The STATEMATE Semantics of Statecharts

Paper by **David Harel** and **Amnon Naamad**

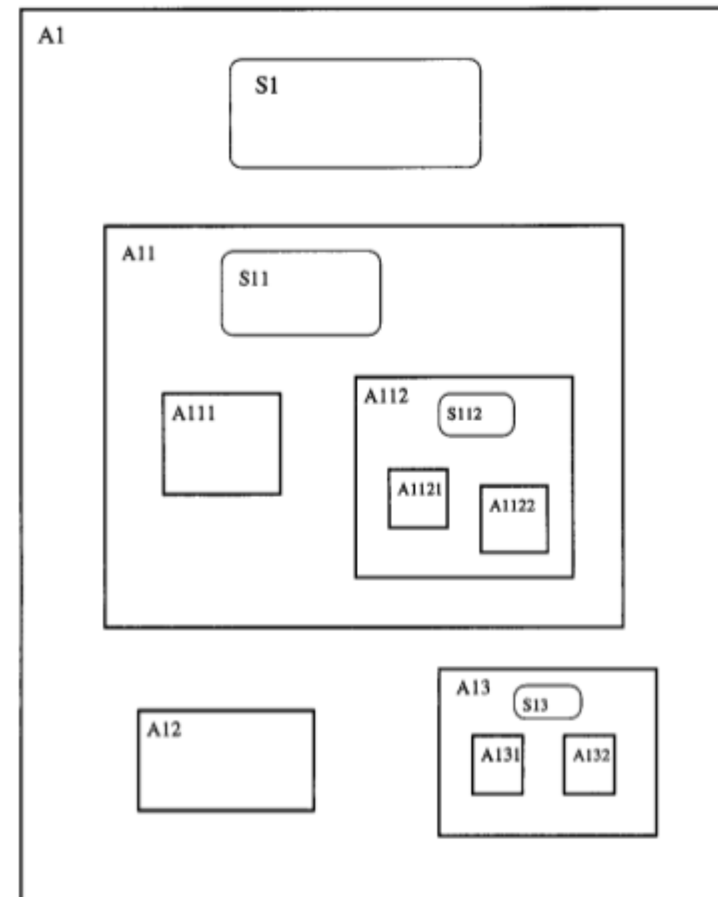
Part 2 presented by **Jon Kotker**

Remember

Taken from http://www.deehathaway.com/whos-awesome

Recap: STATEMATE

- Set of languages used to model reactive systems
- **Statecharts** provide semantics to **activities** in **activity charts**



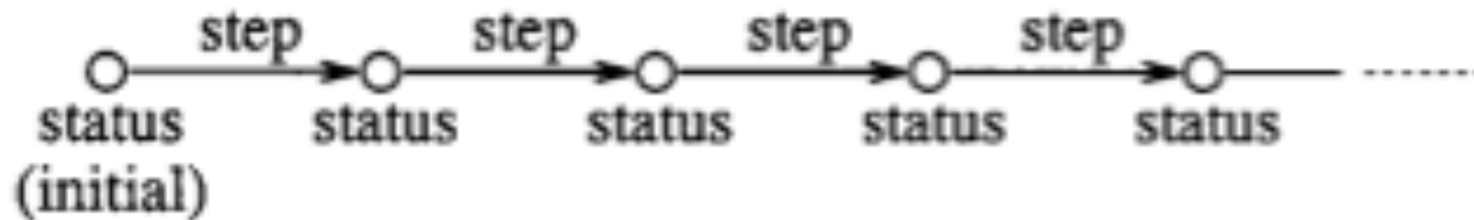
Recap: Terminology

- **Root**: State with no parent state.
- **AND-state**: State with orthogonal components that must *all* be executed.
- **OR-state**: State with orthogonal components of which exactly one must be executed ("exclusive-or").
- **Event**: Triggers a transition to another state.
- **Condition**: Guards a transition from occurring.
- **Action**: Carried out when a transition is taken.

Recap: Terminology

- **Static Reaction (SR)**: Action carried out as long as the system is in the corresponding state.
- **Run**: Responses of system to a sequence of external stimuli from the environment.
- **Status**: Snapshot of the system's situation; a run is a sequence of statuses.
- **Step**: Moving from one status to another.
- **Compound Transition (CT)**: Maximal chain of transition segments, linked by connectors, that are executable simultaneously as a single transition.
- **Conflicting Transitions**: Two transitions are in *conflict* if there is some common state that would be exited if any one of them were to be taken.

How Does a Step "Happen"?



The Basic Step Algorithm

- **Inputs**
  - The *status* of the system
    - List of states in which the system currently resides
    - List of activities currently active
    - Current values of conditions and data-items
    - List of events generated in the previous step
    - List of scheduled actions and their time for execution
    - List of timeout events and their time for occurrence
    - Relevant information on state history
  - The *current time*
  - A list of external changes *from the environment* since the last step.
    - Events that have occurred
    - Changes in the values of conditions and data-items

The Basic Step Algorithm

- Notation
  - `(a, next-a)` is a *scheduled action*: a is an action that is scheduled to happen at time `next-a`
  - `(E, next-E)` is a *timeout event*: E is an event that is scheduled to happen at time `next-E`
  - `E = tm(e, d)`, where e is the event generated, and d is the delay after which `next-E` should be generated
- Output
  - A *new* system status

The Basic Step Algorithm

+ **Stage 1**: *Step Preparation*

1. Add the external events to the list of internally generated events.
2. Execute all the actions implied by the external changes.
3. For each pair `(a, next-a)` in the list of scheduled actions:

```
if next-a <= current-time:
   carry out a and remove (a, next-a) from the list
```

4. For each pair `(E, next-E)`, `E = tm(e, d)`:

```
if e is generated:
   next-E := current-time + d
   else if next-E <= current-time:
      generate E and set next-E := infinity
```

The Basic Step Algorithm

- **Stage 2**: *Compute the contents of the step*

1. Compute the set of enabled CTs.
2. Remove all the CTs that are in conflict with an enabled CT of higher priority.
3. Split the set of enabled CTs into maximal *nonconflicting sets*. (No two CTs in any set are in conflict.)
4. For each set of CTs, compute the set of enabled SRs defined in states that are currently active and are not being exited by any CT in the set.
5. If there are no enabled CTs or SRs

   the step is empty

   else if step 3 produced a single set

      this set constitutes the step

      else pick any one set

The Basic Step Algorithm

- **Stage 3**: *Execute the CTs and SRs*.

  (Define EN to be the set of enabled CTs and SRs from the previous step.)

- For each SR X in EN, execute the action associated with X.

- For each CT X in EN, let Sx and Sn be the sets of states exited and entered by X, respectively.
  - Update the history of all the parents of states in Sx.
  - Delete the states in Sx from the list of current states.
  - Execute actions associated with exiting states in Sx.
  - Execute actions of X.
  - Execute actions associated with entering states in Sn.
  - Add the states in Sn to the list of current states.

The Basic Step Algorithm

- Implementing the *semantics* of a step
  1. Create a list of pairs.
     - Each pair is of the form `<element, new-value>`.
     - `element` will be assigned `new-value` at the end of the step.
     - This guarantees that old values of elements are used.
  2. Assign the elements the new values.
     - When an element is assigned a new value more than once, the last assignment is used – *write-write racing*.

Two Models of Time

- Questions
  - How does real-time relate to steps?
  - When is the internal clock advanced relative to the execution of steps?
  - How long do steps take in terms of the clock?

- Two models of time
  - **Synchronous**: System executes a single step every time unit, reacting to external changes since the last time-unit.
  - **Asynchronous**: System reacts whenever an external change occurs. This allows several external changes to happen simultaneously, and thus several steps to take place within a single time-unit (a *superstep*).

Two Models of Time

- In both models, the execution of a step seems to take *zero time*.
  - No external changes have any effect during execution.
  - As if time *stops* for the duration of execution.
- STATEMATE supports *both* models.

Synchronous Model

- Used for highly synchronous systems.
- Assume the previous step was executed at *t*. We can then issue a GO command during a simulation, which works as:
  - Execute all external changes since completion of last step.
  - Increment clock by one time-unit.
  - Execute all timeout events and scheduled actions that are due.
  - Execute one step.

Asynchronous Model

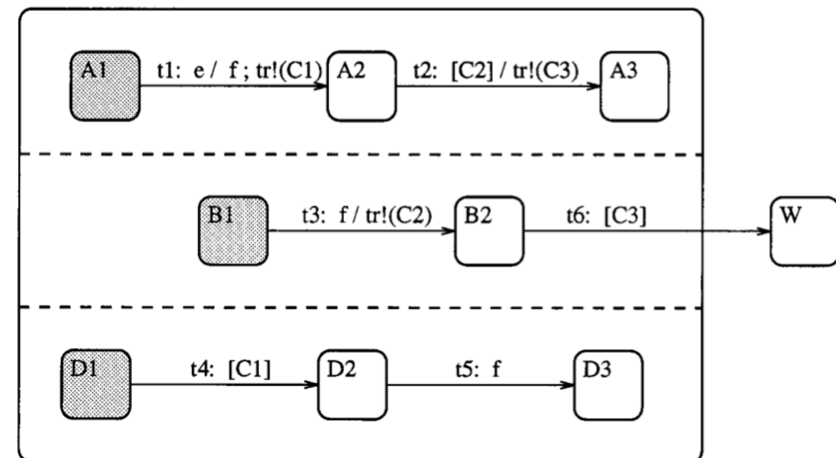
- Used for most kinds of asynchronous systems.
- Since execution of steps “take” *zero internal time*, the simulator must advance the internal time explicitly.
- Different GO commands allow user to control the advance of time:
  - GO-REPEAT
  - GO-ADVANCE
  - GO-STEP
  - GO-NEXT
  - GO-EXTENDED

GO-REPEAT

- Steps:
  - Execute all external changes since completion of previous step.
  - Execute all timeout events and scheduled actions that are due.
  - Repeatedly execute one step until the system is in a *stable state* (there are no generated events and no enabled CTs or SRs).
- Does not increment the internal clock, so many steps can be executed at the same time. The repeat loop is thus a *superstep*.
- Can result in an infinite loop. Suspected infinite loops are reported.

GO-REPEAT

- Assume that C1, C2, C3 are false, and environment generates event e.
- Transition t1 is taken; system goes into {A2, B1, D1}; C1 is now true and f is generated.
- Transitions t3 and t4 are taken; system goes into {A2, B2, D2}; C2 is now true.
- Transition t2 is taken; system goes into {A3, B2, D2}; C3 is now true. t5 is not taken because f is not "alive".
- Transition t6 is taken; system is in {W}.



GO-ADVANCE

- Used in conjunction with `GO-REPEAT` to advance the clock.
- Steps (advance from `t` to `t + n`):
  - Execute all external changes since completion of previous step.
  - Set `t'` := `t + n`.
  - Repeat the following until `t = t'`:
    - Execute all timeout events and scheduled actions that are due.
    - Execute `GO-REPEAT`.
    - Set `t"`:= time of closest scheduled action or timeout event.
    - Set `t` := `min(t', t")`.

Other useful GO commands

- `GO-STEP`: Execute one step without advancing the time.
- `GO-NEXT`: Advance the clock to the time of the next timeout event or scheduled action without carrying out a step. Before the time is actually advanced, all steps that can be executed are executed.
- `GO-EXTENDED`: `GO-NEXT + GO-REPEAT`.
  - Execute all external changes since the previous step.
  - If there are generated events or enabled CTs or SRs:
    - Execute a superstep.
    - Else:
      - Advance clock to time of next timeout event or scheduled action.
      - Execute the scheduled actions and timeout events that are due.
      - Execute a superstep.

STATEMATE Implementation

- Hardware code generators let the user select between two code styles in the generated HDL code:
  - **RTL code style**: Code executes at the rising or falling edge of a clock = Synchronous mode.
  - **Behavioral code style**: Code reacts to any change in the inputs the moment they occur = Asynchronous mode.
- Software code generators generate one style of code, but two different schedulers are provided that support different time models.

STATEMATE Implementation

- One scheduler uses CPU clock time.
  - Steps and supersteps take more than zero time.
  - External changes are sensed only at the start of a step.
  - External changes, timeout events, scheduled actions may occur before system has stabilized.
  - The equivalent of GO-REPEAT is not supported.
- Other scheduler uses simulated clock.
  - Clock only advances after the system is in a stable status.
  - External changes, timeout events, scheduled actions occur only when the system is stable.
  - Behavior identical to asynchronous mode.

Racing Conditions

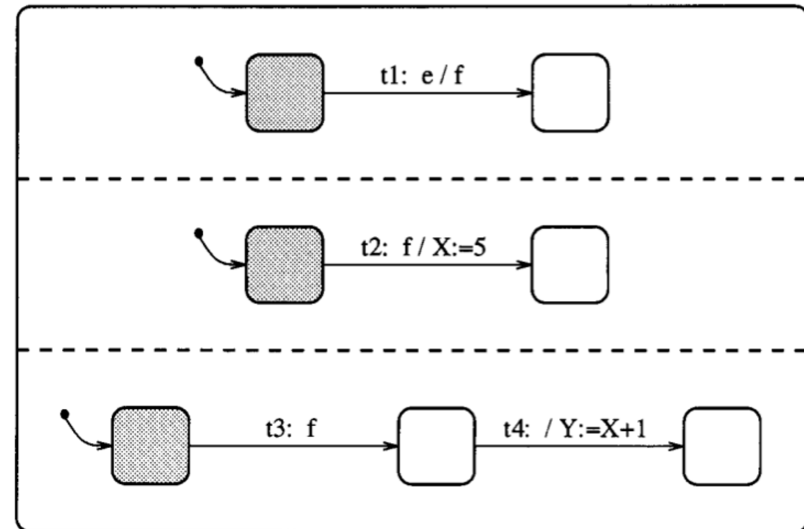
- Occur when value of an element is modified more than once, or is modified and used at a single point in time.
- Our approach is greedy: multiple steps can be executed at “the same point in time”, so racing problems can arise both in a superstep and between transitions or actions executed in different steps.
- However, we should consider causality dependencies between transitions in a single superstep. If there is a transition labeled `e / f; X := 5`, that enables another transition labeled `f / X := 6` to be executed, there is no “racing condition”.

Racing Conditions

- What is a precise definition?
  - In each step and superstep, several transitions may be enabled.
  - Enabled transitions have a specific "enabling order": each transition is to be executed after the ones that enabled it.
  - There is a *race condition* if, had we executed the enabled transitions in a different order (yet legal), we would have obtained a different state.

Racing Conditions

+ When event `e` occurs, transition `t1` will be taken, then `t2` and `t3` in the next step, and finally `t4`.
+ X should get the value 5 and Y should get the value 6.
+ *But*, semantics prescribe that `t1` happen before `t2` and `t3`, and that `t4` happen after `t3`: **`t2` can be postponed** and still produce a “legal” output.
+ Y could have a value different from 6.



Appendix A: Comparison with Other Work

- Candidate for comparison is the RSML language of Leveson et al. (1995): very similar underlying principles, main differences are syntactical.
- von der Beek (1994) lists 19 issues relevant to proposals for semantics of statecharts.
  - Some are questions about which features the language supports.
  - Semantic aspects of most issues are relevant only to supersteps.

jo_ko_berkeley@berkeley.edu