SDL Forum 2007

SDL-RT tutorial

Eric Brunel

eric.brunel@pragmadev.com

State of the art

- C language is predominant (75%)
- C++ has been introduced in non real time parts of embedded (40%)
- Assembler (40%)
- Java is experienced in niches (less than 5%)
- 90% of the real time development projects use no graphical tool

Tendency

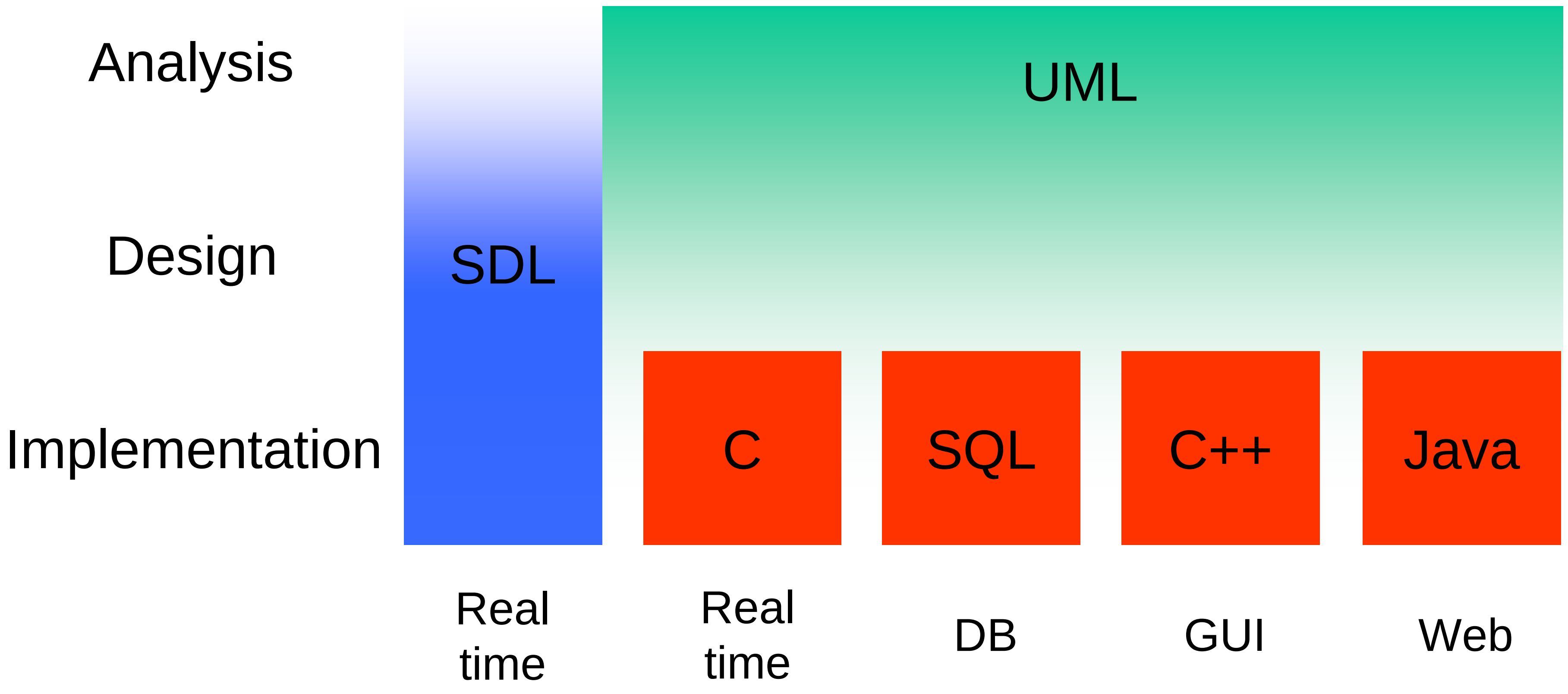
- Number of modules is dramatically increasing because of application complexity,
- Legibility of large systems gets critical,
- Debug on target is costly and sometimes reveals mistakes in the previous development phases,
- Re-use is compulsory regarding:
    - Legacy code,
    - Upcoming code.

Existing languages

- **SDL** (Specification and Description Language) and **MSC** (Message Sequence Chart) are ITU (International Telecommunication Union) standards.
    - Event oriented,
    - Used by ETSI to standardize telecommunication protocols,
    - Graphical,
    - Formal (complete and non-ambiguous), i.e. allows to fully describe the system,
    - Object oriented,

- **UML** (Unified Modeling Language) standardized by the OMG (Object Management Group).
    - Can be used to represent any type of systems,
    - Graphical,
    - Used at a pretty high level of abstraction,
    - Not formal, i.e. another language is necessary to describe in detail (C, C++, Java, SDL),
    - Very object oriented.

Languages positioning



Analysis

Design

Implementation

SDL

UML

C

SQL

C++

Java

Real time

Real time

DB

GUI

Web

No real time specificity in UML

- UML has no graphical representation for classical real time concepts such as: semaphores, messages, timers…
- UML is adapted to C++ for static data representation.
- Deployment diagram perfect for distributed systems.
- In practice UML models are not synchronized with the design.

Will UML2.0 help ?

- UML 2.0 main objective is to support MDA (Model Driven Architecture)
- MDA allows to define domain specific profiles

- ➢ A standard real time profile needs to be defined and used by all tool vendors
- ➢ Meanwhile UML 2.0 models will probably not be portable from one tool to another and have specific notations

UML 2.0 trend

- UML 2.0 Sequence diagram has integrated most of the features of the SDL Message Sequence Chart
- UML 2.0 structural diagram is very similar to the SDL block diagram

- ➢ Interesting things come from SDL
- ➢ ITU-T is standardizing a UML profile based on SDL for telecommunications (Z.109)

SDL: the perfect picture

- SDL graphical abstraction (architecture, communication, behavior) brings a lot to development teams.
- SDL being formal, it is possible to simulate the system behavior on host with graphical debugging facilities.
- SDL being formal, full code generation is possible.
- SDL being object oriented, software components are reusable (ETSI telecommunication protocol standards fully use object orientation).

SDL: the reality

- All existing software modules (RTOS, drivers, legacy code) provide C APIs, not SDL.
- SDL concepts do not map easily to RTOS and implementation languages:
  - Nested scopes for processes / procedures
  - Priority: messages vs. processes
  - Imported / exported vs. global semaphore-protected variables
  - …
- SDL syntax is unusual for C/C++ developers.

➢ **Integration with legacy code is difficult,**

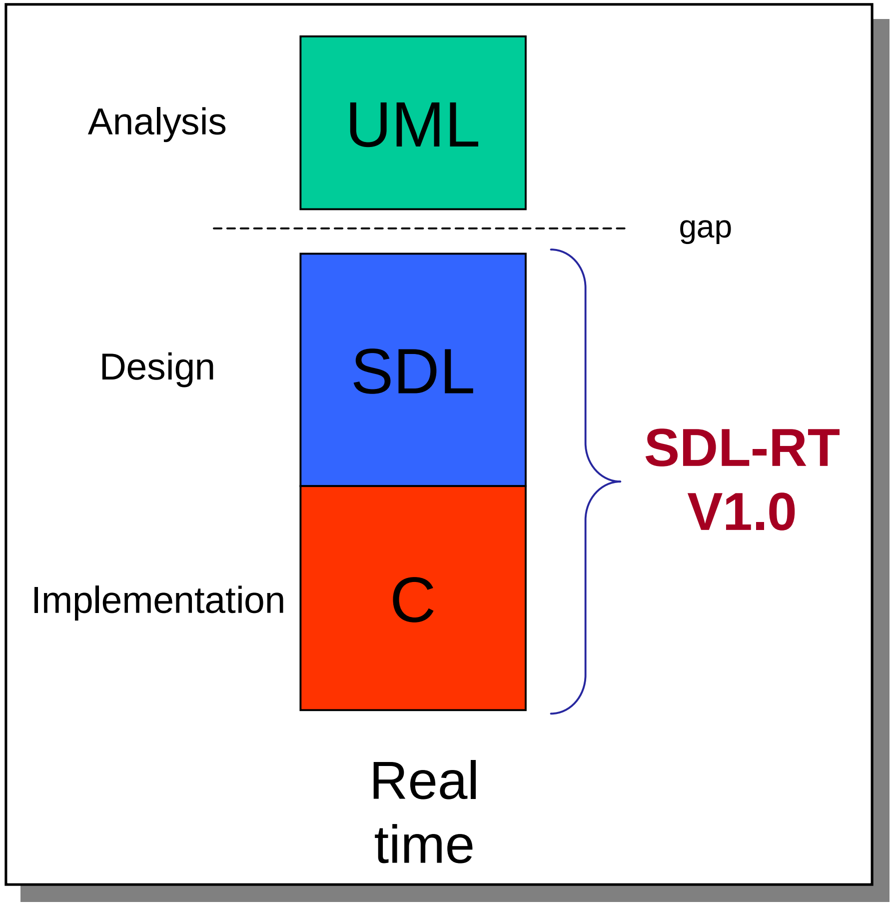
➢ **Integration with off the shelf components is tricky (driver or RTOS),**

➢ **Developers are frustrated,**

➢ **Generated code is not legible.**

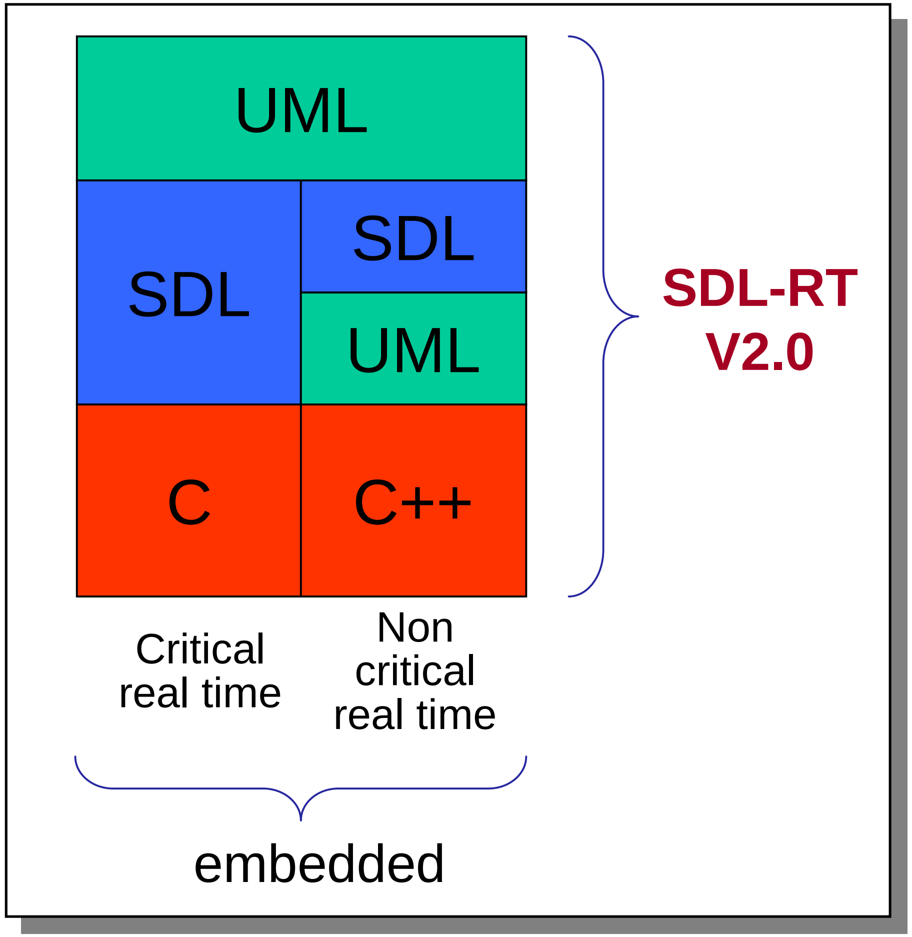
The technical solution: SDL-RT

- Keep UML diagrams at high level during analysis and requirements
- Keep the SDL graphical abstraction (architecture, communication, behavior).
- Introduce C data types and syntax instead of SDL's.
- Remove SDL concepts having no practical implementation.
- Extend SDL to deal with uncovered real time concepts (interrupts, semaphores).



Extended solution: UML integration

- Architecture and dynamic aspects are described with SDL
- Static aspects are described in UML
- Combination of both languages provide perfect graphical representation of all aspects of the system.



UML

SDL

SDL

UML

C

C++

SDL-RT V2.0

Critical real time

Non critical real time

embedded

Languages

SDL-RT combines industrial habits and standardized languages.
It can be considered as a UML 2.0 real time profile.

- SDL
    - Functional object oriented approach
    - Architecture definition
    - Detailed behavior
- UML
    - Pure object oriented approach
    - Library of components
- C
    - High performance
    - Precision
- C++
    - Lower performance
    - Higher layer of the embedded application

SDL RT

specification & description language - real time

Real Time Developer Studio is based on SDL-RT specification

SDL-RT is:

- Available from http://www.sdl-rt.org for free,
- Legible,
- Based on a standardized textual format (XML).

sdl

UNIFIED
MODELING
LANGUAGE
UML™

SDL-RT: graphical representations

- Library of components
- Object oriented specification
- System architecture
- Interface definitions
- Application deployment
- Real time concepts
- Key points in the design

Development process

Architecture based on agents:

- System
- Block
- Process
- Procedure

Development process

Interfaces

- Static

  Messages definition with typed parameters

- Dynamic

  MSC

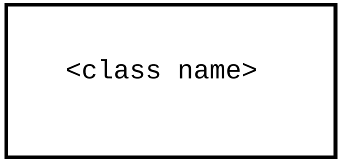
Development process

Behavior

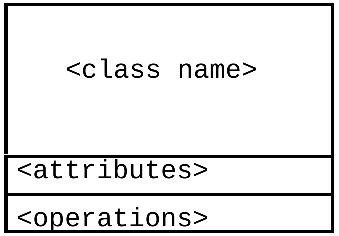
- Finite state machines
- C code
- C++ code

SDL-RT: class concept

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



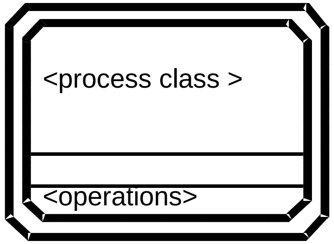
Class symbol with details suppressed



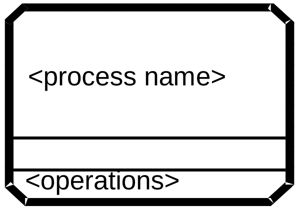
Class symbol full representation

SDL-RT: active class symbols

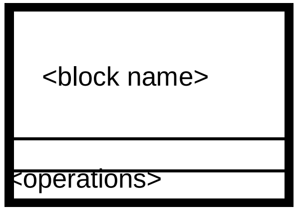
An instance of an active class owns a thread of control and may initiate control activity. An instance of a passive class holds data, but does not initiate control.



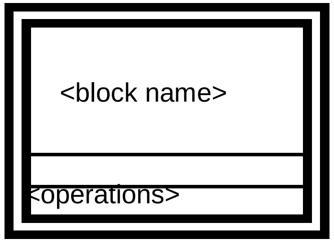
Class stereotyped as a class of process



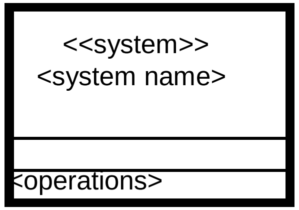
Class stereotyped as a process



Class stereotyped as a block



Class stereotyped as a class of block



Class stereotyped as a system

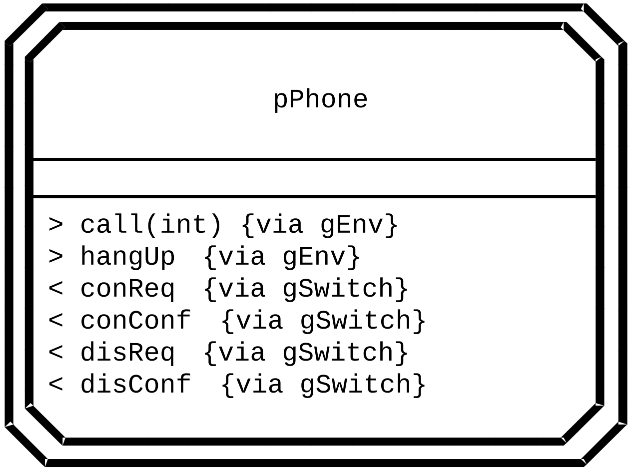
SDL-RT: active class interface

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.



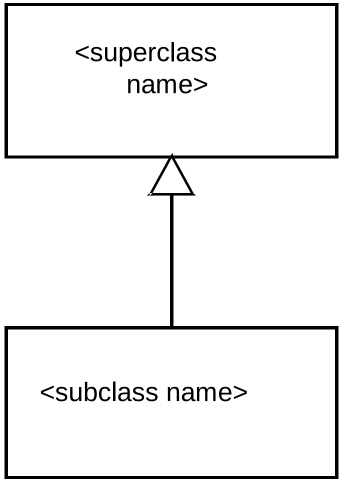
```
pPhone

> call(int) {via gEnv}
> hangUp  {via gEnv}
< conReq  {via gSwitch}
< conConf  {via gSwitch}
< disReq  {via gSwitch}
< disConf  {via gSwitch}
```

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

SDL-RT: class specialisation

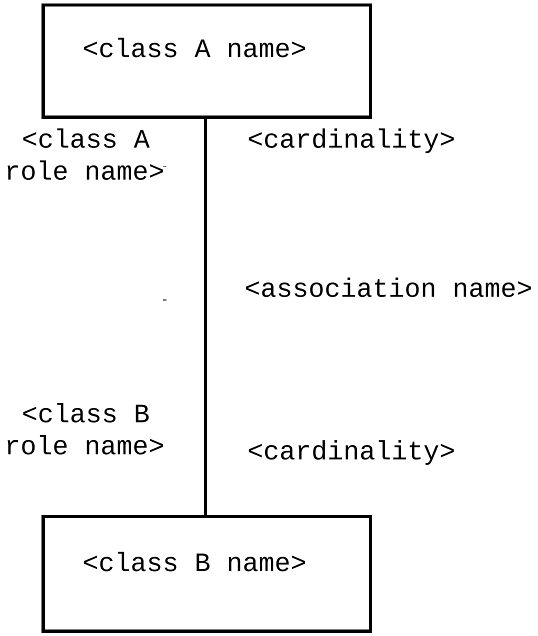
Specialisation defines a ’is a kind of’ relationship between two classes. The most general class is called the superclass and the specialised class is called the subclass.



Subclass is a kind of superclass

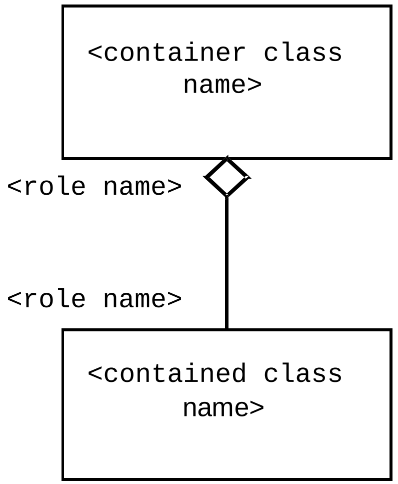
SDL-RT: class association

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 each end of the association.



SDL-RT: class aggregation and composition

- **Aggregation** defines a ’is a part of’ relationship between two classes.
- **Composition** is a strict form of aggregation, in which the parts’ existence depend on the container’s.



<container class name>

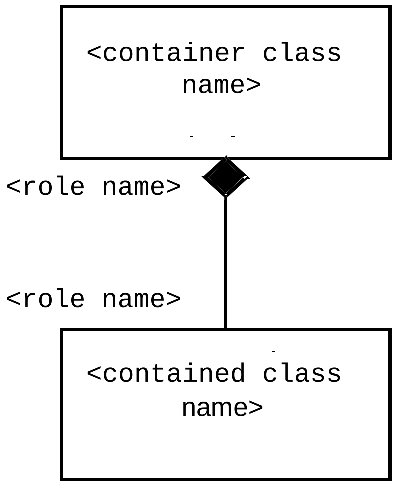
<role name>

<role name>

<contained class name>

contained class is a part of container class

Aggregation



<container class name>

<role name>

<role name>

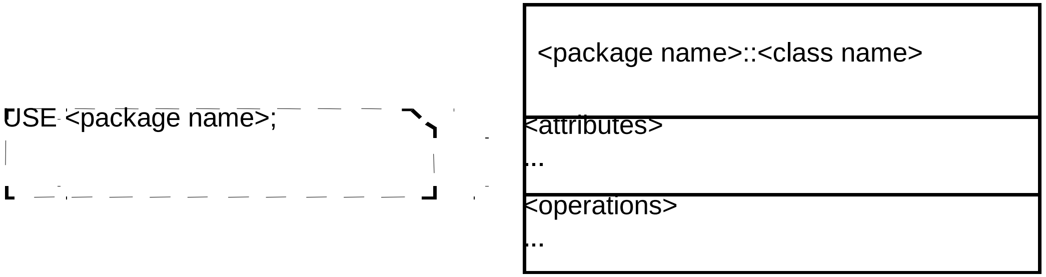
<contained class name>

contained class is a part of container class

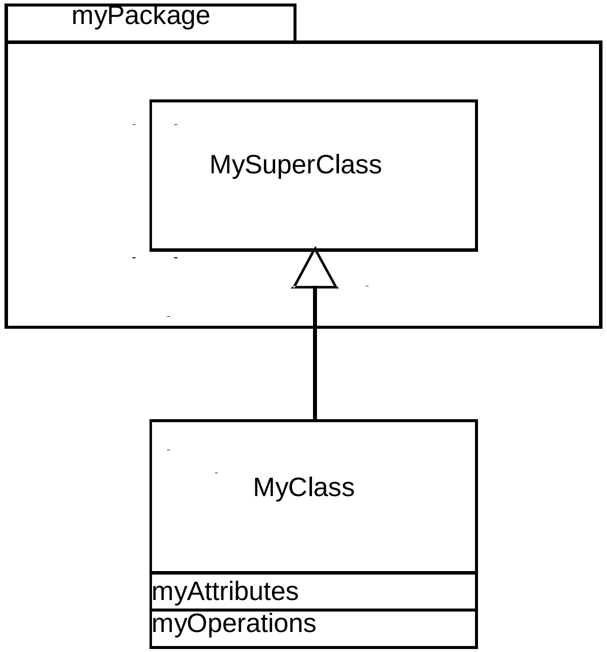
Composition

SDL-RT: class aggregation and composition

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



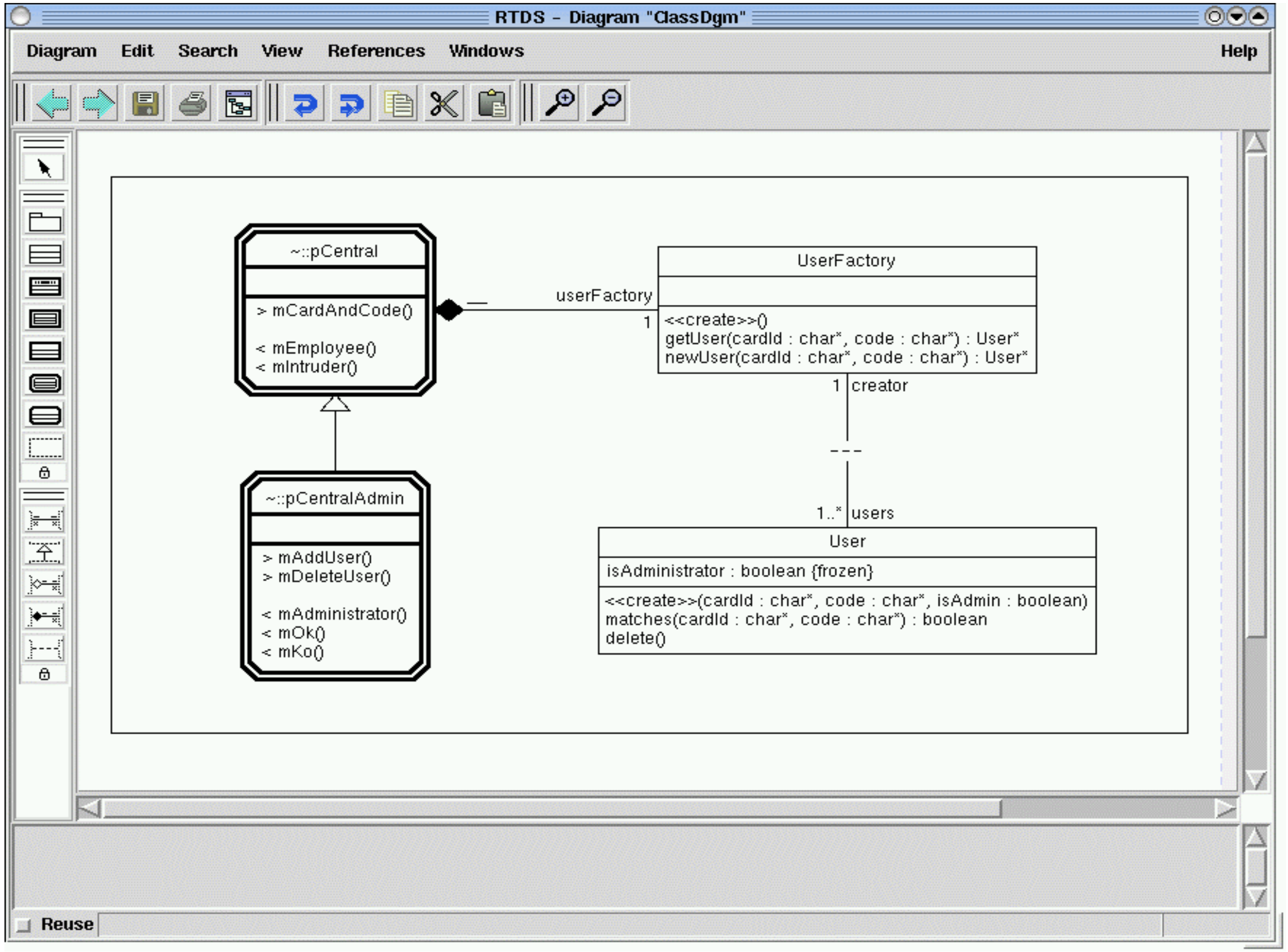
Class <class name> is defined in package<package name>



MyClass specialises MySuperClass defined in myPackage

SDL-RT: class diagram

Relations
between static
classes (C++)
and dynamic
classes (SDL)

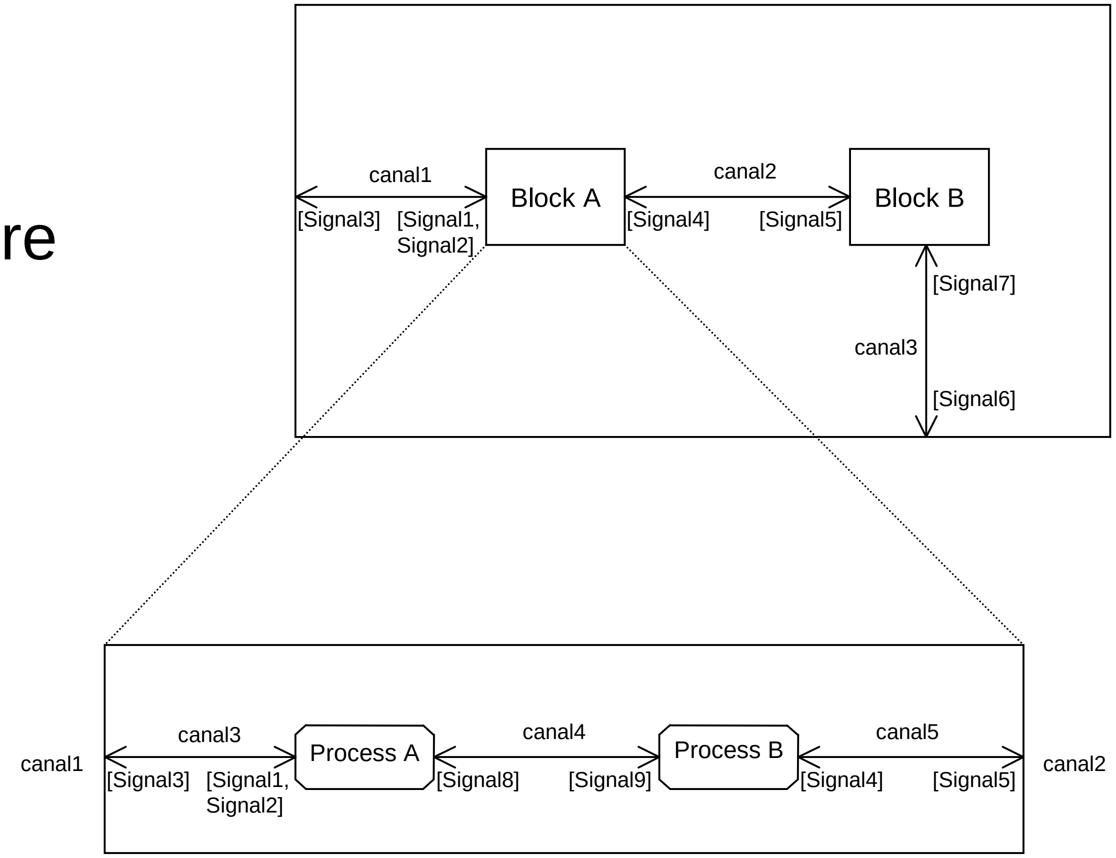


SDL-RT: class relations

- A class diagram only defines the architecture for the static part of the system.
- SDL-RT defines semantic related to cardinality of aggregation or composition:
  - *
  - 1
  - N

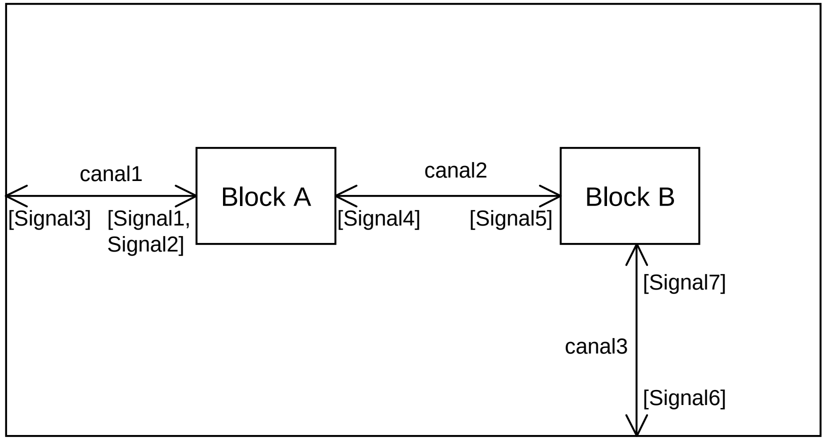
SDL-RT: architecture

Dynamic architecture
and
Communication



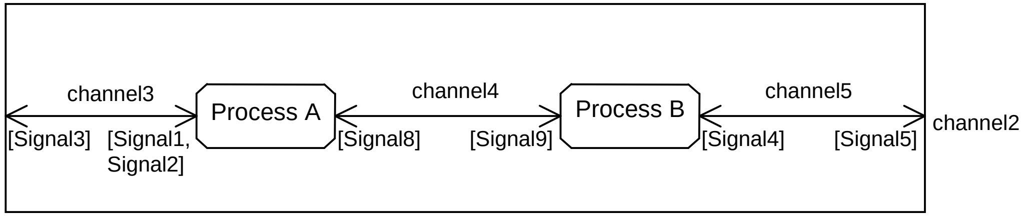
SDL-RT: architecture

- A block is a high level functional entity.
- It is used to organize and architecture the system.
- Block architecture should not change through the development process
- Block are usually not implemented as an execution entity



SDL-RT: architecture

- Lowest level of the architecture is the process
- A process is an execution entity
- Processes execute concurrently
- It has an implicit message queue
- Message based communication is asynchronous



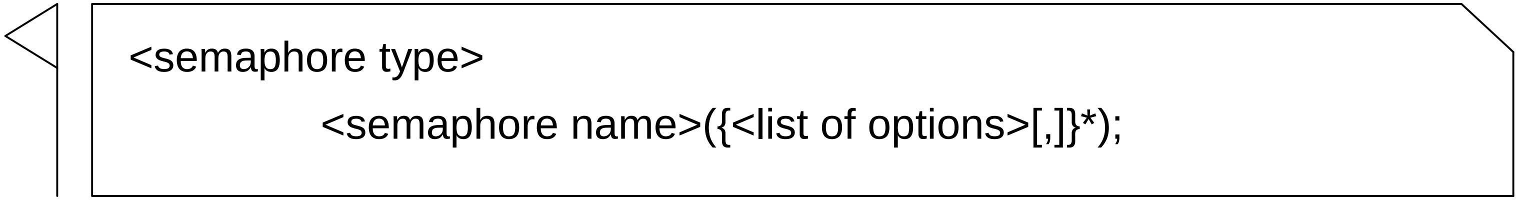
SDL-RT: communication

- Message
  - A message is defined by its name
  - A message can have 0 or several parameters
  - Parameters types are any C or C++ types
  - Messages can be gathered in lists
  - Messages can be defined at any level of the architecture

```
MESSAGE myFirstMsg, mySecondMsg(char, int *);
MESSAGE myThirdMsg(MyStruct *);
MESSAGE_LIST myMessageList=myFirstMsg,mySecondMsg;
MESSAGE_LIST anotherMsgList=(myMessageList),myThirdMsg;
```

SDL-RT: Synchronization

- Semaphore
  - A semaphore is a common resource
  - It can be accessed by any process
  - It may be defined in any agent

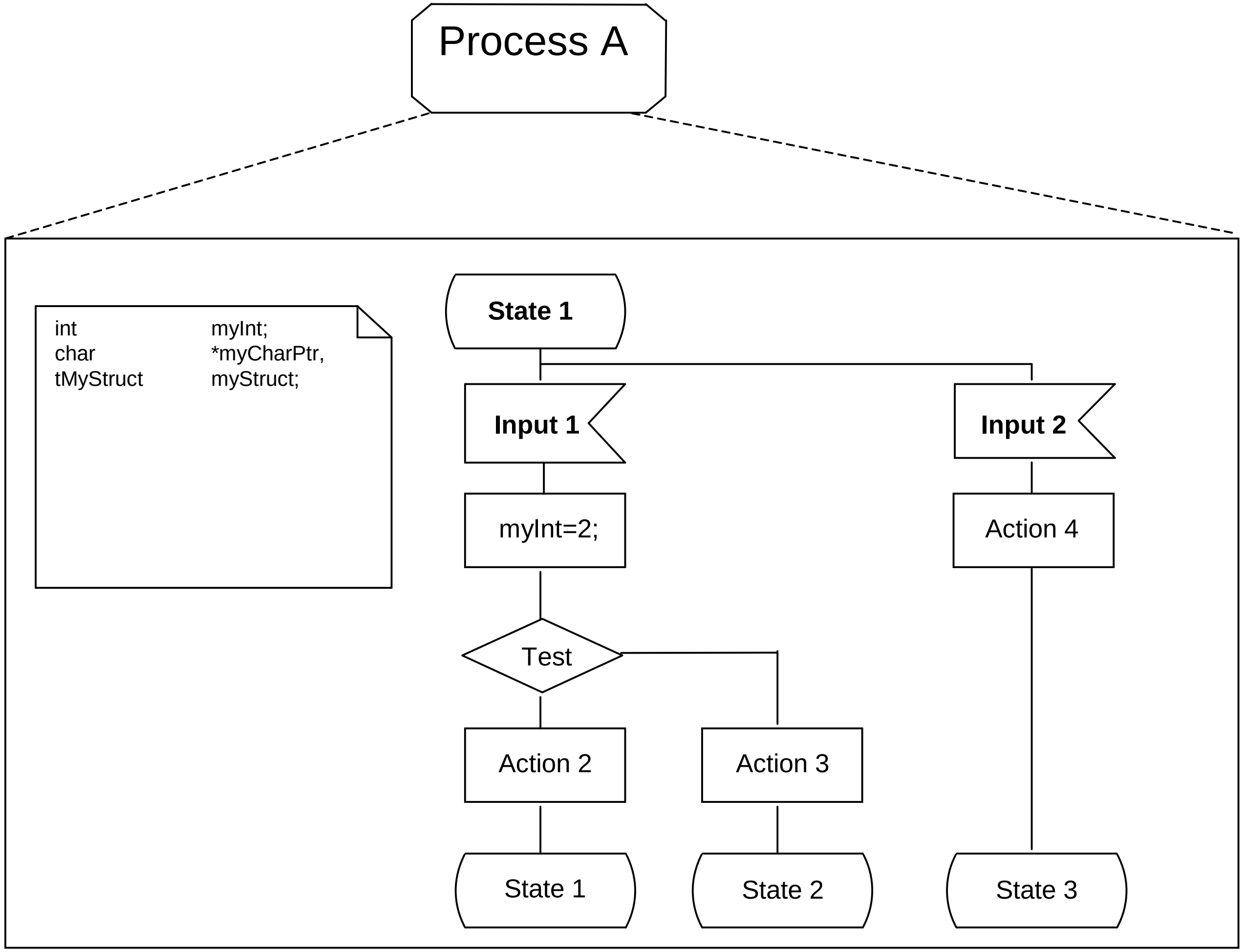


<semaphore type>

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

SDL-RT: behavior and data

- A process behavior is based on a graphical finite state machine
- Data types are C or C++



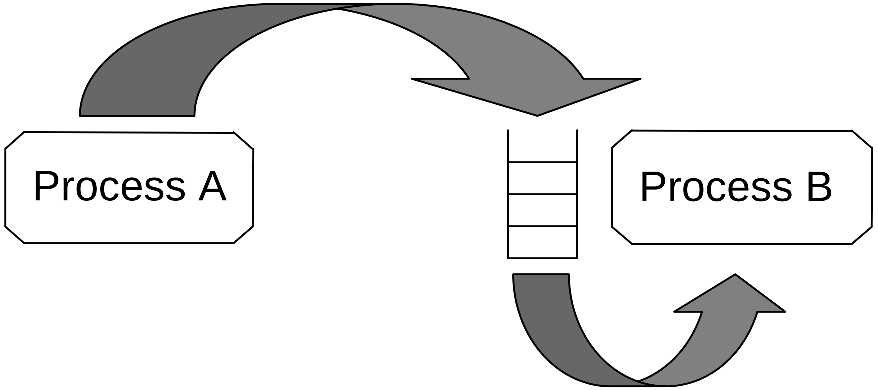
```
int              myInt;
char             *myCharPtr,
tMyStruct        myStruct;
```

SDL-RT: data types

- Data types are C or C++, declared in .h files or in the agent diagram
- Variables can be global to the whole system, declared at system level and defined in an external C file
- Variables can be local to a process, defined in the process diagram
- Variables declared in blocks are automatically visible in the whole underlying architecture of the block.

Process: message queue

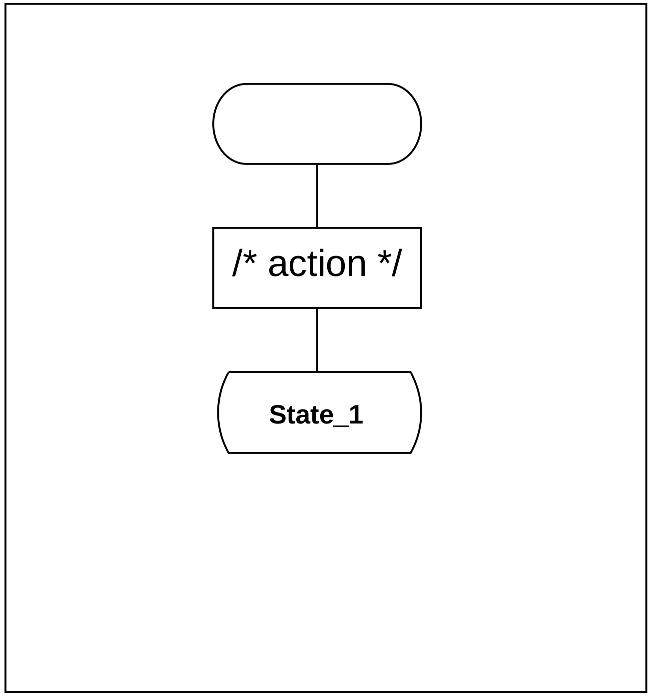
- A process has an implicit message queue
- The message queue is a FIFO
- Process should not send messages to themselves; in some very special cases it might be a solution
- A timer going off is a message in the queue



Process: initial transition

Execution entry point of the process.

- Message output
- Task creation
- Timer start



Process: state

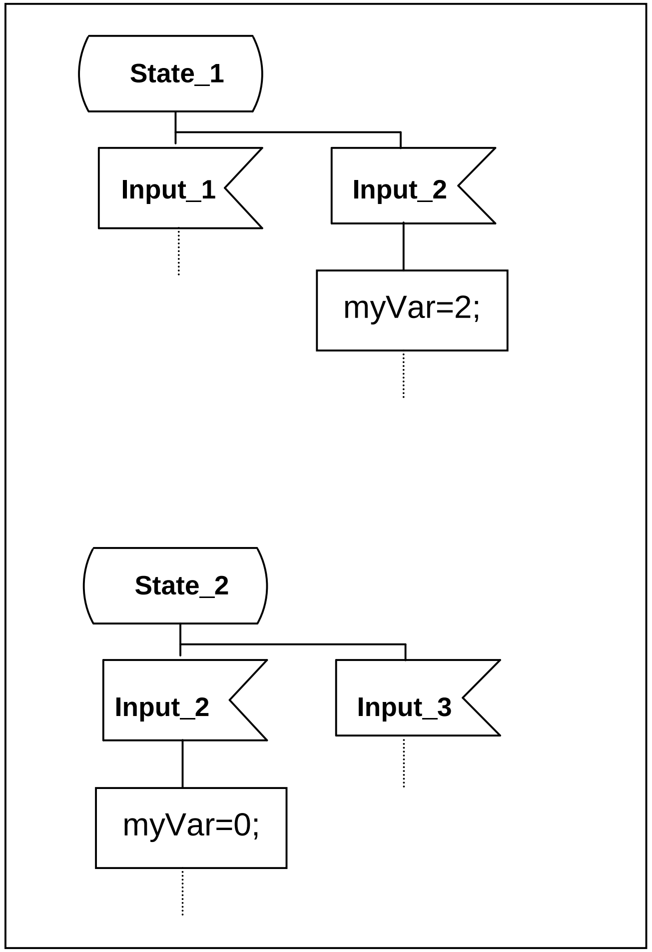
A state means the same trigger will generate a different behavior. A trigger can be:

- Message input

  Read from the message queue with or without parameters

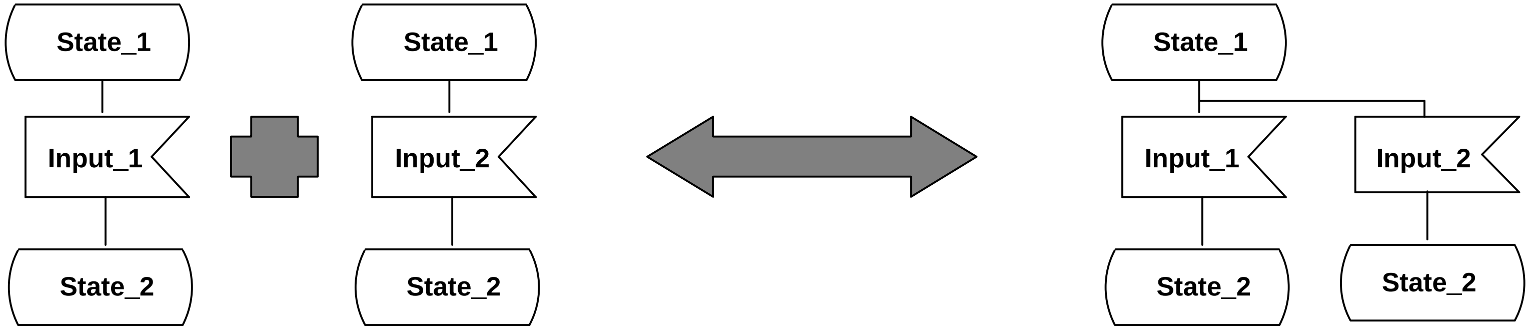
- Continuous signal

  A continuous signal is a condition to check when reaching the state. It is checked before the message queue. Continuous signals have associated priorities



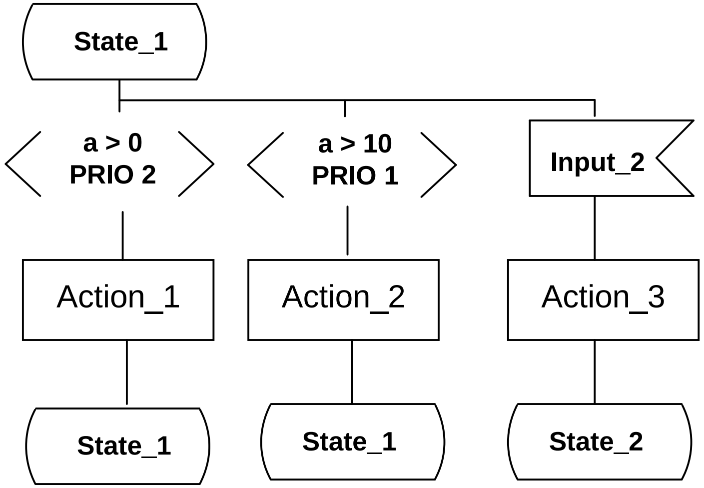
Process: state

- The same state can be described with several symbols
- At the end of a transition, the process goes to a next state. The same state symbol is used.



Process: continuous signal

- Continuous signals have priorities
- Whatever the priority, they will be evaluated before the messages



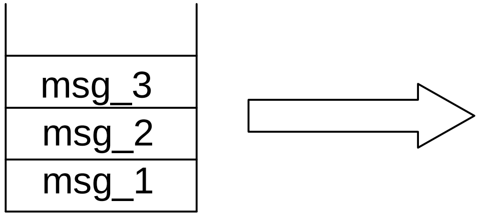
Considering the process gets into State_1 with a =15 and Input_2 is in the queue

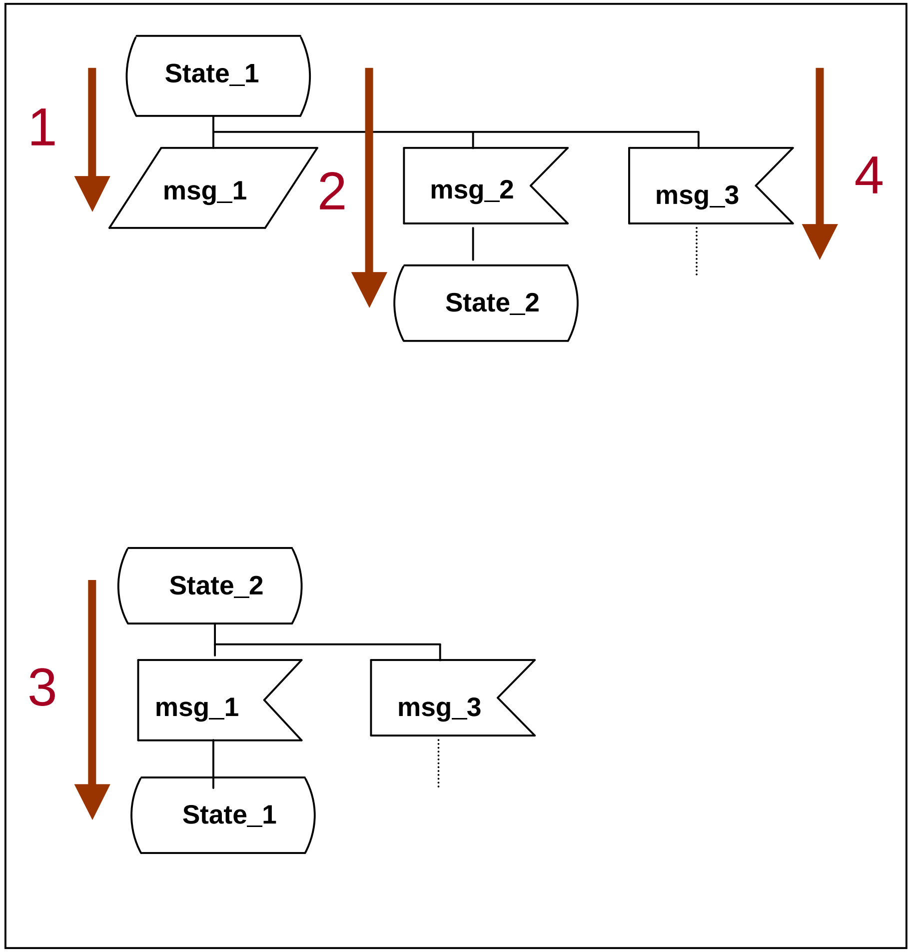
Resulting behavior will be:

- Action_2
- Action_1
- Action_3

Process: message save

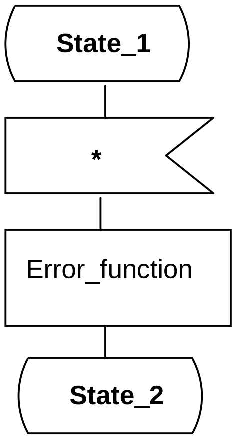
- Messages received while unexpected are thrown away. It is considered normal behavior.
- A message can be saved to be treated when the finite state machine reaches a new state. Then the first saved message will be treated before the other messages.



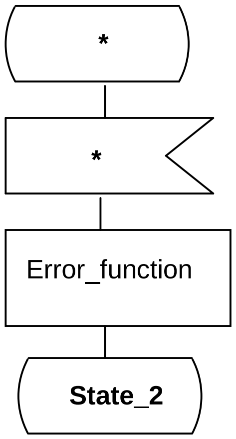


Process: unexpected messages

- Unexpected messages can be treated with the ‘*’ message representing a default behavior.
- All cases can be treated in the ‘*’ state.



All unexpected messages in state will go through the same error function

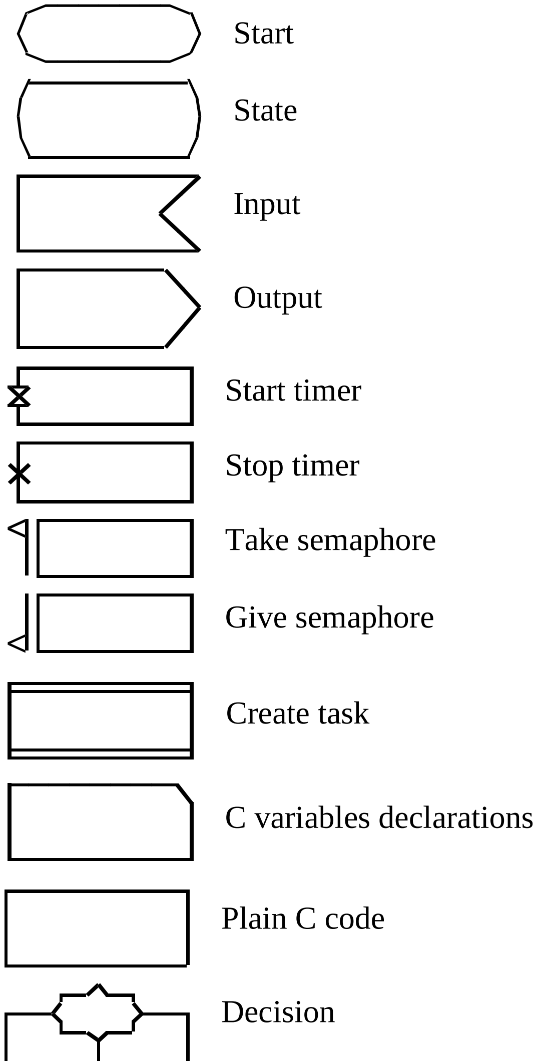


All unexpected messages whatever the state is will go through the same error function

Process: transitions

Actions that can be done in a transition are:

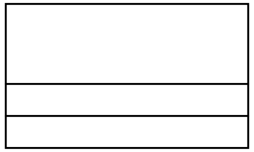
- Send out a message
- Start a timer
- Cancel a timer
- Take a semaphore
- Give a semaphore
- Create a task
- Call a procedure
- Execute any C or C++ code
- Evaluate an expression
- Connect to another branch of code



Process: transitions

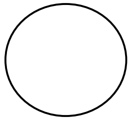
Procedure declaration

Procedure call

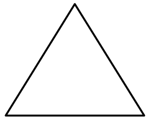


Object creation

Continuous signal



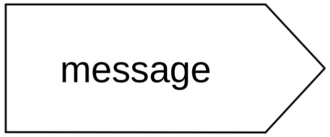
Connector



Transition option

Process: message output

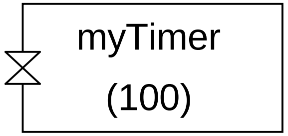
- Messages are sent:
  - TO_NAME <process name>
  - TO_ID <process id>
  - TO_ENV <macro name>
  - VIA <gate or channel name>
- Parameters are copied (shallow copy)
- <process id> special keywords:
  - OFFSPRING
  - SELF
  - PARENT



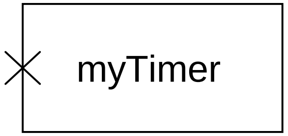
Message send

Process: timers

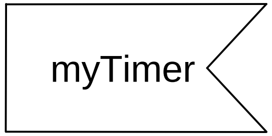
- Timers do not need to be declared.
- When a timer goes off it becomes a message in the queue like any other message. Messages already present will be treated first.
- If the timer is cancelled while the corresponding message is already in the queue; the message will be (virtually) removed from the queue.
- Timer is identified by its name.
- The timer message will have the same name.
- Time unit is system tick.



myTimer is started for 100 units of time (ticks)



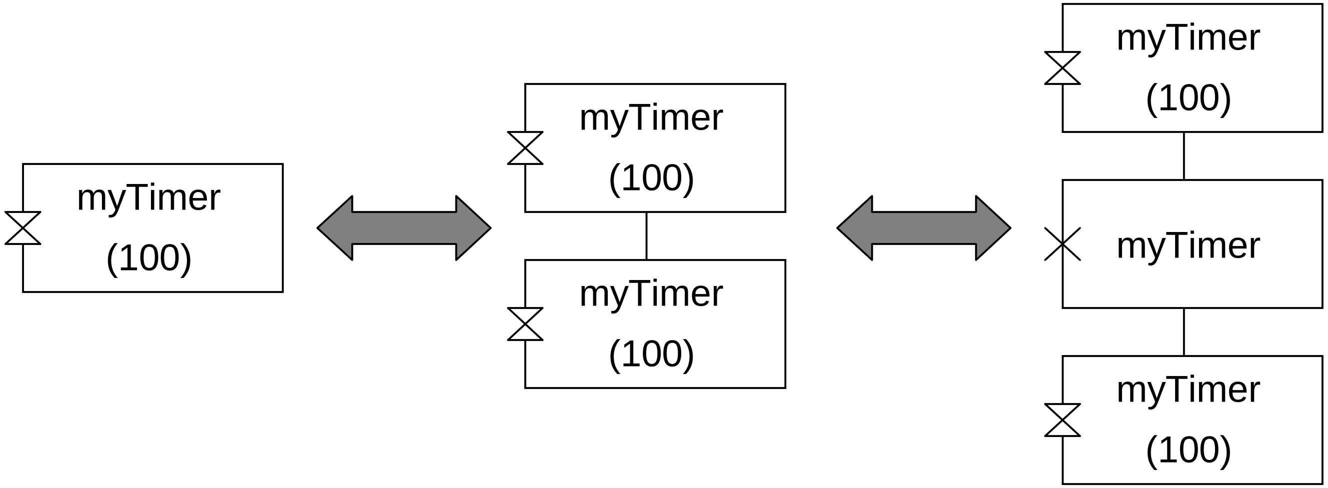
myTimer is cancelled



myTimer is received

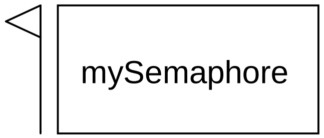
Process: timers

- The timer message can only be received by the process that started it.
- The timer goes off only once. There is no concept of repeat timer.
- A timer can only be started once. If restarted, the previous one is cancelled. That is ideal for checking a response has been received within a given amount of time.

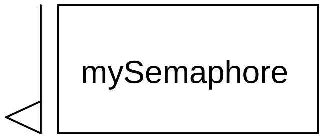


Process: semaphores

- Semaphore handling is RTOS specific
- Binary, Counting, and Mutex are available if the RTOS support them
- Sometimes binary are mapped to counting semaphores. RTOS integration pages should be checked.
- SDL-RT semaphores are identified by their names.
- SDL-RT semaphores are automatically created at startup.



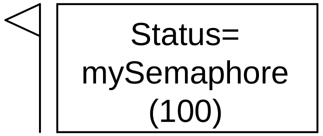
Semaphore take



Semaphore give

Process: semaphores

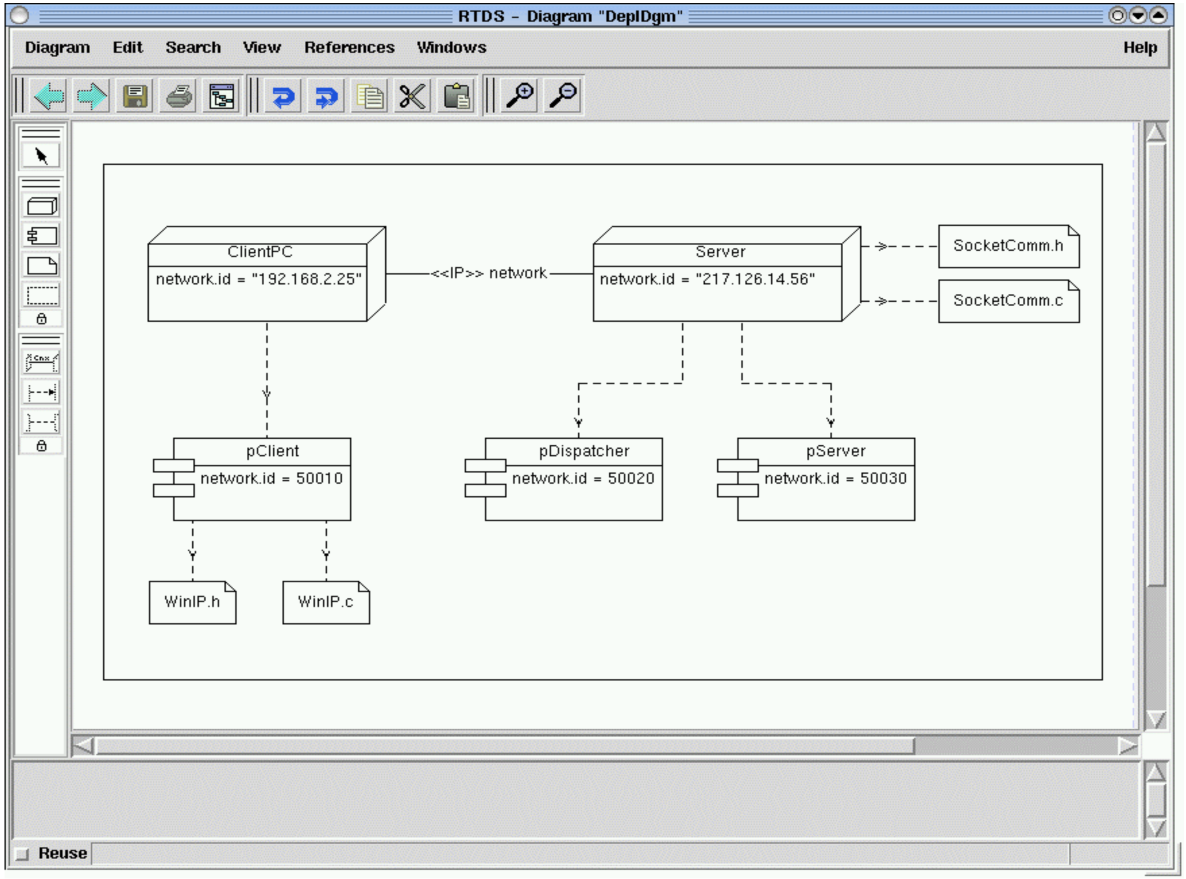
- Taking a semaphore is a blocking action if the semaphore is not available.
- Timeout values can be provided. After that time the transition resumes. Timeout is expressed in time unit (ticks). SDL-RT defines FOREVER and NO_WAIT keywords.
- The return value indicates if the take was successful or not. SDL-RT defines OK and ERROR as keywords.



```
if Status == OK
        Take succeeded
elif Status == ERROR
        Take failed
```

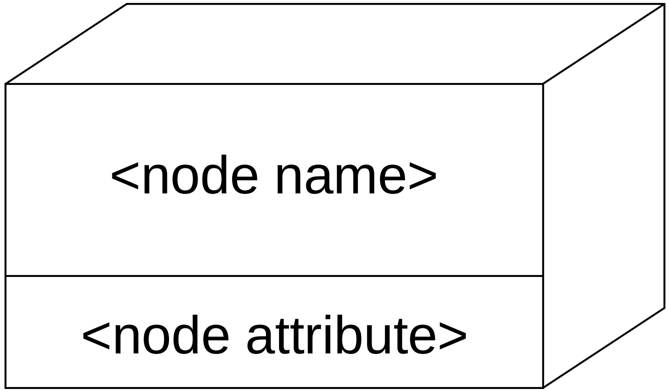
Deployment diagram

Physical deployment



Deployment diagram: node

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

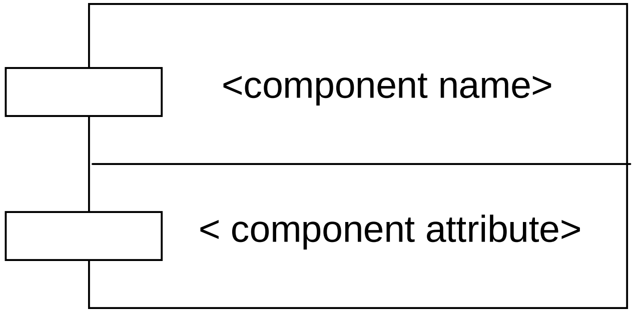


Deployment diagram: node

A **component** represents a distributable piece of implementation of a system.

There are two types of components:

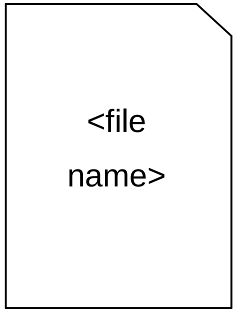
Executable component



<component name>

< component attribute>

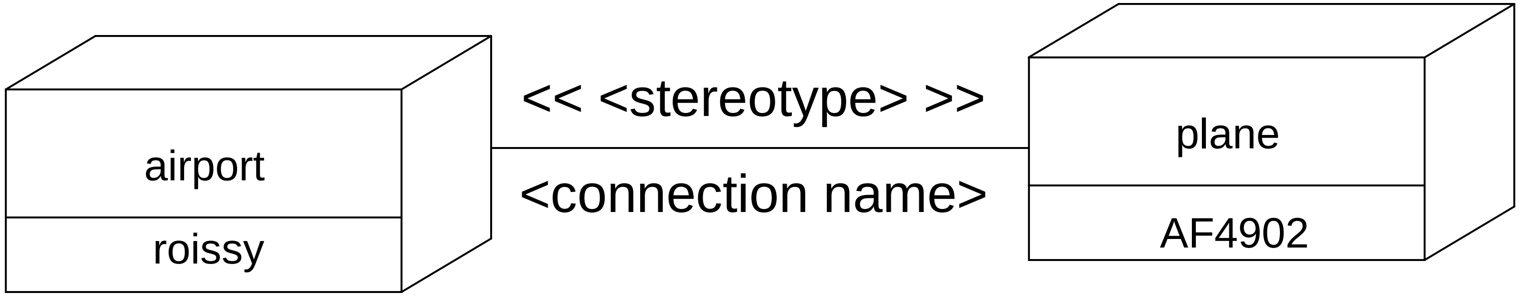
File component



<file name>

Deployment diagram: connection

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



airport

roissy

<< <stereotype> >>

<connection name>

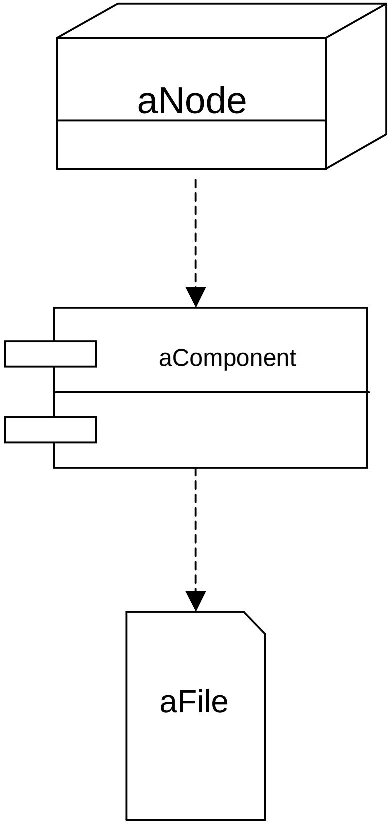
plane

AF4902

Deployment diagram: dependency

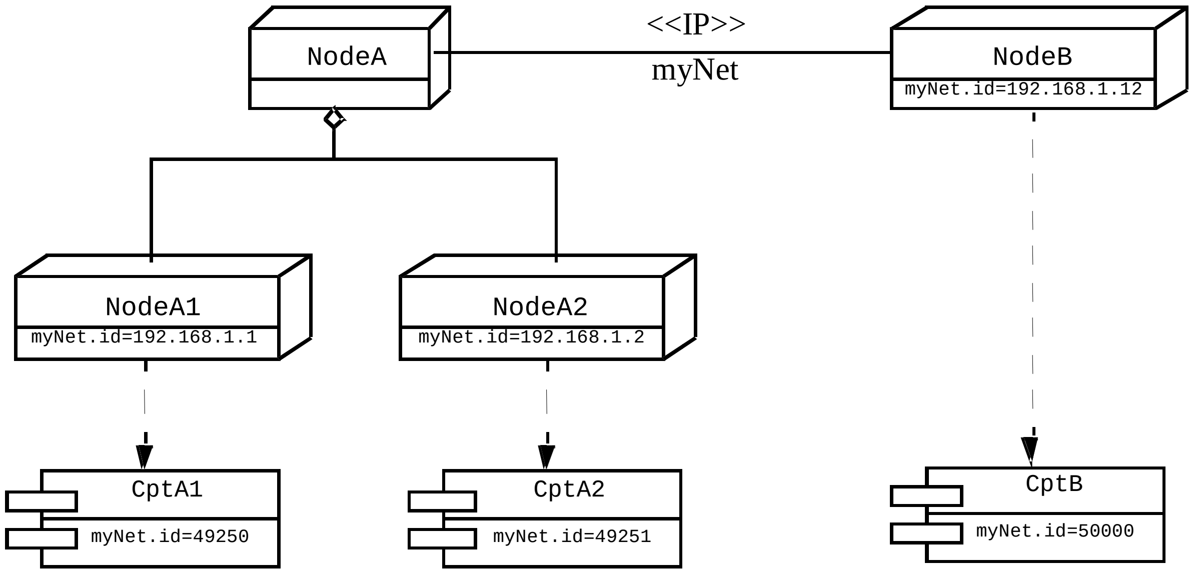
Dependency between elements can be represented graphically.

- 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.



Deployment diagram: identifiers

Aggregation: a node can be subdivided into nodes.



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

Object orientation

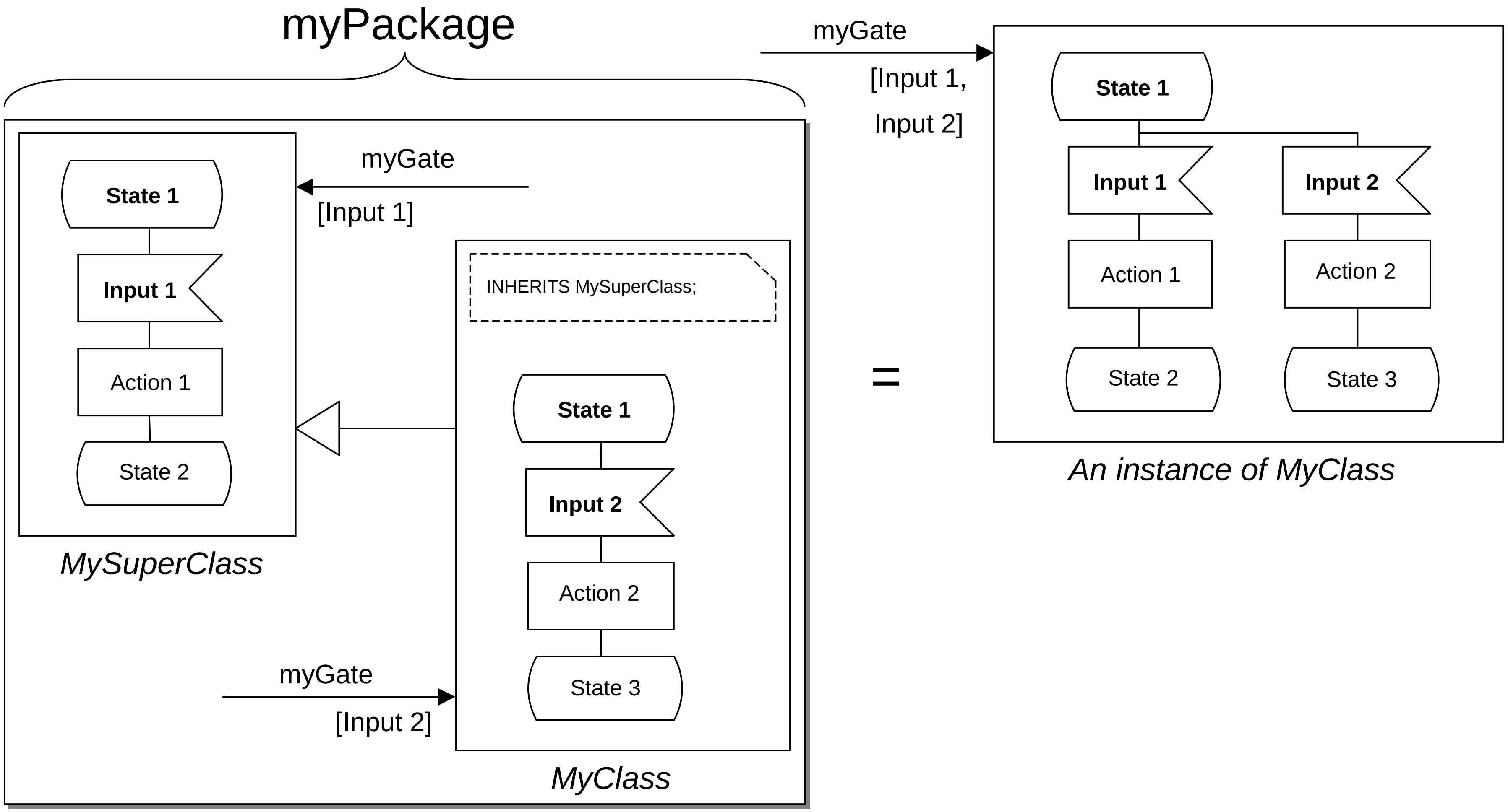
Class definition in applied to real time concepts:

- Block
- Process
- Transition
- Data declaration

OO properties:

- Inheritance
- Specialization
- Encapsulated data
- Abstract classes

Object orientation - example



myPackage

State 1

Input 1

Action 1

State 2

MySuperClass

myGate
[Input 1]

INHERITS MySuperClass;

State 1

Input 2

Action 2

State 3

myGate
[Input 2]

MyClass

=

myGate
[Input 1,
Input 2]

State 1

Input 1

Input 2

Action 1

Action 2

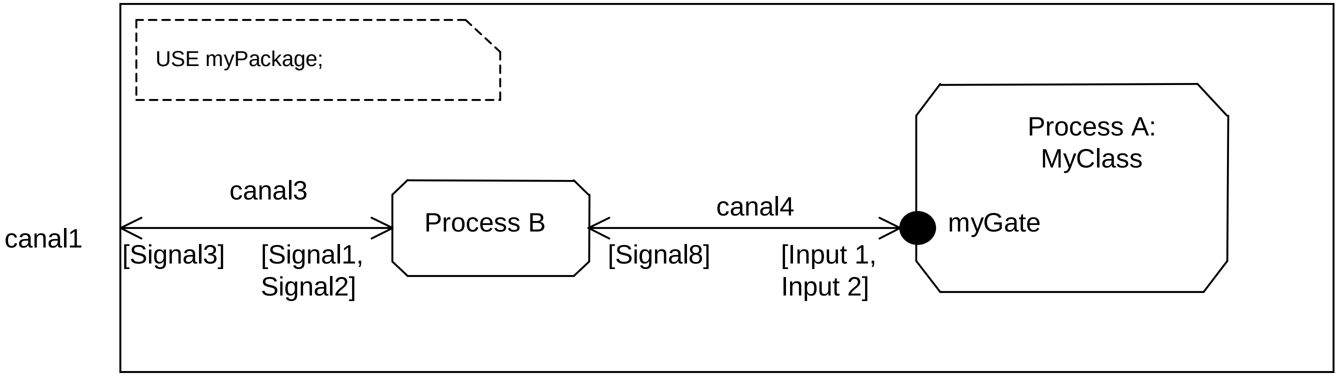
State 2

State 3

An instance of MyClass

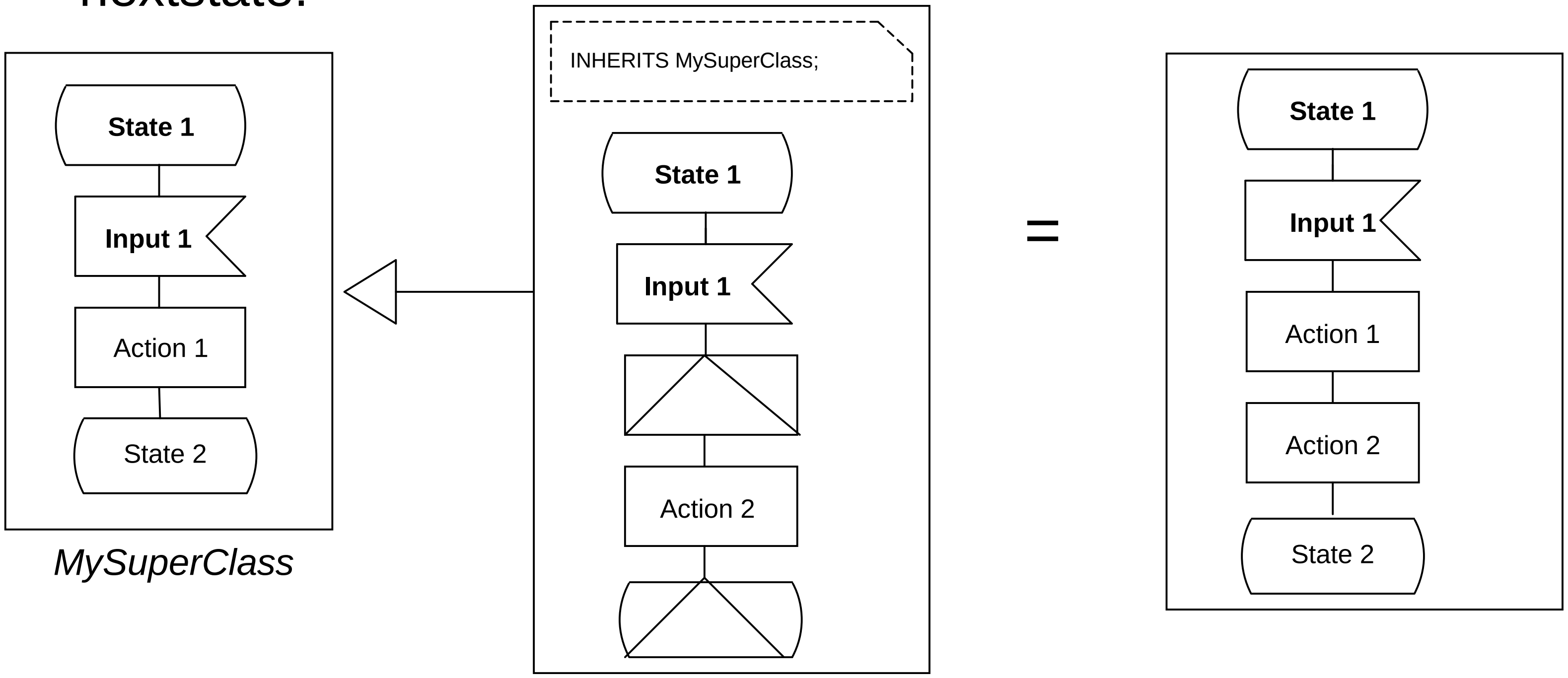
Object orientation – example ctd

The class is instantiated in the system architecture.



Transitions overloading

- Transitions defined in sub-classes may be overloaded in sub-classes
- Special symbols allow to call the super-class's transition body and/or nextstate:



State 1

Input 1

Action 1

State 2

MySuperClass

INHERITS MySuperClass;

State 1

Input 1

Action 2

MyClass

=

State 1

Input 1

Action 1

Action 2

State 2

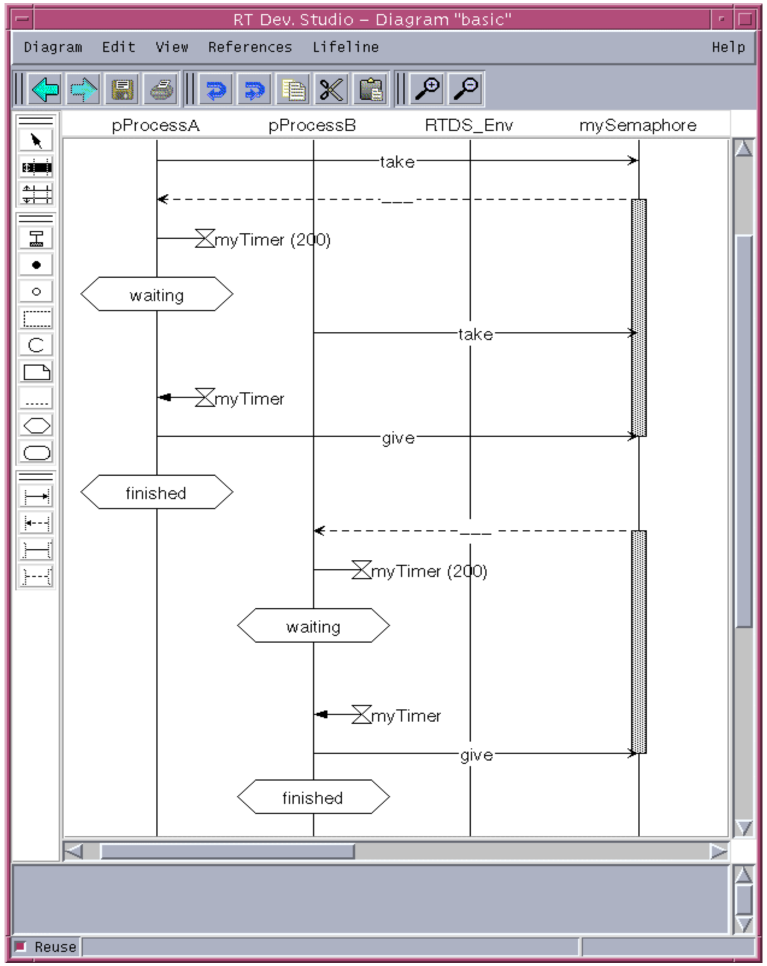
SDL-RT MSC: dynamic view

SDL-RT Message Sequence Chart

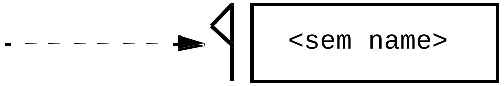
- Vertical lines represent a task, the environment, an object or a semaphore,
- Arrows with a stick arrowhead represent message exchanges, semaphore manipulations or timers,
- Arrows with a filled solid arrowhead represent synchronous operation call

Can be used:

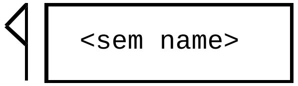
- As specification
- Execution traces



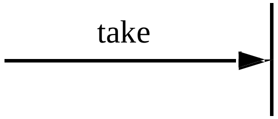
SDL-RT MSC: symbols 1/2



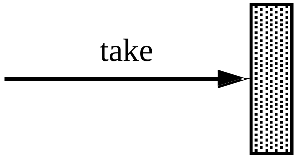
Semaphore creation from a known process.



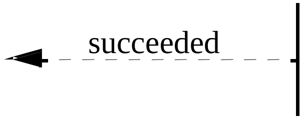
Semaphore creation from an unknown process.



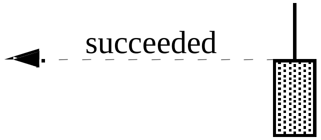
Semaphore take attempt.



Semaphore take attempt on a locked semaphore.



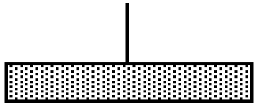
Semaphore take successfull but semaphore is still available.



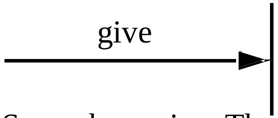
Semaphore take successfull and the semaphore is not available any more.

SDL-RT MSC: symbols 2/2

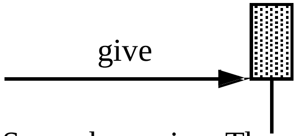
Semaphore take timed out.



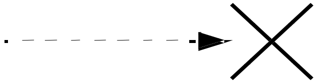
Semaphore continues.



Semaphore give. The semaphore was available before the give.



Semaphore give. The semaphore was unavailable before the give.



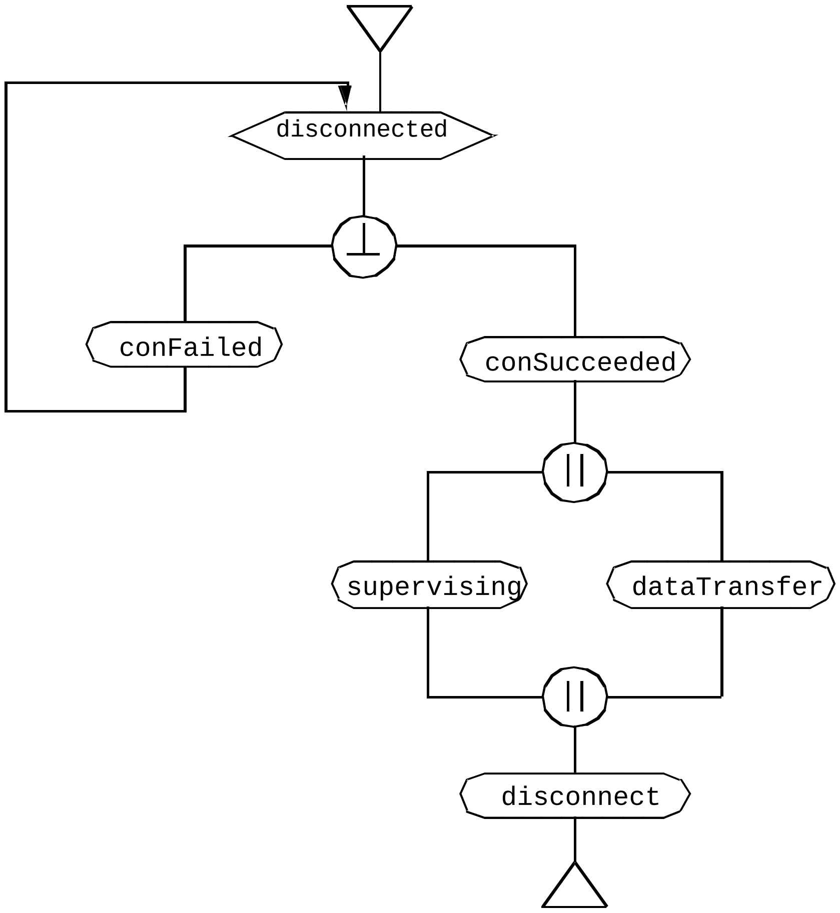
Semaphore is killed by a known process.

Semaphore is killed by an unknown process.

HMSC: dynamic overview

High level Message Sequence Chart

- Sequence of MSCs,
- Parallel independent execution of MSCs.



SDL-RT: XML format example

DTD is provided with SDL-RT specification

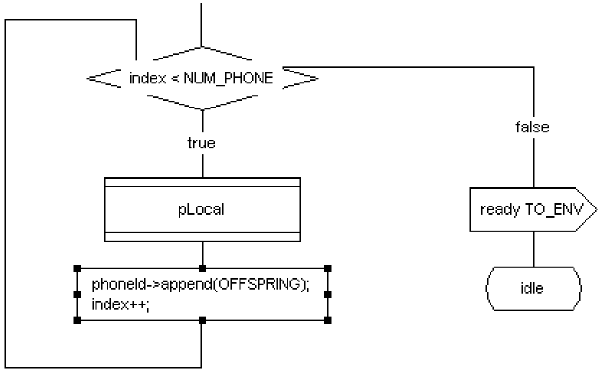
```
<Symbol symbolId="SYMB5" type="sdlTask" xCenter="182"
yCenter="454" color="#000000" fixedDimensions="TRUE"
width="174" height="36">

<Description></Description>

<Text>phoneId-&gt;append(OFFSPRING);

index++;</Text>

</Symbol>
```



```
phoneId->append(OFFSPRING);
index++;
```

Questions