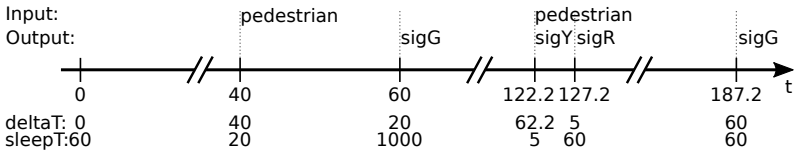


Alexander Schulz Rosengarten, *Reinhard von Hanxleden*
Kiel University

Fréderic Mallet, Robert de Simone, Julien DeAntoni
INRIA Sophia Antipolis

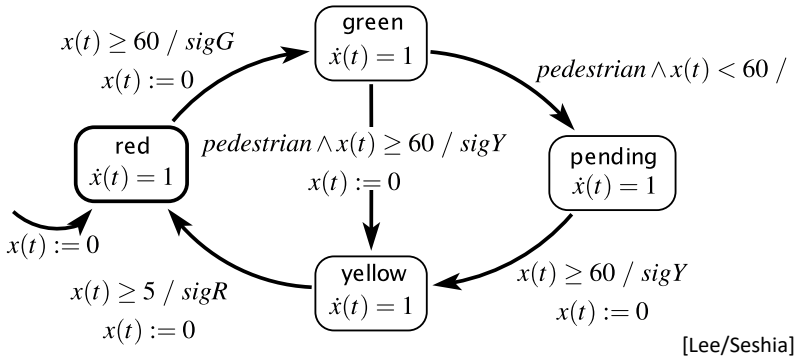
Lecture 15: Time in SCCharts

Alexander Schulz Rosengarten, *Reinhard von Hanxleden*
Kiel University

Fréderic Mallet, Robert de Simone, Julien DeAntoni
INRIA Sophia Antipolis

Traffic Light as Timed Automaton

continuous variable: $x(t)$: $\mathbb{R}$
inputs: *pedestrian*: pure
outputs: *sigR*, *sigG*, *sigY*: pure

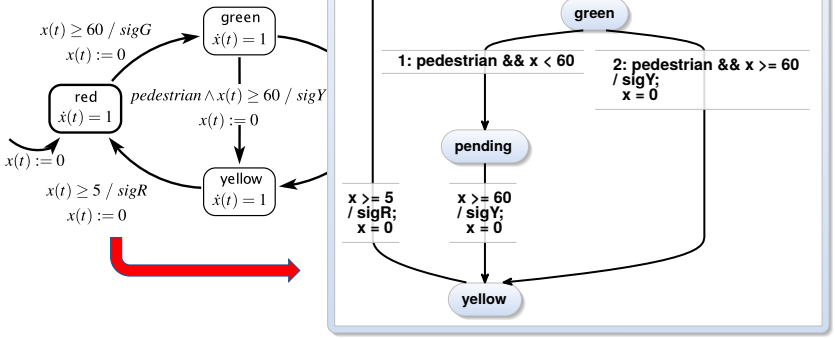


[Lee/Seshia]

Alur, Dill, *A theory of timed automata*, Theoretical Computer Science, 1994

Traffic Light in SCCharts

continuous variable: $x(t)$: $\mathbb{R}$
inputs: *pedestrian*: pure
outputs: *sigR*, *sigG*, *sigY*: pure



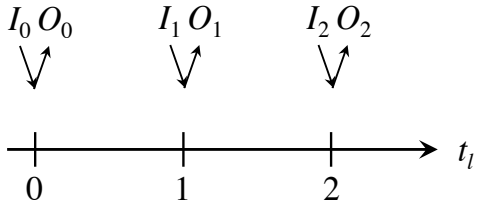
Roadmap

1. Traffic Light Example
2. Execution Models
3. Dynamic Ticks
4. Time in SCCharts: “clock”
5. Multiclocks in SCCharts: “period”
6. Demo

Roadmap

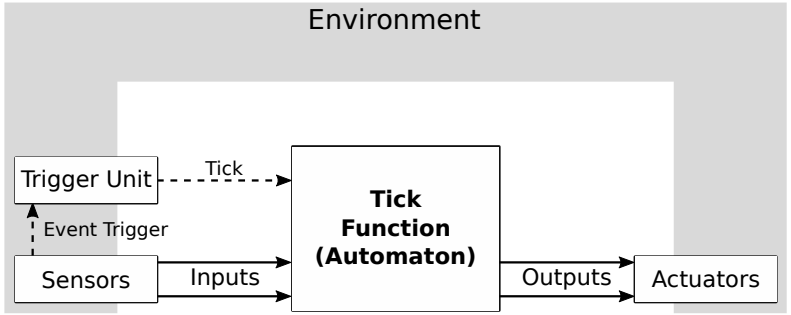
1. Traffic Light Example
2. Execution Models
3. Dynamic Ticks
4. Time in SCCharts: “clock”
5. Multiclocks in SCCharts: “period”
6. Demo

Discrete (Logical) Time in Synchronous Programming



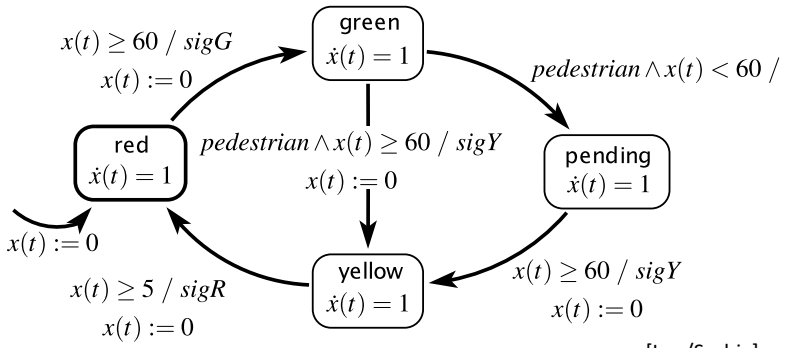
- Synchrony Hypothesis:
  Outputs are synchronous with inputs
- Computation "does not take time"
- Actual computation time does not influence result
- Sequence of outputs **determined** by inputs

Event-Triggered Execution



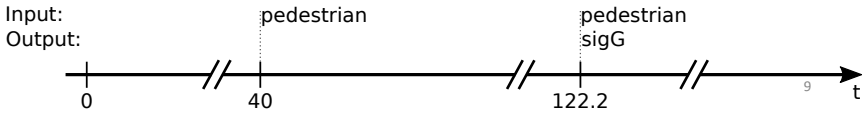
continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure

Assume pedestrian button pressed at t = 40 and t = 122.2



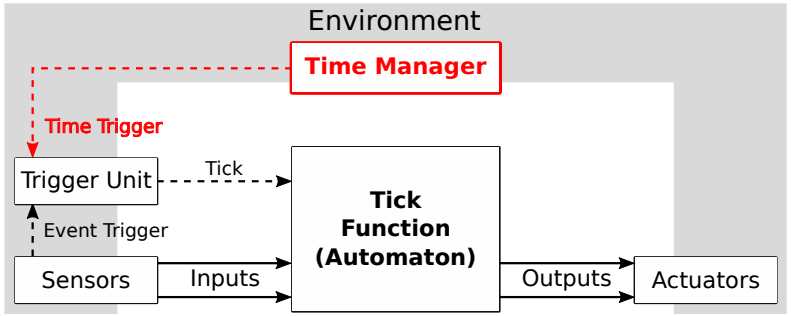
[Lee/Seshia]

Event-Triggered Execution, with initial tick at t = 0:



9

Time-Triggered Execution



11

Synchronous Execution

```
Initialize Memory
for each input event do
    Compute Outputs
    Update Memory
end
```

```
Initialize Memory
for each clock tick do
    Read Inputs
    Compute Outputs
    Update Memory
end
```

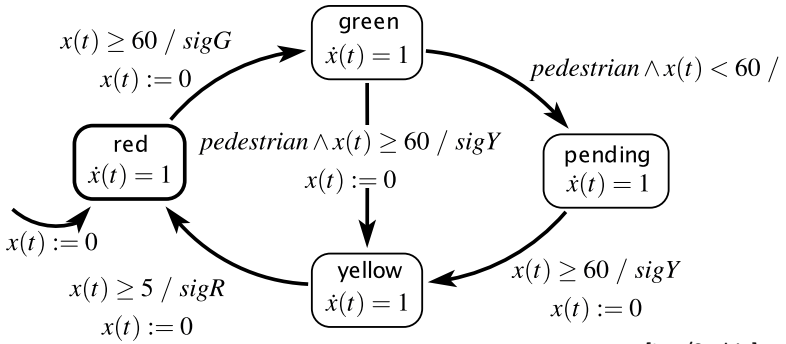
Fig. 1 Two common synchronous execution schemes: event driven (left) and sample driven (right).

[Benveniste et al., *The Synchronous Languages Twelve Years Later*, Proc. IEEE, 2003]

10

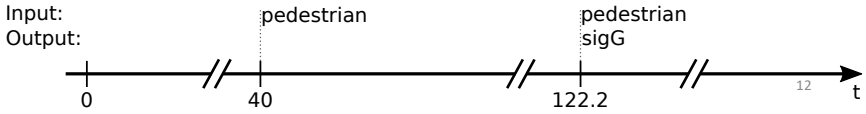
continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure

Assume pedestrian button pressed at t = 40 and t = 122.2



[Lee/Seshia]

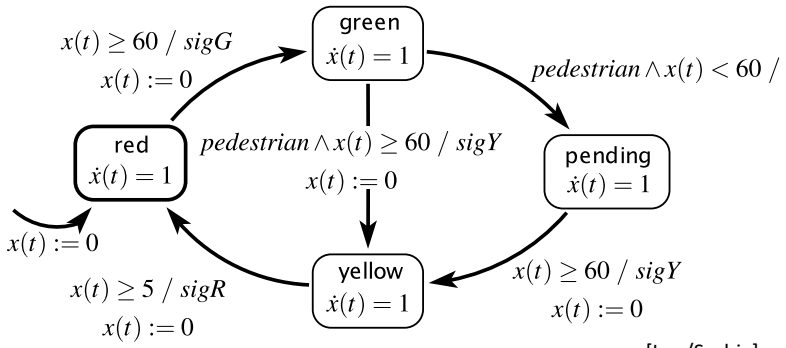
Recall: Event-Triggered Execution:



12

continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian*: pure
outputs: *sigR*, *sigG*, *sigY*: pure

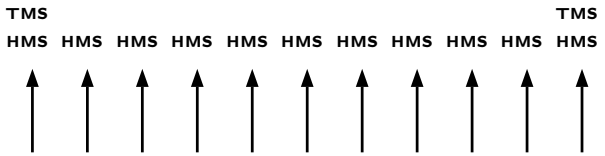
Assume pedestrian button pressed at t = 40 and t = 122.2



[Lee/Seshia]

Time-Triggered Execution (every 5 sec):

Packaging Physical Time as Events



[Timothy Bourke, SYNCHRON 2009]

Event "HMS": 100 µsec have passed since last HMS
Event "TMS": 1000 µsec have passed since last TMS

Multiform Notion of Time

Only the simultaneity and precedence of events are considered.

This means that the physical time does not play any special role.

This is called multiform notion of time.

[https://en.wikipedia.org/wiki/Esterel]

A Problem With That ...

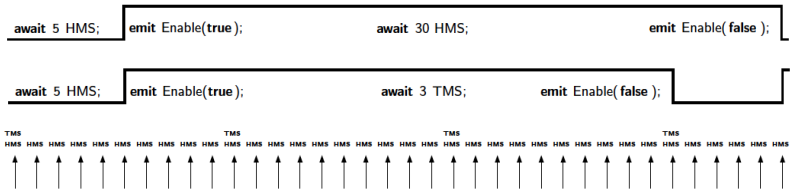
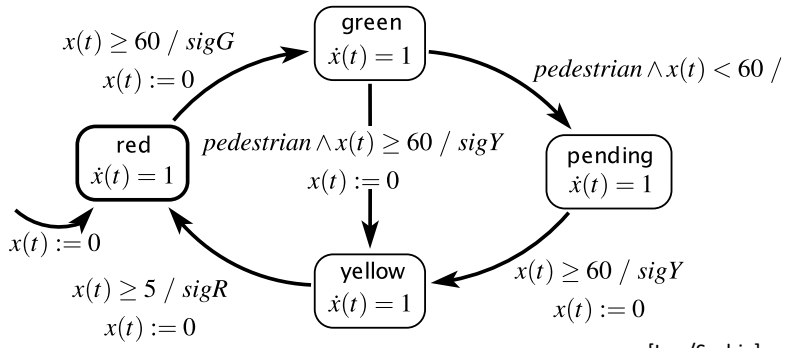


Fig. 4: Granularity of timing inputs

[Timothy Bourke, SYNCHRON 2009]

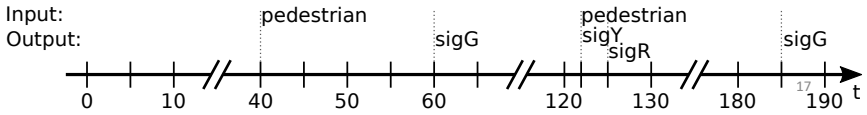
continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure

Assume pedestrian button pressed at t = 40 and t = 122.2



[Lee/Seshia]

Time-Event-Triggered Execution, Multiform Time:

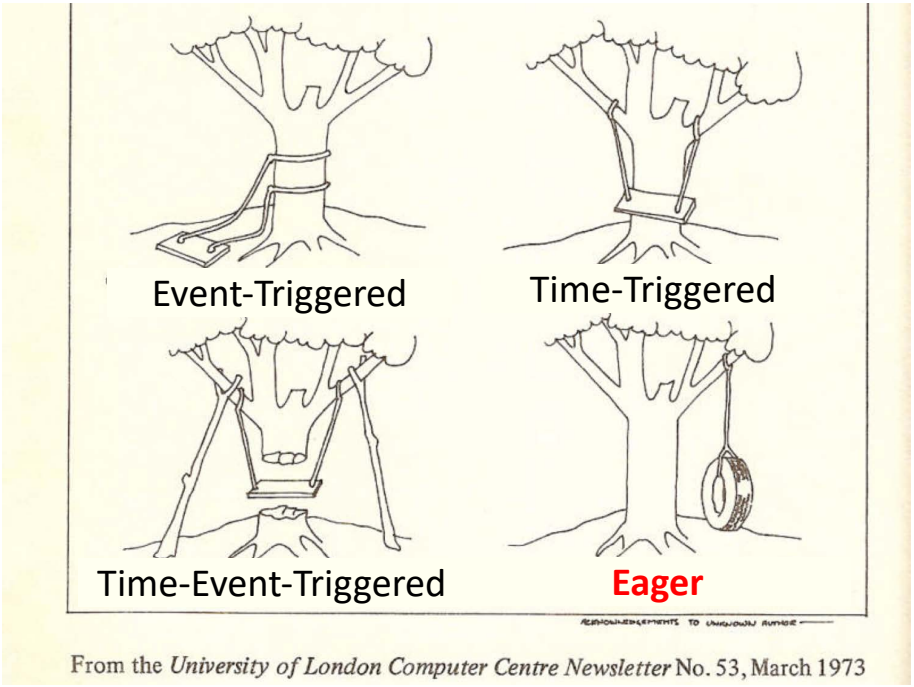


What the User (Probably) Wanted

„We assume here that a transition is taken as soon as it is enabled. Other transition semantics are possible.“

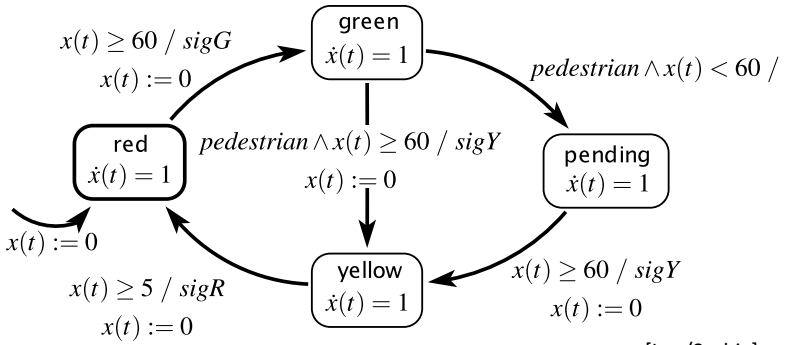
[Lee/Seshia 2017]

We call this **eager** semantics.



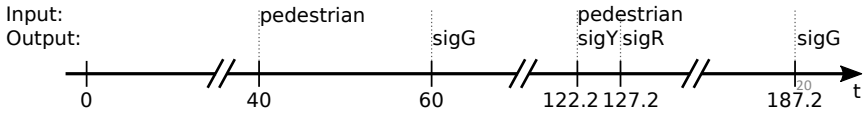
continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure

Assume pedestrian button pressed at t = 40 and t = 122.2



[Lee/Seshia]

Eager Semantics:



Time in SCCharts – Requirements

1. Seamless fit into synchronous paradigm
   - Still deterministic behavior – outputs fully determined by inputs
   - No changes to underlying SC (Sequentially Constructive) MoC
2. Approximate eager semantics
   - Modulo run-time variations and imperfections of physical timers
3. Scalability
   - E.g., allow arbitrary number of (concurrent) timers
4. Fine granularity
   - Gcd may be arbitrarily small, w/o performance penalty
   - E.g., may have timeouts of 1 sec and 3.1415926 msec in same model
5. Time composability
   - E.g., waiting 1 sec. twice should mean the same as waiting 2 sec's once

Time in SCCharts – Requirements

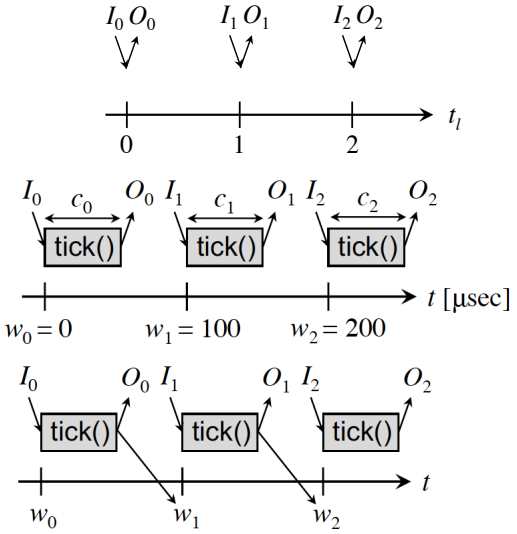
6. Preserve temporal order and simultaneity
   - E.g., timers started in same tick and running same duration should expire in same tick
7. Minimize impact of physical timer variations
   - E.g., avoid accumulations of timer imperfections
8. Give application access to physical time and tick computation time
   - Facilitates e.g. load-dependent execution modes
9. Lean, application-independent interface to environment
   - E.g., interface should not change if number of timers changes
10. Fit into Single Language-Driven Incremental Compilation (SLIC) concept
    - New timing constructs are just syntactic sugar on top of existing SCCharts
    - Transforming away timing constructs requires only local changes
    - No changes needed to compilation back-end

Roadmap

1. Traffic Light Example
2. Execution Models
3. Dynamic Ticks
4. Time in SCCharts: "clock"
5. Multiclocks in SCCharts: "period"
6. Demo

Dynamic Ticks

- Recall logical time:
- Physical time, time-triggered:
- Physical time, dynamic ticks:



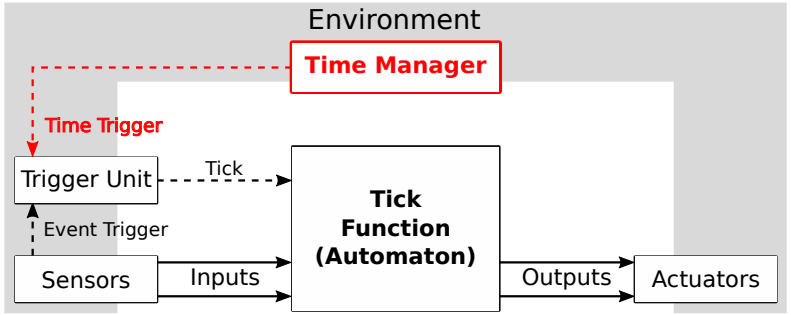
Esterel

```
module PAUSE_USEC:

input current_usec : integer;                  % Simulated time
input wait_usec : integer;                     % Time of delay
function min(integer, integer) : integer;
output wake_usec : combine integer with min;   % Time of next wake up
var my_wake_usec : integer in                  % Local copy of wake_usec

% Compute physical time when PAUSE_USEC should terminate
my_wake_usec := ?current_usec + ?wait_usec;

% Loop until current_usec = my_wake_usec
trap done in
 loop
  emit wake_usec(my_wake_usec);
  pause;
  if ?current_usec = my_wake_usec then
   exit done;
  end if;
 end loop
end trap
end var
end module
```

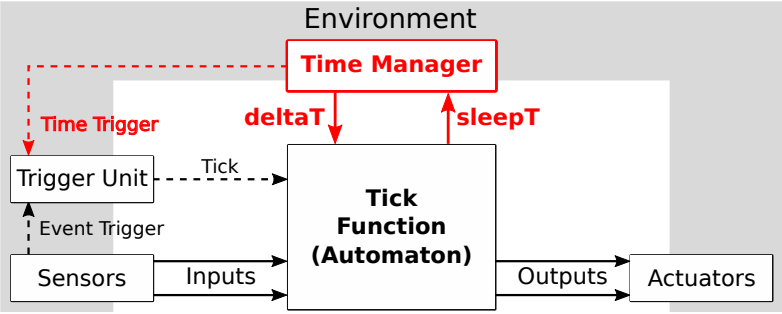

Host Code

```
int main()
{
 int notDone, prev_tick_end_usec = 0;

 RACE_reset();    // Reset automaton
 time_reset();    // Initialize time

 // Loop until tick function terminates
 do {
  // Set inputs
  RACE_I_current_wall_usec(get_current_wall_usec());
  RACE_I_prev_tick_end_usec(prev_tick_end_usec);

  notDone = RACE(); // Call tick function
  prev_tick_end_usec = get_current_wall_usec();

  // Wait until wake_usec
  microsleep(wake_usec - prev_tick_end_usec);
 } while (notDone);

 return 0;
}
```


Recall: Time-Triggered Execution



Eager Execution with Dynamic Ticks

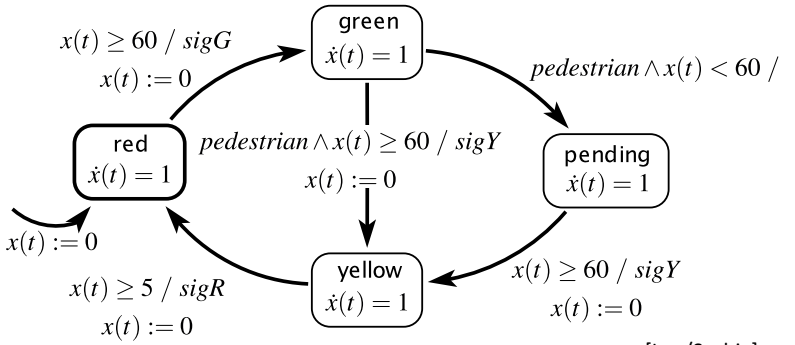


deltaT: Time since last tick

sleepT: Requested delay until next tick

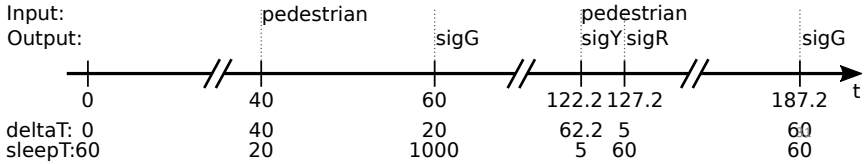
continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure

Assume pedestrian button pressed at t = 40 and t = 122.2



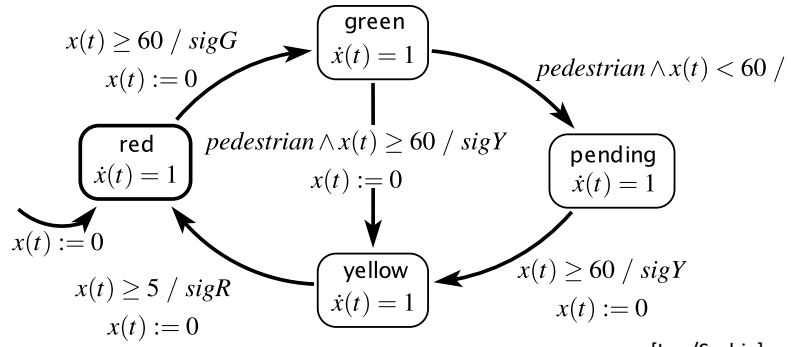
[Lee/Seshia]

Eager Execution with Dynamic Ticks:



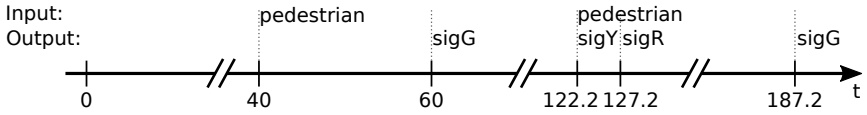
continuous variable: $x(t) : \mathbb{R}$
inputs: *pedestrian* : pure
outputs: *sigR*, *sigG*, *sigY* : pure

Assume pedestrian button pressed at t = 40 and t = 122.2



[Lee/Seshia]

Recall: Eager Semantics



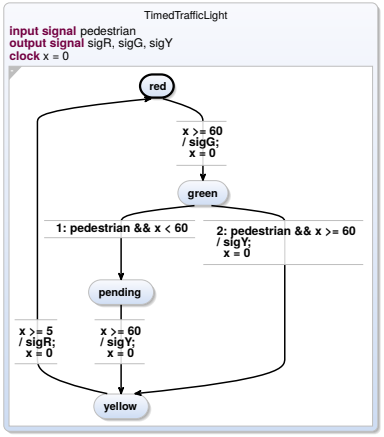
Multiform Notion of Time – Again!

- Semantically, treat clocks (time) as a unit-less number
- As in timed automata, clocks must satisfy *monotonicity* (modulo resets) and *progress*
- Current implementation maps time (clock variables) to an approximation of real numbers (float), interpreted as seconds
- However, could also map clocks to integers, interpreted as Euros spent, fathoms travelled, or beers consumed

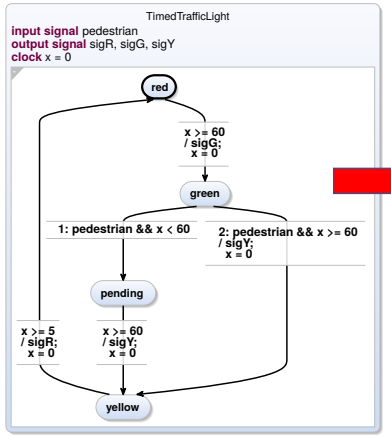
Roadmap

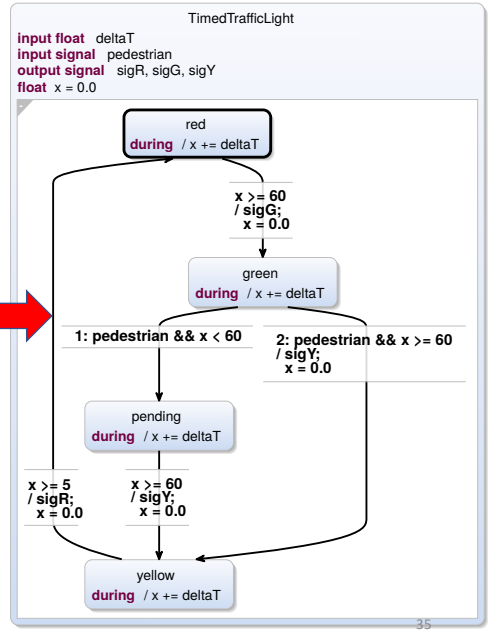
1. Traffic Light Example
2. Execution Models
3. Dynamic Ticks
4. Time in SCCharts: “clock”
5. Multiclocks in SCCharts: “period”
6. Demo

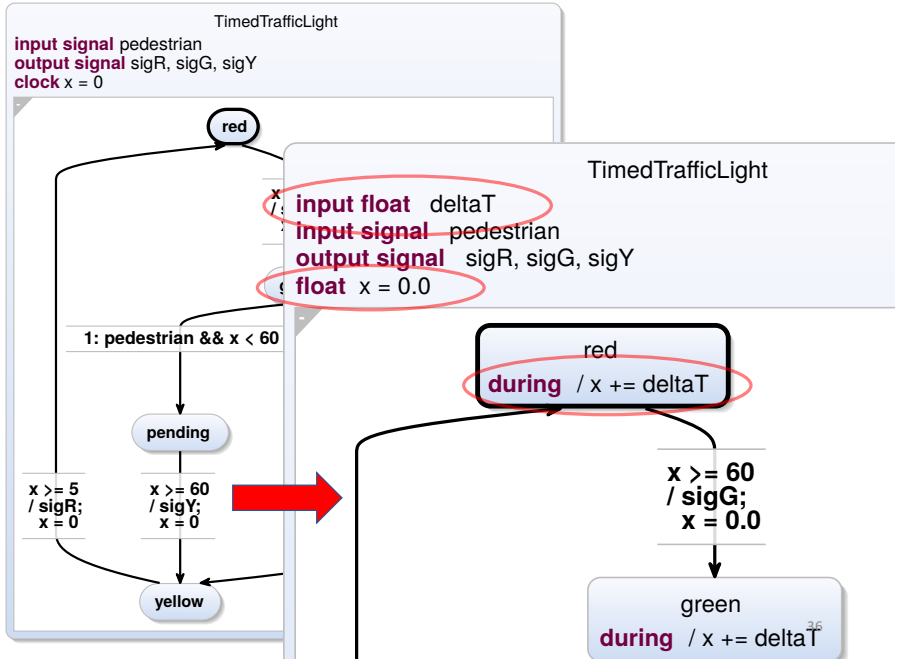
Recall: Traffic Light in SCCharts



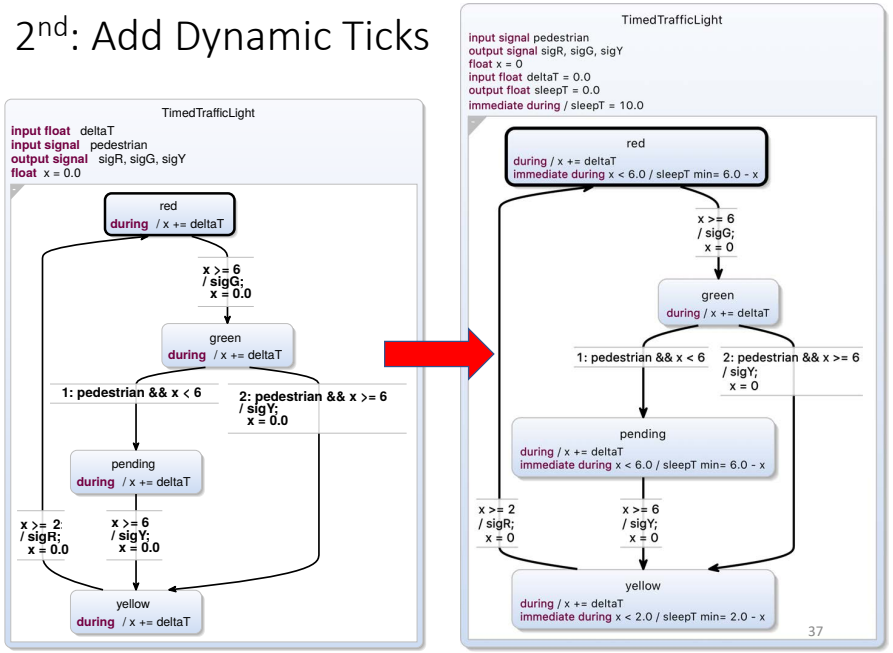
1st: Expand Clock





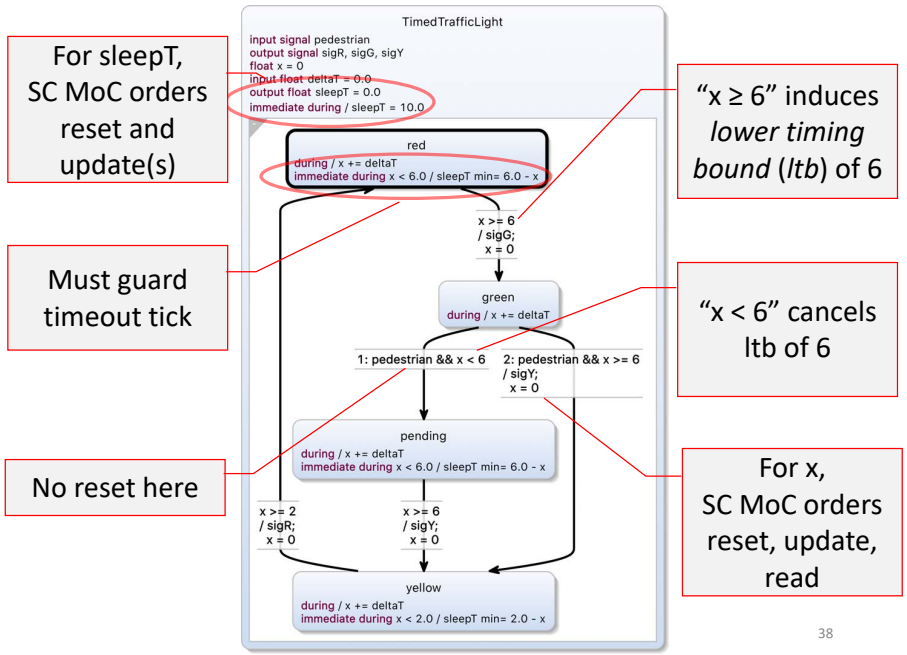


2nd: Add Dynamic Ticks

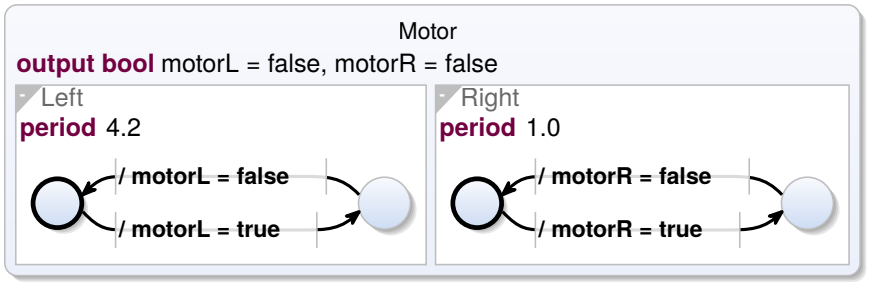


Roadmap

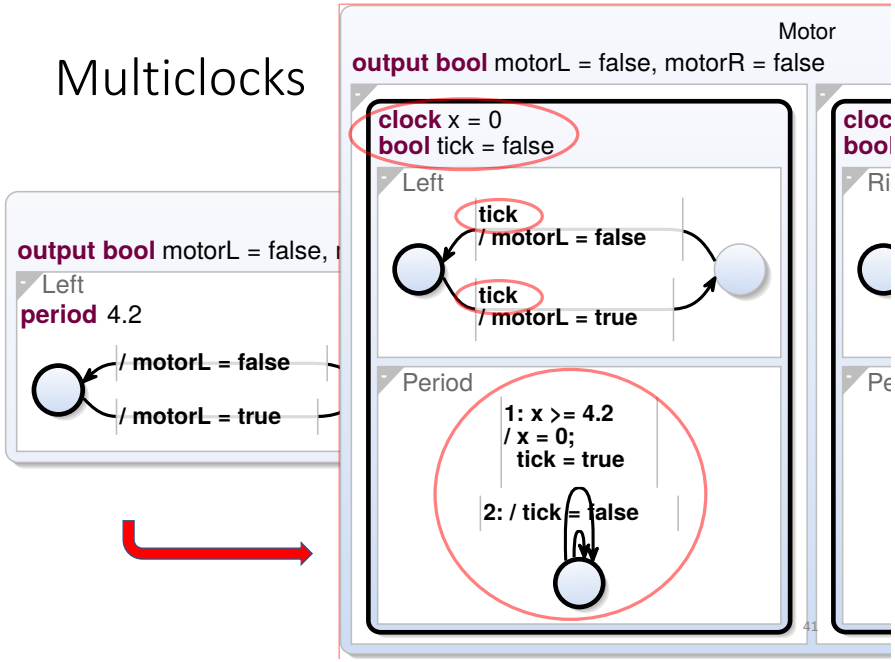
1. Traffic Light Example
2. Execution Models
3. Dynamic Ticks
4. Time in SCCharts: “clock”
5. Multiclocks in SCCharts: “period”
6. Demo

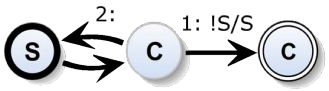


Multiclocks



Multiclocks





SCCharts

http://www.sccharts.com/

KIELER

The Key to Efficient Modeling

http://www.rtsys.informatik.uni-kiel.de/en/research/kieler

Eclipse Layout Kernel

https://www.eclipse.org/elk/

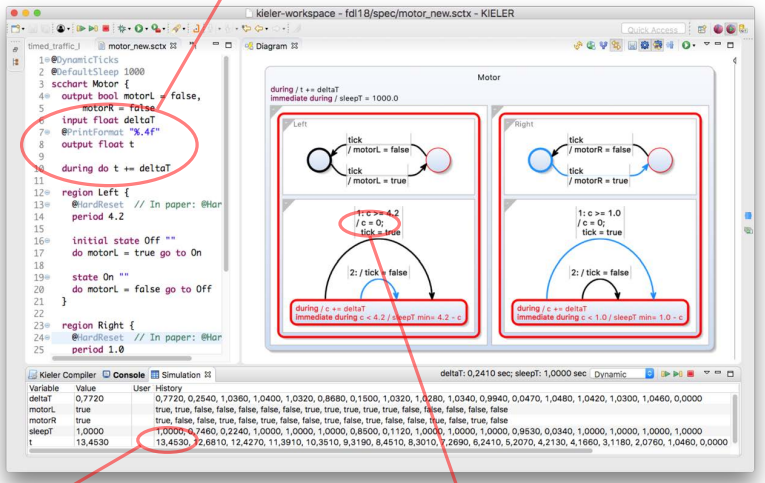
All available as open source under EPL

Roadmap

1. Traffic Light Example
2. Execution Models
3. Dynamic Ticks
4. Time in SCCharts: “clock”
5. Multiclocks in SCCharts: “period”
6. Demo

”avoid accumulations of timer imperfections”

Grab deltaT from environment, derive from it physical time t

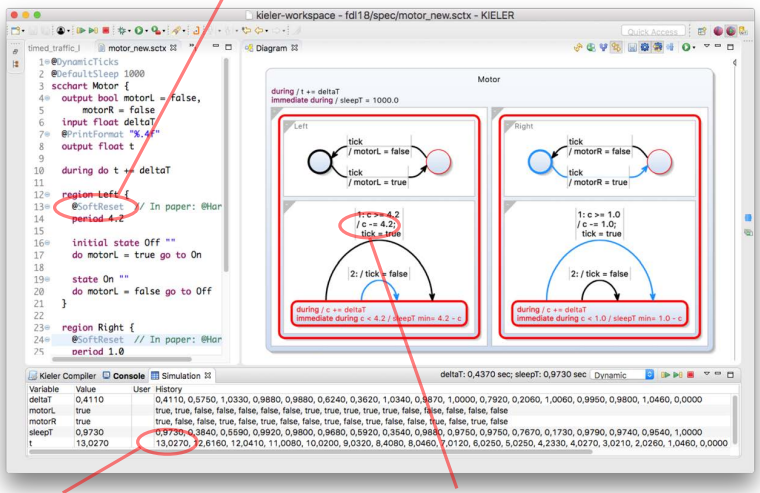


PROBLEM: After 13 sec, accumulated 0.453 sec delay!

The culprit: always reset clocks to 0!

”avoid accumulations of timer imperfections”

SOLUTION: Change @HardReset (in paper) to @SoftReset (now default)



After 13 sec, accumulated only 0.027 sec delay!

Instead of reset to 0, subtract previously requested time-out

To Go Further

- Reinhard von Hanxleden, Timothy Bourke, Alain Girault. Real-Time Ticks for Synchronous Programming. *Proc. Forum on Specification and Design Languages (FDL ’17)*, Verona, Italy, September 2017 https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/fdl17.pdf
- Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, Julien Deantoni. Time in SCCharts. In *Proc. Forum on Specification and Design Languages (FDL ’18)*, Munich, Germany, September 2018 https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/fdl18.pdf

Summary

- Timed automata used not just for verification, but also for synthesis
- Synchronous execution model cleanly contains non-determinism, at timing I/O-interface
- Can extend easily to multi-clock design
- Multiform notion of time retained – but package “time” not as events, but clocks (represented as, e.g., integers)
- Added two keywords (clock, period) as extended SCChart features
- Further annotations (@HardReset, @DefaultSleep) to control external interface
- Same concepts can be applied to other synchronous languages