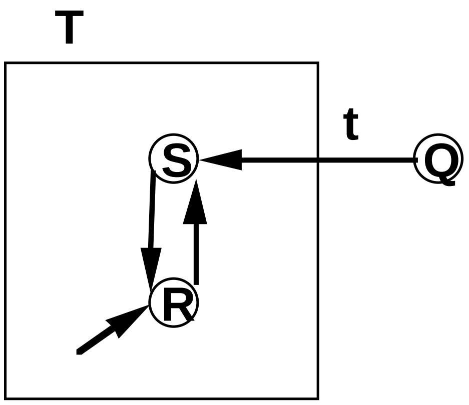
Hierarchical state terminology
The substates of a hierarchical state are said to be **child** states, and the hierarchical state is known as the **parent** state. Since a child state may again be a hierarchical state, this relationship may be extended over several generations. Thus we also speak of **ancestor** and **descendent** states. States are permitted to have the same names if they have different parent states — “fully qualified pathnames” formed by attaching ancestor names separated by '.' may be used for disambiguation

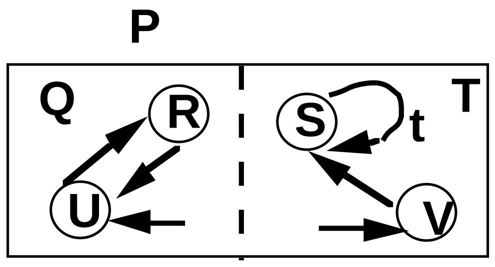
Further rules for events
- If a system is in state S, then not only is in(S) true, but in(T) is true for each ancestor T of S.
- Entering a state S will trigger event en(S), as well as en(T) for every ancestor of S in which the system did not reside when S was entered.
- Exiting a state S will trigger the event ex(S), as well as ex(T) for each ancestor T of S in which the system does not reside after the transition.

For example, in the statechart



after transition t
en(S) = true
en(T) = true
en(R) = false

Another example:



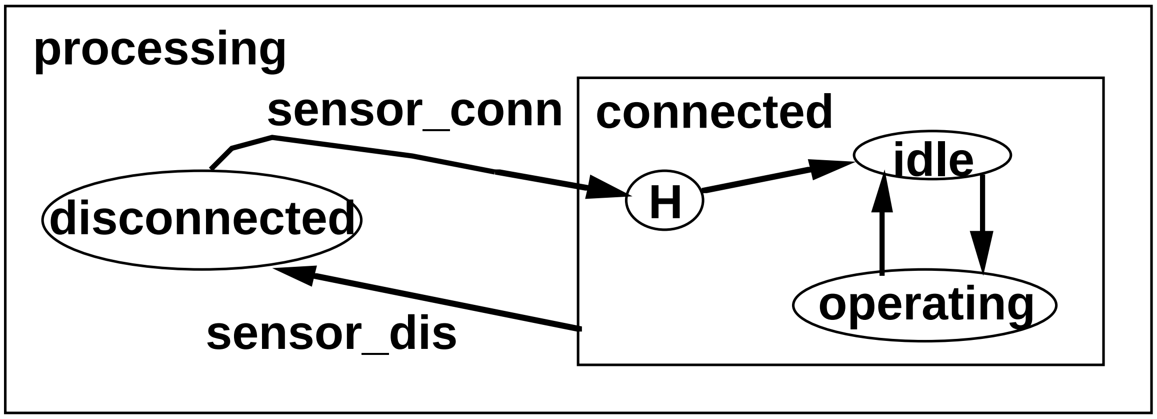
if the configuration is (R,S), after transition t
en(S) = true
en(T) = false
en(P) = false

History entrances

With the “sequential machine” model, nothing is “remembered” about the sequence of transitions that led to a state — in fact, the point is that the state represents the model’s memory of past events, its entire memory! This is no longer the case with statecharts, and there are several components of memory of past activity. For one thing, statecharts have variables that may be assigned values and subsequently tested. These variables provide a means to record selected information for later use. Also, both events and activities can be schedules for the future. Therefore, some component of the system must have an implicit memory to retain knowledge of these actions until they are to occur.

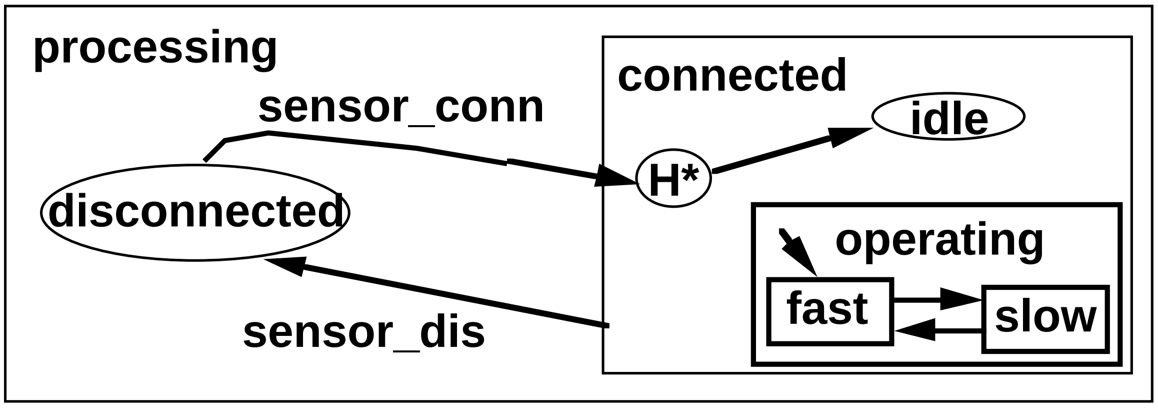
History entrances provide a means to extend this to past state behavior. A history entrance is another pseudo-state — rather than convey a configuration of the system, the history connector has a semantic meaning. A transition to a history connector indicates that the next state in the group should be the last one visited (i.e., resume semantics). The history connector also has a regular outgoing transition showing the state to be entered if there is no history (e.g., entry for the first time).

History example 1



The history connector specified in this figure indicates an entrance by history on the first level only — whenever the 'sensor_conn' event occurs, the CONNECTED component resumes in the state it was last in. However, if state OPERATING had substates (e.g., SLOW and FAST), its last substate would not be remembered, and the resume would use whatever is the initial state. If this were desired, there is a **deep history connector** that is denoted by H*.

History example 2



In this example with the deep history connector, if the system was last in OPERATING.FAST, then that would be the state entered despite the fact that SLOW is the initial state.

With the addition of history connectors, it becomes important to include the capability to “reset” histories. For instance, if we were to add a HALT event for the processing state, it may be natural for the next entrance to OPERATING to be to the initial state regardless of past behavior. For this purpose, two actions are provided, history_clear(S), abbreviated hc!(S), and deep_clear(S), abbreviated dc!(S). The action hc!(S) deletes the history information for state S, but not its descendents. That is, the next time a history (or deep history) connector *in state S* is entered, the behavior is as if it were the first time. The action dc!(S) deletes the history information for S *and* all its descendent states.

Actions
The presence of actions in transitions enhances their passive character, and provides a general processing capability. Basic actions are available to manipulate each of the three types of elements: events, conditions, and variables.

For events, a primitive action is gen(EVENT), mentioning an event name — this causes the named EVENT to be sensed in the next step.

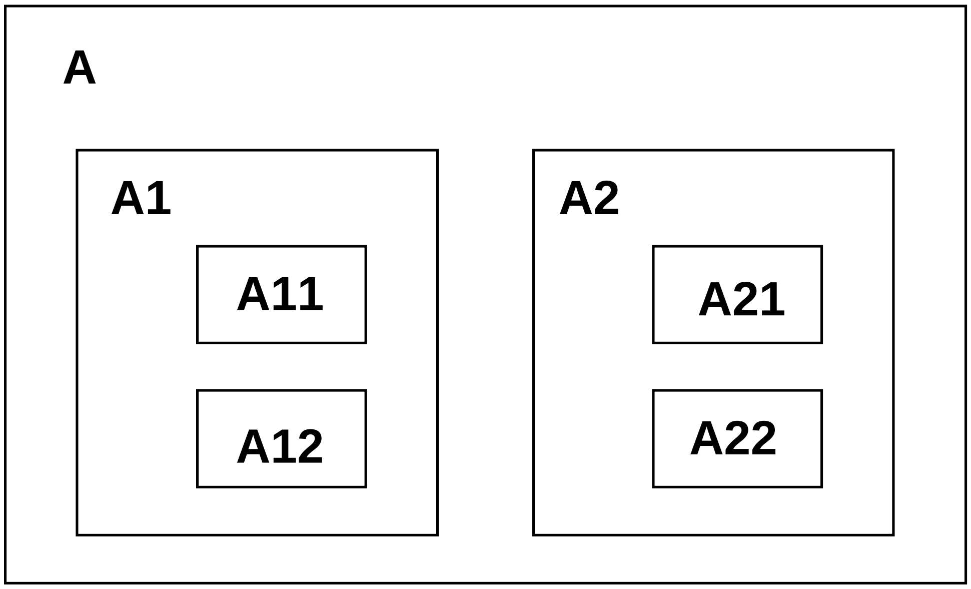
The most primitive form of a condition is a Boolean variable. Such a condition C can be manipulated by assignment (as can any other variable), C:= false. Associated with such a variable are two events, true(C) and false(C) that occur precisely when C *changes*, respectively, to true or false. Note that these events only occur when there is a change — if C is already false, then C:= false does not cause the event false(C)!

General data variables can be assigned in the same way as condition variables. All variable values are persistent — they are available until they are changed. Whenever an assignment to variable X takes place, the event written(X), abbreviated wr(X), occurs. There is also a similar event, changed(X) that occurs when X is *changed*.

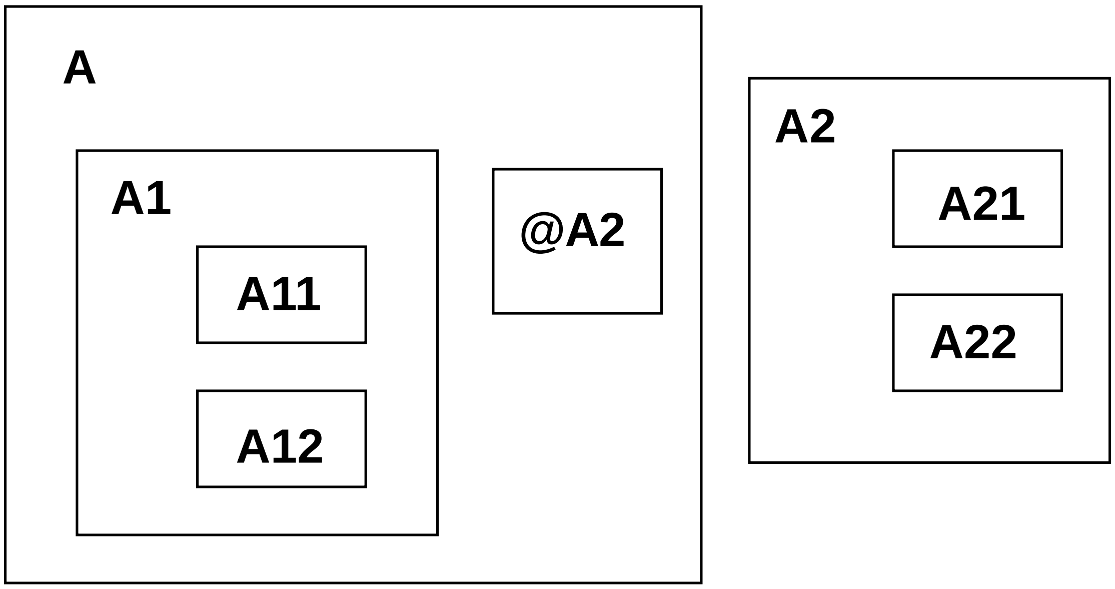
Compound actions
Any sequence of actions can serve as an action. In addition, conditional actions (i.e., if-then-else), iterative actions (while loops), procedures, etc. can be utilized. The Rhapsody system also provides a “macro” tool where a code fragment is placed in their *Data Dictionary*. The precise notation is left unstated here — the Rhapsody tool has versions supporting three languages: C/C++/Java.

When a sequence of actions involves multiple assignments, the timing of access can be troublesome. Recall that the values of variables used in an action are those at the beginning of a step. Hence in a sequence such as X:= 1; Y:= X; the value of X accessed in the second assignment. Since this is sometimes inconvenient, the Rhapsody system provides a Rhapsody allows. *context variables* denoted by a prefixed '$'. Context variables have scope local to an action, and values are those last assigned during the action.

Splitting up charts
The statechart methodology is inherently hierarchical and this greatly facilitates the development of models. However, the graphical nature of statecharts means that even moderately complex descriptions may not neatly fit a page. In fact, just a few levels of nesting may lead to this difficulty. So statecharts use a referential mechanism to allow flexible pagination. For example,

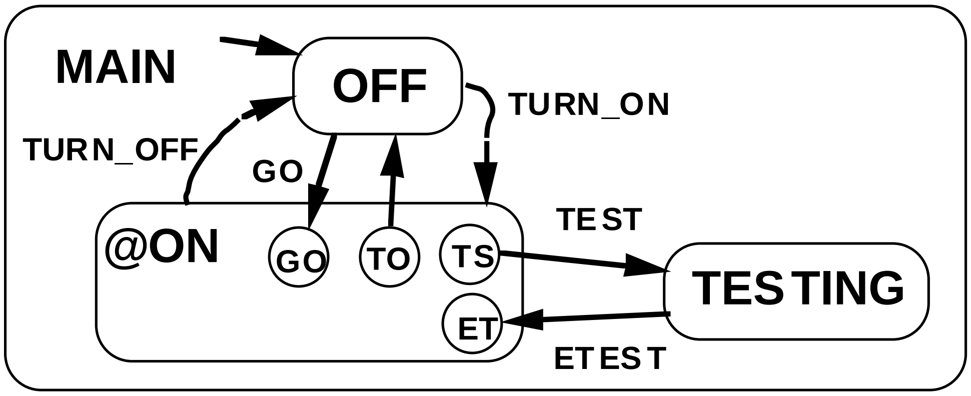


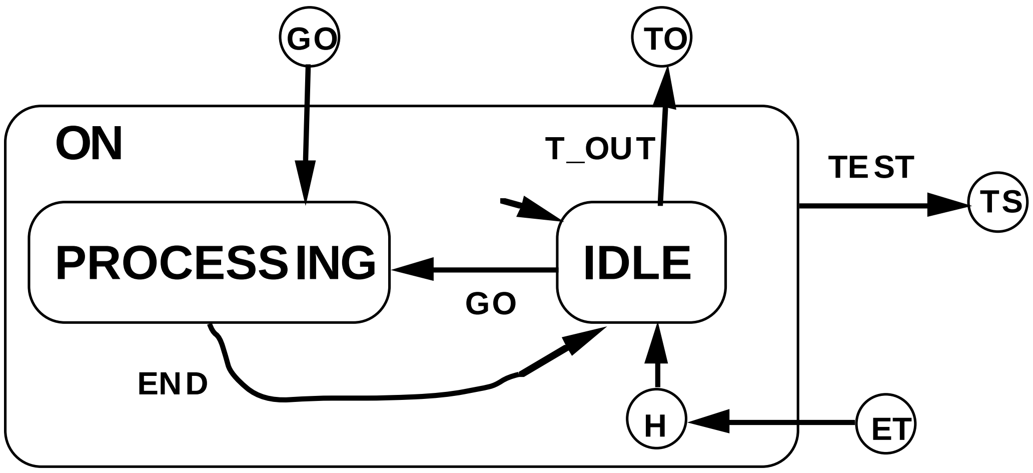
may be depicted as



The reference notation @A designates a physical rearrangement, not a logical rearrangement. That is, the parent/child relationships are not altered by this relocation of the subchart to another physical page.

This relocation of a subchart to another physical page makes the representation of transition arrows more inconvenient. To facilitate this, another form of “connector” is provided — the **off-page connector**. We have already seen the use of condition and history connectors. In a similar spirit, off-page connectors are pseudo-states and are named for identification purposes. They must appear in both the referring box and in the defining subchart diagram with conforming transitions. This is illustrated below.


MAIN
OFF
TURN_ON
TURN_OFF
GO
@ON
GO
TO
TS
ET
TEST
TESTING
ETEST


GO
TO
ON
T_OUT
TEST
TS
PROCESSING
IDLE
GO
END
H
ET