Synchronous Languages—Lecture 10

Prof. Dr. Reinhard von Hanxleden

Christian-Albrechts Universität Kiel
Department of Computer Science
Real-Time Systems and Embedded Systems Group

4 Dec. 2018

Last compiled: January 29, 2019, 10:53 hrs

SyncCharts

The 5-Minute Review Session

1. Why does constructiveness analysis matter in circuit design?

The 5-Minute Review Session

1. Why does constructiveness analysis matter in circuit design?
2. How do cyclic and acyclic circuits differ?

The 5-Minute Review Session

1. Why does constructiveness analysis matter in circuit design?
2. How do cyclic and acyclic circuits differ?
3. How do we analyze constructiveness of a cyclic circuit?

The 5-Minute Review Session

1. Why does constructiveness analysis matter in circuit design?
2. How do cyclic and acyclic circuits differ?
3. How do we analyze constructiveness of a cyclic circuit?
4. How do we extend Boolean functions to three-valued logic?

The 5-Minute Review Session

1. Why does constructiveness analysis matter in circuit design?
2. How do cyclic and acyclic circuits differ?
3. How do we analyze constructiveness of a cyclic circuit?
4. How do we extend Boolean functions to three-valued logic?
5. Are TVFs monotonic? What does that mean?

The 5-Minute Review Session

1. In constructiveness analysis, how can we minimize the number of gate evaluations?

The 5-Minute Review Session

1. In constructiveness analysis, how can we minimize the number of gate evaluations?
2. How do the computational complexities of *Malik* vs. *Shiple* compare?

The 5-Minute Review Session

1. In constructiveness analysis, how can we minimize the number of gate evaluations?
2. How do the computational complexities of *Malik* vs. *Shiple* compare?
3. How do we extend the constructiveness analysis to circuits with *latches*?

The 5-Minute Review Session

1. In constructiveness analysis, how can we minimize the number of gate evaluations?
2. How do the computational complexities of *Malik* vs. *Shiple* compare?
3. How do we extend the constructiveness analysis to circuits with *latches*?
4. What is the computational problem class of the constructiveness analysis problem?
5. What are *Core SCCharts*?

Overview

Statecharts

Statecharts proposed by David Harel [1987]

Statecharts

Statecharts proposed by David Harel [1987]

In a nutshell: Statecharts = Mealy Machines + hierarchy (depth) + orthogonality + broadcast + data

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

Harel-Statecharts vs. SyncCharts—Similarities

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

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

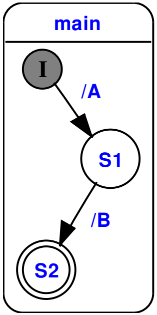
Harel-Statecharts vs. SyncCharts—Differences

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

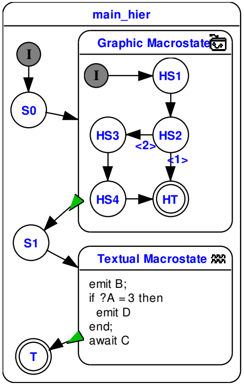
SyncChart:



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

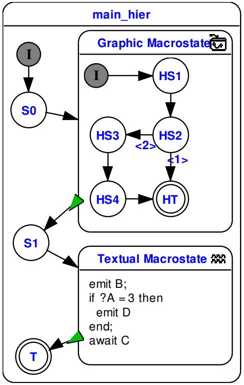


```
emit B;
if ?A = 3 then
  emit D
end;
await C
```

SyncCharts know four types of states:

1. Simple States: Carry just a label.

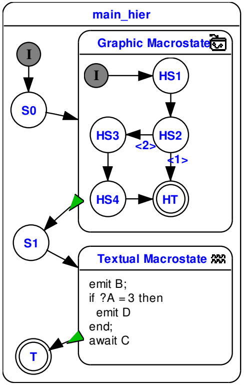
Hierarchic States



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.

Hierarchic States

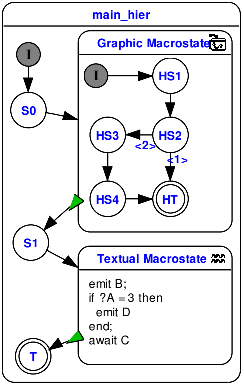


```
emit B;
if ?A = 3 then
  emit D
end;
await C
```

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.

Hierarchic States



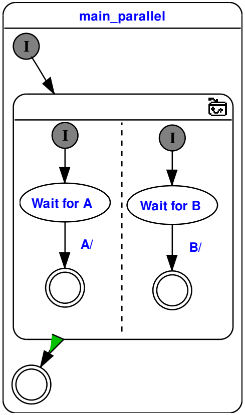
```
emit B;
if ?A = 3 then
  emit D
end;
await C
```

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.

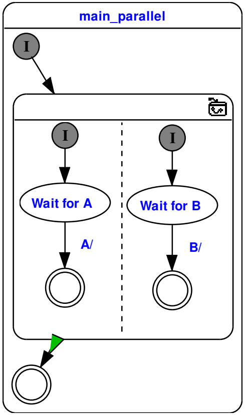
Transitions are **not** allowed to cross the boundaries of graphic macrostates. This is in contrast to other modelling tools.

Parallel States



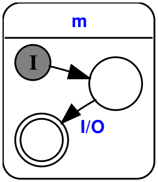
- Dashed lines (horizontal or vertical) separate parallel executed states inside a graphic macrostate.
- Each segment may be segmented into further parallel segments, but iterative segmentation does not introduce additional hierarchy. All parallel segments in a graphic macrostate are at the same level.

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.

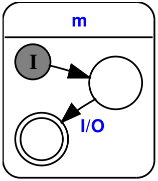
Modules



A module like this with an interface:

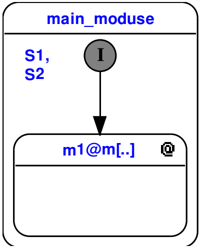
```
input  I;
output O;
```

Modules



A module like this with an interface:

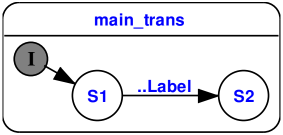
```
input  I;
output O;
```



. . . can be used as a Run Module with these signal bindings:

```
  signal S1 / I;
  signal S2 / O
```

Syntax of Transition Labels



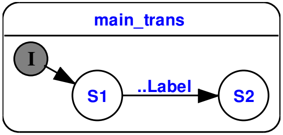
Informal syntax of a transition label between states S1 and S2, all elements are optional:

```
# 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



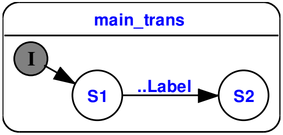
Informal syntax of a transition label between states S1 and S2, all elements are optional:

```
# 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.

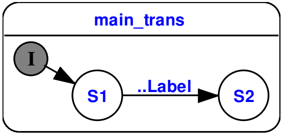
Transition Labels: Examples



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

- `A/B`

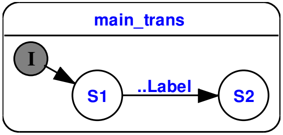
Transition Labels: Examples



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.

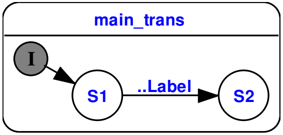
Transition Labels: Examples



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

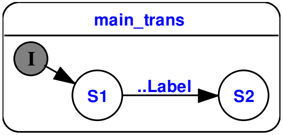
Transition Labels: Examples



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.

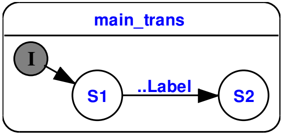
Transition Labels: Examples



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.

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.

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.

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.

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)}/`

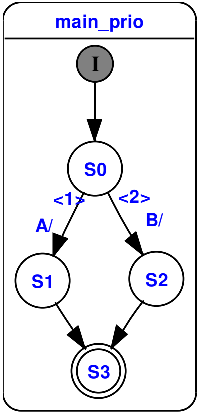
Transition Labels: Examples

- `#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.

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.

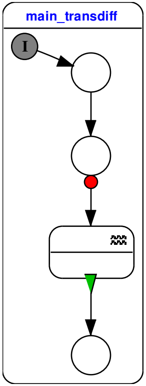
Transition Priorities



- 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.

Transition Types

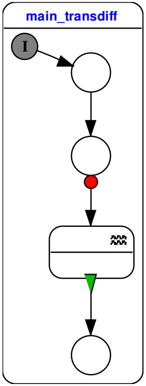
SyncCharts feature four different types of transitions: They are differentiated by a symbol at the arrow root:



1. Initial connector: Initial arc
   Initial arcs connect the initial connectors of the chart with the other states.

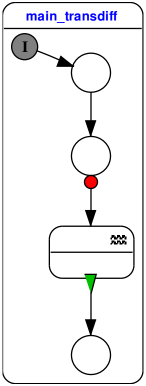
Transition Types

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



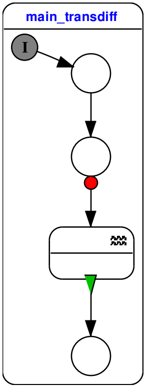
1. Initial connector: Initial arc
   Initial arcs connect the initial connectors of the chart with the other states.
2. No symbol: Weak abort
   When the trigger/condition of the transition is enabled, then the actions of the originating state in the current tick are executed for a last time, then the 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.

Transition Types

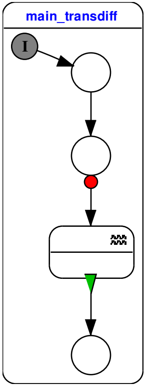


3. Red bullet: Strong abort
   The action for the current tick of the old state is not executed. Only the transition action and the entry action of the new state is executed.
4. Green triangle: Normal termination
   This transition can be used to exit macro states. It is activated when the macro state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



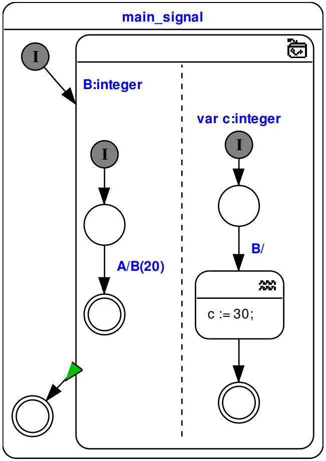
- Initial arc:
  These are always "immediate," therefore the additional flag "#" is not needed.
- Weak abort: No restrictions.
- Strong abort: No restrictions.
- Normal termination:
  They support no triggers or conditions because they are activated by the termination of the originating state. The immediate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

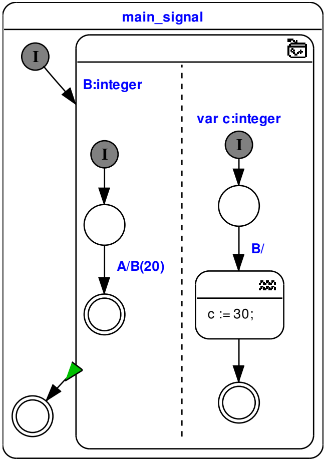
Local Signals and Variables



Local signals

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

Local Signals and Variables



Local signals

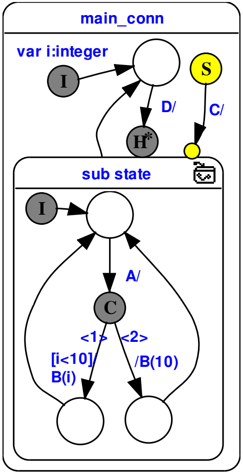
- Defined in the body of a graphical macrostate
- Shared between parallel threads

Local variables

- Not shared

Connectors

This (artificial) SyncChart demonstrates all four connector states:

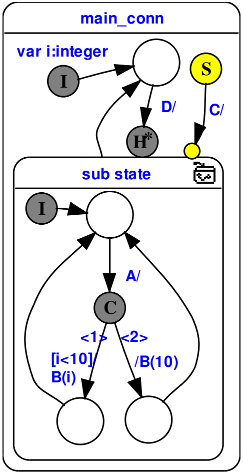


I Initial connector

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

Connectors

This (artificial) SyncChart demonstrates all four connector states:



I Initial connector

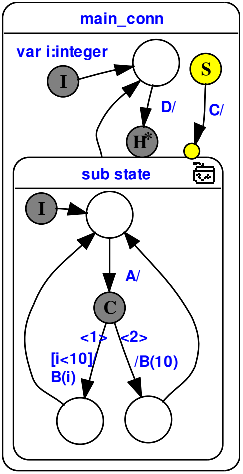
- Activated at activation of the macrostate
- Only departing transitions permitted
- All connected transitions are "immediate"

C Conditional connector

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

Connectors

This (artificial) SyncChart demonstrates all four connector states:

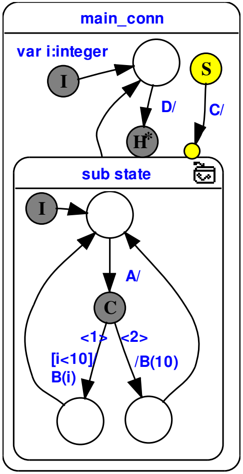


S Suspend connector

- 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.

Connectors

This (artificial) SyncChart demonstrates all four connector states:



H* 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.

Equivalence of SyncCharts and Esterel

- Esterel and SyncCharts look different
- However, underlying model of computation/semantics are equivalent
- Both are based on synchrony hypothesis
- Can translate one into the other

Overview

Motivation

- Can transform Esterel projects into SyncCharts
- Better visualization of the behavior of Esterel projects
- Combine benefits of textual editing and graphical viewing (see KIELER project)
- Didactical purposes

From Esterel to SyncCharts

Step 1: Transform Esterel program into SyncChart with textual macro states

From Esterel to SyncCharts

Step 1: Transform Esterel program into SyncChart with textual macro states

Step 2: Iteratively apply reduction rules to transform Esterel constructs into graphical components

From Esterel to SyncCharts

Step 1: Transform Esterel program into SyncChart with textual macro states

Step 2: Iteratively apply reduction rules to transform Esterel constructs into graphical components

Step 3: Optimize SyncChart

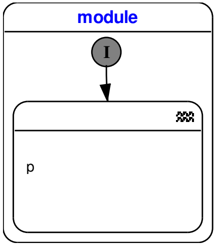
Step 1: Transform Esterel to SyncChart

```
module mod_name:
input $I_1, \ldots, I_n$;
output $O_1, \ldots, O_m$;
$p$
end module
```

Step 1: Transform Esterel to SyncChart

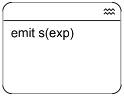
```
module mod_name:
input $I_1, \ldots, I_n$;
output $O_1, \ldots, O_m$;
$p$
end module
```

$\Longrightarrow$

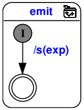


Step 2: Reduce to Fully Graphical SyncChart

Signal emission

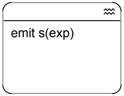


$\Longrightarrow$

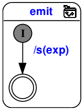


Step 2: Reduce to Fully Graphical SyncChart

Signal emission



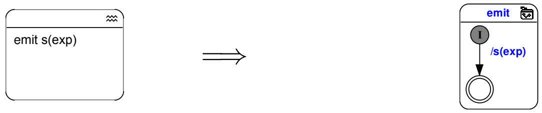
$\Longrightarrow$



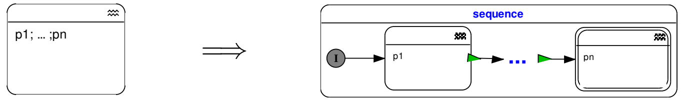
Sequence

Step 2: Reduce to Fully Graphical SyncChart

Signal emission

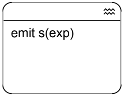


Sequence

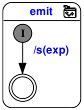


Step 2: Reduce to Fully Graphical SyncChart

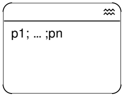
Signal emission



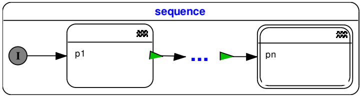
$\Longrightarrow$



Sequence



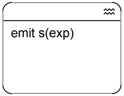
$\Longrightarrow$



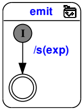
Signal awaiting

Step 2: Reduce to Fully Graphical SyncChart

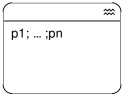
Signal emission



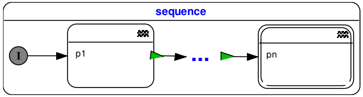
$\Longrightarrow$



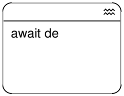
Sequence



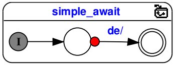
$\Longrightarrow$



Signal awaiting

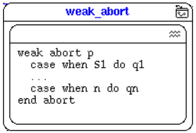


$\Longrightarrow$



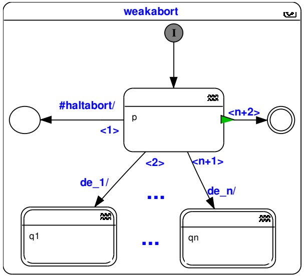
Step 2: Reduce to Fully Graphical SyncChart

Weak Abortion



```
weak abort p
  case when S1 do q1
  ...
  case when n do qn
end abort
```

$\Longrightarrow$



+ 19 additional rules
Translation of traps not trivial—see [Prochnow et al. 2006]

Example: ABRO

Applying Rule (module)

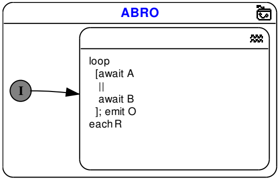
```
module ABRO:
input A, B, R;
output O;
loop
  [
    await A
  ||
    await B
  ];
  emit O;
each R
end module
```

Example: ABRO

Applying Rule (module)

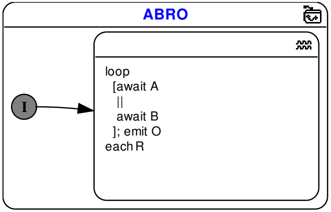
```
module ABRO:
input A, B, R;
output O;
loop
  [
    await A
  ||
    await B
  ];
  emit O;
each R
end module
```

$\Longrightarrow$



Example: ABRO

Applying Rule (loopeach)

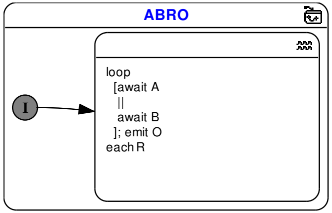


```
loop
  [await A
   ||
   await B
  ]; emit O
each R
```

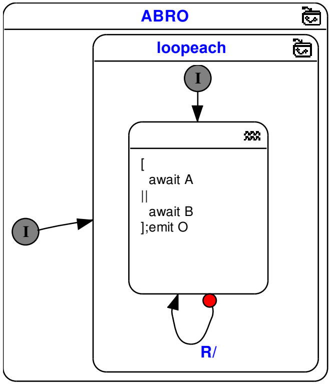
$\Longrightarrow$

Example: ABRO

Applying Rule (loopeach)

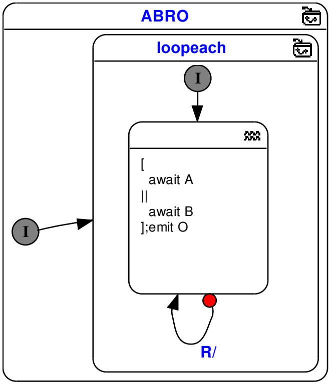


⟹



Example: ABRO

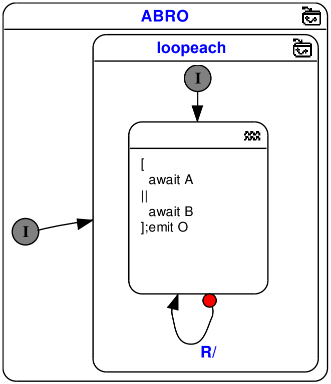
Applying Rule (sequence)



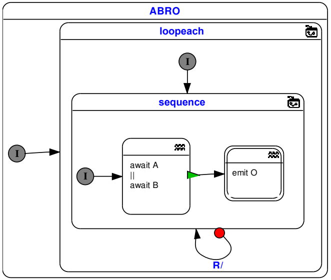
$\Longrightarrow$

Example: ABRO

Applying Rule (sequence)

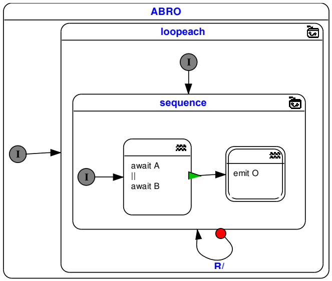


$\Longrightarrow$



Example: ABRO

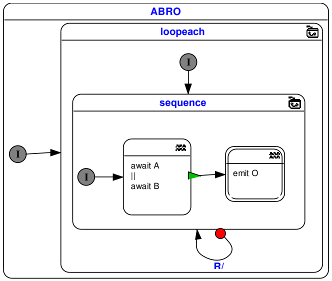
Applying Rules (parallel) + (emit)



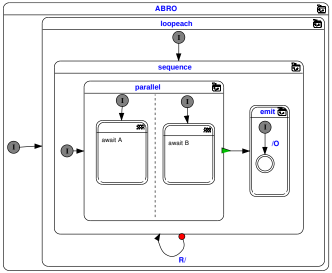
$\Longrightarrow$

Example: ABRO

Applying Rules (parallel) + (emit)

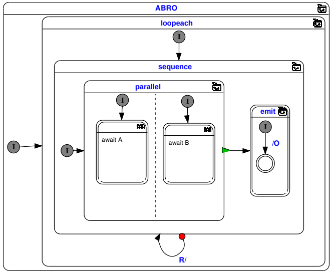


$\Longrightarrow$



Example: ABRO

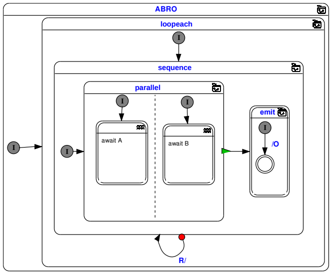
Applying Rule (simple await)



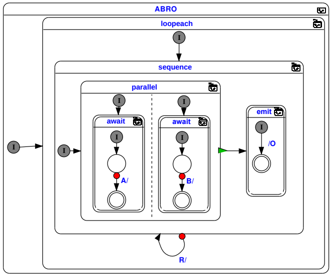
$\Longrightarrow$

Example: ABRO

Applying Rule (simple await)



$\Longrightarrow$



Step 3: Optimizations

Motivation

- Automatic synthesis produces “verbose” modules
- However, also human modelers (esp. novices) may produce sub-optimal models

Step 3: Optimizations

Motivation

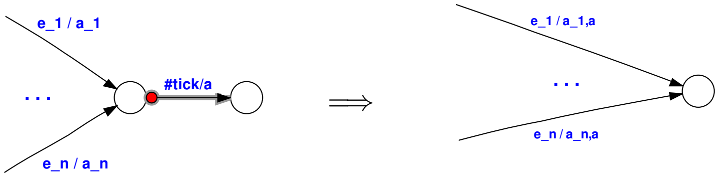
- Automatic synthesis produces “verbose” modules
- However, also human modelers (esp. novices) may produce sub-optimal models

Notes:

- It may be a matter of style/opinion what “optimal” means
- However, consistency in style is desirable in any case—and standardized optimization rules help to achieve this

Step 3: Optimizations

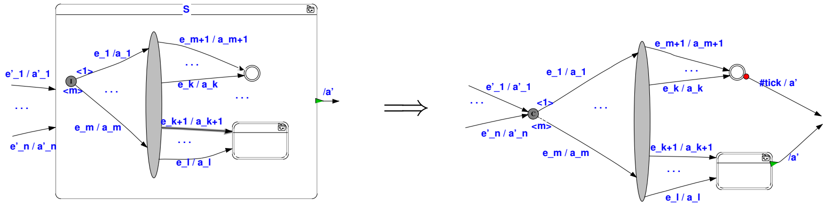
Removing Simple States



Precondition: State must be transient

Step 3: Optimizations

Flattening Hierarchy



Preconditions:

- no abort originates from S
- S has no local signals

+ further rules to remove conditionals, combine terminal states, remove normal terminations

Overview

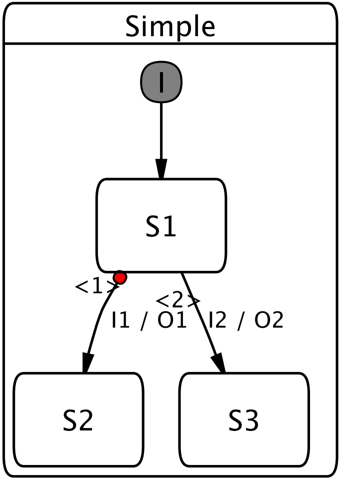
From SyncCharts to Esterel
Motivation

- Intermediate Step from SyncChart to C-Code (or VHDL, ...)
- This is what Esterel Studio does
- This translation is one possibility to define the semantics of SyncCharts.
- The following description of the translation is based on [André 1996] and the synthesis actually done by Esterel Studio (excluding the optimizations done by Esterel Studio)

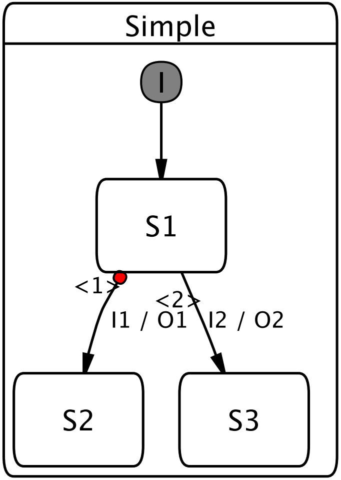
Basic Idea

- Structural Induction on SyncChart
- Simple-States become `await`
- Macro-States become `abort`
- Parallel stays parallel
- Problem: How to order states?

Simple Await

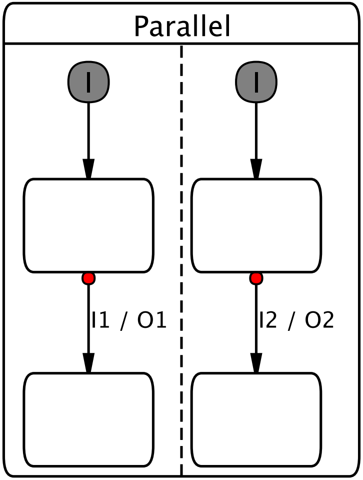


Simple Await

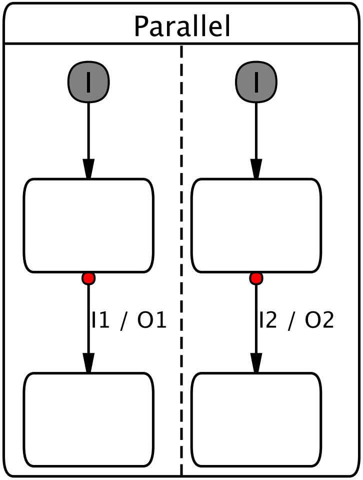


```
nothing;
% state S1
await
case (I1) do
  emit O1;
  % state S2
  halt
case (I2) do
  emit O2;
  % state S3
  halt
end await
```

Parallel

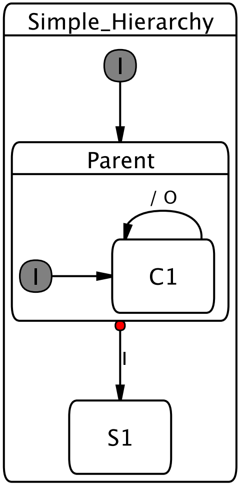


Parallel

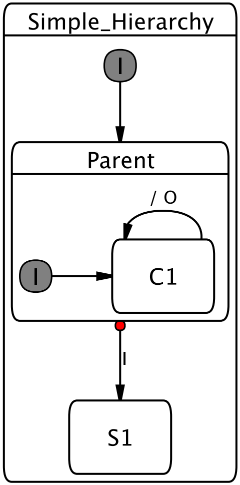


```
nothing;
await case (I1) do
  emit O1;
  halt
end await
||
nothing;
await case (I2) do
  emit O2;
  halt
end await
```

Simple Hierarchy

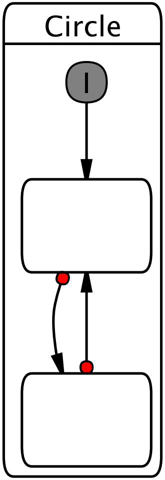


Simple Hierarchy

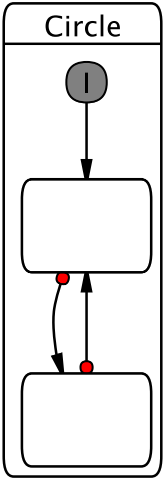


```
nothing;
% state Parent
abort
  nothing;
  loop
    % state C1
    await case (tick) do
      emit O
    end await
  end loop
when (I)
do
   nothing;
   % state S1
   halt
end abort
```

Simple Circle

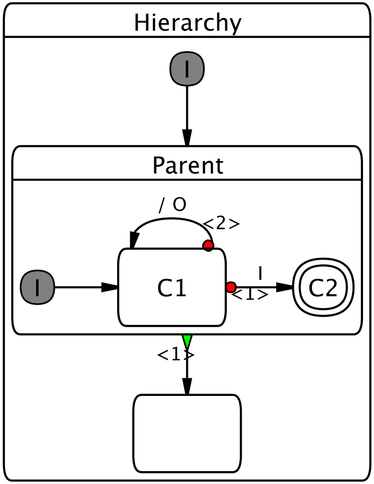


Simple Circle

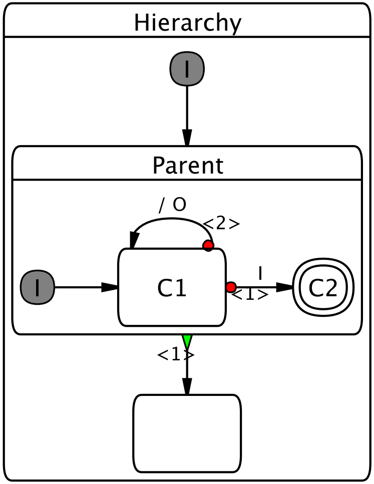


```
module Circle:
input I1, I2;
output O1, O2;
nothing;
loop
  await case (tick) do
  nothing;
  await case (tick) do
  nothing
  end await
  end await
end loop
end module
```

Not–so–simple Hierarchy

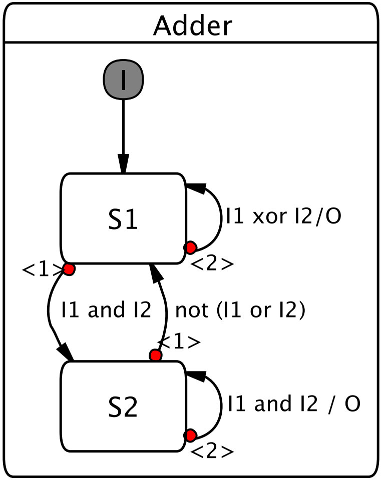


Not–so–simple Hierarchy

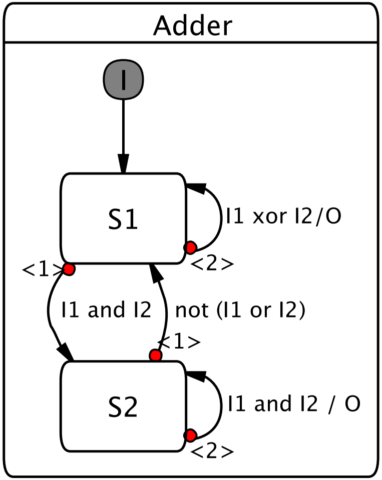


```
nothing;
% state Parent
trap sc_end_automaton1 in
 nothing;
 loop
  % state C1
  await
  case (I) do
   nothing;
   % state C2
   exit sc_end_automaton1
  case (tick) do
   emit O
  end await
 end loop
end trap ;
nothing;
halt
```

Not–so–simple Cycle



Not–so–simple Cycle



```
signal sc_go_35_S2 in
 loop
  if (sc_go_35_S2) then
   await % state S2
   case (not (I1 or I2)) do
    nothing
   case (I1 and I2) do
    emit O; emit sc_go_35_S2
   end await
  else
   await % state S1
   case (I1 and I2) do
    emit sc_go_35_S2
   case (I1 xor I2) do
    emit O
   end await
  end if
 end loop
end signal
```

Components

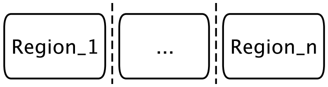
The translation is performed on a variant of the And-/Or-Tree

Reactive Cell	▶ State + Outgoing transitions ▶ State can contain macrostate
Region	▶ Multiple connected Reactive Cells
Macrostate	▶ Nonempty set of Regions

We have to introduce new signals and traps

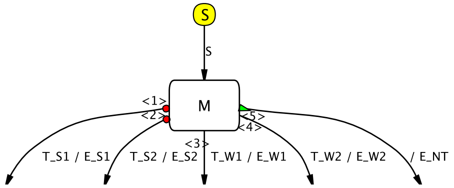
- trap `end_automaton` for normal termination
- signal `goto_s` for each state `s` in a region

Macrostate



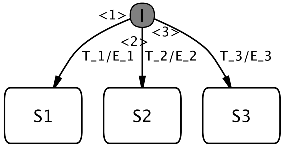
```
[
  run Region_1
||
  ...
||
  run Region_n
]
```

Reactive Cell



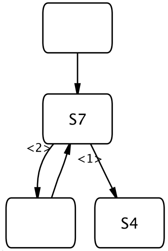
```
module Cell
output trans_1 ... trans_5
abort
  weak abort
    suspend
      run M;
      emit E_NT;
      emit trans_5;
    when S;
  case T_W1 do emit E_W1;
            emit trans_3;
  case T_W2 do emit E_W2;
            emit trans_4;
  end abort
case T_S1 do emit E_S1;
          emit trans_1;
case T_S2 do emit E_S2;
          emit trans_2;
end abort
```

Region Initialization



```
present
  case T_1 then emit E_1;
                emit goto_1;
  case T_2 then emit E_2;
                emit goto_2;
  case T_3 then emit E_3;
                emit goto_3;
end present
```

Region Continuation



```
signal goto_S1, ..., goto_Sn in
trap end_automaton
  loop
    trap L in
      ...
      present goto_S7 then
        signal α in
          emit α;
          run Cell7[goto_S4/trans_1,...];
          exit end_automaton % for terminal states
          present α else exit L end present
        end signal
      end present;
      ...
      halt
    end trap
  end loop
end trap
end signal
```

How to order the states of a Region?

- A state can be entered and left in the same tick
- Thus several states can be active in the same tick
- All immediate reachable states have to be later in the loop
- Could forbid immediate transitions
- This would make modeling more difficult

How to order the states of a Region?

- A state can be entered and left in the same tick
- Thus several states can be active in the same tick
- All immediate reachable states have to be later in the loop
- Could forbid immediate transitions
- This would make modeling more difficult
- All transitions form a dependence graph
- Remove all delayed abortions from this graph
- Note: normal terminations might be immediate
- This graph has to be acyclic!

Summary

- SyncCharts can be translated to Esterel, and vice versa
- Translation of SyncCharts to Esterel:
  - Structural induction of And-/Or-Tree
  - Macrostates, regions and reactive cells are translated separately
  - Challenge: unstructured control flow

Summary

- SyncCharts can be translated to Esterel, and vice versa
- Translation of SyncCharts to Esterel:
  - Structural induction of And-/Or-Tree
  - Macrostates, regions and reactive cells are translated separately
  - Challenge: unstructured control flow
- Translation from Esterel to SyncCharts:
  - Structural induction on Esterel code
  - Resulting SyncChart has to be optimized to become readable
  - Challenge: traps

To Go Further

- David Harel, Statecharts: A Visual Formulation for Complex Systems, *Science of Computer Programming*, 8(3), June 1987, pp. 231–274
- Charles André, Semantics of SyncCharts, Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003, `http://www.i3s.unice.fr/~andre/CA%20Publis/SSM03/overview.html`
- Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden, Synthesizing Safe State Machines from Esterel, *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006, `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/lcts06.pdf`
- Ulf Rüegg, *Interactive Transformations for Visual Models*, B.Sc.-Thesis, Dept. of CS, Kiel University, March 2011, `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/uru-bt.pdf`