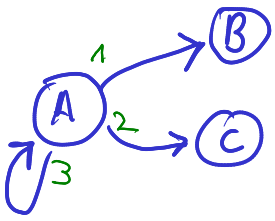
Overview of YAKINDU's (default, event-driven) semantics

Some background first:

To achieve determinism, a priority ordering exists between all transitions that can possibly be enabled simultaneously...

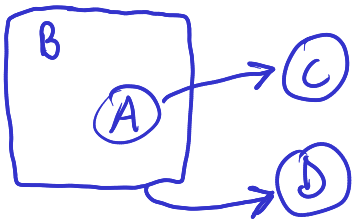
-> What transitions can be enabled simultaneously?

(1) Transitions that have the same source state



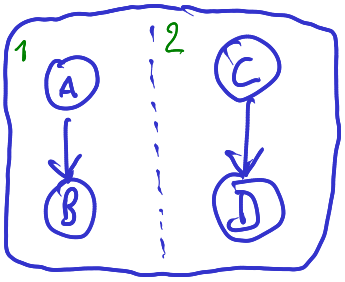
Solution: explicit priorities (numbers) on outgoing transitions

(2) Transitions that have source states that are an ancestor of one another



Solution: give higher priority to 'parent' states (in this case, B)

(3) Transitions that have orthogonal source states



Solution: explicit priorities (numbers) between orthogonal regions

Now, what does the execution of a statechart look like?

First, the statechart is initialized (default states are entered, and default values are assigned to variables).

The statechart has a FIFO queue for input events.
As long as the queue is not empty, the statechart handles 1 input event at a time, by executing a run-to-completion (RTC) step.

An RTC step consists of 2 phases:
- First, the input event is handled
- Next, internal events are handled

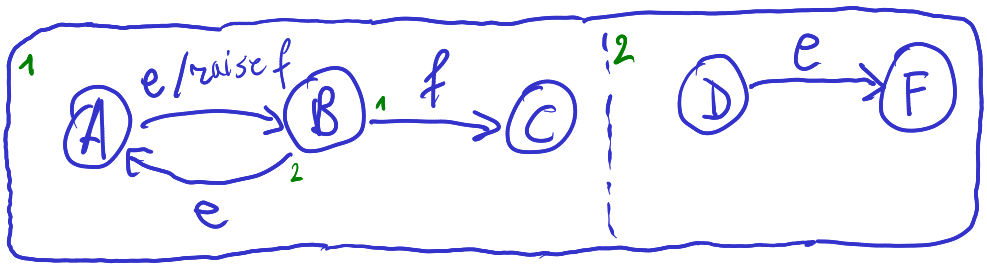
Let us first look at how the input event is handled:
- The input event is considered 'active'. Possibly this causes some transitions to be enabled (i.e. transitions that have the active event as their trigger). Among the enabled transitions, the transition with the highest priority fires. If the firing of the transition raises an internal event, then this event is added to a separate internal event (FIFO) queue, to be dealt with later.
- This is repeated for as long as possible, but with the restriction that only transitions that are orthogonal to the transitions that have already fired, are allowed to fire.

Next, internal events are handled. One internal event is handled at a time.
More precisely, while the internal event queue is not empty, pop an event, and handle it:
- The input event is no longer active. Instead, only the current internal event is active. Similar to how we handled the input event, the highest-priority enabled transition can fire, possibly raising more internal events (also appended to the internal event queue).
- This is repeated, again with the restriction that only transitions can fire that are orthogonal to the transitions already fired while handling the current internal event.

Finally, when the internal event queue is empty, the RTC step ends.

But what about output events? In principle, the set of all raised output events becomes visible only at the end of the RTC step. However, in YAKINDU's generated code, an output event is sensed by the environment immediately after it is raised (so: possibly in the middle of an RTC step).

Example:



input events = {e}

internal events = {f}

input queue: [e]

Suppose we are in state A and D, and our input event queue contains a single event 'e'.
We handle the event 'e' (popping it from the queue) by executing the following RTC step:

1. handle 'e':
   With the event 'e' active, the enabled transitions are: A -> B and D -> F.
   A -> B has the highest priority, and is fired. Event 'f' is added to the internal event queue.
   Now we are in states B and D.
   At this point, the only enabled transition is D -> F. The transition B -> A is not enabled,
   because B -> A is not orthogonal to A -> B, which already fired while we were handling input event 'e'.
   So, D -> F fires and we end up in states B and F.
   Now there are no more enabled transitions.
2. handle internal events:
   pop event 'f':
   Enabled transitions are: B -> C
   B -> C is fired
   Now there are no more enabled transitions.
   The internal event queue is now empty, so we conclude the RTC step.

internal event queue: [f]

What about timed transitions?

A timed transition simply causes a timer to be started behind the screens,
when its source state is entered. When the source state is exited, the timer is canceled.
When the timer elapses, a special input event is generated, indicating that the timed transition can take place.