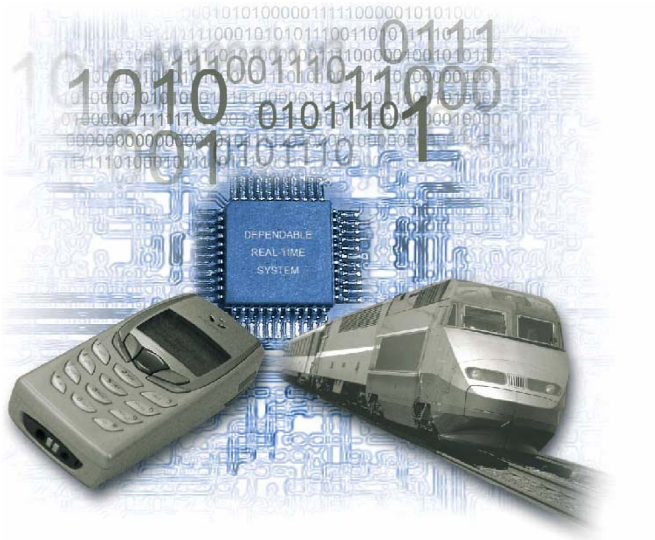
Embedded Systems 2



Exam Policy

- Midterm/End-of-Term Exam/End-of-Semester Exam

Requirement for admission to end-of-term and end-of-semester exams:

- > 50% of points in homeworks and
- > 50% of points in midterm exam

Final grade:

- best grade in end-of-term or end-of-semester exam

Note: exam policy has been modified to ensure consistency with module description.

StateCharts

REVIEW

- StateCharts = the only unused combination of „flow“ or „state“ with „diagram“ or „charts“
- Based on classical automata (FSM):
  StateCharts = FSMs + Hierarchy + Orthogonality + Broadcast communication
- Industry standard for modelling automotive applications
- Appear in UML (Unified Modeling Language), Stateflow, Statemate, …
- *Warning: Syntax and Semantics may vary.*

Duration of computations

- Basic semantic problem: “uncooperative environment” [Koymans, Kuiper, Zijlstra 1988] proceeds at its own, asynchronous pace

=>Proceeds during computations, data sampling, etc., of the embedded system

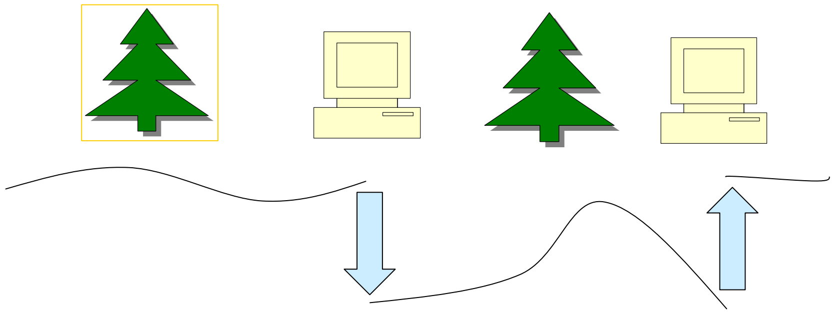
- Induces design decisions for specification formalisms:
  - How much time shall computational actions of the ES take?
  - Shall data sampling take time; if so, how much?
  - What happens to data sampling upon fast environment dynamics?

Duration of computations

- Engineers' familiarity with automata-based design calculi for synchronous circuits makes reuse of the computational model of Mealy automata attractive:
  - Input sampling is instantaneous
  - State changes are instantaneous
  - Output delivery is instantaneous
  - All three happen in the same physical time instant
  - These instants of computational action are separated by phases of idling, where the automaton state is constant
- This abstraction of computation time being negligible has become known as the "synchrony hypothesis"
  - Frees early design stages from worries about implementation details

Semantics of StateCharts

- Execution of a StateChart in its environment consists of instantaneous StateChart actions interspersed by durational environment actions



Hierarchy

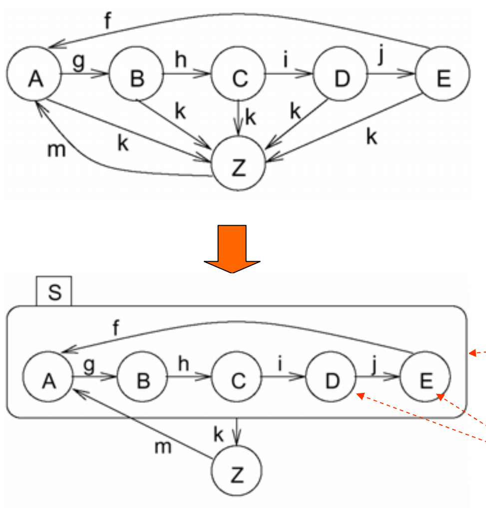
REVIEW

In StateCharts, states are either

- **basic states,** or
- **AND-super-states,** or
- **OR-super-states.**

OR-super-states

REVIEW



FSM will be **in** exactly one of the substates of S if S is **active** (either in A or in B or ..)

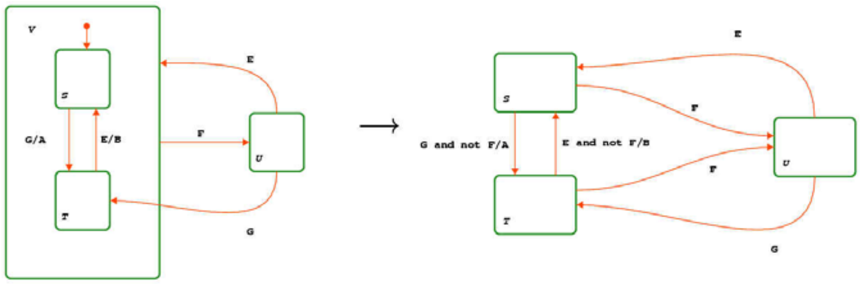
superstate

substates

Priority rules

REVIEW

- Priority of „higher level“ transitions over „lower level“ transitions

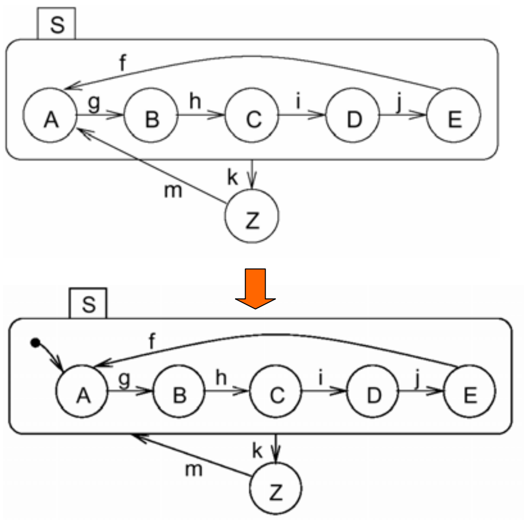


OR-type hierarchy can be explained by flattening out the hierarchical diagram.

Default state mechanism

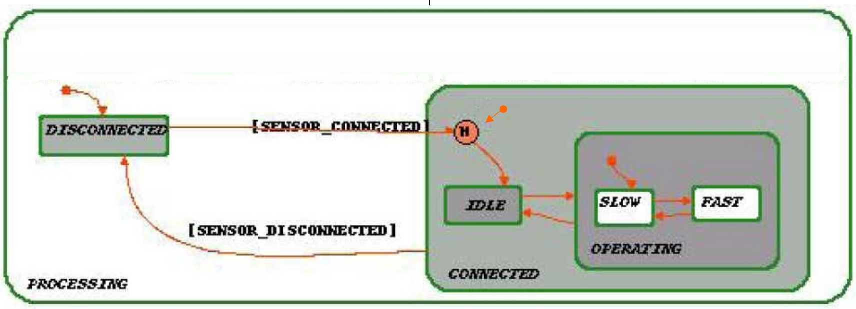
REVIEW

- Filled circle indicates sub-state entered whenever super-state is entered.
- Not a state by itself!
- Allows internal structure to be hidden for outside world



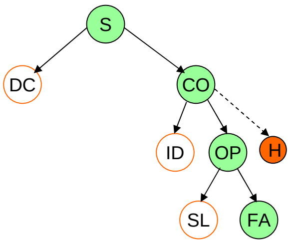
History

REVIEW



Default states

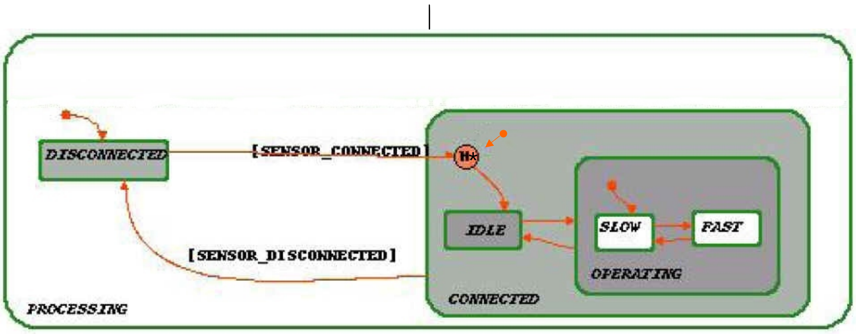
Active states



History connectors remember states **at the same level** as the history connector!

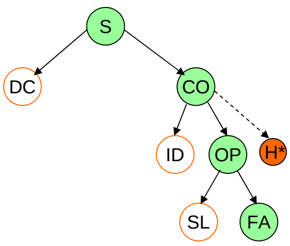
Deep history

REVIEW



Default states

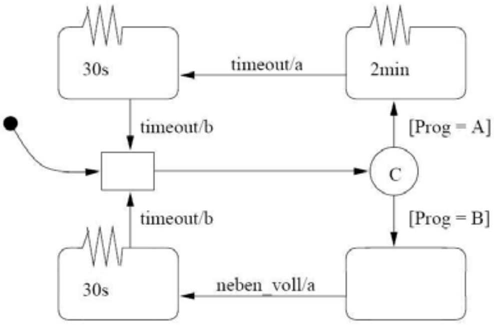
Active states



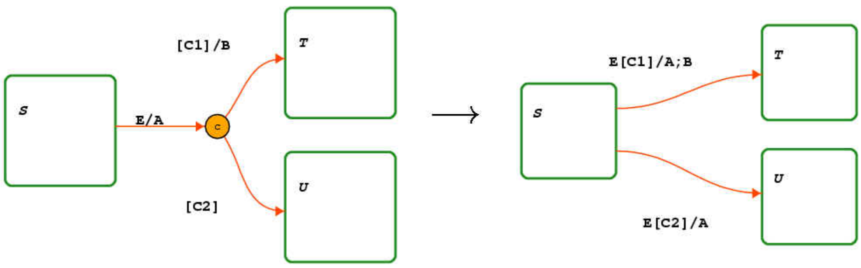
Deep history connectors H* remember **basic states**!

Connectors

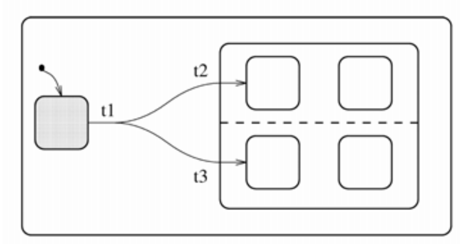
- Example: Traffic light control with two programs

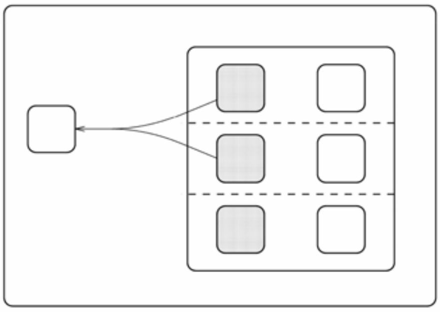


Condition connector

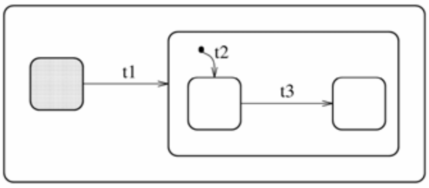


Join and Fork Connectors

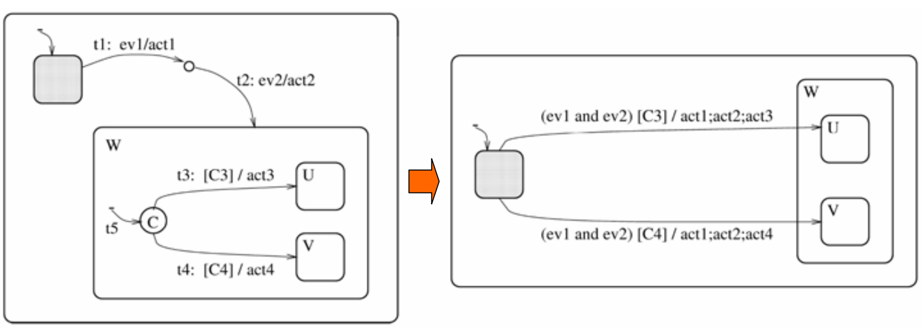




Compound transitions

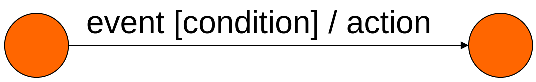


t1 and t2 must be executed together



General form of edge labels

REVIEW



Meaning:
- Transition may be taken, if event occurred in last step and condition is true
- If transition is taken, then reaction is carried out.

Conditions:
- Refer to values of variables

Actions:
- Can either be assignments for variables or creation of events

Example:
- a & [x = 1023] / overflow; x:=0

Variables with complex data types

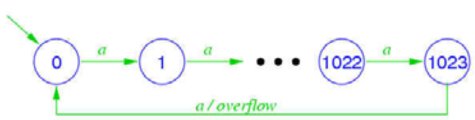
REVIEW

Problem of classical automata:
- Both control and data have to be represented as graphical states

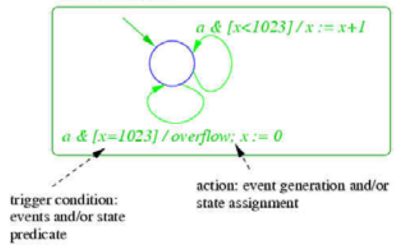
Here:
- Include typed variables (e.g. integers, reals, strings, records) to represent data
- Both „graphical states“ and variables contribute to the state of the statechart.
- Notation:
  - „graphical states“ = states
  - „graphical states“ + variables = **status**

A 10-Bit counter, counting on event *a* and issuing *overflow* after 1024 occurrences:

As FSM:



As Statechart:



Events, conditions, actions

REVIEW

- Possible events (incomplete list):
  - Atomic events
    - Basic events: A, B, BUTTON_PRESSED
    - Entering, exiting a state: en(S), ex(S)
    - Timeout events
    - …
  - Compound events: logical connectives and, or, not
- Possible conditions (incomplete list):
  - Atomic conditions
    - Constants: true, false
    - Condition variables (i.e. variables of type boolean)
    - Relations between values: $X > 1023$, $X \leq Y$
    - Residing in a state: in(S)
    - …
  - Compound conditions: logical connectives and, or, not

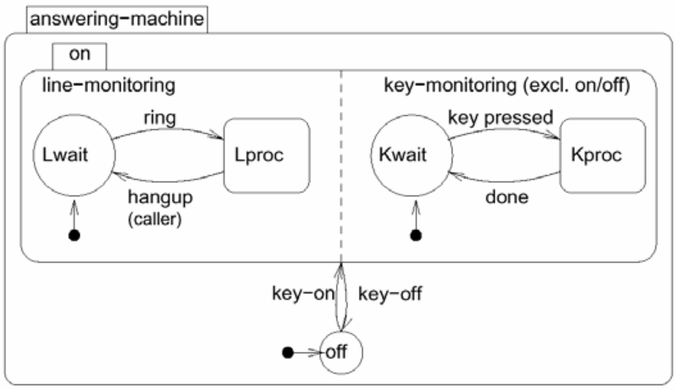
Events, conditions, actions

REVIEW

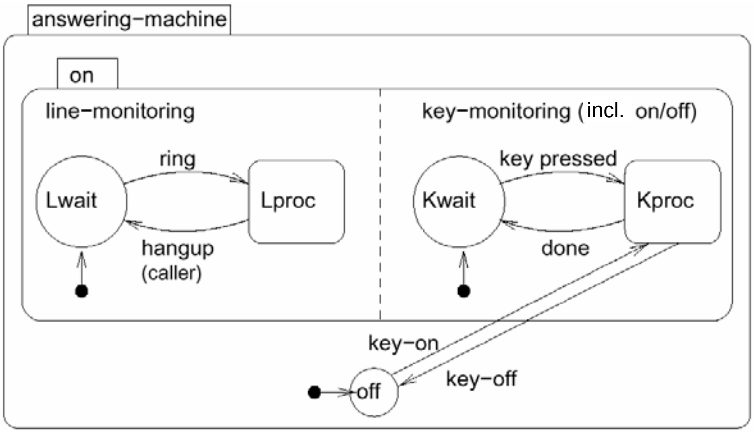
- Possible actions (incomplete list):
  - Atomic actions
    - Emitting events: E (E is event variable)
    - Assignments: X := expression
    - Scheduled actions: sc!(A, N)
      (means perform action after N time units)
  - Compound actions
    - List of actions: A1; A2; A3
    - Conditional action: if cond then A1 else A2

Concurrency REVIEW

- **AND-super-states**: FSM is in **all** (immediate) sub-states of a AND-super-state; Example:



Entering and leaving AND-super-states REVIEW

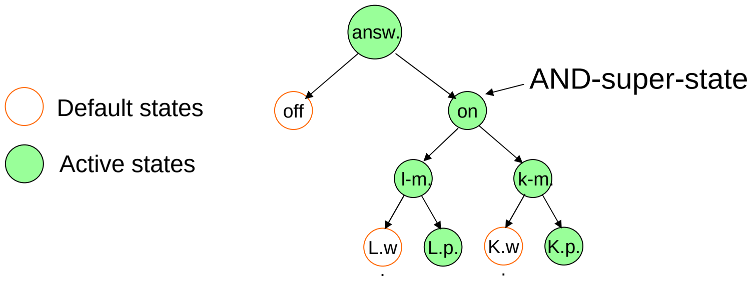


- Line-monitoring and key-monitoring are entered and left, when key-on and key-off events occur.

Concurrency

REVIEW

- Example for active states:



- Classical automata have to compute product automata to express concurrency

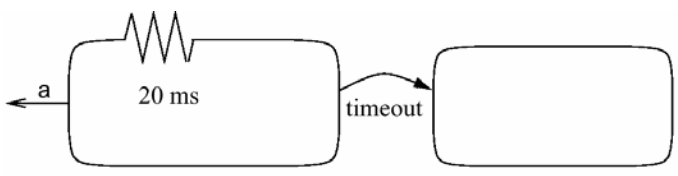
⇒ structural information is lost

⇒ increase in size

Timers

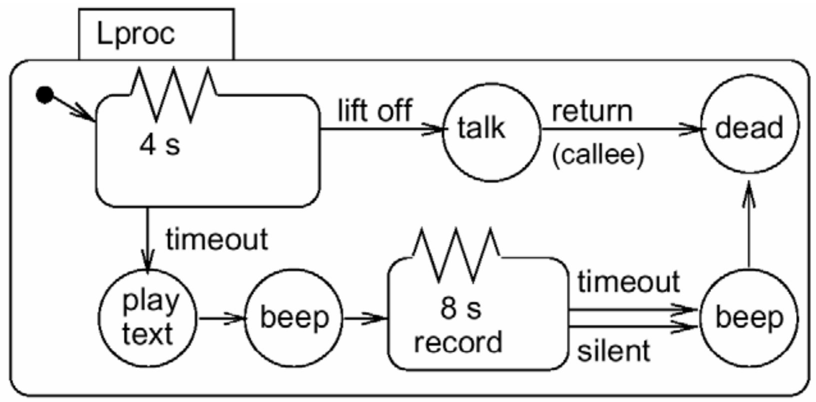
REVIEW

- Since time needs to be modeled in embedded systems, timers need to be modeled.
- In StateCharts, special edges can be used for timeouts.



If event a does not happen while the system is in the left state for 20 ms, a timeout will take place.

Using timers in answering machine **REVIEW**



Timeout events (general)

- Timeout event tm(e,d):
  - Timeout event tm(e, d) is emitted d time units after event e has occured

  ⇒ a timer can be simulated by a state S where the timers‘ timeout events are replaced by tm(en(S), d)

Example: Chess Clock

External events:

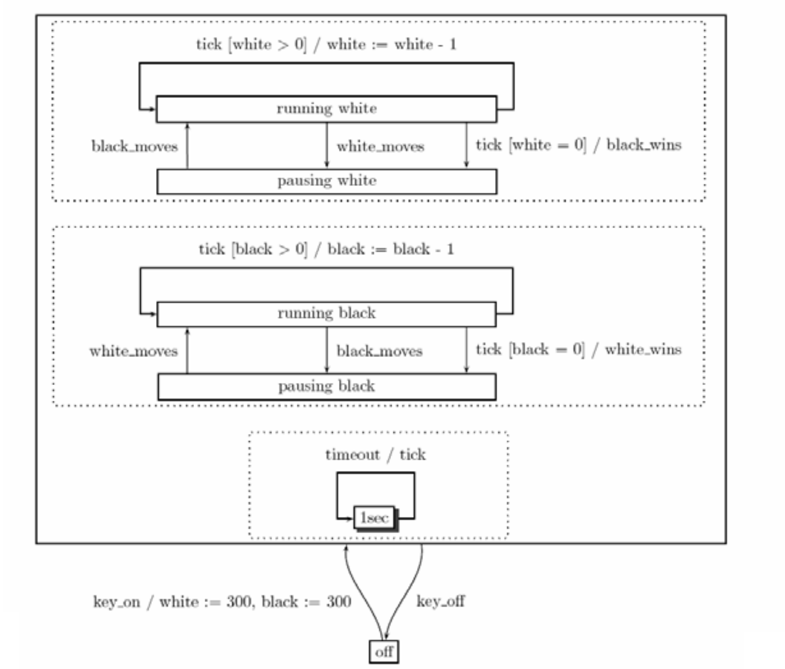
- key_on
- key_off
- black_moves
- white_moves

Internally generated events:

- white_wins
- black_wins
- tick

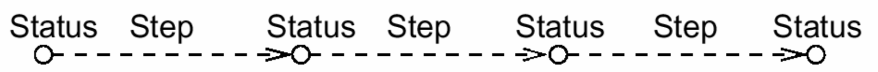
Variables:

- white
- black



Semantics of StateCharts

- Execution of a StateChart model consists of a sequence of **steps**
- A step leads from one **status** to another



- One step:
  - Given:
    - Current system status s_i
    - Current time t
    - External changes Δ
  - Find:
    - New status s_{i+1}

External changes

- External data and external events constitute the interface between system and environment.
- The environment provides external events at certain times and changes external data at certain times.
- External events not yet seen in the previous step and changes of external data not seen in the previous step are called **external changes** for the current step.

Status of the system

The current status of the system is given by

- set of active states
- current values of variables
- the generated events from previous step
- the values of the history connectors
- set of all timeout events <tm(e, d), n> in the state chart with „emission times“ n (times n are initially set to ∞)
- set of currently scheduled actions <sc(a, d), n> with their times n

Overview of a step

- Two stages
  - Preparation (for timeout events and scheduled actions)
  - Execution
- Preparation stage:
  - Fix scheduled actions that will be executed
  - Fix timeout events that will be generated
- Execution stage:
  - Determine the set of transitions to be taken based on internal and external events and on values of internal and external variables
  - Compute the next states and the reactions (evaluate right hand sides of assignments)
  - Transitions become effective, variables obtain new values.

Preparation stage of step at time *t*

- Scheduled actions
  - For all currently scheduled actions *<sc(a, d) , n)* (i.e. actions scheduled but not yet executed):
    - If $n \leq t$ then execute action *a*

    (execution may lead to new events and changes of variables)
- Timeout events
  - For all timeout events *<tm(e, d), n>* in set of timeout events
    - If e is external event not yet seen in previous step or internal event generated in previous step then $n := t + d$ (current time is t) („schedule timeout event")
    - Else: If $n \leq t$ then emit event *tm(e,d)* and reset *n* to ∞

Execution stage – first part

- Determine the maximal set of transitions to be taken based on
  - internal and external events and
  - on values of internal and external variables ($\rightarrow$ conditions!)
- Due to concurrency (AND-states) a transition of a **set of** states to a **set of** states is computed.
- Due to non-determinism several choices for the set of next states are possible

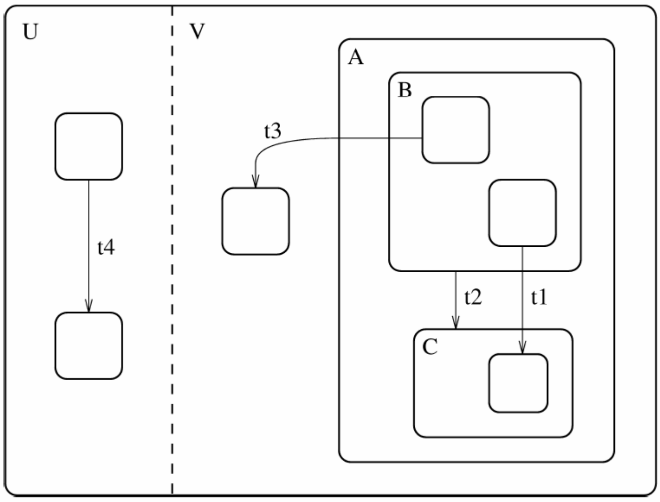
  $\Rightarrow$ non-deterministic choice!

  $\Rightarrow$ each choice represents one possible behaviour of the system

  $\Rightarrow$ The same StateChart with the same sequence of external changes may have several possible status sequences

  Here: Select one subset of enabled transitions leading to a set of basic states.

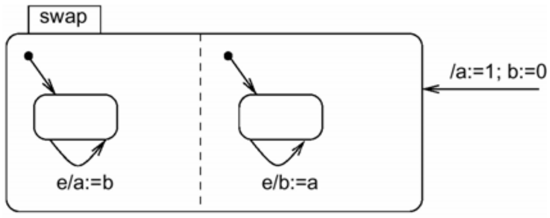
Conflicting transitions



Execution stage – second and third part

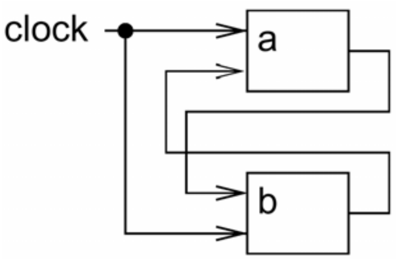
- Second part: Compute
  - the next states and
  - the reactions
    - Generate events for the next step
    - Evaluate right hand sides of assignments, but do not perform assignments yet
- Third part:
  - Transitions become effective:
    - assignments are actually made, i.e. variables obtain new values.
    - History connectors are updated.
    - Next states become active.
- Separation into parts 2 and 3 guarantees deterministic and reproducible behavior of parallel assignments.

Example



- In part 2, variables a and b are assigned to temporary variables. In part 3, these are assigned to a and b. As a result, variables a and b are swapped.
- Without this separation, executing the left state first would assign the old value of b (=0) to a and b. Executing the right state first would assign the old value of a (=1) to a and b. The execution of parallel assignment would be nondeterministic.

Reflects model of clocked hardware



- In an actual clocked (synchronous) hardware system, both registers would be swapped as well.

Same separation into phases found in other languages as well, especially those that are intended to model hardware.

Broadcast mechanism

- Values of variables are visible to all parts of the StateChart model.
- New values become effective in part 3 of the execution stage for the current step and are obtained by all parts of the model in the following step.

☞ StateCharts implicitly assumes a **broadcast** mechanism for variables.

☞ StateCharts is appropriate for local control systems (☺), but not for distributed applications for which updating variables might take some time (☹).

Time models

- External events and external changes of variables are associated with physical times.
- But how does time proceed internally?
- How many steps are performed before external changes are evaluated?

The synchronous time model

- A single step every time unit.
- If the current step is executed at time *t*, then the next step is executed at time *t+1*.

⇒ Events and variable changes are communicated between different states during one time unit.

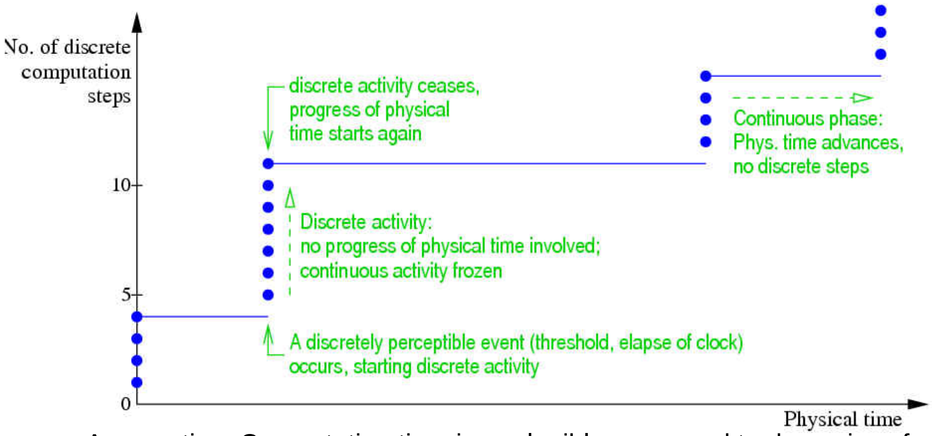
⇒ External changes are only accumulated during one time unit.

The super-step time model (1)

- A step of the statechart does not need time.
- Super-steps are performed:
  - A super-step is a sequence of steps.
  - A super-step terminates when the status of the system is stable.
  - During a super-step the time does not proceed and thus external changes are not considered.
- After a super-step, physical time restarts running, i.e. activity of the environment will be possible again.
- The computation of the statechart is resumed when
  - external changes enable transitions in the statechart
  - Timeout events enable transitions of the statechart

The super-step time model (2)

- Two-dimensional time:



- Assumption: Computation time is neglegible compared to dynamics of the environment.

The super-step time model (3)

- During one super-step the number of communications between different states is not restricted. All communications are assumed to be performed in zero time.
- Simplified model for reality.
- Can only be realistic, if
  - Discrete computations are fast compared to dynamics of the environment.
  - Discrete computations will be stable after a restricted number of steps.
- Timeout events can reactivate a statechart

  $\Rightarrow$ Possible to specify statecharts which permit progress of physical time after a limited number of steps and reactivate themselves via timeout events

Evaluation of StateCharts (1)

Pros:

- Hierarchy allows arbitrary nesting of AND- and OR-superstates.
- Formal semantics (defined in a follow-up paper to original paper).
- Large number of commercial simulation tools available (StateMate, StateFlow, BetterState, ...)
- Available „back-ends“ translate StateCharts into C or VHDL, thus enabling software or hardware implementations.

Evaluation of StateCharts (2)

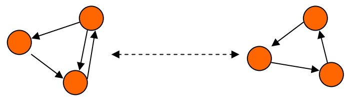
Cons:

- Generated C programs frequently inefficient,
- Not useful for distributed applications,
- No program constructs,
- No description of non-functional behavior,
- No object-orientation,
- No description of structural hierarchy.

Some general properties of languages
1. Synchronous vs. asynchronous languages

- Description of several (concurrent) processes in many languages non-deterministic:
  The order in which executable tasks are executed is not specified (may affect result).
- Synchronous languages: based on automata models. They describe concurrently operating automata. When automata are composed in parallel, a transition of the product is made of the "simultaneous" transitions of all of them.
- Synchronous languages implicitly assume the presence of a (global) clock. Each clock tick, all inputs are considered, new outputs and states are calculated and then the transitions are made.

Some general properties of languages
1. Synchronous vs. asynchronous languages



- This requires a broadcast mechanism for all parts of the model.
- Idealistic view of concurrency.
- Has the advantage of guaranteeing deterministic behavior.

- Statechart steps work synchronously.
  - Broadcast of events and variable changes during each step.

  ⇒ **StateCharts is a synchronous language.**

  ⇒ StateCharts are deterministic, if priority rules are introduced for transitions enabled at the same time.

Some general properties of languages
2. Properties of processes

- **Number of processes**
  static (suitable for hardware);
  dynamic (dynamically changed hardware architecture?)
- Nested **declaration of processes**
  or all declared at the same level
- ☞ **StateCharts comprises a static number of processes and nested declaration of processes.**

Some general properties of languages
3. Communication paradigms

- **Message passing**
  - **Asynchronous message passing = non-blocking communication**
    Sender does not have to wait until message has arrived; potential problem: buffer overflow
  - **Synchronous message passing = blocking communication, *rendez-vous*-based communication**
    Sender will wait until receiver is ready for receiving message ("point of communication")
  - **Extended *rendez-vous***
    Explicit acknowledge from receiver required. Receiver can do checking before sending acknowledgement.

Some general properties of languages
3. Communication paradigms

- **Shared memory**
  Variables accessible to several tasks
  - Problem: Concurrent write.
  - Critical sections = sections at which exclusive access to some resource *r* must be guaranteed.

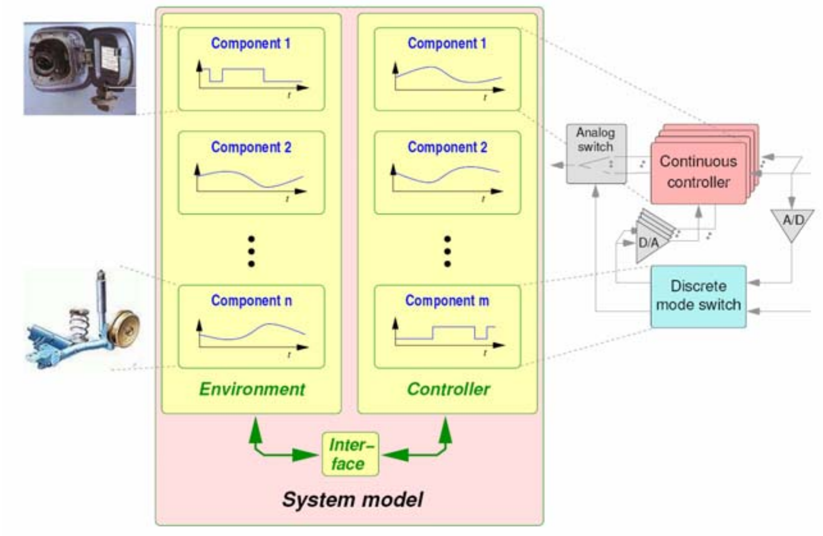
☞ **StateCharts uses shared memory for communication between processes.**

Some general properties of languages
4. Specifying timing

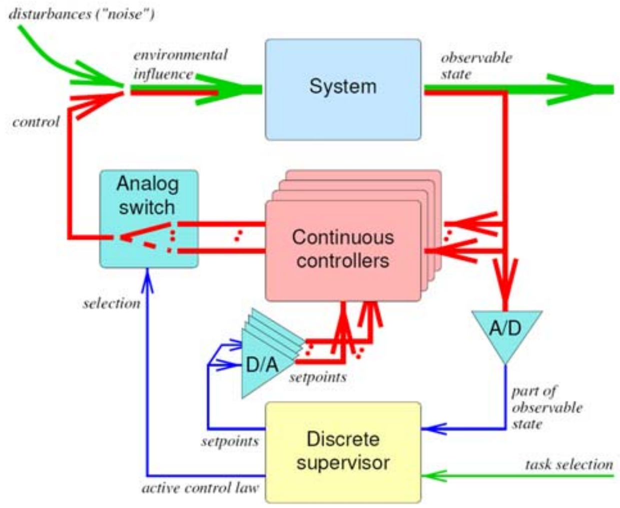
4 types of timing specs required [Burns, 1990]:
- **Measure elapsed time**
  Check, how much time has elapsed since last call
- **Means for delaying processes**
- **Possibility to specify timeouts**
  We would like to be in a certain state only a certain maximum amount of time.
- **Methods for specifying deadlines**
  With current languages not available or specified in separate control file.

☞ **StateCharts comprises a mechanism for specifying timeouts. Other types of timing specs are not supported.**

Hybrid Systems



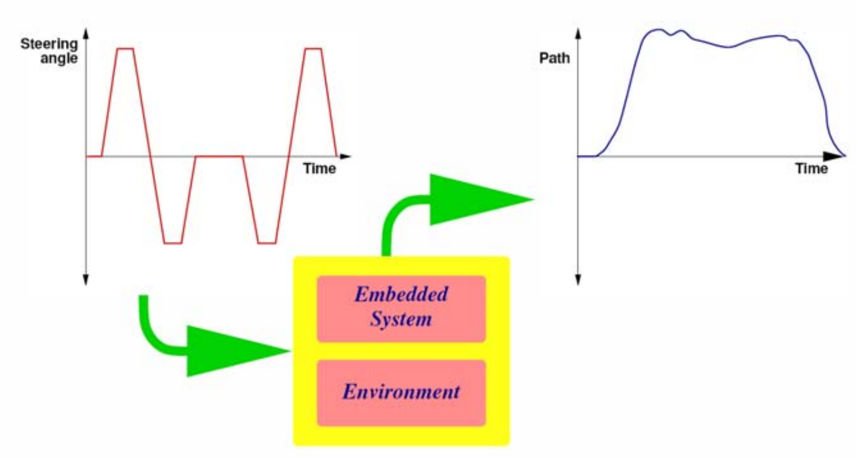
Mode-switching control



Why environment models?

- Testing the embedded system without mimicking (an approximation of) its environment behaviour may lead to
  - design effort invested into unreasonable or even impossible interaction scenarios,
  - overlooking critical scenarios,
  - impossibility to assess criticality of a situation.
- Testing the embedded system within (a prototype of) the real environment
  - may incur intolerable risk,
  - may incur intolerable cost,
  - is often impossible due to the schedule of product design,
  - is more expensive when exploring the design space.

Virtual Prototyping



Why mathematical models?

Mathematical modelling is only approximative due to

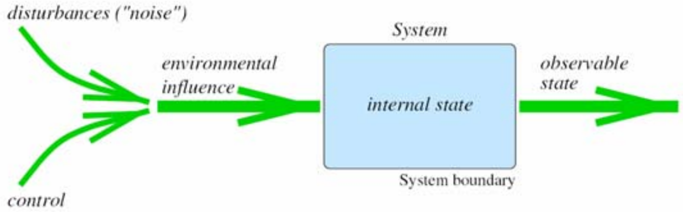
- unknown parameters,
- numerical instabilities of simulation algorithms,
- most dynamic models being only approximate.

It is nevertheless more accurate than mechanical modelling, as a scale $\frac{1}{10}$ model of e.g. a car

- has $\frac{1}{10}$ the length of the original, travelling with $\approx \frac{1}{10}$ the speed,
- has $\frac{1}{1000}$ the volume of the original, thus $\approx \frac{1}{1000}$ of its mass,
- Consequently, it has $E_{kin} = \frac{1}{2}mv^2 \approx \frac{1}{2}\frac{m_{orig}}{1000}\left(\frac{v_{orig}}{10}\right)^2 = 10^{-5}E_{kin,orig}$ kinetic energy.

$\Rightarrow$ Testing ESP, ..., on such a device makes no sense.

Open dynamical systems



- Time is continuous: $\mathbb{R}_{\geq 0}$,
- internal state is a bunch of real-valued (or complex-valued) functions of time: $\vec{x}(.) : \textit{Time} \longrightarrow \mathbb{R}^n$,
- observable state is a time-invariant function (usually projection) thereof,
- environment influence is a bunch of real-valued (or complex-valued) functions of time: $\vec{u}(.) : \textit{Time} \longrightarrow \mathbb{R}^m$.

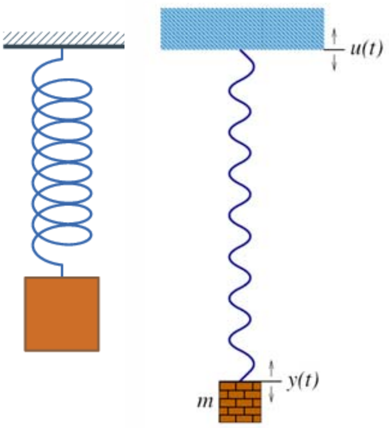
Modeling with differential equations

1. Add further, derived state components: the *derivatives* $\dot{\vec{x}}(.), \ddot{\vec{x}}(.), \ldots$ of the state components.
2. Formulate dynamics as equations between $\dot{\vec{x}}(.), \ddot{\vec{x}}(.), \vec{u}(.), \ldots$

N.B. Higher-order derivatives $x^{(n)}$, $n > 1$, can always be removed by

1. adding a fresh state variable $y(.)$,
2. adding the equation $y(t) = x^{(n)}(t)$,
3. replacing every occurrence of $x^{(n+1)}$ by $\dot{y}$.

Example: spring-mass system



- **Basic model:**

$$\ddot{y}(t) = \frac{F(t)}{m}$$
$$F(t) = k(l(t) - l_0)$$
$$l(t) = u(t) - y(t)$$

- **Replace higher-order derivatives:**

Add $v(t) = \dot{y}(t)$.

Gives $\dot{y}(t) = v(t)$

$$\dot{v}(t) = \frac{k}{m}(u(t) - y(t) - l_0)$$

F: force, m: mass, l: length, l_0: free length

Modeling with functional blocks

- Dynamic system is a network of *basic blocks*:

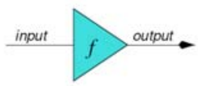
- Blocks are connected via *directed links* that share a state variable

Basic blocks

Basic blocks are *signal transducers* with a ‘simple’ characterization in the time domain, e.g.

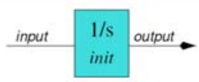
- *‘algebraic’ blocks:* output is a *time-invariant* function of input:

$$out(t) = f\,(in(t))$$



- *state-holding blocks:* integrators & friends, e.g.

$$out(t) = init + \int_0^t in(u)\,du$$



Example

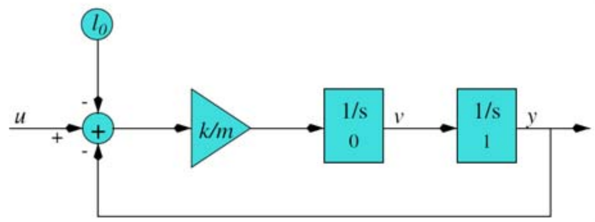
- **DE:**

$$\dot{y}(t) = v(t), \quad y(0) = 1$$
$$\dot{v}(t) = \frac{k}{m}(u(t) - y(t) - l_0), \quad v(0) = 0$$

- **After integration:**

$$y(t) = 1 + \int_0^t v(z)dz$$
$$v(t) = 0 + \int_0^t \frac{k}{m}(u(z) - y(z) - l_0)\,dz$$

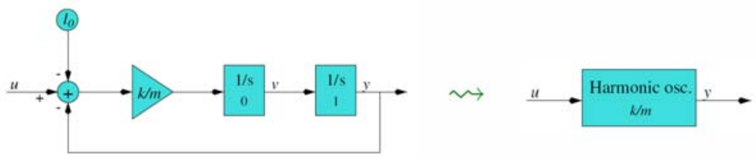
- **Functional block model:**



Compound blocks

- Comprise multiple basic blocks
- Hide the internal state spaces and internal observables

⇒ yields a new, *non-elementary transducer.*



Hybrid Modeling

Rationale: Used to model

1. advanced control techniques (e.g., mode-switching control),
2. embedded system & environment in combination ("Virtual prototyping").

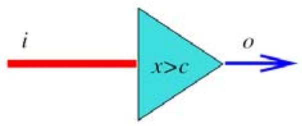
$\Rightarrow$ Need a seamless semantic integration of e.g.

- continuous signal transducers,
- A/D & D/A functional blocks,
- FSMs / Statecharts.

A/D coupling components

have an idealized, *delay-free* semantics:

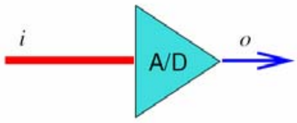
- **Threshold sensor:**



- Analog input i : *Time* $\longrightarrow \mathbb{R}$,
- digital output o : *Time* $\longrightarrow \mathbb{B}$,
- dynamics: $o(t) = (i(t) > c)$.

A/D coupling components

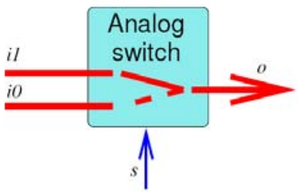
- **A/D converter:**



  - Analog input i : *Time* $\longrightarrow \mathbb{R}$,
  - n-Bit digital output o : *Time* $\longrightarrow \mathbb{B}^n$,
  - dynamics:
    $o(t)$ such that $\left|i(t) - \Sigma_{k=1}^{n} 2^{(k-1)} o_k(t)\right|$ is minimal.

D/A coupling components

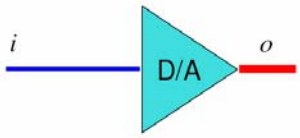
- **Analog switch:**



  - Analog inputs $i_{0,1}$: *Time* $\longrightarrow \mathbb{R}$,
  - digital input s : *Time* $\longrightarrow \mathbb{B}$,
  - analog output o : *Time* $\longrightarrow \mathbb{R}$,
  - dynamics: $o(t) = \begin{cases} i_1(t) & \text{, if } s(t) \\ i_0(t) & \text{, if } \neg s(t) \end{cases}$.

D/A coupling components

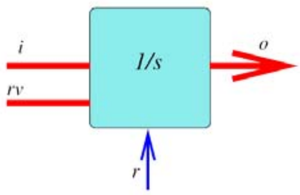
- **D/A converter:**



  - n-Bit digital input i : *Time* $\longrightarrow \mathbb{B}^n$,
  - Analog output o : *Time* $\longrightarrow \mathbb{R}$,
  - dynamics: $o(t) = \Sigma_{k=1}^{n} 2^{(k-1)}\, i_k(t)$.

D/A coupling components

- **Resettable integrator:**

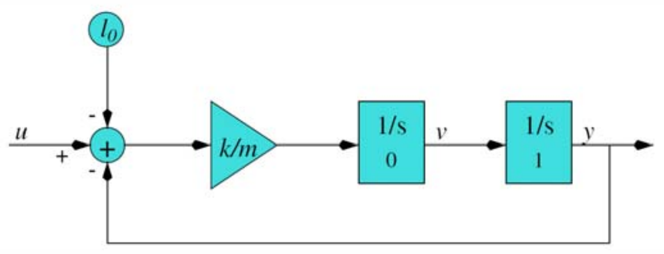


  - Analog inputs/output i, rv, o : *Time* $\longrightarrow \mathbb{R}$,
  - Digital input r : *Time* $\longrightarrow \mathbb{B}$,
  - dynamics: $o(t) = rv(t_r) + \int_{t_r}^{t} i(t)\, dt$, where $t_r = \sup\{t' \leq t \mid r(t')\}$.

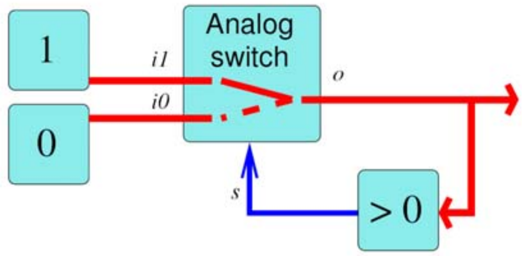
Dynamics of networks

1. The individual blocks impose relations between their input and output waveforms.
2. These relations are adequately covered by the aforementioned characteristic equations of the various basic blocks.
3. Consequently, the dynamics of a network of basic blocks coincides to (solutions of) the conjunction of the characteristic equations of the entailed blocks.

The sane case



The insane case

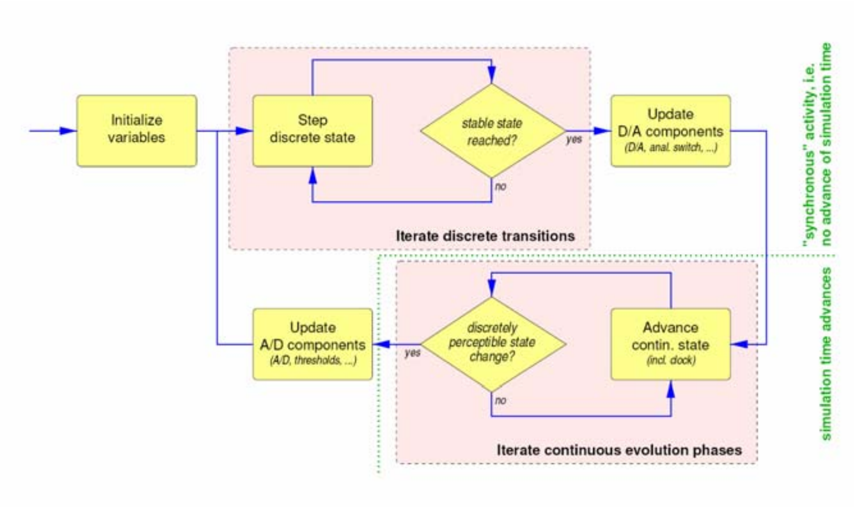


$$o(t) = \begin{cases} 1 & \text{, if } o(t) > 0 \\ 0 & \text{, if } o(t) \leq 0 \end{cases}$$

Avoiding non-causality

1. Simulink (and many other languages) forbids delay-free loops:
   - each loop in the "circuit" has to contain at least one delaying element
     - an integrator
     - a delay block
     - ...
   - if a two-dimensional time model is adopted, even δ-delays suffice!
2. some modeling frameworks interpret delay-free loops as fixed point equations
   - try to solve these equations
   - solution is taken if it is unique

The simulation loop



Iterating discrete transitions

Given the current discrete state and a set of events

1. Evaluate trigger conditions of outgoing transitions
2. Select the enabled transition with highest priority
3. Evaluate its action part
4. Perform action
5. Move control to target state

Procedure is repeated until stable state is reached.

Iterating continuous evolution

Given the current continuous state vector,

1. Numerically compute the output values of all algebraic blocks
2. Find out, whether a discretely observable state change has occurred
3. Use numeric integration to extrapolate state vector to next time instant
   - Most simulation tools offer a selection of integration algorithms
   - Step size may be fixed or adaptive
4. Advance time.

Problems inherited from continuous simulation

- **Numerical instability**
  - it is non-trivial to select an appropriate integration algorithm & basic step size
- **Tractability**
  - Trade-off between precision and computational cost
  - Handling stiff systems
- **Insufficient knowledge of system parameters**
  - knowledge may be too imprecise to allow for a meaningful simulation