2.2 StateCharts

2.2.1 Language Elements

Classical automata: Moore- and Mealy-automata:

- **Moore-automata:**
  represented by two functions λ and δ:
  $\delta : X \times Z \rightarrow Z$
  $\lambda : Z \rightarrow Y$

  X : Set of input values
  Y : Set of output values
  Z : States

  $z^+ = \delta(x, z)$ ist called **next state**,
  with $z, z^+ \in Z$.

- **Mealy automata:**
  output also depends on input, not just on the current state.
  $\delta : X \times Z \rightarrow Z$
  $\lambda : X \times Z \rightarrow Y$

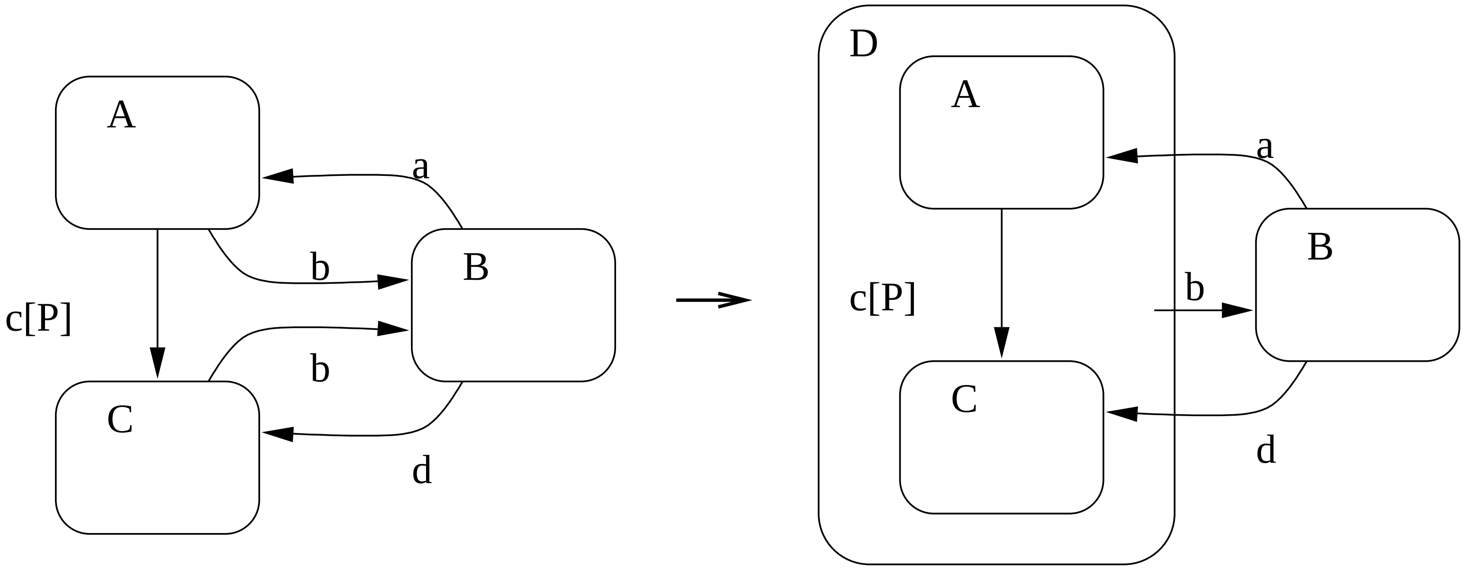
Moore- + Mealy automata =
finite state machines (FSMs).

Introducing Hierarchy

Classical automata not useful for complex systems
→ Introduction of hierarchy in StateCharts (Harel, 1987).

Reason for selecting this name:
StateChart = *the only unused combination of 'flow' or 'state' with 'diagram' or 'chart'*.

States can be grouped into *superstates*.



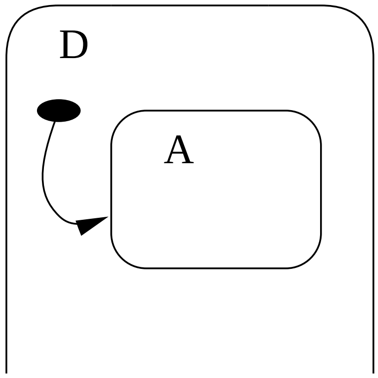
Def.: Superstates of this type are called **(X)OR-States** (system is only in one of the substates).

Def.: States at the lowest level of the hierarchy are called **basic states**.

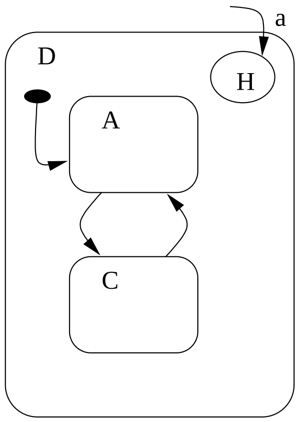
Def.: For each basic state s, the super states containing s are called *anchestor states*.

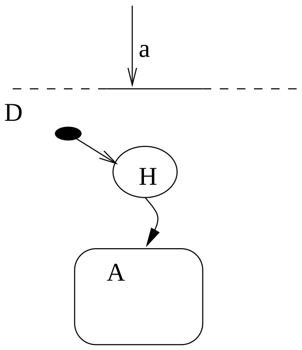
Additional mechanisms for defining the next state:

1. **default state**:

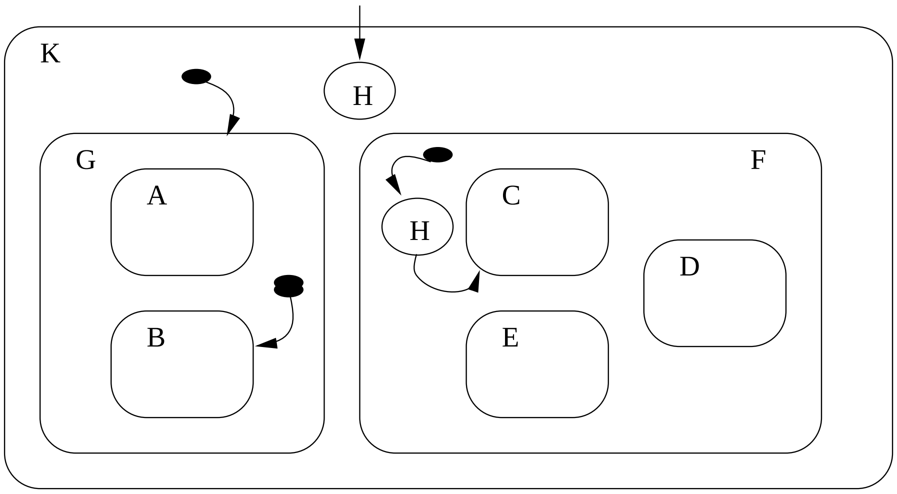


2. **history mechanism** (similar to procedures):



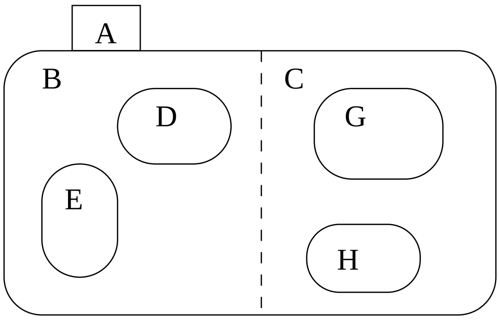


Hierarchical use of history mechanism:



AND-states (orthogonal states):

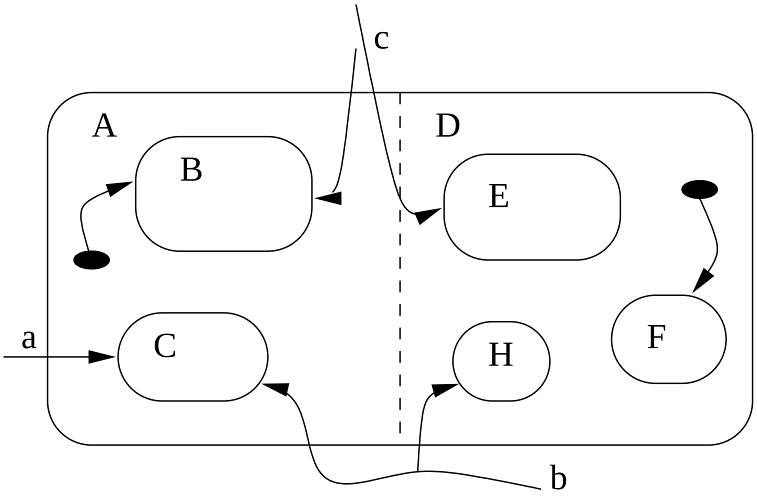
System can be in several substates of a states:



System in states `B` and `C` if `A` has been entered.

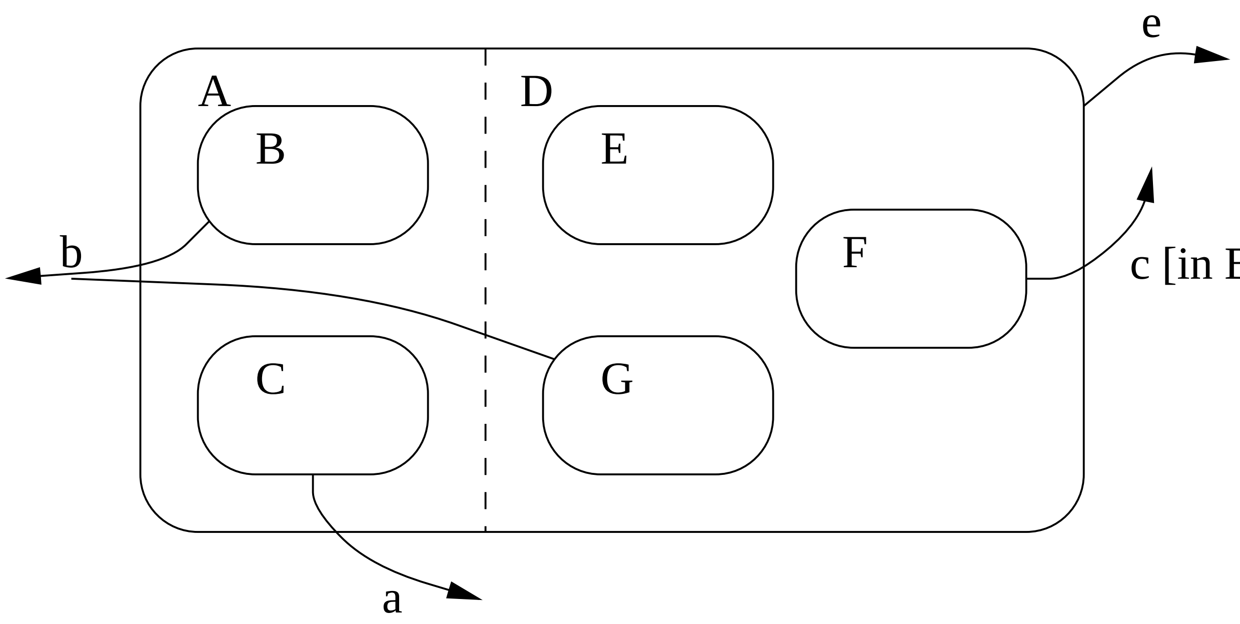
Modelling of concurrency.

Entering AND-states:



- Explicit transistion (transition for `c`)
- history mechanism (transition for `b`)
- default mechanism (initial transistion for `b`)
- mixed (transition for `a`).

Leaving orthogonal states



- Explicit representation of all substates (see transition for `b`)
- Independently of the current substate (see transition for `e`)
- If system is in a certain substate (see transition for `a`).

States in StateCharts are
- AND-States,
- OR-States or
- basic states.

General labelling of edges

$$ev[cond]/react$$

- *ev*: (**events**) are starting **reactions** (possibly state transitions)

  Events exist until the next **step**.

  Events can be generated internally or externally.

  Events $entered(S)$ and $exited(S)$ are generated when `S` is entered or left.

- *cond* is a **condition**.

  Conditions are related to the current state. Typically they exist longer than just until the next step.

  $in(S)$ is a special condition,
  `true` if system is in `S`.

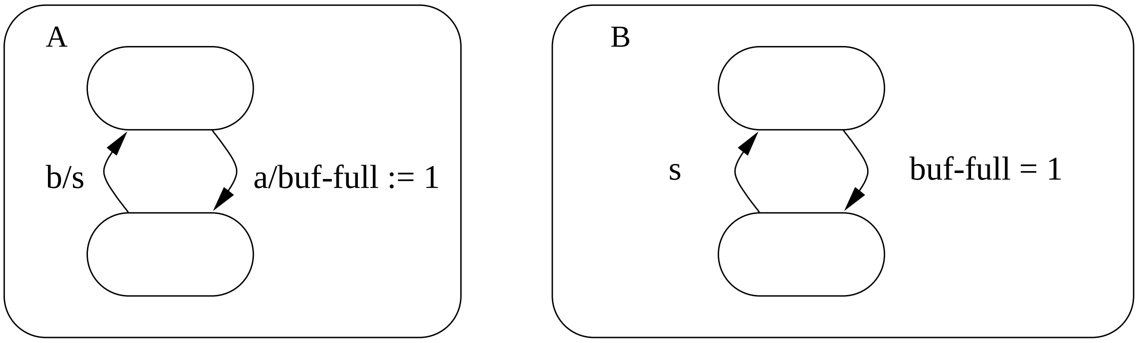
- *react* is a **reaction**.

  Includes generation of events and operations on data.

  Reactions include
  - **actions** (one-shot events, take no time)
  - **activities** (take time)

Broadcast Mechanism

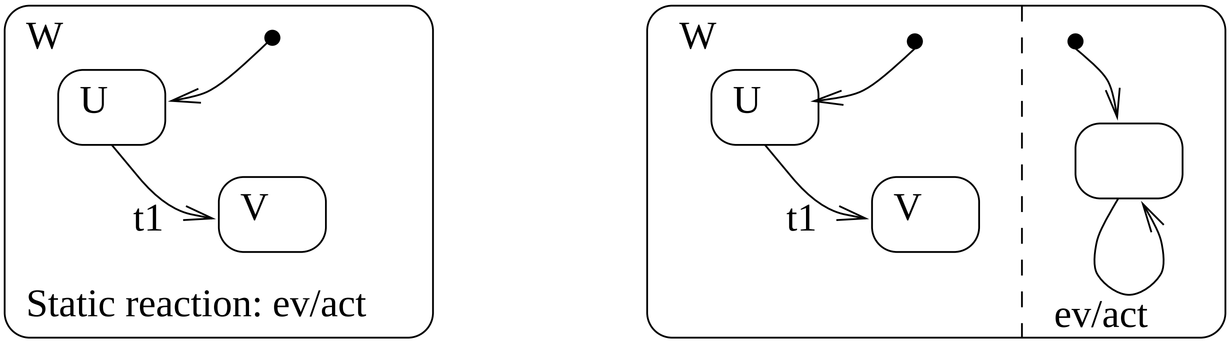
Synchronisation and communication within a model by generation of events or by global access to variables. Example:



`buf-full` is a global variable. Changes are propagated to all references before the next (simulation) step is executed (**broadcast**)

Static reactions:

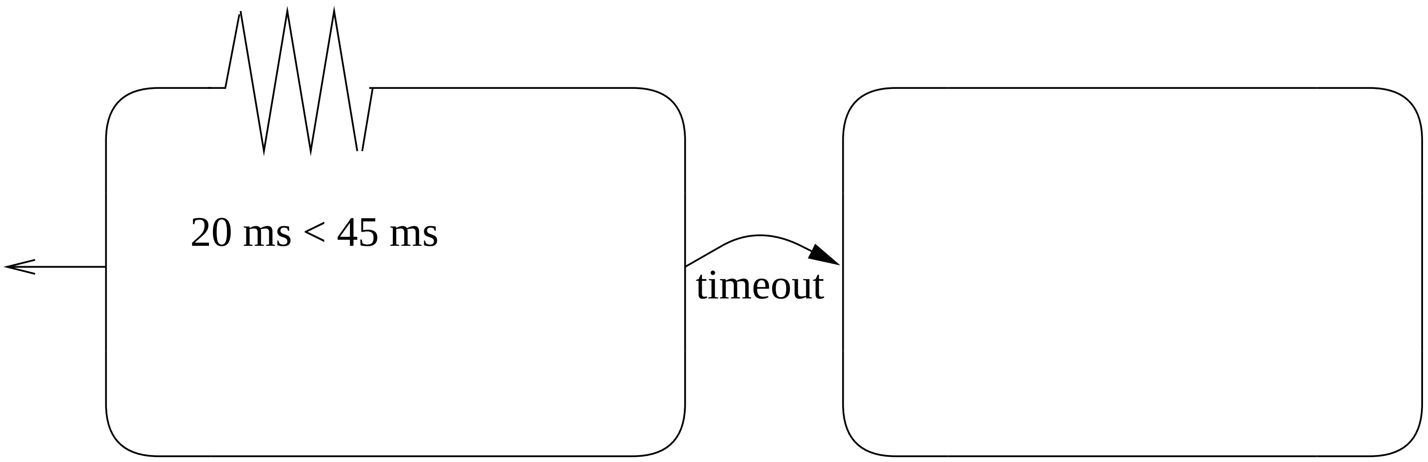
Syntax: like annotations on edges, but within states. Semantics: Reactions of a system within a state. Equivalent to AND-state with annotations on edges. Example: 2 models with same semantics.



Timers

Requirement: modelling of timing behaviour
→ modelling of timers in StateCharts.

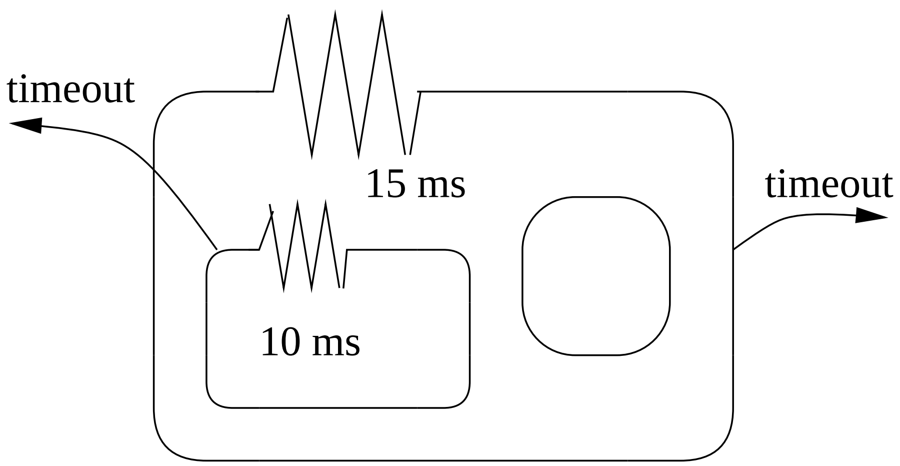
Graphical representation:



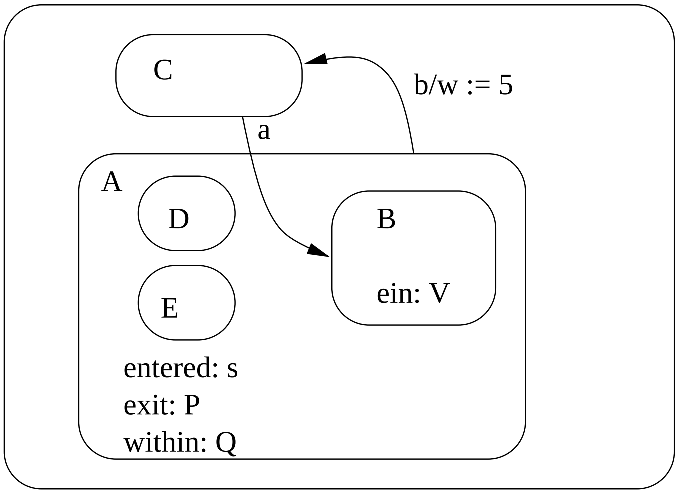
Single time limit: transition after specified time.

Two time values: transition within the time intervall.

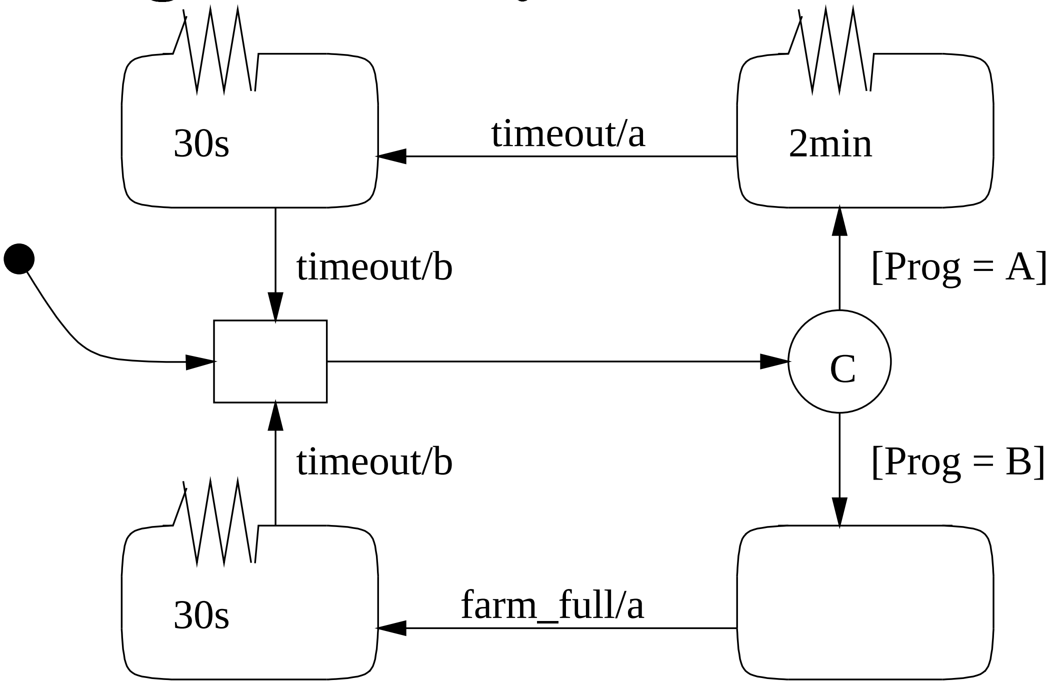
Timing can be specified at various levels of the hierarchy:



Actions and activities



Improving readability with connectors:



Semantics: replacement of C by direct connections.

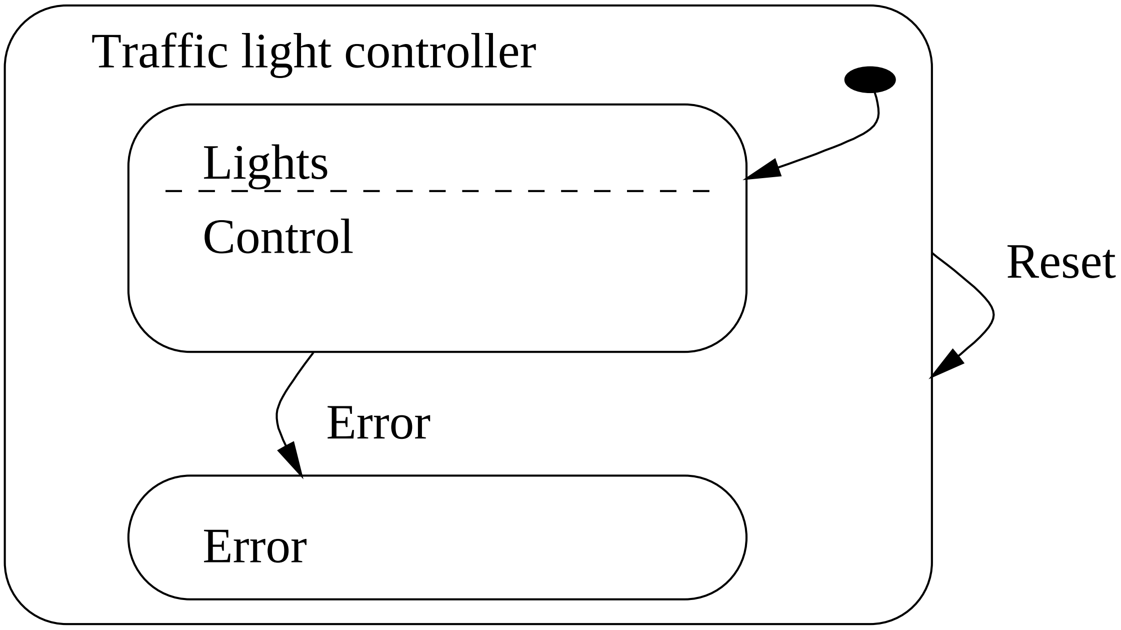
Application example:

Traffic light controller for main road and farm road.

Two programs:

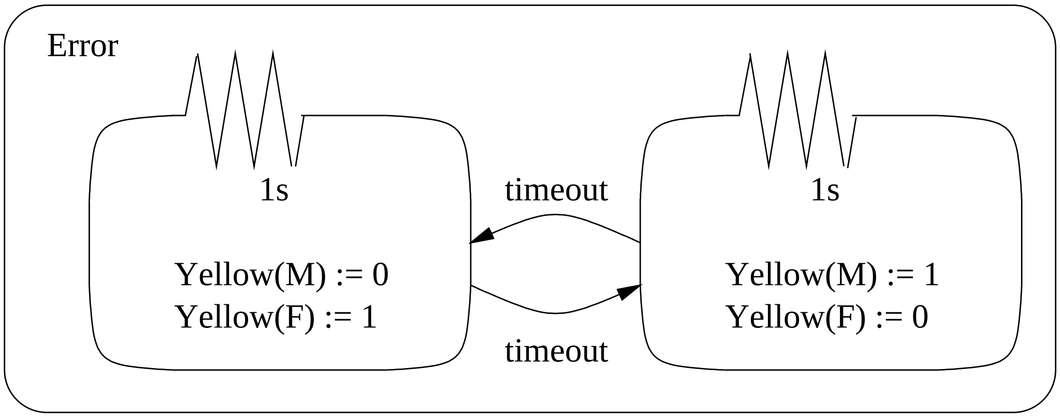
- Program A: Green light for main road for 2 minutes and for farm road for 30 seconds.
- Program B: Green light for main road until farm road signals waiting car.

Systems for normal operating conditions and for error state:

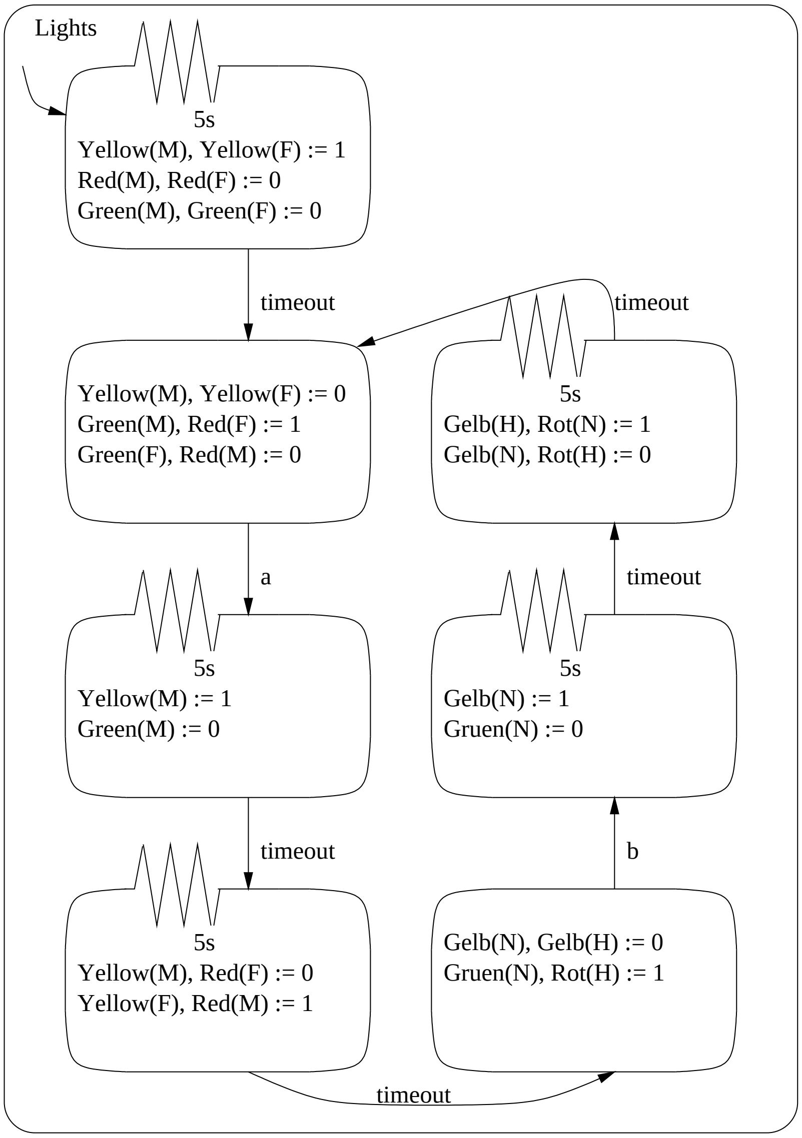


Traffic lights represented by variables $\texttt{Red}_M$, $\texttt{Yellow}_M$, $\texttt{Green}_M$, $\texttt{Red}_F$, $\texttt{Yellow}_F$ und $\texttt{Green}_F$.

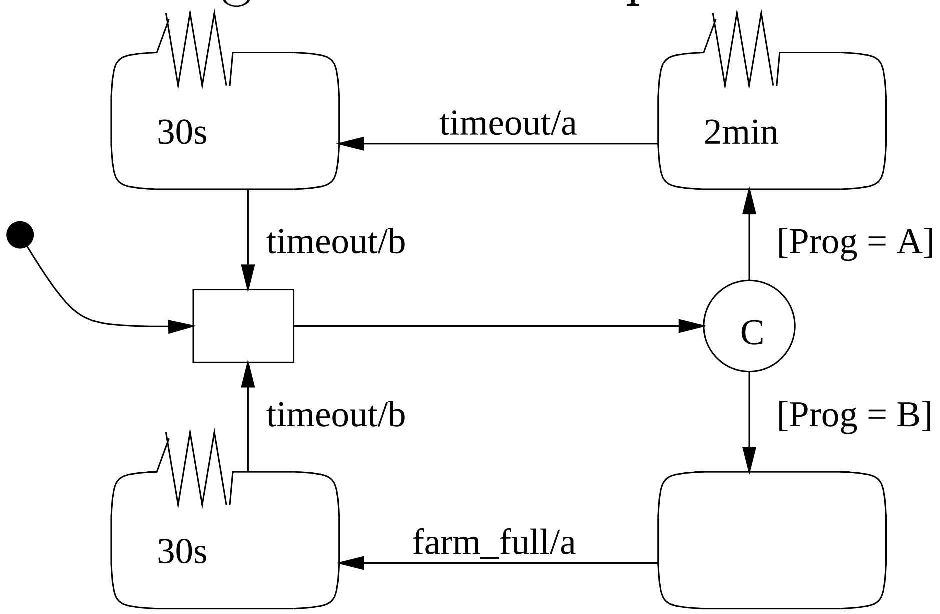
While in the error state, yellow lights are flashing:



Normal operating conditions: all lights yellow initially.



Signals `a` and `b` generated in superstate `control`



Evaluation of StateCharts

Advantages:

Any level of nesting feasible, with a free choice between OR- and AND-states.

Semantical problems removed.

Disadvantages:

No program constructs for describing complex computations.

No method for representing hardware structures.

No representation of non-functional behaviour.

2.2.2 STATEMATE and STATEMATE-Semantics

STATEMATE: very popular commercial product.

STATEMATE can be used to generate VHDL.

Hence, it can be used to generate hardware.

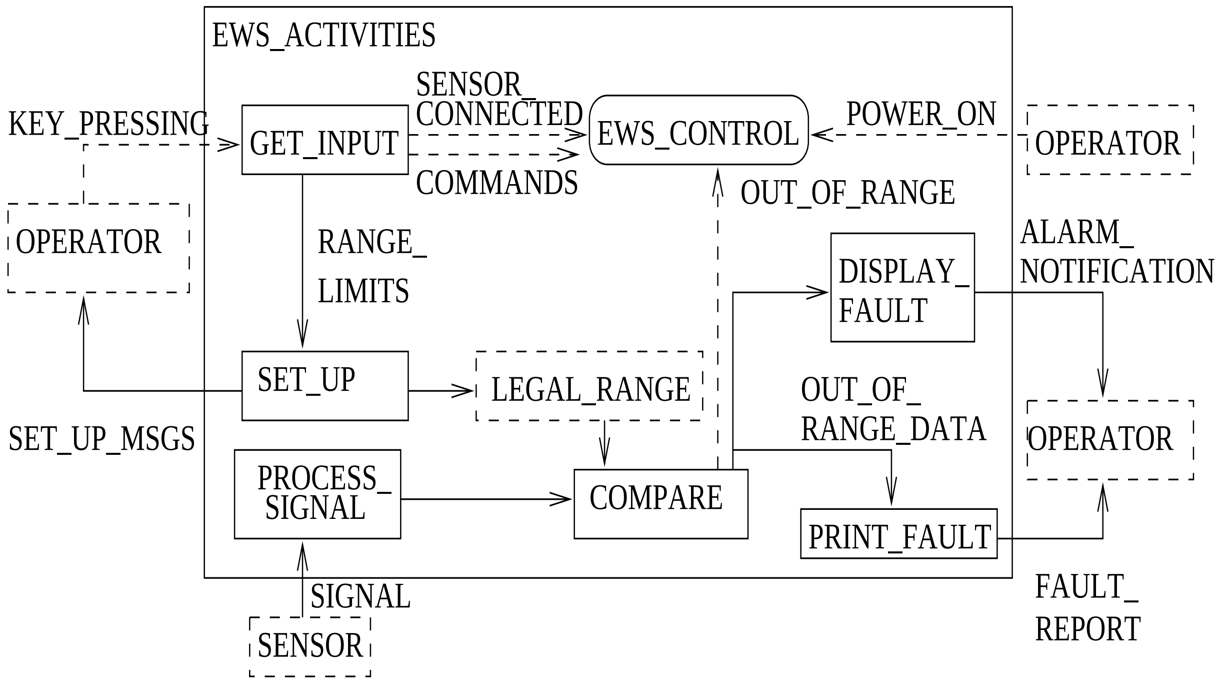
Can also be used to generate C.

2.2.2.1 Activity-Charts and Module-Charts

STATEMATE also supports *activity charts* and *module charts*.

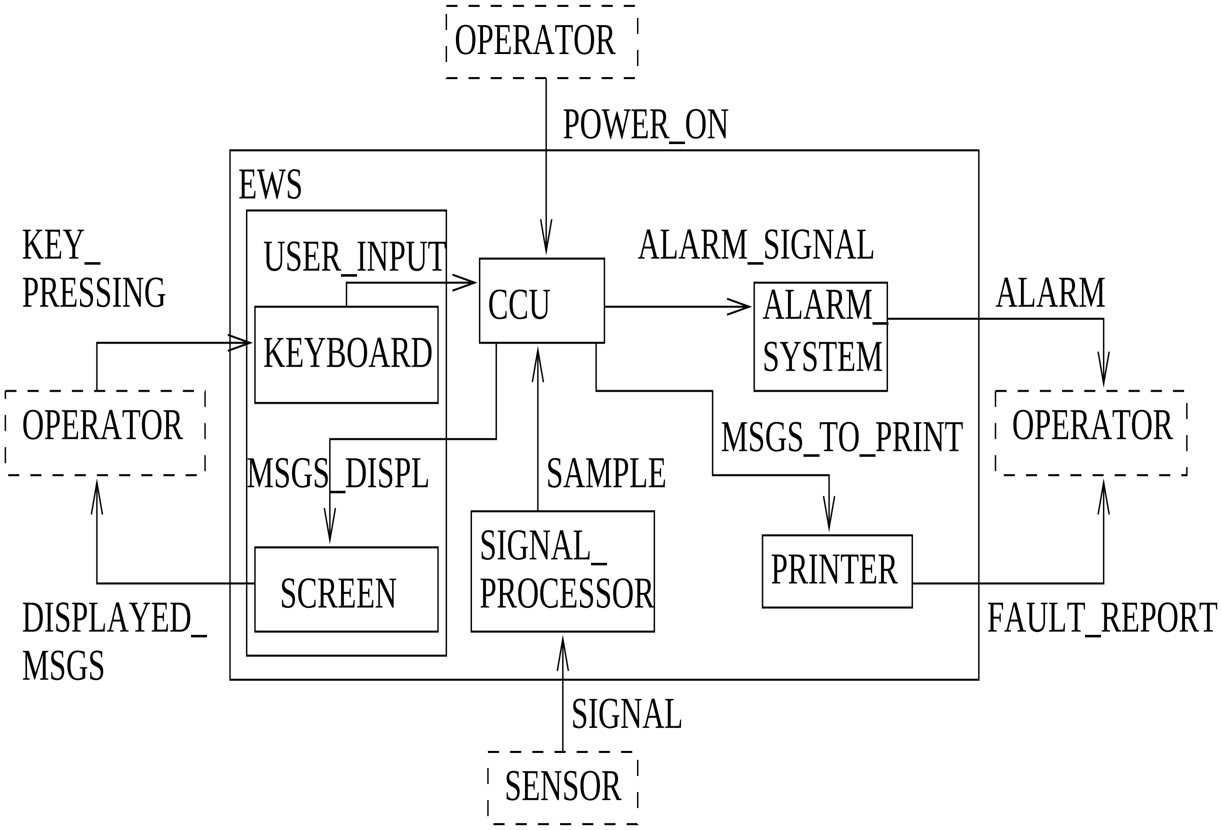
Activity charts can be viewed as multi-level data-flow diagrams. They capture functions, or activities, all organized into hierarchies and connected via the information that flows between them. We adopt extensions that distinguish between data and control information in the arrow types, and also provide several kinds of graphical connectors, as well as a semantics for information that flows to and from non-basic activities.

Figure … illustrates some of these notions … We see internal activities, such as `GET_INPUT` *… and external activities such as* `OPERATOR` *… data flows such as* `RANGE_LIMITS` *… control flows, such as* `COMMANDS` *and the control activity* `EWS_CONTROL`.



A module-chart can also be regarded as a certain kind of data-flow diagram or block diagram. Module-charts are used to describe the modules that constitute the implementation of the system, its division into hardware and software blocks and their inner components, and the communication between them.

Fig. ... shows a module chart for the EWS. It contains internal moduels such as the control and computation unit (`CCU`*), ...*

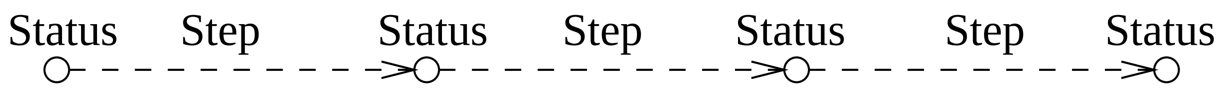


Consistency of different diagrams?

2.2.2.2 Status and step

Each situation of a STATEMATE-system ist represented by the **status**.

The generation of the next status from the current status requires a **step** to be executed.



Executing a step takes 0 time.

Length of time interval between two steps not part of formal semantics.

Principles of formal semantics:

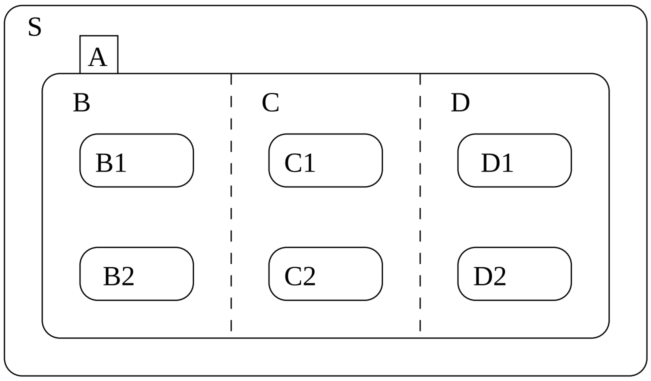
1. Consequences of executing a step can only be observed after the step has been executed.
2. Events only exist until the step following their generation has been executed.
3. All computations are based on the situation before executing the step.
4. Execution will always consist of a maximal non-conflicting set of transitions and static reactions.

STATEMATE supports timing models, for which several steps are executed at the same time.

Def. : Configurations are maximal feasible sets of states.

Configurations are defined by the basic states of a system: sich ein System befindet.

Example:



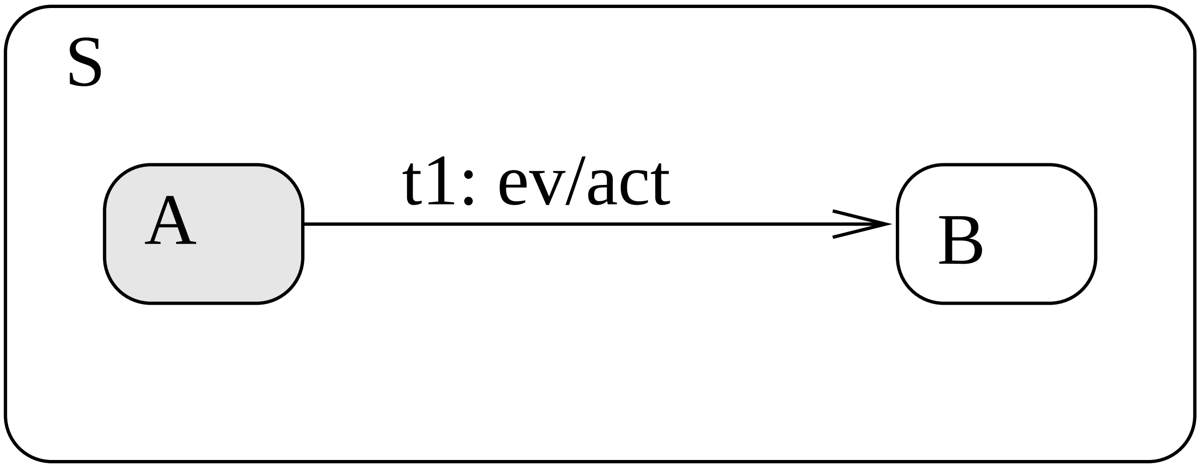
System can be in basic states `B1, C1` und `D1`.

Configuration contains predecessor states `A` und `S`.

Simple Transitions

Semantics of STATEMATE for simple transitions is defined as follows:

Assume a system to be in basic state `A` and event `ev` has taken place:



The following will be executed:

- Transition `t1` takes place, the new state is `B`.
  Action `act` takes place.
- Events `exited (A)` and `entered (B)` are generated.
- Condition `in (A)` becomes false, `in (B)` becomes true.
- Action to be executed when entering `B` are executed.
- Static reactions for `S` are executed.
- Activities specified as '`within(A)`' are deactivated.
  Activities specified as '`throughout (B)`' are activated.

Events and conditions depending on these changes via functions also take place.

All changes become effective for the next step.

The simulation algorithm

Input:

- current status
- current time,
- list of changes since last step

Output:

new status

3 phases:

1. Execution of all conditions becoming necessary due to changes since last step.

   Includes data computations, but no new state.
2. Computation of a maximal set of transitions that can be executed and that are conflict free.
3. Execution of transitions.

Separation into phases 2 and 3 makes it impossible to have changes that affect changes for the same step.