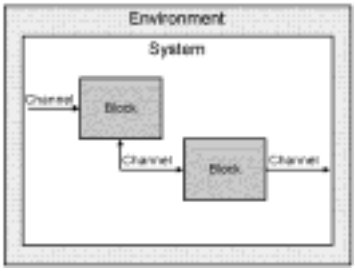
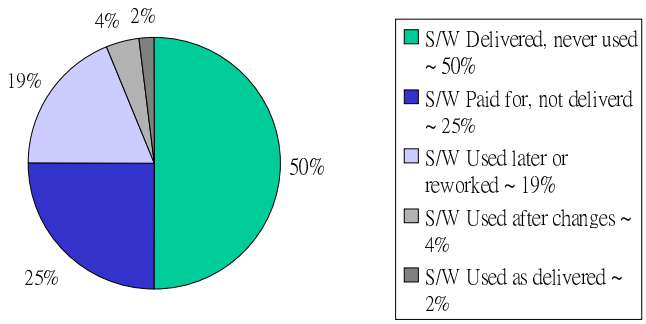
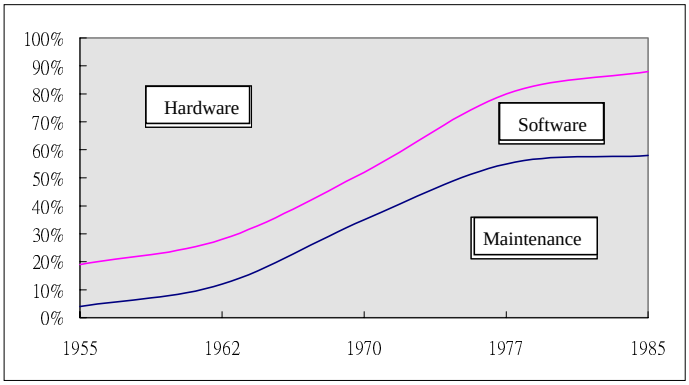
Introduction to SDL



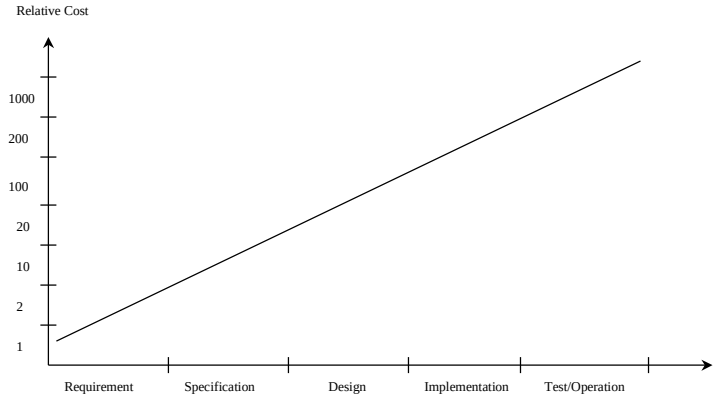
Statistics from 9 federal S/W projects



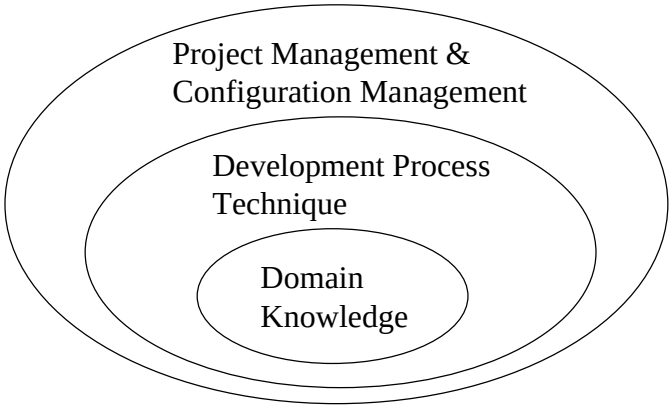
S/W Costs



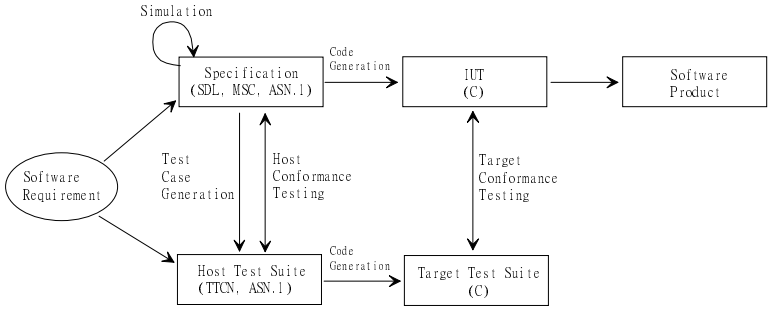
Error Correction



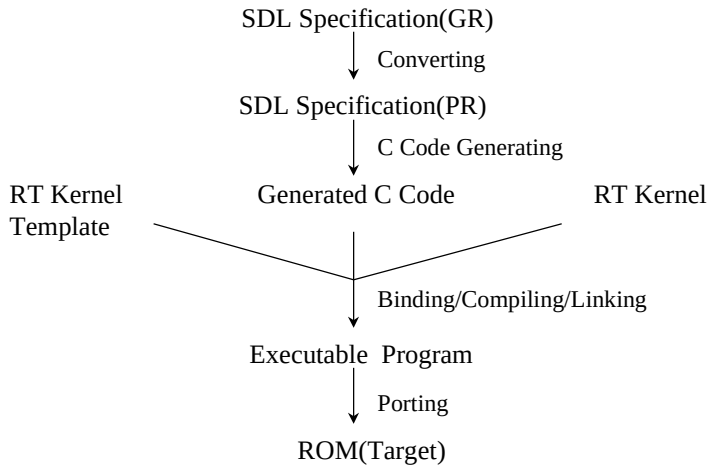
S/W Project Related Knowledge



A Formal Protocol Development Process



Procedure for Building an SDL System



Why SDL

- Graphic Representation
- Hierarchical Representation
- Simulation/Validation Capability
- Other reasons:
  - International Standard(ITU-T, Z.100).
  - Formal description.
  - Easy to understand even for non constructors (graphical representation).
  - Object Oriented language (supports object oriented thinking and reuse of specifications).

Good Specification

- Enables an easier communication between customer and producer.
- Makes it easier to discover errors in an earlier stage of the life cycle.
- Reduces maintenances costs.

Why formal?

The representation technique is formal if the interpretation model is formally defined and guarantee that no ambiguities can occur
i.e.
an ambiguity based on the interpretation is an error and can automatically be detected.

What is SDL ?

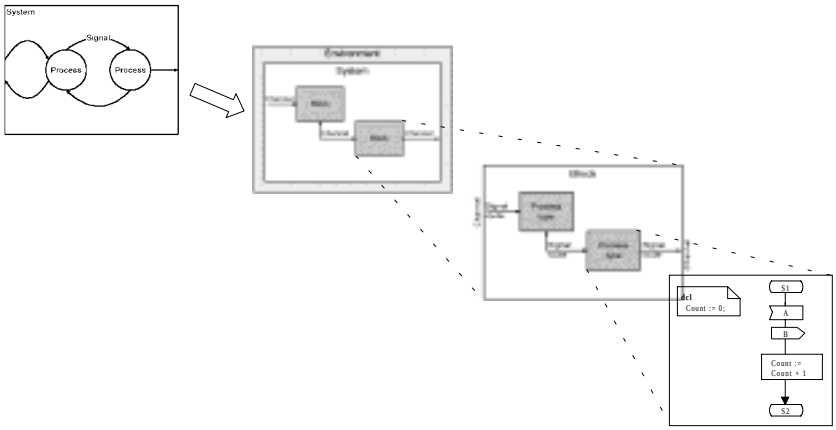
- SDL (Specification and Description Language) is a standard language for the specification and description of systems. It has been developed and standardized by ITU (International Telecommunication Union, formerly CCITT).
- The development of SDL started in 1972 after a period of investigations. The first version of the language was issued 1976, followed by new versions 1980, 19N and 1988.
- SDL88, was approved 1987.
- SDL92, was defined by Z.100.
- Future goal: SDL 2000.

Application of SDL

- type of system:
  - real time, interactive, distributed,
- type of information:
  - behavior and structure,
- level of abstraction:
  - overview to details.
- SDL is widely used in telecommunication field, however, it has broader application area.

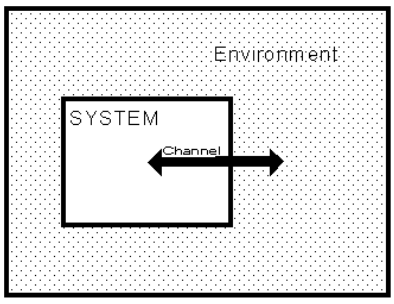
Overview of SDL

- Multi-level description, combined with finite state machine.



SDL System

- The system description constitutes the top level of detail.
- The system is what the SDL description specifies: an abstract machine communicating with its environment.



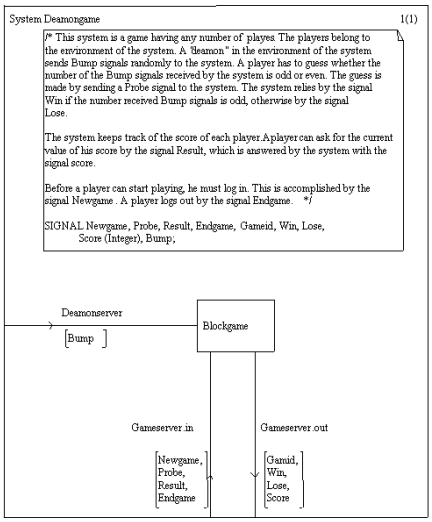
SDL System

- The system contains everything to be specified, but nothing not to be specified It communicates with its environment via channels.
- The communication mechanisms used between the system and the environment are the same as those used inside the system.

A Typical System Contains ...

- system name (Daemongame);
- signal descriptions (for the types of signals interchanged between the blocks of the system or between the blocks and the environment; Newgame, Probe, etc);
- channel descriptions (for the channels connecting the blocks of the system to one another and to the environment of the system; Daemonserver, etc.);
- data type descriptions (for the user defined data types, visible in the whole system and its environment; there are no user defined data types);
- block descriptions (for the blocks into which the system is partitioned; the system contains only one block: Blockgame).

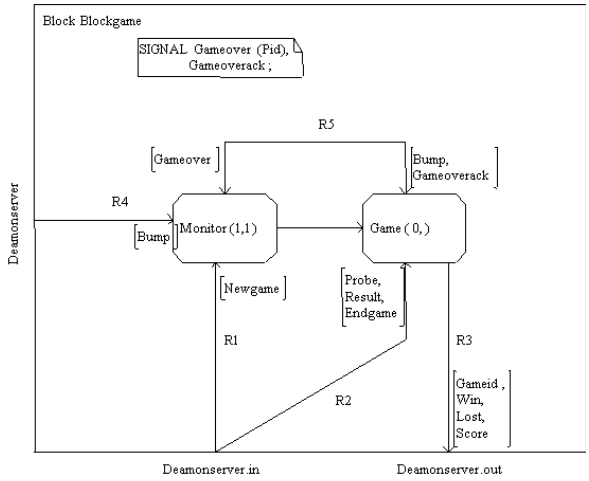
System Deamongame



Block

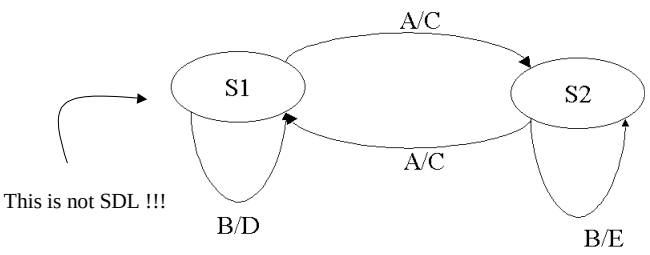
- A block is a part of the system- that can be treated in various respects (development, description, understanding etc.) as a self-contained object.
- A block diagram usually contains the following elements:
  - **block name** (in Blockgame);
  - **signal descriptions** (for the signals local to the block, i.e. not visible outside the block; Gameover, Gameoverack);
  - **signal route descriptions** (for the signal routes connecting the processes of the block to one another and to the environment of the block; R1, R2, etc.);
  - **channel-to-route connections** (the specification of the connections between the channels external to the block and the signal routes internal to the block; l Daemonserver to R4, etc.):
  - **process descriptions** (for the process types that describe the behaviour of the block; Monitor, Game).

Blockgame



Process

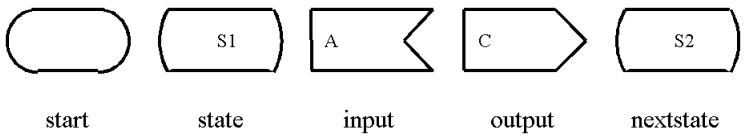
- A process in SDL is an *extended finite state machine*, i.e. a finite state machine that can use manipulate data stored in variables local to the machine.
- The behavior of a finite state machine is described by states and transitions.
- The behavior of a finite state machine may also be defined by a directed graph.



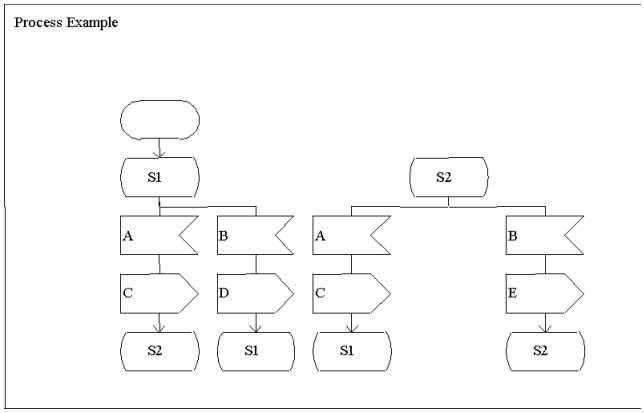
Elements of a Process

- A process diagram usually contains the following elements:
  - Process name,
  - formal parameters,
  - variables descriptions,
  - timer descriptions,
  - procedure description, and
  - process graph (for the description of the finite state machine of the process).

Basic constructs for the description of a process

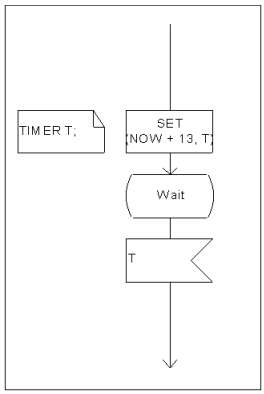


A Process Example



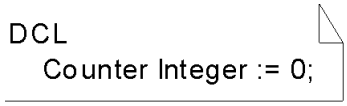
Timer

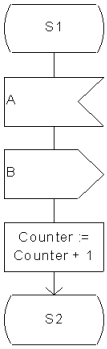
- The timer is an object, owned by a process, that is able to generate a timer signal and put this signal into the input queue of the process.
- During a transition a timer can be activated with the set construct. The set construct has two arguments. First one is the absolute time for the expiration of the timer, and the other one is the name of the timer.
- For the specification of the expiration time, the expression NOW (of the predefined type Time, which is similar to Real) can be used. NOW always gives the current time during the interpretation of the system description.



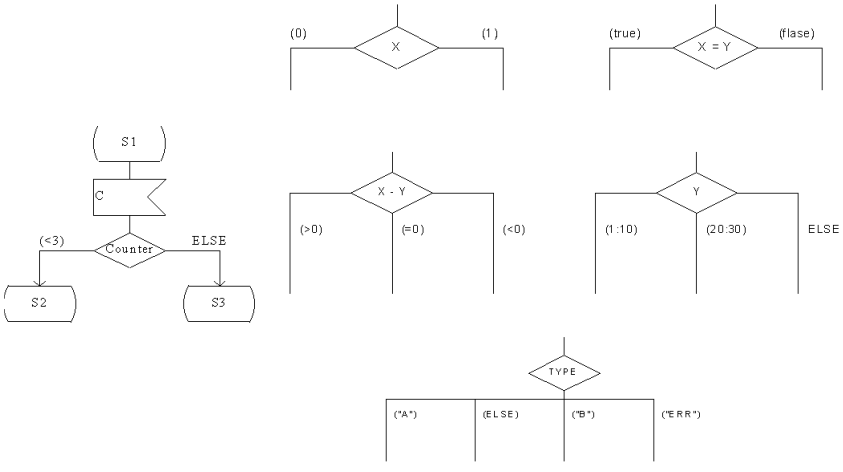
Data

- The description of a variable is following the keyword DCL.
- During a transition the process can use and manipulate its own local variables, using the task construct. A task construct is always an assignment. In SDL/GR the task construct is represented by a task symbol, which is a rectangle.



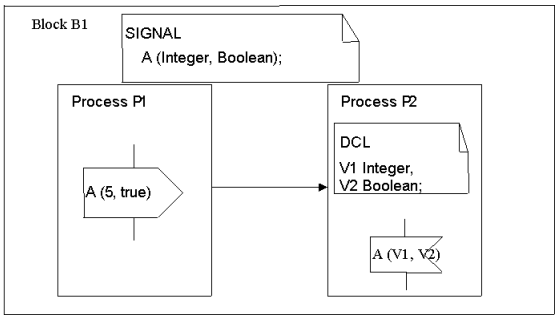


Decision



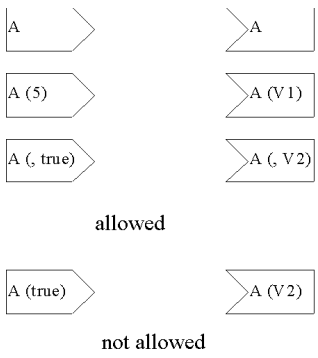
Single I/O

- With the signal A, the values 5 and true are sent from process P1 to process P2. The process P2 owns variables v1 and v2 of type Integer and Boolean, respectively. When signal A is consumed by process P2, the value 5 is assigned to v1 and true is assigned to v2.



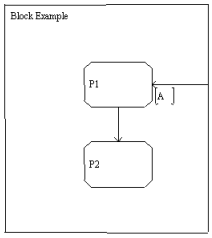
Legal I/O

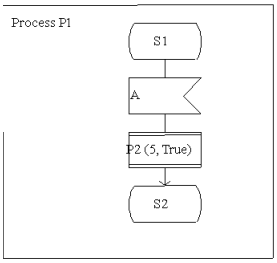
- One or more variables may be omitted in the input construct The corresponding received values are lost in this case.
- Similarly, one or more values may be omitted in an output construct
- But still the order of the remaining values and variables is important.

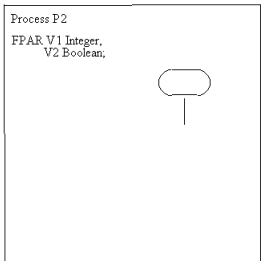


Dynamic Process Creation

- processes can be created by other processes dynamically at interpretation time. This is indicated in a block diagram by a dashed line from the creating process to the created process.

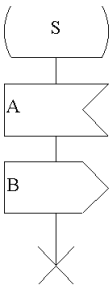




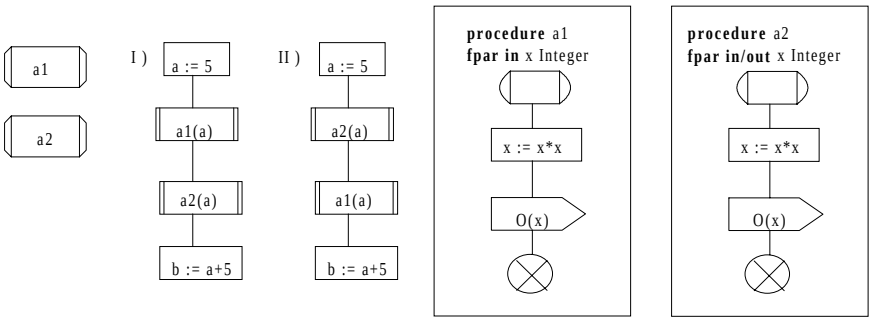


Process Termination

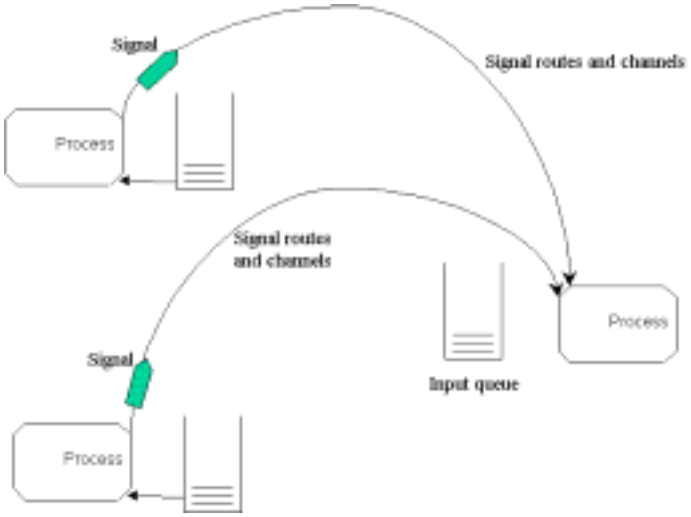
- A process can only be terminated by itself, by using the stop construct.
- Afterwards, all the data values and the contents of the input queue are discarded



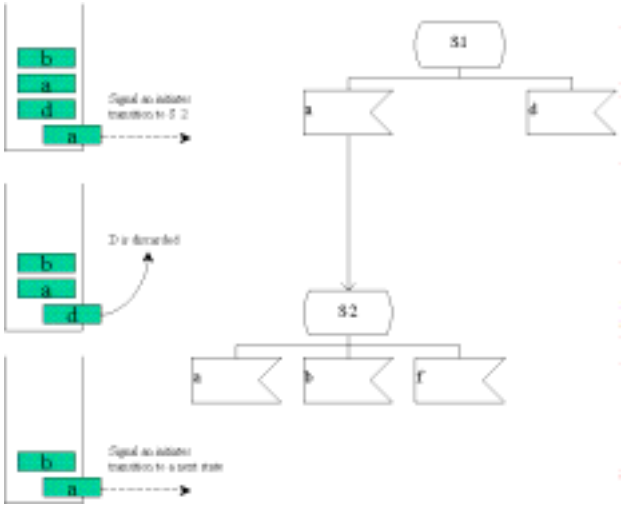
Procedure



Process Communication

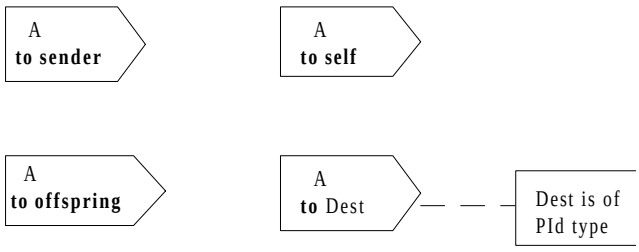


Process Communication Example



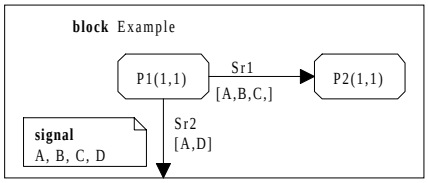
Explicit Addressing

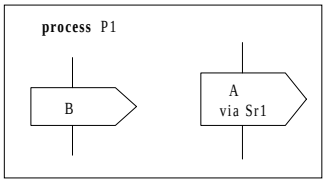
- four predefined expressions of type PId, i.e. **self**, **sender**, **offspring** and **parent**



Implicit Addressing

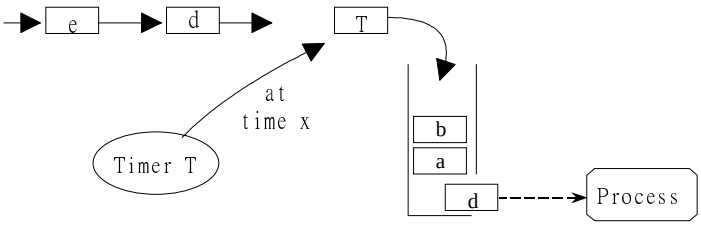
- a unique destination process may be specified by
  -system structure
  -naming a signal route or channel



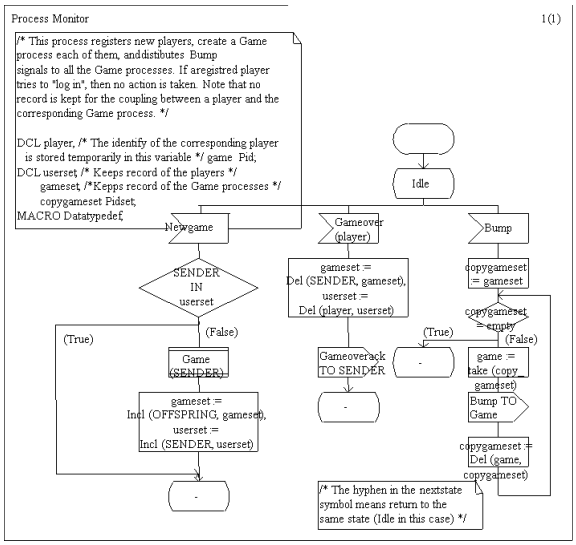


Timer Signal

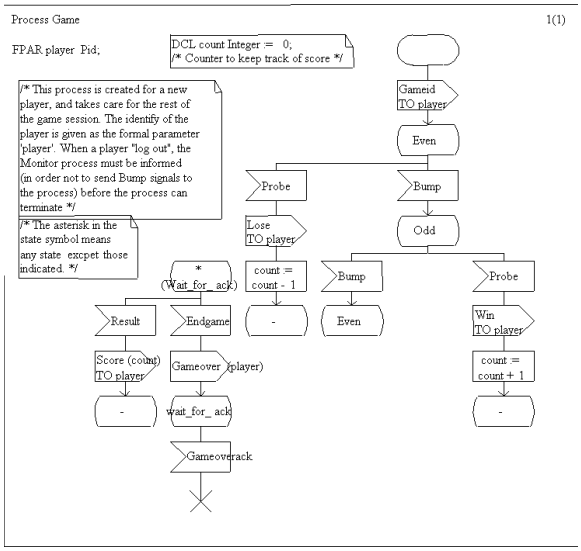
- with set(x,T), the timer signal T will be put into the input port of its process at time x
- timer signal might be removed from the input queue by using reset construct, e.g. reset(T)



Process Monitor



Process Game



SDL Predefined Data Sorts

- Integer
- Natural
- Real
- Boolean
- Character
- Charstring
- Pid
- Duration
- Time

Synonym

With Synonym, constants are declared in SDL.

If a constant is declared as EXTERNAL it means that the constant will be assigned first at system start-up time (i.e. it can be assigned different values each time.)

```
SYNONYM ZERO        INTEGER = 0;
SYNONYM ONE         INTEGER = 1;
SYNONYM NrOfDoors   INTEGER = EXTERNAL;
```

Newtype

- Newtype creates a new data type.

newtype *bool*
  literals *true, false* ;
  operators
    not:*bool* → *bool*;
  axioms
    not(*true*) == *false*;
    not(*false*) == *true*;
endnewtype;

Syntype

With Syntype you can restrict the set of values of a ground type. NOTE! The syntype will not be a new type but a subtype, and can thus be assigned values of the ground type.

```
SYNTYPE Age = Natural
   CONSTANTS 0:150
ENDSYNTYPE Age;
```

Example:

DCL MyAge, YourAge;

TASK
MyAge := 1,
YourAge := 100

Struct

Struct in SDL is similar to **struct** in C and **Record** in Pascal.

```
NEWTYPE Person STRUCT
   theName          Charstring;
   theAge           Age;
   theSex           SexType;
ENDNEWTYPE Person;
```

Example:

```
DCL   SomePerson Person;
      PersonAge Natural;

TASK
SomePerson := (. 'John', 26, Male .),
PersonAge := SomePerson!theAge
```


Generators in SDL

A generator is a parameterized data type template.

The generator defines the operators that should be available for all data types generated by the generator.

The instantiation of a generator is expanded before semantic analysis.

Predefined SDL generators:
- Array
- String
- Powerset

Generators in SDL

In SDL the following is predefined:

GENERATOR Array(TYPE Index, TYPE Itemsort);

The data type must be instantiated with an index type and an element type. The index type is often a syntype with a closed range.

```
SYNTYPE IndexType = Natural
   CONSTANTS 1:100
ENDSYNTYPE;

NEWTYPE PArrayType
   Array(IndexType, Person)
ENDNEWTYPE PArrayType;
```

Example:

```
DCL PersonArray ParrayType,
      I IndexType := 1;

TASK
PersonArray(i) := (. 'John', 26, Male .)
```

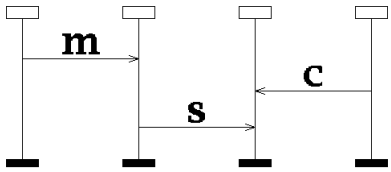

Unified Modeling Language

- Convergence of Booch, OMT, Objectory
- Standardize the artifacts of development
- UML is nonproprietary and open to all

UML Modeling Concepts

- Classes and relationships
- Grouping constructs
- Interaction mechanisms
- Behavior entities
- Extension mechanisms

Introduction to MSC



What is MSC ?

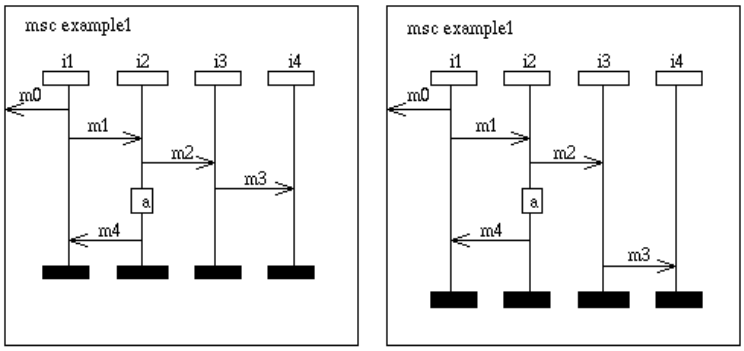
- A Message Sequence Chart is not a description of the complete behavior of a system, it merely expresses one execution trace.
- A collection of Message Sequence Charts may be used to give a more detailed specification of a system.

MSC Features

- A Message Sequence Chart contains the description of the asynchronous communication between instances.
- The complete Message Sequence Chart language, in addition, has primitives for local actions, timers (set, reset and time-out), process creation, process stop and coregions.
- Furthermore sub Message Sequence Charts and conditions can be used to construct modular specifications.

MSC Instances

- A Basic Message Sequence Chart contains a (partial) description of the communication behavior of a number of instances. An instance is an abstract entity of which one can observe (part of) the interaction with other instances



MSC Messages

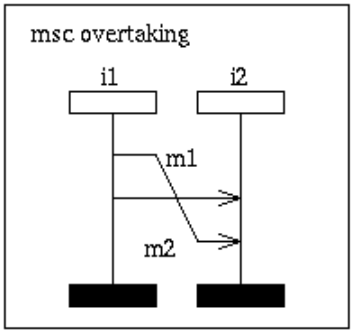
- A communication between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance.
- In the previous figure we consider the messages m1, m2, m3 and m4. Message m0 is sent to the environment. The behavior of the environment is not specified.
- For instance i2 we also define a local action a.

Timing of MSC

- Although the activities along one single instance axis are completely ordered, we will not assume a notion of global time. The only dependencies between the timing of the instances come from the restriction that a message must have been sent before it is received.
- In the figure this implies for example that message m3 is received by i4 only after it has been sent by i3, and, consequently, after the reception of m2 by i3. Thus m1 and m3 are ordered in time, while for m4 and m3 no order is specified. The execution of a local action is only restricted by the ordering of events on its own instance.
- The second Basic Message Sequence Chart in the figure defines the same Basic Message Sequence Chart, but in an alternative drawing.

Overtaking of Messages

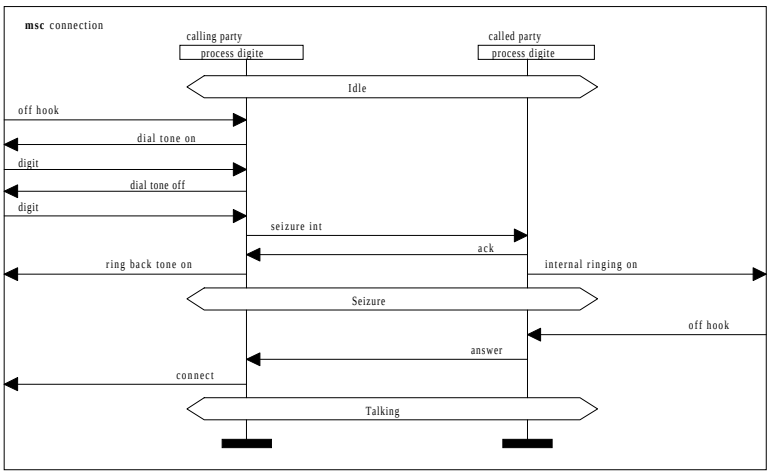
- It would even be possible to first send m3, then send and receive m4, and finally receive m3.



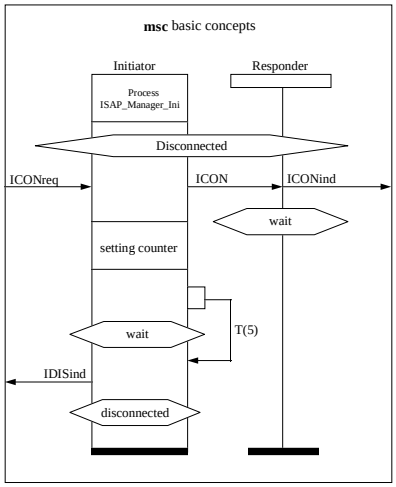
Textual notation

```
msc example1;
    instance i1;
      out m0 to env;
      out m1 to i2;
      in m4 from i2;
    endinstance;
    instance i2;
      in m1 from i1;
      out m2 to i3;
      action a;
      out m4 to i1;
    endinstance;
    instance i3;
      in m2 from i2;
      out m3 to i4;
    endinstance;
    instance i4;
      in m3 from i3;
    endinstance;
endmsc;
```

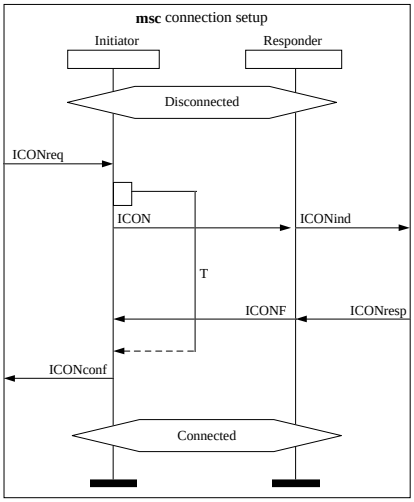

Instance, Message, Environment, Condition



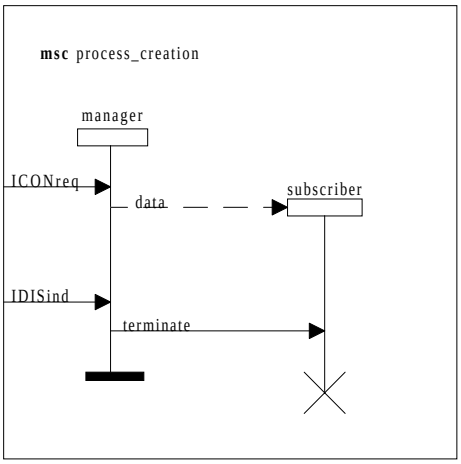
Action, Time Expiration



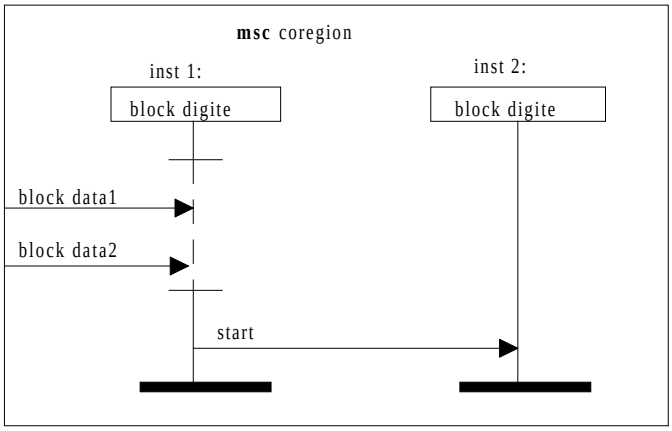
Time Supervision



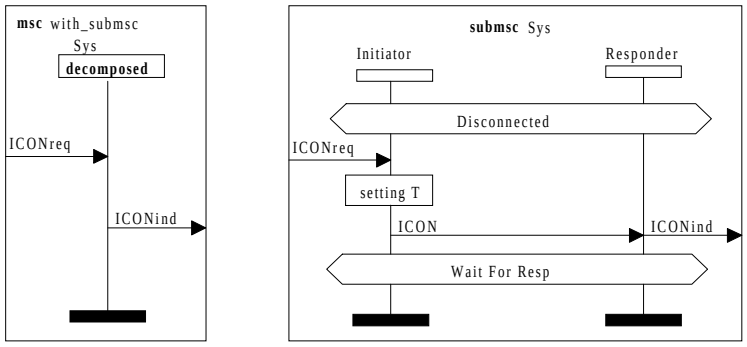
Instance Creation, Instance Stop



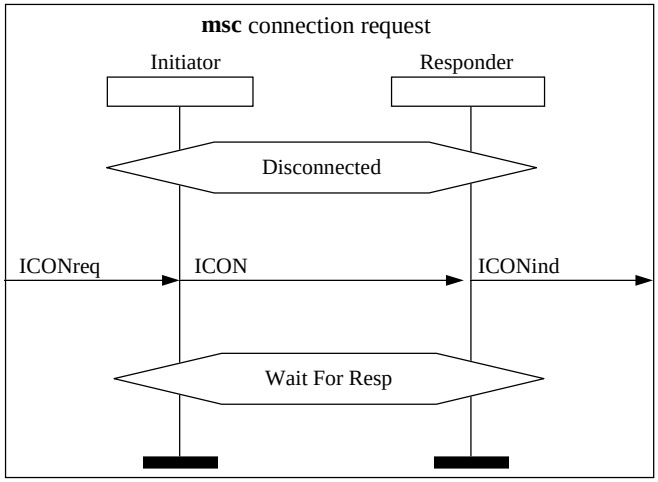
Coregion



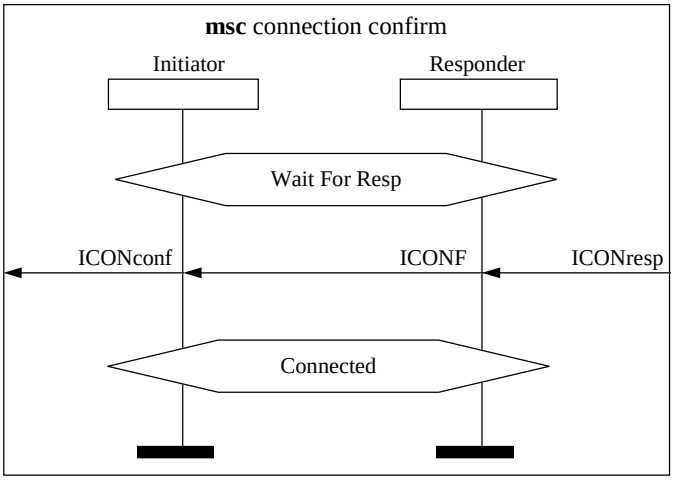
Sub Message Sequence Chart



Composition and Decomposition Rules



Composition and Decomposition Rules



Composition and Decomposition Rules