Cf. “Behavior” on page 13) and has an implicit message queue to receive messages. It is possible to have several instances of the same process running independently. The number of instances present when the system starts and the maximum number of instances are declared between parenthesis after the name of the process. The full syntax in the process symbol is:

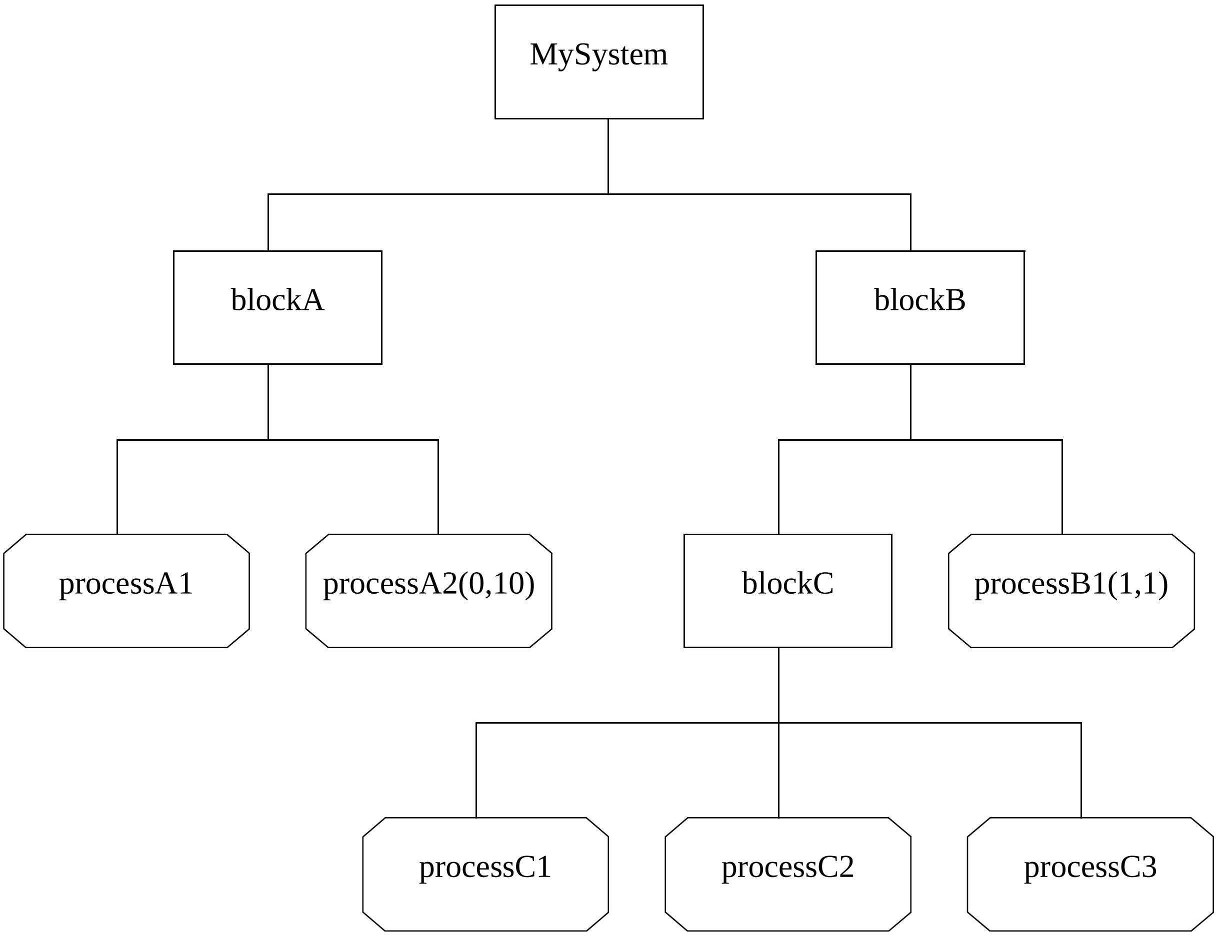
```
<process name>[(<number of instances at startup>, <maximum number of instances>)]
```

If omitted default values are 1 for the number of instances at startup and infinite for the maximum number of instances.



An example process that has no instance at startup and a maximum of 10 instances.

The overall architecture can be seen as a tree where the leaves are the processes.



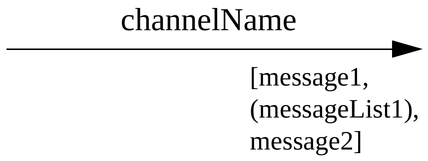
A view of the architecture tree

When viewing a block, depending on the size of the system, it is more comfortable to only represent the current block level without the lower agents.

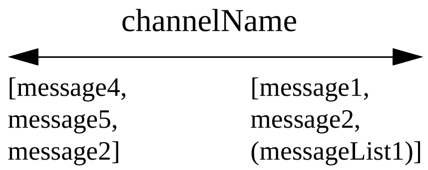
3 - Communication

SDL-RT is event driven, meaning communication is based on message exchanges. A **message** has a name and a parameter that is basically a pointer to some data. Messages go through **channels** that connect agents and end up in the processes implicit queues.
Channels have names and are represented by a one-way or two-ways arrows. A channel name is written next to the arrow without any specific delimiter. The list of messages going in a specific way are listed next to the arrow between brackets and separated by commas. Messages can be gathered in message lists, to indicate a message list in the list of messages going through a channel the message list is surrounded by parenthesis. Note the same message can be listed in both directions.

aOneWayChannel example:

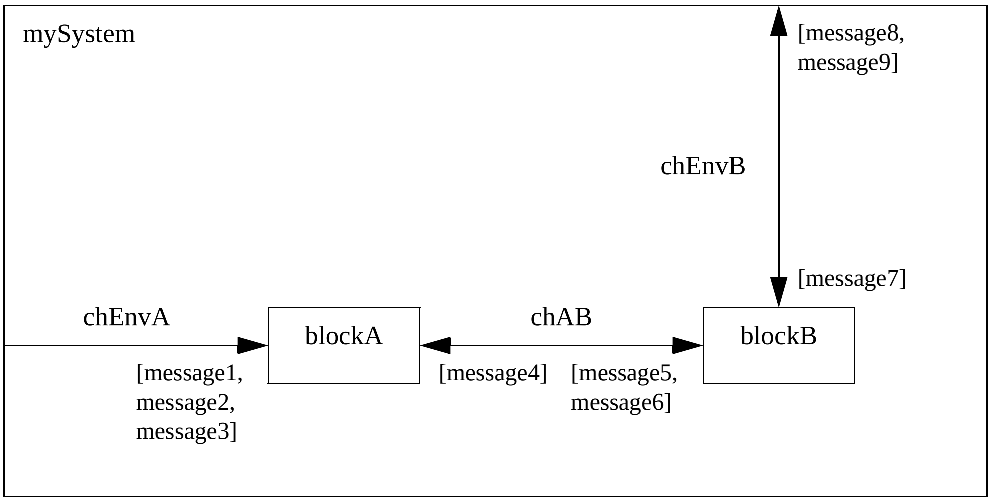


aTwoWayChannel example:



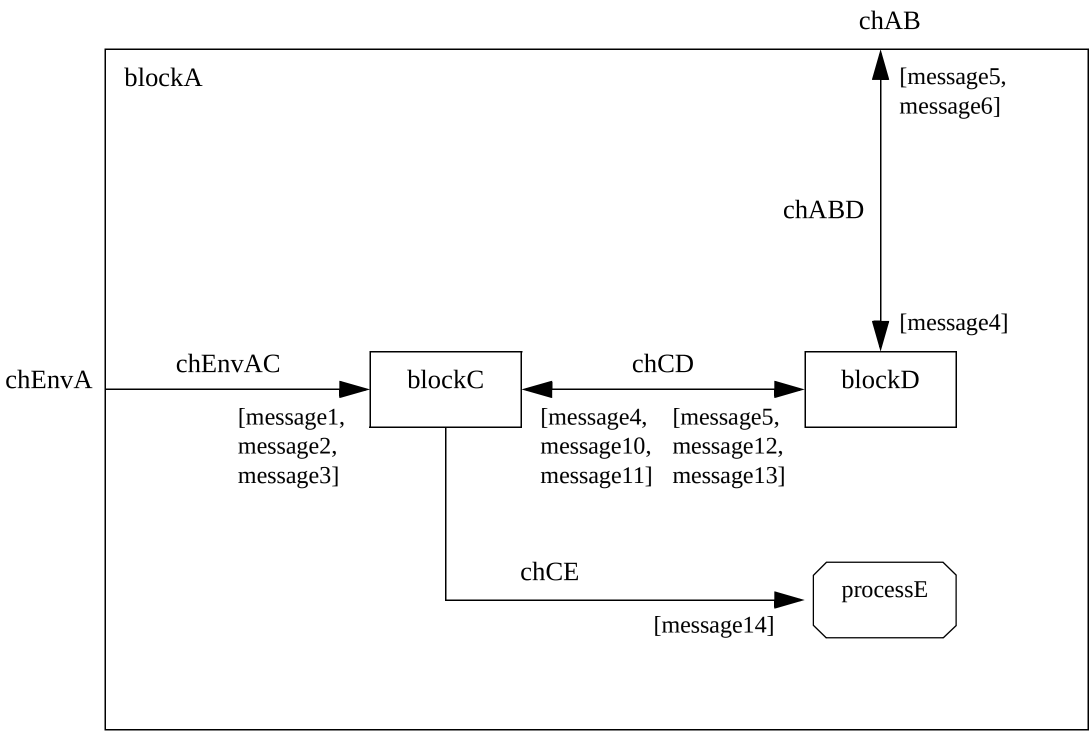
Channels end points can be connected to: the environment, another channel or a process. Graphically a channel can be connected to a block but it is actually connected to another channel inside the block. To represent the outside channels connected to the block at the upper architecture level, a block view is surrounded by a frame representing the edge of the block. The upper level channels connected to the block are then represented outside the frame and channels inside the block can be connected to these upper level channels. Note a channel can be connected to several channels. In any case consistency is kept between levels e.g. all messages in a channel are listed in the upper or lower level channels connected to it.

Example:
Let us consider an SDL-RT system made of two blocks: *blockA* and *blockB*.



An example system view

The channels `chEnvA` and `chEnvB` are connected to the surrounding frame of the system `mySystem`. They define communication with the environment, e.g. the interface of the system. `chEnvA` and `chAB` are connected to `blockA` and define the messages coming in or going out of the block.



An inner block view

The inner view of block *blockA* shows it is made of the blocks *blockC* and *blockD* and of the process *processE*. *chEnvAC* is connected to the upper level channel *chEnvA* and *chABD* is connected to the upper channel *chAB*. The flow of messages is consistent between levels since for example the messages coming in block *blockA* through *chEnvA* (*message1*, *message2*, *message3*) are also listed in *chEnvAC*.

4 - Behavior

First of all a process has an implicit message queue to receive the messages listed in the channels. A process description is based on an extended finite state machine. A process state determines which behavior the process will have when receiving a specific stimulation. A transition is the code between two states. The process can be hanging on its message queue or a semaphore or running e.g. executing code.
SDL-RT processes run concurrently; depending on the underlying RTOS and sometimes on the target hardware the behavior might be slightly different. But messages and semaphores are there to handle process synchronization so the final behavior should be independent of the RTOS and of the hardware. Since SDL-RT is open to any C code it is up to the designer to make sure this statement stays true !
Note that in a state diagram the previous statement is always connected to the symbol upper frame and the next statement is connected to the lower frame or on the side.

4.1 - Start

The start symbol represent the starting point for the execution of the process:

Start symbol

The transition between the Start symbol and the first state of the process is called the start transition. This transition is the first thing the process will do when started. During this initialization phase the process can not receive messages. All other symbols are allowed.

4.2 - State

The name of the process state is written in the state symbol:

State symbol

The state symbol means the process is waiting for some input to go on, the allowed symbols to follow a state symbol are:

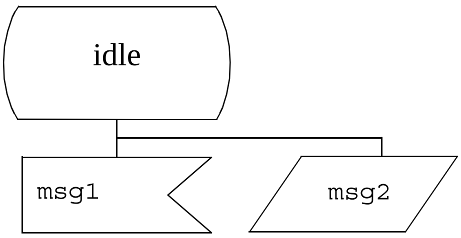
- message input
  the message could be coming from an external channel, or it could be a timer message started by the process itself.
- continuous signal

when reaching a state with continuous signals, the expressions in the continuous signals are evaluated following the defined priorities. All continuous signal expressions are evaluated before the message input !

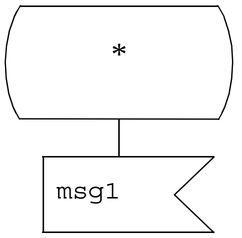
- save
  the incoming message can not be treated in the current process state. It is saved until the process state changes. When the process state has changed the saved messages are treated first (before any other messages in the queue but after continuous signals).

Some transitions can be valid for several states, the different state names are then listed separated by a comma. A star ('*') means all states.

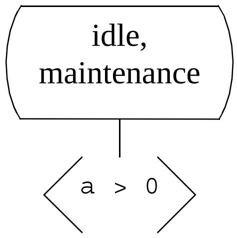
Examples:



In state `idle` `msg1` can be received and `msg2` is saved.



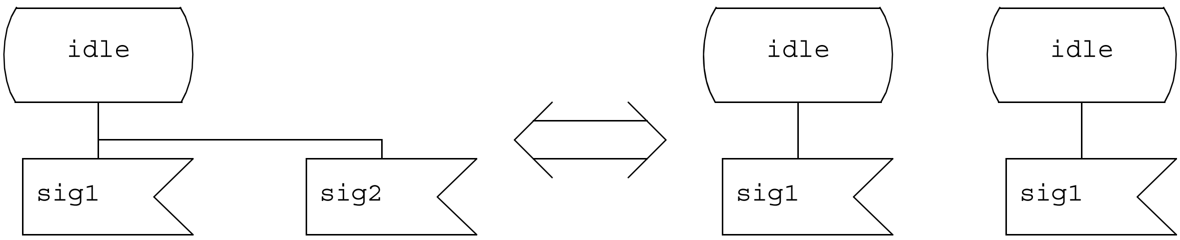
Message `msg1` can be received in any state



In states `idle` and `maintenance` the expression `a>0` is first evaluated.

A process in a specific state can receive several types of messages or treat several continuous signals. To represent such a situation it is possible to have several message inputs connected to the state or to split the state in several symbols with the same name.

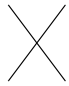
Examples:



Two ways of writing in state `idle`, `sig1` or `sig2` can be received.

4.3 - Stop

A process can terminate itself with the stop symbol.

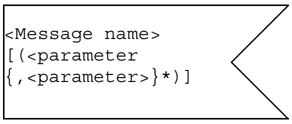


Stop symbol

Note a process can not kill another process, it can only kill itself.
There is no syntax for that symbol.

4.4 - Message input

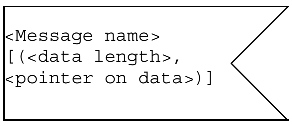
The message input symbol represent the type of message that is expected in an SDL-RT state. It always follows an SDL-RT state symbol and if received the symbols following the input are executed.



Message input symbol

An input has a name and can come with parameters. To receive the parameters it is necessary to declare the variables that will be assigned to the parameters values in accordance with the message definition.
The syntax in the message input symbol is the following:
<Message name> [(<parameter name> {, <parameter name>}*)]
<parameter name> is a variable that needs to be declared.

If the parameter type is undeclared it is still possible to transmit unstructured data with the parameter length and a pointer on the data.
If the parameter length is unknown, because the parameters are unstructured data, it is also possible to get the parameter length assigned to a pre-declared variable.



Message with undeclared parameters

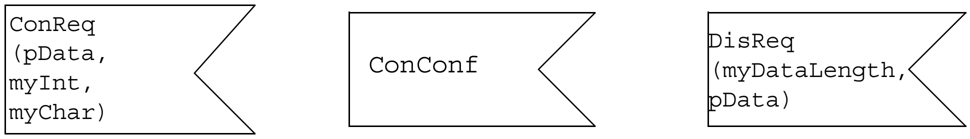
The syntax in the message input symbol is the following:
<Message name> [(<data length>, <pointer on data>)]

<data length> is a variable that needs to be declared as a long.
<pointer on data> is a variable that needs to be declared as an unsigned char *.

Examples:

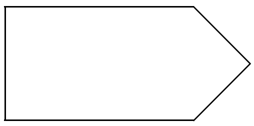
```
MESSAGE
  ConReq(myStruct *, int, char),
  ConConf,
  DisReq;
```

```
myStruct          *pData;
int               myInt;
char              myChar;
long              myDataLength;
unsigned char     *myData;
```



4.5 - Message output

A message output is used to exchange information. It puts data in the receiver's message queue in an asynchronous way.



Message output symbol

When a message has parameters, user defined local variables are used to assign the parameters. General syntax in the output symbol is:

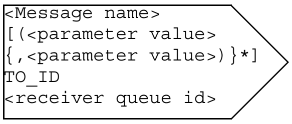
```
<message name>[(<parameter value> {,<parameter value>}*)] TO_XXX...
```

If the parameter is undefined the length of data and a pointer on the data can be provided. In that case, the symbol syntax is:

```
<message name>[(<data length>, <pointer on data>)] TO_XXX...
```

The syntax in the message output symbol can be written in several ways depending if the queue Id or the name of the receiver is known or not. A message can be sent to a queue Id or to a process name or via a channel or a gate. When communicating with the environment, a special syntax is provided.

4.5.1 To a queue Id



Message output to a queue id

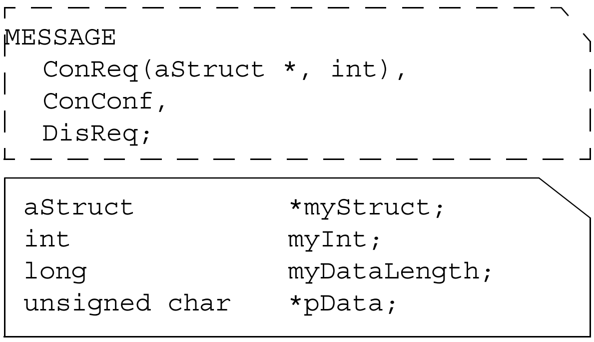
The symbol syntax is:

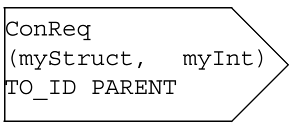
`<message name>[(<parameter value> {,<parameter value>}*)] TO_ID <receiver queue id>`

It can take the value given by the SDL-RT keywords:

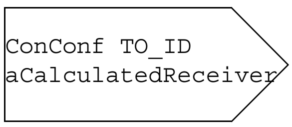
`PARENT`	The queue id of the parent process.
`SELF`	The queue id of the current process.
`OFFSPRING`	The queue id of the last created process if any or NULL if none.
`SENDER`	The queue id of the sender of the last received message.

Examples:

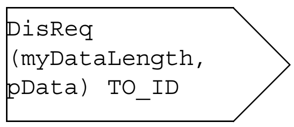




ConReq take 2 parameters. A pointer on `aStruct` and an `int`.

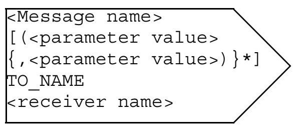


There is no parameter associated with the message `ConConf`.



DisReq parameter is undefined. Length of data and pointer on data are given.

4.5.2 To a process name



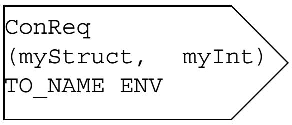
Message output to a process name

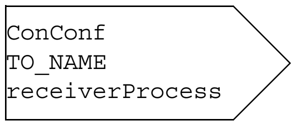
The syntax is:

`<message name>[(<parameter value> {,<parameter value>}*)] TO_NAME <receiver name>`

`<receiver name>` is the name of a process if unique or it can be `ENV` when simulating and the message is sent out of the SDL system.

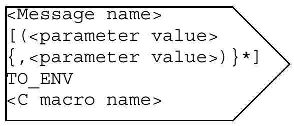
Examples:





Note:
If several instances have the same process name (several instances of the same process for example), the ’TO_NAME’ will send the message to the first created process with the corresponding name. Therefore this method should no be used when the process name is not unique within the system.

4.5.3 To the environment



Message output to environment

The symbol syntax is:

`<message name>[(<parameter value> {,<parameter value>}*)] TO_ENV [<C macro name>]`

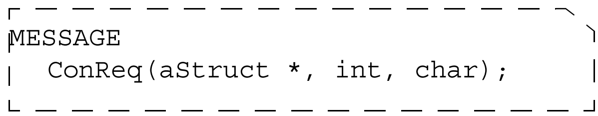
`<C macro name>` is the name of the macro that will be called when this SDL output symbol is hit. The macro will take 3 parameters:

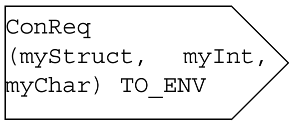
- name of message,
- length of a C struct that contains all parameters,
- pointer on the C struct containing all parameters.

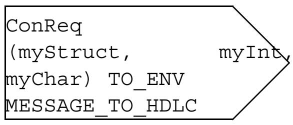
The fields of the implicit C struct will have the same type as the types defined for the message.

If no macro is declared the message will be sent to the environment.

Example:







In this second example the generated code will be:

```
MESSAGE_TO_HDLC(ConReq,implicitC-
StructLength,implicitCStructPointer)
```

The implicit C struct will have the following definition:

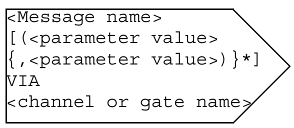
```
typedef struct implicitCStruct {
  aStruct*param1,
  int     param2;
  char    param3;
  } implicitCStruct;
```

That allows to re-use the same macro with different types of messages.

Note:
The implicit C struct memory space is implictly allocated and it is the C macro responsability to ensure it will be freed at some point.

4.5.4 Via a channel or a gate

A message can be sent via a channel in the case of a process or via a gate in the case of a process class.



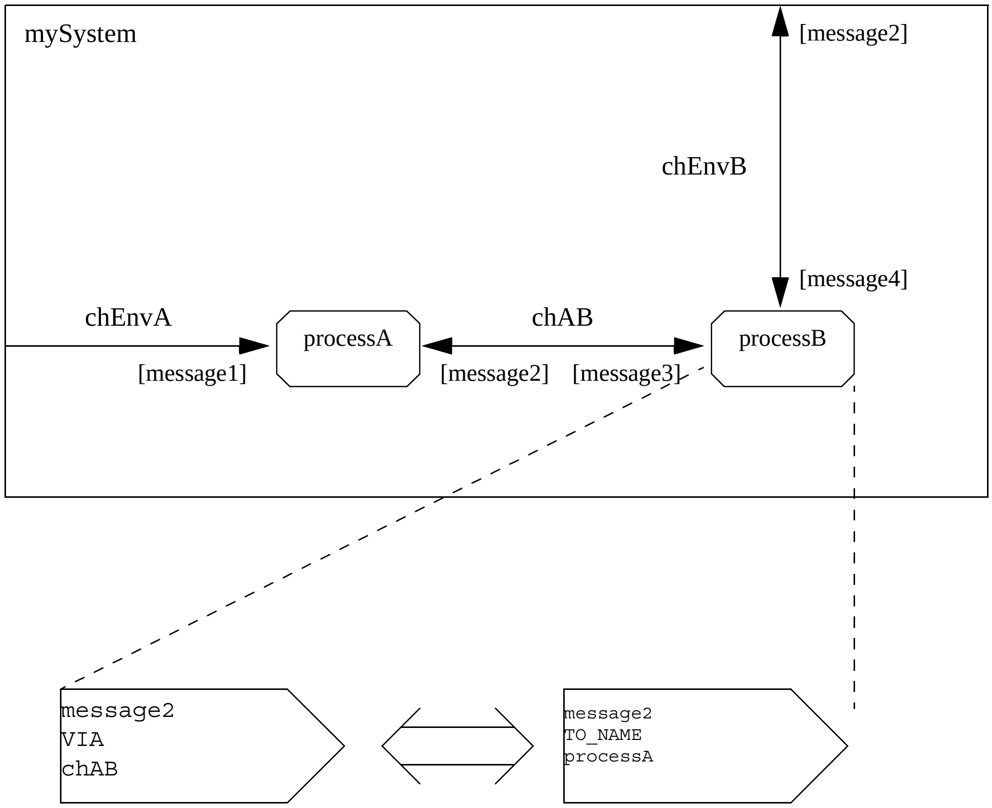
Message output via a channel or a gate

The symbol syntax is:

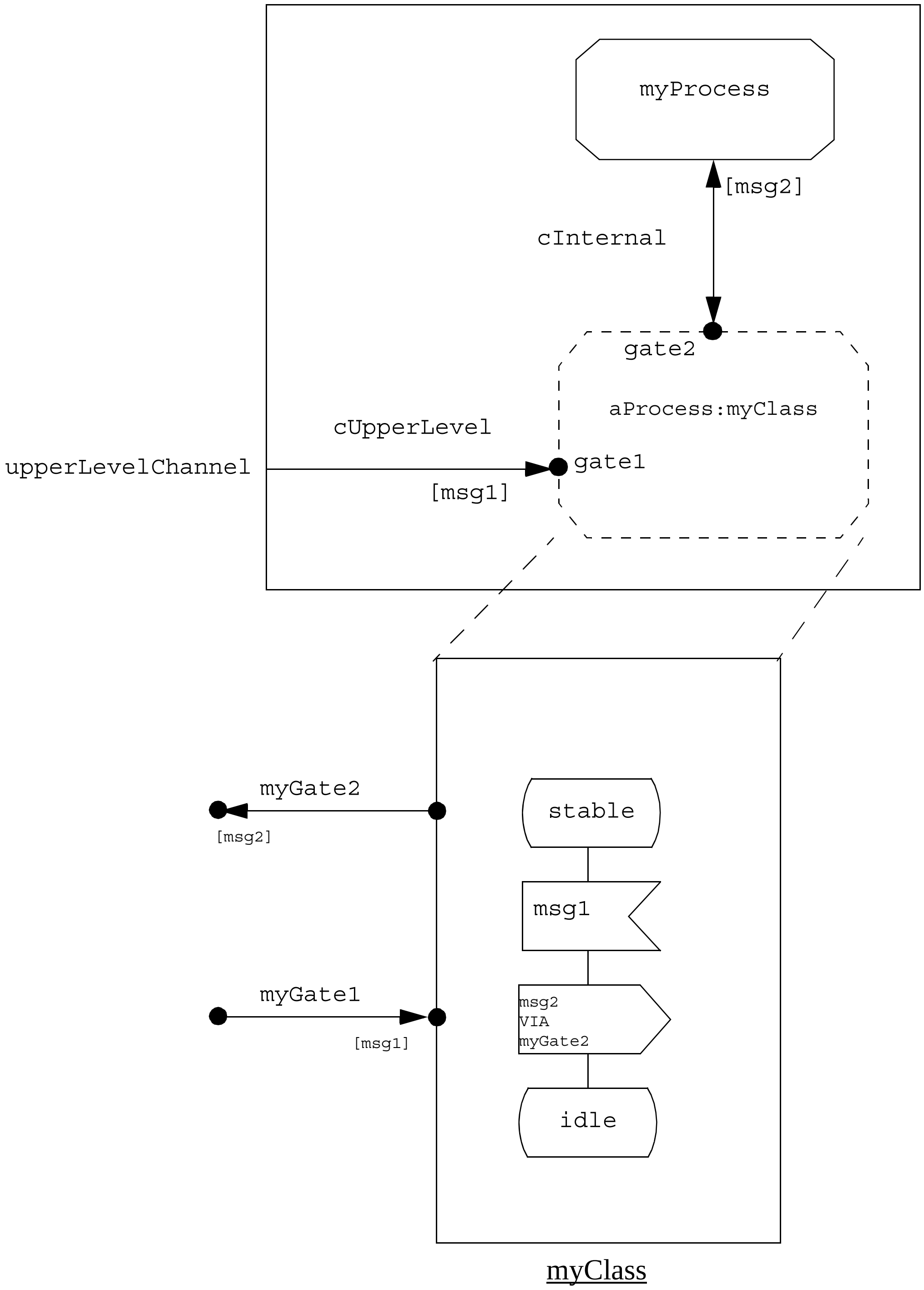
`<message name>[(<parameter value> {,<parameter value>}*)] VIA <channel or gate name>`

`<channel or gate name>` is the name of the channel or gate the message will go through.
This concept is especially usefull when using object orientation since classes are not supposed to know their environment; so messages are sent via the gates that will be connected to the surrouding environment when instanciated.

Examples:



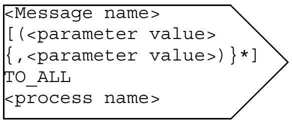
With the architecture defined above, both outputs are equivalent.



`aProcess` sends `msg2` to `myProcess` without knowing its name nor its PID

4.5.5 As a broadcast

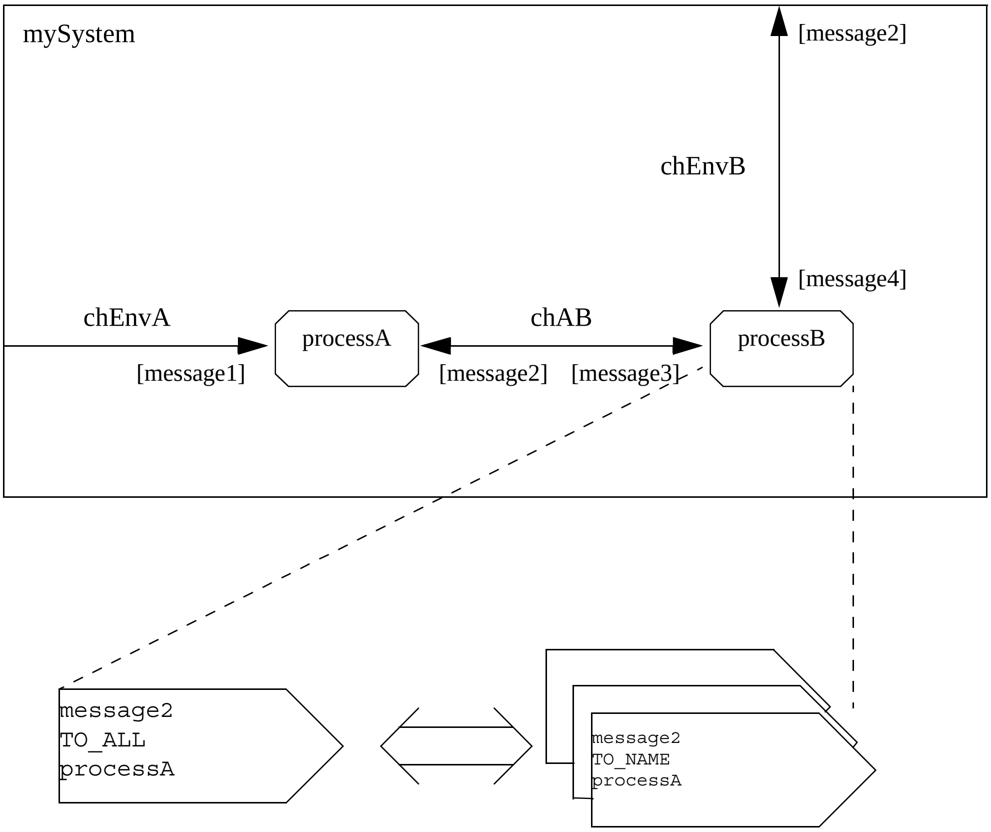
A message can be sent to all possible receivers at once with the broadcast. There are two typical situations: either there are several instances of the same process that can receive the message, or there are several processes connected via a channel that can carry the message. In the first case the TO_ALL keyword is used:



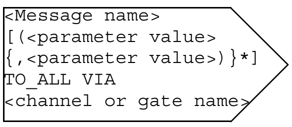
Message broadcast to all instances of a process

The symbol syntax is:
`<message name>[(<parameter value> {,<parameter value>}*)] TO_ALL <process name>`
`<process name>` is the name of the receiver process that may have several instances.
This concept is especially useful when some messages are to be received by all instances of the same process. It avoids having to store all active pids and to loop on the message output.

Examples:



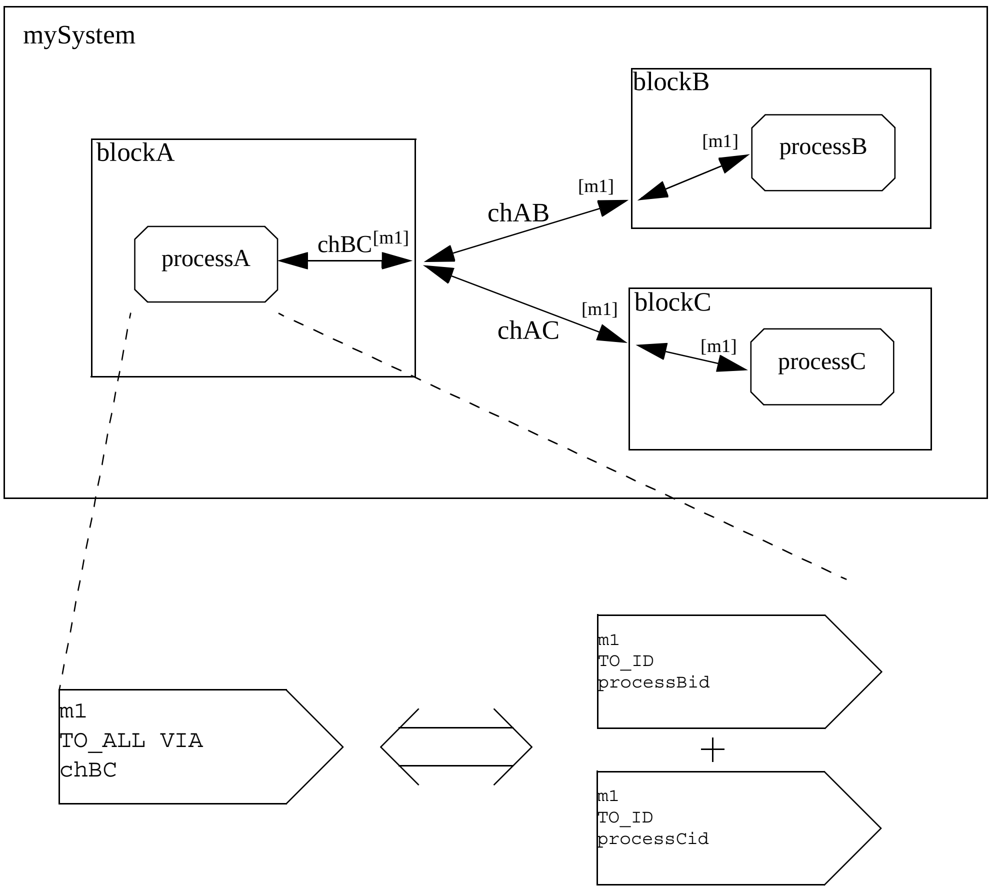
The broadcast can also be based on the channel connexion.



Message broadcast to all possible receivers at the end of the channel path

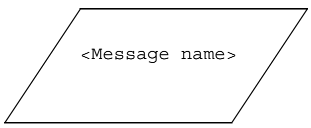
The symbol syntax is:
`<message name>[(<parameter value> {,<parameter value>}*)] TO_ALL VIA <channel or gate name>`
`<channel or gate name>` is the name of the channel or of the gate that carries the messsage. All possible receivers at the end of the channel path will receive a copy of the message.

Examples:



4.6 - Message save

A process may have intermediate states that can not deal with new request until the on-going job is done. These new requests should not be lost but kept until the process reaches a stable state. Save concept has been made for that matter, it basically holds the message until it can be treated.



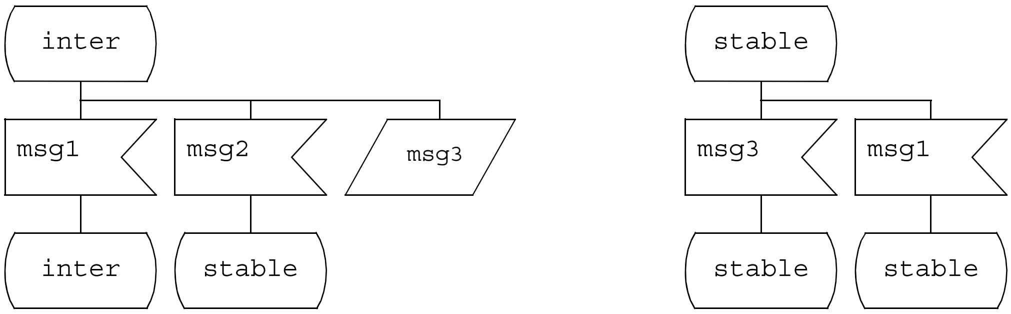
Save symbol

The Save symbol is followed by no symbol. When the process changes to a new state the saved messages will be the first to be treated (after continuous signals if any).
The symbol syntax is:

```
<message name>
```

Even if the message has parameters.

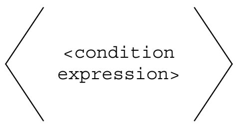
Example:



Let's consider the above process in state `inter` to receive the following messages: `msg3`, `msg2`, `msg1`. `msg3` will be saved, `msg2` will make the process go to state `stable`. Since `msg3` has been saved it will first be treated and finally `msg1`.

4.7 - Continuous signal

A continuous signal is an expression that is evaluated right after a process reaches a new state. It is evaluated before any message input or saved messages.

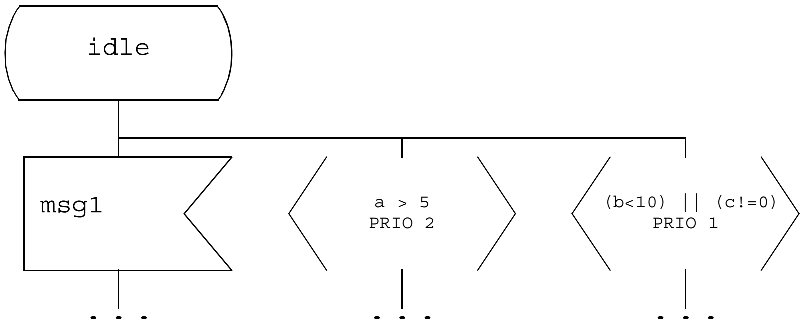


Continuous signal symbol

The continuous signal expression to evaluate can contain any standard C expression that returns a C true/false expression. Since an SDL state can contain several continuous signal a priority level needs to be defined with the `PRIO` keyword. Lower values correspond to higher priorities. A continuous signal symbol can be followed by any other symbol except another continuous signal or a message input. The syntax is:

```
<C condition expression>
PRIO <priority level>
```

Example:



In the above example, when the process gets in state idle it will first evaluate expression `(b<10) || (c!=0)`. If the expression is not true or if the process stayed in the same state it will evaluate expression `a > 5`. If the expression is not true or if the process stayed in the same state it will execute `msg1` transition.

4.8 - Action

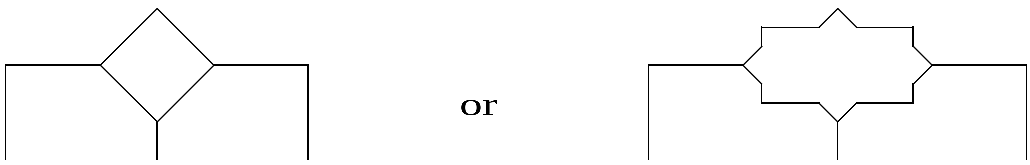
An action symbol contains a set of instructions in C code. The syntax is the one of C language.

Example:

```
/* Say hi to your friend */
printf("Hello world !\n");
for (i=0;i<MAX;i++)
   {
   newString[i] = oldString[i];
   }
```

4.9 - Decision

A decision symbol can be seen as a C switch / case.



Decision symbols

Since it is graphical and therefore uses quite some space on the diagram it is recommended to use it when its result modifies the resulting process state. The decision symbol is a diamond with branches. Since a diamond is one of the worst shape to put text in it, it can be a "diamonded" rectangle. Each branch can be seen as a case of the switch.
The expression to evaluate in the symbol can contain:

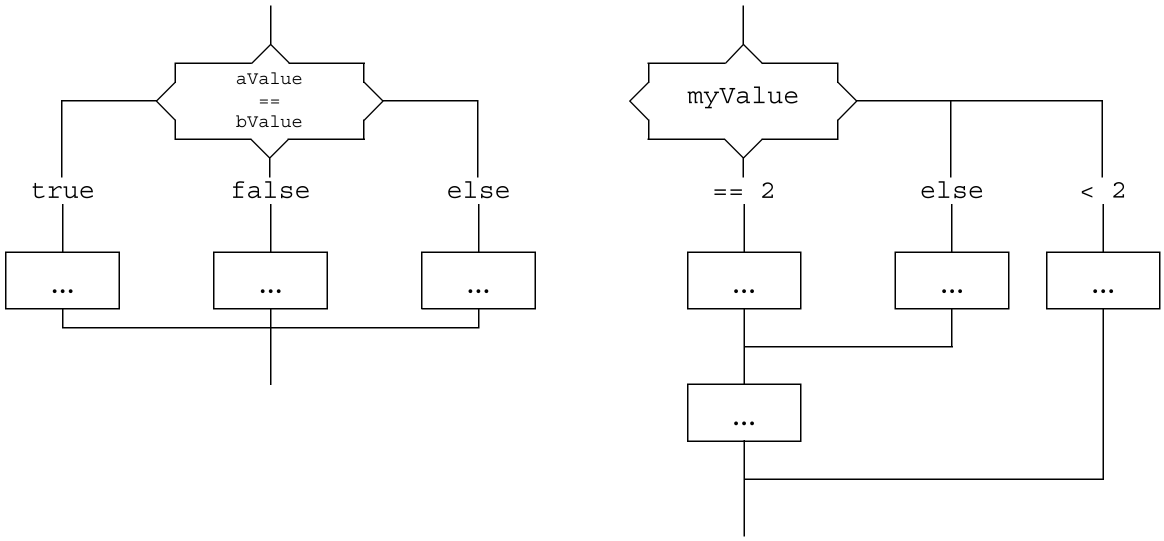
- any standard C expression that returns a C true/false expression,
- an expression that will be evaluated against the values in the decision branches.

The values of the branches have keyword expressions such as:

- `>, <, >=, <=, !=, ==`
- `true, false, else`

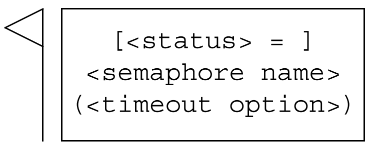
The `else` branch contains the default branch if no other branch made it.

Examples:



4.10 - Semaphore take

The Semaphore take symbol is used when the process attempts to take a semaphore.



Semaphore take symbol

To take a semaphore, the syntax in the 'semaphore take SDL-RT graphical symbol' is:

`[<status> = ] <semaphore name>(<timeout option>)`

where `<timeout option>` is:

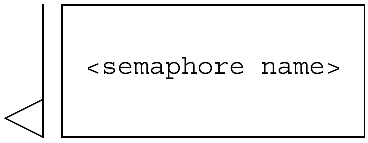
- `FOREVER`
  Hangs on the semaphore forever if not available.
- `NO_WAIT`
  Does not hang on the semaphore at all if not available.
- `<number of ticks to wait for>`
  Hangs on the semaphore the specified number of ticks if not available.

and `<status>` is:

- `OK`
  If the semaphore has been successfully taken
- `ERROR`

If the semaphore was not found or if the take attempt timed out.

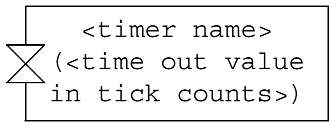
4.11 - Semaphore give



Semaphore give symbol

To give a semaphore, the syntax in the ‘semaphore give SDL-RT graphical symbol’ is:
`<semaphore name>`

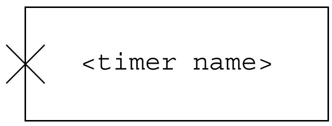
4.12 - Timer start



Timer start symbol

To start a timer the syntax in the ‘start timer SDL-RT graphical symbol’ is :
`<timer name>(<time value in tick counts>)`
`<time value in tick counts>` is usually an ‘int’ but is RTOS and target dependant.

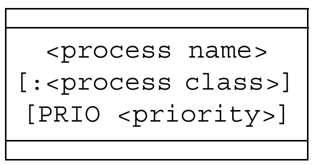
4.13 - Timer stop



Timer stop symbol

To cancel a timer the syntax in the ‘cancel timer SDL-RT graphical symbol’ is :
`<timer name>`

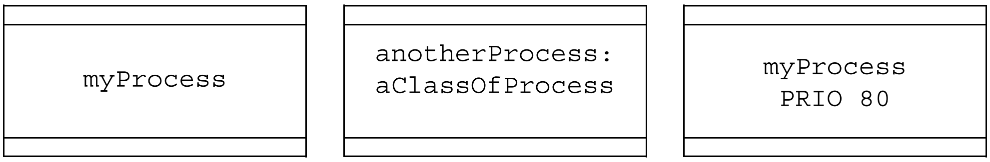
4.14 - Task creation



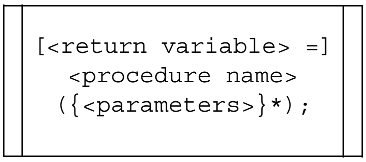
Task creation symbol

To create a process the syntax in the create process symbol is:
`<process name>[:<process class>] [PRIO <priority>]`
to create one instance of `<process class>` named `<process name>` with priority `<priority>`.

Examples:



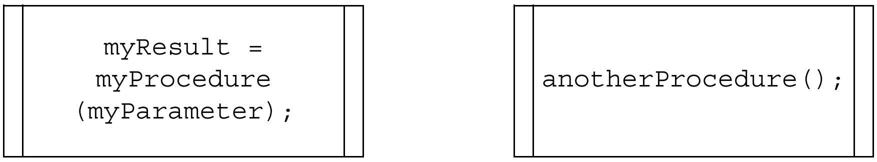
4.15 - Procedure call



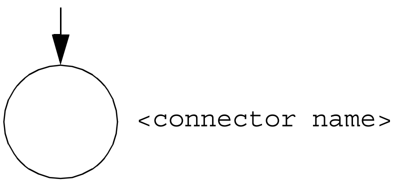
Procedure call symbol

The procedure call symbol is used to call an SDL-RT procedure (Cf. “Procedure declaration” on page 40). Since it is possible to call any C function in an SDL-RT action symbol it is important to note SDL-RT procedures are different because they know the calling process context, e.g. SDL-RT keywords such as SENDER, OFFSPRING, PARENT are the ones of the calling process.
The syntax in the procedure call SDL graphical symbol is the standard C syntax:
`[<return variable> =] <procedure name>({<parameters>}*);`

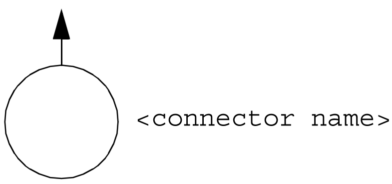
Examples:



4.16 - Connectors



Connector out



Connector in

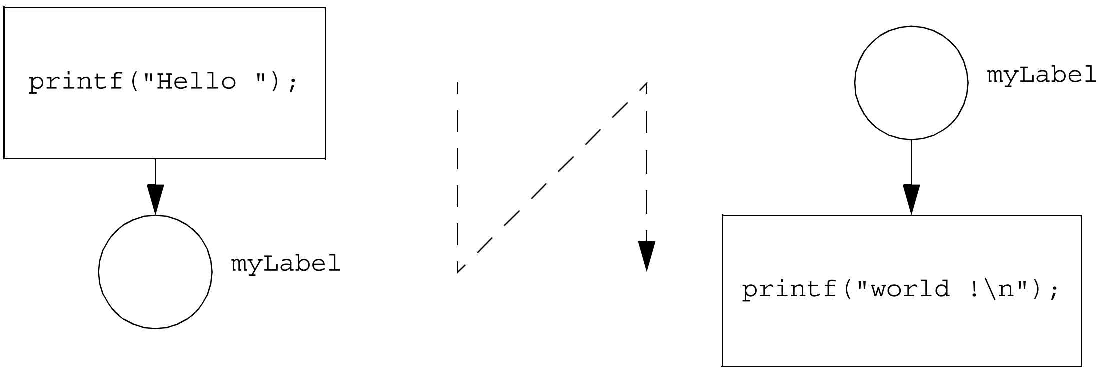
Connectors are used to:

- split a transition into several pieces so that the diagram stays legible and printable,
- to gather different branches to a same point.

A connector-out symbol has a name that relates to a connector-in. The flow of execution goes from the connector out to the connector in symbol.
A connector contains a name that has to be unique in the process. The syntax is:

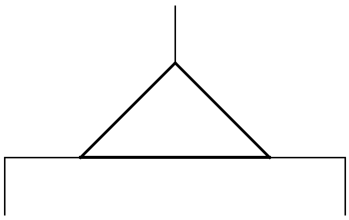
```
<connector name>
```

Examples:



4.17 - Transition option

Transition options are similar to C `#ifdef`.



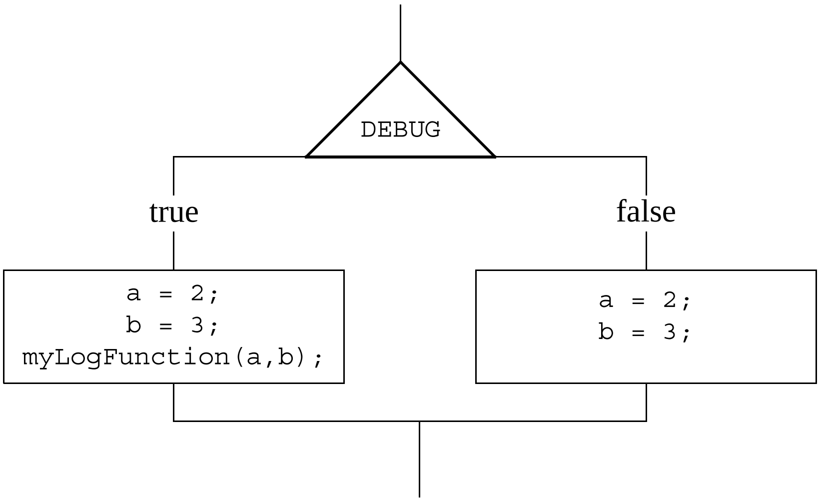
Transition option symbol

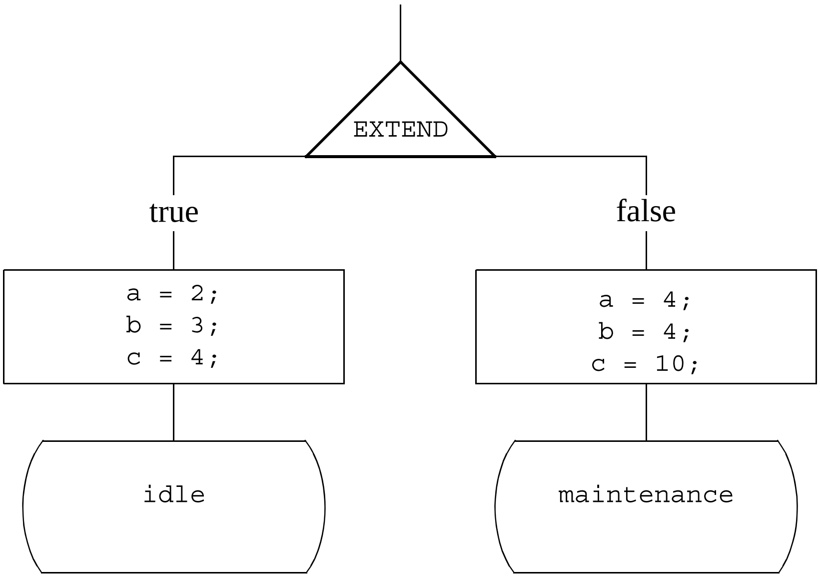
The branches of the symbol have values `true` or `false`. The `true` branch is defined when the expression is defined so the equivalent C code is:

```
#ifdef <expression>
```

The branches can stay separated to the end of the transition or they can meet again and close the option as would do an `#endif`.

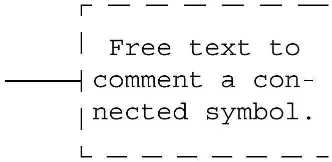
Examples:





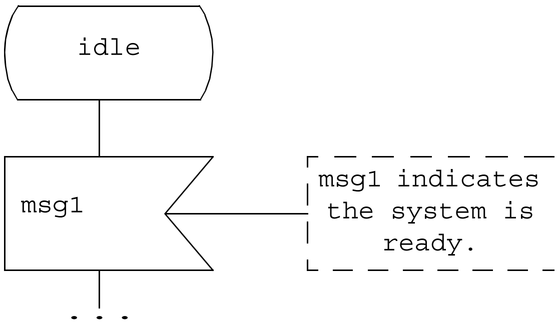
4.18 - Comment

The comment symbol allows to write any type of informal text and connect it to the desired symbol. If needed the comment symbol can be left unconnected.



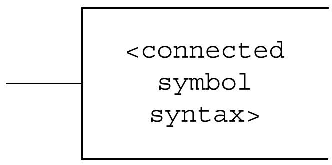
Comment symbol

Example:



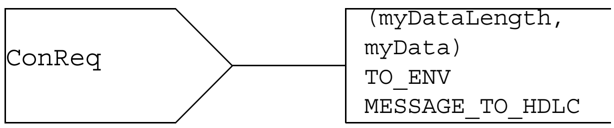
4.19 - Extension

The extension symbol is used to complete an expression in a symbol. The expression in the extension symbol is considered part of the expression in the connected symbol. Therefore the syntax is the one of the connected symbol.

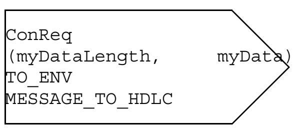


Extension symbol

Example:



is equivalent to:



4.20 - Procedure start

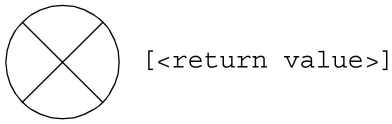
This symbol is specific to a procedure diagram. It indicates the procedure entry point.

Procedure start symbol

There is no syntax associated with this symbol.

4.21 - Procedure return

This symbol is specific to a procedure diagram. It indicates the end of the procedure.

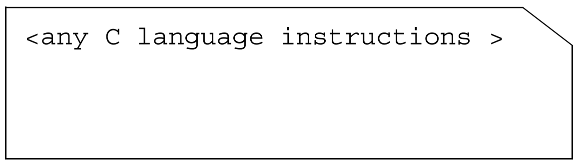


Procedure return symbol

This symbol is specific to a procedure diagram. It indicates the end of the procedure. If the procedure has a return value it should be placed by the symbol.

4.22 - Text symbol

This symbol is used to declare C types variables.

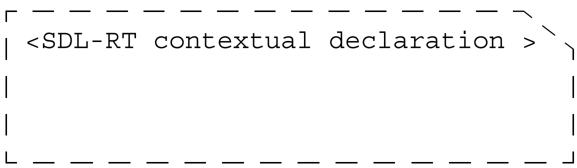


Text symbol

The syntax is C language syntax.

4.23 - Additional heading symbol

This symbol is used to declare SDL-RT specific headings.



Additional heading symbol

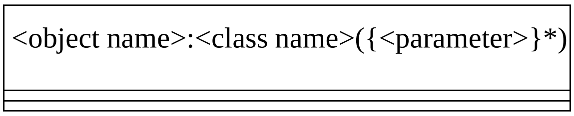
It has a specific syntax depending in which diagram it is used.

- Block heading
  Allows to declare messages and messages lists:

```
MESSAGE <message name> [(<param type>)] {,<msg name> [(<param type>)]};
MESSAGE_LIST <message list name> = <message name> {,<message name>}*;
```

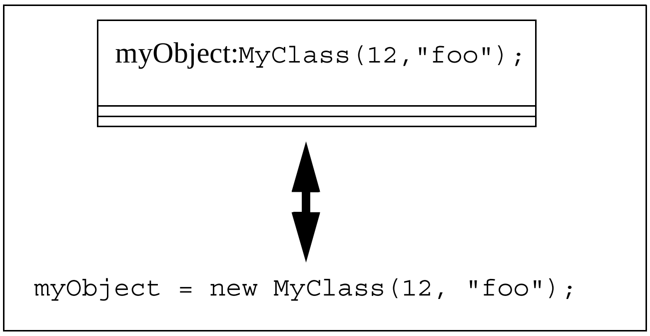
- Process class heading
  Allows to specify the superclass to inherit from:
  ```
  INHERITS <superclass name>;
  ```
- System, Block, Block class heading
  Allows to specify the package to use:
  ```
  USE <package name>;
  ```
- Process or Process class heading
  Allows to define the stack size:
  ```
  STACK <stack size value>;
  ```

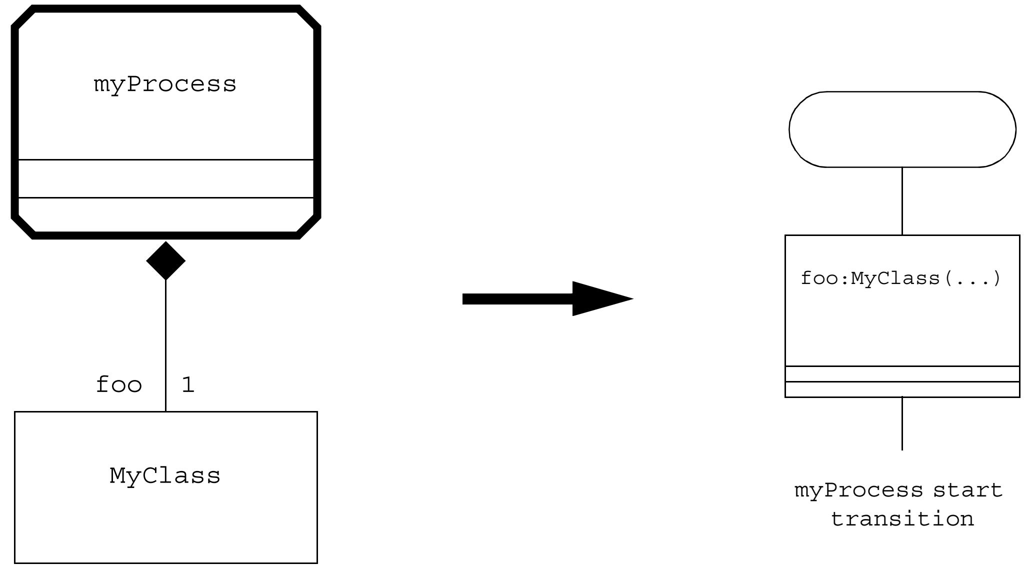
4.24 - Object creation symbol



This is equivalent to creating an instance of class <class name> named <object name>.
This symbol can be used by tools to check consistency between the dynamic SDL view and the static UML view.

<u>Examples:</u>

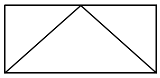




The relation described in the class diagram implies the instance of `MyClass` named after role name `foo` must be created in the start transition.

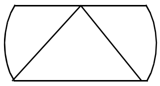
4.25 - Super class transition symbol

This symbol is used to call the corresponding super class transition. It can be used anywhere in the transition between the “Message input” symbol and the next “State” symbol. The sub class transition signature must be exactly the same as the super class transition signature including the variale names. More explanations in “Object orientation” on page 67.



4.26 - Super class next state symbol

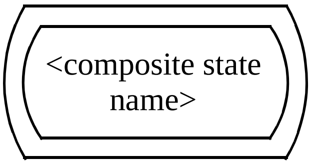
This symbol is used to set the next state to the one of the super class. It replaces the standard “State” symbol at the end of a transition. More explanations in “Object orientation” on page 67.



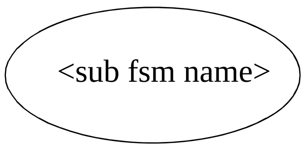
4.27 - Composite state

4.27.1 Composite state definition

A composite state is a state composed of sub finite state machines. Each sub-fsm handles a different subset of messages. The super-fsm also handles its own inputs. When a message is for one of the sub-fsm the super-state does not change. But when a message is for the super-fsm all sub-fsm are terminated.

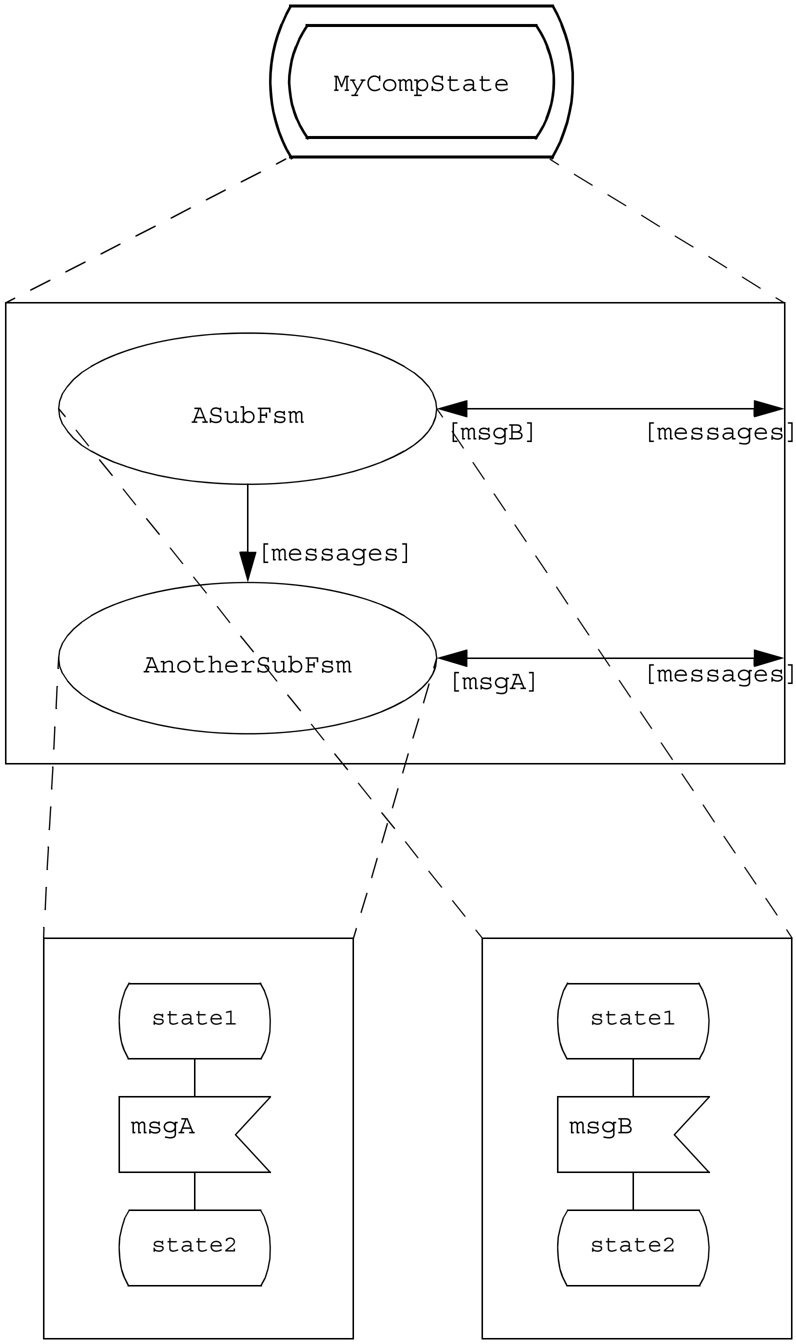


Composite state definition symbol



Sub fsm definition symbol

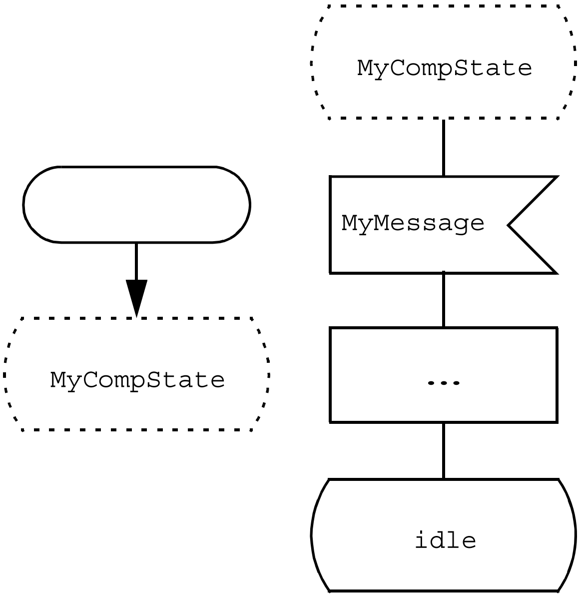
The Sub-fsm definition symbols are connected to channels. Each message is routed to a specific sub-fsm, the same message can not be received by two different sub-fsm.
The definition is done as described below:



This mecanism is currently known as "state hierarchy" in UML or "services" in SDL.

4.27.2 Composite state usage

A dashed state symbol is used to indicate the fsm is getting into a composite state.



When in the composite state `MyCompState`, messages are routed toward the corresponding sub-fsm. When receiving the `MyMessage` message, the sub-fsm are terminated and the super fsm transition is executed. If the same message can be received by the super fsm and by one of the sub fsm, the super fsm transition has priority.

4.28 - Symbols ordering

The following table shows which symbols can be connected to a specific symbol.

The symbol in this column can be followed by the ticked symbols in its row.	start	state	stop	input	output	save	continuous signal	action	decision	semaphore take	semaphore give	timer start	timer stop	task creation	procedure call	connector in	connector out	transition option	procedure start	procedure return	object creation	super class transition	super class next state
start	-	X	X	-	X	-	-	X	X	X	X	X	-	X	X	X	X	X	-	-	X	X	X
state	-	-	-	X	-	X	X	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
stop	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
input	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
output	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
save	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
continuous	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	-	-
action	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
semaphore take	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
semaphore give	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
timer start	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
timer stop	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
task creation	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
procedure call	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	X	X	-	X	X	X	X
connector out	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
connector in	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	-	-	X	-	X	X	X	X
transition option	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	-	X	X	-	X	X	X	X
procedure start	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	-	X	X	-	X	X	X	X
procedure return	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
super class transition	-	X	X	-	X	-	-	X	X	X	X	X	X	X	X	X	-	X	-	X	X	-	X
super class next state	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

The table above should be read row by row. The symbol in the left column can be followed by the ticked symbols on its row. For example the stop symbol can not be followed by any other symbol. The state symbol can be followed by input, save, or continuous signal symbols.

5 - Declarations

5.1 - Process

A process is implicitly declared in the architecture of the system (Cf. “Architecture” on page 8) since the communication channels need to be connected.

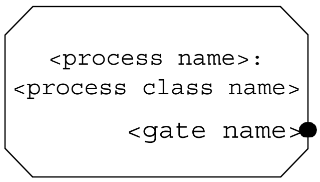
Process symbol

A process has an initial number of instances at startup and a maximum number of instances. A process can also be an instance of a process class (Cf. “Object orientation” on page 67), in that case the name of the class follows the name of the instance after a colon.
The general syntax is:

```
<process instance name>[:<process class>][(<initial number of instances>, <maximum number of instances>)] [PRIO <priority>]
```

The priority is the one of the target RTOS.
Please note the stack size can be defined in the process or process class additional heading symbol as described in paragraph “Additional heading symbol” on page 32.

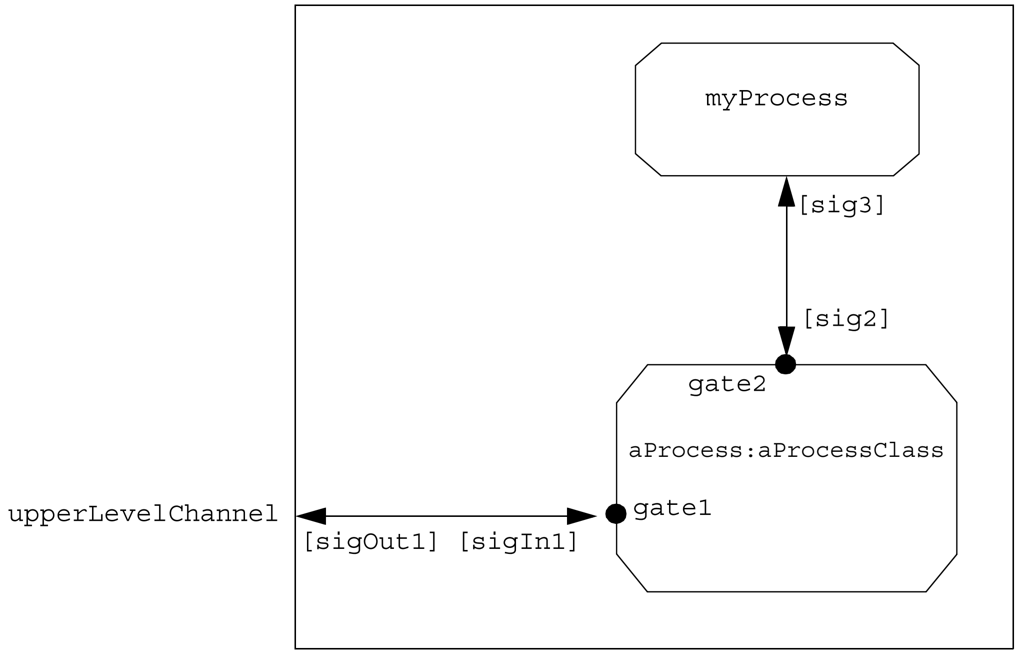
When a process is an instance of a process class the gates of the process class need to be connected in the architecture diagram. The names of the gates appear in the process symbol with a black circle representing the connection point.



Process class instance

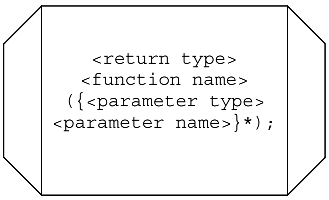
The messages defined in the package going through the gates must be consistent with the messages listed in the architecture diagram where the process instance is defined.

Example:



5.2 - Procedure declaration

An SDL-RT procedure can be defined in any diagram: system, block, or process. It is usually not connected to the architecture but since it can output messages a channel can be connected to it for informational purpose.



Procedure declaration symbol

The declaration syntax is the same as a C function. A procedure definition can be done graphically with SDL-RT or textually in a standard C file.

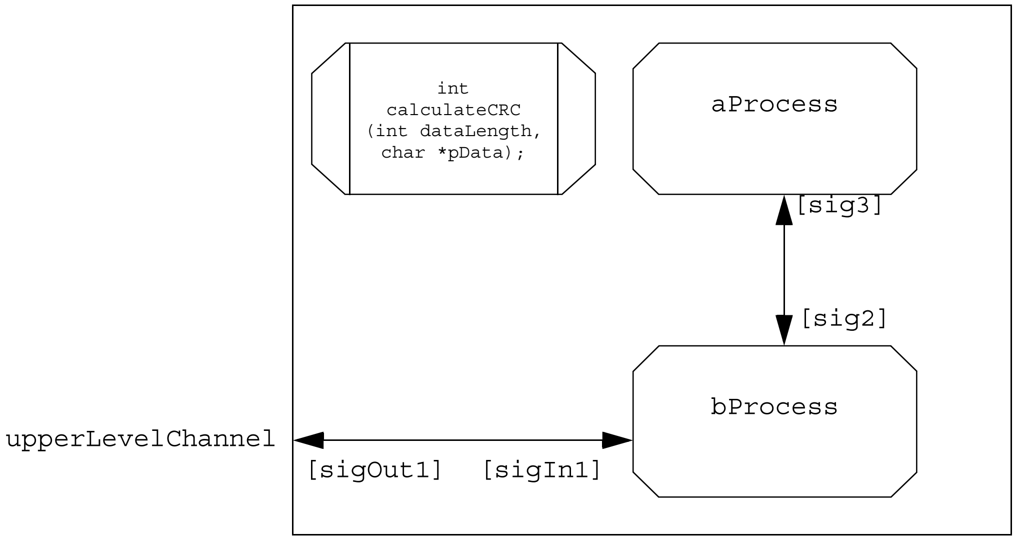
5.2.1 SDL-RT defined procedure

If defined with SDL-RT the calling process context is implicitly given to the procedure. So if a message output is done, the message will be output from the process calling the procedure. That is why the message should be defined in one of the channels connected to the process instead of a channel connected to a procedure. To call such a procedure the procedure call symbol should be used.

5.2.2 C defined procedure

If defined in C language the process context is not present. To call such a procedure a standard C statement should be used in a action symbol.

Example:



5.3 - Messages

Messages are declared at any architecture level in the additional heading symbol. A message declaration may include one or several parameters. The parameters data types are declared in C. The syntax is:

```
MESSAGE <message name> [(<parameter type> {,parameter type}*)] {,<message name> [(<parameter type>)]}*;
```

It is also possible to declare message lists to make the architecture view more synthetic. Such a declaration can be made at any architecture level in the additional heading symbol. The syntax is:

```
MESSAGE_LIST <message list name> = <message name> {, <message name>}*{, (<message list name>)}*;
```

The message parameters are not present when defining a message list. A message list can contain a message list, in that case the included message list name is surrounded by parenthesis.

Example:

```
MESSAGE
  msg1(myStruct *, int, char),
  msg2(void),
  msg3(void *, short),
  msg4(int *),
  msg5;

MESSAGE_LIST
  myMessageList = msg1, msg2;
MESSAGE_LIST
  anotherMessageList = (myMessageList), msg3;
```

5.4 - Timers

There is no need to declare timers. They are self declared when used in a diagram.

5.5 - Semaphores

Semaphores can be declared at any architectural level. The general syntax in the declaration symbol is:

```
<semaphore type>
  <semaphore name>({<list of options>[,]}*);
```

Semaphore declaration

Where the types and options are:

- BINARY <semaphore name> ([PRIO | FIFO], [INITIAL_EMPTY | INITIAL_FULL])
- MUTEX <semaphore name> ([PRIO | FIFO] [, DELETE_SAFE] [, INVERSION_SAFE])
- COUNTING <semaphore name> ([PRIO | FIFO], <initial count>)

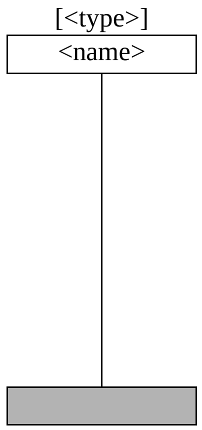
It is important to note the semaphore is identified by its name.

6 - MSC

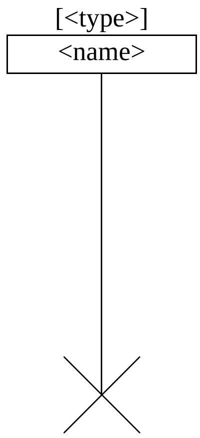
SDL-RT integrates the Message Sequence Chart dynamic view. On such a diagram, time flows from top to bottom. Lifelines represent SDL-RT agents or semaphores and key SDL-RT events are represented. The diagram put up front the sequence in which the events occur.
In the case of embedded C++ it is possible to use a lifeline to represent an object. In that case the type is `object` and the name should be `<object name>:<class name>`

6.1 - Agent instance

An agent instance starts with an agent instance head followed by an instance axis and ends with an instance tail or an instance stop as shown in the diagrams below.



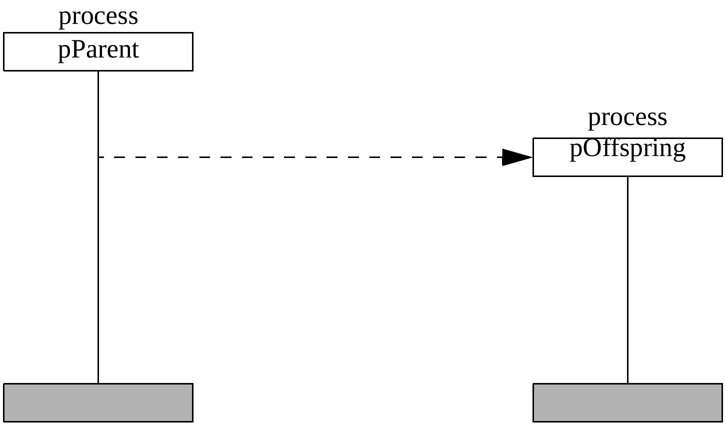
<u>Lifeline with an instance tail symbol</u>



<u>Lifeline with an instance stop symbol</u>

The type of the agent can be specified on top of the head symbol and the name of the agent is written in the instance head symbol. The instance tail symbol means the agent lives after the diagram. The instance stop symbol means the agent no longer exist after the symbol.
When an agent creates another agent a dashed arrow goes from the parent agent to the child agent.

Example:



Process pParent creates process pOffspring

6.2 - Semaphore representation

A semaphore representation is made of a semaphore head, a lifeline, and a semaphore end or tail. The symbols are the same as for a process except for the head of the semaphore.

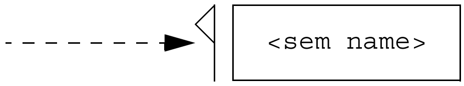
Semaphore with an instance tail symbol

Semaphore with an instance stop symbol

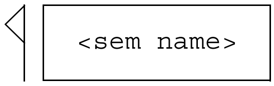
6.3 - Semaphore manipulations

Several cases are to be considered with semaphore manipulations. A process makes an attempt to take a semaphore, its attempt can be successful or unsuccessful, if successful the semaphore might still be available (counting semaphore) or become unavailable. During the time the semaphore is unavailable, its lifeline gets thicker until it is released.

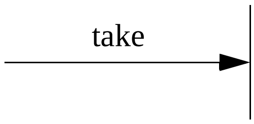
The manipulation symbols are the following:



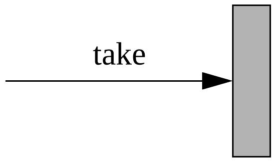
Semaphore creation from a known process.



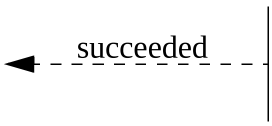
Semaphore creation from an unknown process.



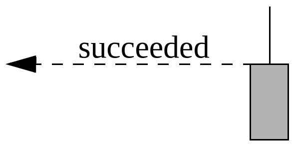
Semaphore take attempt.



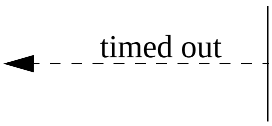
Semaphore take attempt on a locked semaphore.



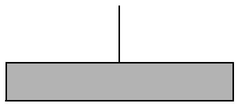
Semaphore take successfull but semaphore is still available.



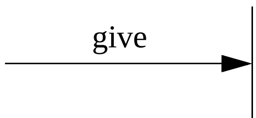
Semaphore take successfull and the semaphore is not available any more.



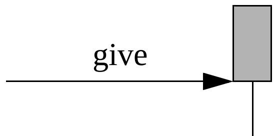
Semaphore take timed out.



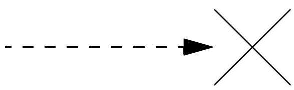
Semaphore continues.



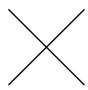
Semaphore give. The semaphore was available before the give.



Semaphore give. The semaphore was unavailable before the give.

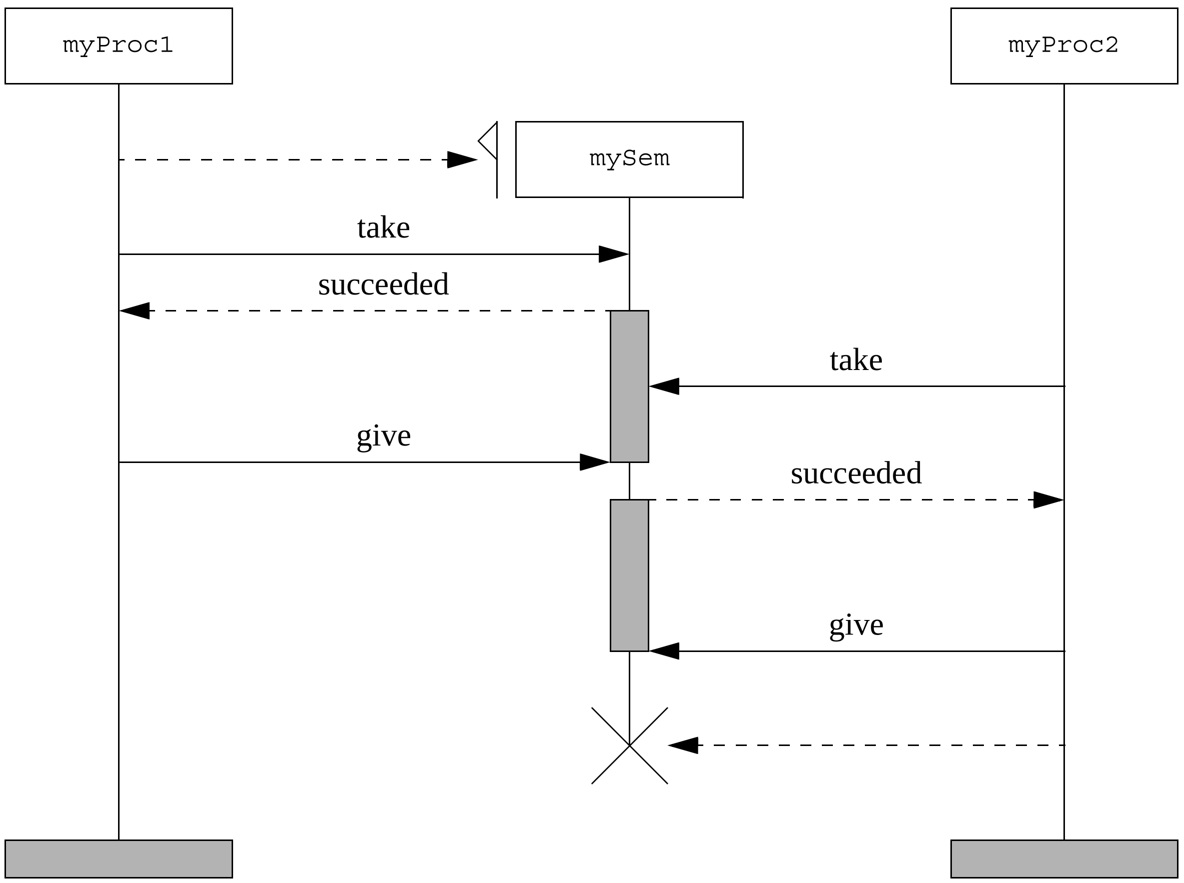


Semaphore is killed by a known process.



Semaphore is killed by an unknown process.

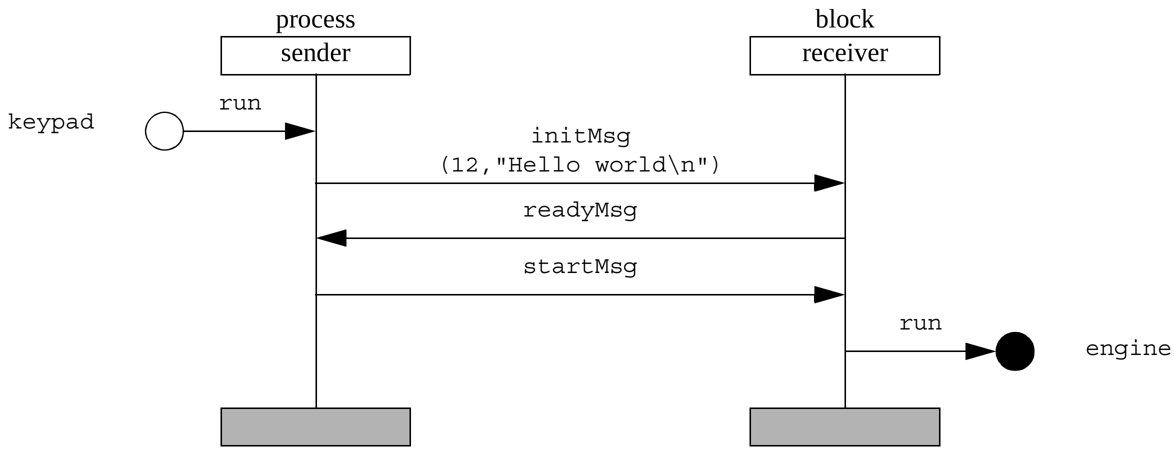
Example:



Process `myProc1` first creates semaphore `mySem`, then takes it successfully. Process `myProc2` makes an attempt to take semaphore `mySem` but gets blocked on it. Process `myProc1` releases the semaphore so `myProc2` successfully gets the semaphore. Process `myProc2` gives it back, and kills it.

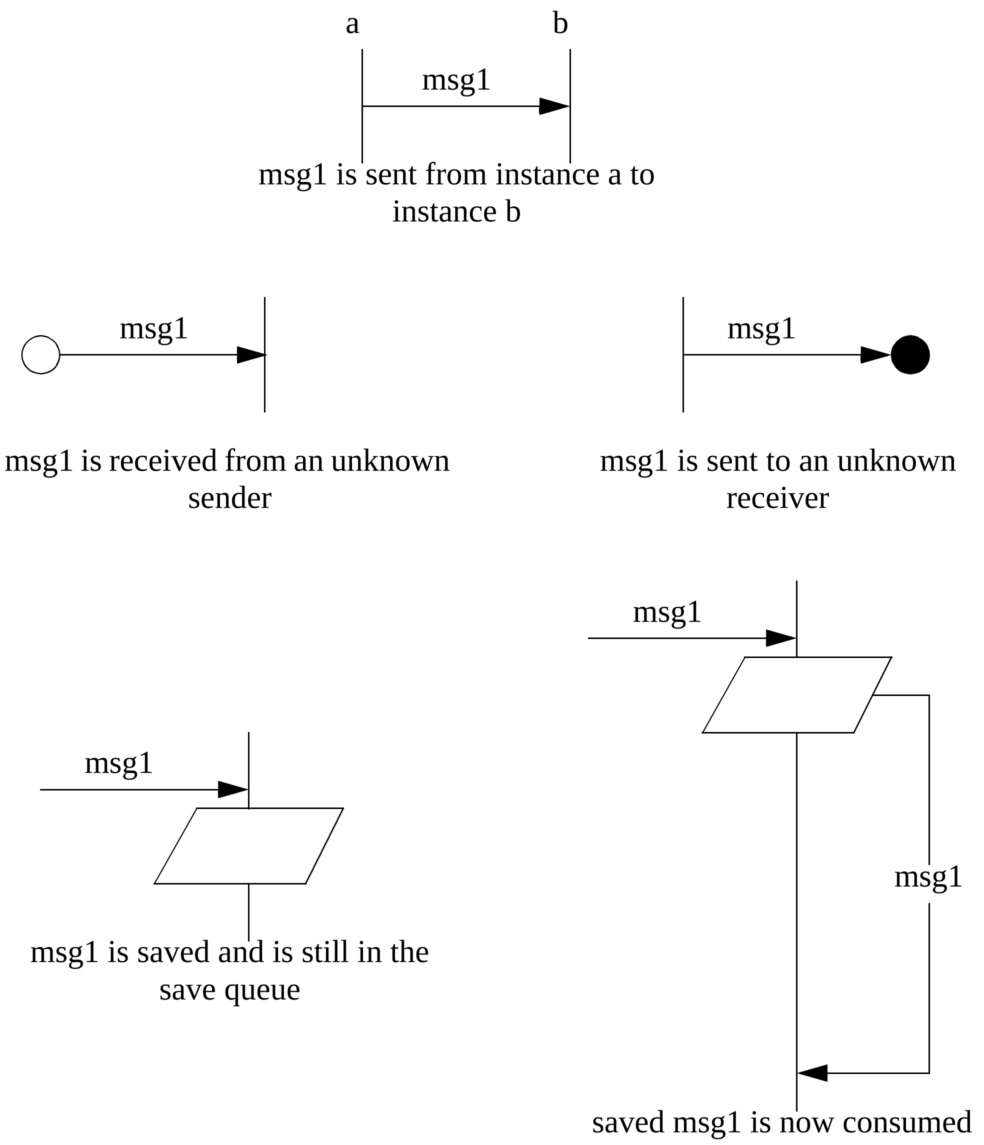
6.4 - Message exchange

A message symbol is a simple arrow with its name and optional parameters next to it. The arrow can be horizontal meaning the message arrived instantly to the receiver or the arrow can go down to show the message arrived after a certain time or after another event. A message can not go up ! When the sender and the receiver are represented on the diagram the arrow is connected to their instances. If the sender is missing it is replaced by a white circle, if the receiver is missing it is replaced by a black circle.The name of the sender or the receiver can optionally be written next to the circle.



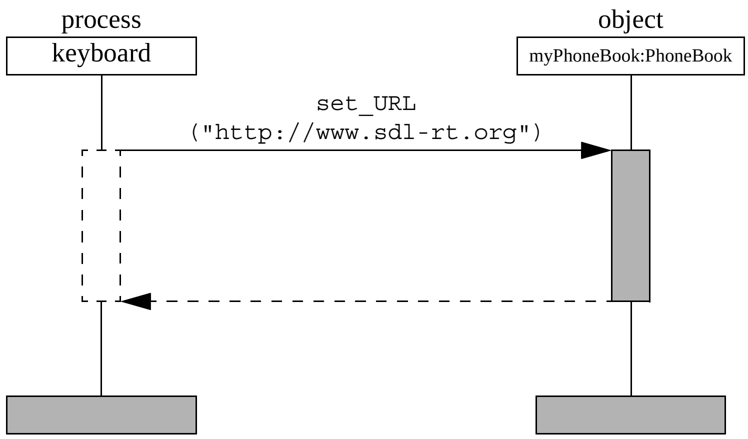
An external agent called `keypad` sends `run` message to process `sender`. Process `sender` sends `initMsg` that is considered to be received immediatly to block `receiver`. Block `receiver` replies `readyMsg`, process `sender` sends `startMsg`, and block `receiver` sends `run` to an external agent.

A message is considered received by an agent when it is read from the agent's message queue; not when it arrives in the message queue !



6.5 - Synchronous calls

This representation is used when using embedded C++ to show method calls on an object. Object can be represented by lifelines. Synchronous calls are shown with an arrow to the instance representing the object. While the object has the focus its lifeline becomes a black rectangle and the agent lifeline becomes a white rectangle. That means the execution flow has been transferred to the object. When the method returns a dashed arrow return to the method caller.



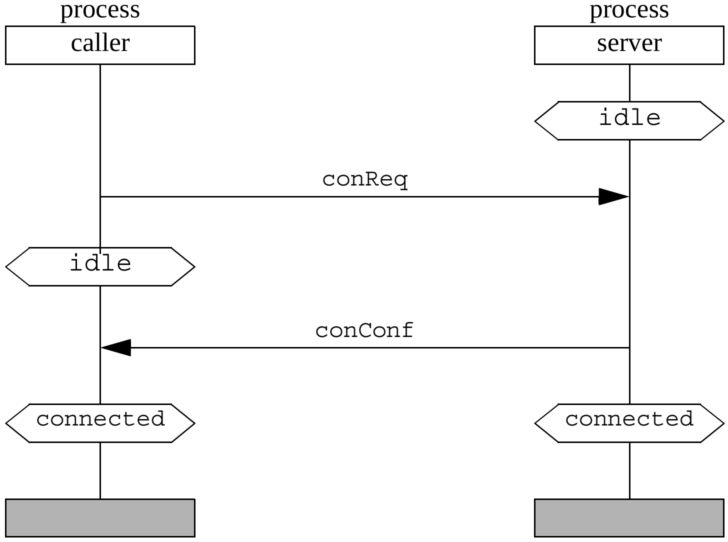
Process `keyboard` calls method `set_URL` from `myPhoneBook` object that is an instance of `PhoneBook` class.

6.6 - State

A lifeline represents a process and depending on its internal state a process reacts differently to the same message. It is interesting to represent a process state on its lifeline. It is also interesting to represent a global state for information. In that case the state symbol covers the concerned instances. In both cases the same symbol is used.

State symbol

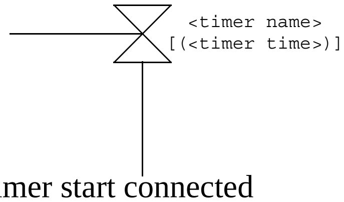
Example:



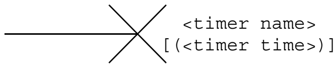
Process `server` goes to `idle` state. Process `caller` in its start transition sends a `conReq` to `server` and goes to state `idle`. Process `server` returns an `conConf` message and goes to `connected` state. When `conConf` message is received by process `caller` it also moves to `connected` state.

6.7 - Timers

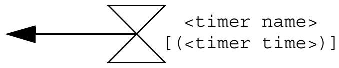
Two symbols are available for each timer action depending if the beginning and the end of the timer are connected or not. The timer name is by the cross and timeout value is optional. When specified the timeout value unit is not specified; it is usually RTOS tick counts.



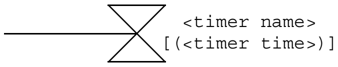
Timer start connected



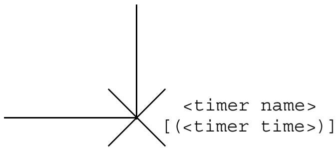
Timer stop unconnected



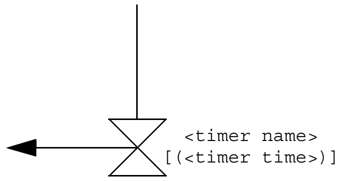
Timeout unconnected



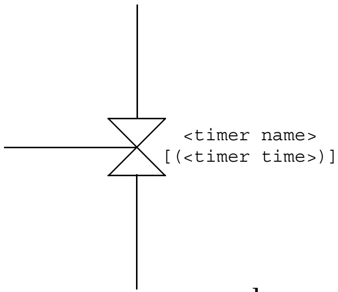
Timer start unconnected



Timer stop connected

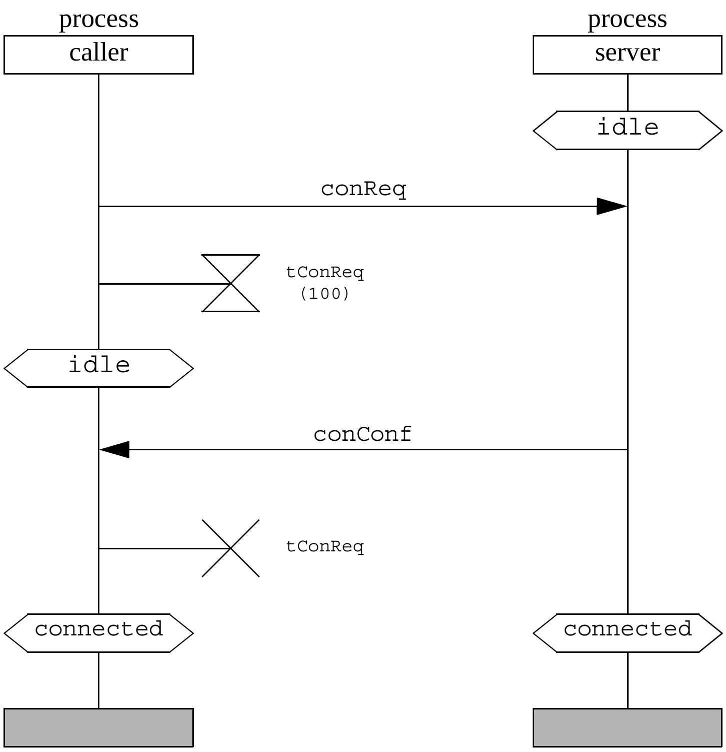


Timeout connected

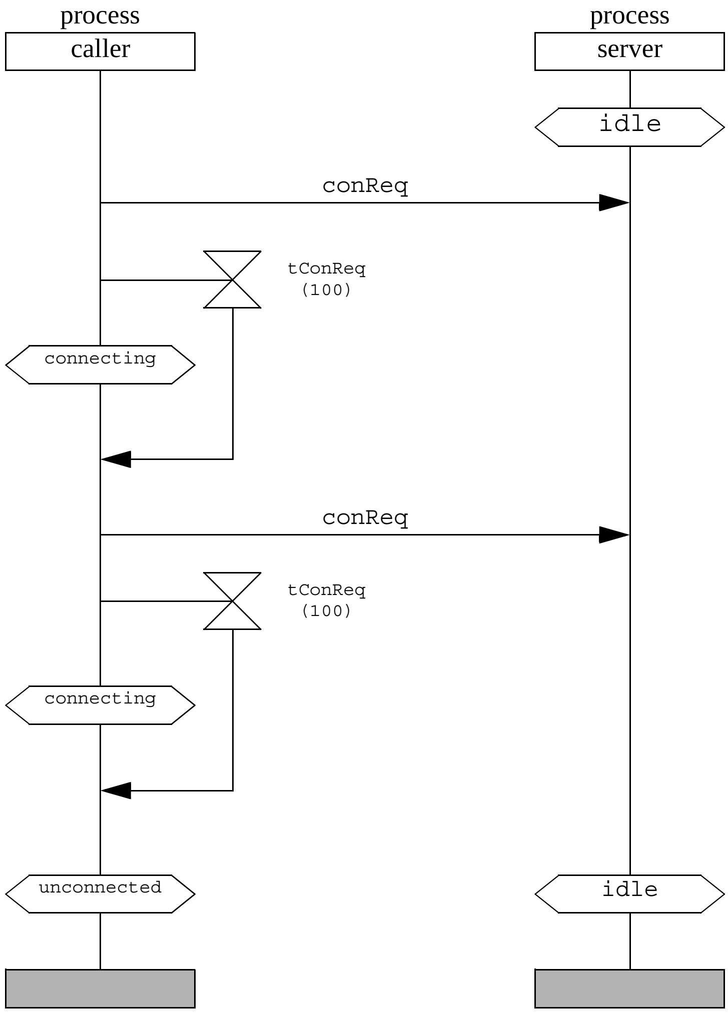


Timer restart connected

Examples:



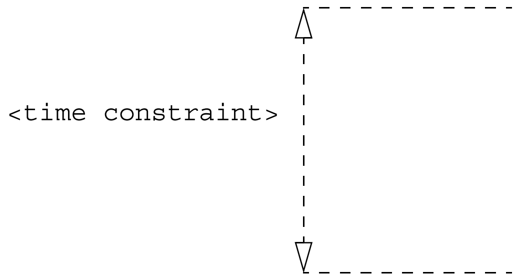
Process `caller` tries to initiate connection with `conReq` message. At the same time it starts timer `tConReq` so that if no answer is received it will retry connecting. If an answer is received the timer is cancelled and process `caller` goes to state `connected`.



Process `caller` tries to initiate connection with `conReq` message. Since it receives no answer after two tries it gives up and goes to unconnected state.

6.8 - Time interval

To specify a time interval between two events the following symbol is used.



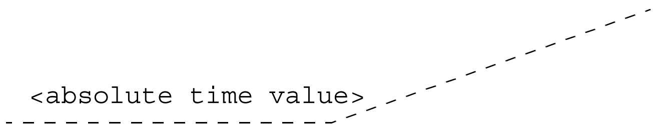
Time constraint syntax is the following:

- absolute time is expressed with an @ up front the time value,

- relative time is expressed with nothing up front its value,
- time interval is expressed between square brackets,
- time unit is RTOS specific -usually tick counts- unless specified (s, ms, µs).

Note it is possible to use time constraint on a single MSC reference.

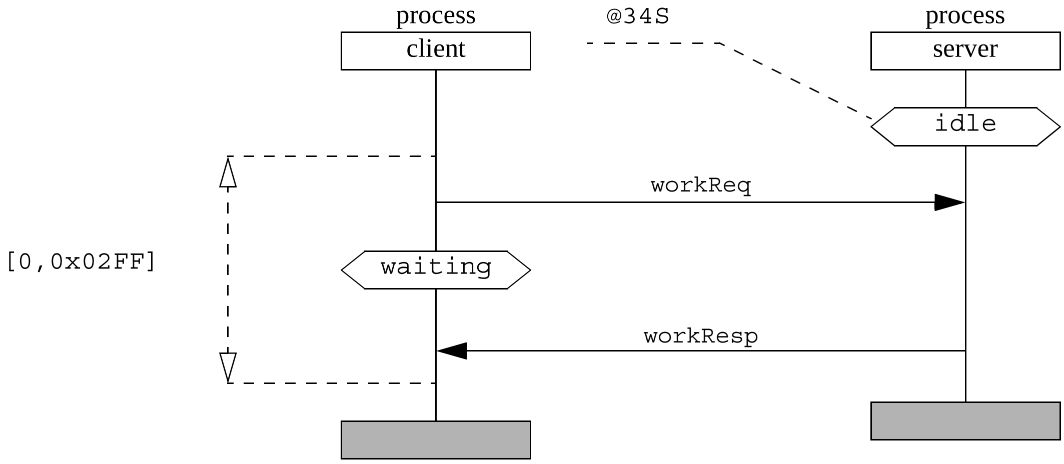
Absolute time can also be specified with the following symbol:



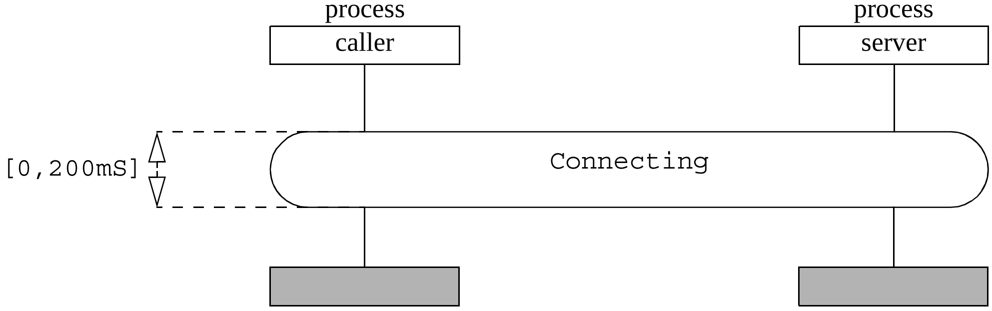
Examples:

Table 1: Examples of time constraint expressions

Expression	Meaning
1.3ms	takes 1.3 ms to do
[1,3]	takes a minimum of 1 to a maximum of 3 time units
@[12.4s,14.7s]	should not occur before absolute time 12.4 s and should not finish after absolute time 14.7 s.
<5	takes less than 5 time units



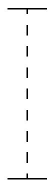
Process server reaches state idle at absolute time 34 Sec.
Process client request process server to compute some work in less than 0x02FF time units.



Connecting MSC should take less than 200mS.

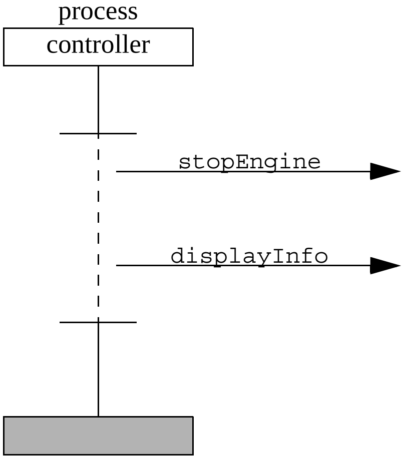
6.9 - Coregion

Coregion is used whenever the sequence of events does not matter. Events in a coregion can happen in any order. The coregion symbol replaces the lifeline instance.



Coregion symbol

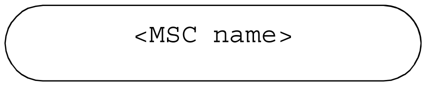
<u>Example:</u>



Process controller
sends stopEngine and displayInfo or
sends displayInfo and stopEngine.

6.10 - MSC reference

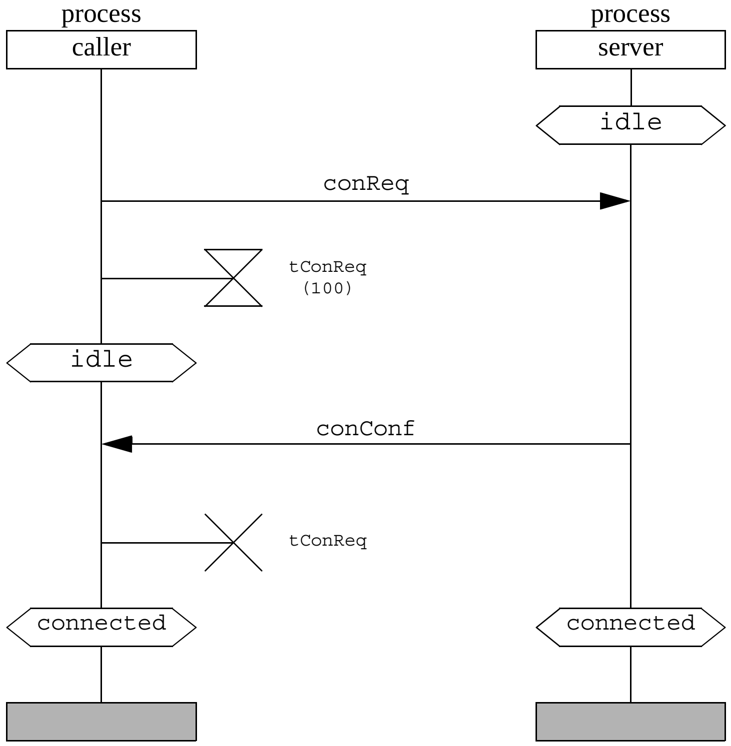
MSC reference allows to refer to another MSC. The resulting MSC is smaller and more legible.



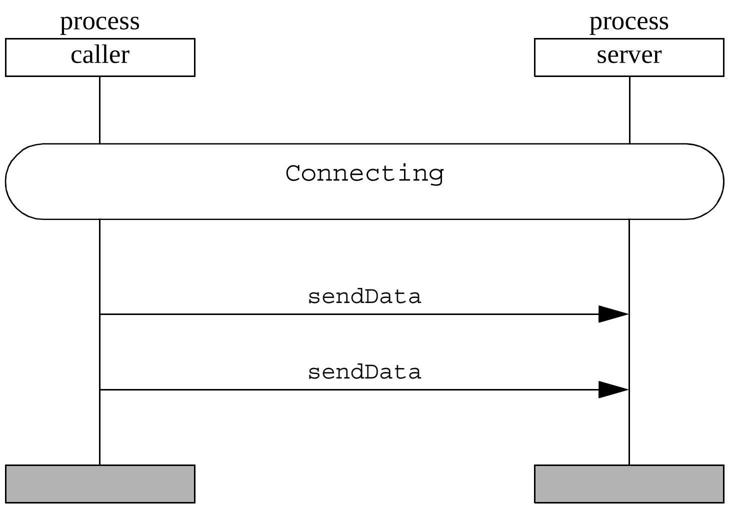
MSC reference symbol

A reference concerns the connected instances. An instance is connected if its lifeline disappears in the symbol. An instance is not connected if it goes over the reference symbol.

Example:



`Connecting` MSC

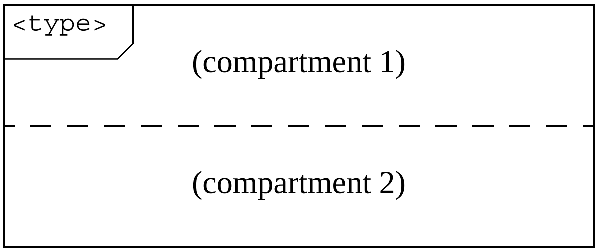


`DataTransfer` MSC

The `DataTransfer` MSC starts with a reference to `Connecting` MSC. That means the scenario described in `Connecting` MSC is to be done before the rest of the `DataTransfer` MSC occur.

6.11 - Inline expressions

Inline expressions allow to add semantics to the MSC diagrams. An inline expression is a box spanning one or more instances, and that can have one or several compartments:



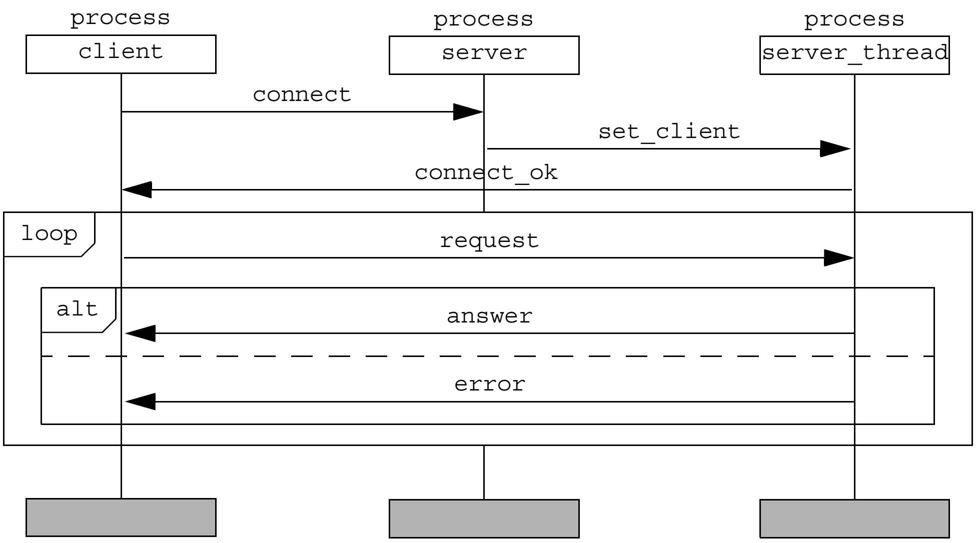
Inline expression symbol

An inline expression can specify:

- That the sequence of events it contains is optional (inline expression type `opt`);
- That its compartments are mutually exclusive: one and exactly one of them must happen, but not several (inline expression type `alt`);
- That the events in all its compartments happen in parallel (inline expression type `par`);
- That the events within it may happen one or several times, without optional minimum and maximum numbers of repeats (inline expression type `loop`);
- That the events within it may happen in any order across the instances, while still being ordered as displayed on a single instance (inline expression type `seq`);
- That the events within it generate an exception, ending the scenario described by the MSC (inline expression type `exc`).

An instance not concerned by an inline expression will appear to go behind it. Inline expression can be nested.

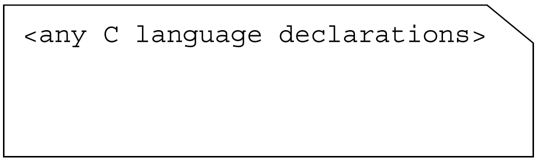
Example:



The `client` process sends a `connect` message to the `server`, which sends the `set_client` message to the `server_thread` process serving the client. The `server_thread` process sends back the connected message to the `client`. The `client` can then send any number of `loop` messages to the `server_thread` (‘`loop`’ inline expression), which will send back either an `answer` message or an `error` message (‘`alt`’ inline expression).

6.12 - Text symbol

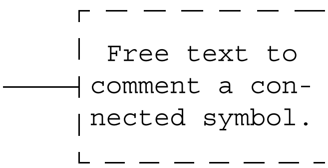
The text symbol contains data or variable declarations if needed in the MSC.



Text symbol

6.13 - Comment

As its name states...



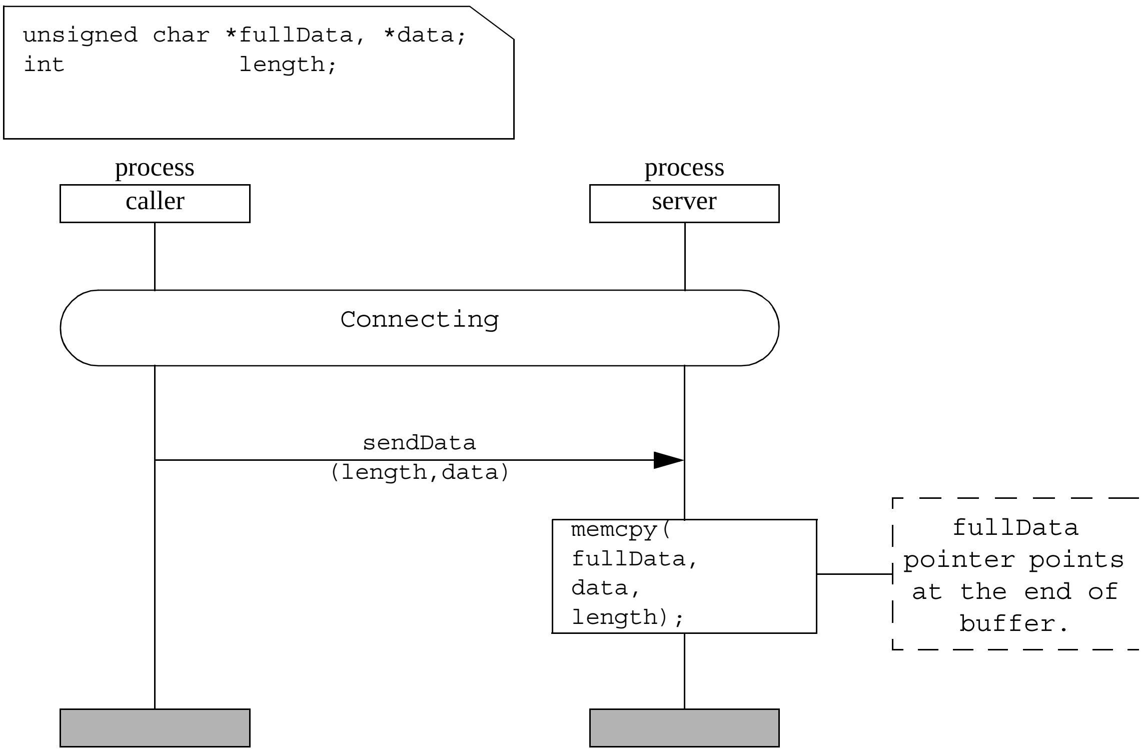
Comment symbol

6.14 - Action

An action symbol contains a set of instructions in C code. The syntax is the one of C language.

Examples:

```
/* Say hi to your friend */
printf("Hello world !\n");
for (i=0;i<MAX;i++)
   {
   newString[i] = oldString[i];
   }
```



DataTransfer MSC

The action symbol contains standard C instructions related to data declarations.

6.15 - Property Sequence Charts (PSC)

SDL-RT integrates the concept of Property Sequence Chart, allowing to create diagrams very similar to MSC diagrams specifying properties that must be matched by an execution trace. PSC diagrams are represented in the same way as MSC diagrams, with a few additional concepts, that are described in the following paragraphs.

6.15.1 Component instance

A component instance in a PSC diagram has the same semantics as an agent instance in a MSC diagram and is represented the same way, as described in “Agent instance” on page 43.

6.15.2 Normal, required and fail messages

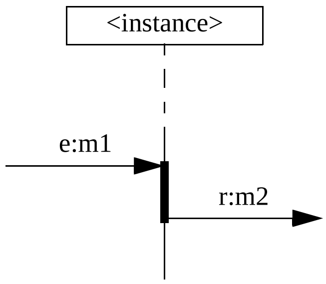
A normal message is a message that is part of the necessary events for the property to be checked. They are part of the ‘cause’ part of the PSC. If any of these are not matched, the property doesn’t apply and shouldn’t be checked. Required and failed messages are part of the ‘consequence’ part of the PSC. The first one is a message that must occur, and the second one is a message that must not occur.
Normal, required and fail messages are represented the same way as messages in a MSC as described in “Message exchange” on page 46. However, the text for the message has an additional prefix, which is ‘`e:`’ for normal messages, ‘`r:`’ for required ones and ‘`f:`’ for fail ones.

6.15.3 Parallel, alternative and loop operator

These operators in PSC diagrams have the same semantics and representation as the corresponding inline expressions in a MSC diagram, as described in “Inline expressions” on page 58.

6.15.4 Strict operator

In a PSC diagram, the ordering of the message sends and receives is by default a loose one: the order shown in the PSC must be the one in the trace, but any event can occur in between in the trace. The strict operator allows to specify that two events must be found one after the other, with no other event happening in between. It is represented with a thick line on the instance on which the events occur:



The receiving of the normal message m1 has to be directly followed by the sending of the required message m2, or the property won’t match. Without the strict operator, m2 could have been sent after any number of events on the instance.

6.15.5 Relative time constraint

Relative time constraints have the same semantics and representation as time intervals in MSC diagrams, as described in “Time interval” on page 53.

6.15.6 Unwanted/wanted message or chain constraints

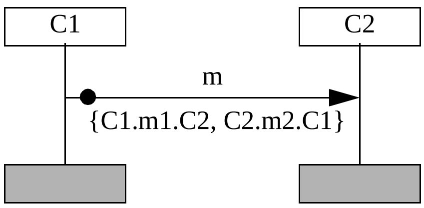
PSC diagrams allow to specify constraints on message, giving a condition for them to be matched. These constraints can be of 3 kinds:

- Unwanted message constraints specify a list of messages that must not occur before or after the message to be matched. If any of the messages in the list happens, the property does not match.
- Unwanted chain constraints specify a sequence of messages that must not occur as a whole in the specified order, before or after the message to be matched. If the whole sequence appears, then the property does not match.
- Wanted chain constraints specify a sequence of messages that must occur as a whole in the specified order, before or after the message to be matched. If the whole sequence does not appears, then the property does not match.

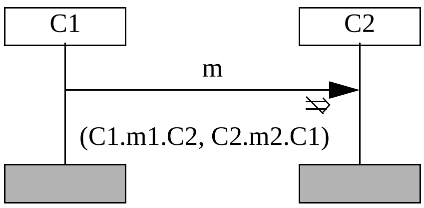
A constraint is specified via a symbol, displayed either near the sending of the message for a constraint to match before the message, or near its receiving for a constraint to be matched after the message. A text describing the messages in the constraint appear under the symbol. Each message in this text is written with a specific syntax:

`<sender instance name>.<message name>.<receiver instance name>`

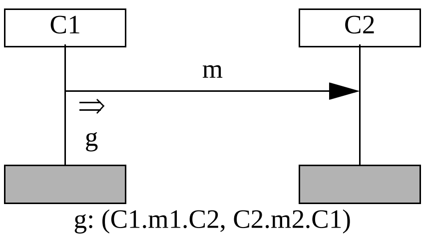
The text can also just be a label, with the actual constraint given in a text box associated to the label:



Unwanted message constraint:
The message m matches only if none of the messages m1 (from C1 to C2) and m2 (from C2 to C1) occur before it, since the constraint is near its sending.

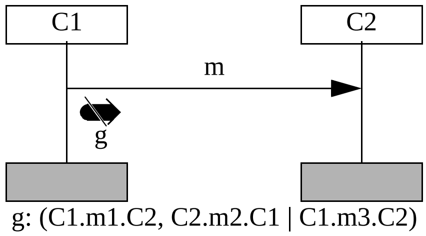


Unwanted chain constraint:
The message m matches only if all the messages m1 (from C1 to C2) and m2 (from C2 to C1) do not occur in this order after it, since the constraint is near its receiving.



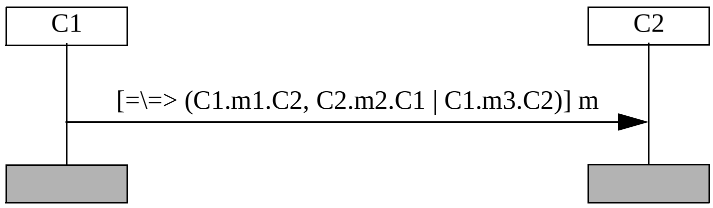
Wanted chain constraint (with a label):
The message m matches only if all the messages m1 (from C1 to C2) and m2 (from C2 to C1) do occur in this order before it, since the constraint is near its receiving.

SDL-RT MSC/PSC diagrams actually introduce a more general unwanted constraint, called an alternative chain constraint, that combines the features of the unwanted message constraint and of the unwanted chain constraint:

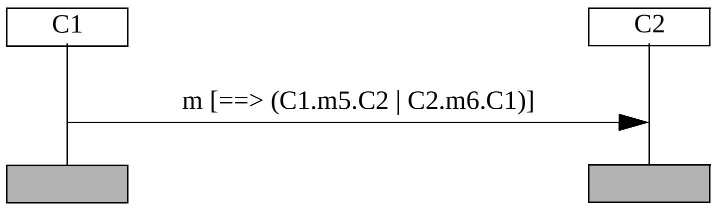


Unwanted alternative chain constraint:
The message m matches only if neither the sequence of messages m1 (from C1 to C2) and m2 (from C2 to C1), not the message m3 (from C1 to C2) occur before it.

Note that the syntax is compatible with unwanted chain constraint: as long a no | character is present in the constraint text, the unwanted alternative chain constraint is the same as the unwanted chain constraint with the same text.
SDL-RT PSC diagrams also allow to specify constraints between square brackets before or after the message text. The symbols are then given in a textual form. For example:

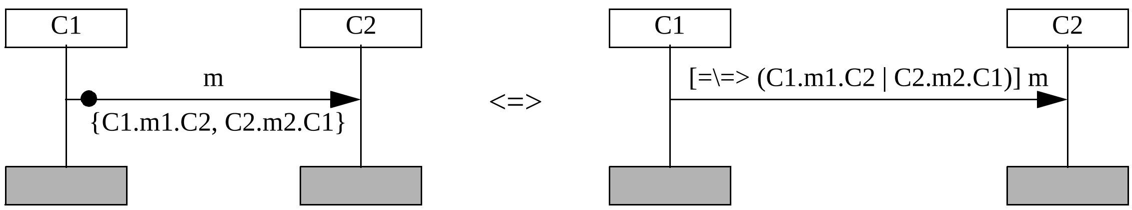


Unwanted alternative chain constraint (before message)



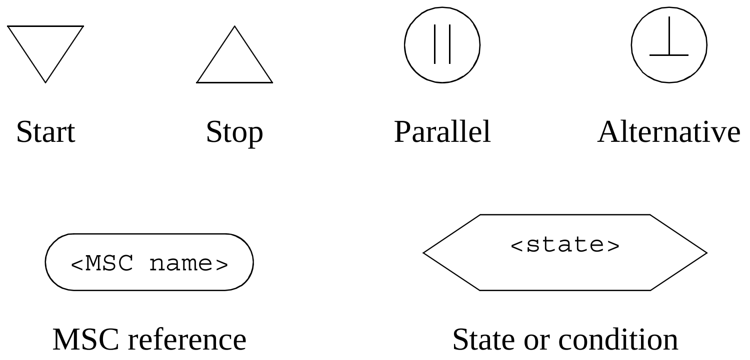
Wanted chain constraint (after message)

Note that unwanted message constraints cannot be represented textually. However, they might be specified with the equivalent unwanted alternative chain constraint:



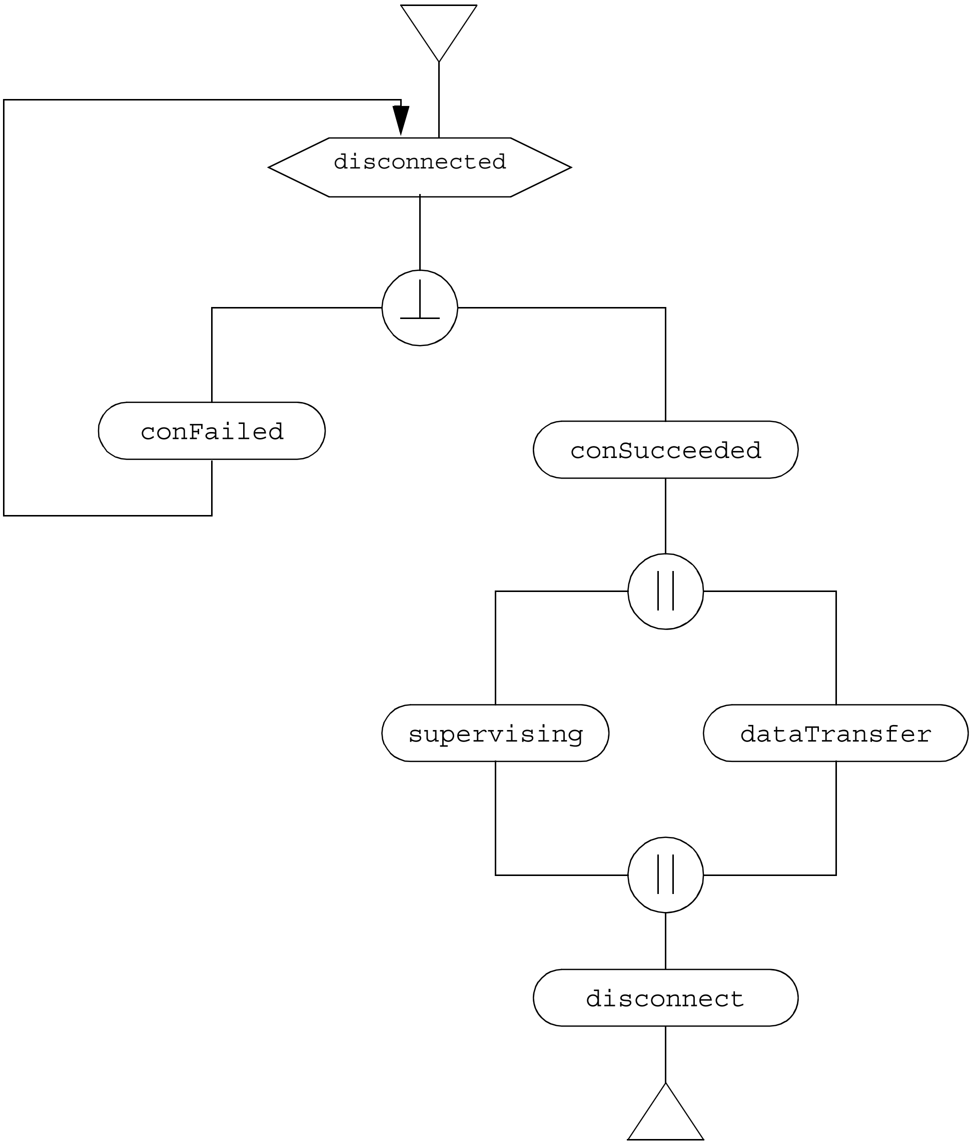
6.16 - High-level MSC (HMSC)

High level MSC diagram is a synthetic view of how MSCs relate to each other. It is only a few symbols: start, stop, alternative, parallel, state or condition, and MSC reference.



The SDL-RT HMSC starts with the start symbol and ends with the stop symbol. The parallel symbol means the following connected path will be executed in parallel. The Alternative symbol means one and only one of the connected path is executed. Whenever two paths meet again the path separator symbol is to be repeated. That means if a parallel symbol is used that creates two different paths, the parallel symbol should be used when the path merge back.
Symbols are connected with lines or arrows if clearer. A symbol is entered by its upper level edge and leaved by any other edge.

Example:



The system starts in `disconnected` state. Connection attempts are made, either the `conFailed` scenario or the `conSucceeded` scenario is executed. If `conSucceeded` is executed `supervising` and `dataTransfer` are executing in parallel. They merge back to `disconnect` and end the HMSC scenario.

7 - Data types

The data types, the syntax and the semantic are the ones of ANSI C and C++ languages. In order to ease readibility in the rest of the document, the expression 'C code' implicitly means 'ANSI C and C++ code'. There is no SDL-RT predefined data types at all but just some keywords that should not be used in the C code. Considering the SDL-RT architecture and concepts surrounding the C code some important aspects need to be described.

7.1 - Type definitions and headers

Types are declared in the text symbol:

```
<Any C type declaration >
```

Types declared in an agent are only visible in the architecture below the agent.

7.2 - Variables

Variables are declared after the type definitions in the same text symbol.

```
<Any C type definition >
<Any C global variable definition >
```

Variables declared in an agent are only visible in the architecture below the agent. For example global variables are to be declared at system level. A variable declared in a block level is not seen by an upper level block. Variables declared in an SDL-RT process in a text symbol are local to the process. They can not be seen or manipulated by any other process.

7.3 - C functions

SDL-RT internal C functions are to be defined through the SDL-RT procedure symbol. An SDL-RT procedure can be defined graphically in SDL-RT or textually in C. When defined in C the procedure call symbol should not be used. A standard C statement in an action symbol should be used.

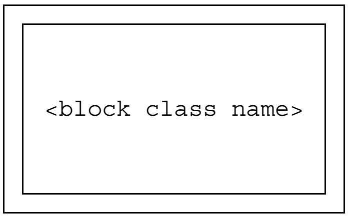
7.4 - External functions

External C functions can be called from the SDL-RT system. These should be prototyped in the system or in an external C header. It is up to an SDL-RT tool to gather the right files when compiling and linking.

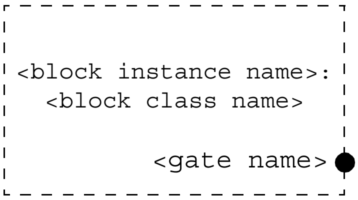
8 - Object orientation

8.1 - Block class

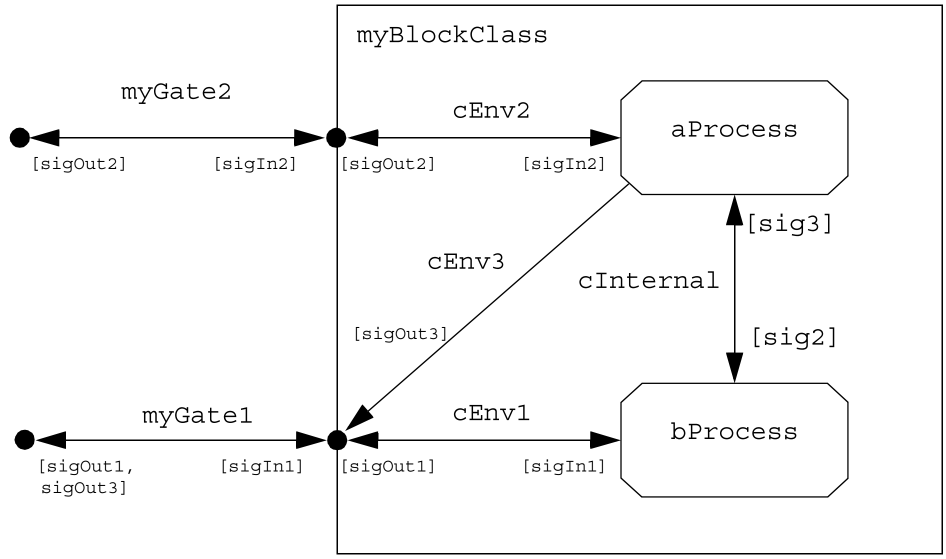
Defining a block class allows to use the same block several times in the SDL-RT system. The SDL-RT block does not support any other object oriented features. A block class symbol is a block symbol with a double frame. It has no channels connected to it.



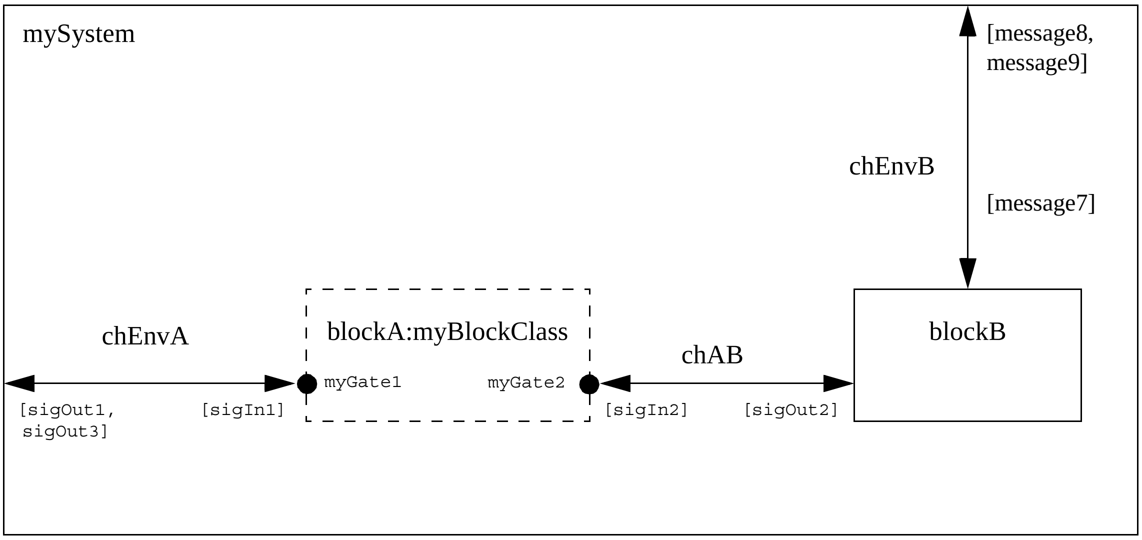
A block class can be instantiated in a block or system. The syntax in the block symbol is:
`<block instance name>:<block class name>`
Messages come in and go out of a block class through gates. In the block class diagram gates are represented out of the block class frame. When a block class is instantiated the gates are connected to the surrounding SDL-RT architecture. The messages listed in the gates are to be consistent with the messages listed in the connected channels.



Example:



Definition diagram of myBlockClass block class



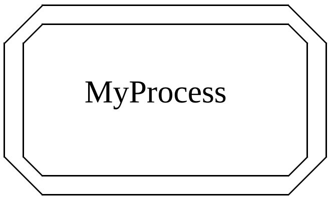
blockA is an instance of myBlockClass

8.2 - Process class

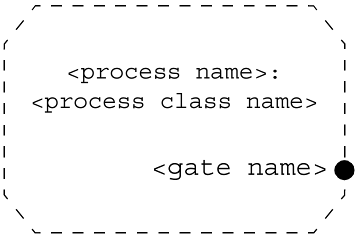
Defining a process class allows to:

- have several instances of the same process in different places of the SDL-RT architecture,
- inherit from a process super-class,
- specialize transitions and states.

A process class symbol is a process symbol with a double frame. It is has no channels connected to it.

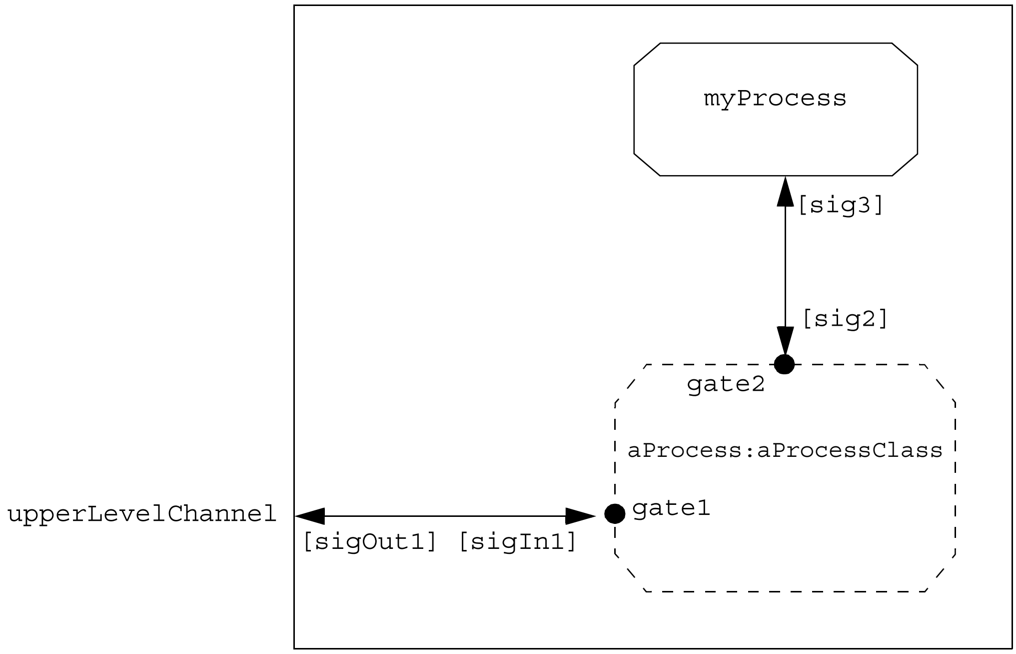


A process class can be instantiated in a block or a system. The syntax in the process symbol is:
`<process instance name>:<process class name>`
Messages come in and go out of a process class through gates. In the process class diagram, gates are represented out of the process class frame. When a process class is instantiated the gates are connected to the surrounding SDL-RT architecture. The messages listed in the gates are to be consistent with the messages listed in the connected channels. The names of the gates appear in the process symbol with a black circle representing the connection point.



Since a class is not supposed to know the surrounding architecture, message outputs should not use the TO_NAME concept. Instead TO_ID, VIA, or TO_ENV should be used.

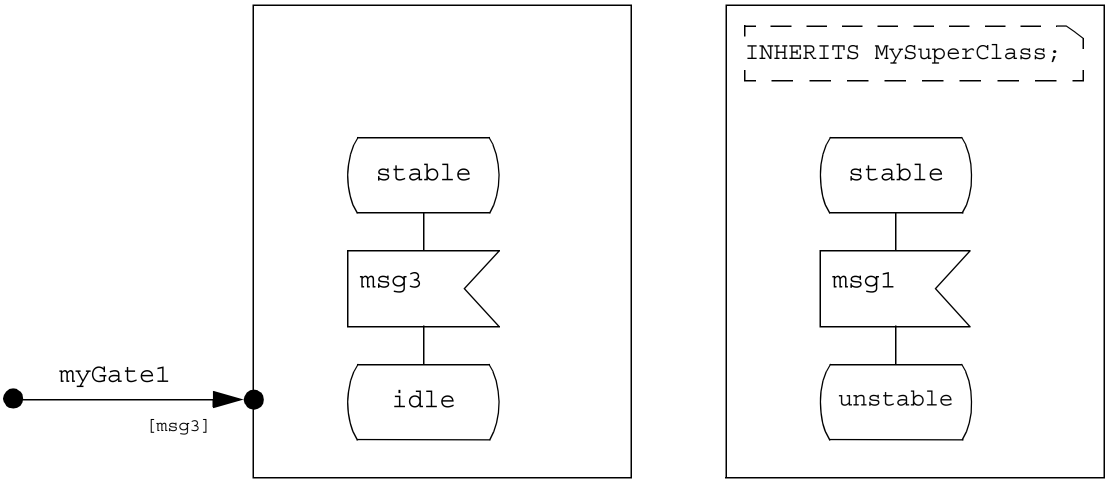
Example:



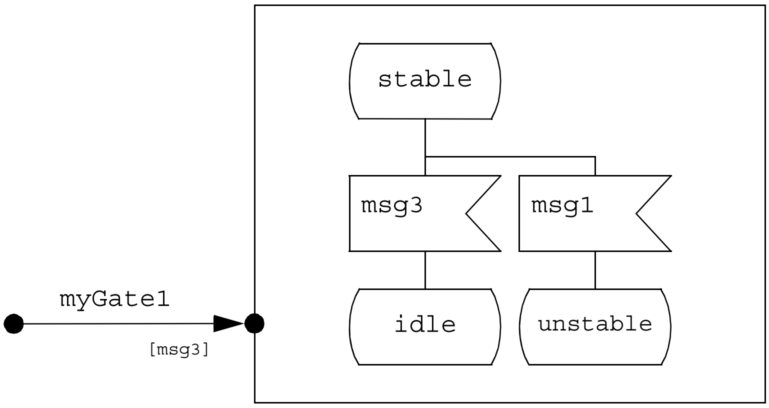
SDL-RT transitions, gates and data are the elements that can be redefined when specializing. In the sub class, the super class to inherit from is defined with the `INHERITS` keyword in an **additional heading symbol**. There are several ways to specialize a process class depending on what is defined in the super class.

8.2.1 Adding a transition

If the transition is new in the sub class, it is simply added to the super class definition.



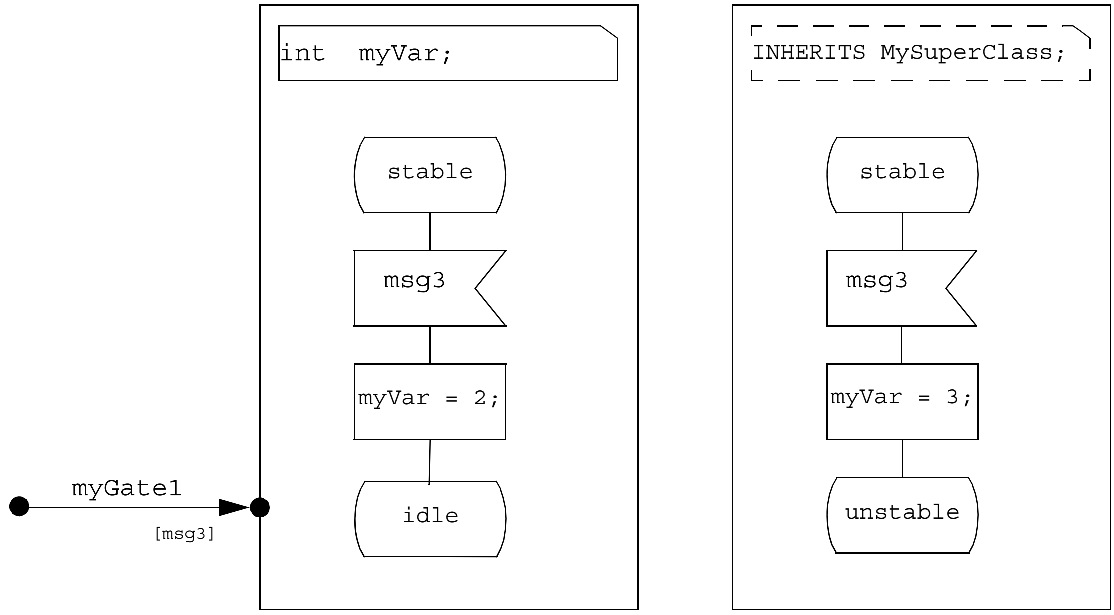
MySuperClass MyClass



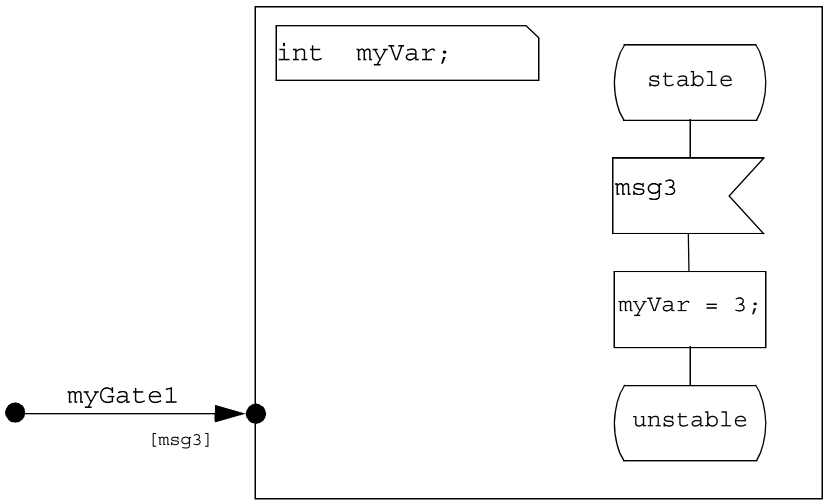
An instance of MyClass

8.2.2 Overload a transition

If the element exists in the super class, the new element definition overwrites the one of the super class,.



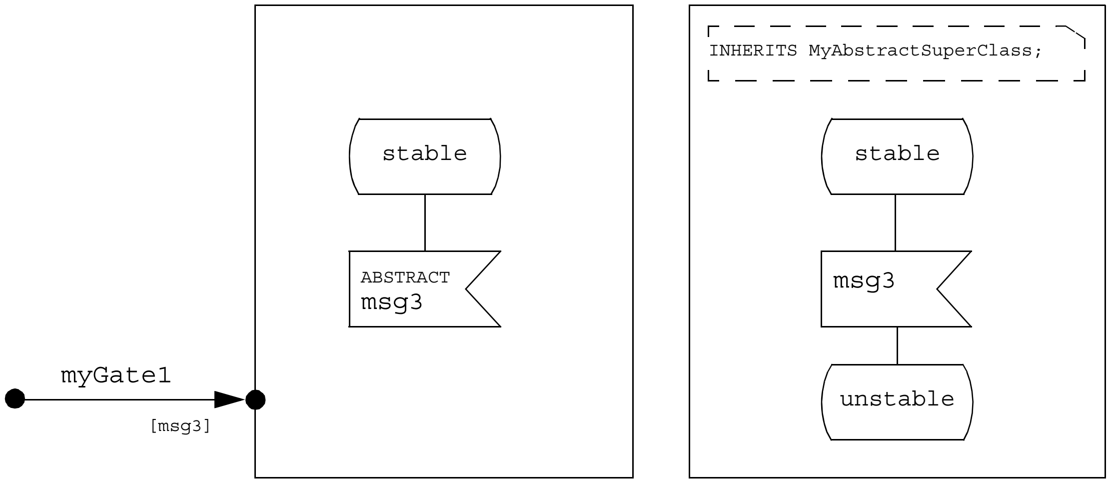
MySuperClass MyClass



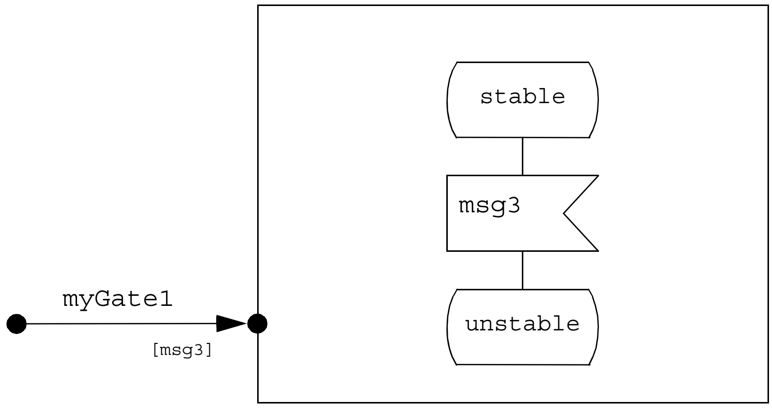
An instance of MyClass

8.2.3 Abstract transition

A class can be defined as abstract with the ABSTRACT keyword. It means the class can not be instantiated as is; it needs to be specialized. A class can define abstract transitions or abstract gates. It means the abstract transition or gate exists but that it is not defined. Such a class is obviously abstract and needs to be defined as such.



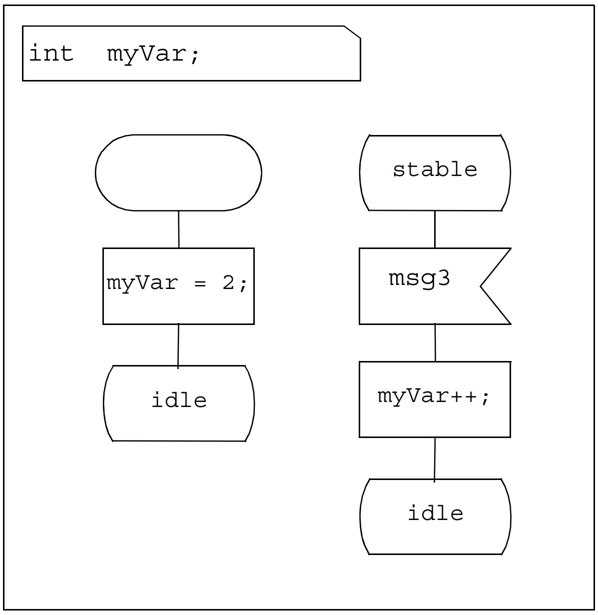
MyAbstractSuperClass MyClass



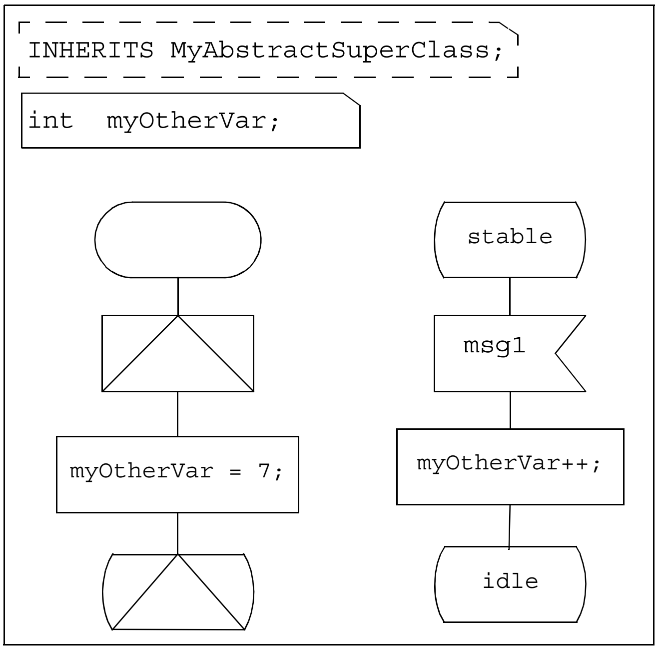
An instance of MyClass

8.2.4 Reference to the super class

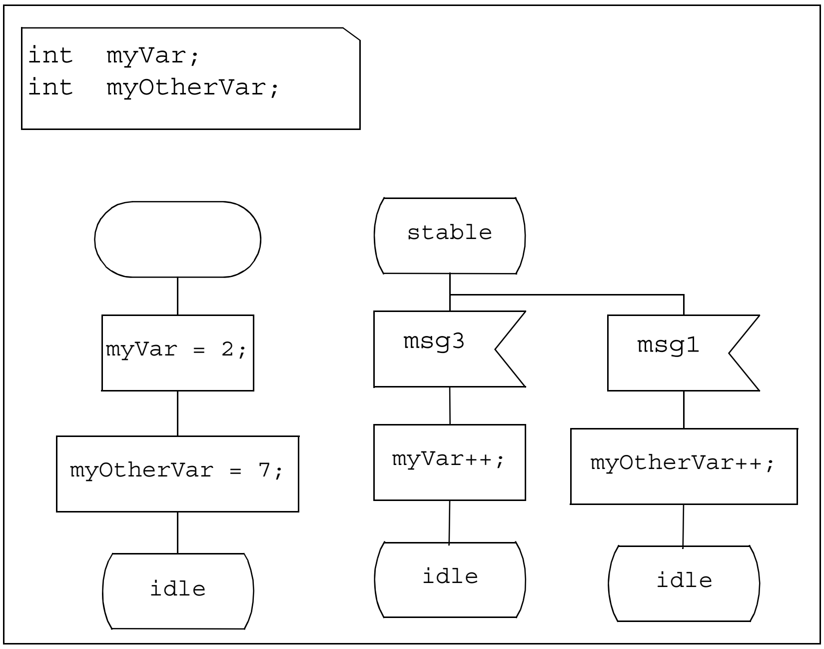
When specializing a class it is important to be able to refer to the super class transition and next state. A typical example is the start transition of a sub class that needs to execute the super class initialization:



MySuperClass

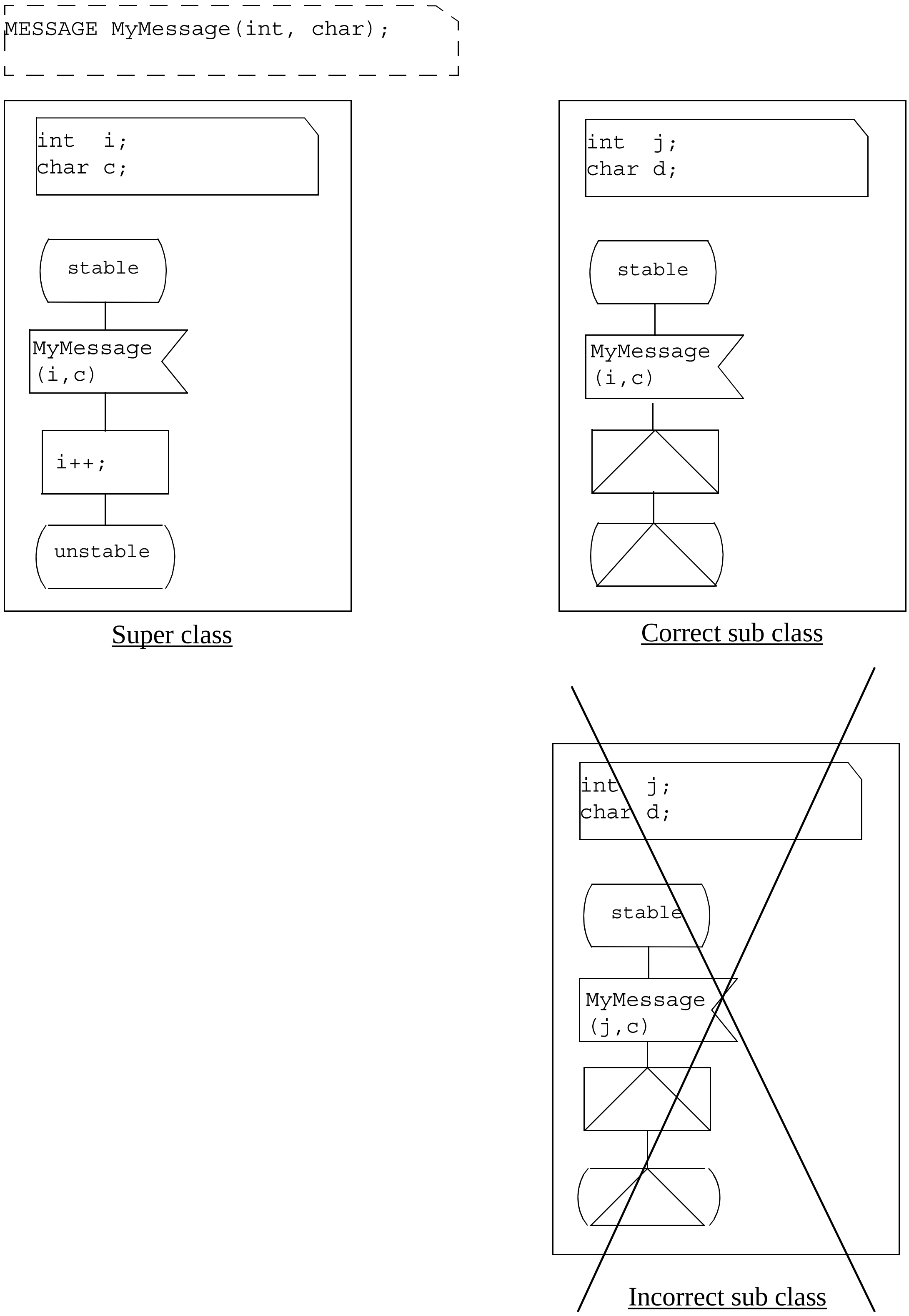


MyClass



An instance of MyClass

Please note the input signature must be the same in the super class and in the sub class including the variable names. For example if the super class transition is MyMsg(a) where a is an int, the sub class transition must be MyMsg(a) as well. It can not be MyMsg(b) even if b is also an int.



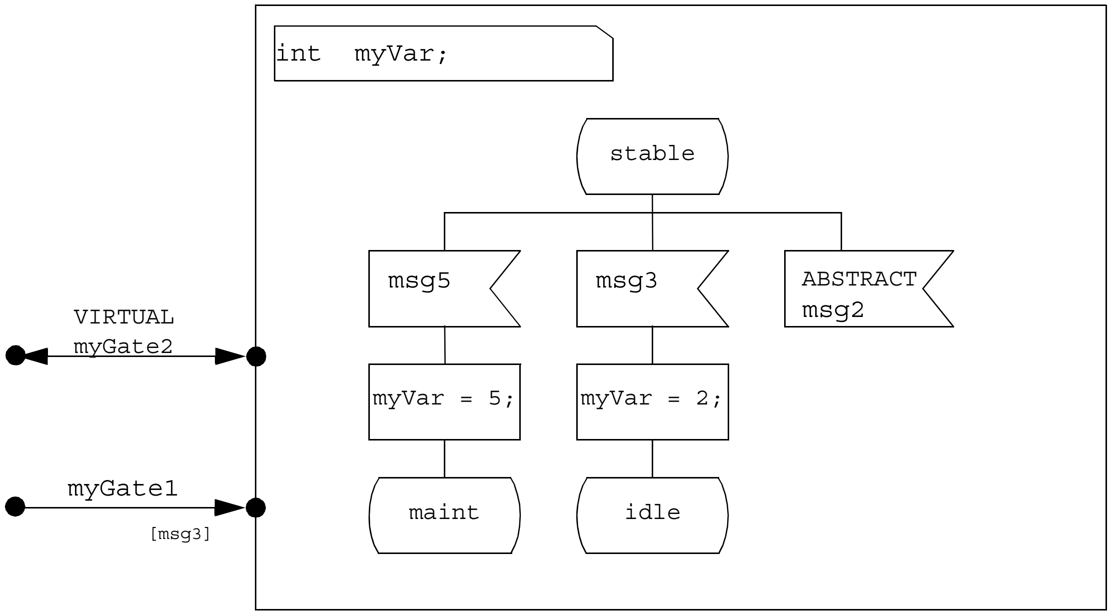
Super class

Correct sub class

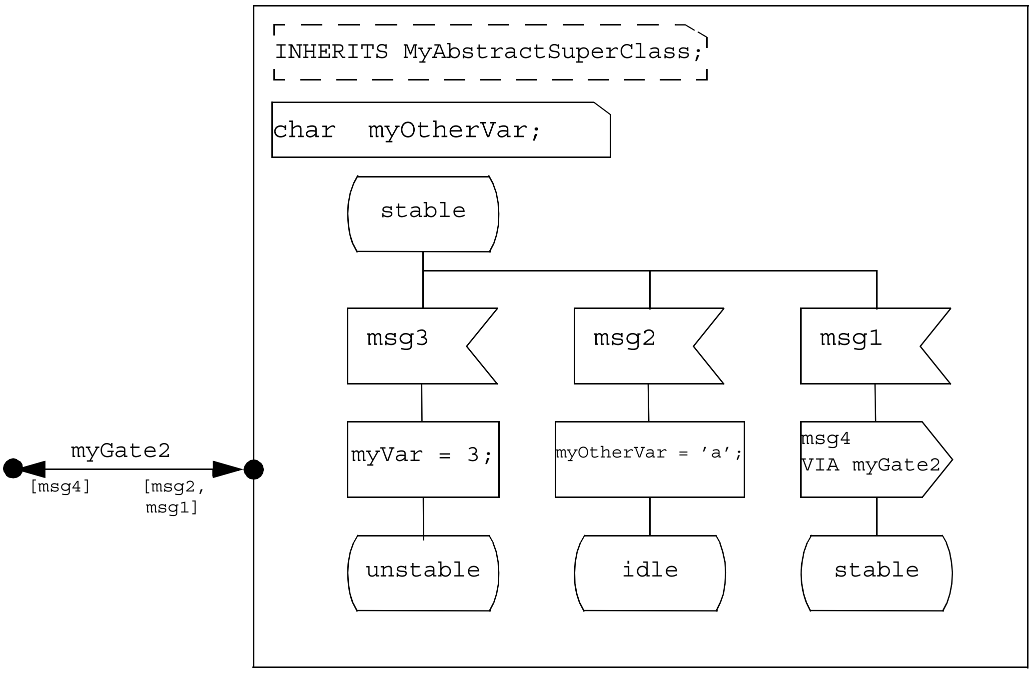
Incorrect sub class
The transition signature is inconsistent

8.2.5 Example

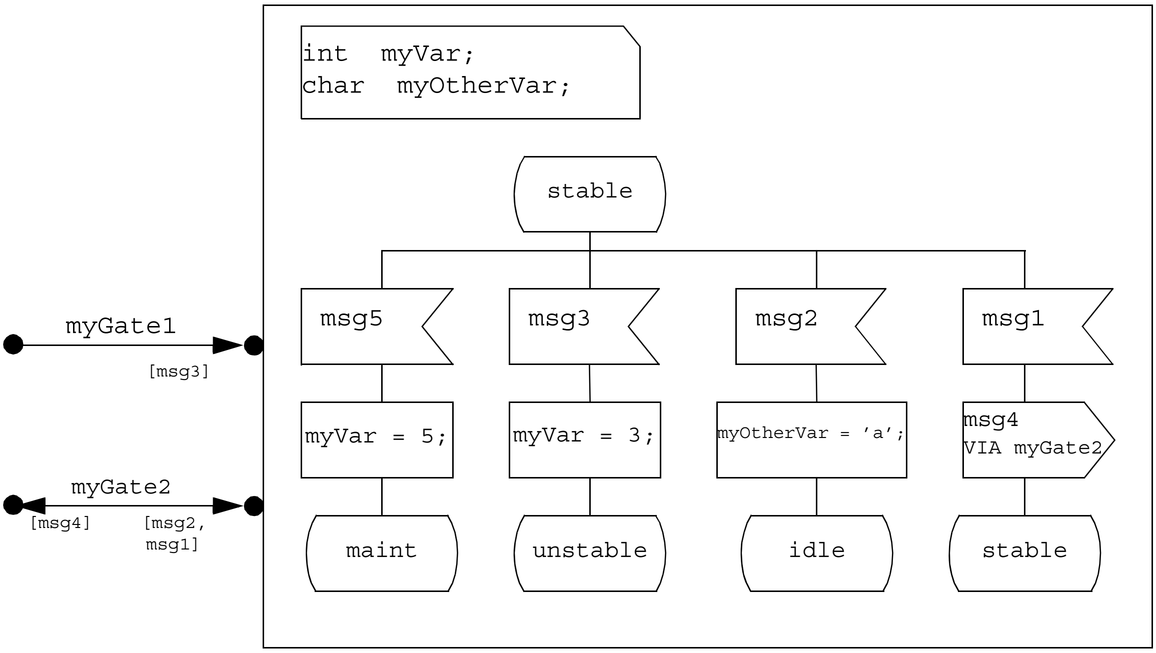
Here comes an example mixing some object oriented concepts and the resulting object:



MyAbstractSuperClass



MyClass



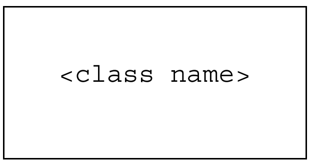
An instance of MyClass

8.3 - Class diagram

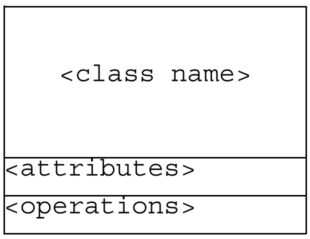
The SDL-RT class diagram is conform to UML 1.3 class diagram. Normalised stereotypes with specific graphical symbols are defined to link with SDL graphical representation. All symbols are briefly explained in the paragraphs below. Detailed information can be found in the OMG UML v1.3 specification.

8.3.1 Class

A **class** is the descriptor for a set of objects with similar structure, behavior, and relationships.

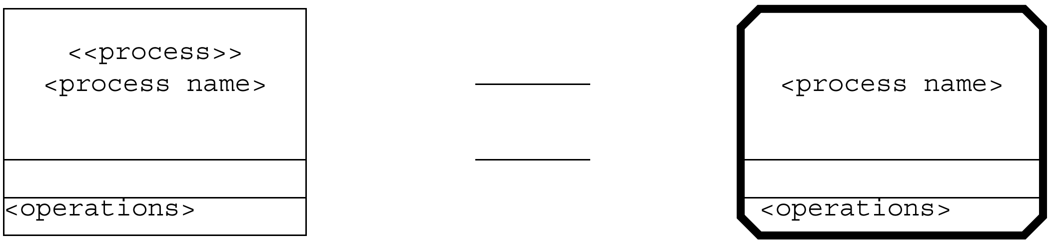


Class symbol with details suppressed



Class symbol full representation

A **stereotype** is an extension of the UML vocabulary allowing to create specific types of classes. If present, the stereotype is placed above the class name within guillemets. Alternatively to this purely textual notation, special symbols may be used in place of the class symbol.



Class stereotyped as a process

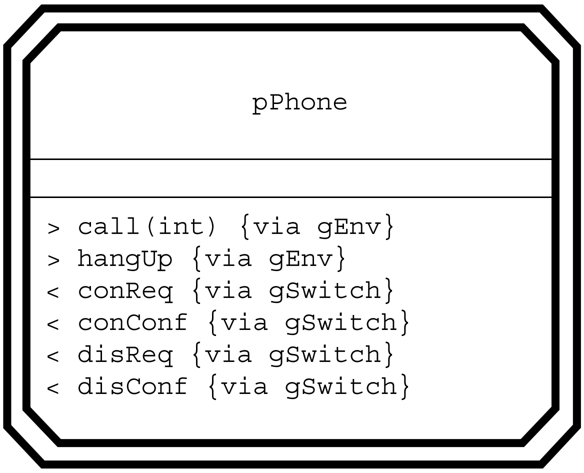
Class stereotyped as a process

Classes are divided in **active classes** and **passive classes**. An instance of an active class owns a thread of control and may initiate control activity. An instance of a passive class holds data, but does not initiate control. In the class diagram, agents are represented by active classes. Agent type is defined by the class stereotype. Known stereotypes are: `system`, `block`, `block class`, `process`, and `process class`. Active classes do not have any attribute. Operations defined for an active class are incoming or outgoing asynchronous messages. The syntax is:

`<message way> <message name> [(<parameter type>)] [{via <gate name>}]`

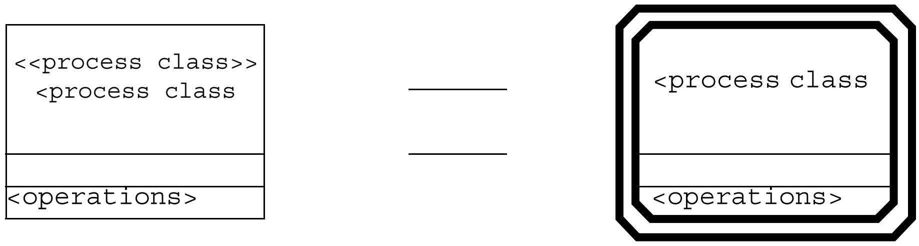
`<message way>` can be one of the characters:

- '`>`' for incoming messages,
- '`<`' for outgoing messages.



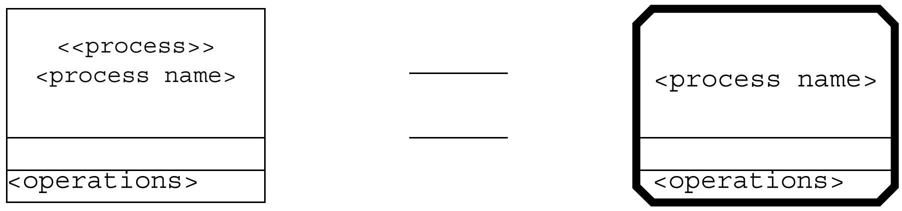
Process class pPhone can receive messages `call` and `hangUp` through gate `gEnv` and send `conReq`, `conConf`, `disReq`, `disConf` through gate `gSwitch`.

Pre-defined graphical symbols for stereotyped classes are described below:



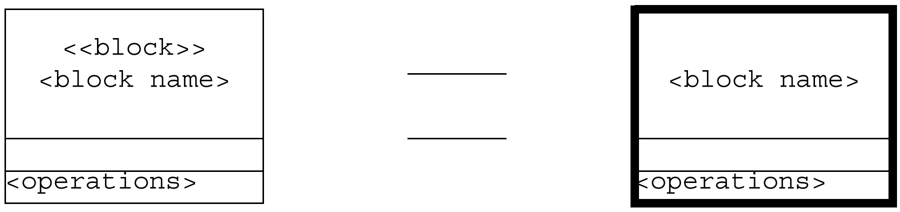
Class stereotyped as a class of process

Class stereotyped as a class of process



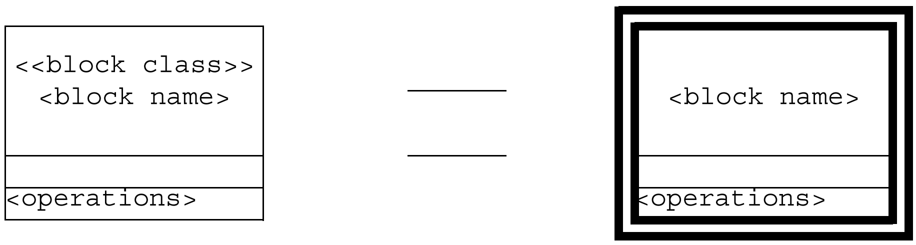
Class stereotyped as a process

Class stereotyped as a process



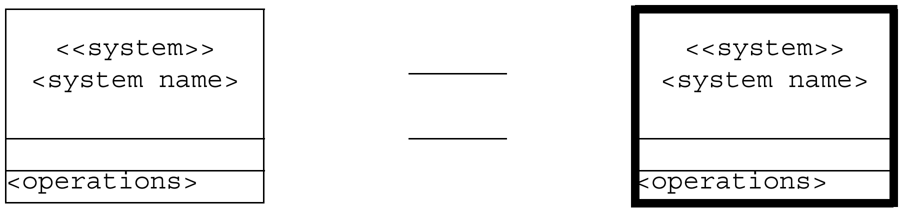
Class stereotyped as a block

Class stereotyped as a block



Class stereotyped as a class of block

Class stereotyped as a class of block

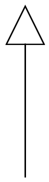


Class stereotyped as a system

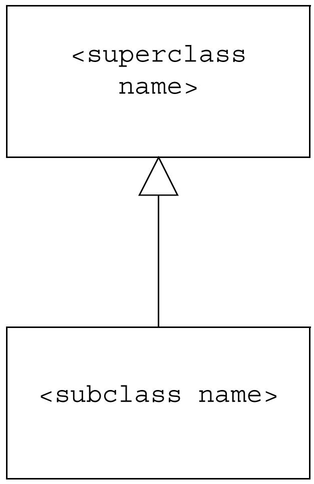
Class stereotyped as a system

8.3.2 Specialisation

Specialisation defines a 'is a kind of' relationship between two classes. The most general class is called the superclass and the specialised class is called the subclass.



Specialisation link

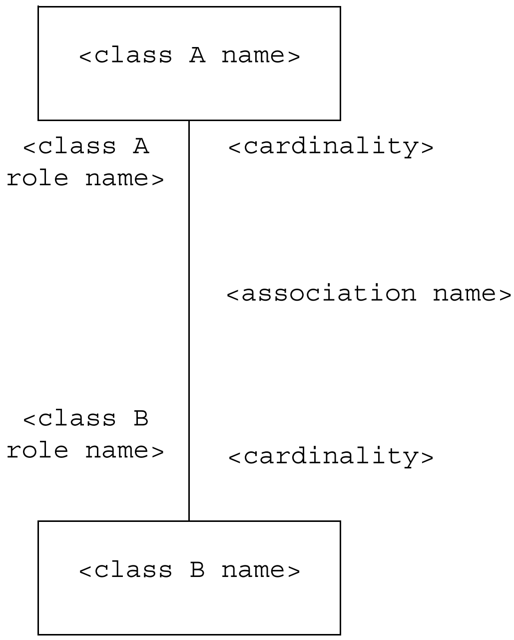


Subclass is a kind of superclass

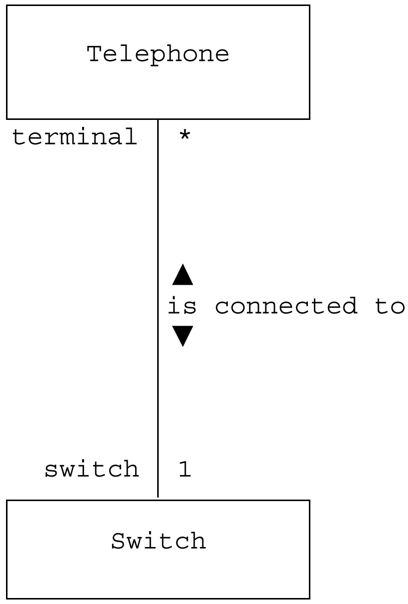
The relationship from the subclass to the superclass is called **generalisation**.

8.3.3 Association

An **association** is a relationship between two classes. It enables objects to communicate with each other. The meaning of an association is defined by its name or the role names of the associated classes. **Cardinality** indicates how many objects are connected at the end of the association.



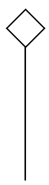
Each Telephone is connected to one Switch. A Switch is connected to several Telephone.
A Telephone is a terminal for a Switch.



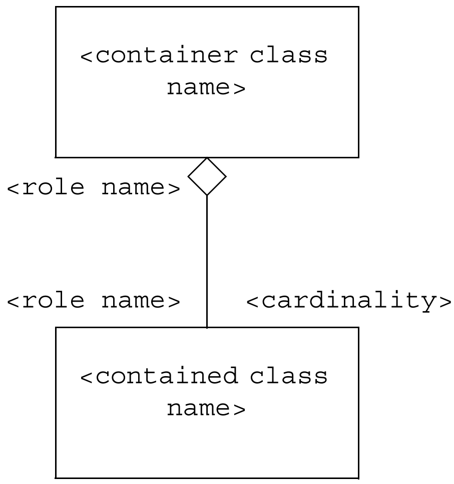
Instances of a class are identified by the associated class via its role name.
In the example above an isntance of Switch identifies the instances of Telephone it is connected to via the name terminal.

8.3.4 Aggregation

Aggregation defines a ’is a part of’ relationship between two classes.



Aggregation link

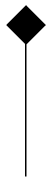


contained class is a part of container class

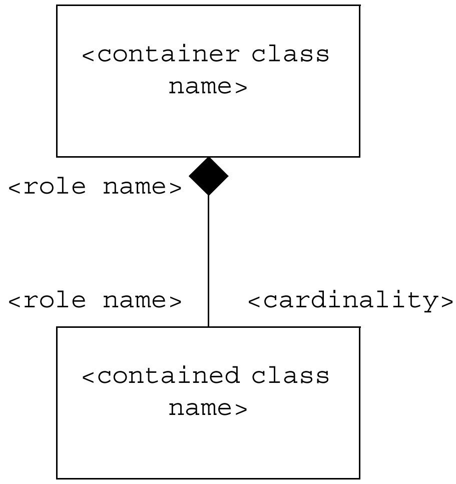
Objects identify each other as described for regular associations (Cf. “Association” on page 81).

8.3.5 Composition

Composition is a strict form of aggregation, in which the parts are existence dependent on the container.



Aggregation link

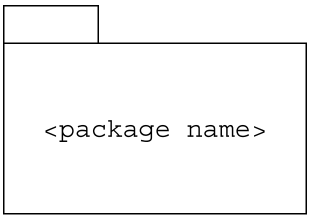


contained class is a part of container class

Objects identify each other as described for regular associations (Cf. “Association” on page 81).

8.4 - Package

A **package** is a separated entity that contains classes, agents or classes of agents. It is referenced by its name.

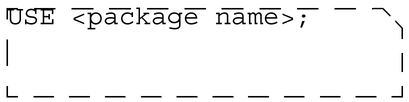


It can contain:

- classes,
- systems,
- blocks,
- classes of blocks,
- processes,
- classes of processes,
- procedures,
- data definitions.

8.4.1 Usage in an agent

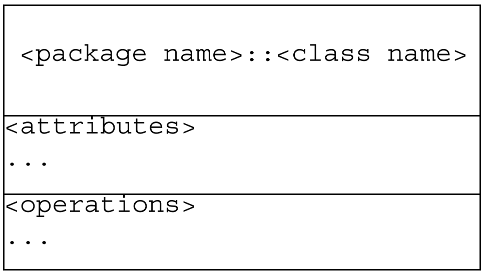
Agent classes definitions can be gathered in a package. To be able to use classes defined in a package, an SDL-RT system should explicitly import the package with USE keyword in an additional heading symbol at system level.



8.4.2 Usage in a class diagram

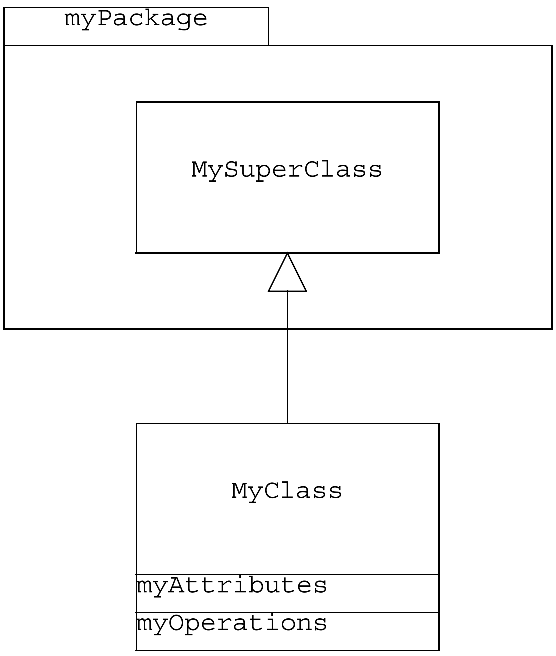
Classes defined in a package can be referenced in 2 ways:

- prefix the class name with the package name



Class `<class name>` is defined in package `<package name>`

- use the package graphical symbol as a container of the class symbol



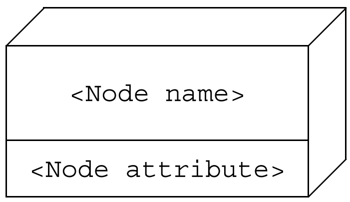
`MyClass` specialises `MySuperClass` defined in `myPackage`.

9 - Deployment diagram

The Deployment diagram shows the physical configuration of run-time processing elements of a distributed system.

9.1 - Node

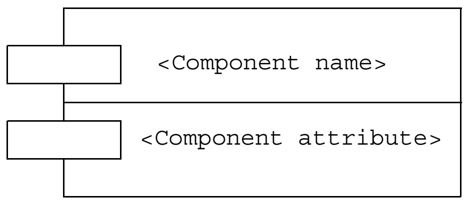
A **node** is a physical object that represents a processing resource.



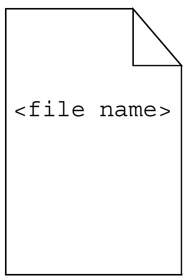
9.2 - Component

A **component** represents a distributable piece of implementation of a system. There are two types of components:

- Executable component

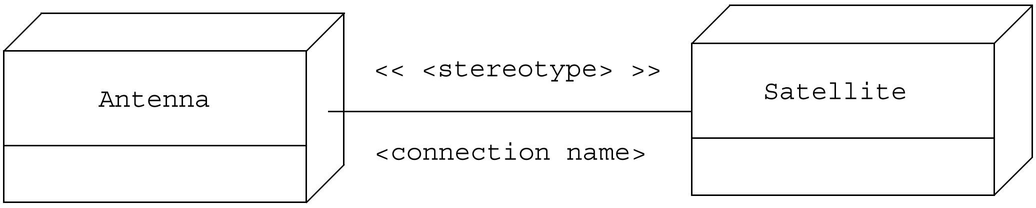


- File component



9.3 - Connection

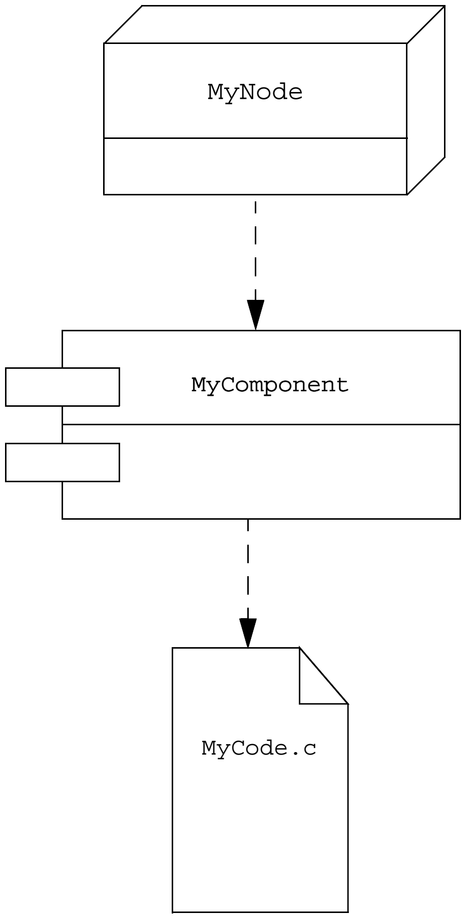
A **connection** is a physical link between two nodes or two executable components. It is defined by its name and stereotype.



9.4 - Dependency

Dependency between elements can be represented graphically.

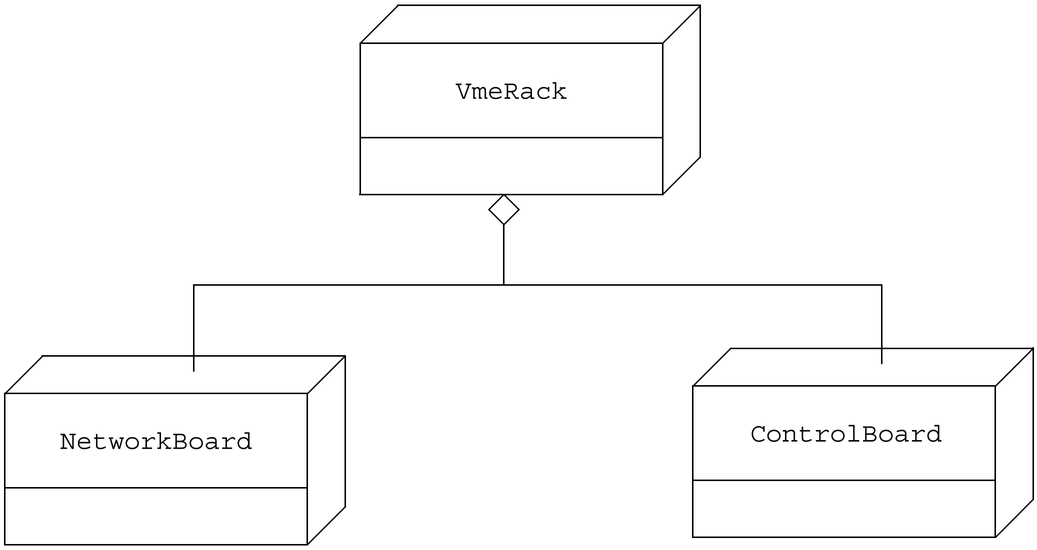
- A dependency from a node to an executable component means the executable is running on the node.
- A dependency from a component to a file component means the component needs the file to be built.
- A dependency from a node to a file means that all the executable components running on the node need the file to be built.



`MyComponent` runs on `MyNode` and needs `MyCode.c` file to be built.

9.5 - Aggregation

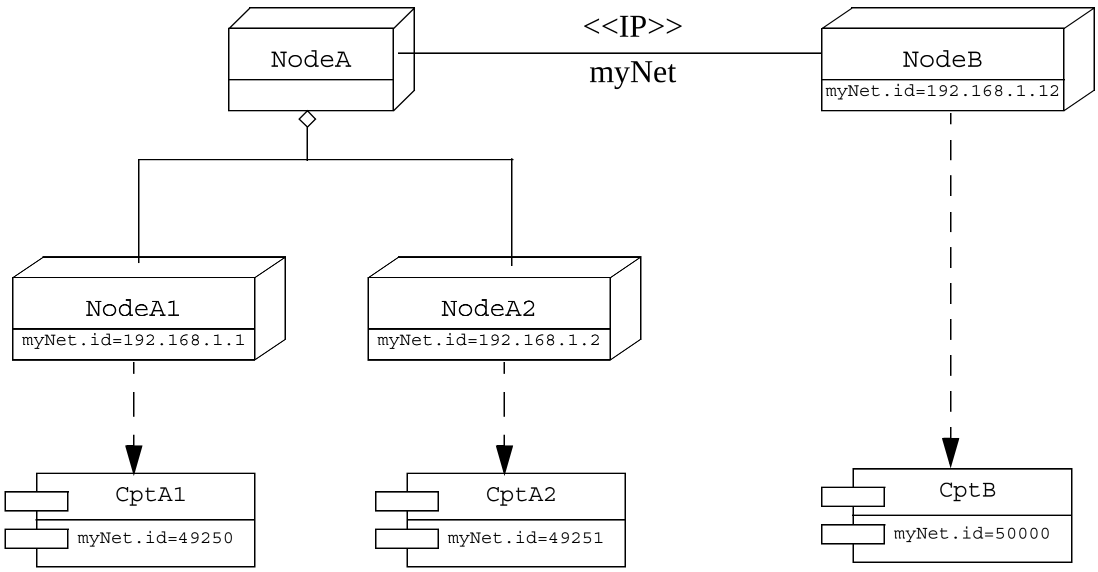
A node can be subdivided of nodes.



VmeRack node is subdivided of NewtorkBoard and ControlBoard

9.6 - Node and components identifiers

Attributes are used by connected nodes or components to identify each other.



CptB can connect to CptA1 via myNet connection by using NodeA1 myNet.id attribute and CptA1 myNet.id attribute.

Nodes' attribute can be omitted if not needed.

10 - Symbols contained in diagrams

The table below shows what symbols can be contained in a specific diagram type.

In the diagrams listed in this column the ticked symbols on the right can be used.	package	block class	process class	block	process	procedure declaration	semaphore declaration	channel	additional heading	text	gate definition	gate usage	behavior symbols	class	association composition	specialisation	node component connection dependency	aggregation
package	x	x	x	x	x	x	x	x	x	x	x	-	-	x	x	x	-	x
class diagram	x	x	x	x	x	-	-	-	-	-	-	-	-	x	x	x	-	x
block class	-	-	-	x	x	x	x	x	x	x	x	x	-	-	-	-	-	-
process class	-	-	-	-	-	-	-	-	x	x	x	-	x	-	-	-	-	-
block	-	-	-	x	x	x	x	x	x	x	-	x	-	-	-	-	-	-
process	-	-	-	-	-	-	-	-	x	x	-	-	x	-	-	-	-	-
procedure	-	-	-	-	-	-	-	-	-	x	-	-	x	-	-	-	-	-
deployment																	x	x

A diagram listed in the first column can contain the ticked symbols in the other columns. For example the process symbol can contain the additional heading symbol, the text symbol and all the behavior symbols. The behavior symbols are all symbols described in “Behavior” on page 13.

11 - Textual representation

The storage format for SDL-RT diagrams follows the XML (eXtensible Markup Language standard from W3C available at http://www.w3.org) standard. There are 3 quite different kind of diagrams that are actually represented differently in the XML:

- “Standard” diagrams, consisting in symbols that can have links between them, with no particular structure associated to them. These diagrams include for example system or block diagrams, deployement diagrams, and so on.
- Behavioral diagrams, which have a specific structure in terms on state blocks and transitions. These diagrams include for example process and procedure diagrams.
- MSC diagrams that have a very specific structure which is very different from all other diagrams.

The following paragraphs describe how each diagram kind is represented, starting with the XML definitions that are common to all diagram kinds.

11.1 - Common XML definitions

```
<!-- Entity for booleans -->
<!-- =================== -->

<!ENTITY % boolean "(TRUE|FALSE)">


<!-- Entity for language -->
<!-- =================== -->

<!ENTITY % Language "(sdl|sdl-rt|none)">


<!-- Entities for symbol types -->
<!-- ========================= -->

<!ENTITY % sdlSymbolTypes1      "sdlSysDgmFrm|sdlSysTypeDgmFrm|sdlBlkDgmFrm|sdlBlkTypeDgmFrm|
        sdlBlkType|sdlBlk|sdlBlkTypeInst|sdlPrcsType|sdlPrcs|sdlPrcsTypeInst">
<!ENTITY % sdlSymbolTypes2      "sdlInherits|sdlPrcsTypeDgmFrm|sdlPrcsDgmFrm|sdlPrcdDgmFrm|
        sdlStart|sdlState|sdlInputSig|sdlSendSig|sdlSaveSig|sdlContSig">
<!ENTITY % sdlSymbolTypes3      "sdlTask|sdlDecision|sdlTransOpt|sdlAnswer|sdlJoin|sdlText|
        sdlComment|sdlTextExt|sdlCnctrOut|sdlCnctrIn|sdlPrcsCreation|sdlStop|sdlObjCre">
<!ENTITY % sdlSymbolTypes4      "sdlInitTimer|sdlResetTimer|sdlSemDecl|sdlSemTake|sdlSemGive|
        sdlPrcdProto|sdlPrcdDecl|sdlPrcdCall|sdlPrcdStart|sdlPrcdReturn">
<!ENTITY % sdlSymbolTypes5      "sdlMacroDecl|sdlMacroCall|sdlMacroDgmFrm|sdlMacroInlet|
        sdlMacroOutlet|sdlPrioInputSig|sdlPrioSendSig|sdlCompState">
<!ENTITY % sdlSymbolTypes6      "sdlCompStateDef|sdlService|sdlCompStateDgmFrm|sdlServDgmFrm|
        sdlSuperTransitionCall|sdlSuperNextstateCall">
<!ENTITY % sdlSymbolTypes       "%sdlSymbolTypes1;|%sdlSymbolTypes2;|%sdlSymbolTypes3;|
        %sdlSymbolTypes4;|%sdlSymbolTypes5;|%sdlSymbolTypes6;">

<!ENTITY % hmscSymbolTypes      "hmscDgmFrm|hmscParallel|hmscStart|hmscEnd|hmscCondition|
        hmscMscRef|hmscAlternativePoint">

<!ENTITY % umlClassSymbolTypes  "umlClassDgmFrm|umlPckg|umlClass|umlComment|umlSys|umlBlkCls|
        umlBlk|umlPrcsCls|umlPrcs">
<!ENTITY % umlDeplSymbolTypes   "umlDeplDgmFrm|umlNode|umlComp|umlFile">
<!ENTITY % umlUCSymbolTypes     "umlUCDgmFrm|umlUseCase|umlActor">
```

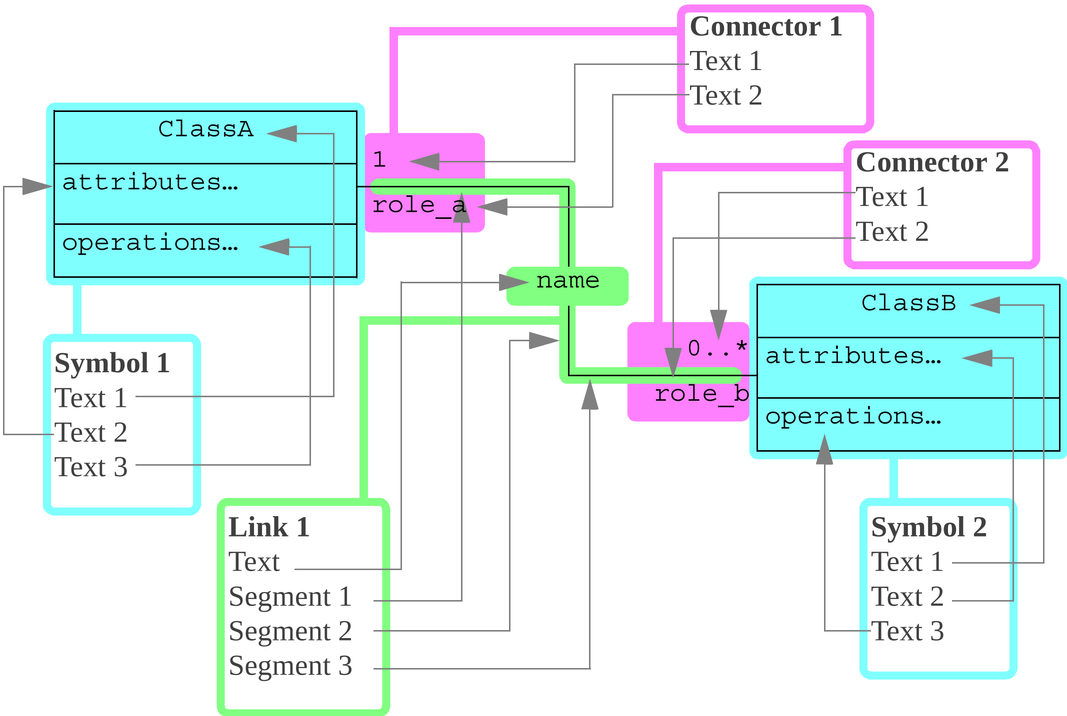
```
<!ENTITY % SymbolType "(%sdlSymbolTypes;|%hmscSymbolTypes;|%umlClassSymbolTypes;|
         %umlDeplSymbolTypes;|%umlUCSymbolTypes;)">


<!-- Entity for connector types -->
<!-- ========================== -->

<!ENTITY % ConnectorType "(void|chnl|chnlgate|sdlarrow|mscvoid|mscgate|mscarrowgate|hmscarrow|
         umlcvoid|umlassoc|umlrole|umldvoid)">


<!-- Entity for side for connectors -->
<!-- ============================== -->

<!ENTITY % Side "(n|s|w|e|x|y)">


<!-- Entity for end types for connectors -->
<!-- =================================== -->

<!ENTITY % ConnectorEndType "(voidend|arrow|midarrow|fullarrow|outltri|outldiam|filldiam|
         outldiamarw|filldiamarw)">


<!-- Entity for link segment orientation -->
<!-- =================================== -->

<!ENTITY % Orientation "(h|v)">


<!-- Entity for link types -->
<!-- ===================== -->

<!ENTITY % LinkType "(sbvoid|dbvoid|ssvoid|dsvoid|chnl|dec|transopt|msg|meth|rtn|instcre|assoc|
         spec|aggr|comp|cnx|dep)">


<!-- Entity for diagram types -->
<!-- ======================== -->

<!ENTITY % DiagramType "(sys|systype|blk|blktype|prcs|prcstype|prcd|macro|msc|hmsc|mscdoc|class|
         usec|depl|compstate|service|ifobs)">


<!-- Entity for message parameters visibility -->
<!-- ======================================== -->

<!ENTITY % MsgParamVis "(no|abbr|full)">
```

11.2 - XML elements for standard diagrams

11.2.1 Principles

The file describes the whole diagram in terms of symbols, links and connectors. A link is subdivided into link segments. Each of these elements have one or several texts.

Here is an example of this organization with a UML class diagram:



11.2.2 XML elements

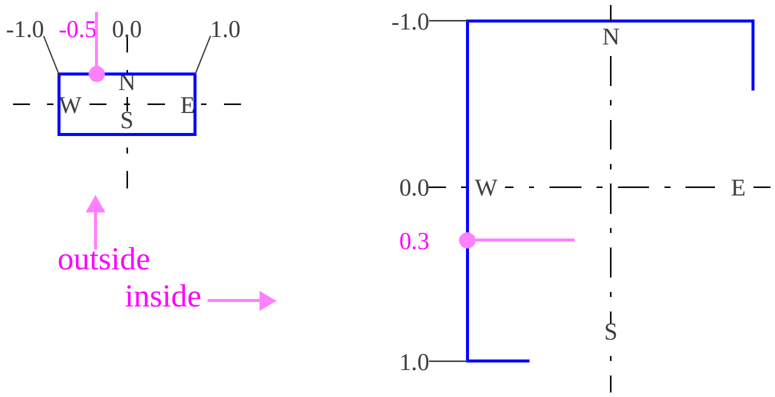
```
<!-- Element for text in symbols/links/... -->
<!-- ==================================== -->
<!-- The "hidden" attribute is only used for attributes and operation boxes in UML class diagrams
         -->

<!ELEMENT Text (#PCDATA)>
<!ATTLIST Text
  id       CDATA     "0"
  hidden   %boolean; "FALSE"
>


<!-- Element for symbols -->
<!-- =================== -->
<!-- The "Symbol*" component is for contained symbols -->

<!ELEMENT Symbol (Text+, Symbol*)>
<!ATTLIST Symbol
  symbolId        ID             #REQUIRED
  type            %SymbolType;   #REQUIRED
  xCenter         CDATA          #REQUIRED
  yCenter         CDATA          #REQUIRED
  fixedDimensions %boolean;      "FALSE"
  width           CDATA          "10"
  height          CDATA          "10"
  color           CDATA          "#000000"
  fillColor       CDATA          "#ffffff"
>
```

```
<!-- Element for connectors -->
<!-- ===================== -->

<!ELEMENT Connector (Text, Text)>
<!ATTLIST Connector
  connectorId       CDATA               ""
  attachedSymbolId  IDREF               #REQUIRED
  type              %ConnectorType;     #REQUIRED
  isOutside         %boolean;           #REQUIRED
  side              %Side;              #REQUIRED
  position          CDATA               #REQUIRED
  endType           %ConnectorEndType;  #REQUIRED
>


<!-- Element for link segments -->
<!-- ======================== -->

<!ELEMENT LinkSegment EMPTY>
<!ATTLIST LinkSegment
  orientation %Orientation; #REQUIRED
  length      CDATA         #REQUIRED
>


<!-- Element for links -->
<!-- ================= -->

<!ELEMENT Link (Text, Connector, Connector, LinkSegment*)>
<!ATTLIST Link
  linkId          CDATA       ""
  type            %LinkType;  #REQUIRED
  textSegmentNum  CDATA       #REQUIRED
  color           CDATA       "#000000"
  reverseRead     %boolean;   "FALSE"
>


<!-- Element PageSpecification -->
<!-- ======================== -->
<!-- Attributes for diagram pages; all dimensions are centimetres -->
<!ELEMENT PageSpecification EMPTY>
<!ATTLIST PageSpecification
  pageWidth     CDATA       "21"
  pageHeight    CDATA       "29.7"
  topMargin     CDATA       "1.5"
  bottomMargin  CDATA       "1.5"
  leftMargin    CDATA       "1.5"
  rightMargin   CDATA       "1.5"
  pageFooter    %boolean;   "TRUE"
  scaleFactor   CDATA       "1.0"
>


<!-- Element DiagramPartition -->
<!-- ======================= -->
<!-- A partition in a diagram -->
<!ELEMENT DiagramPartition (PageSpecification, Symbol, Link*, UnifiedPublication*)>
<!ATTLIST DiagramPartition
  name      CDATA   ""
  nbPagesH  CDATA   "1"
  nbPagesV  CDATA   "1"
>
```

```
<!-- Element for diagrams -->
<!-- ==================== -->

<!ELEMENT Diagram (DiagramPartition+)>
<!-- Warning: attributes pageWidth, pageHeight, nbPagesH & nbPagesV are obsolete -->
<!ATTLIST Diagram
  language                %Language;     "none"
  type                    %DiagramType;  #REQUIRED
  name                    CDATA          ""
  cellWidthMm             CDATA          "5"
>
```

11.2.3 Explanations

The main XML element is `<Diagram>`. It is split in several partitions, each being a rectangular set of pages having the same page size and margins within a partition (XML element `<PageSpecification>`). A partition always has an external frame, which is a symbol, containing all the symbols in it. Attributes for a symbol are all the required information to display it. In addition to its display attributes, a link also knows the index of the segment bearing its text (`textSegmentNum`) and a special indicator for association links giving the reading order of the association (`reverseRead`). The position for connectors is given by the side of the symbol to which they attach, the relative position along the symbol's border and a boolean saying if the connector is outside the symbol (normal case) or inside it (links connecting to a frame border):



A connector also has an `endType` specifying the symbol that appears at the link end, which may vary for a given connector type (e.g connectors for association links may end with a filled or outlined diamond to indication the association is actually a composition or aggregation).

11.3 - Behavioral diagrams

11.3.1 Principles

Behavioral diagrams are not described in terms of symbols with links between them, but in terms of logical elements:

- State blocks, which are blocks of symbols starting with a state symbol with transitions attached to it, each starting either with a message input symbol, a continuous signal symbol or a save symbol.

- “Standalone” transitions, such as the start transition, or a sequence of symbols starting with a label. These are also called “non-state transitions”, as they do not appear in a state block.
- Symbols alone that are not attached to any other, such as declaration symbols.

Some specific constructions are used for state blocks starting on a state symbol that is also a next state in another transition, and for symbols that are attached to other ones, such as comment symbols.

11.3.2 XML elements

```
<!-- =================================== -->
<!-- ELEMENTS FOR STATE MACHINE DIAGRAMS -->
<!-- =================================== -->


<!-- Element SdlFSMDiagram -->
<!-- ~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for the whole state machine diagram. The symbols are the declaration symbols in the
        diagram. -->

<!ELEMENT SdlFSMDiagram (%SdlFSMPartition*)>
<!-- NB: entities and elements not defined in this file are in rtdsDiagram.dtd, where this DTD is
        included -->
<!ATTLIST SdlFSMDiagram
  language      %Language;    "-"
  type          %DiagramType; "prcs"
>


<!-- Element SdlFSMPartition -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a partition in the state machine diagram. The symbols are the declaration symbols
        in the partition. -->

<!ELEMENT SdlFSMPartition (SdlFSMSymbol*, (NonStateTransition | StateTransitionBlock)*,
        PageSpecification?)>
<!ATTLIST SdlFSMPartition
  name          CDATA       ""
>


<!-- Element StateTransitionBlock -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a set of transitions attached to a state symbol. The symbol is the state. -->

<!ELEMENT StateTransitionBlock ((SdlFSMSymbol|SdlFSMSymbolReference), (SaveTransition|
        NormalStateTransition)*)>
<!ATTLIST StateTransitionBlock
  top_left_coordinates   CDATA    ""
>


<!-- Element SdlFSMSymbolReference -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a reference on a symbol. Used to connect state transition blocks to nextstate
        symbols -->

<!ELEMENT SdlFSMSymbolReference EMPTY>
<!ATTLIST SdlFSMSymbolReference
  identifier   IDREF #REQUIRED
```

```
>


<!-- Element NonStateTransition -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a transition not attached to a state symbol, such as a start transition or an "in"
        connector transition. The symbol is the starting one (start or connector). -->

<!ELEMENT NonStateTransition (SdlFSMSymbol, Transition)>
<!ATTLIST StateTransitionBlock
  top_left_coordinates  CDATA   ""
>


<!-- Element NormalStateTransition -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a "normal" transition in a state block, i.e starting with an input or a continuous
        signal. The symbol is the input or continuous signal. -->

<!ELEMENT NormalStateTransition (SdlFSMSymbol, Transition)>


<!-- Element SaveTransition -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a transition in a state block that is only a save symbol. The symbol is the save.
        -->

<!ELEMENT SaveTransition (SdlFSMSymbol)>


<!-- Element Transition -->
<!-- ~~~~~~~~~~~~~~~~~~ -->
<!-- Element for the behavioral part of a transition (not including the header symbol such as the
        input or the start). -->

<!ELEMENT Transition ((SdlFSMSymbol | DecisionBlock)*)>


<!-- Element DecisionBlock -->
<!-- ~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a decision block in a transition. The symbol is the decision, or transition
        option. -->

<!ELEMENT DecisionBlock (SdlFSMSymbol, DecisionBranch*)>


<!-- Element DecisionBranch -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a branch in a decision. The symbol is the answer. -->

<!ELEMENT DecisionBranch (SdlFSMSymbol, Transition)>


<!-- Element SdlFSMSymbol -->
<!-- ~~~~~~~~~~~~~~~~~~~~ -->
<!-- Element for a single symbol. The default value "-" for the colors means they should use the
        default color. -->

<!ELEMENT SdlFSMSymbol EMPTY>
<!ATTLIST SdlFSMSymbol
  identifier          ID            #REQUIRED
  type                %SymbolType;  #REQUIRED
  text                CDATA         ""
  outline_color       CDATA         "-"
  background_color    CDATA         "-"
```

```
  center_coordinates  CDATA        ""
  dimensions          CDATA        ""
  attached_symbol_id  CDATA        ""
>
```

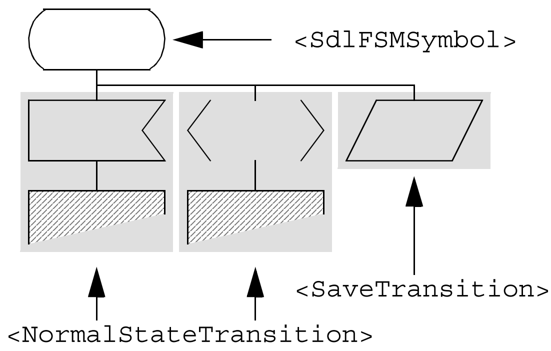
11.3.3 Explanations

The top-level XML element for a behavioral diagram is `<SdlFSMDiagram>`. Just as for standard diagrams, each diagram is split into partitions described via the `<SdlFSMPartition>` element. A partition has a page specification, just like partitions in standard diagrams, but their contents is specific:

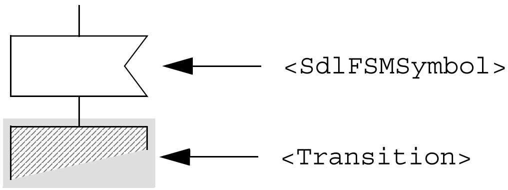
- Each of the top-level symbols appearing in it (e.g, a declaration symbol) is described via a `<SdlFSMSymbol>` element;
- Each state transition block appearing in it is described via a `<StateTransitionBlock>` element;
- Each non-state transition appearing in it (e.g, the start transition) is described via a `<NonStateTransition>` element.

More precisely, here is what each XML element represents and how it is structured:

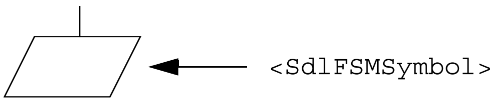
- `<StateTransitionBlock>`:



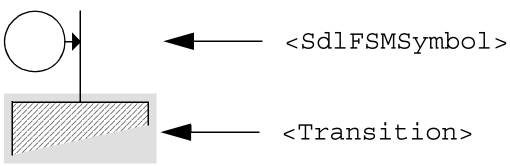
- `<NormalStateTransition>`:



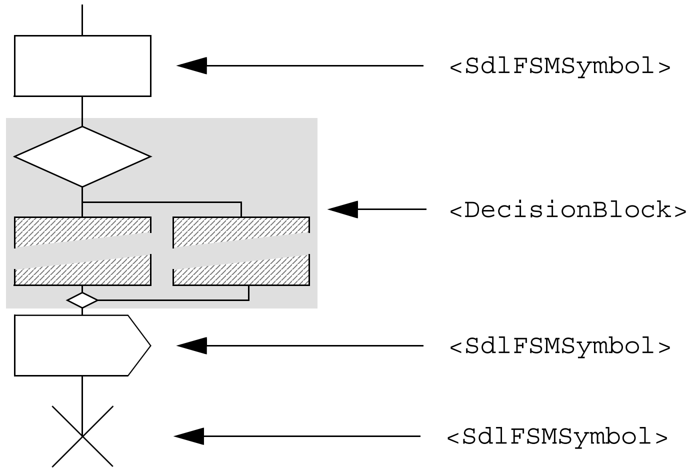
- `<SaveTransition>`:



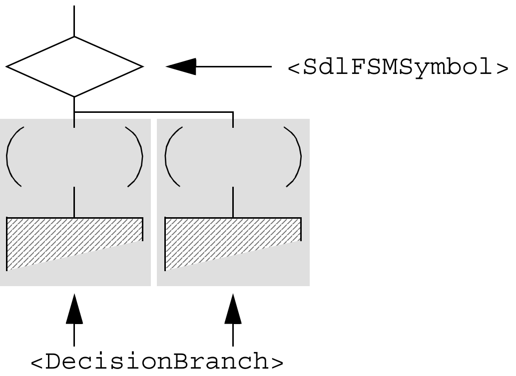
- `<NonStateTransition>`:



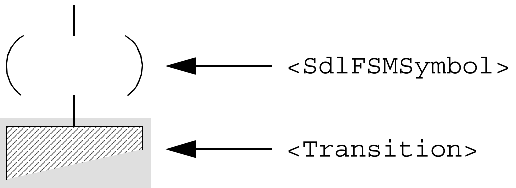
- <Transition>:



- <DecisionBlock>:



- <DecisionBranch>:



There are no explicit links described for this kind of diagram: all displayed links are deduced from the position of the symbols they link.
State transition blocks that start on a next state symbol in a transition will have the `<SdlFSMSymbol>` representing the state replaced by a `<SdlFSMSymbolReference>`, with its `identifier` attribute being the identifier for the next state symbol in the other transition. Note that in this case, the state transition block's `top_left_coordinates` attribute is irrelevant and ignored, since its position can be computed from its state symbol.
Symbols attached to other ones such as comment symbols will appear as top-level symbols in their parent partition, but will have their `attached_symbol_id` attribute set to the identifier of the symbol they are attached to. The link is made both ways: the symbol they are attached to will also have its `attached_symbol_id` attribute set to the identifier of the attached symbol.

11.4 - MSC diagram DTD

11.4.1 Principles

A MSC diagram is described mainly as a series of events happening on lifelines (instances). More precisely, it has 3 main parts:

- The first part gives the sequence of lifelines in the diagram, ordered as they should be displayed. All lifelines are listed, including collapsed ones.
- The second part lists the collapsed lifelines and the lifelines they “contain”.
- The third part gives the events actually happening in the diagram. Each event has a type and a set of attributes connecting it to lifelines or other events, or giving information about how it should be displayed.

11.4.2 DTD text

```
<!ENTITY % MscEventType1 "MESSAGE_SEND|MESSAGE_RECEIVE|MESSAGE_SAVE|OPERATION_CALL|
         OPERATION_RETURN">
<!ENTITY % MscEventType2 "TIMER_START|TIMER_CANCEL|TIMER_TIMEOUT|TIME_CONSTRAINT_START|
         TIME_CONSTRAINT_END">
<!ENTITY % MscEventType3 "PROCESS_CREATION|PROCESS_START|PROCESS_END">
<!ENTITY % MscEventType4 "SEMAPHORE_CREATION|SEMAPHORE_START|SEMAPHORE_END|SEMAPHORE_DELETION">
<!ENTITY % MscEventType5 "SEMAPHORE_TAKE|SEMAPHORE_TAKE_SUCCEEDED|SEMAPHORE_TAKE_TIMEOUT|
         SEMAPHORE_GIVE">
<!ENTITY % MscEventType6 "SEGMENT_START|SEGMENT_END|ACTION_SYMBOL_START|ACTION_SYMBOL_SPACER|
         ACTION_SYMBOL_END">
<!ENTITY % MscEventType7 "MSC_REFERENCE_START|MSC_REFERENCE_END|INLINE_EXPRESSION_START|
         INLINE_EXPRESSION_SEP|INLINE_EXPRESSION_END|CONDITION_START|CONDITION_END">
<!ENTITY % MscEventType8 "TEXT_SYMBOL|UNATTACHED_COMMENT|ATTACHED_COMMENT|ABSOLUTE_TIME">
<!ENTITY % MscEventType "(%MscEventType1;|%MscEventType2;|%MscEventType3;|%MscEventType4;|
         %MscEventType5;|%MscEventType6;|%MscEventType7;|%MscEventType8;)">

<!ELEMENT MscDiagram (LifelineOrdering, CollapsedLifeline*, MscEventRow*, PageSpecification?)>
<!ATTLIST MscDiagram
  message_params_visibility %MsgParamVis; "full"
>

<!ELEMENT LifelineOrdering EMPTY>
<!ATTLIST LifelineOrdering identifiers CDATA "">

<!ELEMENT CollapsedLifeline EMPTY>
<!ATTLIST CollapsedLifeline
  identifier              CDATA #REQUIRED
  collapsed_lifelines_ids CDATA #REQUIRED
>

<!ELEMENT MscEventRow (MscEvent+)>

<!ELEMENT MscEvent EMPTY>
<!ATTLIST MscEvent
  event_type          %MscEventType;   #REQUIRED
  lifeline_ids        CDATA            ""
  color               CDATA            "#000000"
  background          CDATA            "#ffffff"
  identifier          CDATA            ""
  type                CDATA            ""
  text                CDATA            ""
  text_size           CDATA            ""
>
```

11.4.3 Explanations

The main XML element is `<MscDiagram>`.

The first part in the diagram is given in the `<LifelineOrdering>` XML element: its identifiers attribute contains a space-separated list of lifeline identifiers.

The second part is given in the `<CollapsedLifeline>` XML elements, each associating an `identifier` for the collapsed lifeline to the space-separated list of identifiers for lifelines they contain (`collapsed_lifelines_ids`).

The third part is the sequence of `<MscEvent>` XML elements. Each of these elements is a description of an event happening in the diagram. The attributes for an event are:

- `event_type`: type for the event.
- `lifeline_ids`: space-separated list of identifiers for lifelines involved in the event.
- `color`: text and outline color for the event.
- `background`: background color for the event if applicable.
- `identifier`: identifier for the event itself if one is needed. This identifier is used to link 2 related events, such as a couple `MESSAGE_SEND` / `MESSAGE_RECEIVE` or a couple `TIME_CONSTRAINT_START` / `TIME_CONSTRAINT_END`.
- `type`: type for object described by the event.
- `text`: text for the object described by the event if any.
- `text_size`: size for the text for the object described by the event, giving its width and its height, separated by a space, both of which can be a '`-`' if it's missing or not significant.

All attributes do not apply to all event types, and may not contain the same inforamtion. Here is the compatibility matrix:

Table 2: Event attributes

Event type	lifeline_ids	color	identifier	type	text	text_size
MESSAGE_SEND	X	X	X	.	X	.
MESSAGE_RECEIVE	X	X	X	.	X	.
MESSAGE_SAVE	X(1)	X	.	.	.	.
OPERATION_CALL	X(2)	X	.	.	X	.
OPERATION_RETURN	X(2)	X	.	.	X	.
TIMER_START	X(1)	X	.	.	X	.
TIMER_CANCEL	X(1)	X	.	.	X	.
TIMER_TIMEOUT	X(1)	X	.	.	X	.
TIME_CONSTRAINT_START	X(1)	X	X	.	X	.
TIME_CONSTRAINT_END	X(1)	X	X	.	X	.

Table 2: Event attributes

Event type	lifeline_ids	color	identifier	type	text	text_size
PROCESS_CREATION	X(2)	X	.	.	.	.
PROCESS_START	X(1)	X	.	.	X	X(6)
PROCESS_END	X(1)	X	.	.	.	.
SEMAPHORE_CREATION	X(2)	X	.	.	.	.
SEMAPHORE_START	X(1)	X	.	.	X	.
SEMAPHORE_END	X(1)	X	.	.	.	.
SEMAPHORE_DELETION	X(1)	X	.	.	.	.
SEMAPHORE_TAKE	X(1)	X	.	.	X	.
SEMAPHORE_TAKE_SUCCEEDED	X(1)	X	.	.	.	.
SEMAPHORE_TAKE_TIMEOUT	X(1)	X	.	.	.	.
SEMAPHORE_GIVE	X(1)	X	.	.	.	.
SEGMENT_START	X(1)	X	X	X(4)	.	.
SEGMENT_END	X(1)	X	X	X(4)	.	.
ACTION_SYMBOL_START	X(1)	X	.	.	X	.
ACTION_SYMBOL_SPACER	X(1)	X	.	.	X	.
ACTION_SYMBOL_END	X(1)	X	.	.	X	.
MSC_REFERENCE_START	X(3)	X	X	.	X	.
MSC_REFERENCE_END	X(3)	X	X	.	X	.
INLINE_EXPRESSION_START	X(3)	X	X	X(5)	.	X(7)
INLINE_EXPRESSION_SEP	X(3)	X	X	X(5)	.	.
INLINE_EXPRESSION_END	X(3)	X	X	X(5)	.	.
CONDITION_START	X(3)	X	X	.	X	.
CONDITION_END	X(3)	X	X	.	X	.
TEXT_SYMBOL	.	X	.	.	X	.

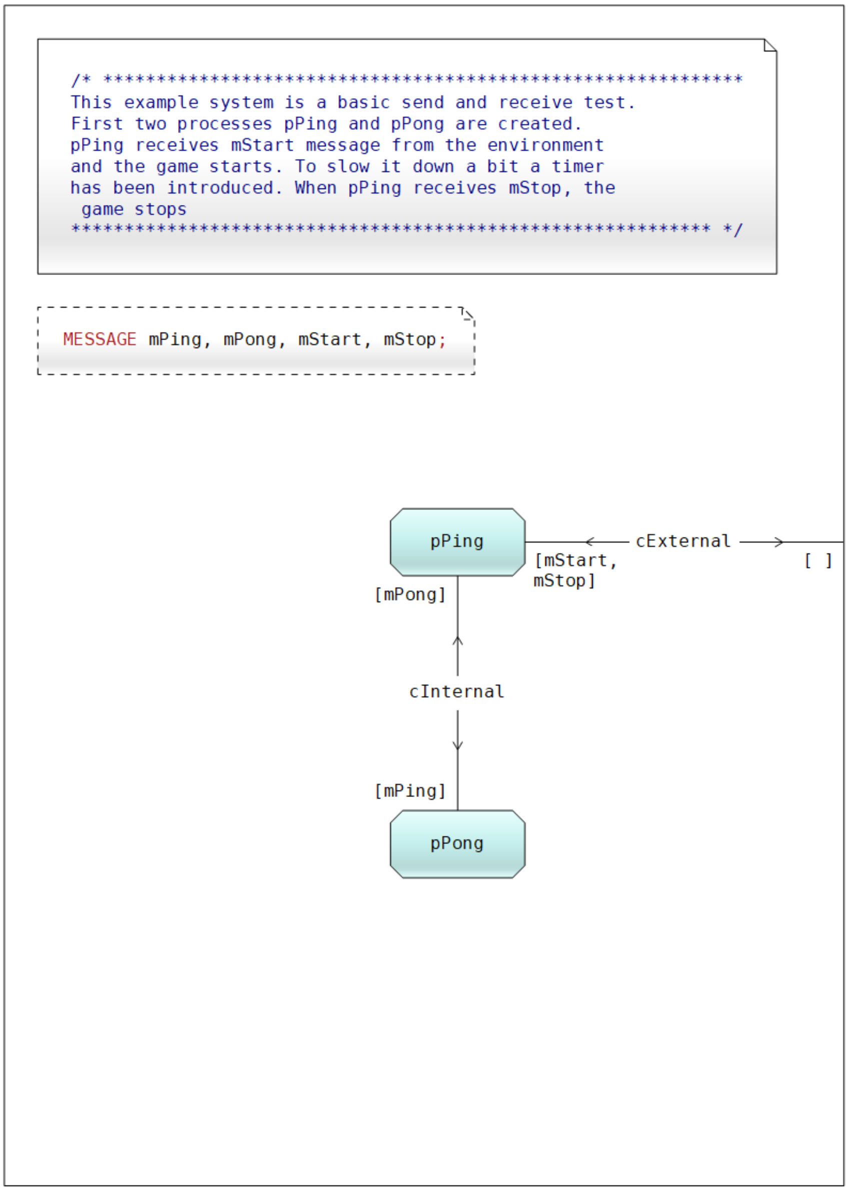
Table 2: Event attributes

Event type	lifeline_ids	color	identifier	type	text	text_size
UNATTACHED_COMMENT	.	X	.	.	X	.
ATTACHED_COMMENT	.	X	.	.	X	.
ABSOLUTE_TIME	.	X	.	.	X	.

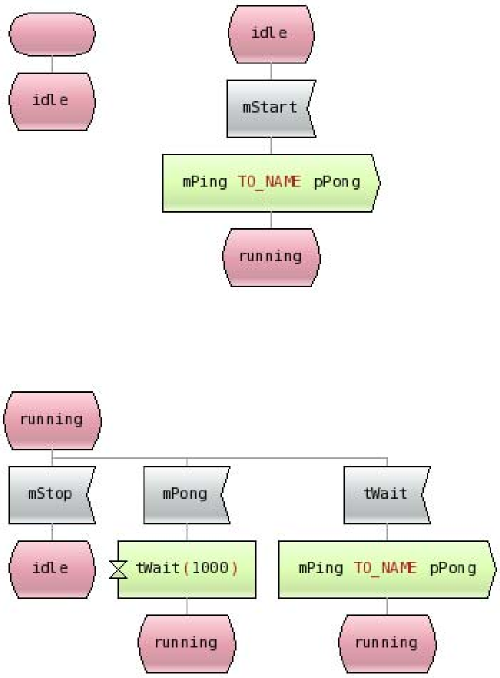
1. Only one lifeline identifier is allowed, which is the one on which the event happens.
2. Two lifelines must be specified. The first is the one onto which the event happens (called one, created one, etc…); the second one is the causing the action (caller, creator, etc…).
3. Several lifelines can be specified, listing all those spanned by the symbol.
4. The type is the type for the segment: suspended, method, co-region, etc…
5. The type is the type for the inline expression: `opt`, `alt`, `par`, etc…
6. The text size only specifies the width for the text, allowing to size the lifeline head correctly.
7. The text size allows to give the proper dimensions to the tag border in the inline expression.

12 - Example systems

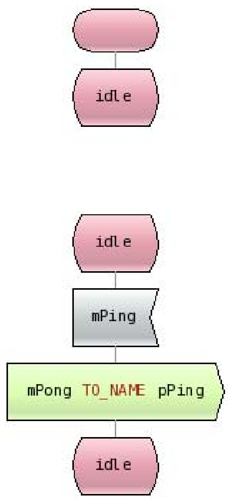
12.1 - Ping Pong



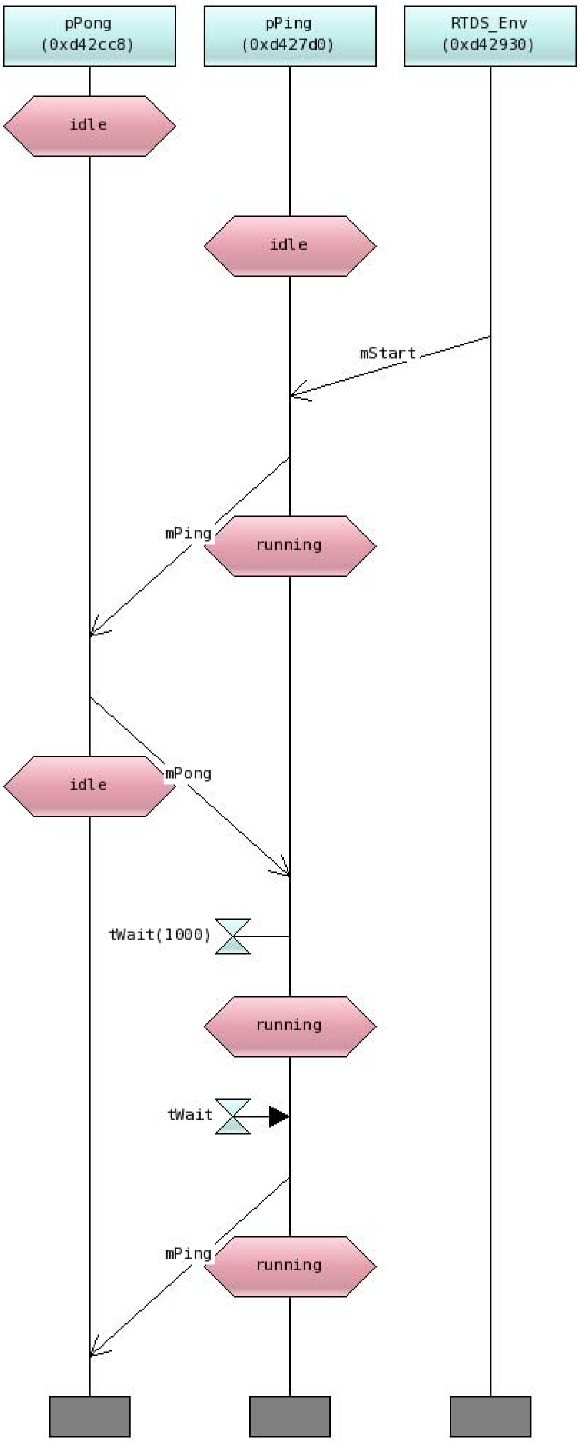
Ping pong system view



Ping process



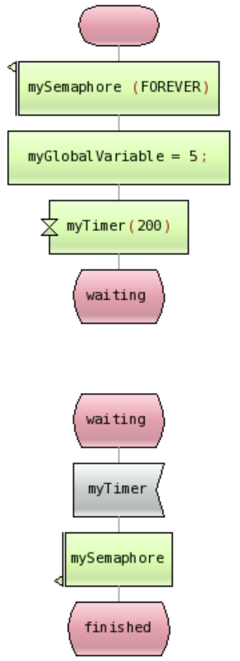
Pong process



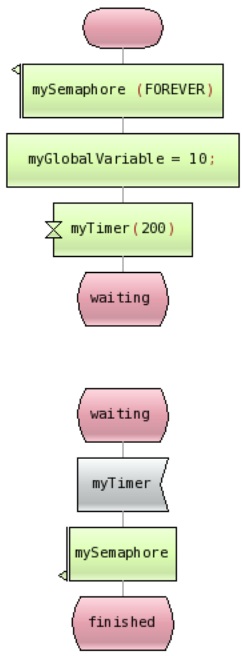
MSC trace of the ping pong system

12.2 - A global variable manipulation

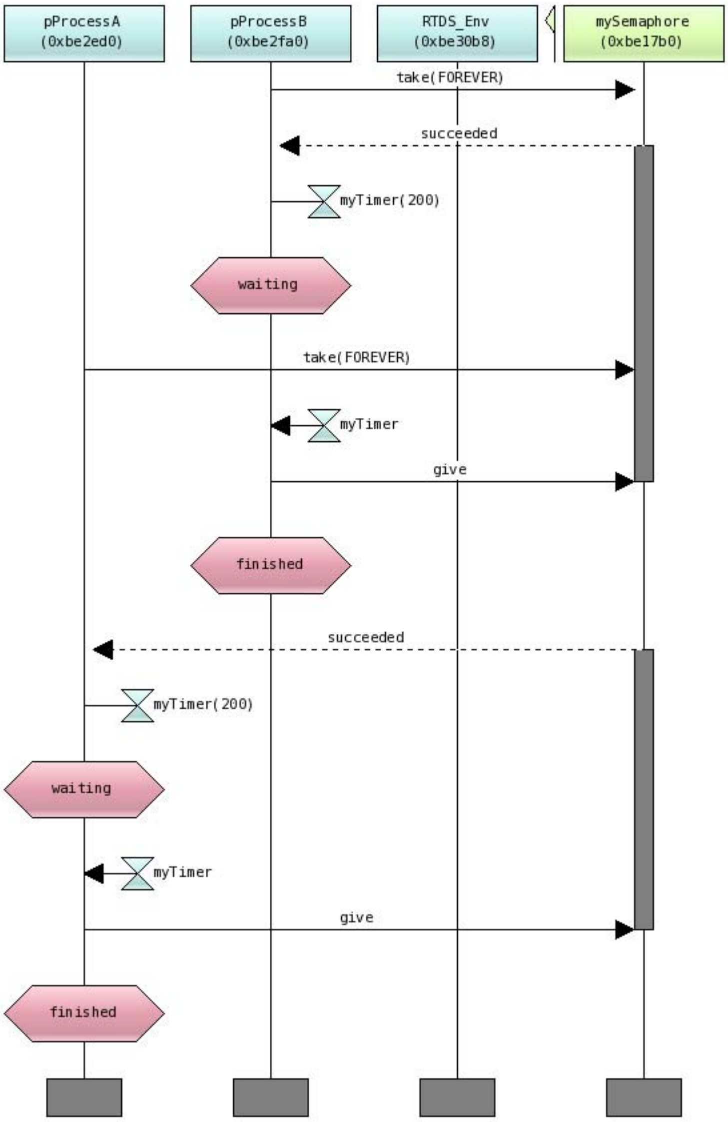
Global variable manipulation example system



Process A



Process B

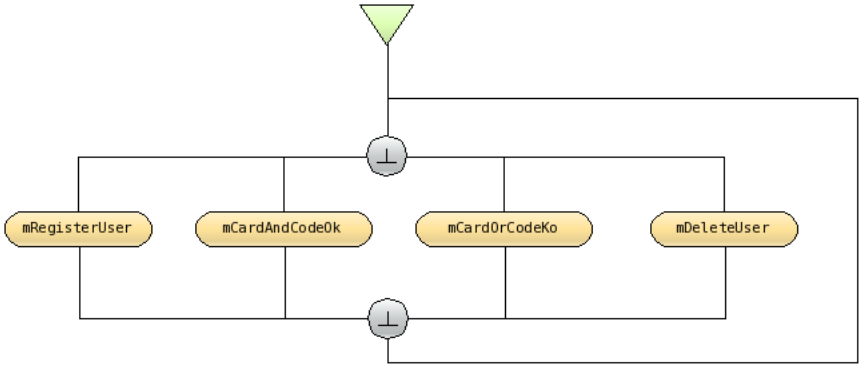


MSC trace of the global variable manipulation

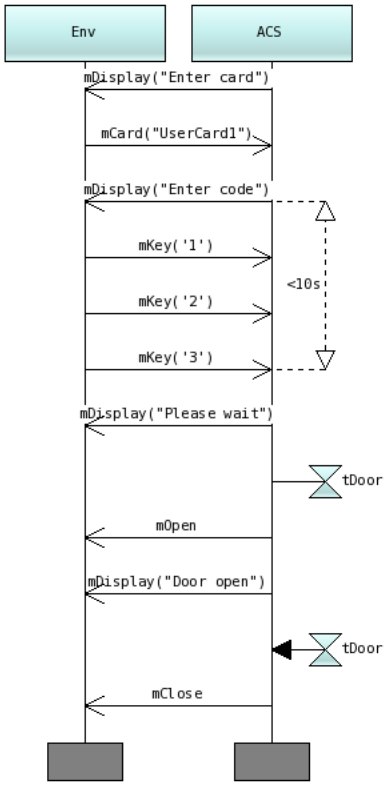
12.3 - Access Control System

This system controls the access to a building. To get in, one need to insert a card and type a code. The database is in the central block. When starting the system there is no user registered in the base so the first user needs to be the administrator.

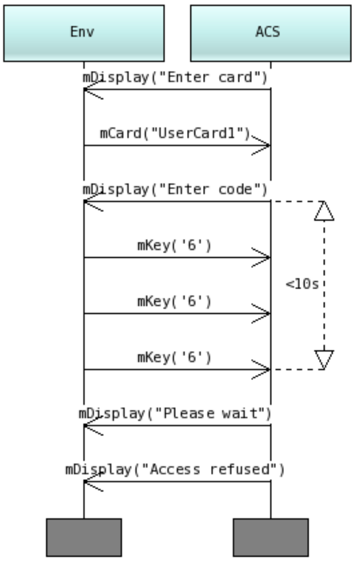
12.3.1 Requirements



Either one of the MSCs can be executed indefinitly

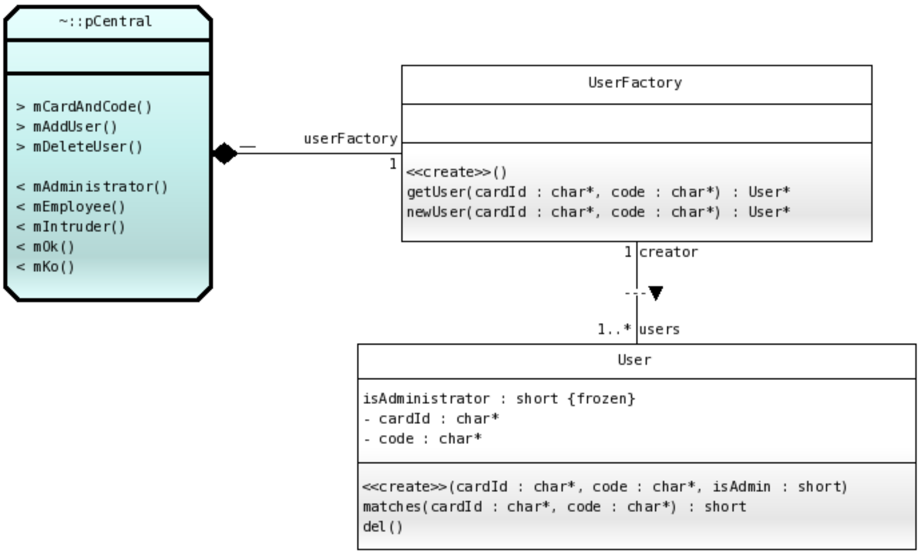


Standard scenario



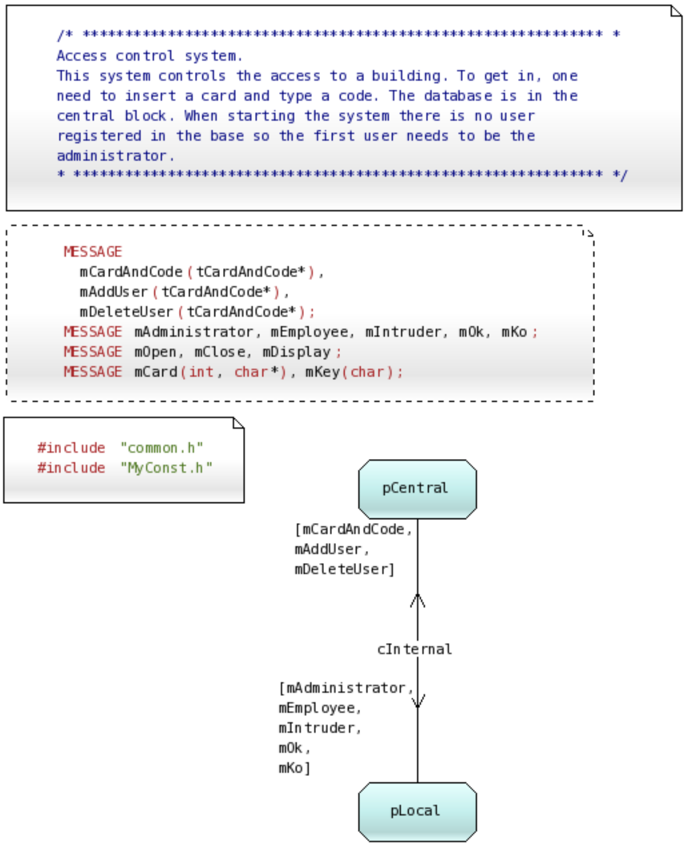
Standard refusal scenario

12.3.2 Analysis



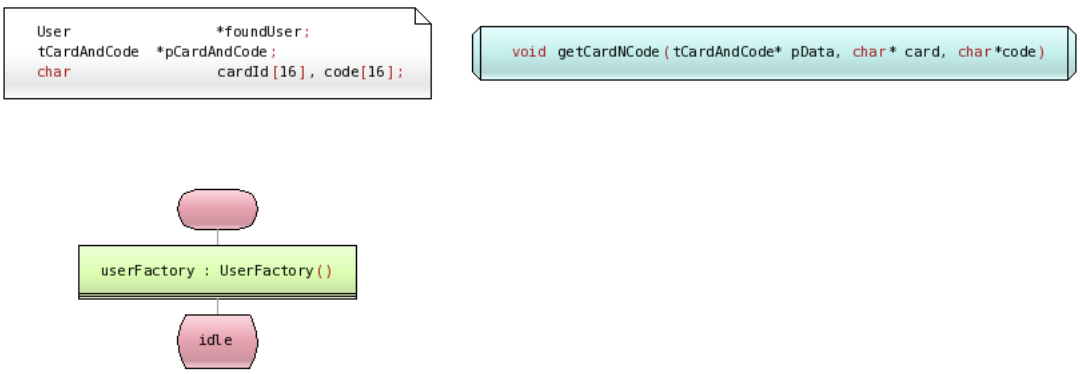
The class diagram shows the relation between pCentral (task) active class and UserFactory and User passive classes (C++)

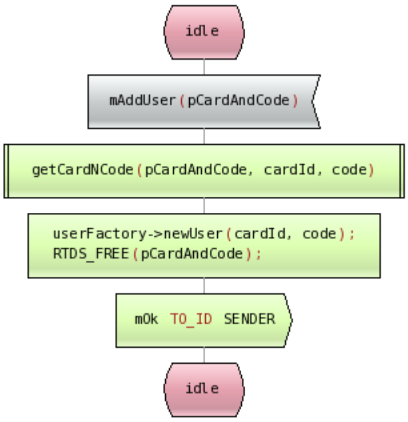
12.3.3 Architecture

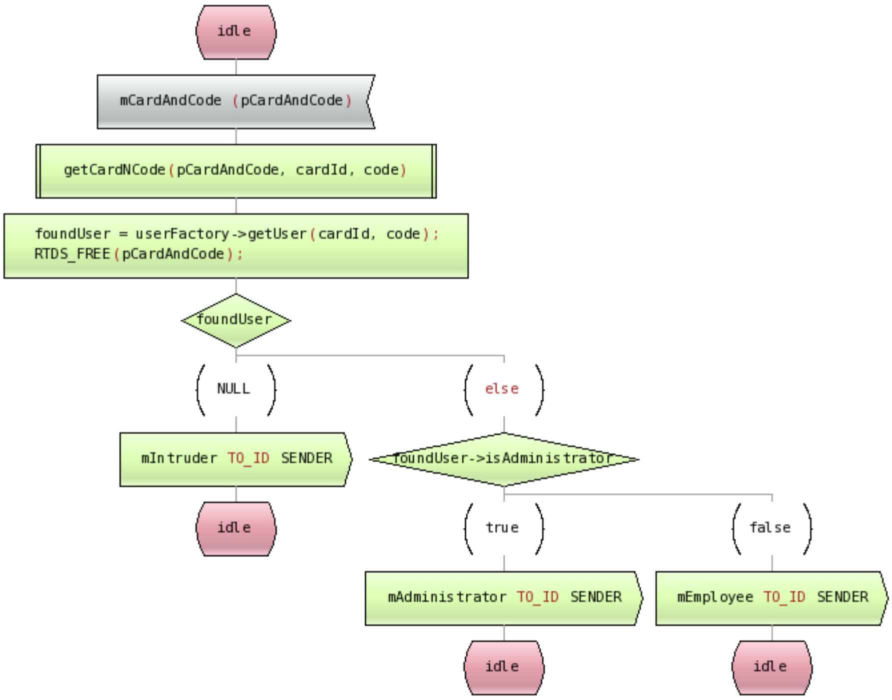


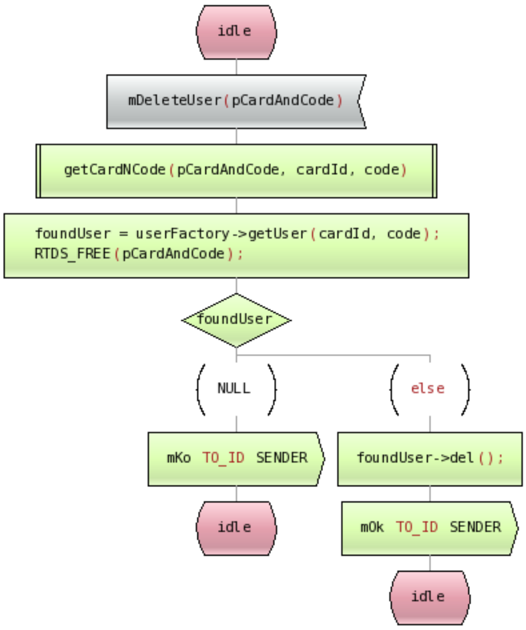
The system is made of two tasks: pCentral and pLocal

12.3.4 pCentral process

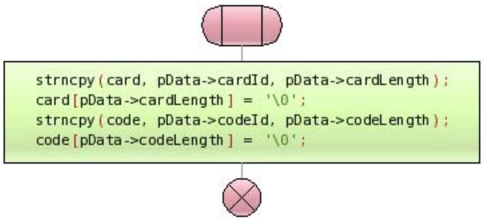



idle
mAddUser(pCardAndCode)
getCardNCode(pCardAndCode, cardId, code)
userFactory->newUser(cardId, code);
RTDS_FREE(pCardAndCode);
mOk TO_ID SENDER
idle

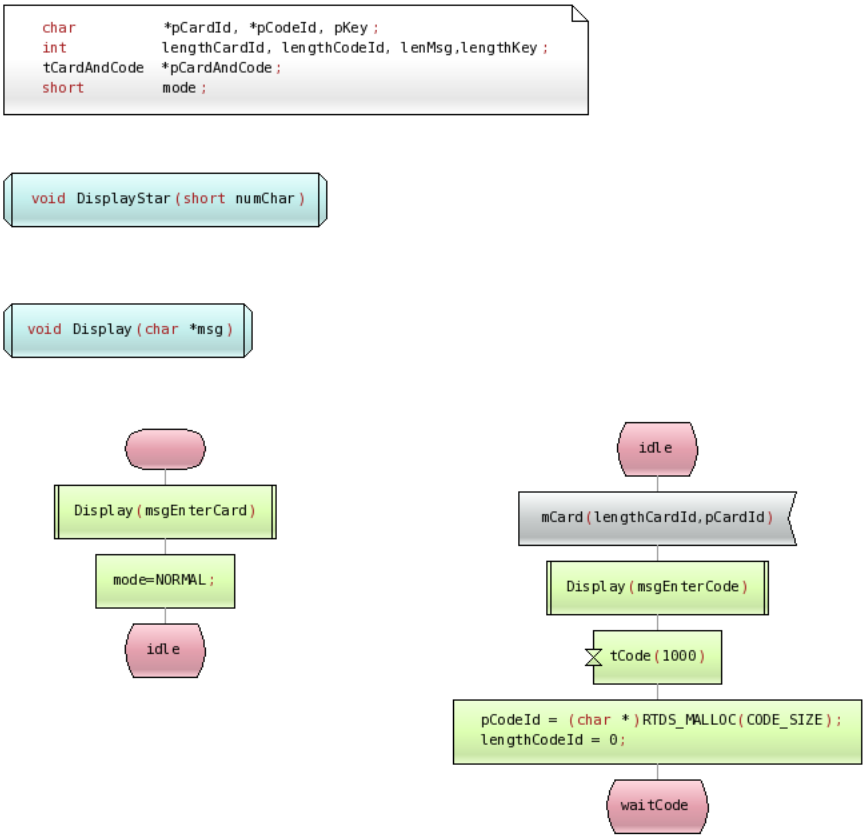

idle
mCardAndCode (pCardAndCode)
getCardNCode(pCardAndCode, cardId, code)
foundUser = userFactory->getUser(cardId, code);
RTDS_FREE(pCardAndCode);
foundUser
NULL
else
mIntruder TO_ID SENDER
foundUser->isAdministrator
idle
true
false
mAdministrator TO_ID SENDER
mEmployee TO_ID SENDER
idle
idle

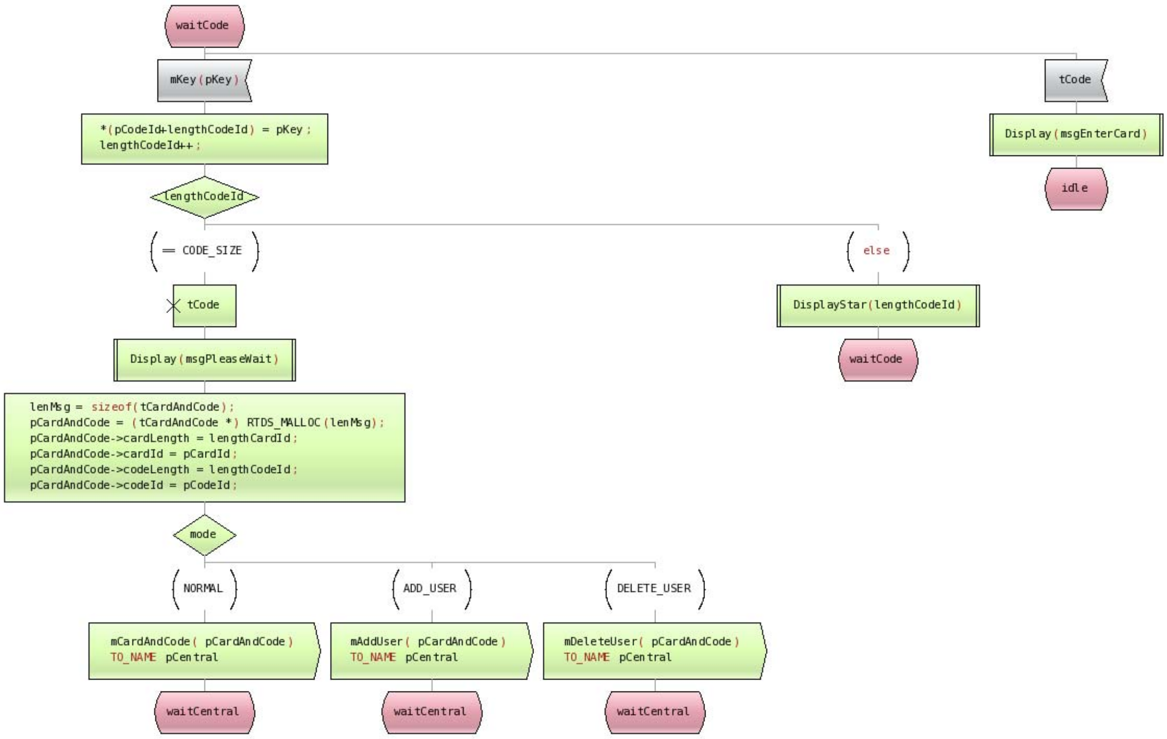


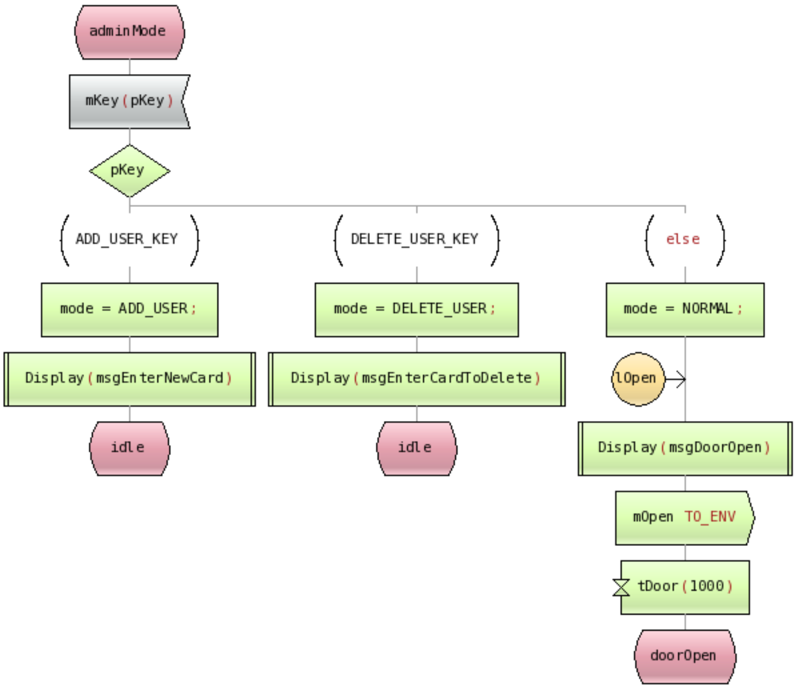
12.3.5 getCardNCode procedure

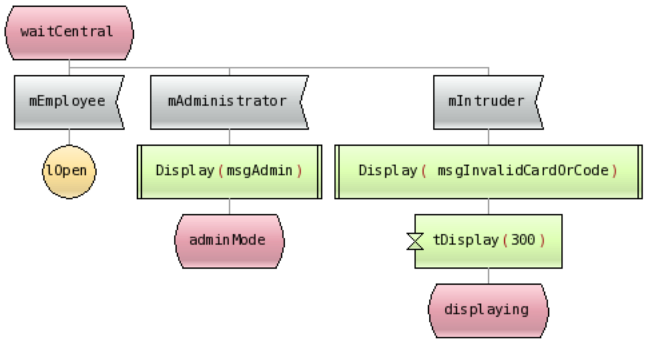


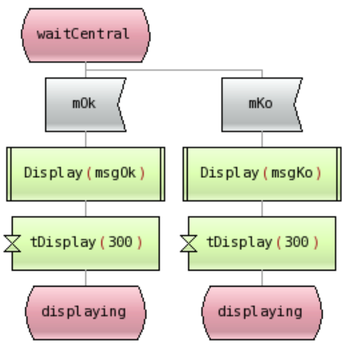
12.3.6 pLocal process

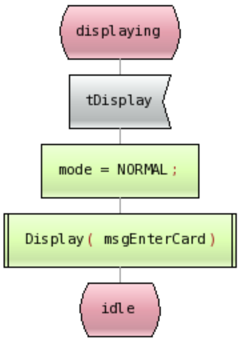


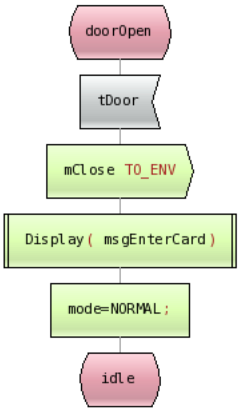

waitCode
mKey(pKey)
tCode
*(pCodeId+lengthCodeId) = pKey;
lengthCodeId++;
Display(msgEnterCard)
idle
lengthCodeId
== CODE_SIZE
else
tCode
DisplayStar(lengthCodeId)
Display(msgPleaseWait)
waitCode
lenMsg = sizeof(tCardAndCode);
pCardAndCode = (tCardAndCode *) RTDS_MALLOC(lenMsg);
pCardAndCode->cardLength = lengthCardId;
pCardAndCode->cardId = pCardId;
pCardAndCode->codeLength = lengthCodeId;
pCardAndCode->codeId = pCodeId;
mode
NORMAL
ADD_USER
DELETE_USER
mCardAndCode(pCardAndCode)
TO_NAME pCentral
mAddUser(pCardAndCode)
TO_NAME pCentral
mDeleteUser(pCardAndCode)
TO_NAME pCentral
waitCentral
waitCentral
waitCentral


adminMode
mKey(pKey)
pKey
ADD_USER_KEY
DELETE_USER_KEY
else
mode = ADD_USER;
mode = DELETE_USER;
mode = NORMAL;
Display(msgEnterNewCard)
Display(msgEnterCardToDelete)
lOpen
idle
idle
Display(msgDoorOpen)
mOpen TO_ENV
tDoor(1000)
doorOpen

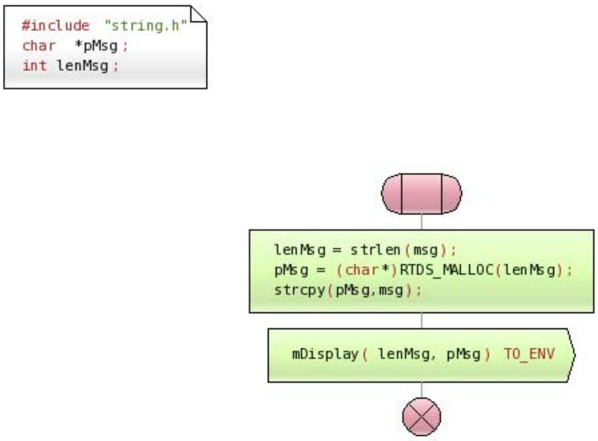

waitCentral
mEmployee
mAdministrator
mIntruder
lOpen
Display(msgAdmin)
Display(msgInvalidCardOrCode)
adminMode
tDisplay(300)
displaying


waitCentral
mOk
mKo
Display(msgOk)
Display(msgKo)
tDisplay(300)
tDisplay(300)
displaying
displaying

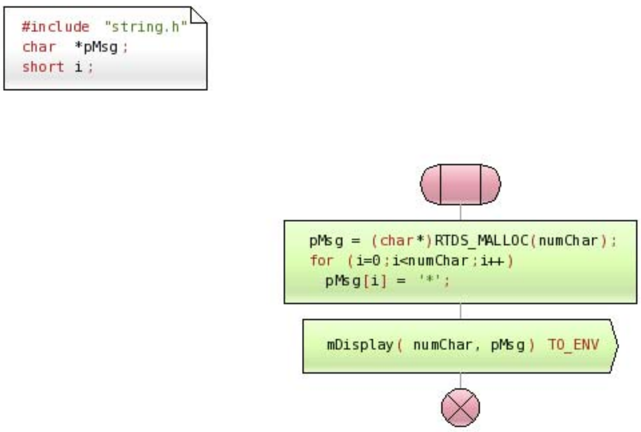

displaying
tDisplay
mode = NORMAL;
Display(msgEnterCard)
idle


doorOpen
tDoor
mClose TO_ENV
Display(msgEnterCard)
mode=NORMAL;
idle

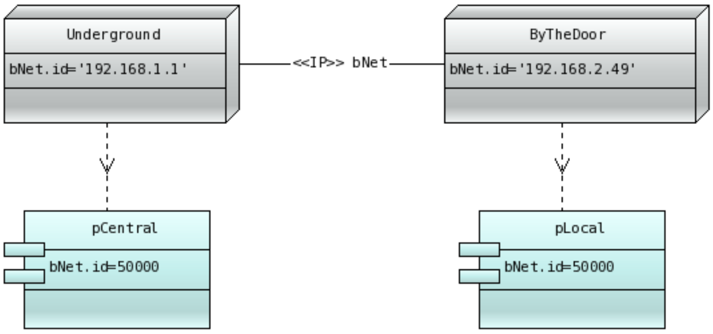
12.3.7 Display procedure



12.3.8 DisplayStar procedure



12.3.9 Deployment



The components communicate through IP

13 - Differences with classical SDL

It is difficult to list all the differences between SDL-RT and SDL even though an SDL developer would understand SDL-RT and vice versa. Still to be able to clearly state the differences between these languages we will pinpoint the main differences in the paragraphs below.

13.1 - Data types

This is the most significant difference between SDL and SDL-RT. Classical SDL has its own data types and syntax where SDL-RT basically uses ANSI C language. Some symbols have a specific syntax with SDL-RT since there is no C equivalent instruction such as output, input, save, or semaphore manipulations.
The advantages are obvious:

- the syntax is known by all real time developers,
- it implicitly introduces the concept of pointers that does not exist in SDL,
- it eases integration of legacy code where it is quite tricky to do from classical SDL,
- and last but not least it makes code generation out of SDL-RT quite straightforward.

13.2 - Semaphores

Semaphore is a key concept in real time systems that classical SDL misses. Specific semaphore symbols have been introduced in SDL-RT to answer the real time developer needs.

13.3 - Inputs

Classical SDL has nice concepts when it comes to dealing with message exchanges. But these concepts are not so interesting in real time development and are quite tricky to implement on a real target or operating system. That is why SDL-RT has removed the following concepts: enabling conditions when receiving a message, internal messages, two levels priority messages.

13.4 - Names

Classical SDL uses exotic names for some well known concepts such as "signal" where it is basically related to a "message". Since "message" is the usual name in Real Time Operating Systems SDL-RT uses the same term.
When it comes to object orientation classical SDL talks about "type" instead of the usual "class" term. SDL-RT uses the common developer word "class".

13.5 - Object orientation

Classical SDL uses "virtual", "redefined", and "finalized" when it comes to object oriented concepts. For example a super class should specify a transition is "virtual" so that the sub class is allowed "redefine" or "finalize" it. This is C++ like but actually quite painful when it comes to write and does not make things any clearer. SDL-RT takes the Java notation instead where there is no need to specify anything to be able to redefine it in a sub class.

14 - Memory management

Real time systems need to exchange information. The best way to do so is to have a reserved chunk of shared memory that several tasks can access. SDL-RT implicitly runs on such an underlying architecture since it supports global variables and exchanges message parameters through pointers. That raises memory management rules to follow to ensure a proper design.

14.1 - Global variables

SDL-RT processes can share global variables. This is very powerful but also very dangerous since the data can be corrupted if manipulated without caution. It is strongly recommended to use semaphores to access global variables to be sure data is consistent. An example of such a design is given later in this document.

14.2 - Message parameters

Parameters of a message are passed through a pointer. This implies the data pointed by the sending process will be accessible by the receiving process. Therefore a good design should meet the following rules:

- Sending processes allocate specific memory areas to store parameters,
- Once the message is sent the parameter memory area should never be manipulated again by the sending process,
- Receiver processes are responsible for freeing memory containing message parameters.

15 - Keywords

The following keyword have a meaning at in some specific SDL-RT symbols listed below:

keywords	concerned symbols
`PRIO`	Task definition Task creation Continuous signal

`TO_NAME`
`TO_ID`
`TO_ENV`
`VIA`
`TO_ALL`
	Message output

`FOREVER` `NO_WAIT`	semaphore manipulation

`>, <, >=, <=, !=, ==`
`true, false,`
`else`
	decision branches

`USE`
`MESSAGE`
`MESSAGE_LIST`
`STACK`
	additional heading symbol

Table 3: Keywords in symbols

16 - Syntax

All SDL-RT names must be a combination of alphabetical characters, numerical characters, and underscores. No other symbols are allowed.

Examples:

```
myProcessName
my_procedure_name
block_1
_semaphoreName
```

17 - Naming convention

Since some SDL-RT concepts can be reached through their names (processes, semaphores) each name in the system must be unique. This will make the design more legible and ease the support of SDL-RT in a tool.

It is suggested to use the following convention for names:

- block names should start with ’b’,
- process names should start with ’p’,
- timer names should start with ’t’,
- semaphore names should start with ’s’,
- global variables should start with ’g’.

18 - Lexical rules

A subset of the BNF (Backus-Naur Form) is used in these pages :

<traditional English expression>	as it says…
[<stuff>]	stuff is optional
{<stuff>}+	stuff is present at least one or more times
{<stuff>}*	stuff is present 0 or more times

19 - Glossary

ANSI	American National Standards Institute
BNF	Backus-Naur Form
ITU	International Telecommunication Union
MSC	Message Sequence Chart
OMG	Object Management Group
RTOS	Real Time Operating System
SDL	Specification and Description Language
SDL-RT	Specification and Description Language - Real Time
UML	Unified Modeling Language
XML	eXtensible Markup Language

20 - Modifications from previous releases

20.1 - V1.0 to V1.1

- *Semaphore manipulation*:
  The semaphore take now returns a status that indicates if the take attempt timed out or was successfull. The semaphore lifeline gets grayed when the semaphore is unavailable.

20.2 - V1.1 to V1.2

- *Object orientation*:
  There has been an error in the object orientation chapter: it is not possible to declare a process class or a block class in a block class definition diagram.
- *Messages*:
  - Messages now needs to be declared.
  - Message parameters are now typed with C types.
  - Parameter length can be omited if the parameter is structured. Then the length is implicitly the sizeof the parameter type.
  - The VIA concept has been introduced.
- *MSC*:
  Saved messages representation introduced.

20.3 - V1.2 to V2.0

- *Object orientation*:
  - UML class diagram has been introduced
  - UML deployment diagram has been introduced
  - Object creation symbol introduced in the behavior diagram
- *Tasks*:
  STACK parameter has been added as a parameter when creating a task.
- *Organisation*:
  Chapters have been re-organized.

20.4 - V2.0 to V2.1

- *Messages*:
  Messages can have multiple parameters. Declaration, inputs, and outputs have changed.

20.5 - V2.1 to V2.2

- *Object orientation*:
  - Super class transition symbol added
  - Super class next state symbol added

- *New concepts*:
  "Composite state" has been introduced.

20.6 - V2.2 to V2.3

- *MSC*:
  - Inline expressions support added.
  - Property Sequence Charts (PSC) support added.
  - Change in MSC diagram DTD for textual representation.

20.7 - V2.3 to V2.4

- *Messages*:
  Broadcast introduced.
- *Textual representation*:
  New representation for behavioral diagrams.

21 - Index

A

Action
  symbol 25
Action symbol
  MSC symbol 60
Additional heading symbol 32
Agents 8
Aggregation
  class 82
  node 88
alt
  Inline expression 58
Alternative operator
  PSC 61
Association 81

B

Block
  class 67
BROADCAST 22

C

call
  procedure 28
Cardinality 81
channels 10
Class
  active 79
  block 67
  definition 78
  passive 79
  process 68
Comment 30
  MSC symbol 59
Component 85
Component instance
  PSC 61
Composition 83
Connection 86
Connectors 28
Constraints
  PSC 61
Continuous signal 24
Coregion 55
creation
  task 27

D

Data type
  difference with classical SDL 121
Data types 66
Decision 25
Declaration
  message 41
  procedure 40
  process 39
  semaphore 42
  timer 42
  variables 32
Dependency 87
Diagram
  architecture 8
  behavior 13
  class 78
  communication 10
  contained symbols 89
  deployment 85
  MSC 43
Distributed system 85

E

else
decision 26
keyword 124
Environment
definition 8
message output 18
exc
Inline expression 58
Extension 31

F

false
decision 26
keyword 124
transition option 29
FOREVER
keyword 124

G

Generalisation 81
give
semaphore 27

H

HMSC 64

I

if 25
ifdef 29
Inline expression 58
alt 58
exc 58
loop 58
opt 58
par 58
seq 58
Input
difference with classical SDL 121
message 15
instance
MSC 43

K

Keywords 124

L

Lexical rules 127
loop
Inline expression 58
Loop operator
PSC 61

M

Memory
management 123
MESSAGE
keyword 124
Message
broadcast 21
communication principles 10
declaration 41
input 15
list 41
memory management 123
MSC 46
output 16
parameters 123
PSC 61
save 23
MESSAGE_LIST
keyword 124

MSC 43
action 60
agent instance 43
comment 59
reference 56
semaphore 44
text symbol 59

N

Naming
convention 126
difference with classical SDL 121
syntax 125
NO_WAIT
keyword 124
Node 85

O

Object
difference with classical SDL 121
OFFSPRING
procedure 28
Operators
PSC 61
opt
Inline expression 58
output 16

P

Package 83
par
Inline expression 58
Parallel operator
PSC 61
PARENT
procedure 28
PRIO
continuous signal 24
keyword 124
Procedure
call 28
declaration 40
return 32
start 31
Process
behavior 13
class 68
declaration 39
priority 39
Property Sequence Chart 60
PSC 60
Alternative operator 61
Component instance 61
Constraints 61
Loop operator 61
Message 61
Operators 61
Parallel operator 61
Relative time constraint 61
Strict operator 61
Unwanted chain constraint 61
Unwanted message constraint 61
Wanted chain constraint 61

R

reference
MSC 56
Relative time constraint
PSC 61
return
procedure 32

S

save 23
SDL-RT
Lexical rules 127
Semaphore
declaration 42

difference with classical SDL 121
give 27
global variable 123
MSC 44
take 26
SENDER
procedure 28
seq
Inline expression 58
Specialisation 81
STACK
keyword 124
Stack
size definition 33
Start
procedure 31
symbol 13
timer 27
State 13
MSC 49
Super class 34
Stereotype 78
Stop
symbol 14
timer 27
Storage format 90
Strict operator
PSC 61
Super class
state 34
transition 34
Symbol
additional heading 32
in diagram 89
ordering 33
text 32
Synchronous calls
MSC 48
System 8

T

take
semaphore 26
Task
creation symbol 27
Text
MSC symbol 59
symbol 32
Time interval
MSC 53
Timer
declaration 42
MSC 51
start 27
stop 27
TO_ALL 22
keyword 124
TO_ENV 18
keyword 124
TO_ID 17
keyword 124
TO_NAME 18
keyword 124
Transition
Super class 34
Transition option 29
true
decision 26
keyword 124
transition option 29

U

Unwanted chain constraint
PSC 61
Unwanted message constraint
PSC 61
USE
keyword 124

V

VIA 19
keyword 124

W

Wanted chain constraint
PSC 61

X

XML
data storage 90