Overview of the synchronous approach

Partha Roop, Avinash Malik, Sidharta Andalam

THE UNIVERSITY OF AUCKLAND
NEW ZEALAND
Te Whare Wānanga o Tāmaki Makaurau

BioRemulation™ Research Group
The University of Auckland

December 2016

www.pretzel.ece.auckland.ac.nz/bio

Outline

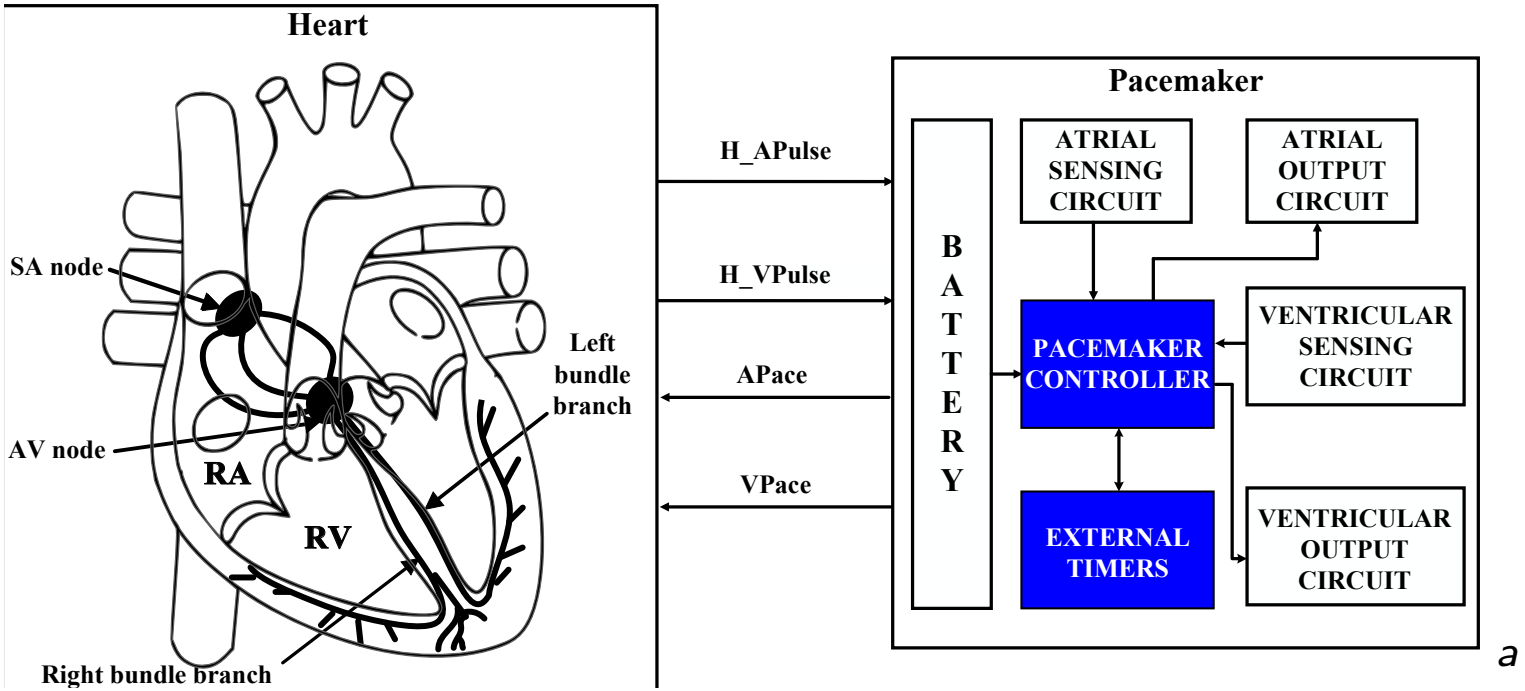
Some acknowledgements

1. Prof. Reinhard von Hanxleden for sharing his lecture notes on SyncCharts and for permission to reuse some Figures.
2. The developers of the Kieler project `http://www.rtsys.informatik.uni-kiel.de/en/research/kieler/` for help with the SCCharts tools.

Learning outcomes

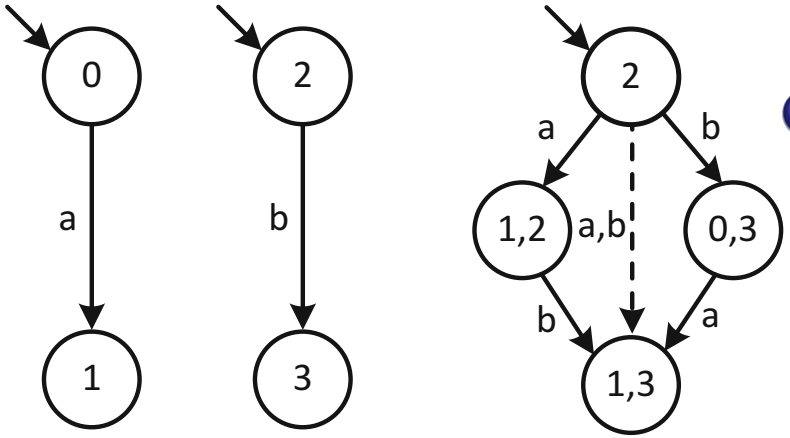
1. Students will be able to appreciate the "synchronous abstraction" for the modelling and synthesis of reactive / real-time systems.
2. They will be able to understand the synchronous and real-time nature of the heart and the pacemaker.
3. They will be able to use the synchronous approach to simplify code generation from both cardiac models (plant models) and medical devices (controllers).
4. They will learn two different synchronous languages: Esterel following the imperative style and SCCharts following a visual style.
5. They will be able to design simple pacemakers using the synchronous approach.

A cyber-physical system (CPS)



[a] Zhao and Roop, "Model Driven Design of Cardiac Pacemakers using IEC61499, CRC Press, 2015".

Asynchrony vs synchrony



1. Asynchronous concurrency is adopted in process algebras such as *communicating sequential processes* (CSP) [3] and *calculus of communicating systems* (CCS) [4].
2. Asynchronous composition leads to interleaving: $a \rightarrow b$ (a followed by b) or $b \rightarrow a$ (b followed by a) or a, b (a and b happening simultaneously) are all feasible.
3. We will see that with the *synchronous approach* [1], such interleaving disappears.

1. Idealized reactive system where there is no delay between input and the corresponding output as the system executes infinitely fast.
2. Atomicity of reactions–new events are not accepted until current reaction completes.
3. Advantages–all correct synchronous programs are deterministic and reactive.
4. Disadvantage–causality issues.
5. I have based the Esterel part of this lecture on [6].

ABRO

```
module ABRO:
input A, B, R;
output O;

loop

    [await A || await B];
    emit O;

each R

end module
```

- Developed by Gerard Berry, this is known as the "Hello World" of synchronous programming [2].
- Captures many features of Esterel: *sequential composition, parallel composition, preemption, priority, signal emission, and delay*.
- While input may grow exponentially, the program grows linearly.

Kernel statements

Statement	Meaning	Type
`nothing`	Terminate immediately	Instantaneous
`emit S`	Emit signal S	Instantaneous
`present S then` t `else` u `end`	Run t if S is present; otherwise u	Instantaneous
`suspend t when S`	Freezes the state of the body `t` when `S` is present	Either
`pause`	Pause execution till the next instant	Delayed
$t;u$	Run t, and then u in sequence	Either
$t \| \| u$	Run t and u concurrently	Either
`loop` t `end`	Repeat t forever	Delayed
`trap T in` t `end`	Declare and catch exception T in t	Either
`exit T`	Raise exception for T	Instantaneous
`signal S in t end`	Declare a local signal S whose scope is `t`	N/A

Await

```
%%derived statement%%
await S;
%%equivalent mapping using kernel statements%%
trap T in
    loop
        pause;
        present S then exit T end;
    end loop
end trap
```

(a) Await mapping

```
%%derived statement%%
await immediate S;
%%equivalent mapping using kernel statements%%
trap T in
  loop
     present S then exit T end;
     pause;
  end loop
end trap
```

(b) Await immediate mapping

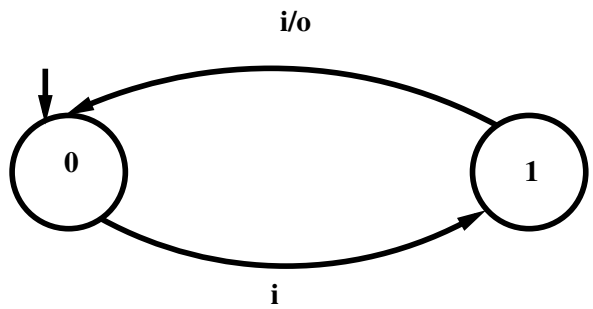
Sustain and Weak Abort

```
1     %%derived statement%%
2     sustain S;
3     %%equivalent mapping using kernel statements%%
4     loop
5       emit S;
6       pause;
7     end loop
8     %%derived statement%%
9     weak abort t when S;
10    %%equivalent mapping using kernel statements%%
11    trap T in
12      t;
13      exit T;
14    ||
15      await S;
16      exit T;
17    end trap
```

FSM Mapping

```
1  module OneBit:
2  input i;
3  output o;
4
5  var state := 0: integer in
6  loop
7    trap T1 in
8    pause
9     if state = 0 then
10     present i then
11      state := 1;
12      exit T1;
13     end present;
14    end if;
15    if state = 1 then
16     present i  then
17      state := 0;
18      emit o;
19      exit T1;
20     end present;
21    end if
22   end trap
23  end loop
24 end var
25 end module
```

(c) FSM Mapping to Esterel



(d) Simple FSM

Four bit counter – module reuse

```
1  module FourBit:
2  input i;
3  output stop;
4  signal a, b, c  in
5      %Thread -- T1
6     run OneBit[signal i/i, a/o];
7     ||
8      %Thread -- T2
9     run OneBit[signal a/i, b/o];
10    ||
11     %Thread -- T3
12    run OneBit[signal b/i, c/o];
13    ||
14     %Thread -- T4
15    run OneBit[signal c/i, stop/o];
16 end loop
17 end var
18 end signal
19 end module
```

(e) A four-bit counter

Tick	Inputs	Outputs	T4,T3,T2,T1
0	-	-	0, 0, 0, 0
1	i	-	0, 0, 0, 1
2	i	a	0, 0, 1, 0
3	i	-	0, 0, 1, 1
4	i	a, b	0, 1, 0, 0
5	i	-	0, 1, 0, 1
6	i	a	0, 1, 1, 0
7	i	-	0, 1, 1, 1
8	i	a, b, c	1, 0, 0, 0
9	i	-	1, 0, 0, 1
10	i	a	1, 0, 1, 0
11	i	-	1, 0, 1, 1
12	i	a, b	1, 1, 0, 0
13	i	-	1, 1, 0, 1
14	i	a	1, 1, 1, 0
15	i	-	1, 1, 1, 1
16	i	a, b, c, stop	0, 0, 0, 0

(f) A sample trace

A simple example

```
module Simple:
  input I; output Q, O := false : boolean;
  trap T in
    loop
      emit O(not pre(?O));
      present I then exit T end; pause;
    end loop
  ||
    loop
      await immediate pre(O);
      emit Q; pause;
    end loop
  end trap
end module
```

(g) Code

Tick	Inputs	Outputs
0	–	O (true)
1	–	O (false), Q
2	I	O (true), Q *(program exits)*

(h) Behaviour

1. The code in Figure 1(g) is an Esterel module with concurrency, synchronisation, and preemption.
2. One trace of this program is shown in (Figure 1(h)).

User Interface of an MP3-Player

```
module MP3_UI:
input Play, Stop, Lock, Unlock;
output CodeForPlay, CodeForStop;
signal Change, Locked in
[loop
      suspend
         await Play;
         emit Change;
      when Locked;
      abort
         sustain CodeForPlay;
      when Change
   end]
||
[loop
      suspend
         await Stop;
         emit Change;
      when Locked;
      abort
         sustain CodeForStop;
      when Change
   end]
||
[
every Lock do
      abort
         sustain Locked
      when Unlock
   end
]
end signal
end module
```

Types of abort

Abort Type	Abort checking is done during the tick	Preemption can happen in the 1st instant	Checking order
Strong	Start of tick	No	Outer to inner
Strong Immediate	Start of tick	Yes	Outer to inner
Weak	End of tick	No	Inner to outer
Weak Immediate	End of tick	Yes	Inner to outer

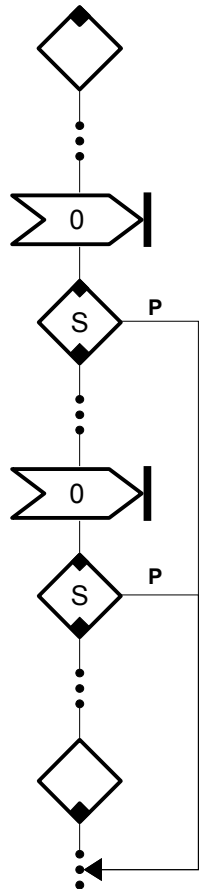
Strong abort

```
1 abort
2     % instantaneous statements
3     pause;
4     % instantaneous statements
5     pause;
6     % instantaneous statements
7 when S
8 % instantaneous statements
```

(a)

```
1       ABORT Sn CONT
2       ; some instructions
3       PAUSE #n
4       CHKABORT STRONG
5       ; some instructions
6       PAUSE #n
7       CHKABORT STRONG
8       ; some instructions
9 CONT ; some instructions
```

(b)



(c)

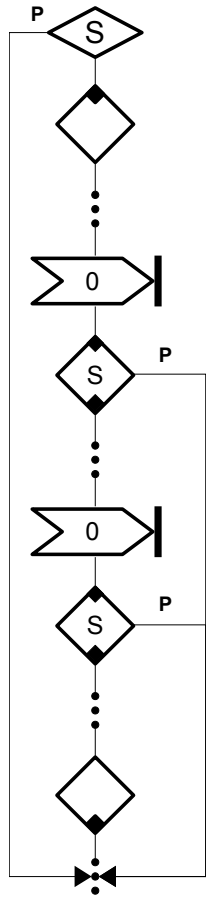
Strong immediate of abort

```
1  abort
2      % instantaneous statements
3      pause;
4      % instantaneous statements
5      pause;
6      % instantaneous statements
7  when immediate S
8  % instantaneous statements
```

(a)

```
1       ABSENT Sn CONT
2       ABORT Sn CONT
3       ; some instructions
4       PAUSE #n
5       CHKABORT STRONG
6       ; some instructions
7       PAUSE #n
8       CHKABORT STRONG
9       ; some instructions
10 CONT ; some instructions
```

(b)



(c)

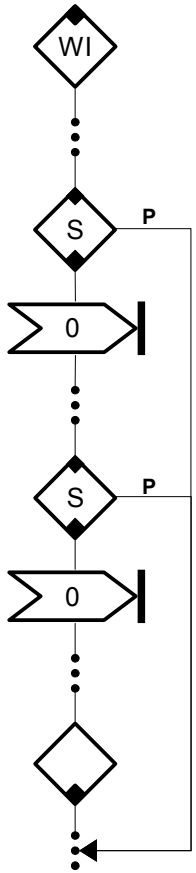
Weak immediate of abort

```
1 weak abort
2     % instantaneous statements
3     pause;
4     % instantaneous statements
5     pause;
6     % instantaneous statements
7 when immediate S
8 % instantaneous statements
```

(a)

```
1       WIABORT Sn CONT
2       ; some instructions
3       CHKABORT WEAK
4       PAUSE #n
5       ; some instructions
6       CHKABORT WEAK
7       PAUSE #n
8       ; some instructions
9 CONT  ; some instructions
```

(b)



(c)

Weak abort

- Is weak-abort in any way special?
- Structural translation – homework.

```
suspend
        pause; //state-A
        emit X;
        pause; //state-B
        emit Y;
when S;
emit Z;
```

Behaviour

1.- ---> - new state A

2.- ---> X new state B

3.S ---> - new state B

4.S ---> - new state B

5.- ---> Y,Z state done

Statecharts proposed by David Harel [1987]

Statecharts proposed by David Harel [1987]

In a nutshell:

Statecharts	=	Mealy Machines
	+	hierarchy (depth)
	+	orthogonality
	+	broadcast
	+	data

SyncCharts are made up of elements common to most Statecharts dialects:

- States
- Initial/terminal states
- Transitions
- Signals/Events
- Hierarchy
- Modularity
- Parallelism

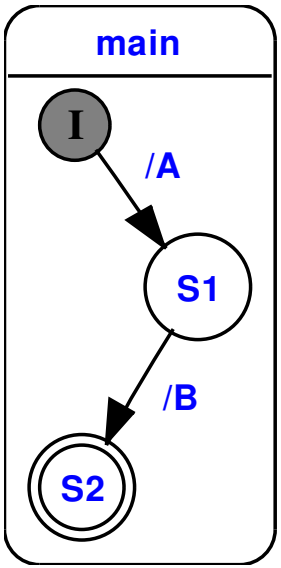
SyncCharts differ from other implementations of Statecharts:

- Synchronous framework
- Determinism
- Compilation into backend language Esterel
- No interpretation for simulations
- No hidden behaviour
- Multiple events
- Negation of events
- No inter-level transitions

Simple Sequential Automaton

SyncChart:

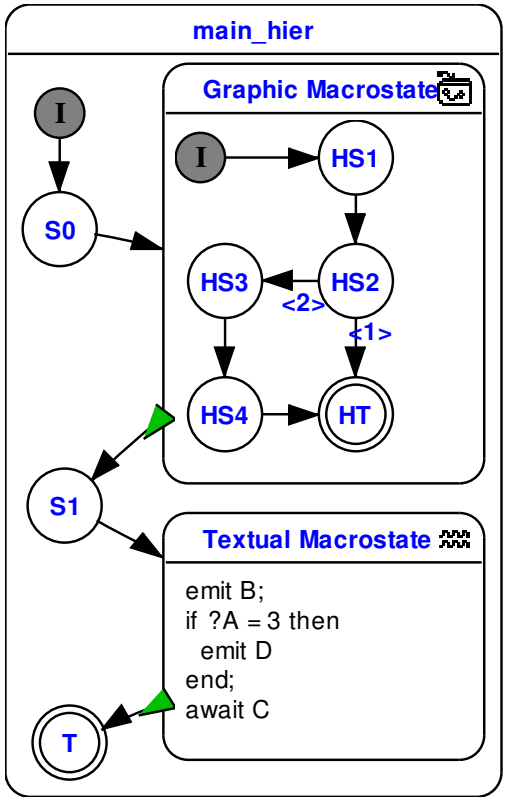
SyncChart:



Elements:

- States:
  - Regular state (circle)
  - Terminal state (doubled circle)
  - Hierarchic state (box with rounded edges)
- Transitions:
  - Arrows with labels
- Connectors:
  - Colored circles with single letters

Hierarchic States

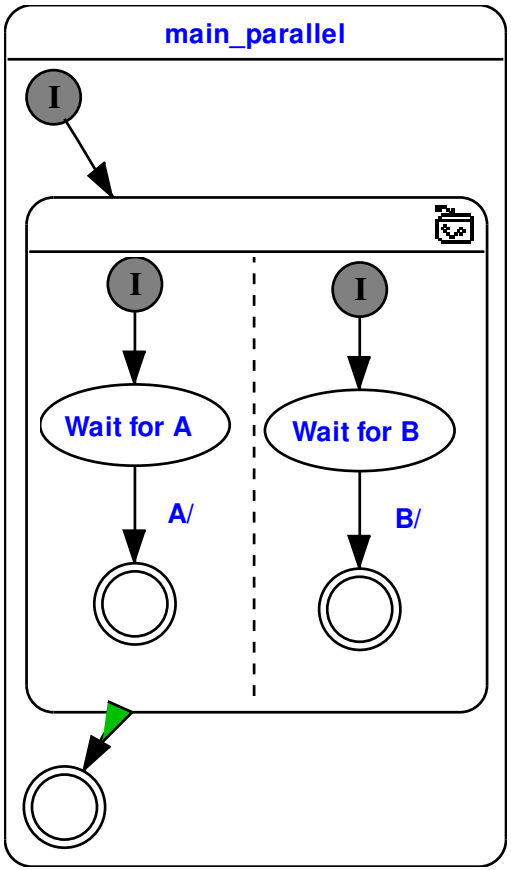


SyncCharts know four types of states:

1. Simple States: Carry just a label.
2. Graphic Macrostates: Encapsulates a hierarchy of other states, including further graphic states.
3. Textual Macrostates: Contain statements of the Esterel language. They are executed on entry of the state.
4. Run Modules: Include other modules.

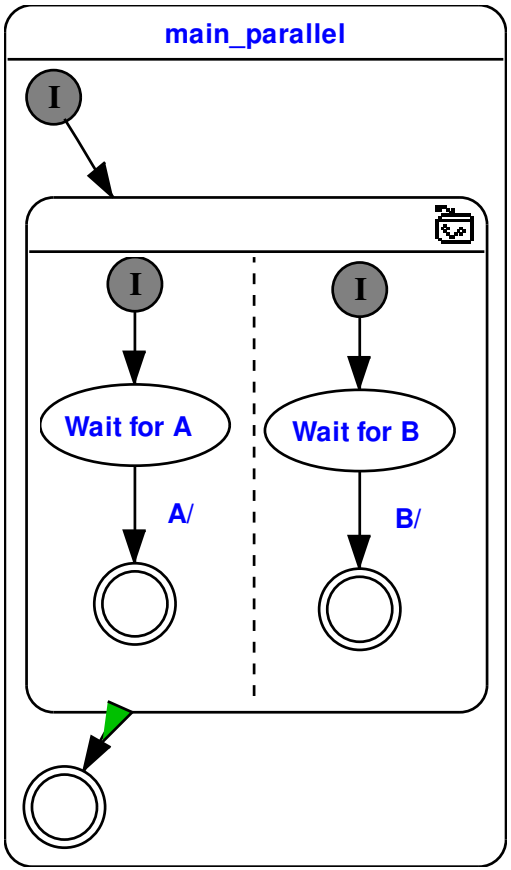
Transitions are **not** allowed to cross the boundaries of graphic macrostates. This is in contrast to other modelling tools.

Parallel States



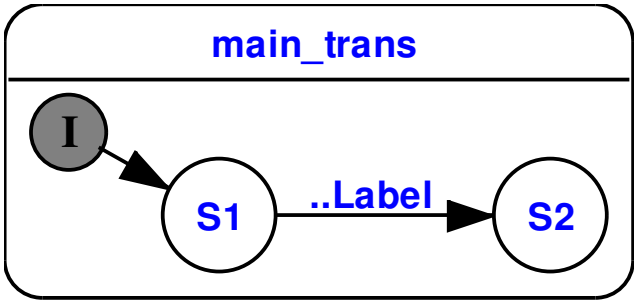
- Dashed lines (horizontal or vertical) separate parallel executed states inside a graphic macrostate.
- Each segment may be segmented into further parallel segments, but iterative segmentation does not introduce additional hierarchy. All parallel segments in a graphic macrostate are at the same level.

Parallel States



- A transition outside the graphic macrostate with normal termination is activated, when all parallel segments have reached their terminal state.
- If just one segment does not have one or if it is not reached, then the normal termination transition will never be activated.

Syntax of Transition Labels



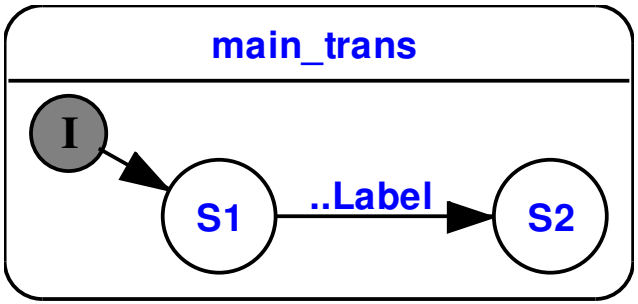
Informal syntax of a transition label between states S1 and S2, all elements are optional:

```
# factor trigger {condition} / effect
```

Basic activation and action:

- `trigger` is an expression of signal presence like "`A or B`"
- Enclosed in braces is the `condition`. It is a data expression over signal values or variables like "?A=42"
- Behind a single "/" follows the *effect*' as a list of emitted signals if the transition is executed. Multiple signal names are separated with ",".

Syntax of Transition Labels



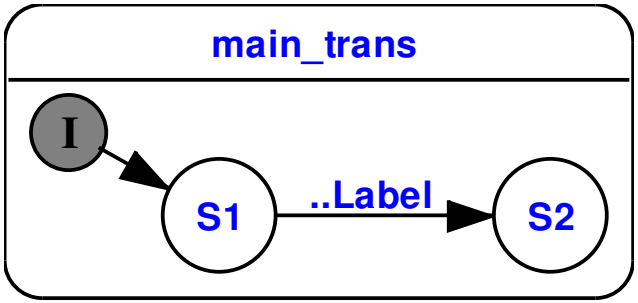
Informal syntax of a transition label between states S1 and S2, all elements are optional:

```
# factor trigger {condition} / effect
```

Extensions:

- "#" is the flag for an immediate transition
- "factor" is the (natural) number of ticks a transition must be active before it is executed. These active ticks does not need to be consecutive, but S1 must be active all the time.

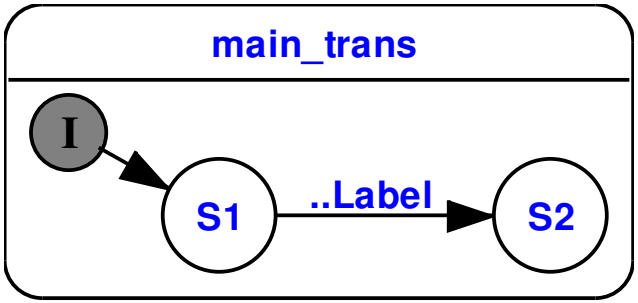
Transition Labels: Examples



The following label examples belong to the transition originating at S1 and leading to S2:

- A/B

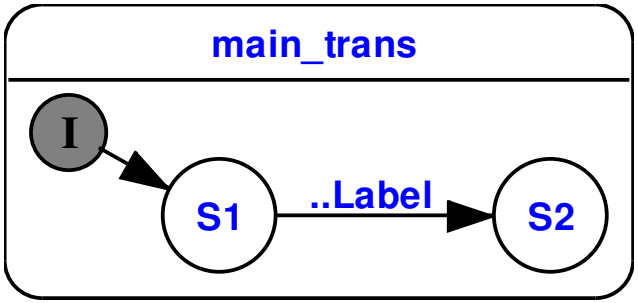
Transition Labels: Examples



The following label examples belong to the transition originating at S1 and leading to S2:

- `A/B`
  After entering S1 the signal `A` is tested from the next tick on. If `A` is present, then `B` is emitted in the same tick and state S2 is entered.
- `/B`

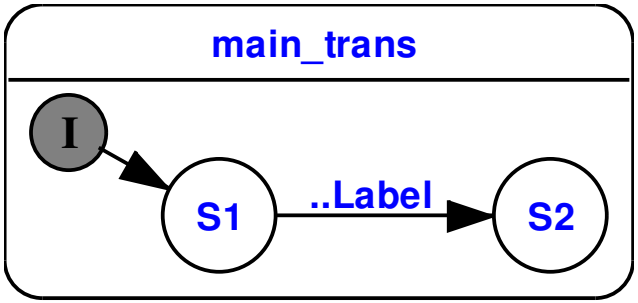
Transition Labels: Examples



The following label examples belong to the transition originating at S1 and leading to S2:

- `A/B`
  After entering `S1` the signal `A` is tested from the next tick on. If `A` is present, then `B` is emitted in the same tick and state `S2` is entered.
- `/B`
  After enabling `S1`, `B` is emitted in the next tick and `S2` is entered.
- `3 A/`

Transition Labels: Examples



The following label examples belong to the transition originating at S1 and leading to S2:

- `A/B`
  After entering `S1` the signal `A` is tested from the next tick on. If `A` is present, then `B` is emitted in the same tick and state `S2` is entered.
- `/B`
  After enabling `S1`, `B` is emitted in the next tick and `S2` is entered.
- `3 A/`
  The transition is executed, if `S1` is active consecutively and signal `A` is present for 3 times.

Transition Labels: Examples

- `#A/`
  If `S1` is entered, signal `A` is tested from the same tick on. If `A` is present in the tick `S1` is entered then state `S2` is entered in the same tick.
- `{?A=42}/`

Transition Labels: Examples

- `#A/`
  If `S1` is entered, signal `A` is tested from the same tick on. If `A` is present in the tick `S1` is entered then state `S2` is entered in the same tick.
- `{?A=42}/`
  The transition is executed, if the (valued) signal `A` carries the value `42`. `A` does not need to be present for this test.
- `A {?A=42}/`

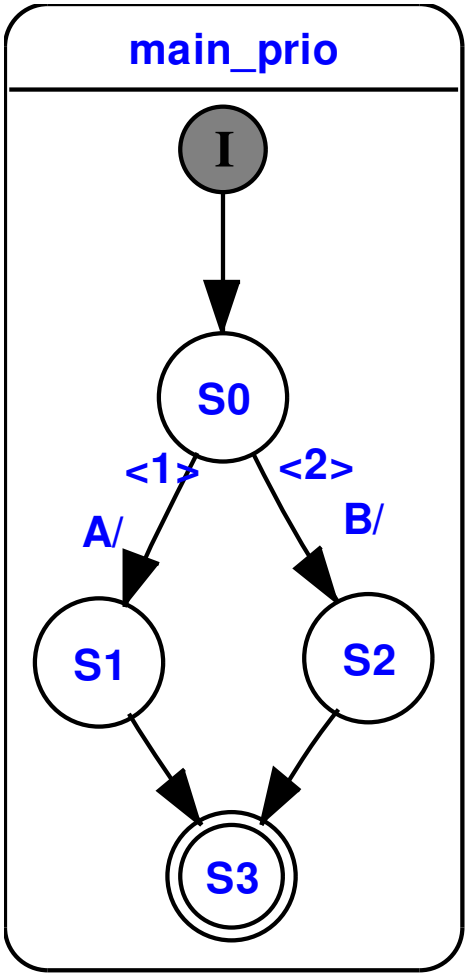
Transition Labels: Examples

- `#A/`
  If `S1` is entered, signal `A` is tested from the same tick on. If `A` is present in the tick `S1` is entered then state `S2` is entered in the same tick.
- `{?A=42}/`
  The transition is executed, if the (valued) signal `A` carries the value `42`. `A` does not need to be present for this test.
- `A {?A=42}/`
  This test succeeds if `A` is present and carries the value `42`.
- `A and (B or C)/`
  Logical combination of signal presence.
- `{?A=10 and (?B<3 or ?C=1)}/`

Transition Labels: Examples

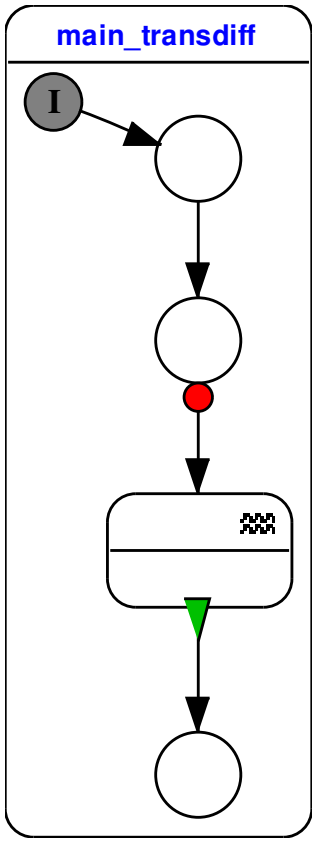
- `#A/`
  If `S1` is entered, signal `A` is tested from the same tick on. If `A` is present in the tick `S1` is entered then state `S2` is entered in the same tick.
- `{?A=42}/`
  The transition is executed, if the (valued) signal `A` carries the value `42`. `A` does not need to be present for this test.
- `A {?A=42}/`
  This test succeeds if `A` is present and carries the value `42`.
- `A and (B or C)/`
  Logical combination of signal presence.
- `{?A=10 and (?B<3 or ?C=1)}/`
  Logical combination of value tests.
- `/A(2), B(4)`
  Emission of multiple valued signals.

Transition Priorities



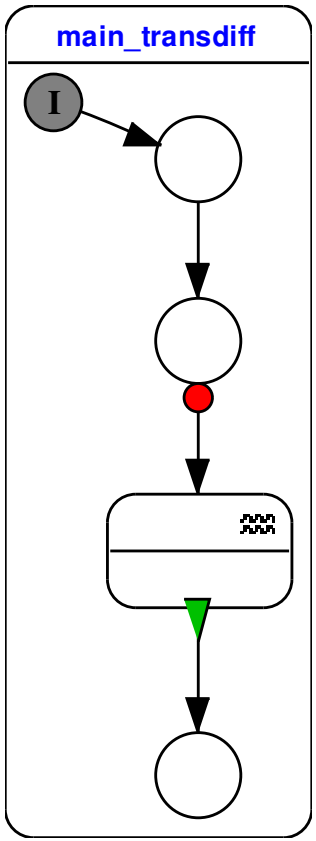
- When more than one transition departs a state, an automatic (but editable) priority ordering is established.
- The transition labels are evaluated according to their priority.
- The first label that succeeds activates its transition.
- Low numbers mean high priority.

SyncCharts feature four different types of transitions: They are differentiated by a symbol at the arrow root:



1. Initial connector: Initial arc
   Initial arcs connect the initial connectors of the chart with the other states.
2. No symbol: Weak abort
   When the trigger/condition of the transition is enabled, then the actions of the originating state in the current tick are executed for a last time, then the transition action, and the entry action of the new state.
   In other words:
   The old state can “express it’s last will”.

Transition Types

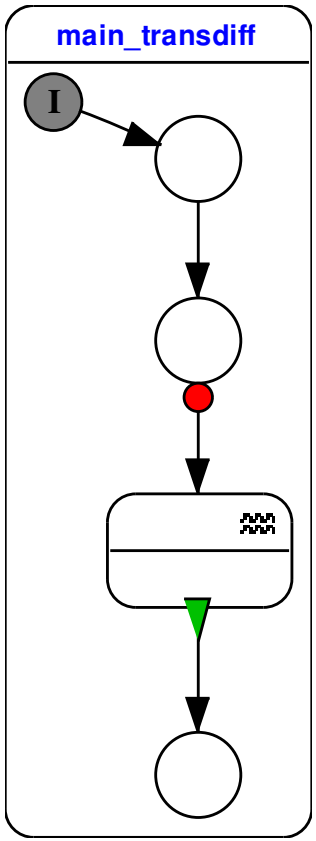


3. Red bullet: Strong abort
   The action for the current tick of the old state is not executed. Only the transition action and the entry action of the new state is executed.
4. Green triangle: Normal termination
   This transition can be used to exit macro states. It is activated when the macro state terminates.

All these transition types must not be confused with "immediate" or "delayed" evaluation of the transition label (label prefix "#").

Transition Types and Labels

Some transition types have restrictions on their labels:



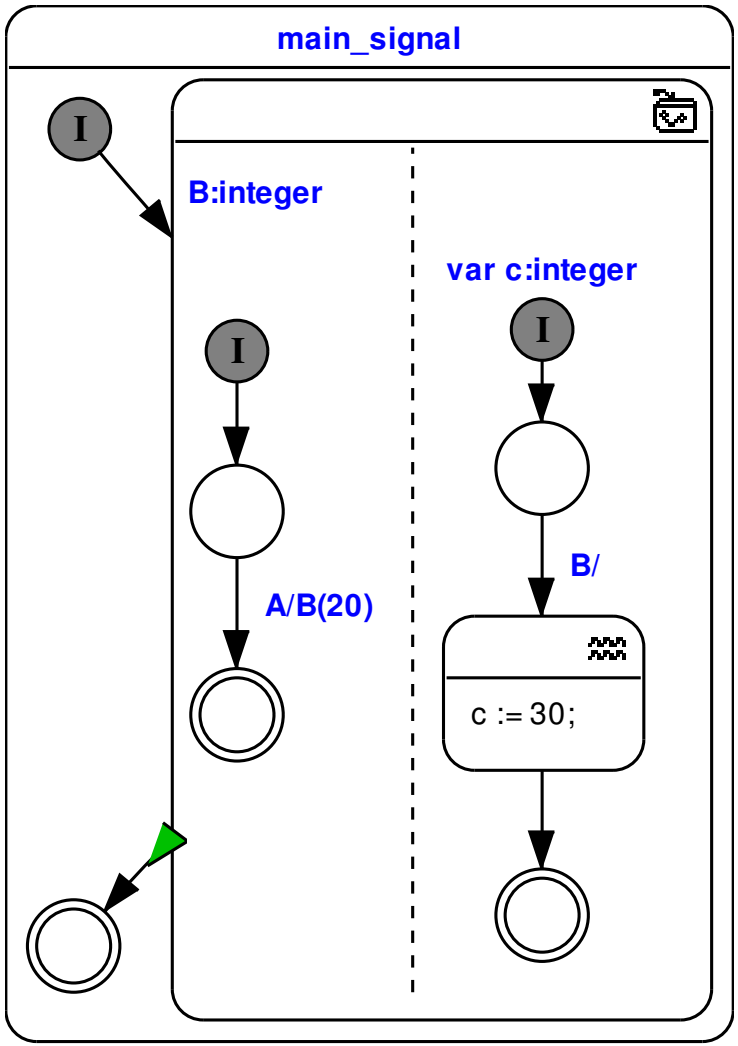
- Initial arc:
  These are always "immediate," therefore the additional flag "#" is not needed.
- Weak abort: No restrictions.
- Strong abort: No restrictions.
- Normal termination:
  They support no triggers or conditions because they are activated by the termination of the originating state. The immediate flag is not used either.

Transition Types and Priorities

The type of a transition interacts with it's priority:

- Strong abort: Highest priority
- Weak abort: Middle priority
- Normal termination: Lowest priority

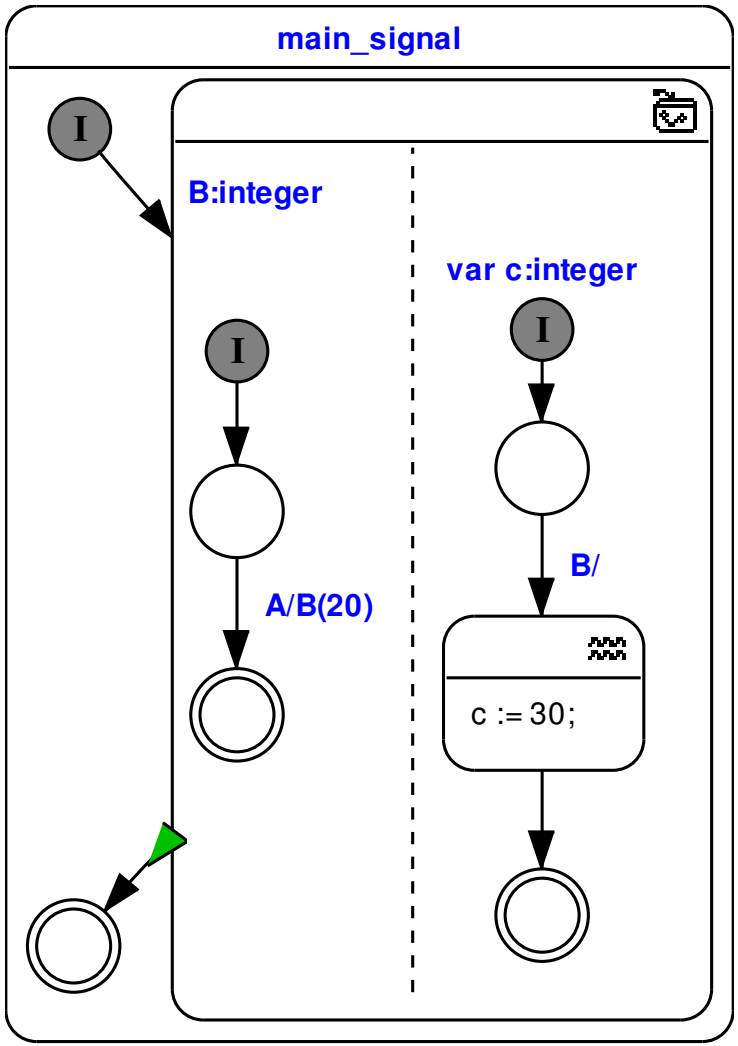
Local Signals and Variables



Local signals

- Defined in the body of a graphical macrostate
- Shared between parallel threads

Local Signals and Variables



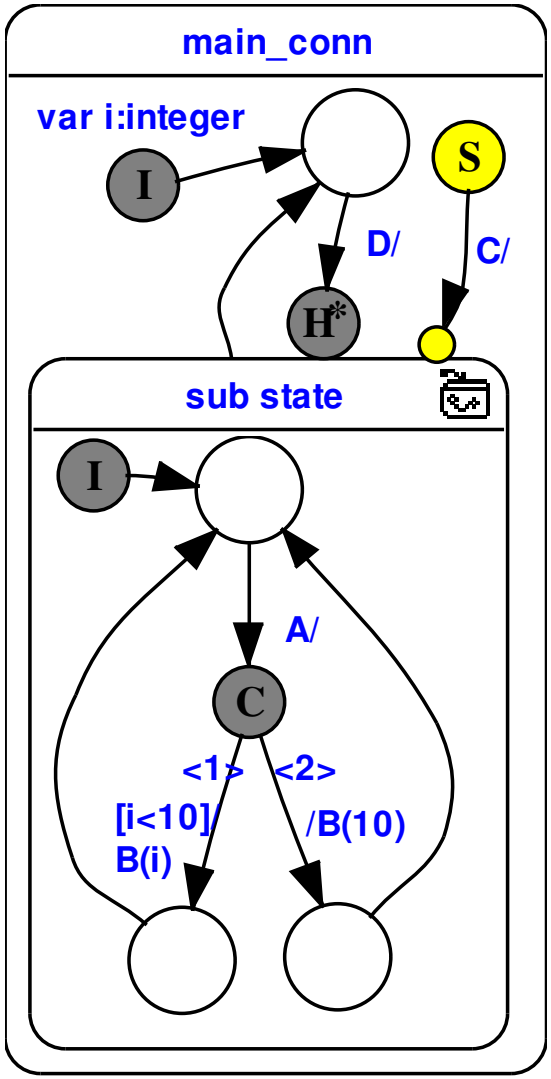
Local signals

- Defined in the body of a graphical macrostate
- Shared between parallel threads

Local variables

- Not shared

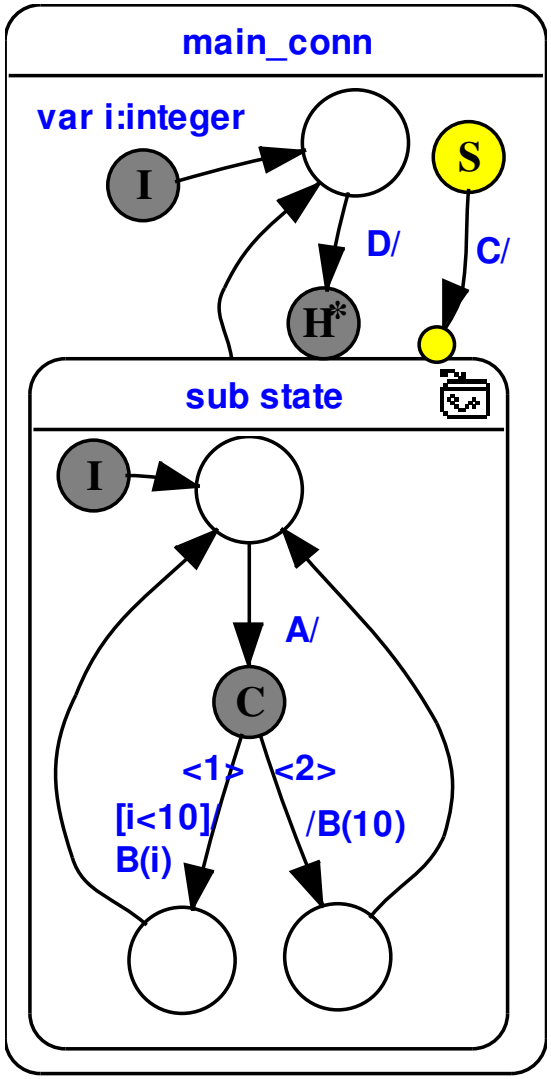
This (artificial) SyncChart demonstrates all four connector states:



Initial connector

- Activated at activation of the macrostate
- Only departing transitions permitted
- All connected transitions are “immediate”

This (artificial) SyncChart demonstrates all four connector states:



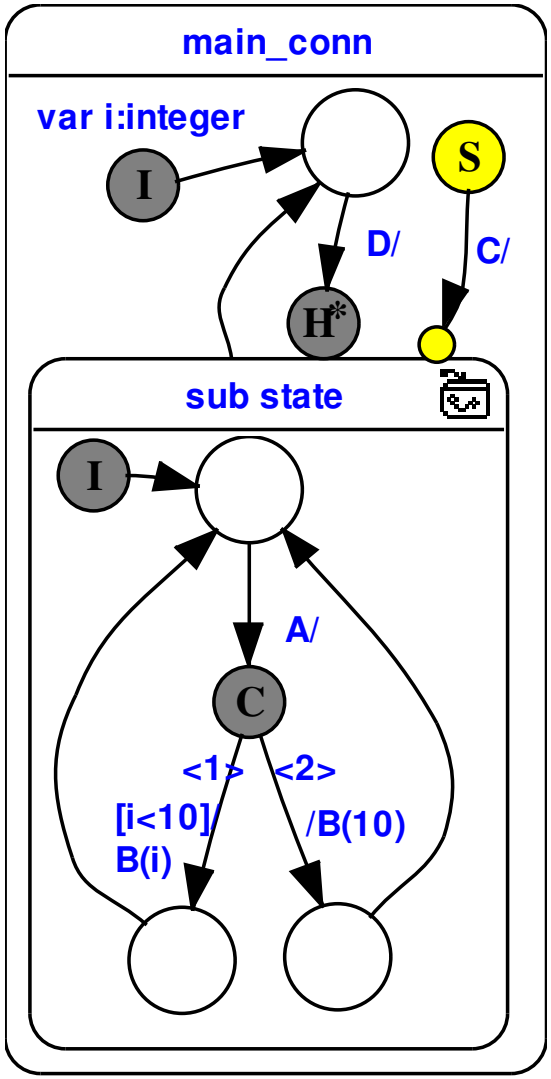
Initial connector

- Activated at activation of the macrostate
- Only departing transitions permitted
- All connected transitions are "immediate"

Conditional connector

- All departing transitions are "immediate"
- One departing "default" transition without condition must be present.

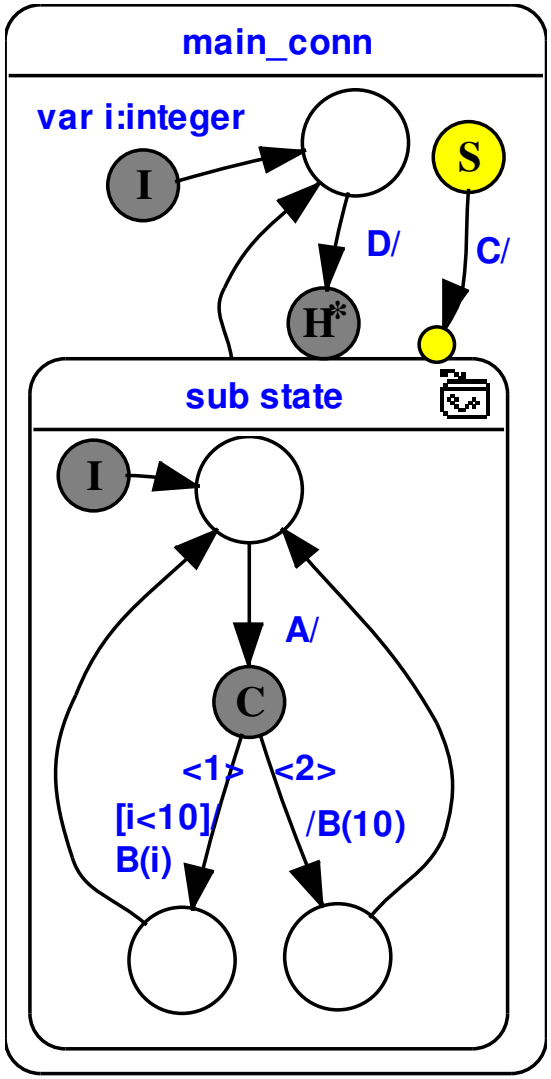
This (artificial) SyncChart demonstrates all four connector states:



Suspend connector

- The suspend state is always active.
- Only one departing transitions is permitted.
- The transition can only hold a trigger expression.
- The "immediate" flag can be enabled on demand.
- When the transition is activated, then the target state is (strongly) suspended.

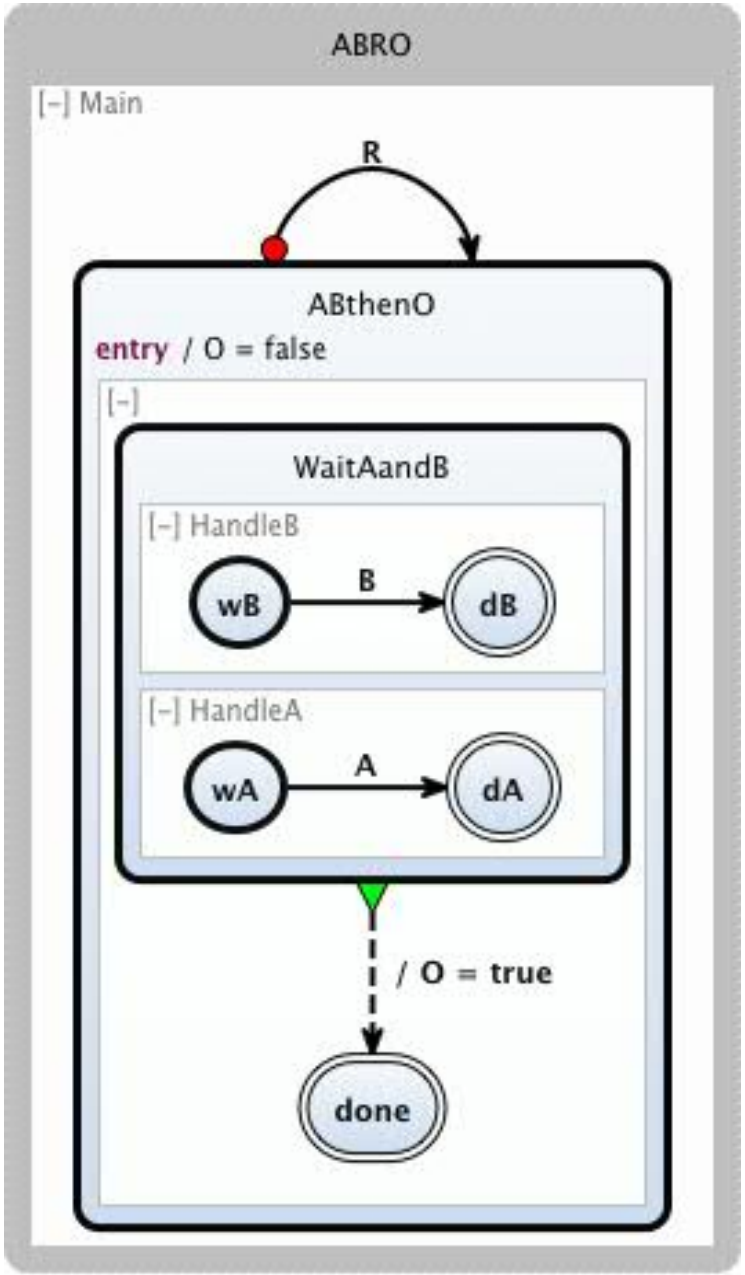
This (artificial) SyncChart demonstrates all four connector states:



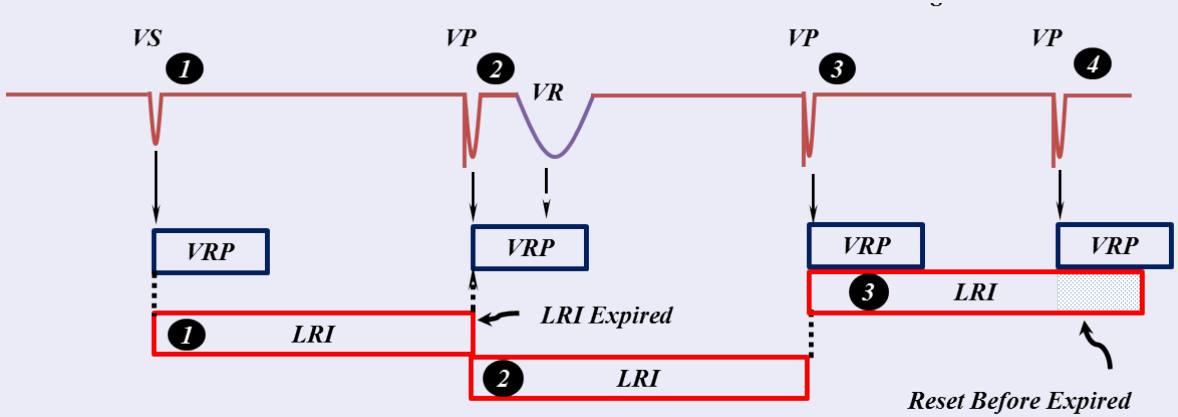
History connector

- This connector is directly attached to macrostates
- Only incoming transitions can connect.
- The previous state of the macrostate is restored when it is entered through a history connector.

ABRO in SCCharts [5]



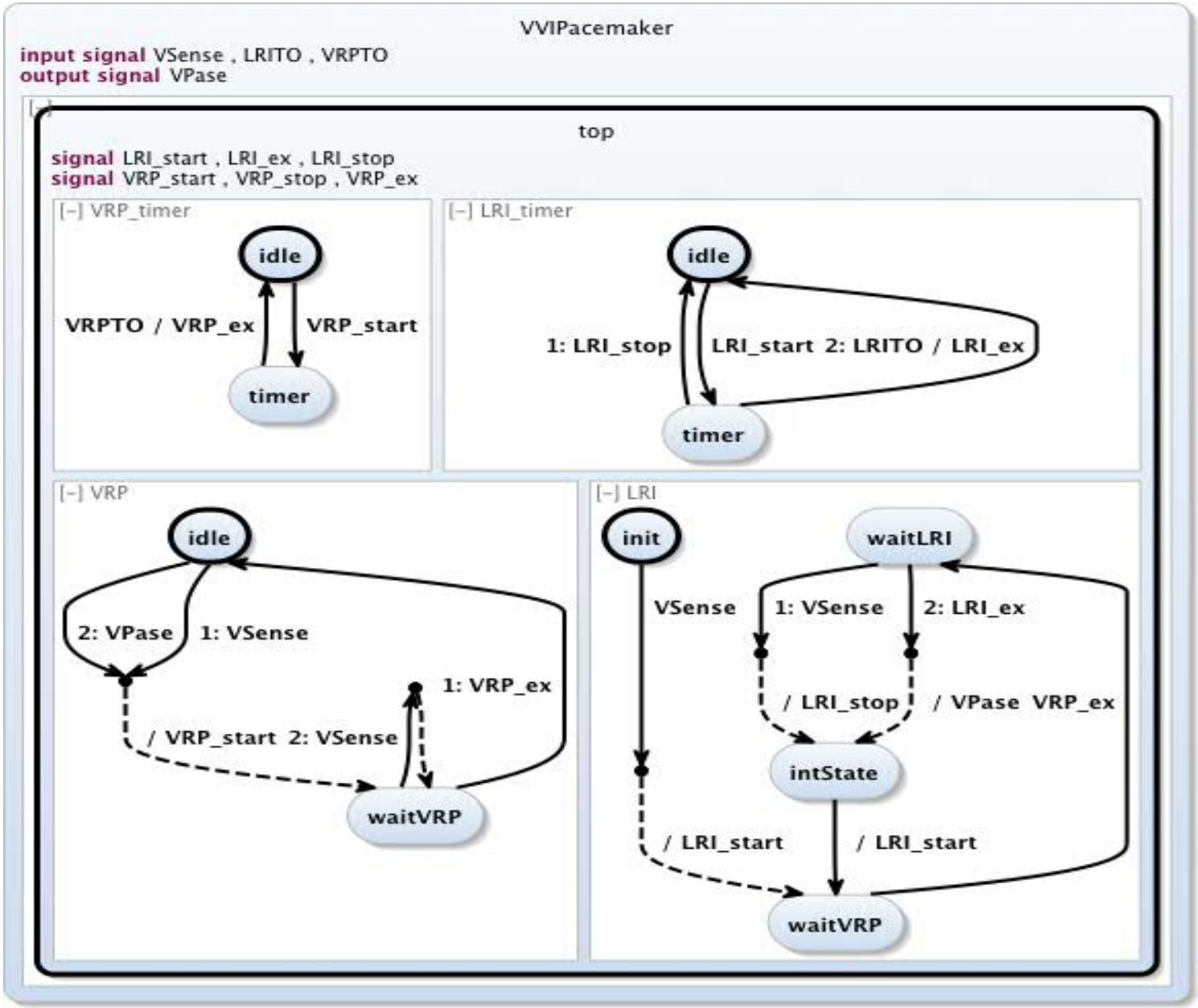
Pacemaker – VVI Mode in SCCharts



[7].

- VVI - *ventricular* sensed, *ventricular* paced with *inhibited* response to sensing.
- The pacemaker monitors the ventricular activity and ensures two things: 1) that once a ventricular sense event has happened, there is no more sensed events within the *ventricular refractory period* (VRP), and 2) that the maximum time separation between two ventricular events is at most the *lower rate interval* (LRI). In the event of a violation, the pacemaker paces the ventricle.

Pacemaker – VVI Mode in SCCharts



Conclusions

1. The synchronous approach is widely used while designing safety-critical systems in aerospace and automotive domains.
2. The synchronous approach is based on the *synchrony hypothesis*.
3. This hypothesis assumes that the *reactive system operates infinitely fast relative to its environment*.
4. We discussed two synchronous languages: Esterel (imperative style) and SCCharts / SyncCharts (graphical style).

Conclusions..

1. We also examined modelling of a pacemaker using the synchronous approach.
2. This approach will be the basis of our work on bio-remulation.
3. We will elaborate in future lectures on how this approach can be used for modelling hybrid systems, such as the human heart.
4. I have based the Esterel part of this lecture on [6] and the SyncCharts part has some slides which I thankfully acknowledge Prof. Reinhard von Hanxleden from Kiel University, Germany.

Next lecture

1. Temporal logic and model checking
2. Validation of timed systems using Uppaal
3. Pacemaker validation: VVI and DDD modes.


Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone.
The Synchronous Languages 12 Years Later.
Proceedings of the IEEE, 91(1):64–83, January 2003.


Gérard Berry.
The Esterel v5 Language Primer, Version v5_91.
Centre de Mathématiques Appliquées, Ecole des Mines, Sophia-Antipolis, July 2000.


Charles Antony Richard Hoare.
Communicating Sequential Processes.
Prentice Hall International, New Jersey, 1985.


Robin Milner.
Communication and concurrency, volume 84.
Prentice hall New York etc., 1989.


Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien.

SCCharts: Sequentially Constructive Statecharts for safety-critical applications.
In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.

Li Hsien Yoong, Partha S Roop, Zeeshan E Bhatti, and Matthew MY Kuo.
Introduction to synchronous programming using esterel.
In *Model-Driven Design Using IEC 61499*, pages 35–64. Springer, 2015.

Yu Zhao and Partha S Roop.
Model-driven design of cardiac pacemaker using IEC 61499 function blocks.
Distributed Control Applications: Guidelines, Design Patterns, and Application Examples with the IEC 61499, 9:335, 2016.