Embedded Software Engineering

Topic 4

Task-level Modeling of Embedded Applications

Task-level Modeling

- System models
- Task modelling in D(S)CS
- Task modelling in CoCS
- Task modelling in HCCS
- An overview on graphics notations and languages for CCS

System Modelling

- Context of analyzes
- Different models present the system from different perspectives
  - External perspective showing the system’s context or environment
  - Behavioural perspective showing the behaviour of the system
  - Structural perspective showing the system or data architecture
- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers

System Modelling

Structured Methods

Structured methods incorporate system modelling as an inherent part of the method

Methods define a set of models, a process for deriving these models and rules and guidelines that should apply to the models

System Modelling

Model Types

- Data processing model

  Showing how the data is processed at different stages

- Classification model

  Showing how entities have common characteristics

- Stimulus/response model

  Showing the system's reaction to events.

- Composition model

  Showing how entities are composed of other entities

- Architectural model

  Showing principal sub-systems

System Modelling

Context Models

Context models are used to illustrate the boundaries of a system:

- Social and organisational concerns may affect the decision on where to position system boundaries
- Architectural models show a system and its relationship with other systems

System Modelling

Process-Oriented Models

- A process-oriented model is based on a set of interacting processes (tasks) running under a real-time kernel or a static schedule.
- Process-oriented models emphasize the functional decomposition of real-time systems.
- Process-oriented models address naturally the problems of scheduling and schedulability analysis.

System Modelling

Object-Oriented Models

Object-oriented specifications emphasize structural decomposition.

System is conceived as a composition of interacting objects.

System Modelling

Behavioural Models

- ✧ Behavioural models are used to describe the overall behaviour of a system.
- ✧ Two types of behavioural model are shown here:
  - ➢ Computational models showing how data and signal transformations are organized
  - ➢ Control-flow models that show the systems response to events and time passage

Both of these models are required for a description of the system's behaviour.

System Modelling

Component-Oriented Models

Components are represented in the context of control systems as:

- ✧ Schedulable computational element having well defined behaviour and having a set of interfaces to communicate to its environment
- ✧ A main difference form object-oriented model is that the component is self-dependent and maybe it is a set of objects

Task modelling in D(S)CS

Types

- Automata models
- Symbolic execution of state machines –
  Programmable Logic Controller (PLC)
- State-logic execution –
  State Logic Controller (SLC)

Task modelling in D(S)CS

Types

- Automata models
- Symbolic execution of state machines –
  Programmable Logic Controller (PLC)
- State-logic execution –
  State Logic Controller (SLC)

Task modelling in D(S)CS

<u>Types</u>

- Automata models

- Symbolic execution of state machines – Programmable Logic Controller (PLC)

- State-logic execution – State Logic Controller (SLC)

Task modelling in D(S)CS

Types

- Automata models
- Symbolic execution of state machines –
  Programmable Logic Controller (PLC)
- State-logic execution –
  State Logic Controller (SLC)

Task modelling in D(S)CS

Automata Models

- Finite State Machines
- Petri Nets

Task modelling in D(S)CS

Finite State Machines

Automata Models

Finite State Machines

The theory of finite-state machines (FSM) is concerned with mathematical models which serve as approximations to physical or abstract phenomena.

FSMs are applicable to problems in many field of research from philology to communication.

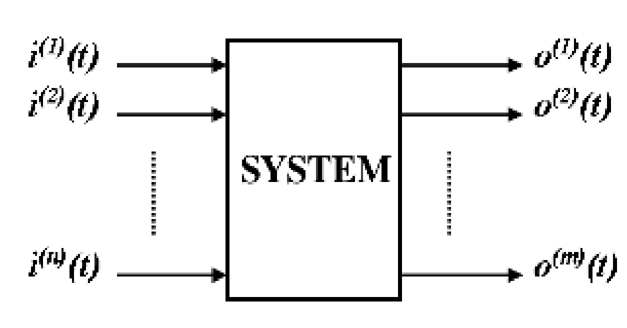
Now the unifying nature of the theory of FSM is of apparent value and it has been widely applied in wide research areas.

The significance of FSM’s theory is that its models are not confined to any particular scientific area.

Finite State Machines

Preliminary Definitions:

- ***State variables***, which represent the internal quantities of the system. While the input and output variables are system quantities which can be observed and measured, state variables are often immeasurable. These variables can also be denoted as intermediate variables .



Finite State Machines

<u>Definition:</u>

A **Finite State Machine** *M* is a 5-tuple

$$(I, O, S, R, r)$$

R(s, i, s′, o) means that for input ***i***, there is a transition from state **s** to state **s′** producing output ***o***. This is also denoted by

$$s \xrightarrow{i/o}_M s'$$

the initial state

(both

the output and transition relation

Finite State Machines

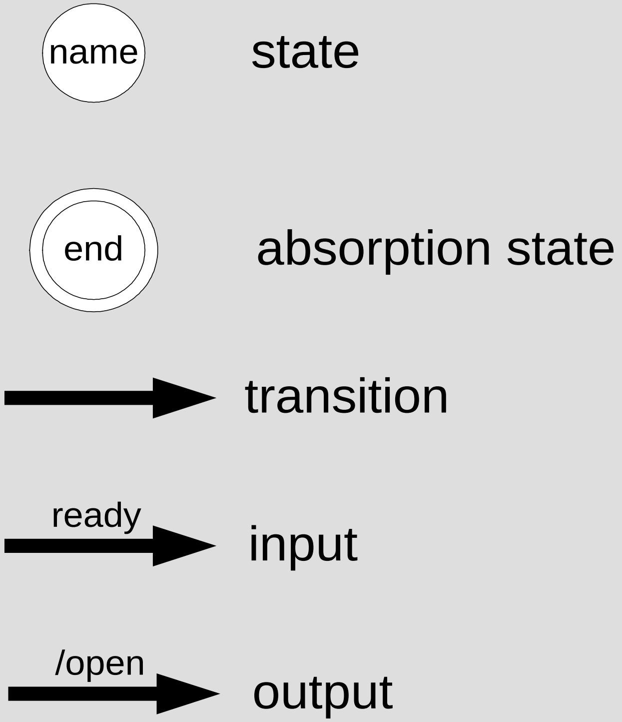
Definition:

A **Finite State Machine** *(FSM)* can be interpreted as an automaton over the alphabet $\boldsymbol{I} \times \boldsymbol{O}$.

The set of all pairs $(\boldsymbol{I_k}, \boldsymbol{O_k})$ such that sequence $\boldsymbol{O_k}$ is produced as output on applying sequence $\boldsymbol{I_k}$ as the input, gives the language of the automaton.

Finite State Machines

Graphical Notation:



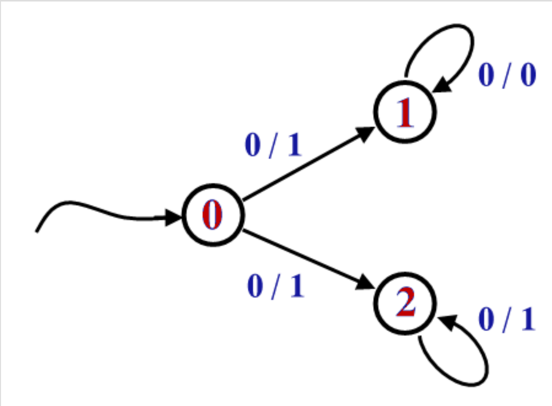
Finite State Machines

Classification

- ✧ [illegible] (PNDFSM)

[illegible]

Otherwise, it is said to be **incompletely specified**.



Finite State Machines

In the automaton sense

- Both DFSM and PNDFSM are ***deterministic***

  The underlying automaton for a PNDFSM makes a unique transition for a given I / O pair.

- NDFSM is ***non-deterministic***

Finite State Machines

Time-discreetness

- **Synchronous FSM:**
  A FSM conforming time-discreteness assumption.
- **Asynchronous FSM**
  A FSM doesn't conforming time-discreteness assumption.

Some state variables are not measured continuously, but only at the discrete instants of time at which a certain specified event is exhibited by its entity.

This specified event is called a ***synchronizing signal***.

Therefore, a system variable $V(i)$ can be written as $\boldsymbol{V_i}$, which designates the value of variable $\boldsymbol{V}$ at the ***i***-th sampling time.

The behavior of the system at any sampling time t_i is independent of the interval between t_i and the previous sampling time t_{i-1}. Thus, the true independent quantity is not time, but the ordinal number associated with the sampling times.

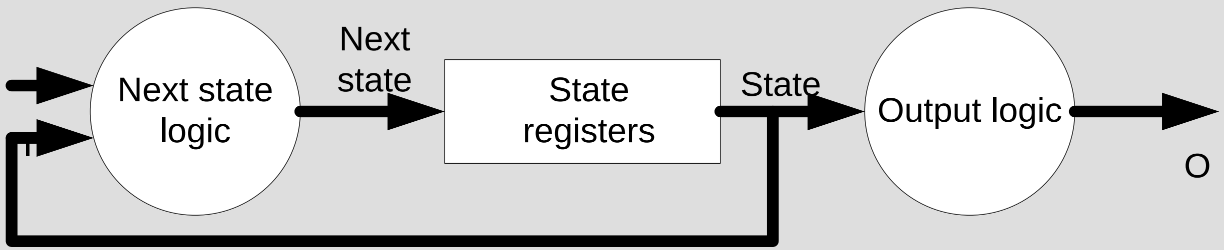
Finite State Machines

Types :

- Moore machine
- Mealy machines
- Medvedev machine

Finite State Machines

Moore machine



Moore machine is a 5-tuple:

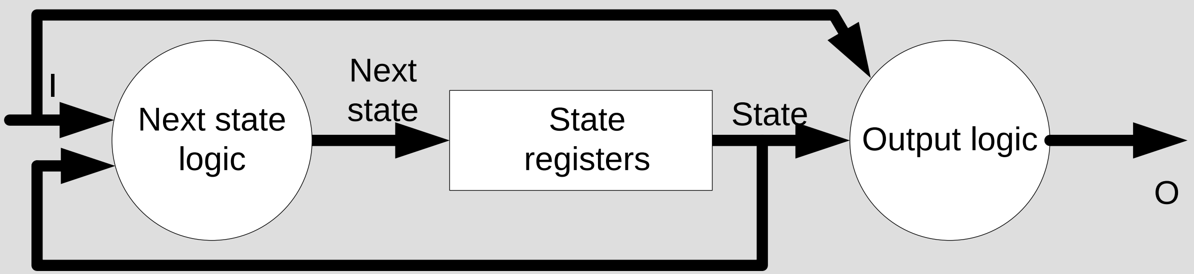
$$M = (S; I; O; \delta; \lambda)$$

$\lambda : S \rightarrow O$

Output function -> the output vector (O) is a function of state vector (S)

Finite State Machines

Mealy machine



Mealy machine is a 5-tuple:

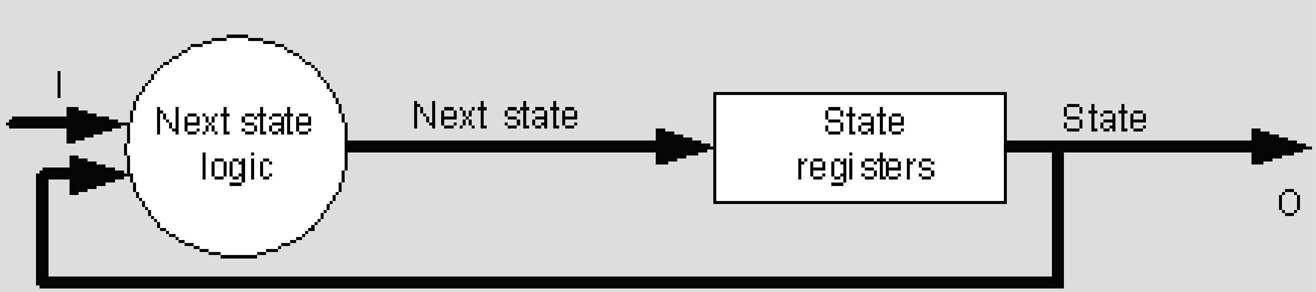
$$M = (S; I; O; \delta; \lambda)$$

$\lambda : S \times I \rightarrow O$

Output function -> the output vector (O) is a function of state vector (S) and the input vector (I)

Finite State Machines

Medvedev machine



Medvedev machine is a 5-tuple:

M = (S; I; O; δ; λ)

λ : S = O

Output function -> the output vector (O) resembles the state vector (S)

Finite State Machines

Modelling Aspects:

- ✧ Medvedev is inflexible
- ✧ Moore is preferred because of safe operation.
- ✧ Mealy is more flexible but has disadvantages –> possible spikes, long paths.

Every Moore automaton can be transferred to Mealy and vice versa.

Task modelling in D(S)CS

Petri Nets

Automata Models

Petri Nets

Petri net concept is a mathematical model.

Developed in the 1962[d] by **Carl Adam Petri** for describing the cause and effect relationships between events in a system.

The Petri Nets consist of two basic parts:

- ✧ a *static* graph structure for specifying the relationships mentioned above;
- ✧ an evaluation method (called the firing rule) for describing the *dynamic* behaviour of the system based on these relationships.

Petri Nets

Definition 1:

The triple $X = (P, T, F)$ is a set, where:

- $P \cap T = \phi$

A graphical notation :

- Each element of the set is connected to at least one
- for places - *circle*
- for transitions - *rectangle*

$F \subseteq (P \times T) \cup (T \times P)$

finite state of arcs

$$(F \neq \phi) \wedge (\forall x \in P \cup T, \exists y \in P \cup T : xFy \vee yFx)$$

- In the ***set X*** there are no pair of *positions* incident to one and the same *transition* :

$$\forall p_1, p_2 \in P : ({}^*p_1 = {}^*p_2) \wedge (p_1^* = p_2^*) \Rightarrow (p_1 = p_2)$$

*x : the set of its input elements $\{ y \mid yFx \}, \forall x \in X$

x* : the set of its output elements $\{ y \mid xFy \}, \forall x \in X$

Petri Nets

Definition 2:

A Petri nets in a five-tuple $PN = (P, T, F, W, M_0)$:

$M_0 : P \rightarrow \{ 0, 1, 2, ... \}$

the initial marking

$M_0(p), p \in P$ is the marking (i.e. numbers of tokens) of the place p.

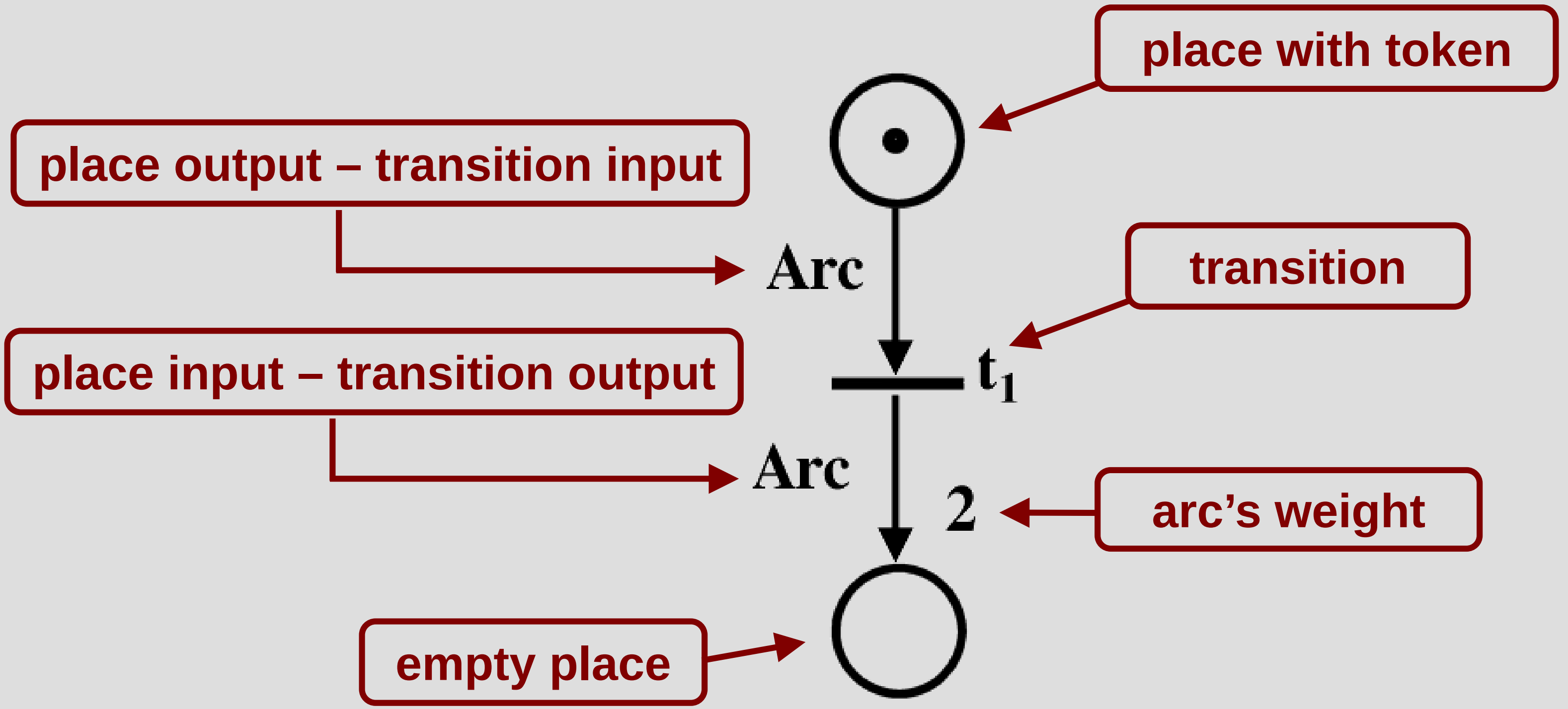
Petri Nets

Notations:

- *t : the set of input places of transition ***t***. That is the set of places ***p*** such that ***(p, t)*** ∈ ***F***
- t* : the set of output places of transition ***t***. That is ***p*** such that ***(t, p)*** ∈ ***F***
- *p : the set of input transition of place ***p***. That is the set of transitions ***t*** such that ***(t, p)*** ∈ ***F***
- p* : the set of output transition of place ***p***. That is ***t*** such that ***(p, t)*** ∈ ***F***

Petri Nets

Graphical Notation



Petri Nets

Activities:

- The transition t ∈ T is enabled if for some marking *M* for *PN*,

$$\forall p \in {}^{*}t : M(p) \geq F(p,t)$$

that means whatever place ***p*** from the input places of some transition ***t*** contains number of tokens greater or equivalent than the weight of the arc connecting them.

Petri Nets

Activities:

- Firing the transition *t* in the context of marking *M* results to new marking *M'* as follows
  - removing *W(p,t)* tokens from each , meaning that from each input place ***p*** of the transition ***t*** are removed number of tokens equal to the weight of the arc connecting the place and the transition;
  - adding *W(p,t)* tokens to each , meaning that to each output place ***p*** for the transition ***t*** are added number of tokens equal to the weight of the arc connecting the transition and the place.

Petri Nets

Activities:

- ✧ Firing the transition *t* in the context of marking *M* results to new marking *M'* as follows:

$$\forall p \in P : M'(p) = M(p) - F(p,t) + F(t,p)$$

equivalent to

$$M' = M - {}^{*}F(t) + F^{*}(t).$$

Petri Nets

Activities:

On the set of markings *M* is introduced relation $\longrightarrow$ for direct marking sequence as:

$$M \longrightarrow M' \Leftrightarrow \exists t \in T : (M \geq {}^{*}F(t)) \wedge (M' = M - {}^{*}F(t) + F^{*}(t))$$

If *M'* follows directly form *M* as a result of firing the transition ***t*** it is said that

$$\boldsymbol{M \xrightarrow{t} M'}$$

Petri Nets

Activities:

Marking ***M'*** is reachable from ***M*** if there is a sequence of markings ***M, M₁, M₂, …, M'*** and sequence of transition firings $\tau = < t_1 t_2 \ldots t_k >$ on the ***T***, so that

$$M \xrightarrow{t_1} M \xrightarrow{t_2} M \xrightarrow{t_3} \ldots \longrightarrow M \xrightarrow{t_k} M'$$

We will say $\boldsymbol{\tau} = < t_1\, t_2 \ldots\, t_k >$ to be a vector of firings.

Petri Nets

Activities:

To say that ***M'*** is reachable from ***M*** it is written

$$\boldsymbol{M} \longrightarrow \boldsymbol{M'} \quad \text{or} \quad \boldsymbol{M} \xrightarrow{\tau} \boldsymbol{M'}$$

A set of markings { *M'* | *M* $\rightarrow$ *M'* } reachable in the set *PN* from the marking *M* is represent as *R(PN,M)*.

The set *R(PN)* = *R(PN,M_0)* or the set of all markings reachable from the initial marking M_0 is named

the reachable places set.

Petri Nets

Incidence matrix:

The matrix $A = [\, a_{ij} \,]$ is a $n \times m$ matrix and for each a_{ij} is given

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$$

where:

- $a_{ij}^{+} = w(i,j)$

  It is equal to the weight of the output arc from transition ***j*** to its output place ***i***.

- $a_{ij}^{-} = w(j,i)$

  it is equal to the weight of the input arc from place ***i*** to its output transition ***j***.

Petri Nets

Incidence matrix:

The incidence matrix can be represented as two different matrixes A^+ and A^-

- ✧ A^+ : represents the weight of output arcs from every transition to every place,
- ✧ A^- : represents the weight of output arcs from every place to every transition.

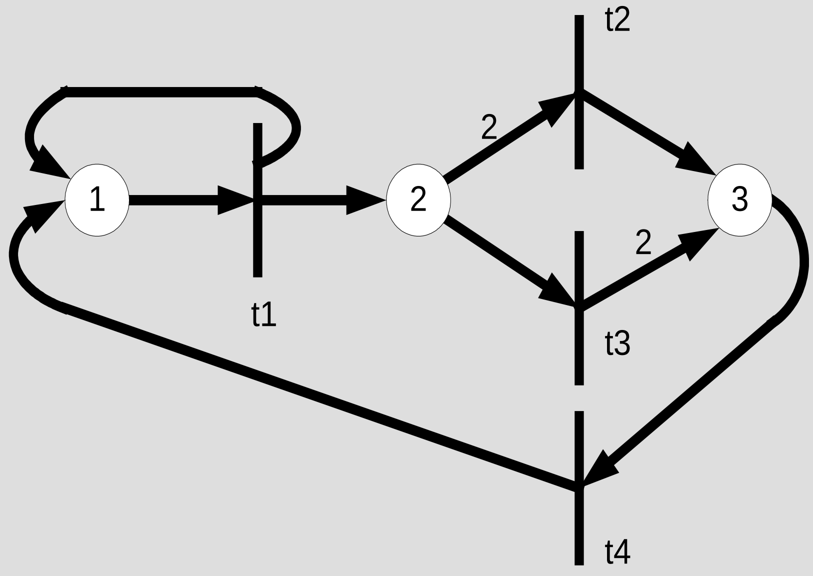
Petri Nets

Incidence matrix: *Example*

A sample Petri net and corresponding matrixes of incidence:

A⁺:

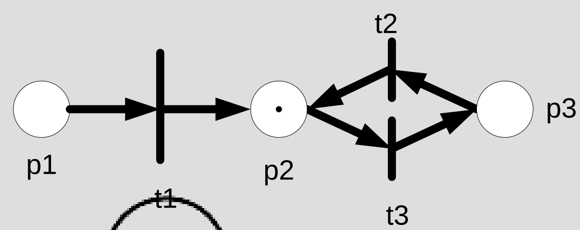
	p_1	p_2	t_3 p_3	t_4
p_1	01	01	000	0
p_2	00	-22	111	0
p_3	00	-01	022	1
t_4	11	00	-10	

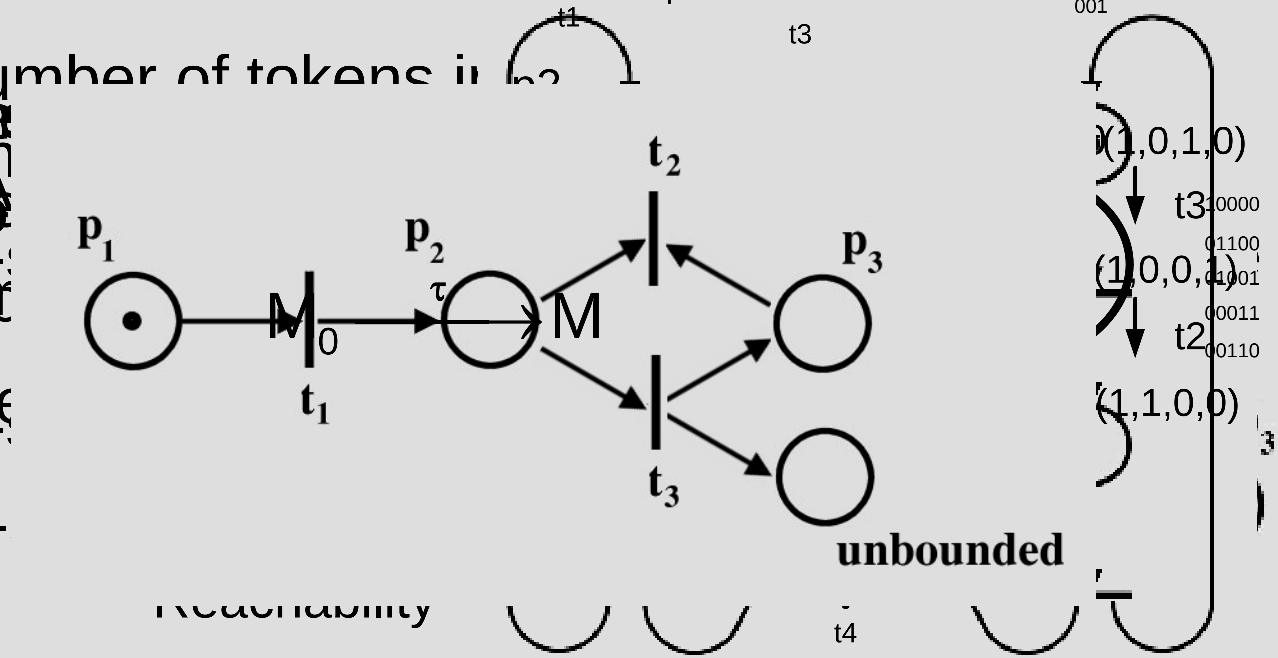


Petri Nets

Properties:

Boundedness



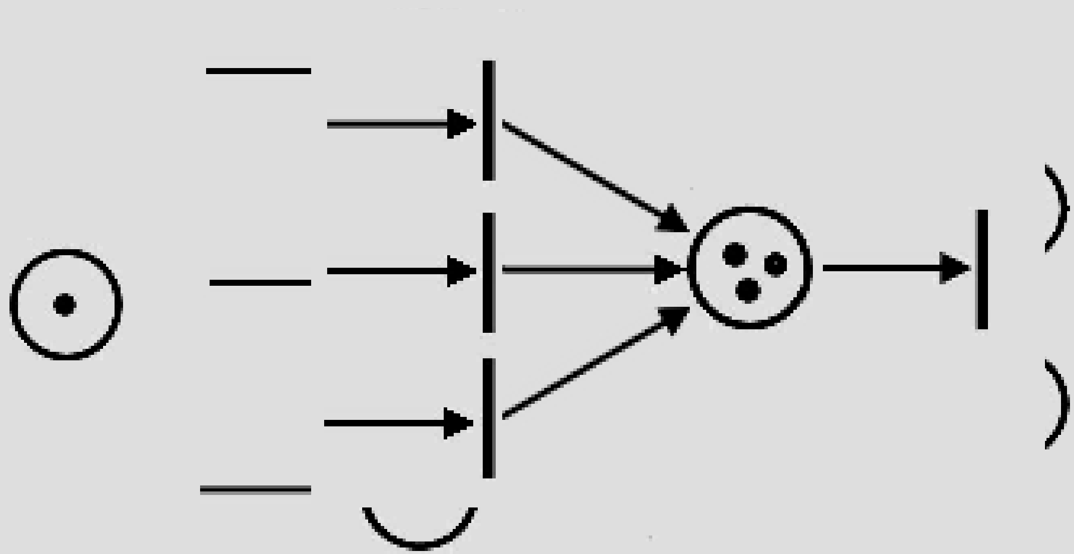


Reachability graph

Petri Nets

In progress modelling:

1. Sequential execution
2. Synchronization
3. Merging
4. Conflict/nondeterminist

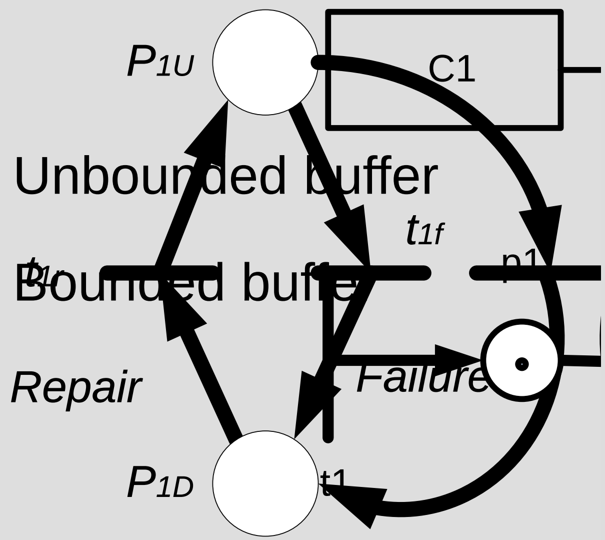


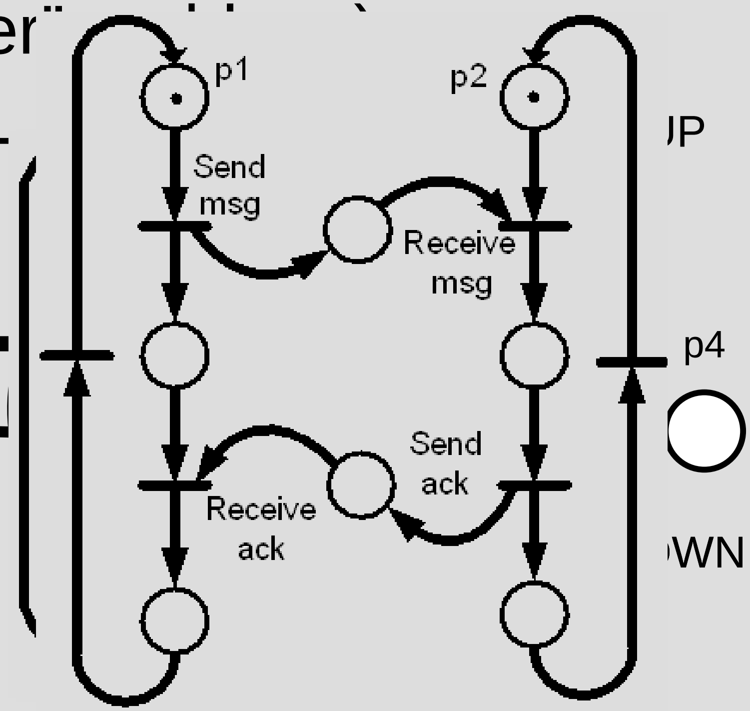
Petri Nets

Examples:

[illegible] buffer
(a “Producer / Consumer”

- Unbounded buffer
- Bounded buffer





Petri Nets

Analyzes :

- Structural analysis techniques
  - Incidence matrix
  - T- and S- Invariants
- State Space Analysis techniques
  - Coverability Tree
  - Reachability Graph

Petri Nets

Advantages and Disadvantages :

- Easy modeling of precedence relations, concurrency, conflicts and mutual exclusion of systems;
- Modelling of time reaction is not easy;
- Very close to state explosion situation thus making graphical and mathematical representation hard;
- Formal graphics representation enables easy understanding of structure of complex systems;
- Hard to follow changes in system structure.
- Analyzes possible for detecting deadlocks, starvations, overflow, irreversive situations, etc;
- Performance analyzes based of deterministic timed Petri nets or stochastic timed Petri nets;

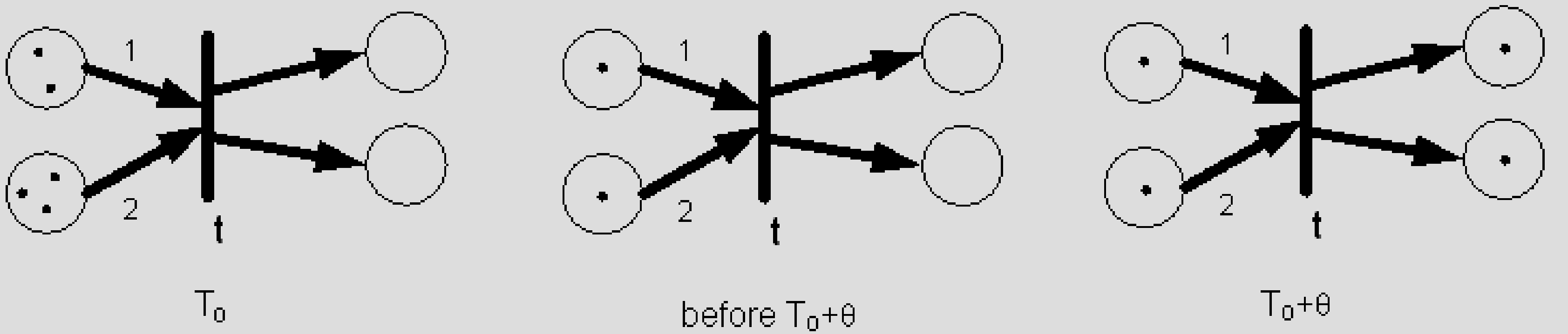
Petri Nets

Timed Petri Nets

Timing of places – when time is associated with a place it represents the minimal amount of time token have to remain in the place as a result of a firing.

✧ Let us assume that time associated with transition is *θ*, and that firing of transition *t* starts at moment T_0. Then, firing *t* consists of:

- ➢ removing *w(p,t)* tokens from every at time T_0.



an operation is performed.

Programmable Logic Controller

Definition:

The programmable logic controller is a *reconfigurable* software architecture that has been specifically developed for synchronous (time-driven) sequential controllers.

Programmable Logic Controller

Characteristics:

The PLC emulates the circuit diagram of the state machine (and ***indirectly*** – its behaviour).

Implemented by periodical computation of two sets of Boolean functions describing the logical structure of the state machine:

- *state transition functions*
- *output functions* (or state transition functions used in conjunction with control memory).

Programmable Logic Controller

Characteristics:

Reconfiguration can be done in several ways:

- By special-purpose graphical languages, such as function block diagrams and ladder diagrams defined in standard *IEC 61131-3* in order to specify the structural model of the implemented controller, which is subsequently translated into executable code
- By generating a data structure containing the machine-level description of controller functions, which is interpreted subsequently by a standard software routine, i.e. the state machine driver.

Programmable Logic Controller

Characteristics:

Binary decision diagram's are used for table description of controller's functions :

- *Binary decision diagram (BDD)* is a special technique used to represent and efficiently compute Boolean functions.

  Its main advantage is in terms of computation time, which grows linearly with the number of arguments evaluated (in most cases).

Programmable Logic Controller

Characteristics:

The PLC-type state machine driver provides a solution to the problem of designing re-configurable and reusable state machines. However, this approach has also some shortcomings:

- It is necessary to carry out the logical design of the state machine before implementing it in software. This might be a difficult task when implementing complex state machines having a great number of states and transitions.

Programmable Logic Controller

Characteristics:

The PLC-type state machine driver provides a solution to the problem of designing re-configurable and reusable state machines.

Drawbacks:

- It is necessary to carry out the logical design of the state machine before implementing it in software.
- PLSs are not optimal in terms of execution time. They do not exploit the *locality of behaviour.*

State Logic Controller

Problems of non-optimality of PLCs can be eliminated with the other type of software architecture:

the ***State Logic Controller (SLC).***

SLC are used to implement reconfigurable and reusable state machines.

State Logic Controller

The SLC is built around a data structure, which contains the machine-level representation of the formal model specifying system behaviour - a state transition graph.

SLC interprets the model, resulting in ***direct*** emulation of system behaviour. (vs. indirect emulation of PLCs).

System model is implemented as a table consisting of modified binary decision diagrams that represent the next-state mappings of various states, as specified by the state transition graph.

State Logic Controller

The SLC-type state machine driver is :

- ✧ powerful
- ✧ simple
- ✧ it can be used to implement all types of :
  - ➢ sequential controller including synchronous (time-driven)
  - ➢ asynchronous (event-driven) state machines
- ✧ it can be extended to continuous and hybrid control systems as well as other applications.

Continuous Control Systems

Formal specifications of control systems (computational models)

- **Data-flow models**

  An *instruction / module* is executed when the operands required become available.

- **Control-flow models**

  An *instruction / module* executed when the previous *instruction / module* in a defined sequence has been executed.

Continuous Control Systems

Formal specifications of control systems (computational models)

- **Data-flow models**

  An *instruction / module* is executed when the operands required become available.

- **Control-flow models**

  An *instruction / module* executed when the previous *instruction / module* in a defined sequence has been executed.

Continuous Control Systems

Formal specifications of control systems (computational models)

- **Data-flow models**

  An *instruction / module* is executed when the operands required become available.

- **Control-flow models**

  An *instruction / module* executed when the previous *instruction / module* in a defined sequence has been executed.

Data Flow Systems

<u>History</u>

Proposed by **Kahn** in 1958 as a formal model.

Data flow technique was originally developedin 1960s by **Karp** and **Miller** as graphicmeans of representing computations.

Unique attribute: deterministic

Data Flow Systems

Applications of dataflow

- Block-diagram specifications
- Circuit diagrams
- Linear / Nonlinear Control Systems
- Signal processing
- Suggest dataflow semantics
- Common in Electrical Engineering

Data Flow Systems

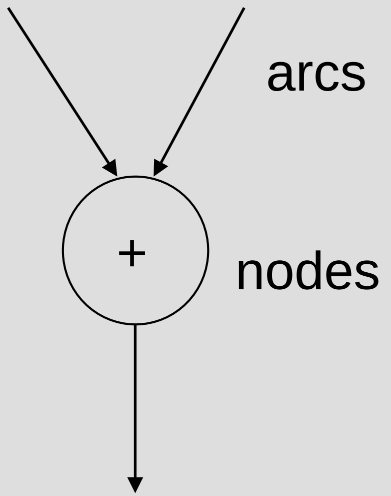
Data Flow Models

It uses a directed graph called :

a *data dependence graph*

or

a *dataflow graph*



Data Flow Systems

Types :

- Static -> Only one instance of a node firing at a time:
  - A node fires when all input tokens appear and the previous output token have been consumed.
  - More than one token is allowed on each arc and previous output tokens need not be consumed before the node van fire again

These rules allows pipeline computations and loops but not recursion or code sharing.

There is a handshaking acknowledgement mechanism.

- Dynamic -> the multiple instances of a particular node-firing is possible :
  - Nodes fire when all input tokens with the same tag number appear.
  - Input tokens are then realised and new output tokens are generated.

Dataflow graph being executed is not fixed though such as actions as recursion and code sharing.

There is no requirement of acknowlegment mechanism.

Data Flow Systems

Macrodataflow:

1. Definition:

- Dataflow in which each node can represent complex serial function.
- The nodes represent procedures/functions, the input tokens carry procedure/function parameters and the output tokens carry procedure/function results.

2. Rules:

- **Standard firing rule**

  Node fires when all of the operands are recieved

- **Non-standard firing rule**

  Node fires when certain specific operand are recieved. Each nodal operation is completed when all necessary operand are received.

Data Flow Systems

Intuitive Semantics

- Actors (often stateless) perform computation
- Unbounded FIFOs perform communication via *sequences of tokens* carrying values
  - integer, float, fixed point
  - matrix of integer, float, fixed point
  - image of pixels
- State implemented as self-loop

- Determinacy
  - unique output sequences given unique input sequences
    matrix of integer, float, fixed point
  - sufficient condition
    Blocking read -> process cannot test input queues for emptiness

Data Flow Systems

Kahn Process

The process is constrained to be continuous.

- ✧ Prefix ordering of sequences: $X \subseteq Y$
- ✧ Set of sequence can be ordered as well:

$$X \subseteq Y, \text{ if } X_i \subseteq Y_i \text{ for } \forall i$$

- ✧ Increasing chain of sequences

$$\chi = \{ X_0, X_1, \ldots \},$$

where: $X_0 \subseteq X_1 \subseteq \ldots.$

Data Flow Systems

Kahn Process

The process is constrained to be continuous.

- Least upper bound of χ : $\cup\ \chi$
- Functional process $F: S^p \rightarrow S^q$:
  - Continuity

    $$F(\cup\chi) = \cup F(\chi)$$

  - Monotonicity

    $$X \subseteq X' \Rightarrow F(X) \subseteq F(X')$$

- Reading an empty channel **blocks** until data is available.

Data Flow Systems

Network of Processes

- A network: a set of relations between sequences.

$$X = F(X, I)$$

- Any ***X*** that forms a solution is called a fixed point. Continuity of ***F*** implies that there will be exactly one “minimal” fixed point.
- Execute the network by first setting ***I*** = ⊥ and finding the minimal fixed point, then by iterative computation to find other solutions.

Data Flow Systems

Synchronous Dataflow (SDF)

Edward Lee and David Messerchmitt, Berkeley, 1987

Restriction of Kahn Networks to allow compile-time scheduling.

Basic idea:

Each process reads and writes a fixed number of tokens each time it fires.

Data Flow Systems

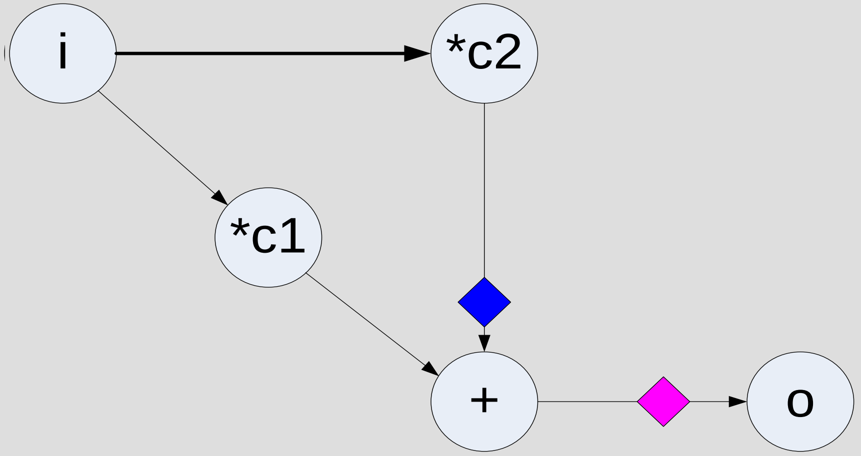
Synchronous Dataflow (SDF)

- Firing rules: Fixed token consumption/production
- Can be scheduled statically
  - Solve balance equations to establish rates
  - A correct simulation produces a schedule if one exists
- Looped schedules
  - For code generation: implies loops in generated code
  - Recursive SCC Decomposition

Data Flow Systems

Synchronous Dataflow (SDF)

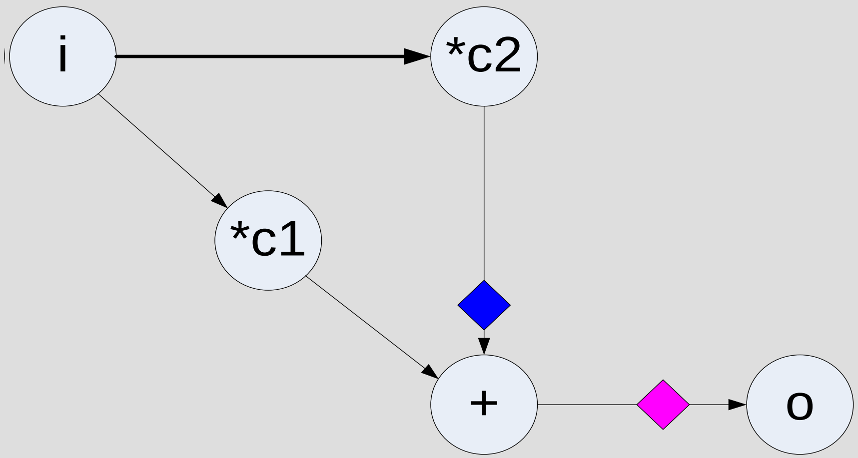
- Example:



Data Flow Systems

Synchronous Dataflow (SDF)

- Example: step-by-step execution



Control Flow Systems

- Representing system reactions instead of data flow.
- An *instruction / module* executed when the previous *instruction / module* in a defined sequence has been executed.
- Data presence/absence has no influence to activation process

This model is very effective to represent event-driven systems.

The simplest model is FSM.

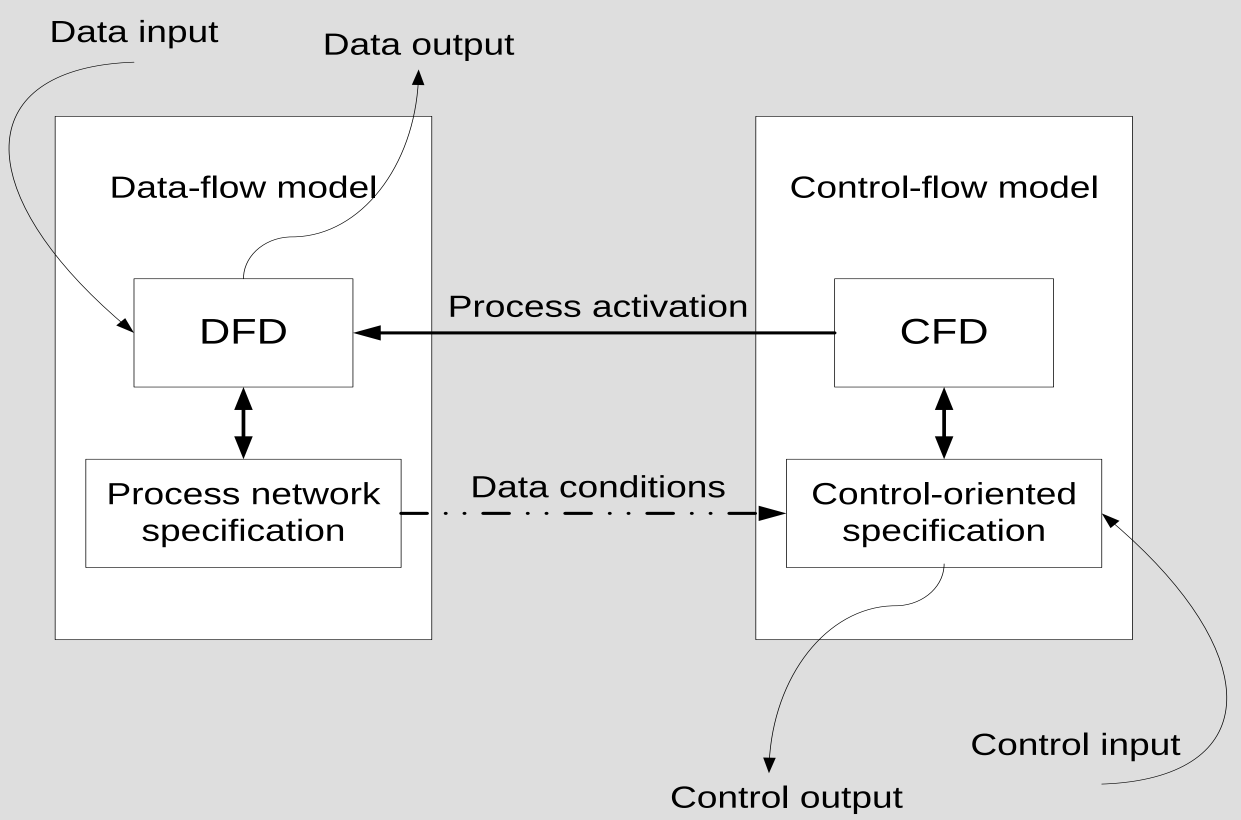
Control Flow Systems

<u>Steps in building CFM</u>

- Remove data flows from data-flow diagram (DFD)
- Add control flows, stores, events and windows
- Events may be:
  - sensor inputs
  - data conditions
  - switches and interrupts
  - signals from other programs

Control Flow Systems

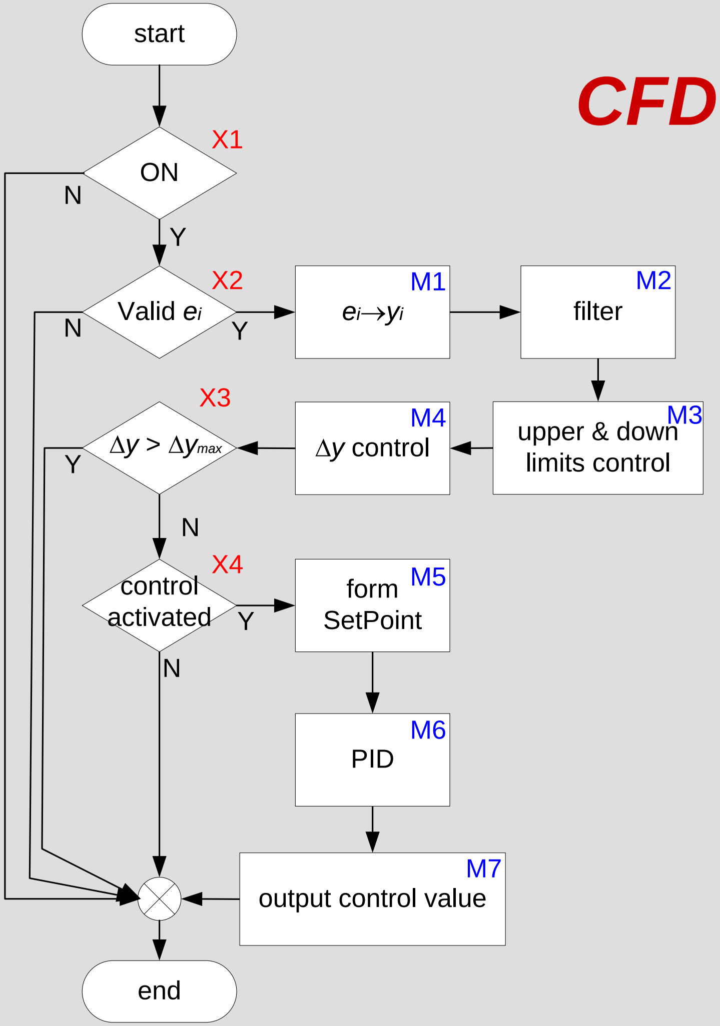
Relations between DFM and CFM



Control Flow Systems

Example:

CFD creation

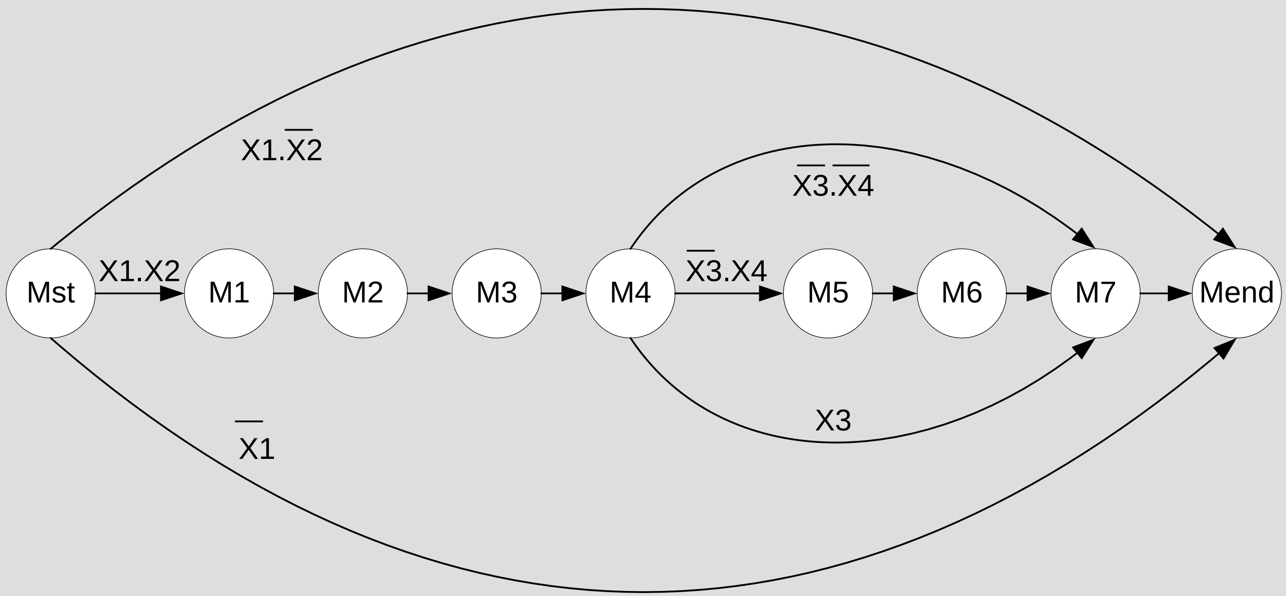


Flow-chart of PID contour

Control Flow Systems

Example:

CFD creation



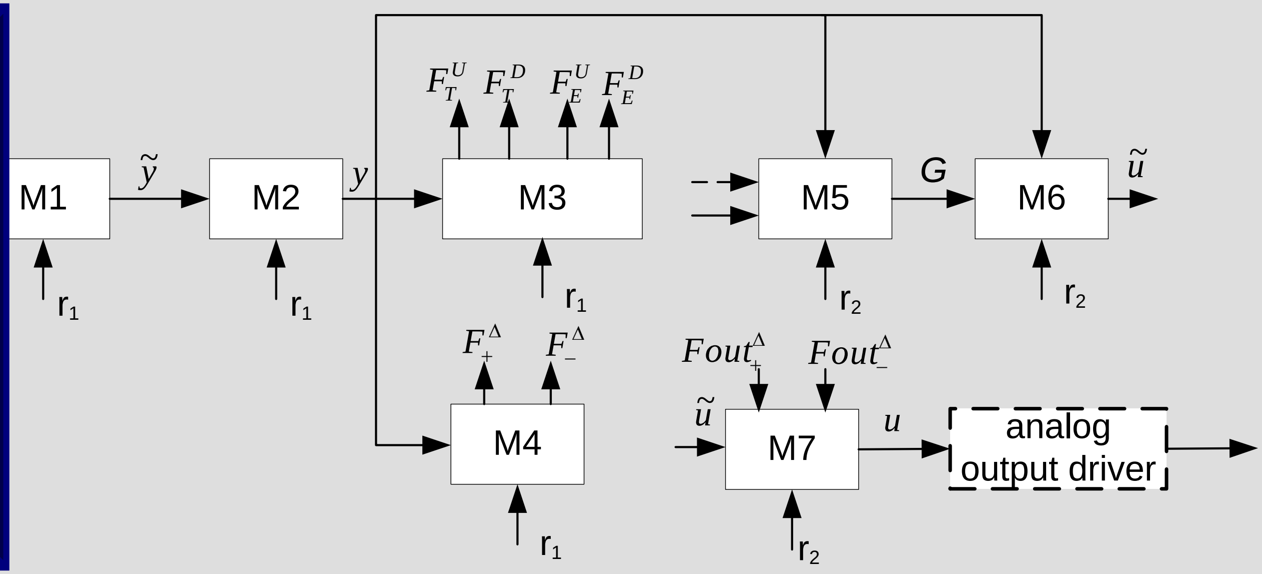
Control-flow graph

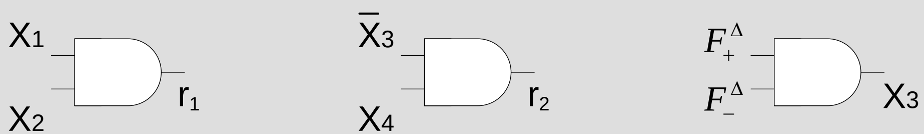
Control Flow Systems

Example:

CFD creation

$$\tilde{y} = f_1(e)$$
$$y = f_2(\tilde{y})$$
$$\tilde{u} = f_6(G, y)$$
$$u = f_7(\tilde{u})$$
$$G = f_s(\ )$$
$$u = f_7(f_6(f_2(f_1(e))))$$





Data-flow graph

Control Flow Systems

Example: ***CFD creation***

Steps of work:

1. Analog input driver converts signal to code
2. If conversion is correct M1 transforms **e** to technological value $\tilde{y}$
3. M2 eliminates signal noise $(\tilde{y} \to y)$
4. M3 controls signal value to be between technological and emergency limits
5. M4 controls signal value to has gradient between upper and down limits
6. M5 generates control set point – all of possible strategies for set point generation can be implemented
7. M6 is PID regulator – it generates output signal $\tilde{u}$
8. M7 controls generated output value $\tilde{u}$ to correspond to upper / down output gradient $(\tilde{u} \to u)$
9. Analog output driver converts control value *u* to analog signal

Hybrid Control Systems

Hybrid control systems are reactive control systems that involve both continuous and discrete dynamics and continuous and discrete controls.

Hybrid Control Systems

<u>Possible approaches</u>

- ✧ Two tier control system – supervisory discrete automaton and number of sequential controllers
- ✧ Extended FSM possible to implement signal transformations (continuous control) as state activities.

Hybrid Control Systems

<u>The simplest solution</u>

Designed by considering the continuous and discrete event systems separately - by suppressing either

<u>Difficulties:</u>

The interaction between the continuous and discrete event systems complicate analysis.

becomes a differential equation

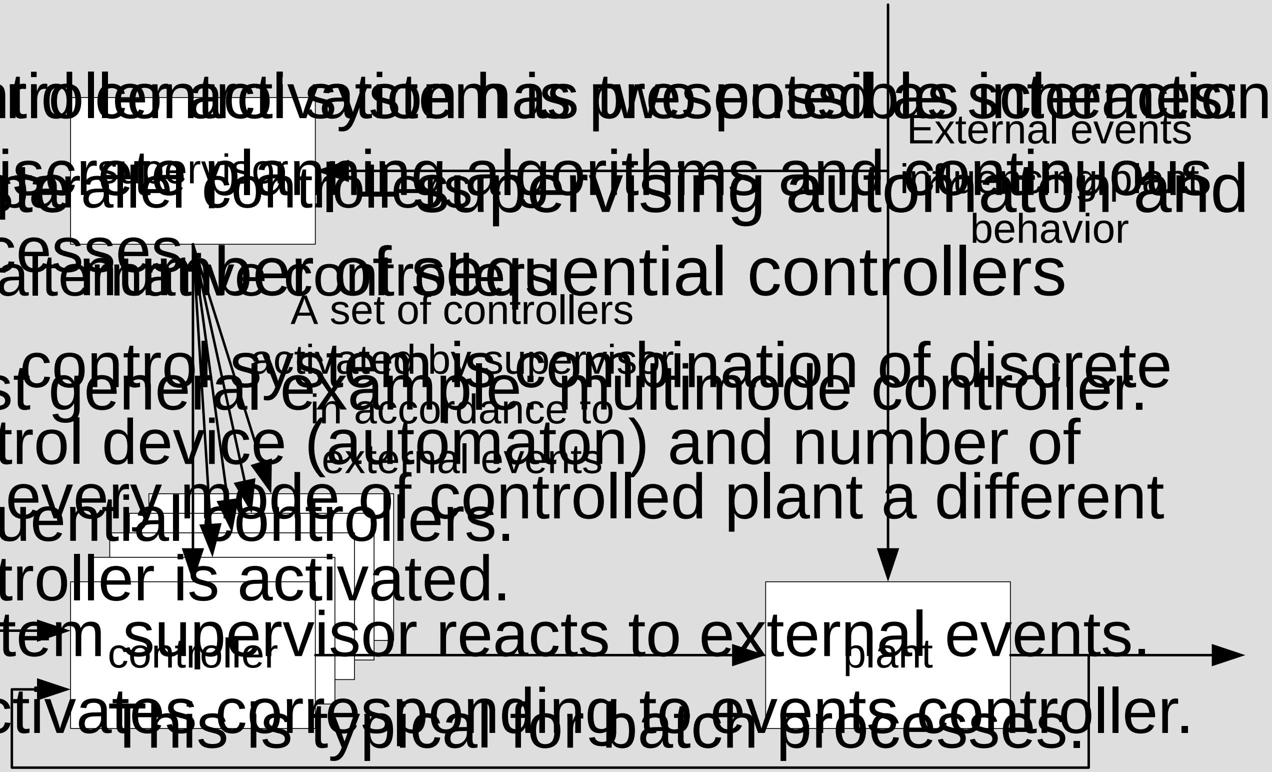
Hybrid Control Systems

Modelling by combining state machine(s) and sequential flow controllers

- Hybrid control system is presented as interaction of discrete planning algorithms and continuous processes.
- The control system is combination of discrete control device (automaton) and number of sequential controllers.

- Controller activation has two possible schemes:
  - ➢ parallel controllers
  - ➢ alternative controllers

System structure: supervising automaton and number of sequential controllers



- Most general example: multimode controller.
- For every mode of controlled plant a different controller is activated.
- System supervisor reacts to external events. It activates corresponding to events controller.

This is typical for batch processes.

Hybrid Control Systems

Modelling by hybrid controllers – extended state machines

Modern approach in system design.

Control system behavior is represented as extended form of Moore machine able to make signal transformations as node operations.

Hybrid Control Systems

Modelling by hybrid controllers – extended state machines

1. Formal basis :

$$P = \{A, X, C, Y, E, R, S\},$$

A set of predicates, C = { c_k }

A set of signal transformations, specifying the computafion of output signals during the execution of the coresponding reactions

Hybrid Control Systems

Modelling by hybrid controllers – extended state machines

2. Specification of system reactions

$$R = \{ r_i \} ,$$

where :

$$r_i \; : \quad A \times E \times C \rightarrow Y$$

Hybrid Control Systems

<u>Modelling by hybrid controllers – extended state machines</u>

3. Specification of signal transformations

Signal transformation functions specify how the output signals are generated within the corresponding system reactions

$$\forall y_k \in Y \leftrightarrow s_k \in S$$

$$s_k : X_k(t) \quad\quad y_k(t),\ X_k \subseteq X$$

s_k may be represented as a composition of simple functions,

i.e.: $y_k = w_l^k \quad w_{l-1}^k \quad w_{l-2}^k \ldots\ldots\ldots \quad w_1^k$

Hybrid Control Systems

Modelling by hybrid controllers – extended state machines

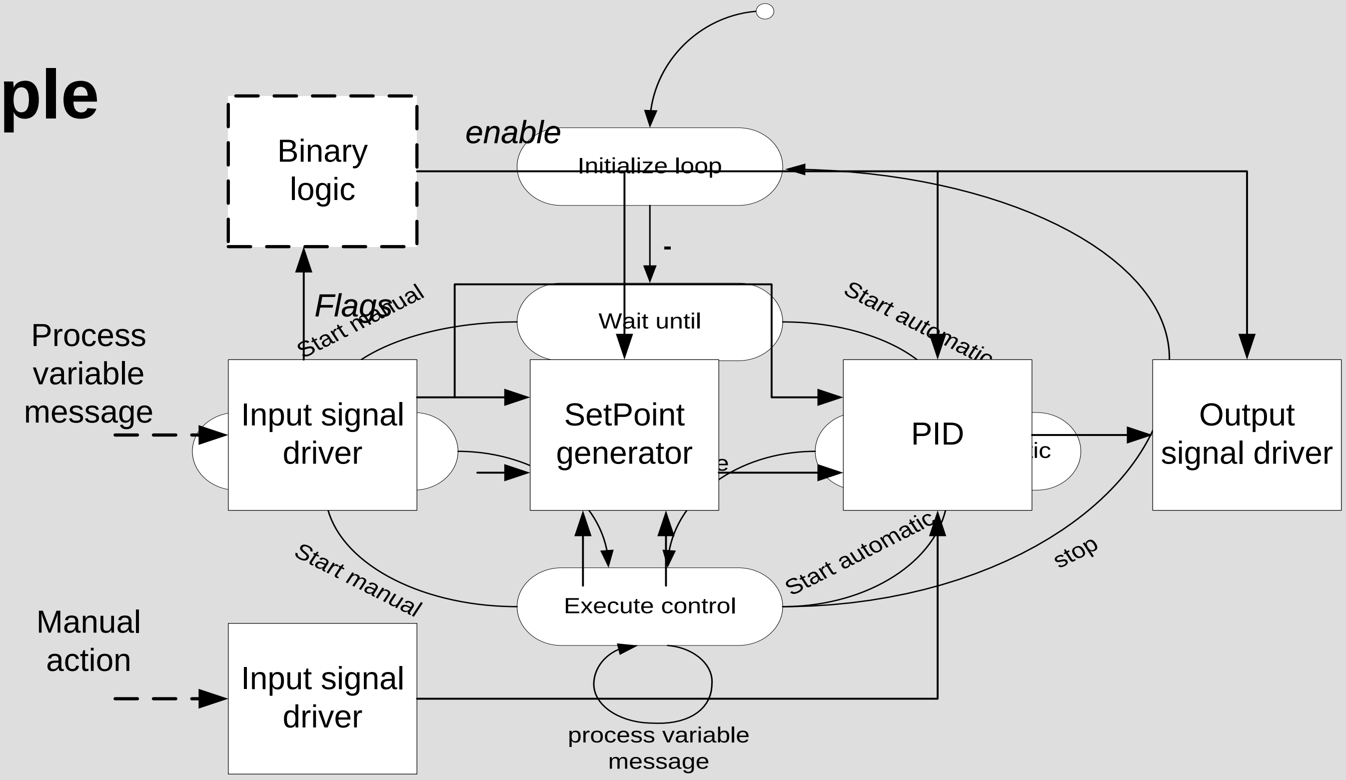
3. Specification of signal transformations

Combination of system reactions defined as for Moore machine and signal transformations as dataflow leads to new hybrid (extended) Moore machine.

Hybrid Control Systems

Modelling by hybrid controllers – extended state machines

4. Example



Graphic Notations and Languages

- Specification languages
  - UML
  - SDL
  - VHDL
- Languages defined in standards

  IEC 61131 and IEC 61499

UML

- The Unified Modelling Language (UML), is the language that can be used to model systems.
- Unified Modeling Language is:
  - An emerging standard for modeling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)

UML

- UML provides the ability to capture the characteristics of a system by using notations.

- UML provides a wide array of simple, easy to understand notations for documenting systems based on the object-oriented design principles.

- UML does not have any dependencies with respect to any technologies or languages.

- UML can be used to model applications and systems based on either of the current hot technologies.

UML

Diagrams

- Use case diagram
- Class diagram
- Object diagram
- State diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram
- Component diagram
- Deployment diagram

The underlying premise of UML is that no one diagram can capture the different elements of a system in its entirety. Hence, UML is made up of nine diagrams that can be used to model a system at different points of time in the software life cycle of a system.

UML

Diagram Classification:

A software system can be said to have two distinct characteristics:

- ✧ a structural ("static“) part
- ✧ a behavioural ("dynamic“) part.

An additional characteristic of a software system possesses is related to implementation.

UML

Diagram Classification:

- Static - this is essentially the structural aspect of the system. It defines what parts the system is made up of.
- Dynamic - this is behavioural features of a system
- Implementation - describes the different elements required for deploying a system

UML

Diagram Classification:

- Static:
  - Use case diagram
  - Class diagram

The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define what parts the system is made up of.

- Dynamic:
  - Object diagram
  - State diagram
  - Activity diagram
  - Sequence diagram
  - Collaboration diagram

The behavioural features of a system - the ways a system behaves in response to certain events or actions are the dynamic characteristics of a system.

- Implementation
  - Component diagram
  - Deployment diagram

The implementation characteristic of a system is a feature that describes the different elements required for deploying a system.

UML

Production life-time view:

- ✧ Design View
- ✧ Process View
- ✧ Component View
- ✧ Deployment View
- ✧ Use case View

SDL

Specification and Description Language (SDL) is a textual and graphical language that is both formal and object-oriented.

SDL

SDL is “an object-oriented, formal language …” intended for the specification of:

- complex applications
- event-driven applications
- real-time applications
- interactive applications

involving many concurrent activities that communicate using discrete signals.

SDL

Main characteristics:

- Provides clear and unambiguous system description
- Simulation, Validation, and Verification tools available for SDLs.
- Conversion tools available to take SDL in and generate "useable code".
- Communicating by signals exchange

- Time abstraction: easy mapped to target system.
- Exception handling (SDL 2000).
- SDL is constructed in a manner to enable description of system's behaviour in abstract terms.
- Theoretical model implemented is extended independent FSMs running in parallel and
- Object-orientation – structure types concept (abstract data types).

SDL

Semantic models for system representation

- Architecture of a system
  - Hierarchical organization following the top-down method of design
  - Decomposition into functional entities: system, block, sub-block, channel, process
- Behaviour of a system
  - A set of cooperating processes, communicating through the exchange of signals and through distributed variables
  - Extended Finite State Machine model
- Data structures of system
  - Abstract data types, C like data structures or ASN.1

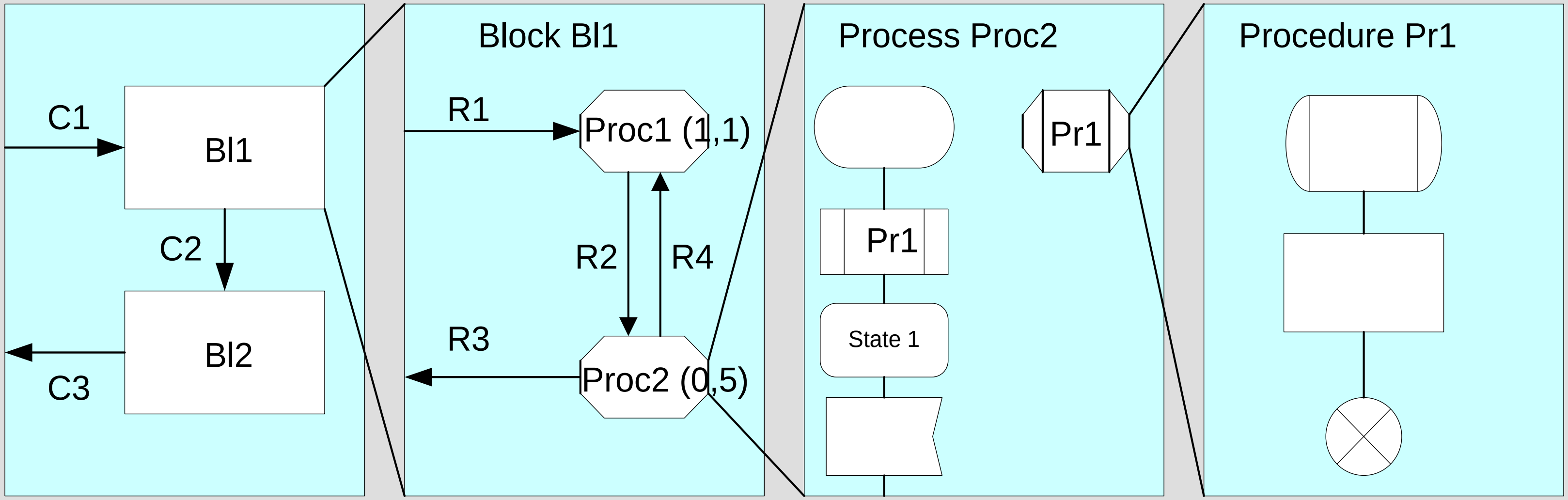
SDL

Program elements

- System block
- One or more Blocks
- Process (at least one)
- Procedure
- Processes can be dynamically allocated, blocks can not.

SDL

Relation among program elements



VHDL

<u>VHDL means</u>

Very High Speed Integrated Circuits Hardware Description Language

<u>Basic definition</u>

in Language Reference Manual (IEEE Design Automation Standards Committee, 1993a) :

"…a formal notation intended for use in all phases of the creation of electronic systems. …it supports the development, verification, synthesis and testing of hardware designs,…".

VHDL

VHDL means

Very High Speed Integrated Circuits Hardware Description Language

Basic definition

"…a formal notation intended for use in all phases of the creation of electronic systems. …it supports the development, verification, synthesis and testing of hardware designs,…".

in Language Reference Manual (IEEE Design Automation Standards Committee, 1993a)

VHDL

VHDL is combination of programming language and hardware modelling language. When it was designed the main goal of VHDL was to simulate circuits.

Now it is used not only for simulation but for synthesis. Accounting current situation of VHDL implementations its abbreviation can be changed to VHSIC Hardware Design Language.

VHDL

Language characteristics and elements

- Sequential Procedural language: inspired by ADA
- Concurrency: statically allocated network of processes
- Timing constructs
- Discrete-event simulation semantics
- Object-oriented elements: libraries, packages, polymorphism

VHDL

- One of most important parts of basic and extended versions of VHDL is time flow control.

Original version of VHDL is synchronous.

on an input port.

- The module reacts by running the code of its behavioural description and scheduling new

This is called scheduling a *transaction* on that signal. .

VHDL

VHDL is used outside electronics design.
An extension VHDL-ACS supports simulation and description of circuits having continuous over time and over amplitude behaviour.

It is technologically independent.

VHDL

VHDL is widely used for
simulation / description / synthesis of FSMs.

For the needs of transitional systems analyses it simulates non-zero-time atomic operations, thus enabling time-correctness analyses.

VHDL is very useful for simulation and analyzes of parallel event-driven processes.

VHDL

Main drawbacks

- it is hard for learning (except the user is not used in ADA)
- not all descriptions can be synthesized
- one and the same behaviour leads to different structures

Standard 61131

The IEC 1131 standards were developed to be a common and open framework for PLC architecture, agreed to by many standards groups and manufacturers.

They were initially approved in 1992, and since then they have been reviewed as the IEC-61131 standards.

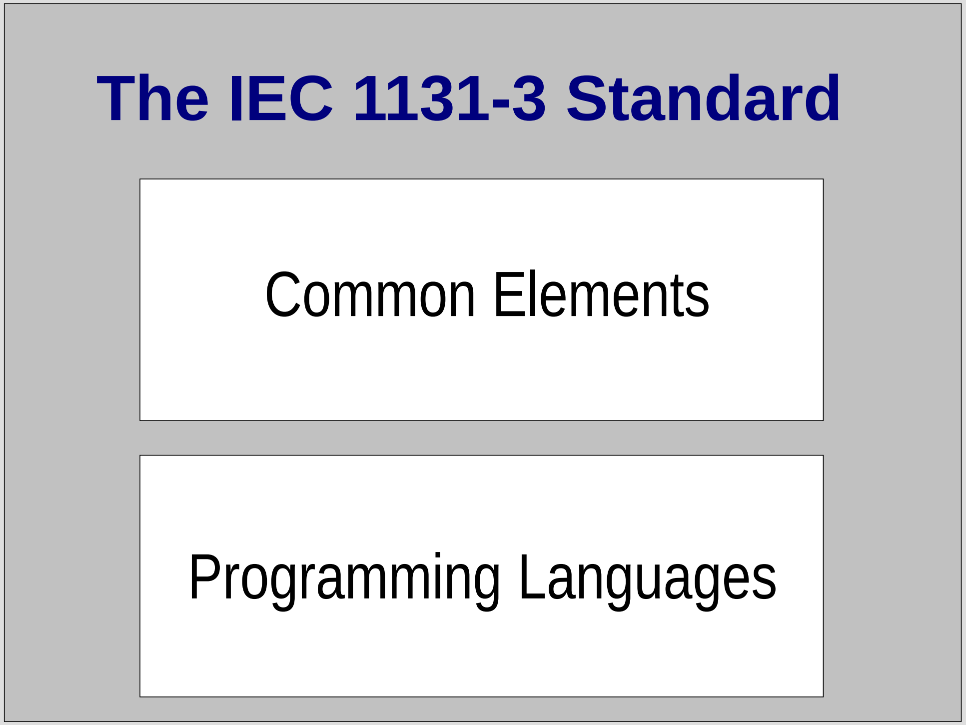
Standard 61131

<u>The main components</u>

- ✧ IEC 61131-1 Overview
- ✧ IEC 61131-2 Requirements and Test Procedures
- ✧ IEC 61131-3 Data types and programming
- ✧ IEC 61131-4 User Guidelines
- ✧ IEC 61131-5 Communications
- ✧ IEC 61131-7 Fuzzy control

Standard 61131

The programming models (IEC 61131-3) have the greatest impact on the user.

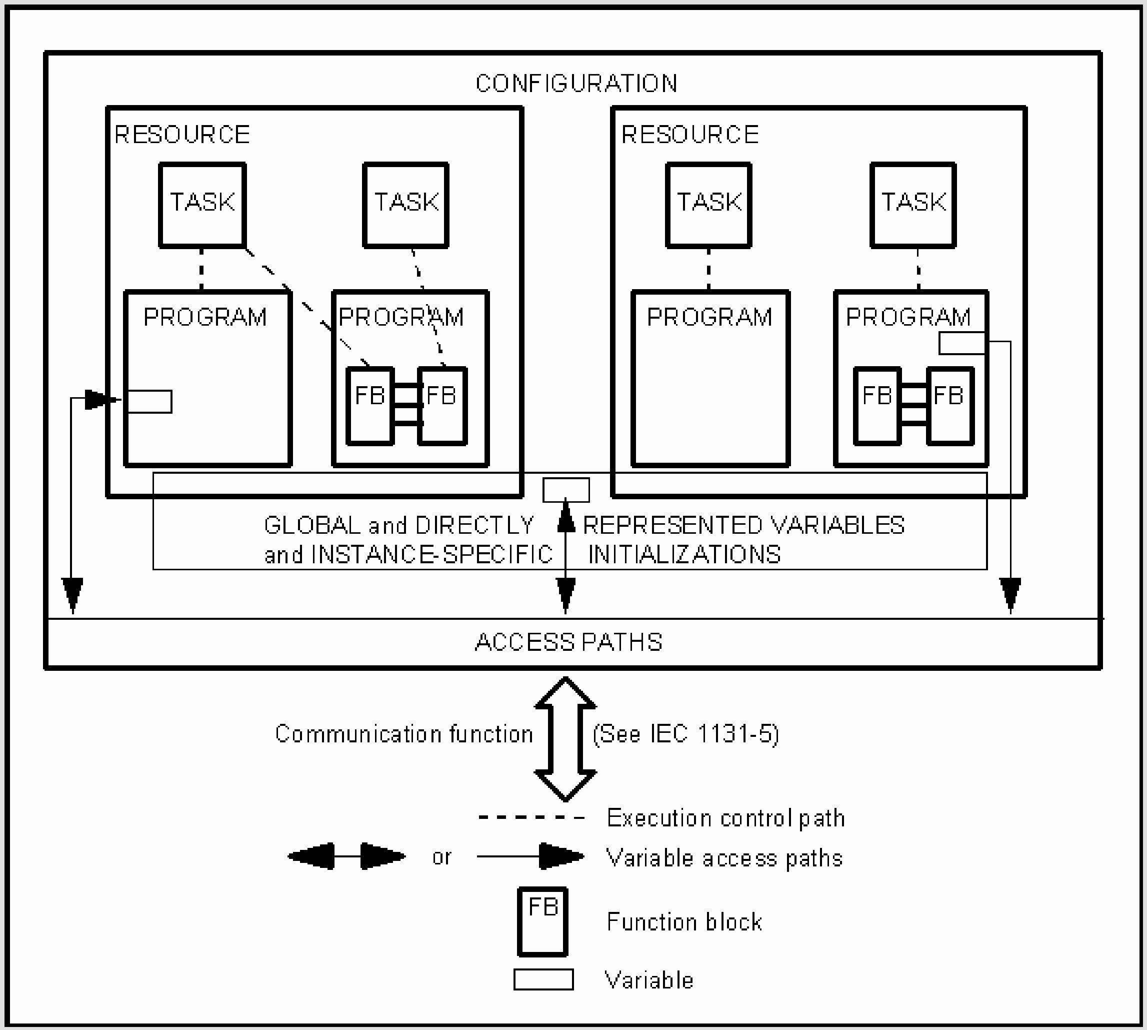


Standard 61131

Common Elements

- Data Type
- Variables
- Input variables
- Output variables
- Global variables
- Configuration, Resources and Tasks

Standard 61131



Standard 61131

<u>Program Organization Units</u>

- ✧ Functions
- ✧ Function Blocks (FBs)
- ✧ Programs

Standard 61131

Sequential Function Chart (SFC)

- SFC describes graphically the sequential behaviour of a control program.

- SFC structures

  The internal organization of a program, and helps to decompose a control problem into manageable parts, while maintaining the overview.

Standard 61131

Sequential Function Chart (SFC)

- SFC consists of :
  - Steps

Each element can be programmed in any of the IEC languages, including SFC itself.

It is very similar to GRAFSET language.

A transition is associated with a condition, which, when true, causes the step before the transition to be deactivated, and the next step to be activated.

Standard 61131

<u>Programming Languages</u>

- Textual:
  - Instruction List (IL)
  - Structured Text (ST)
- Graphical:
  - Ladder Diagram (LD)
  - Function Block Diagram (FBD)
  - Sequential Function Chart (SFC)

Standard 61131

Programming Languages

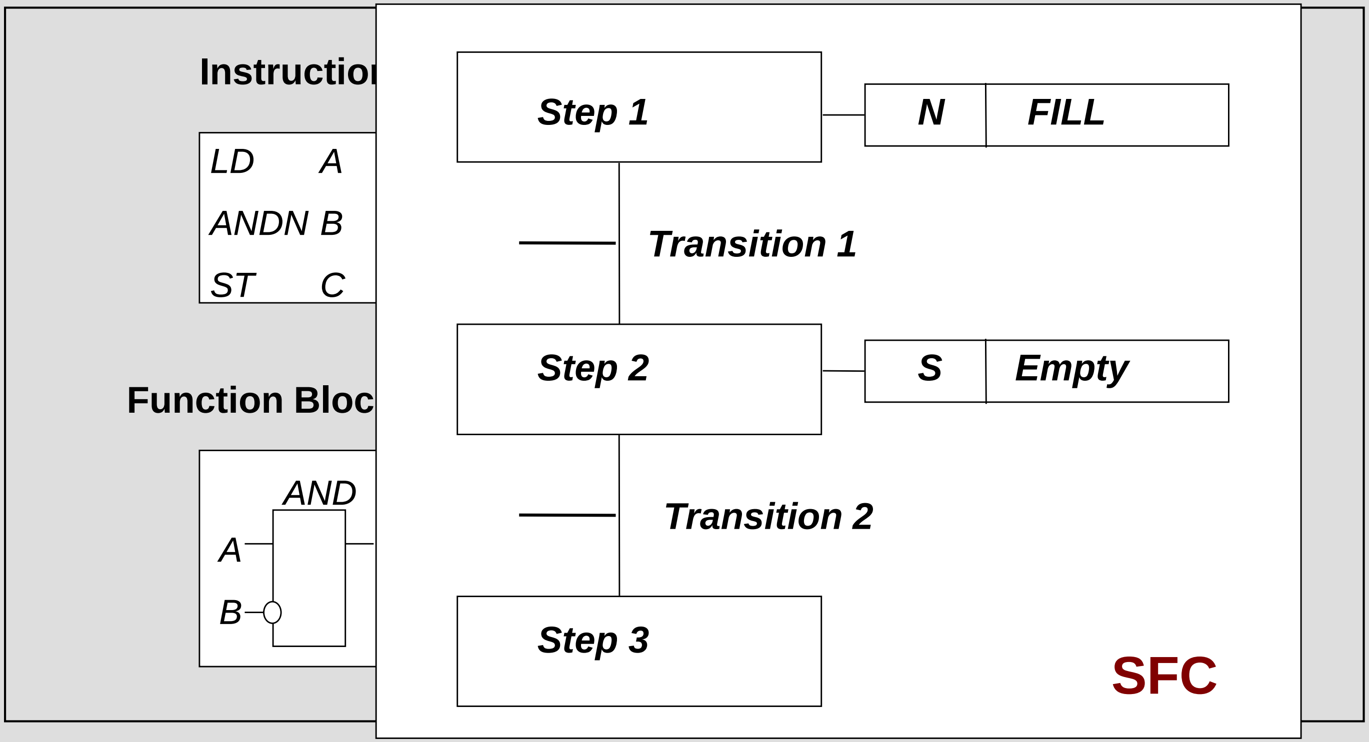
The choice of programming language is dependent on:

- the programmers' background
- what kind of the control problem
- the structure of the control system

All five languages are inter-linked: they provide a common suite, with a link to existing experience. In this way they also provide a communication tool, combining people of different backgrounds.

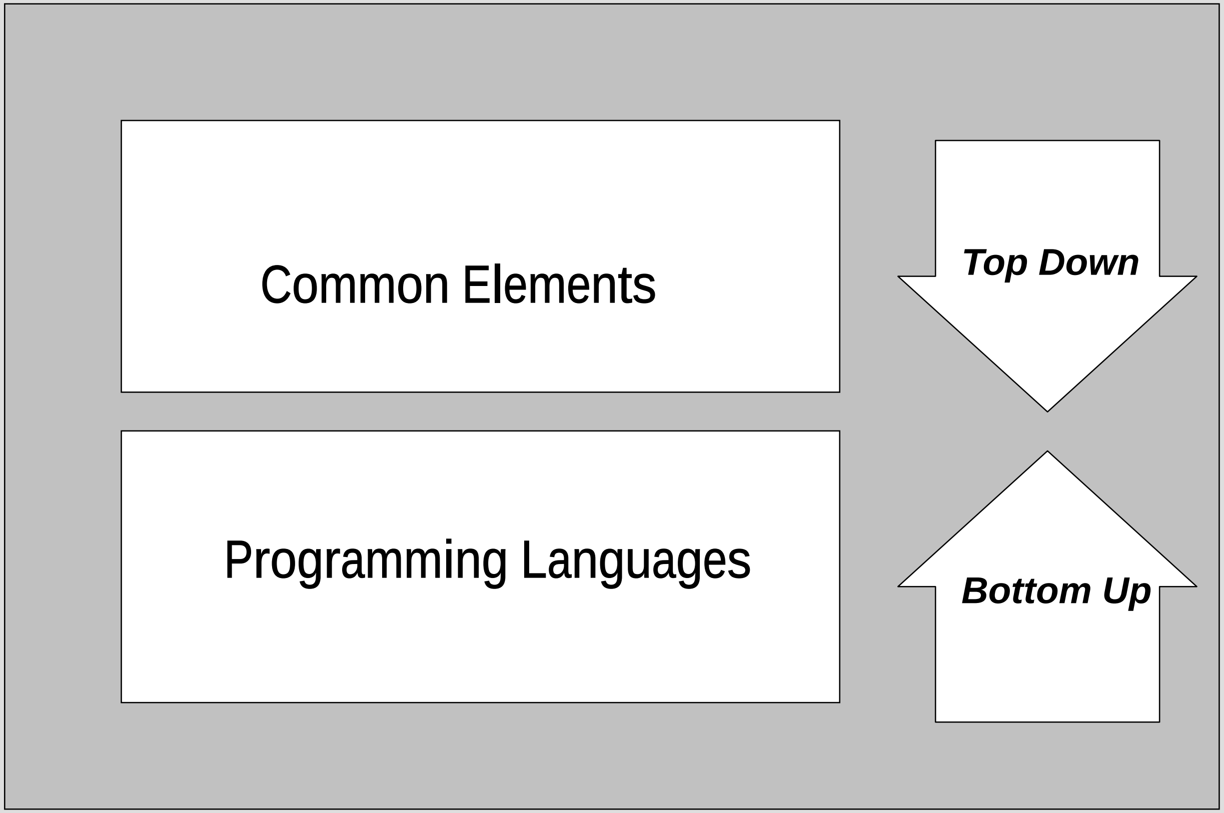
Standard 61131

Programming Languages - examples



Standard 61131

Top-down vs. bottom-up



Standard 61131

<u>Advantages</u>

- reduced waste of human resources, in training, debugging, maintenance and consultancy
- creating a focus to problem solving via a high level of software reusability
- reduced misunderstanding and errors
- programming techniques usable in a broad environment: general industrial control
- combining different components from different programs, projects, locations, companies
- etc.

Standard 61149

The standard IEC 61499 defines how function blocks can be used in distributed industrial process, measurement and control systems.

In industrial systems, function blocks are an established concept for defining robust, re-usable software components.

Standard 61149

IEC 61499 defines a general model and methodology for describing functions blocks in a format that is independent of implementation.

The methodology can be used by system designers to construct distributed control systems.

It allows a system to be defined in terms of logically connected function blocks that run on different processing resources.

Standard 61149

Phases in the design of a DCCS

- Functional design phase

  Process engineers analyze the physical plant design, to create the top-level functional requirements

- Functional distribution phase

  A further design phase is required to define the distribution of control functionality on to processing resources.

Standard 61149

IEC 61499 is a multi-part standard be in development for a number of years.

- Part 1 covers the architecture and concepts for designing and modelling function block oriented systems

- Part 2 addresses the definition of a formal information models that will enable CASE tools and utilities to manipulate and exchange system designs based on function blocks.

Task-level Modeling

The END