# Statemate Course


Kai Baukus

Statemate/SDL

W.-P. de Roever — D. Hogrefe

K. Baukus — H. Neukirchen

CAU Kiel — MU Lübeck




## Introduction to Statecharts

Everything you wanted to know about Statecharts
but were afraid to ask

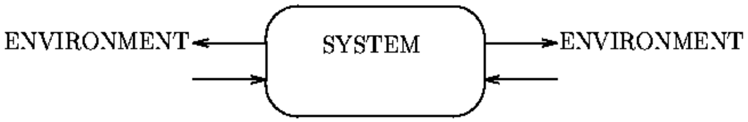

**Abstract:** The notion of a reactive system and the language Statecharts are introduced. The rationale behind the design decisions of Statecharts is explained in relation to the specific nature of reactive systems.


**Literature:** Introduction to Design Choices in the Semantics of Statecharts, C. Huizing, W.-P. de Roever, Information Processing Letters 37, p. 205-213, 1991



## What is a system?

A composed and relative independent whole that can exchange (material, energy, information) with its environment.




The two essential elements in this definition are:

1. group of components, that form a connected whole
2. the exchange with its environment.



## How to Characterize Systems?

The systems are often divided in two categories that make distinction between more simple and more difficult systems.
Examples of this separation are:

- sequential versus parallel,

## How to Characterize Systems?

The systems are often divided in two categories that make distinction between more simple and more difficult systems.
Examples of this separation are:

- sequential versus parallel,
- central versus distributed,



## How to Characterize Systems?

The systems are often divided in two categories that make distinction between more simple and more difficult systems.
Examples of this separation are:

- sequential versus parallel,
- central versus distributed,
- deterministic versus nondeterministic,



## How to Characterize Systems?

The systems are often divided in two categories that make distinction between more simple and more difficult systems.
Examples of this separation are:

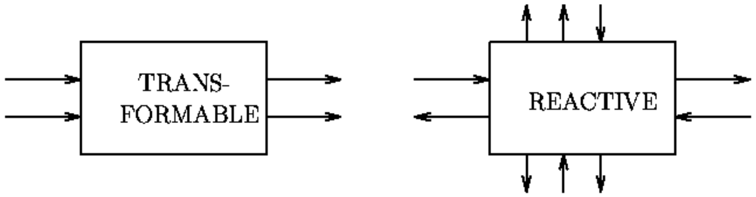
- sequential versus parallel,
- central versus distributed,
- deterministic versus nondeterministic,
- terminating versus nonterminating.



## Another Separation

There is another separation, namely:

- **transformational** and
- **reactive systems**.



## Transformational systems

are described by the relation between initial and corresponding final states; they have a linear structure, because only the initial and corresponding final states are of interest.

**Examples include:** sorting algorithms, compilers, and other algorithms computing a function as discussed in your data structures and complexity of algorithms course.



## Reactive systems

do not compute a function, but are in continuous **interaction** with their environment.

**Examples:** your tv set, digital watches, chips, interactive software systems, game programs s.a. trackman, monkey island, tomb raider, and other interactive computer games, but also one's heart monitor at an intensive care unit.



## Why use Formal Methods?

- in the formalization process ambiguities, omissions and contradictions can be found in the informal formulation of the problem,



## Why use Formal Methods?

- in the formalization process ambiguities, omissions and contradictions can be found in the informal formulation of the problem,
- the formal model can be proven correct with mathematical methods,

## Why use Formal Methods?

- in the formalization process ambiguities, omissions and contradictions can be found in the informal formulation of the problem,
- the formal model can be proven correct with mathematical methods,
- a formal specified system can be analyzed to have or to have not some wanted properties,



## Why use Formal Methods?

- in the formalization process ambiguities, omissions and contradictions can be found in the informal formulation of the problem,
- the formal model can be proven correct with mathematical methods,
- a formal specified system can be analyzed to have or to have not some wanted properties,
- a formal verified subsystem can be embedded in a greater system with more confidence,



## Why use Formal Methods?

- in the formalization process ambiguities, omissions and contradictions can be found in the informal formulation of the problem,
- the formal model can be proven correct with mathematical methods,
- a formal specified system can be analyzed to have or to have not some wanted properties,
- a formal verified subsystem can be embedded in a greater system with more confidence,
- the formal model leads to (partly) automated development methods and tools like simulators,



## Why use Formal Methods?

- in the formalization process ambiguities, omissions and contradictions can be found in the informal formulation of the problem,
- the formal model can be proven correct with mathematical methods,
- a formal specified system can be analyzed to have or to have not some wanted properties,
- a formal verified subsystem can be embedded in a greater system with more confidence,
- the formal model leads to (partly) automated development methods and tools like simulators,
- several designs can be compared with each other.

## Transformational/Reactive Systems

- Transformational systems are well-studied; for their programming and analysis many good languages and theories exist.
- We explain why the language Statecharts is a good candidate for **specifying** and **programming** reactive systems.

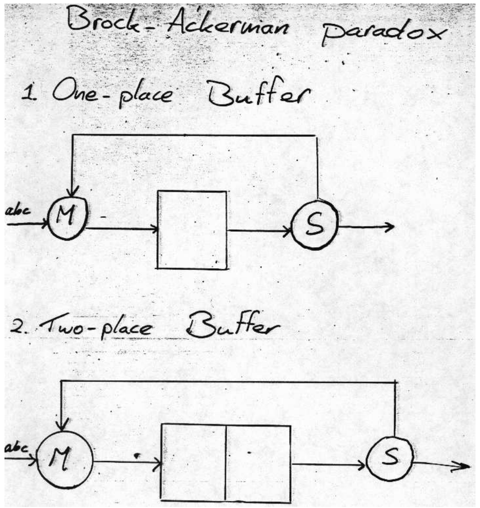


## Why not use transformational techniques?

- If transformational systems are so well studied, why doesn't one consider a reactive system as a transformational one?
  Simply say that a reactive system transforms a **sequence** of inputs to a sequence of outputs.
- This doesn't work, because of "feedback", as illustrated by the **Brock-Ackermann** paradox.
- Consider two systems, a one-place buffer and a two-place buffer. If you consider these transformationally, they display the same initial-final state behavior.



## Brock-Ackermann paradox




**in**: $abc\ldots$
**out**: 1. e.g. $abac\ldots$
2. e.g. $abca\ldots$
not $abac\ldots$



## Conclusion

If the output of these systems is fed back, and merged with their input they behave differently.

The relative order of output events relative to the input events needs to be specified, in order to characterize the semantics of a system with interaction with its environment through feedback.

(One needs to know when an output is produced)

## Initial/Final State Behavior

Transformational systems have a linear structure, and so have the conventional languages for specifying and programming them. What one describes is how a final state is produced from an initial one. **The relative time when intermediate states are computed is not important**, and neither is their identity as long as the corresponding final state is known!



## Reactive Behavior

- For reactive systems this is completely different:
  - The "moment" a new input arrives is relevant to the behavior of the system $\Longrightarrow$
  - The internal state of the system at the time of input is important for the systems reaction.
  - Reactive systems may not even have a final state!
- So, in reactive systems **there is no main sequential flow of control** (as in transformational systems) and **statements can have several entry and exit points.**



***Finite State Machines***



## Finite Automata

Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

## Finite Automata

Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

| | |
|---|---|
| $Q$ | is a finite set of states, |



## Finite Automata

Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

| | |
|---|---|
| $Q$ | is a finite set of states, |
| $\Sigma$ | is the input alphabet, |



## Finite Automata

Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

| | |
|---|---|
| $Q$ | is a finite set of states, |
| $\Sigma$ | is the input alphabet, |
| $q_0 \in Q$ | is the begin state, |



## Finite Automata

Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

| | |
|---|---|
| $Q$ | is a finite set of states, |
| $\Sigma$ | is the input alphabet, |
| $q_0 \in Q$ | is the begin state, |
| $F \subseteq Q$ | is the set of final states, |

## Finite Automata

Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

| | |
|---|---|
| $Q$ | is a finite set of states, |
| $\Sigma$ | is the input alphabet, |
| $q_0 \in Q$ | is the begin state, |
| $F \subseteq Q$ | is the set of final states, |
| $\delta : Q \times \Sigma \rightarrow Q$ | is the transition function. |



## Finite Automata

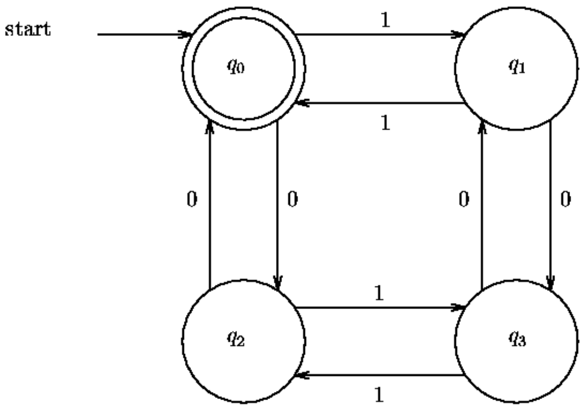
Formally a *finite automaton* is defined as a five tuple $(Q, \Sigma, \delta, q_0, F)$ where

| | |
|---|---|
| $Q$ | is a finite set of states, |
| $\Sigma$ | is the input alphabet, |
| $q_0 \in Q$ | is the begin state, |
| $F \subseteq Q$ | is the set of final states, |
| $\delta : Q \times \Sigma \rightarrow Q$ | is the transition function. |

The transition function gives for every state $q$ and every input symbol $a$ the new state $\delta(q, a)$ that arises as reaction on the execution of $a$ in state $q$.



## State Diagram




For each state, the possible reactions to input that arrives in that state is specified by a transition to other states.



## Extensions

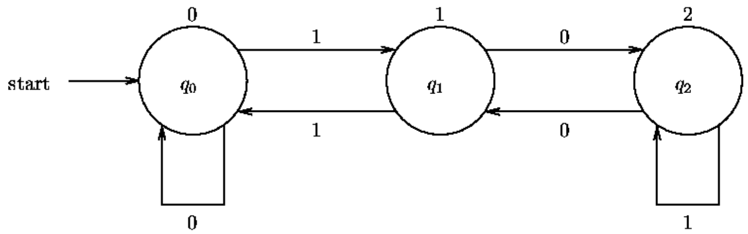
A restriction of these kind of automata as defined above is the fact that they have an input alphabet but not an output alphabet. There are two ways to extend the above model with output: output can be associated with a state (a so called Moore machine) or with a transition (a so called Mealy machine). These are formally defined as follows.

- A $Moore\ machine$ is a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where $Q, \Sigma, \delta, q_0$ are the same as in the definition of the finite automaton,

  | | |
  |---|---|
  | $\Delta$ | is the $output\ alphabet$ and |
  | $\delta : Q \rightarrow \Delta$ | is the $output\ function$. |

- A $Mealy\ machine$ is also a 6-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$ but now $\lambda$ is a function from $Q \times \Sigma$ to $\Delta$.

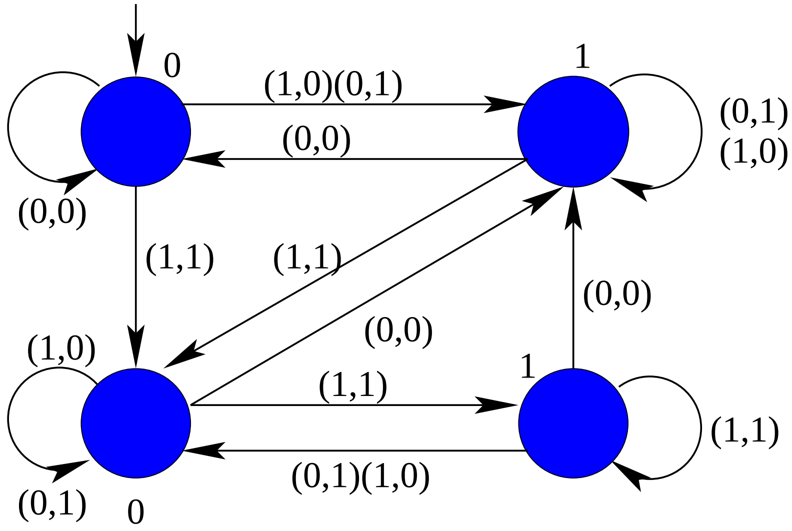
## Example: Moore Machine




For a Moore machine $\lambda$ the output is thus associated with every state, while for a Mealy machine $\lambda(q, a)$ gives the output associated with the transition of state $q$ on input $a$.

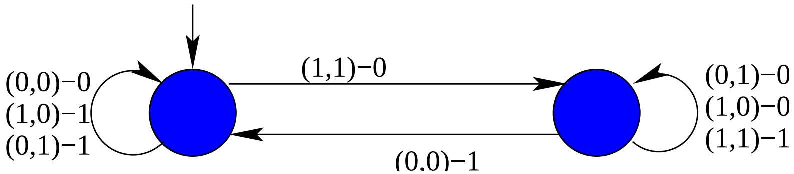


## Serial Addition: Moore Machine






## Serial Addition: Mealy Machine



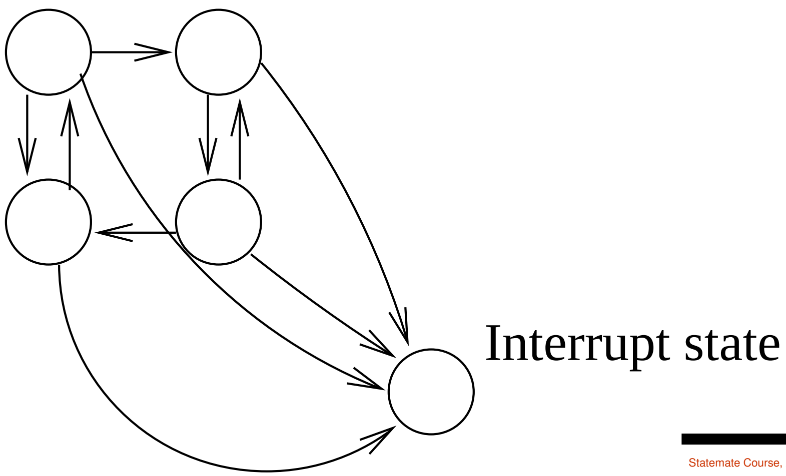



## Disadvantages

- **They have no structure**. There is no strategy for their top-down or bottom-up development.
- state-transition diagrams are “flat”, i.e., without a natural notion of depth, hierarchy or modularity,
- state-transition diagrams are uneconomical concerning their transitions:think for instance of a high-level interrupt

## Interrupt Transition

They are not economical w.r.t. transitions, when one event has all transitions as a starting point as in case of interrupts:






## Disadvantages (cont'd)

- concerning the states state-transition diagrams are even very uneconomical: exponential blow-up
- They are not economical w.r.t. **parallel composition**: **Exponential growth** in the number of states when composed in parallel.
- the nature of state-transition diagrams is inherently sequential and so parallelism can't be represented in a natural way.



## Statecharts

We need a formalism for the hierarchical development and refinement of Mealy machines.
This is provided by Statecharts, invented by David Harel.

Statecharts display **hierarchy** and **structure**, and enable hierarchical development.



## *Statecharts*

## *Hierarchy and Structure*

The concepts of hierarchy and structure in Statecharts are introduced using a quite familiar example of a reactive system: that of a **television set with remote control**.



## *First concept: Hierarchy*

Hierarchy or depth in states, and interrupts. This is achieved by drawing states as boxes that contain other boxes as sub-states.

- The television set can be in two states: **on** and **standby**. Switching between them is done by pushing the **on** and **off** buttons, generating the **on** and **off** events:

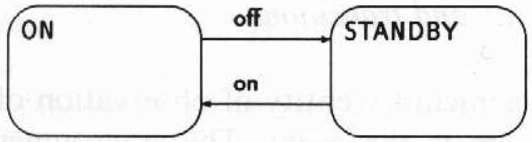



Fig. 2.



## *Zooming into* ON

In state **on** the tv set can be in two sub-states: **normal** and **videotext**:

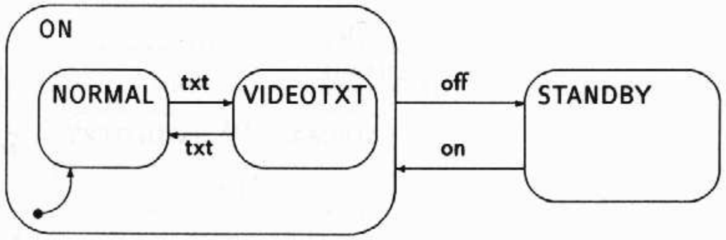



Fig. 3.

The $\longrightarrow$ arrow leading to **normal** specifies which sub-state should be entered when the higher level state**on** is entered, namely **normal**.



## *Superstates*

When in **on** an event **off** is generated, this acts like an interrupt and state **on** (incl. all its sub-states) is left, and control switches to state **standby**. In this way interrupts are handled without cluttering the picture with arrows.

## Second concept: Orthogonality

- Two independent components can be put together into an **AND-state**, separated by a dotted line

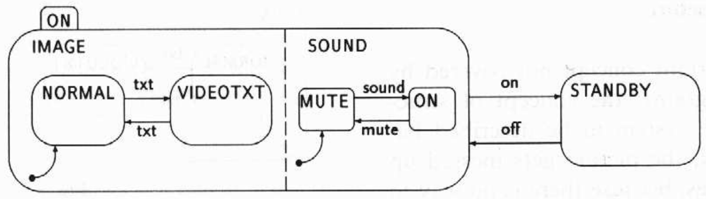



Fig. 4.

- Being in an AND-state means being in all of its immediate sub-states **at the same time**. This prevents the exponential blow-up familiar from composing FSMs in parallel.



## Third concept: Broadcast

In our case we split state **normal** in two orthogonal components **channel**, for selecting channels, and **sm** for switching to mute:

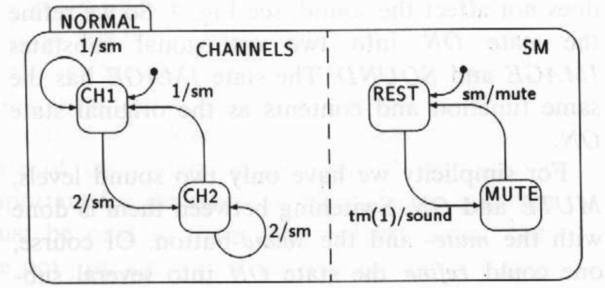



Fig. 5.



## Behavior

- When a channel button (1 or 2, for simplicity) is pressed, one switches to that channel and the internal event **sm** is generated. This causes the event **mute** by a transition in **sm** to state **mute**, and the sound will be turned off.
- After **one second** the event **sound** is generated to turn the sound on, again. This is done by the special **time-out** event **tm(1)**.



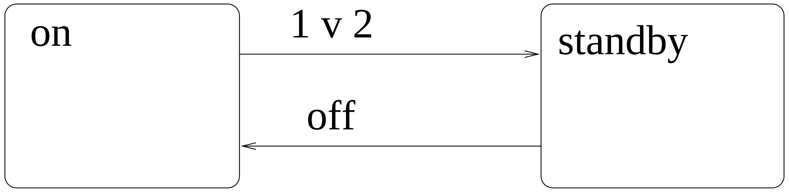
## Actions and Transitions

Thus one sees that orthogonal components can communicate by generating events which are broadcast, and that this can be done in a time-dependent manner: introducing the **generation of events** $e/a_1; \ldots ; a_n$ and **time-out events tm(1), ...**

In general the label of a transition consists of two parts: a **trigger** that determines if and when a transition will be taken, and an **action** that is performed when a transition is taken. This action is the generation of a **set of events**.

## Fourth concept: Compound events

When in state **standby**, dependent on whether one presses button 1 or 2 one makes sure to switch to states **ch1** or **ch2** in **on**. This is indicated as follows:






## Transition Labels

- In general one can label transitions by **compound events** s.a. $(\neg a \wedge b) \vee c, a \wedge b, c \vee d, \neg a$, etc

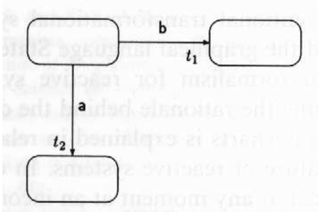



Fig. 7.

- E.g., in:
  $a$ can be replaced by $a \wedge \neg b$ to express priority of event $b$ over event $a$.



## Summa Summarum

In a nutshell, one may say with David Harel:

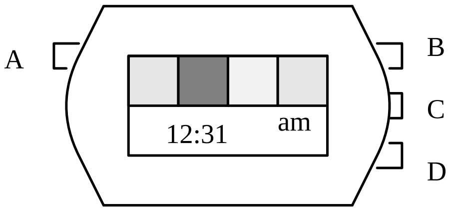
$$\begin{aligned} \text{Statecharts} &= \text{Mealy Machines} \\ &+ \text{depth} \\ &+ \text{orthogonality} \\ &+ \text{broadcast} \\ &+ \text{data} \end{aligned}$$



***Another Example***

## Alarm watch

As an example of a statechart we use that of a simple digital watch with four buttons $A, B, C$ and $D$ like in the below picture:






## Events

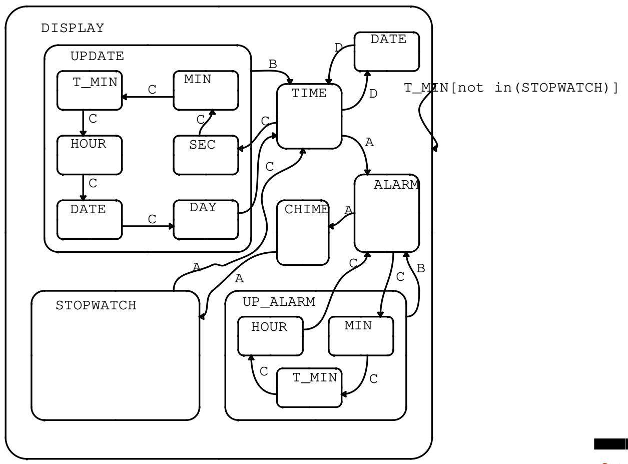
The following events are considered as external:

- $A, B, C$ and $D$ describe the pushing of the four buttons and $B\_up$ the release of button $B$.
- The events $Bt\_In, Bt\_Rm, Bt\_Dy$ and $Bt\_Wk$ describe respectively the putting in, removal, drop dead and weakening of the battery.
- $T\_hits\_Hr$ describes that the internal time has reached a whole hour and $T\_Hits\_Tm$ describes that the internal time has reached the alarm time.
- $T\_Min$ describes that there are two minutes passed since for the last time a button has been pushed.



## Display control

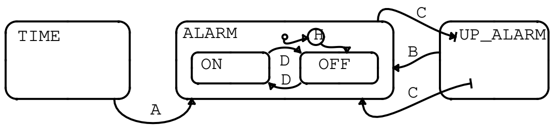
There is a special state $up - alarm$ for the changing of the internal state of the alarm. Note that $T_min$ takes care of the resetting of a state.



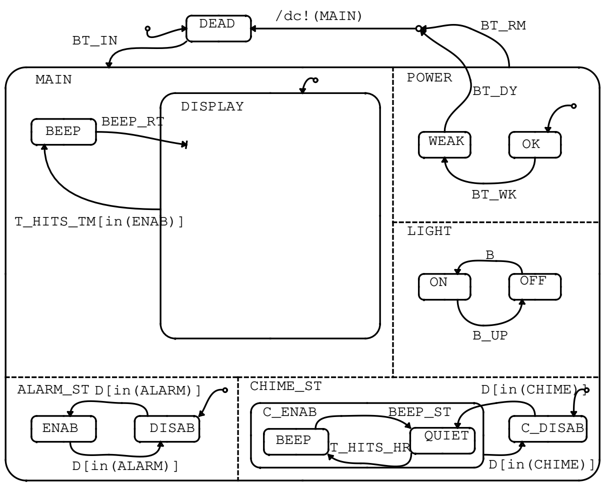



## History of states

A frequently used way to enter a group of states is by the history of that group. The most simple example of this is the one where you enter the most recently visited state of a group. In the watch example this happens in the zoom-in of the $alarm$ state with two substates $on$ and $off$. The problem is that the initial default is the $off$ state but when we put on the alarm we want to get back the next time in state $on$. In the next statechart this described by the **H** connector.

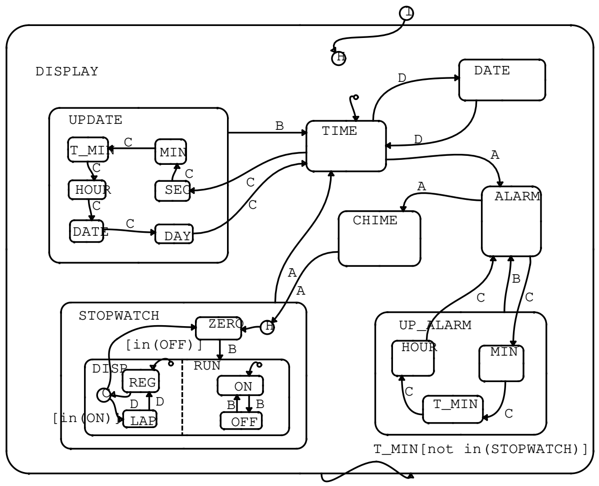


# The Watch






# Display with Stopwatch






# Events

The basic events and condition are external, for example for the watch the pressing of a button is an external event and $T\_Hits\_Tm$ is an external condition. Besides the external events the following internal events are allowed:

| | | |
|---|---|---|
| $entered(S)$, | abbreviation | $en(S)$, |
| $exit(S)$, | abbreviation | $ex(S)$, |
| $timeout(E, X)$, | abbreviation | $tm(E, X)$, |
| $true(C)$, | abbreviation | $tr(C)$, |
| $false(C)$, | abbreviation | $fs(C)$. |



# Action

An action can be an uninterpreted event symbol, called primitive event, and causes then other transitions in the statechart. Furthermore actions can turn on or off uninterpreted condition symbols. The following primitive actions are allowed:

| | | |
|---|---|---|
| $make\_true(C)$, | abbreviation | $tr!(C)$, |
| $make\_false(C)$, | abbreviation | $fs!(C)$, |
| $history\_clear(S)$, | abbreviation | $hc!(S)$, |
| $deep\_clear(S)$, | abbreviation | $dc!(S)$. |