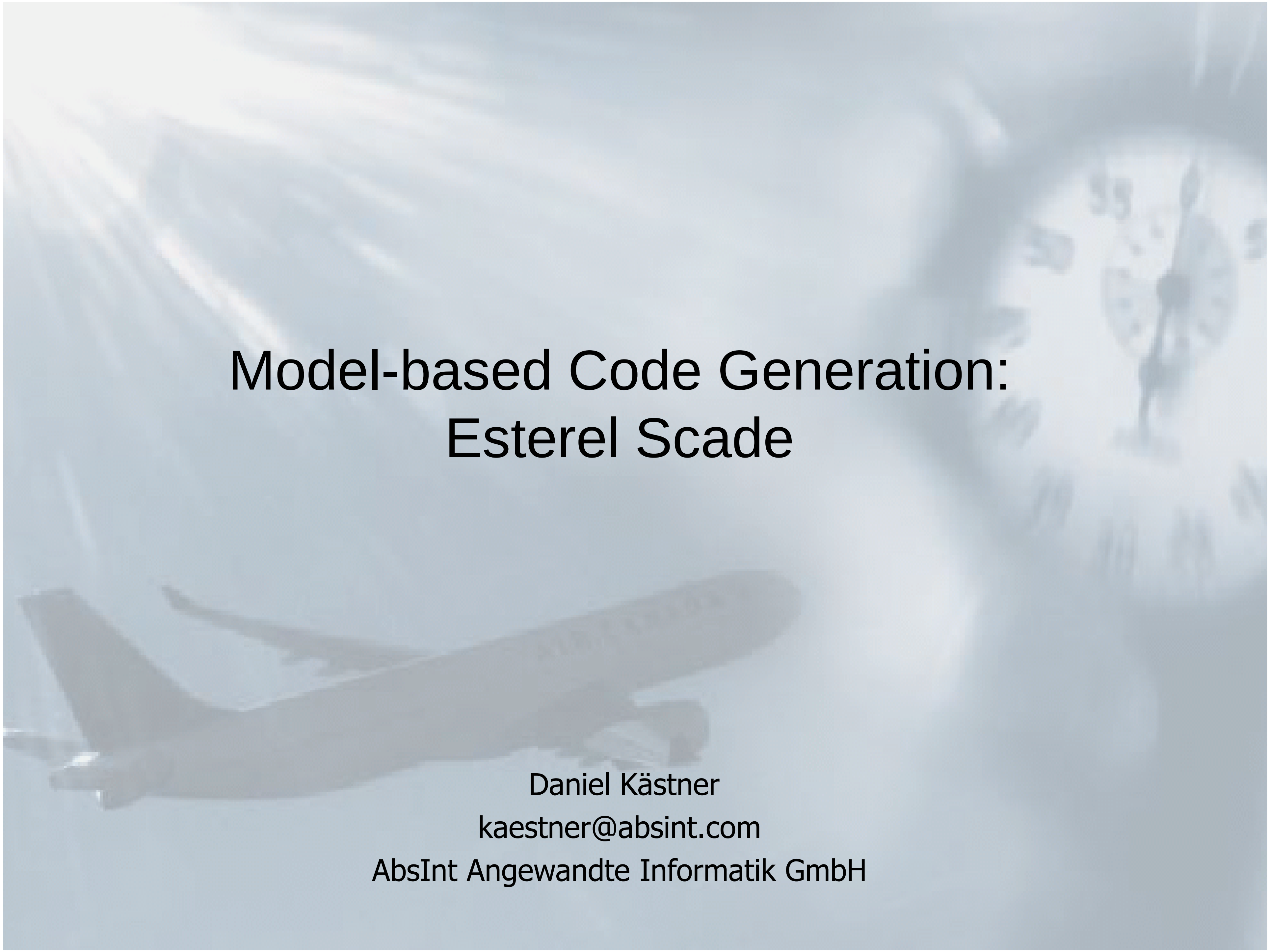
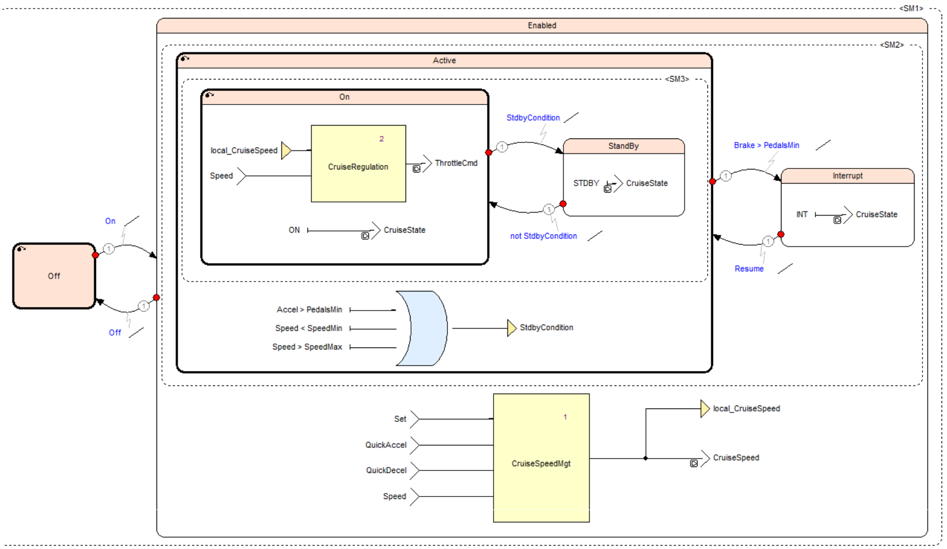


Model-based Code Generation:
Esterel Scade

Daniel Kästner
kaestner@absint.com
AbsInt Angewandte Informatik GmbH

AbsInt GmbH

Model-based Software Development



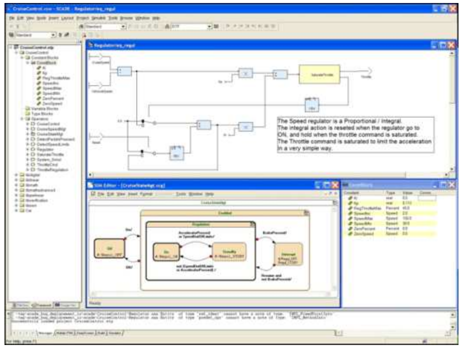
- Model is software specification.

- Hardware/Software codesign.
- Prototyping.
- Formal verification.

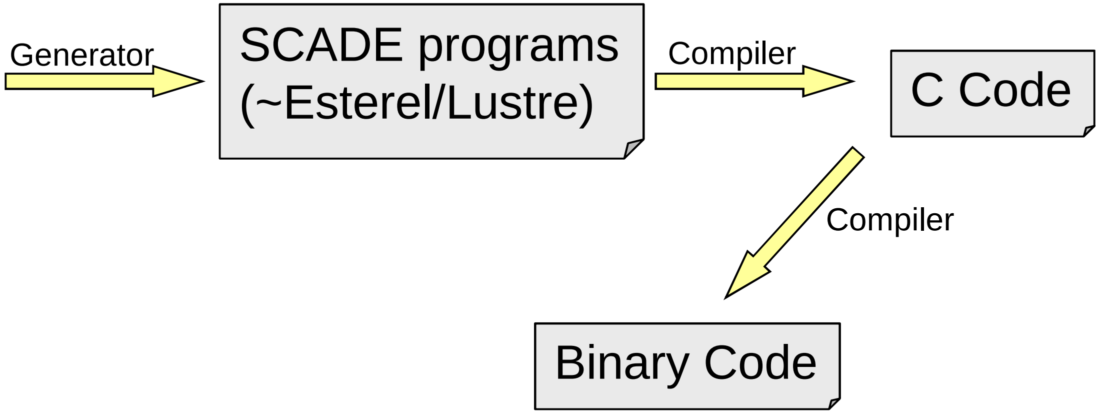
- Automated & integrated development tools:
  - Simulation.
  - Documentation.
  - Automatic code generation.

- Automated & integrated verification and test methods
  - Model checking
  - Static system analysis
  - Synthesis of test suites

Model-based Software Development



Esterel Scade Suite
– SCADE language
– SyncCharts/SSM



Generator

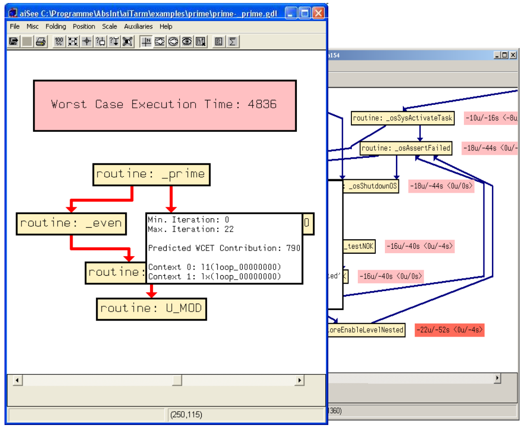
SCADE programs
(~Esterel/Lustre)

Compiler

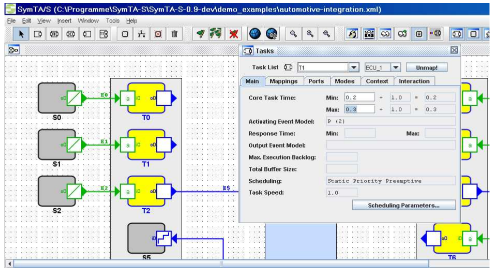
C Code

Compiler

Binary Code



– aiT WCET Analyzer
– StackAnalyzer



SymTA/S
– System-level
Schedulability Analysis

Embedded Systems

- Typically, embedded systems are reactive systems:

  „A reactive system is one which is in continual interaction with is environment and executes at a pace determined by that environment“
  [Bergé, 1995]

  Behavior depends on input and current state.

➢ automata model appropriate

Finite Automata

- Non-deterministic finite automaton (NFA):
  $M = (\Sigma, Q, \Delta, q_0, F)$ where
  - Σ: finite alphabet
  - Q: finite set of states
  - $q_0 \in Q$: initial state
  - $F \subseteq Q$: final states
  - $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$

- M is called a deterministic finite automaton, if Δ is a partial function

$$\delta : Q \times \Sigma \rightarrow Q$$

Mealy Automata

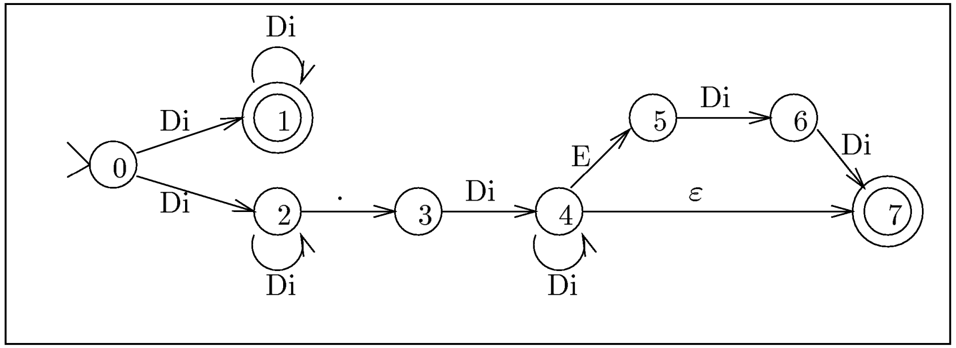
- Mealy automata are finite-state machines that act as transducers, or translators, taking a string on an input alphabet and producing a string of equal length on an output alphabet.
- A machine in state q_j, after reading symbol sigma σ_k writes symbol λ_k; the output symbol depends on the state just reached and the corresponding input symbol.
- A Mealy automaton is a six-tuple $M_E=(Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where
  - Q is a finite set of states
  - Σ is a finite input alphabet
  - Γ is a finite output alphabet
  - $\delta: Q \times \Sigma \rightarrow Q$ is the transition function
  - $\lambda: Q \times \Sigma \rightarrow \Gamma$ is the output function
  - q_0 is the initial state

Moore Automata

- Moore automata are finite-state machines that act as transducers, or translators, taking a string on an input alphabet and producing a string of equal length on an output alphabet.
- Symbols are output after the transition to a new state is completed; output symbol depends only on the state just reached.
- A Moore automaton is a six-tuple $M_O=(Q, \Sigma, \Gamma, \delta, \lambda, q_0)$ where
  - Q is a finite set of states
  - Σ is a finite input alphabet
  - Γ is a finite output alphabet
  - $\delta: Q \times \Sigma \text{ -> } Q$ is the transition function
  - $\lambda: Q \text{ -> } \Gamma$ is the output function
  - q_0 is the initial state

Simple State Transition Diagram

- Used to represent a finite automaton
- Nodes: states
- q_0 has special entry mark
- Final states are doubly circled
- An edge from *p* to *q* is labelled by *a* if $(p,a,q) \in \Delta$
- Example: integer and real constants:



- Problem: all combinations of states have to be represented explicitly, leading to exponential blow-up.

SyncCharts

- Visual formalism for describing states and transitions of a system in a modular fashion.
- Extension of state-transition diagrams (Mealy/Moore automata):
  - Hierarchy
  - Modularity
  - Parallelism
- Is fully deterministic.
- Tailored to control-oriented applications (drivers, protocols).
- Implements sychronous principle.

Synchroneous Programming

- Program typically implements an automaton:
  - state: valuations of memory
  - transition: reaction, possibly involving many computations

- Synchronous paradigm:
  - Reactions are considered atomic, ie they take no time. (Computational steps execute like combinatorial circuits.)
  - Synchronous broadcast: instantaneous communication, ie each automaton in the system considers the outputs of others as being part of its own inputs.

Synchronous Programming

- Important requirement: guaranteeing deterministic behavior.
- Time is divided into discrete ticks (also called cycles, steps, instants).
- Simple implementation: sampling / cyclic executive:

```
<Initialize Memory>
Foreach period do
        <Read Inputs>
        <Compute Outputs>
        <Update Memory>
```

- Verification of timing behavior: prove that the worst-case execution time (WCET) of any reaction fits between two iterations of the cyclic executive.
- Implicit assumption: presence of a global clock. This makes application in distributed environments difficult.

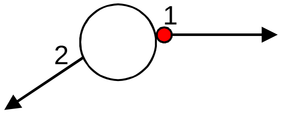
Overview

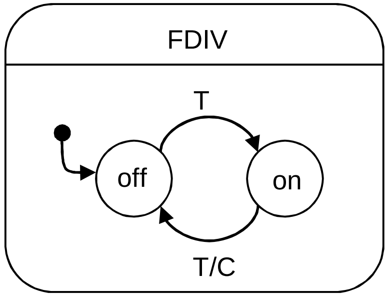
- StateCharts:
  - First, and probably most popular formal language for the design of reactive systems.
  - Focus on specification and design, not designed as a programming language.
  - Determinism is not ensured.
  - No standardized semantics.

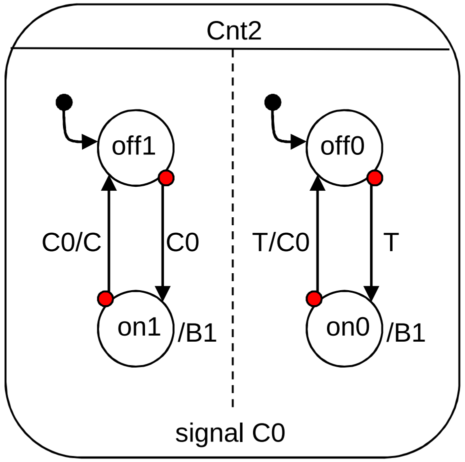
- Programming languages for designing reactive systems:
  - ESTEREL [Berry]: textual imperative language.
  - LUSTRE [Caspi, Halbwachs]: textual declarative language. Tailored to data-flow oriented systems (e.g. regulation systems).
  - SCADE [Esterel Inc.]. Enhanced LUSTRE, graphical and textual formalism.
  - SyncCharts / SSM: Graphical formalism corresponding to ESTEREL.

SyncCharts

- States (circles and rectangles):
  - can be named
  - two types:
    - simple state (circle)
    - macrostate (rounded rectangle): contain a hierarchy of other states
  - are optionally labelled*: /<effect>
- Transitions (arrows):
  - are labelled*: <trigger>/<effect>
    - All components are optional.
  - three types:
    - strong abort ●→
    - weak abort →
    - normal termination ▶→
  - can have priorities (-> determinism)

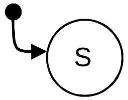


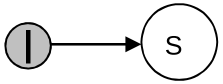


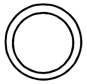


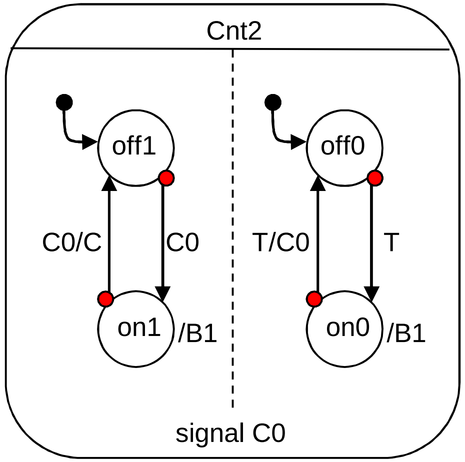
*Triggers and effects are signals, or combinations of signals using boolean operations *or*, *and* and *not.*

States & State Transition Graphs

- Special states:
  - Initial state: 

(alternative notation: 

)
  - Terminal state: 



- State Transition Graph: connected labeled graph made of states connected by transitions, with an initial state.
- Two types of states:
  - Simple state: just carries a label.
  - Macrostate: contains at least one state transition graph.

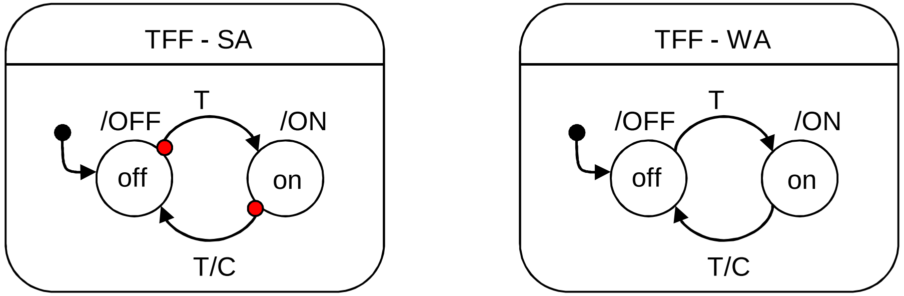
- At each instant there is one and only one active state.
- An active state waits for the satisfaction of the trigger of one of its outgoing transitions, at an instant strictly posterior to its entering (activation).

State and Transition Labels

- Signals are characterized by their presence status (+,-, ⊥).
  - Valued signal: signals conveys a value of a given type.
  - Pure signal: no value conveyed.
- tick: implicit signal present at every instant.
- A trigger is satisfied ⇔ associated signal is present.
- Transition labels:
  - When the trigger is satisfied, the transition is said to be enabled.
  - The transition is immediately taken and emits the associated signals.
  - The firing of a transition is fully deterministic and takes no time.
- Node labels:
  - Signal emission depends on transition type (strong/weak abort)
  - Signals are emitted when…

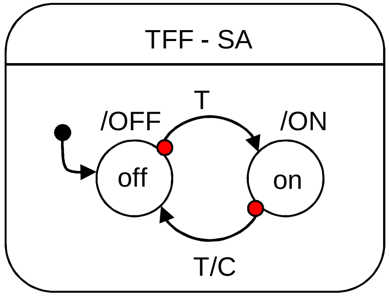
	…entering	…in	…exiting
Weak abort	Yes	Yes	Yes
Strong abort	Yes	Yes	No

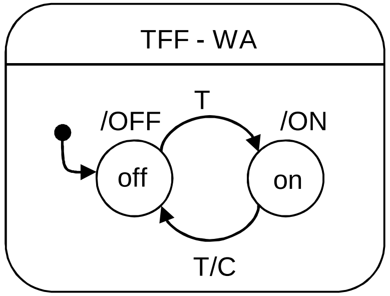
Example: Strong vs. Weak Abort



Instant	Input	TFF-SA Output	TFF-WA Output
1			
2	T		
3			
4	T		
5			
6	T		
7	T		
8	T		
9			

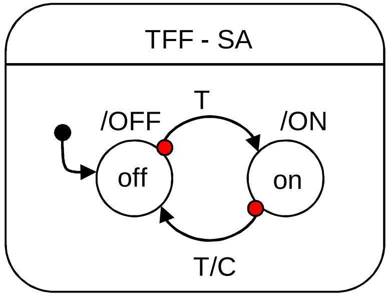
Example: Strong vs. Weak Abort

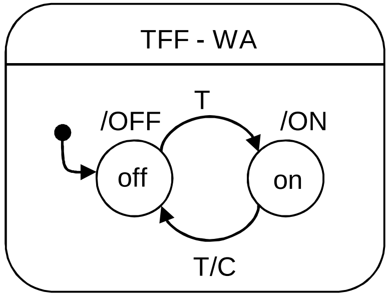




Instant	Input	TFF-SA Output	TFF-WA Output
1		OFF	
2	T	ON	
3		ON	
4	T	C, OFF	
5		OFF	
6	T	ON	
7	T	C,OFF	
8	T	ON	
9		ON	

Example: Strong vs. Weak Abort

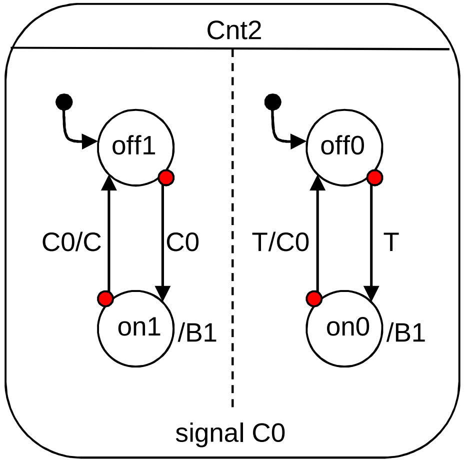




Instant	Input	TFF-SA Output	TFF-WA Output
1		OFF	OFF
2	T	ON	OFF,ON
3		ON	ON
4	T	C, OFF	ON,C,OFF
5		OFF	OFF
6	T	ON	OFF,ON
7	T	C,OFF	ON,C,OFF
8	T	ON	OFF,ON
9		ON	ON

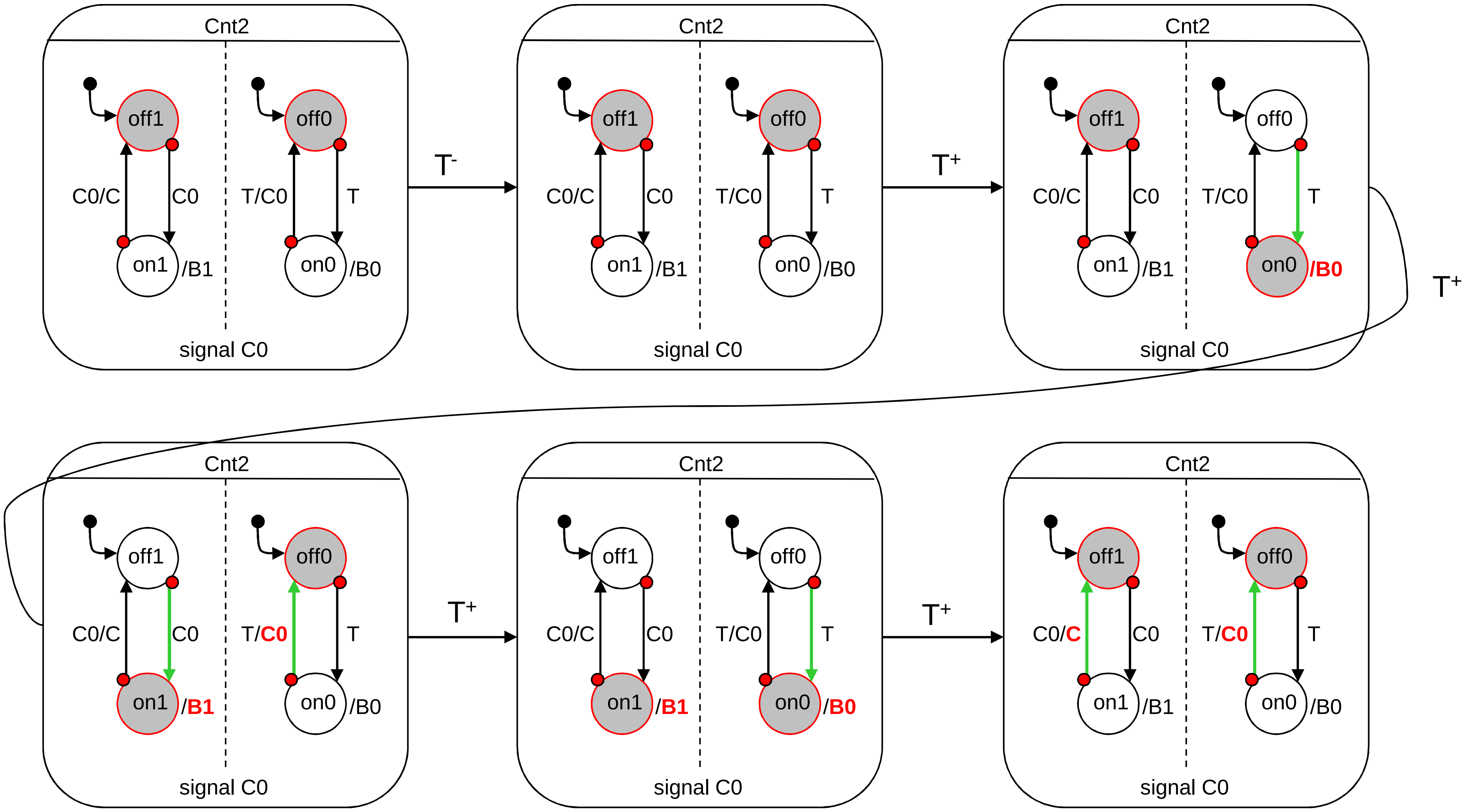
Concurrency

- A macrostate can contain a parallel composition of separate concurrent STGs. Graphical notation: dashed separation line.
- STGs are coupled by shared signals.
- A local signal is declared by the keyword signal and its scope is the containing macrostate.
- A set of (concurrent) active states is called a configuration.



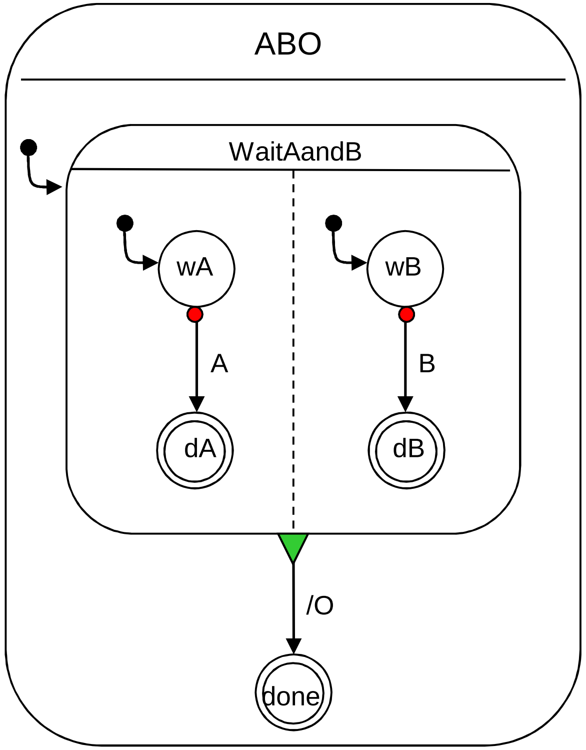
- Notation:
  - Active state ● Taken transition → Emitted signal S
  - S^+: presence of signal S S^-: absence of signal S

Example Reaction

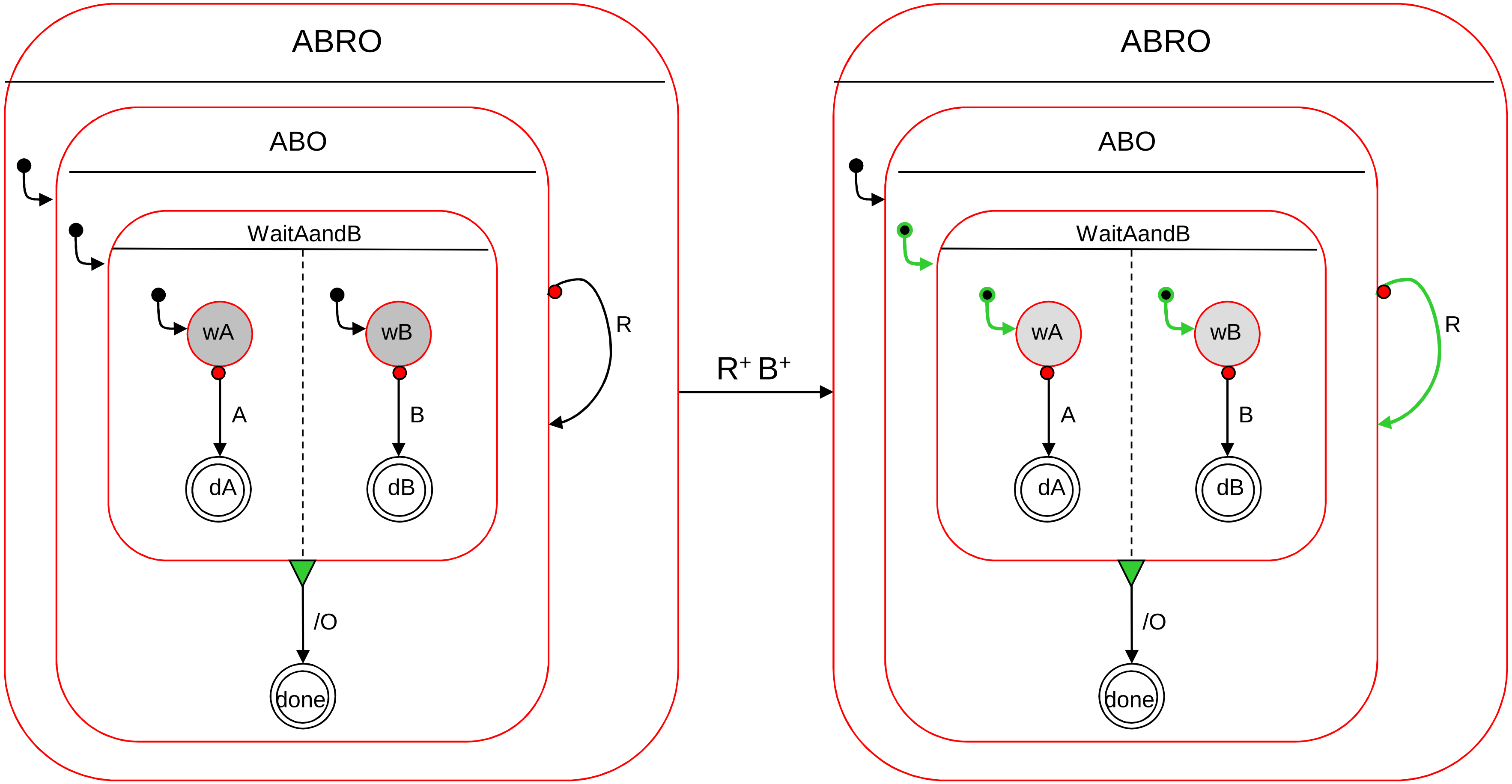


Concurrency and Normal Termination

- When each concurrent STG in a macrostate reaches a final state, then the macrostate is immediately exited by its normal termination transition.



Concurrency and Abort



Transitions

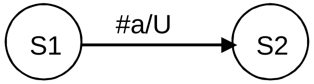
- A strong abort prevents any execution in the preempted state.

- For any state
  - every outgoing transition has a different priority
  - any strong abort transition has priority over any weak abort transition
  - any weak abort transition has priority over a normal termination transition

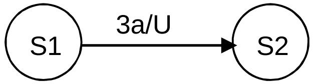
- There are no inter-level transitions.

SyncCharts: Advanced Constructs

- Immediate transition
  - Syntax: #<trigger>/<effect>

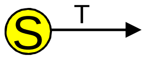


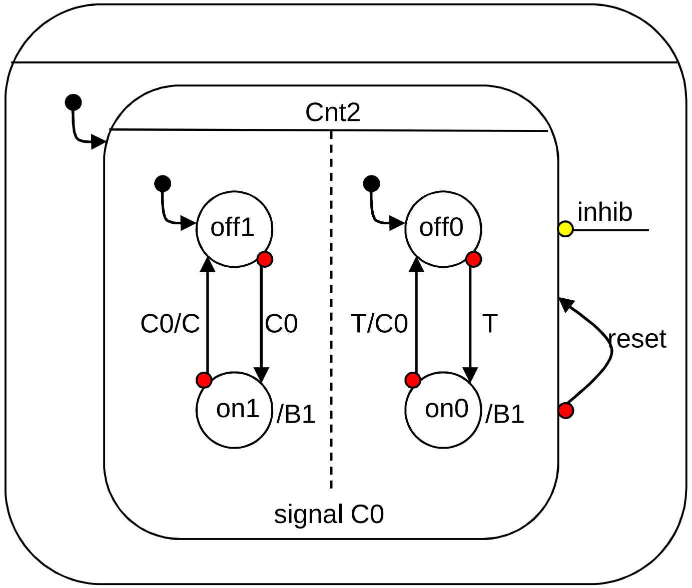
  - The trigger may be satisfied as soon as the state is entered: An active state waits for the satisfaction of the trigger of one of its outgoing transitions, at an instant strictly posterior to its entering, or immediately in case of an immediate transition.
- Count delays for transitions
  - Syntax: <factor><trigger>/<effect>



  - <factor> is the natural number of instants a transition must be active before it is executed. These active instants need not be consecutive, but the source state (S1) must be active all the time.

Suspension

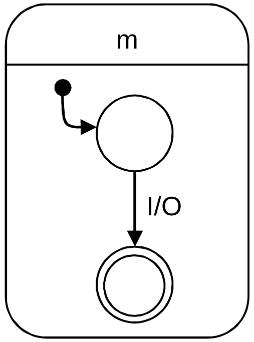
- A suspension is associated with a trigger. If the trigger is satisfied the reaction is suspended in the target state: the execution of the preempted state is frozen.
- Notation: ,or: 
- Note: aborts take priority over suspensions.



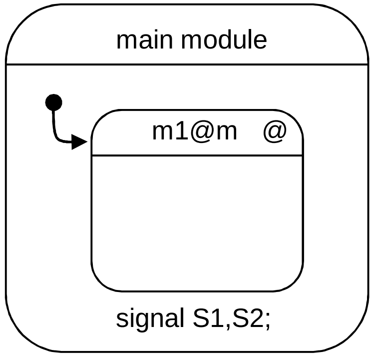
Run Modules

- Macrostates are states that have their own behavior (also called processes). They can be abstracted as modules, similarly to procedures in programming languages.
- Modules are instantiated by using run modules (corresponding to procedure calls).
- A signal interface renaming has to be defined for run modules.

Module M with interface
input I;
output O;

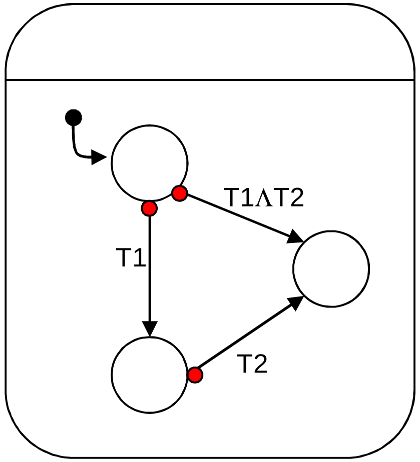


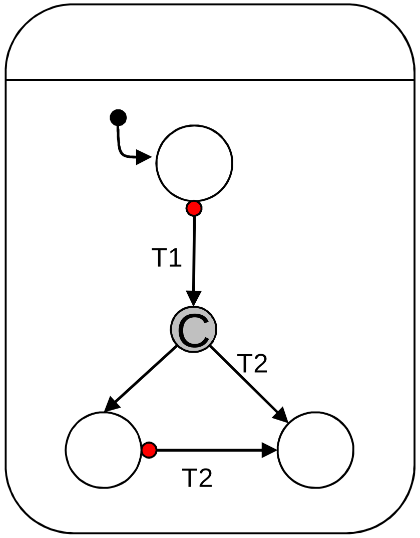
… used as a run module with the following signal binding:
signal S1 / I;
signal S2 / O;



Conditional Connector

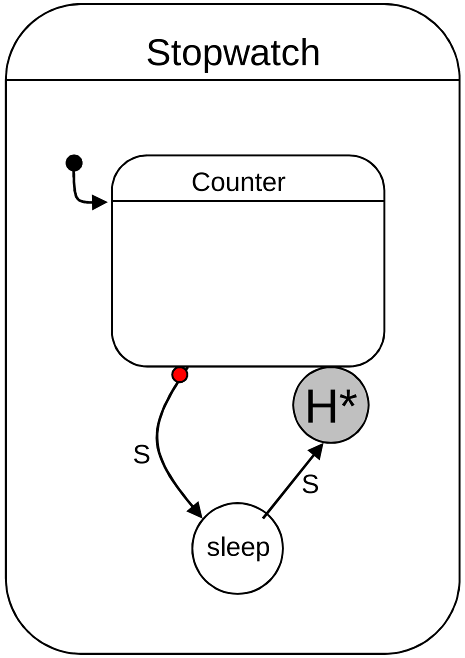
- Introduction of conditional pseudostate Ⓒ
- Concise representation of scenarios where a common trigger is shared by several outgoing transitions.
- All departing transitions are immediate.
- One departing “default” transition without condition must be present.



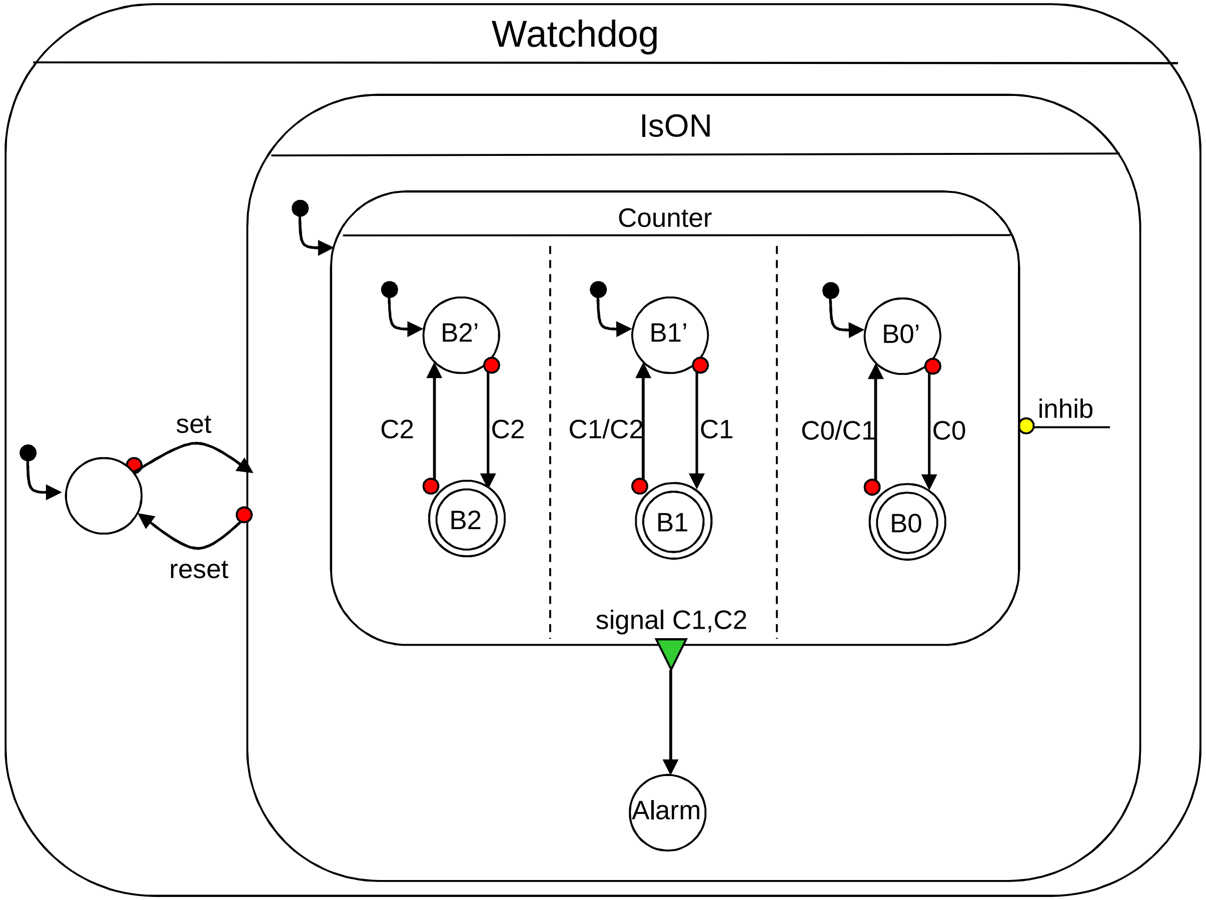


History Connector

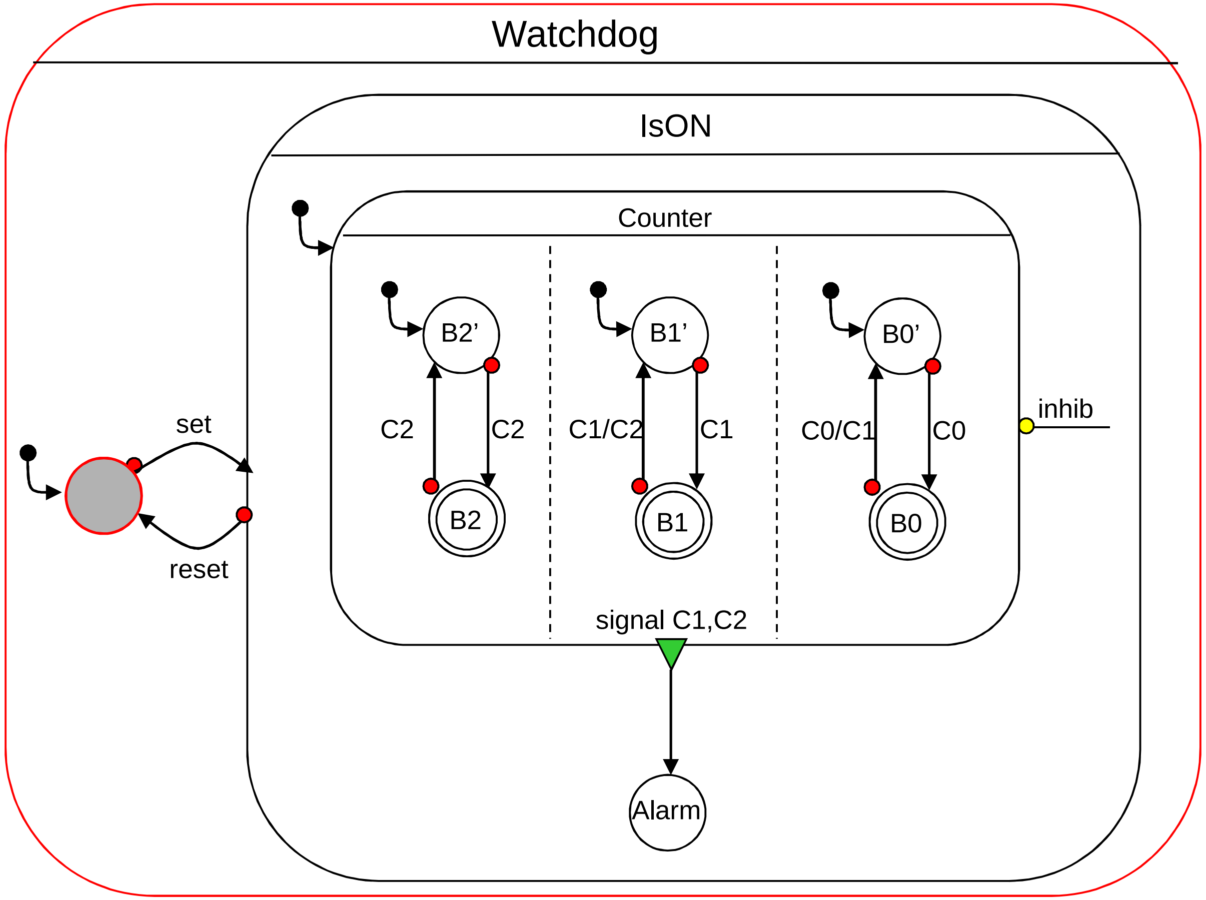
- Directly attached to macrostates
- Only incoming transitions can connect
- The previous state of the macrostate is restored when it is entered through a history connector.



Example: Watchdog

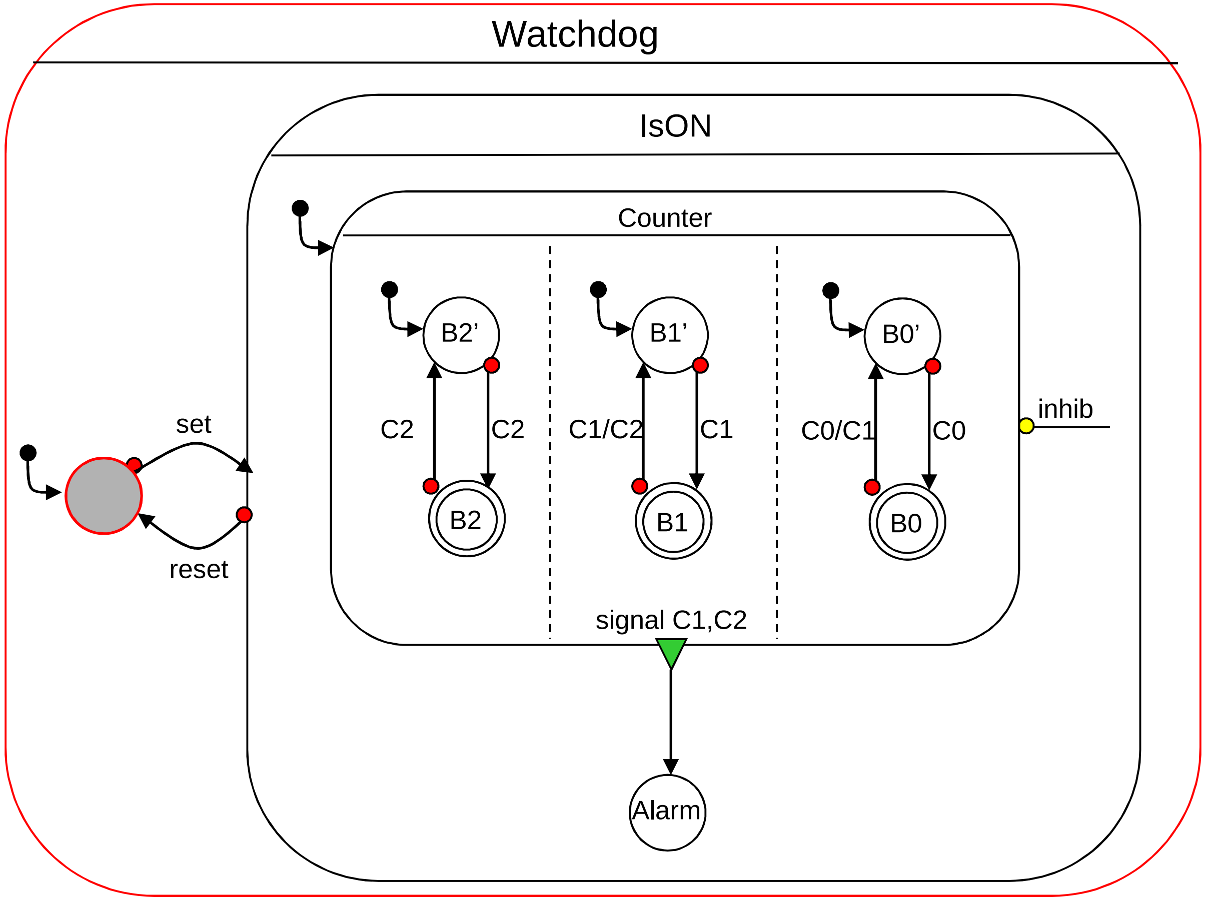


Example: Watchdog



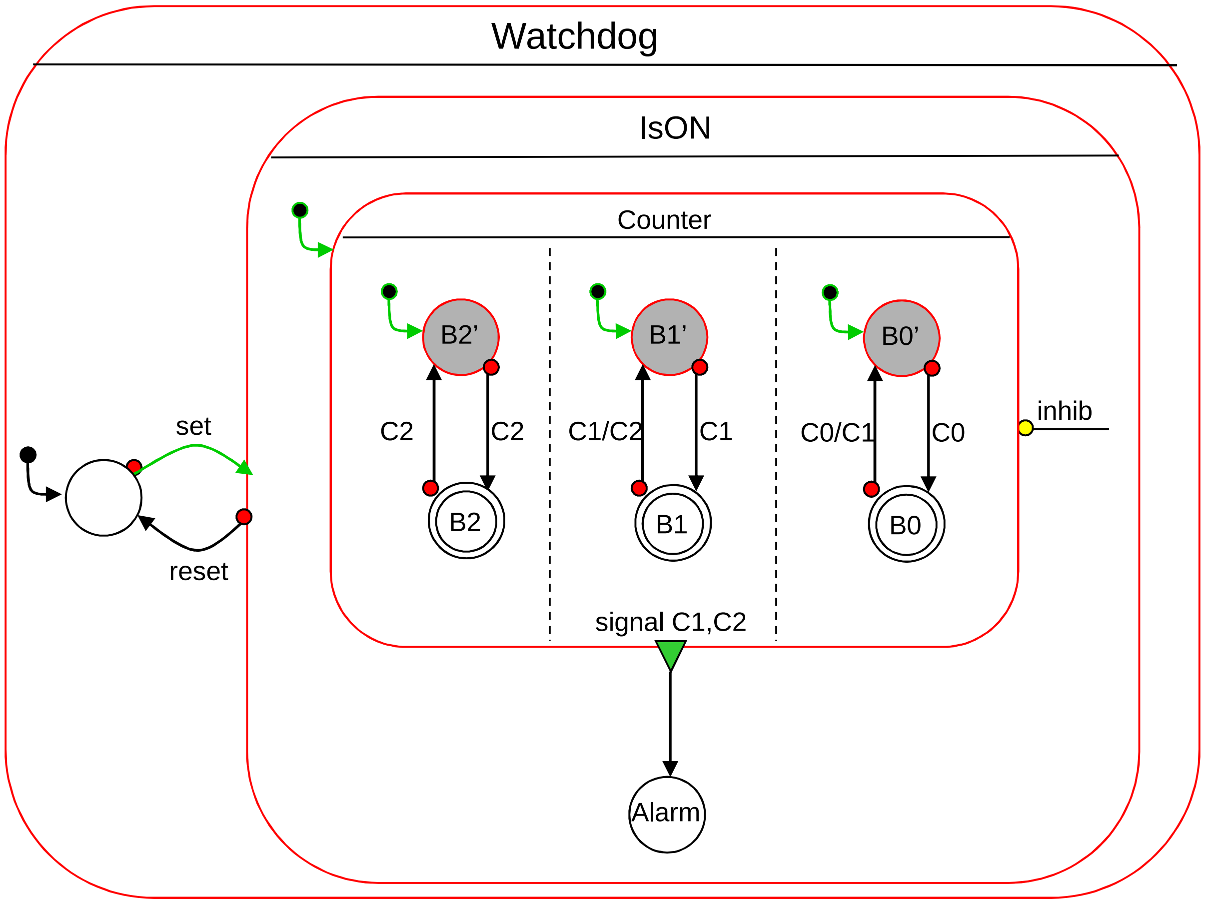
Example: Watchdog

$C0^+$



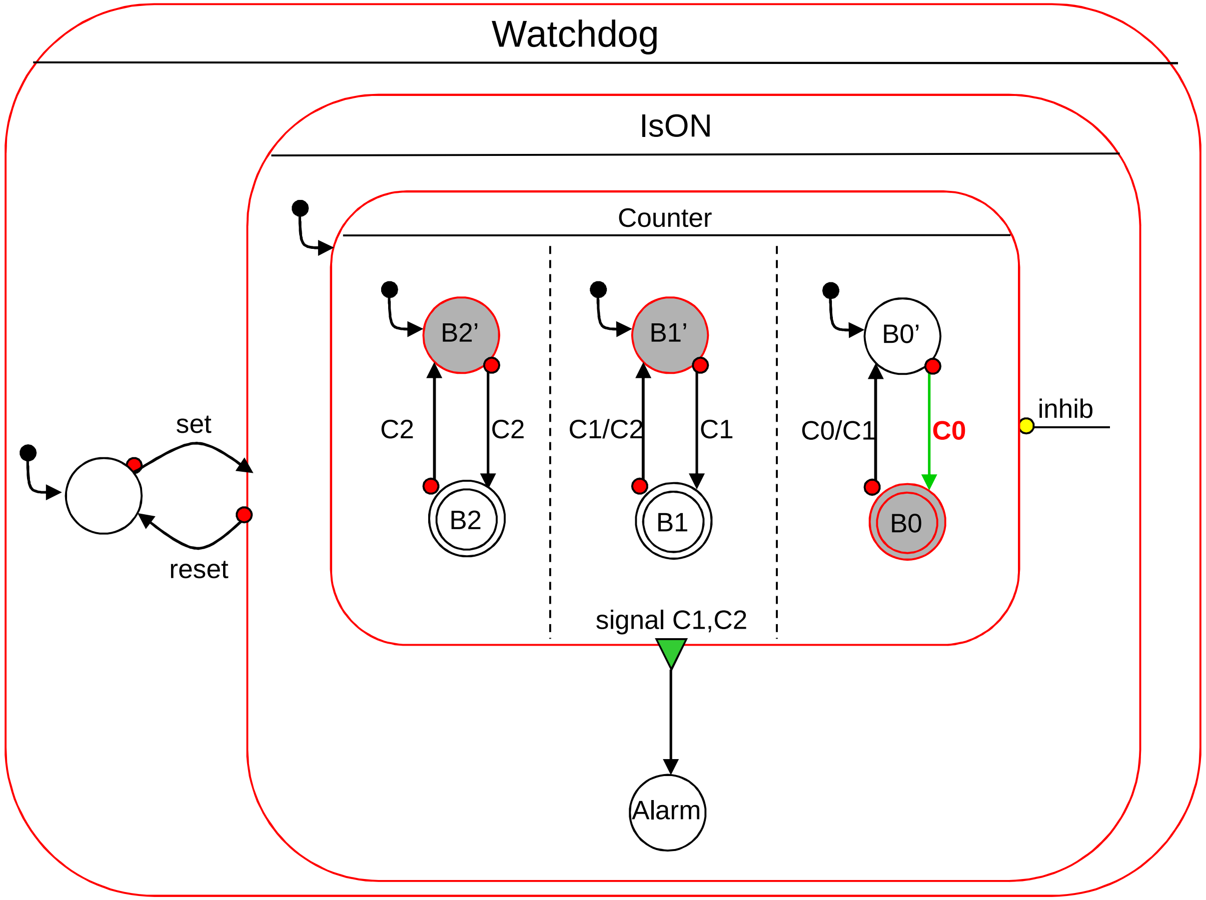
Example: Watchdog

set⁺



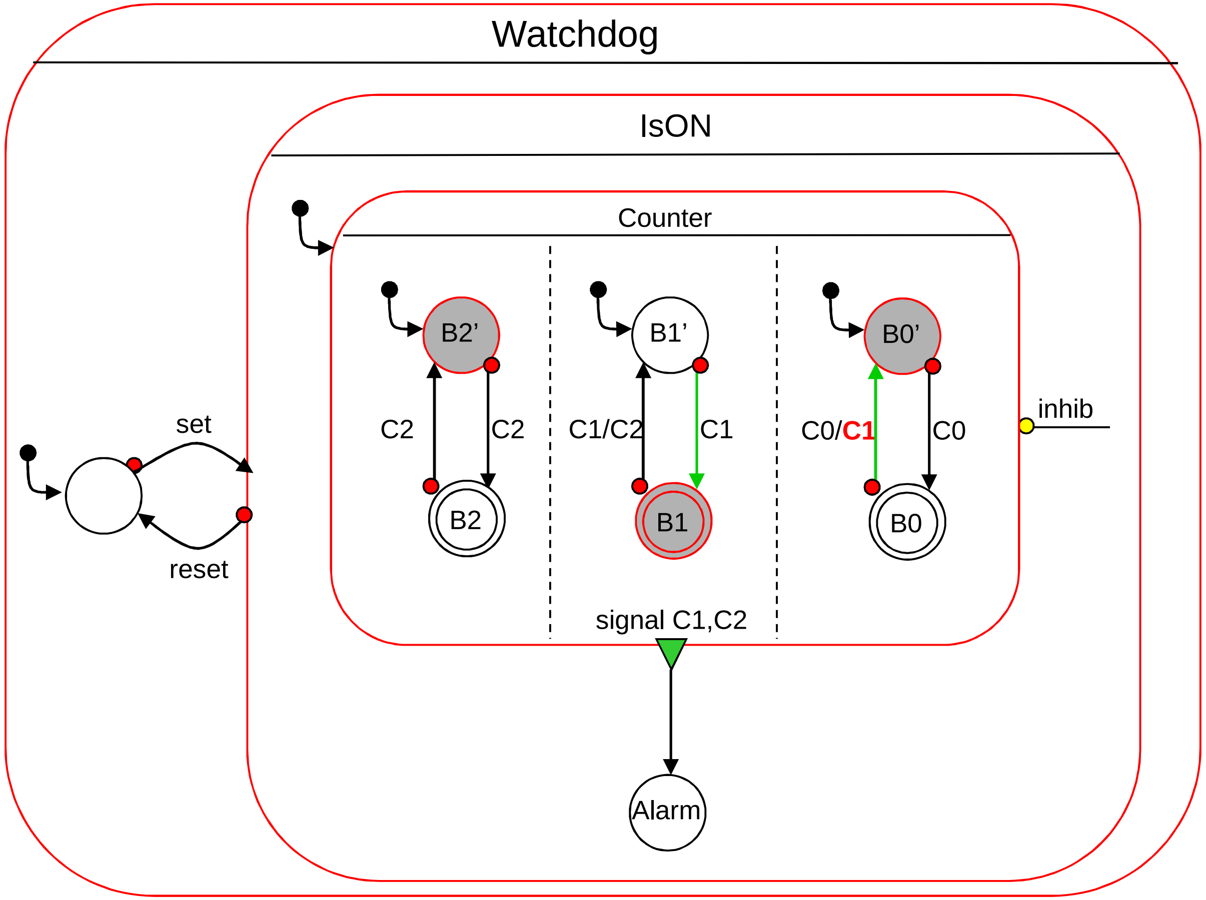
Example: Watchdog

$C0^{+}$



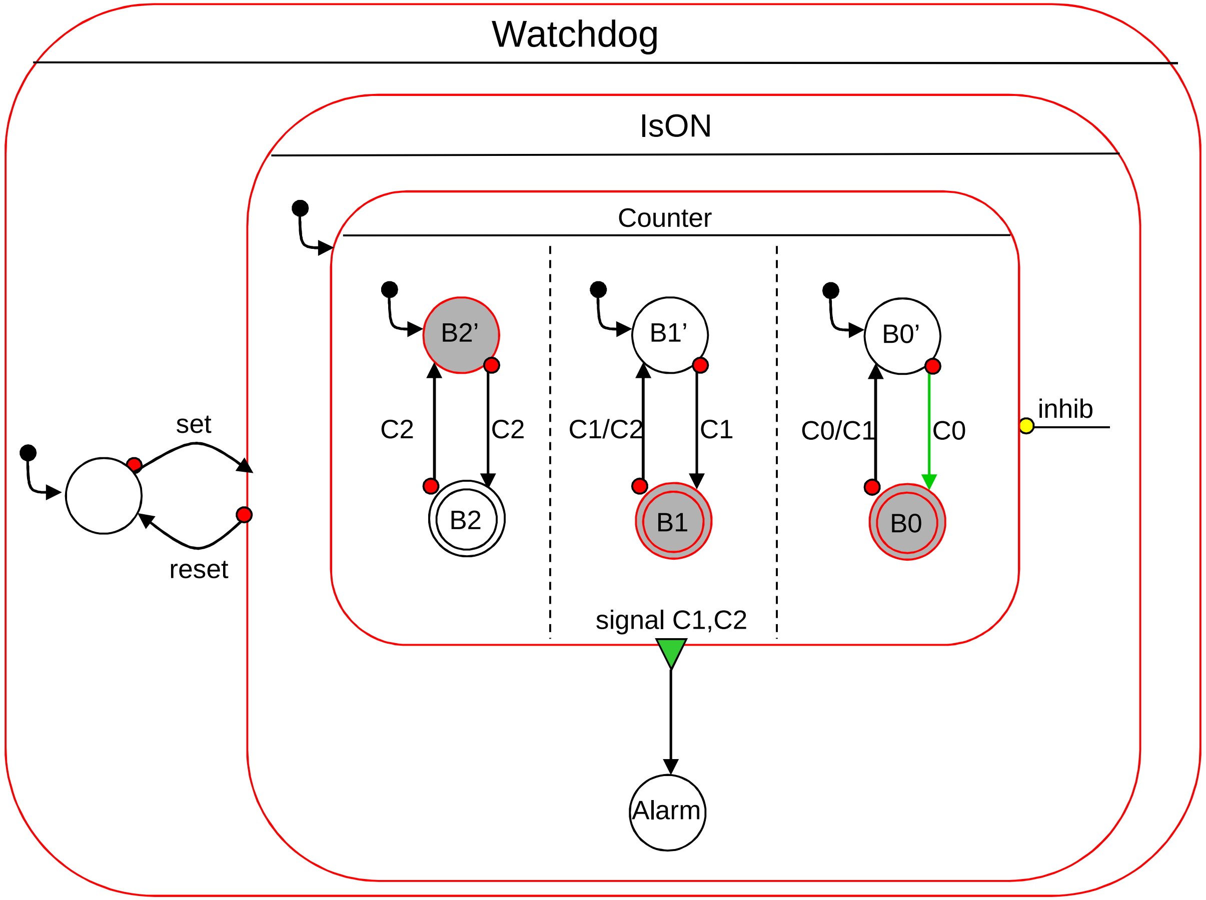
Example: Watchdog

$C0^+$



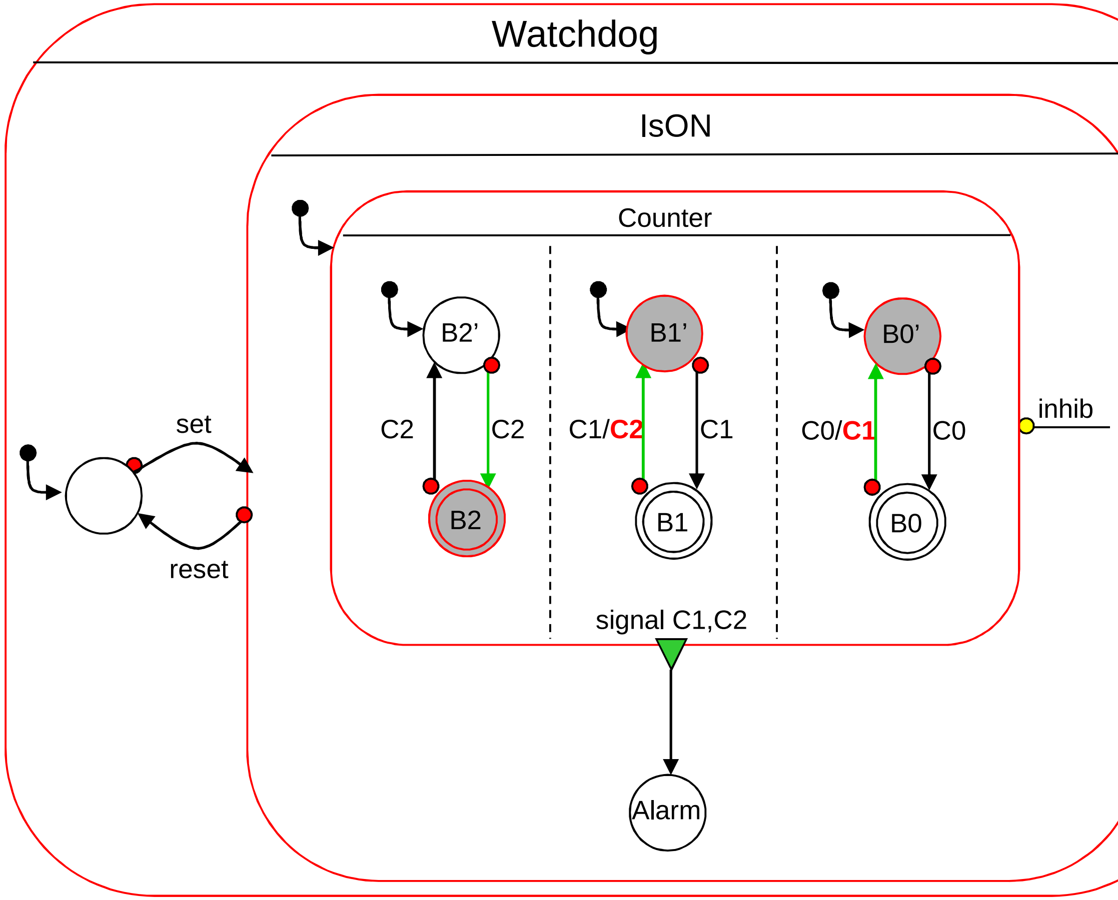
Example: Watchdog

C0$^+$



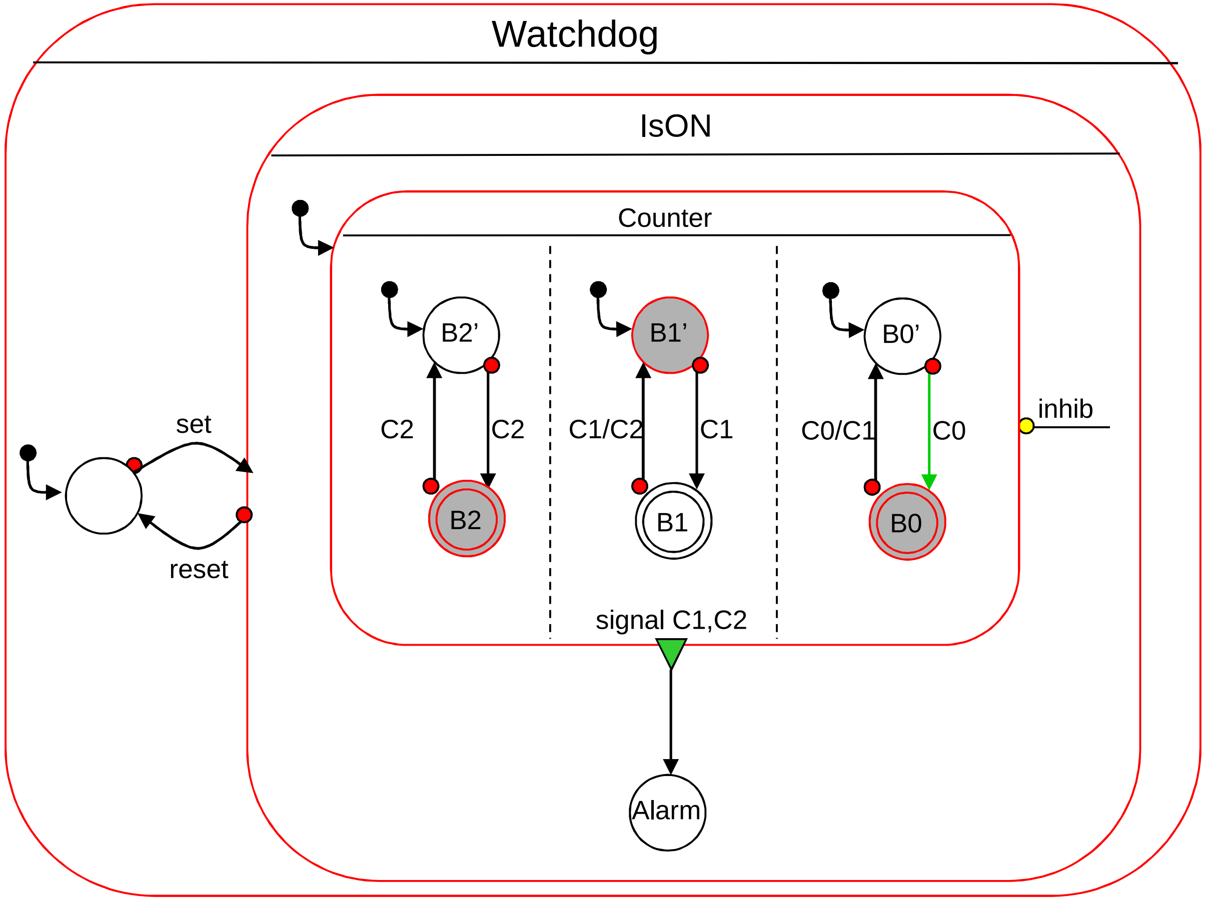
Example: Watchdog

C0⁺



Example: Watchdog

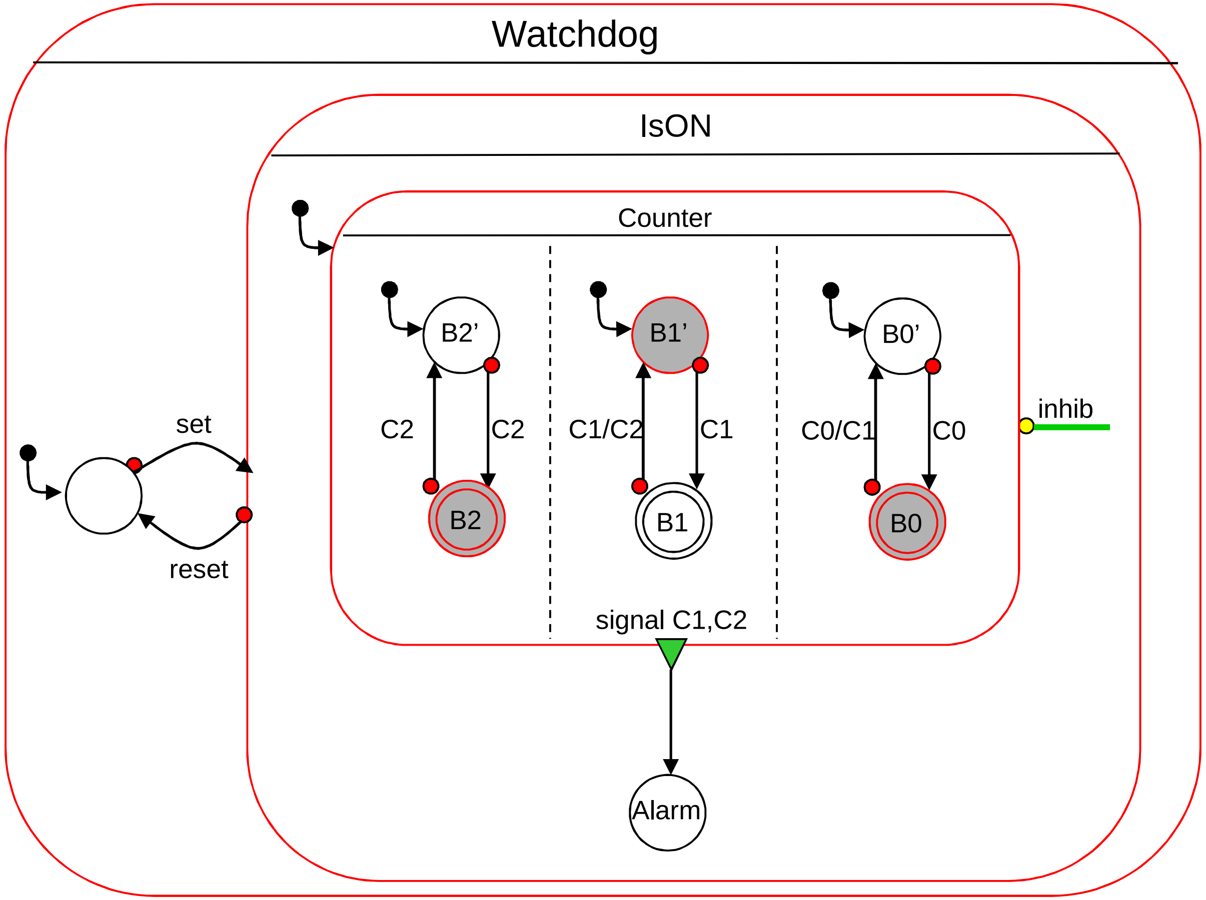
$C0^+$



Example: Watchdog

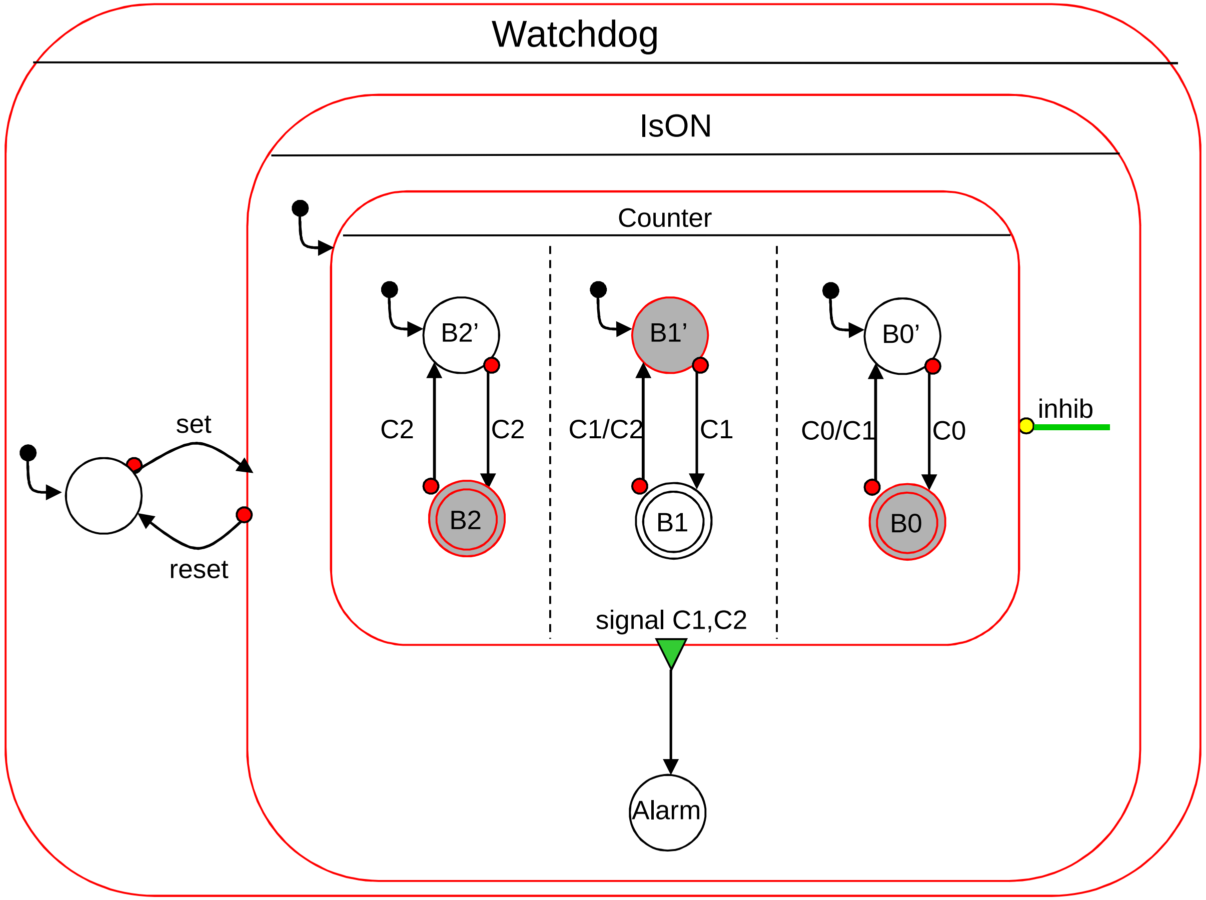
inhib$^+$

C0$^+$



Example: Watchdog

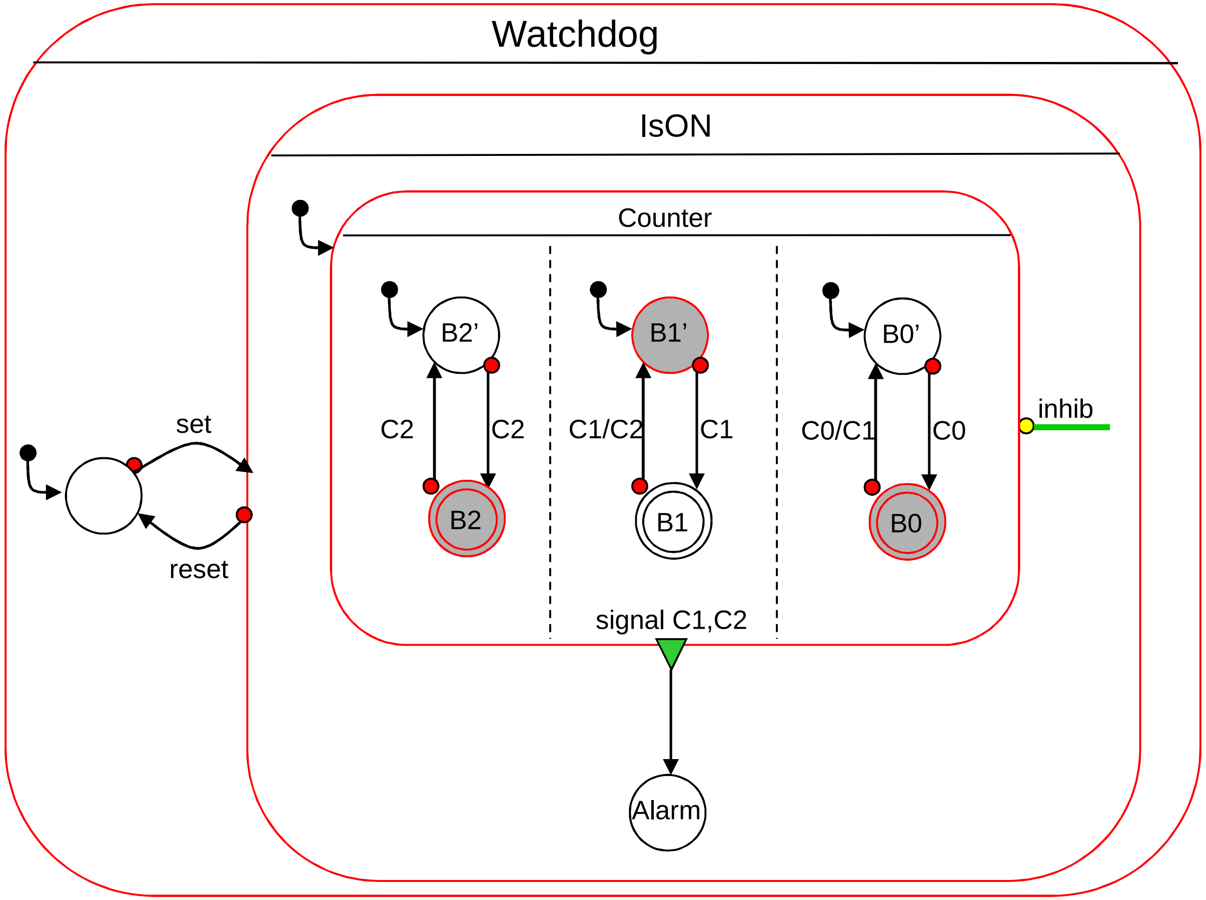
inhib⁺



Example: Watchdog

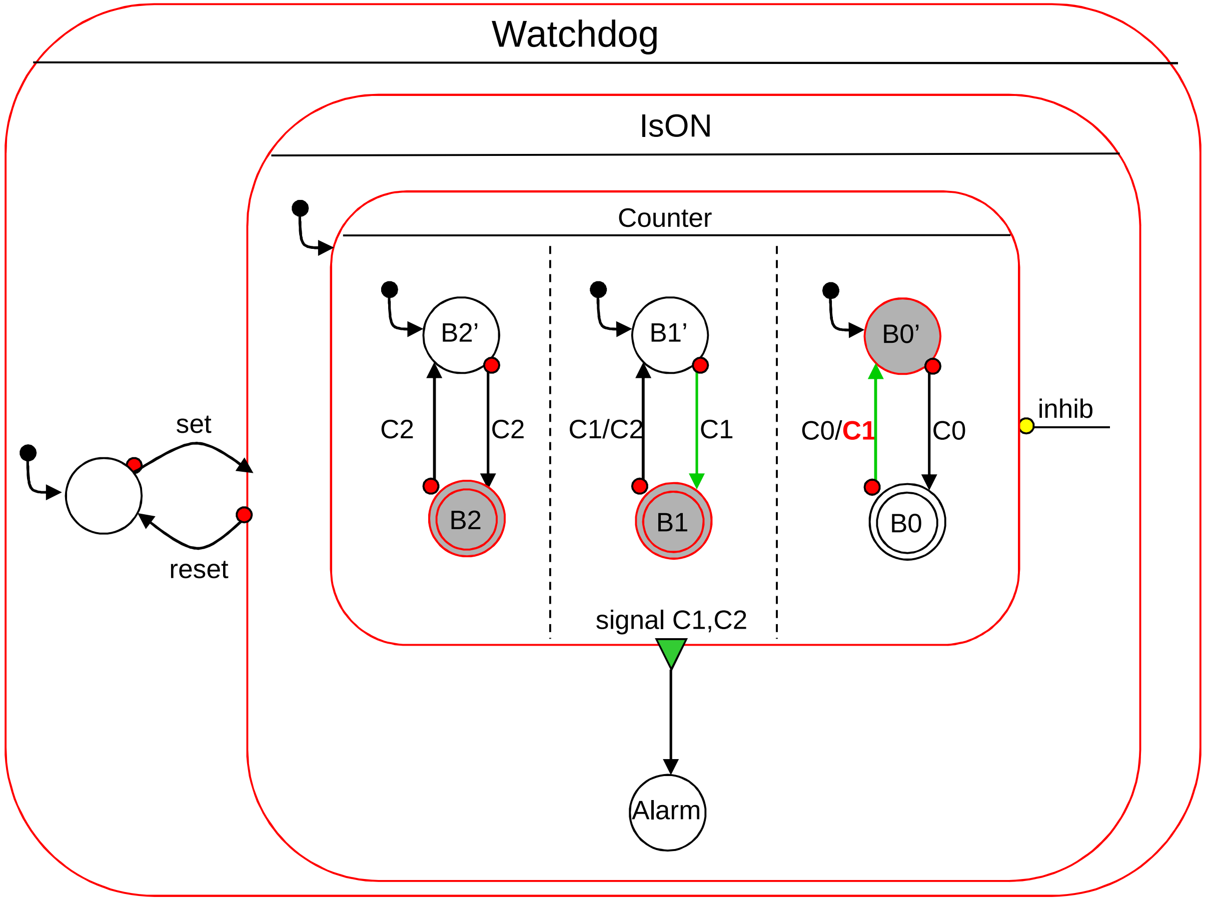
inhib$^+$

C0$^+$



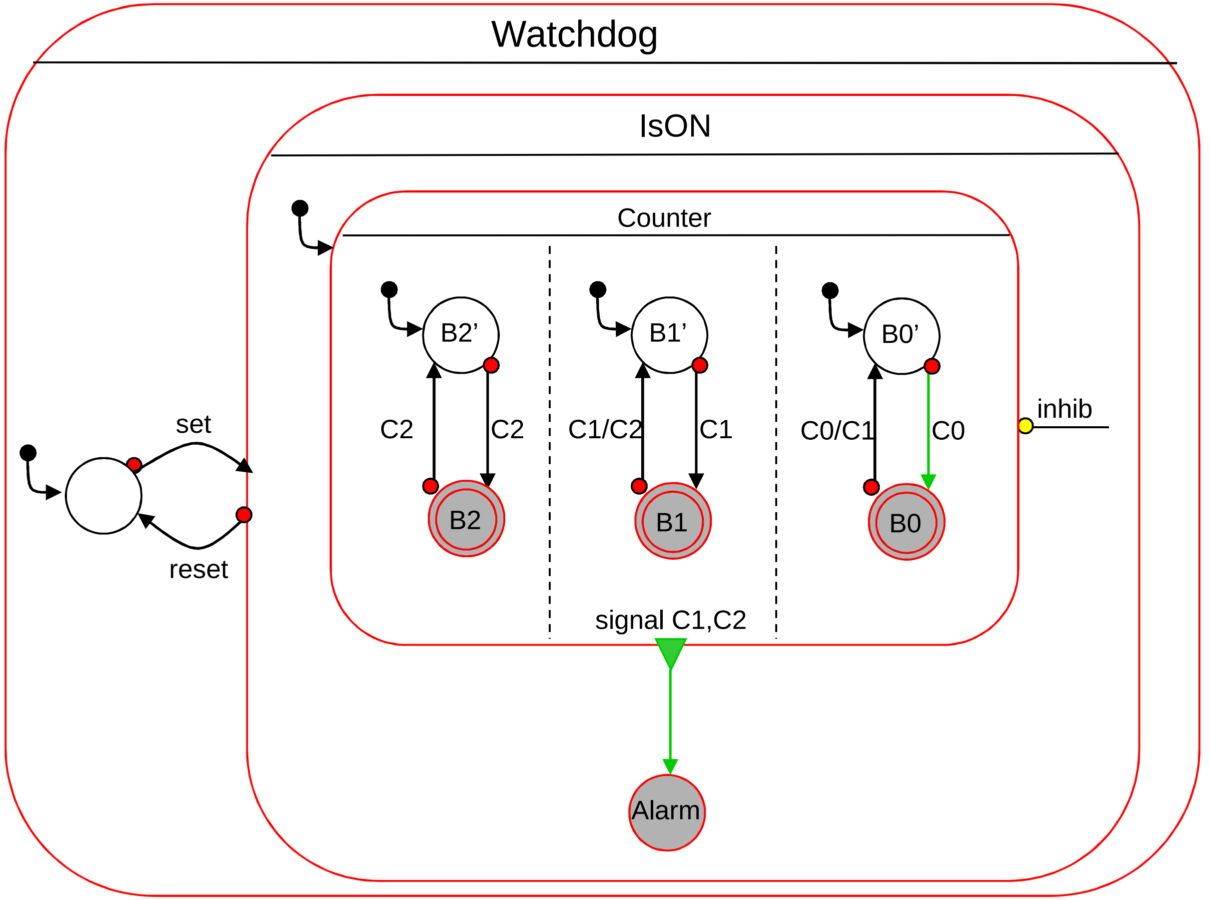
Example: Watchdog

$C0^+$



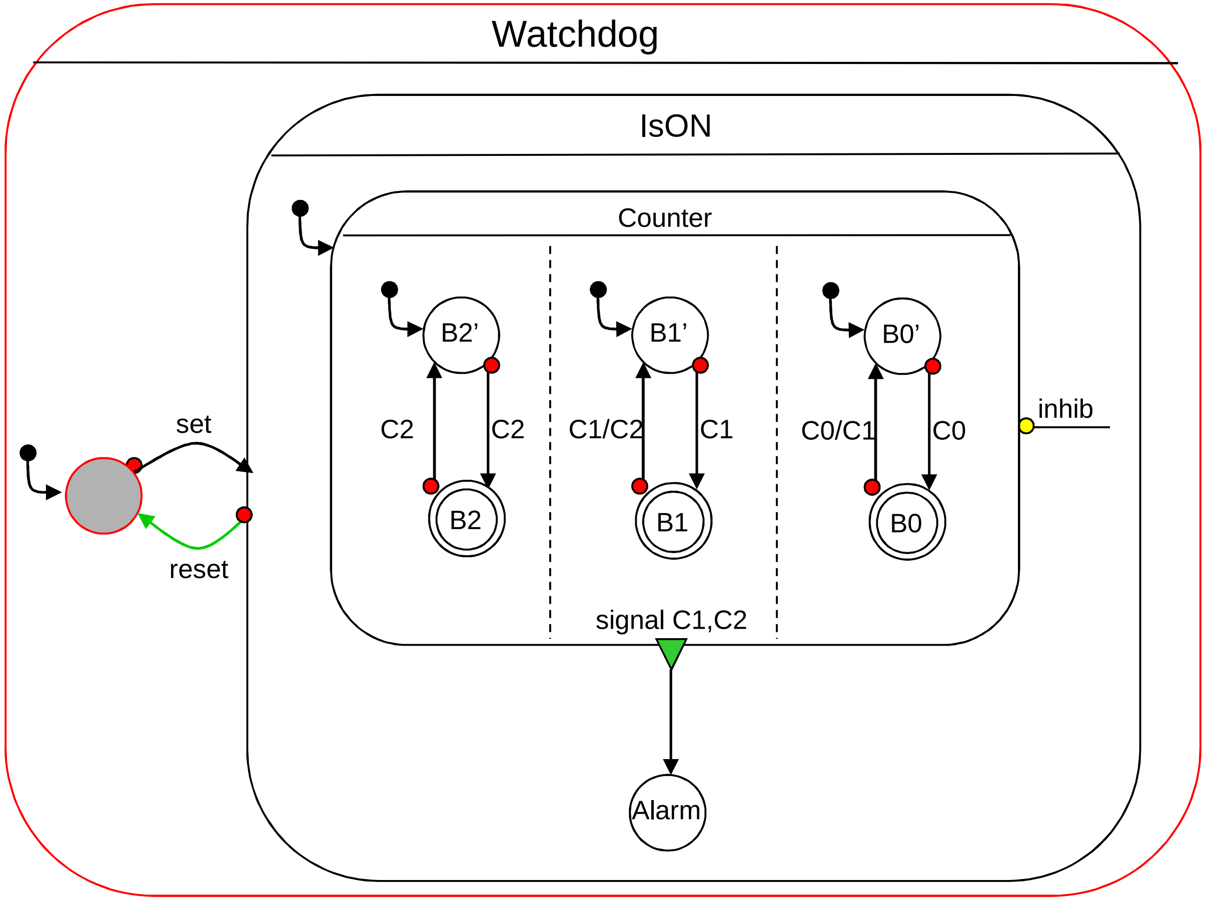
Example: Watchdog

$C0^+$

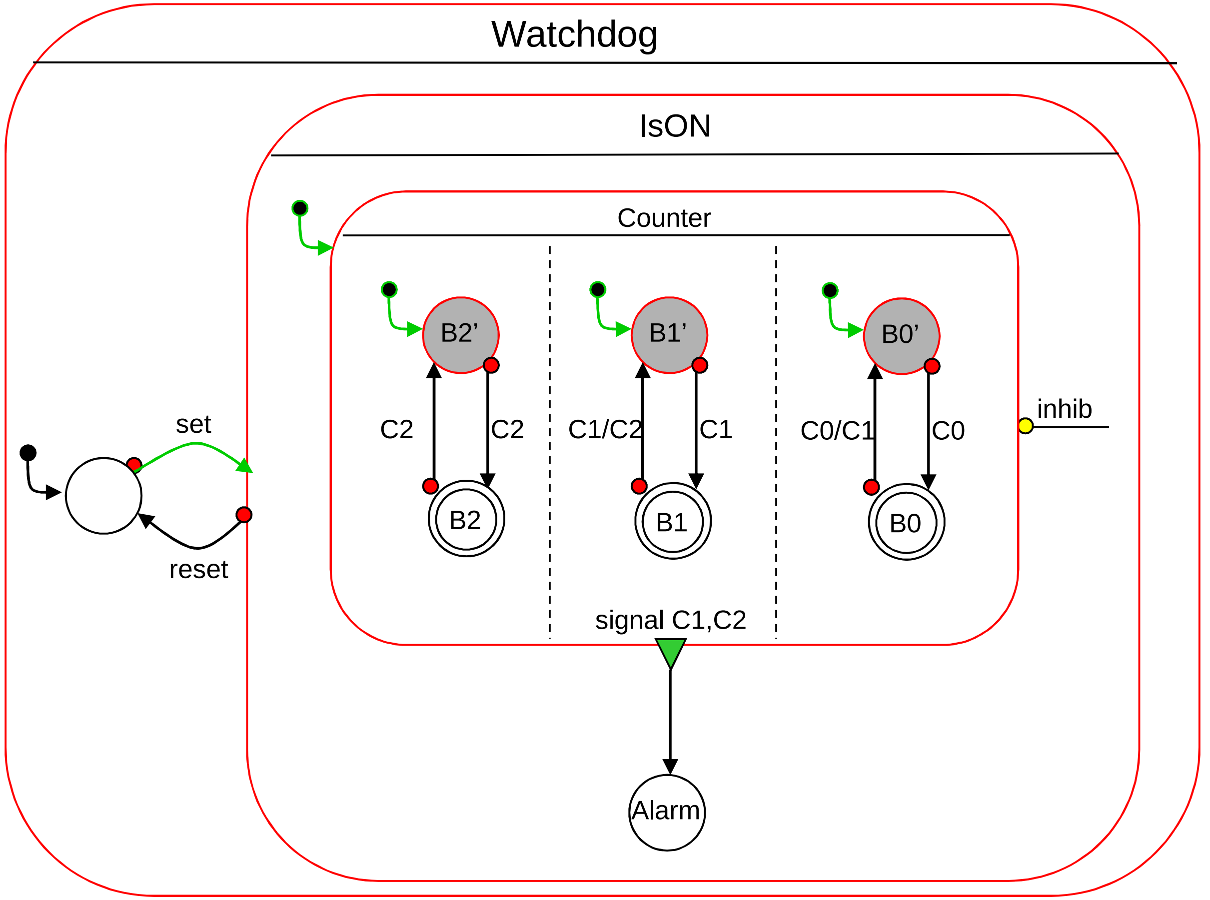


Example: Watchdog

reset^{+}



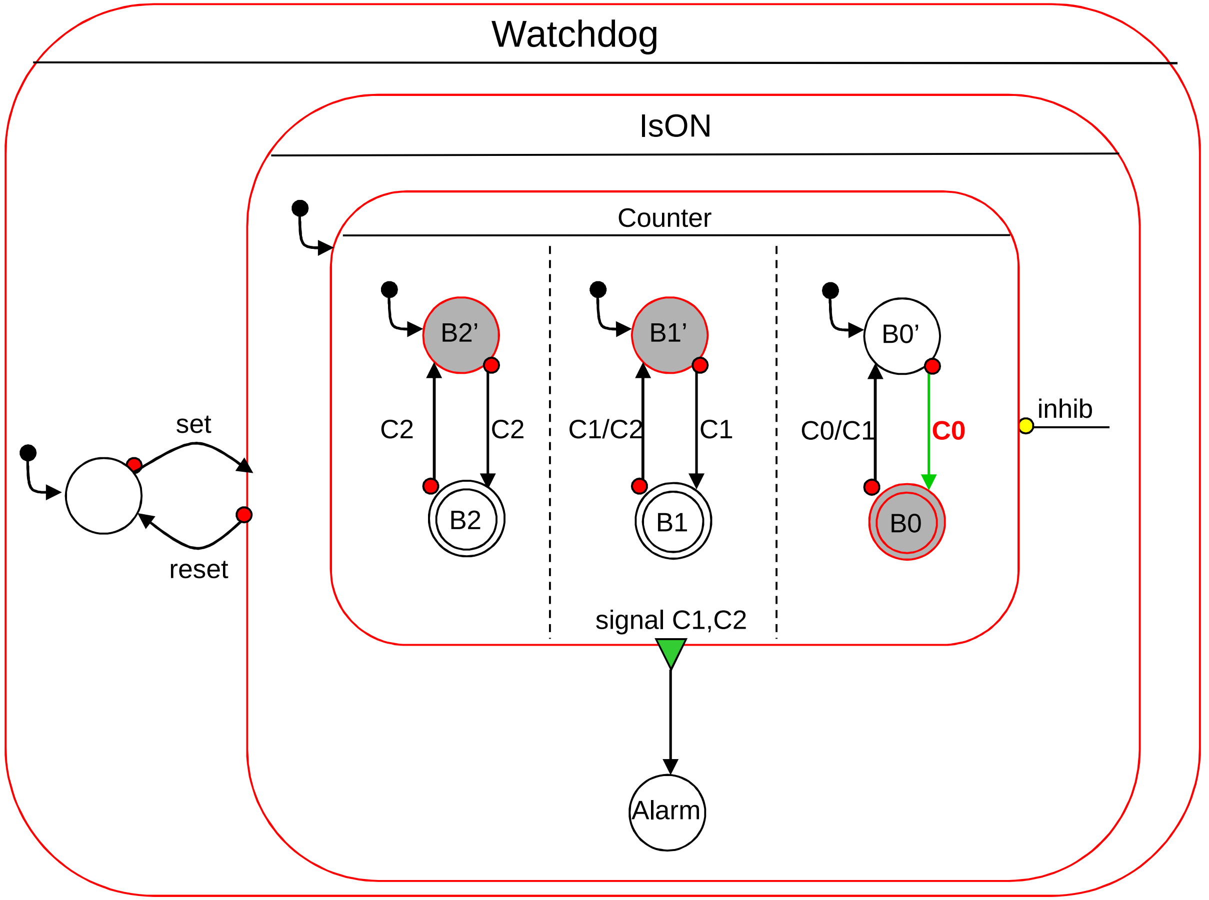
Example: Watchdog



set⁺

Example: Watchdog

$C0^+$

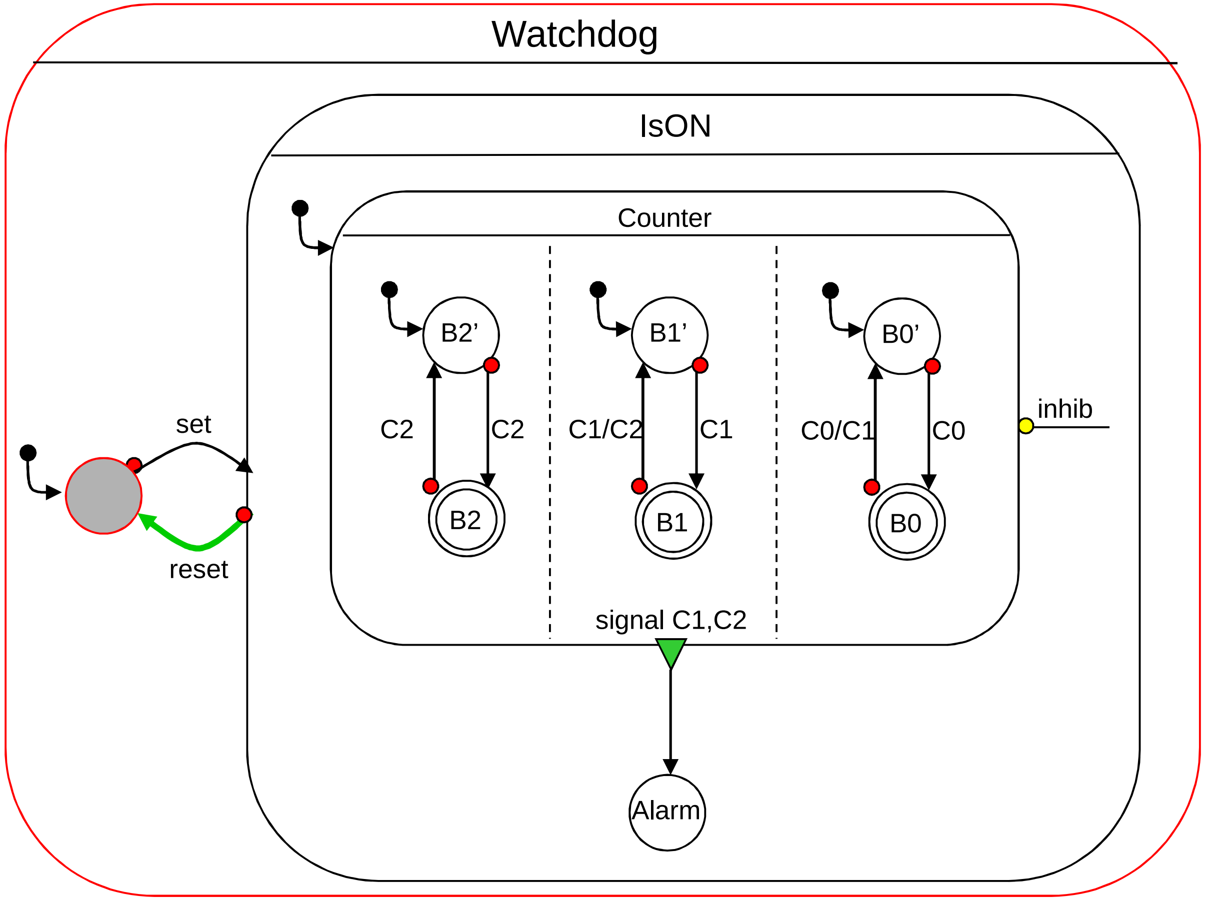


Example: Watchdog

reset$^+$

inhib$^+$

C0$^+$

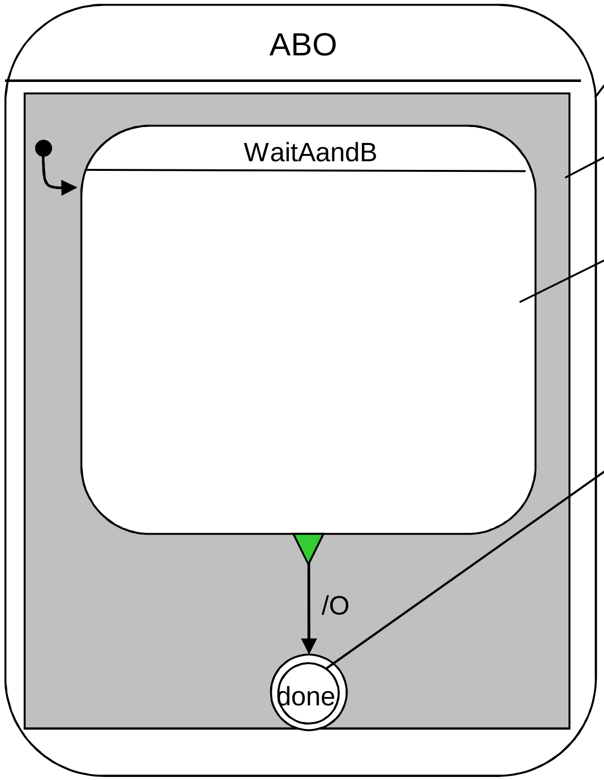


Computing a Reaction – Definitions

- Each SSM can be represented by unique macrostate, called *Top*, which designates the root of the state containment hierarchy.

- A state-transition graph (STG) *G* is a tuple *G = (S, ini)* where *S* is a non-empty set of reactive cells, and *ini* is the initial reactive cell.
- A reactive cell is a tuple *C=(B,R,S)* such that its body *B* is a state (simple state or macrostate) and *R* the set of all its outgoing transitions. The status *S* of a reactive cell is either *IDLE* or *ACTIVE*.
- A macrostate *M* is a quadruple *M=(G,I,O,L)* composed from a non-empty set *G* of STGs, and three possibly empty sets of signals: input signals (*I*), output signals (*O*), local signals (*L*).
- A transition has a destination and a label. The destination is a reactive cell, the label is composed of three optional fields: a type, a trigger, an effect. A transition is denoted as a quadruple *R=<type, trigger, effect, targetID>.* Feasible types are *sA* (strong abort), *wA* (weak abort), *nT* (normal termination).

Illustration

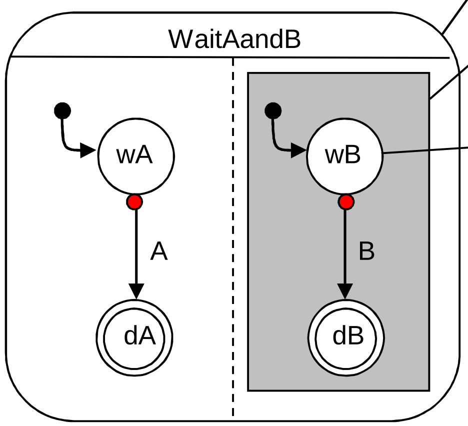


A macrostate made of 1 STG.

STG made of 2 reactive cells.

Initial reactive cell: Body is macrostate, 1 transition

Reactive cell whose body is simple state.

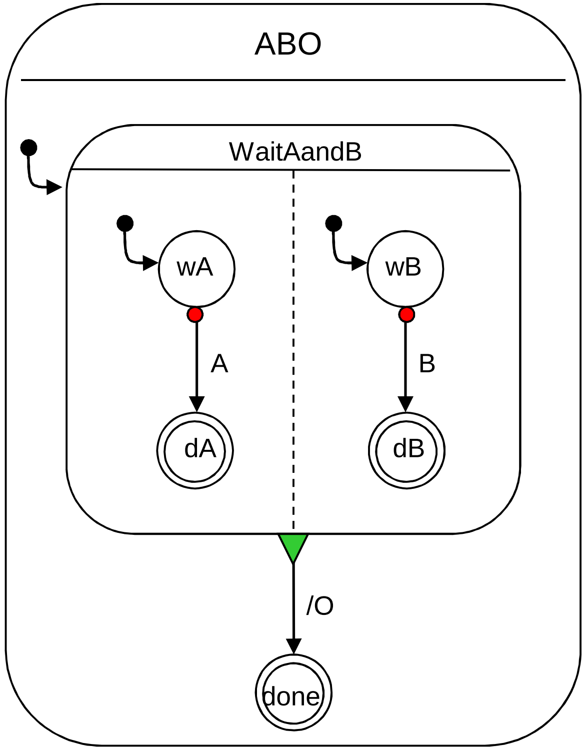


Macrostate made of 2 STGs.

STG made of 2 reactive cells.

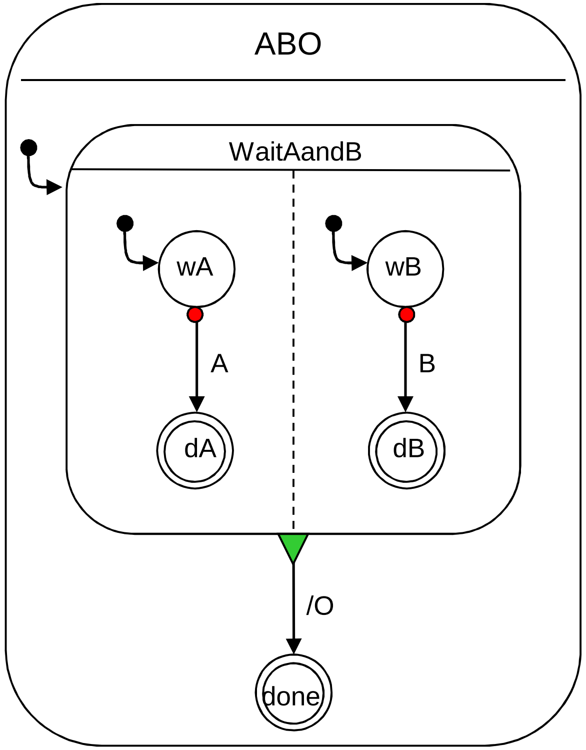
Initial reactive cell: Body is simple state, 1 transition

Detailed Example



- Macrostate ABO=TOP:
    - $M_{ABO}.G = \{ABO.g\}$
    - $M_{ABO}.I=\{A,B\}$
    - $M_{ABO}.O=\{O\}$
    - $M_{ABO}.L= \emptyset$
- State-Transition Graph ABO.g:
    - $ABO.g.S = \{RC_{WaitAandB}, RC_{done}\}$
    - $ABO.g.ini = RC_{WaitAandB}$
- Reactive cell $RC_{WaitAandB}$
    - Body: Macrostate $M_{WaitAandB}$
    - $RC_{WaitAandB}.out=\{<nT,,O,RC_{done}>\}$
- Macrostate $M_{WaitAandB}$:
    - $M_{WaitAandB}.G=\{WaitAandB.g_1, WaitAandB.g_2\}$
    - $M_{WaitAandB}.I=\{A,B\}$
    - $M_{WaitAandB}.O= \emptyset$
    - $M_{WaitAandB}.L= \emptyset$
- Reactive cell RC_{done}:
    - Body: Simple state S_{done}
    - $RC_{done}.out= \emptyset$

Detailed Example

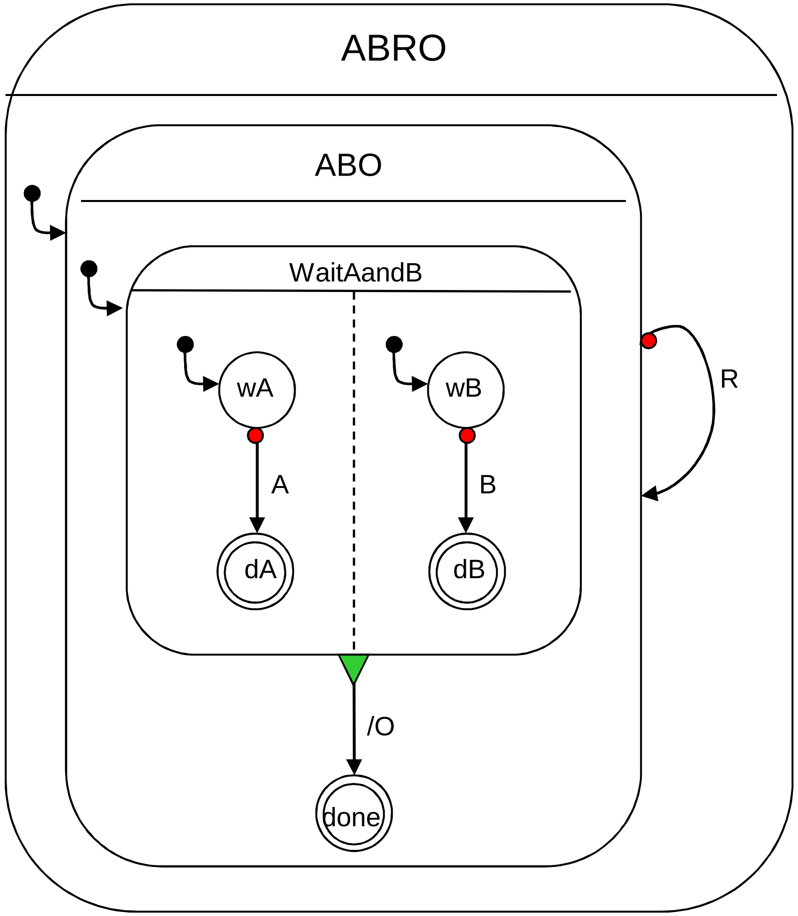


- State-Transition Graph WaitAandB.g_1
  - WaitAandB.g_1.S={RC_{wA},RC_{dA}}
  - WaitAandB.g_1.ini=RC_{wA}
- State-Transition Graph WaitAandB.g_2
  - WaitAandB.g_2.S={RC_{wB},RC_{dB}}
  - WaitAandB.g_2.ini=RC_{wB}
- Reactive cell RC_{wA}
  - Body: simple state S_{wA}
  - RC_{wA}.out = {<sA,A,,dA>}
- Reactive cell RC_{dA}
  - Body: simple state S_{dA}
  - RC_{dA}.out = Ø
- Reactive cell RC_{wB}
  - Body: simple state S_{wB}
  - RC_{wB}.out = {<sA,B,,dB>}
- Reactive cell RC_{dB}
  - Body: simple state S_{dB}
  - RC_{dB}.out = Ø

Configurations

- A configuration is a maximal set of states (macrostates or simple states) the system could be in simultaneously. (Note that formally status is associated with reactive cells.)
- Let T be the top macrostate associated with an SSM. A legal configuration C for T must satisfy the following rules:
  1. T in C
  2. If a macrostate M is in C, then C must also contain for each STG G directly contained in M exactly one state directly contained in G.
  3. C is maximal and contains only states satisfying rules 1 and 2.
- A stable configuration is a legal configuration that the SSM can reach after a sequence of reactions. Only the stable configurations are of interest for the user.

Example

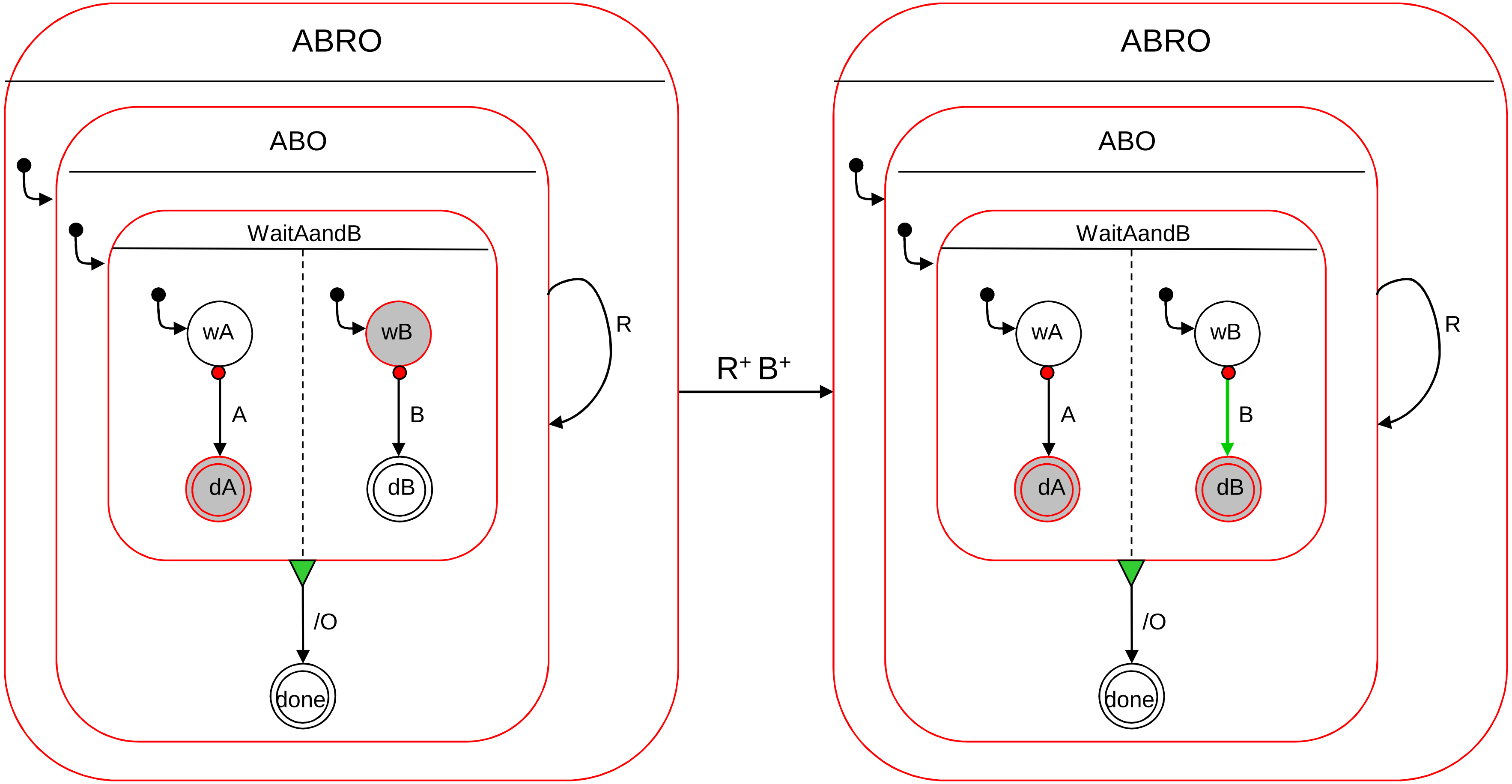


- Legal configurations:
  - {ABRO,ABO,done}
  - {ABRO,ABO,WaitAandB,wA,wB}
  - {ABRO,ABO,WaitAandB,wA,dB}
  - {ABRO,ABO,WaitAandB,dA,wB}
  - {ABRO,ABO,WaitAandB,dA,dB}
- Stable configurations:
  - all legal configurations except {ABRO,ABO,WaitAandB,dA,dB}

Computing a reaction

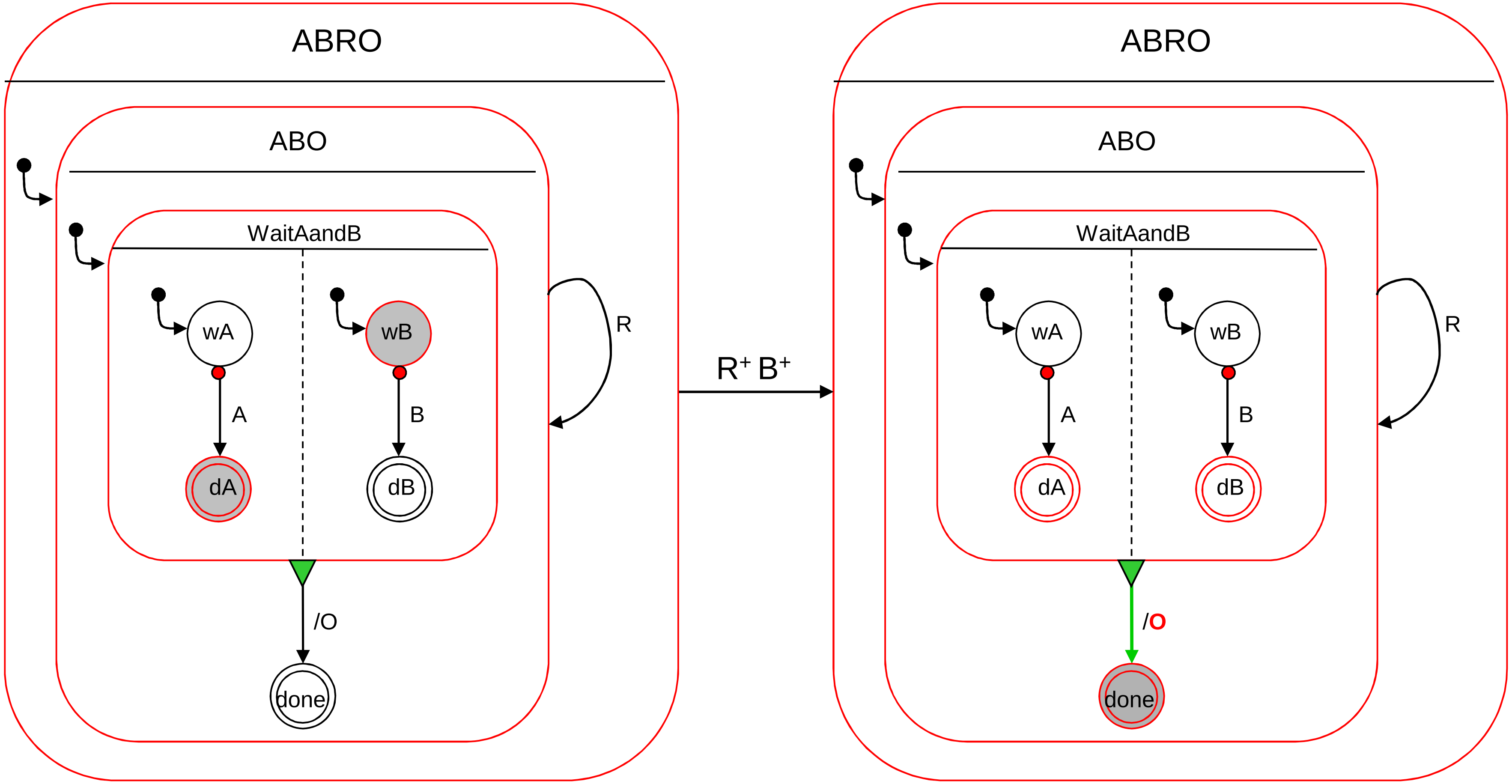
- Computing a reaction is done by concurrent threads which suspend their execution when a trigger cannot be evaluated and can resume when new signal statuses are broadcast.
- Reactions are computed as a sequence of microsteps, all executed during the same instant but in the order that respects causality.
- A transition is taken (ie a microstep is executed) only when its trigger is surely satisfied (no possibility of backtracking).
- Termination codes of components (reactive cell, STG, macrostate, simple state):
  - DONE: execution has been terminated, but can go on in same instant
  - PAUSE: nothing left to do until next instant
  - DEAD: nothing left to do at the current instant and in the future (final state); component is candidate to join a normal termination.
  - Partial order: DEAD < PAUSE

Concurrency and Weak Abort



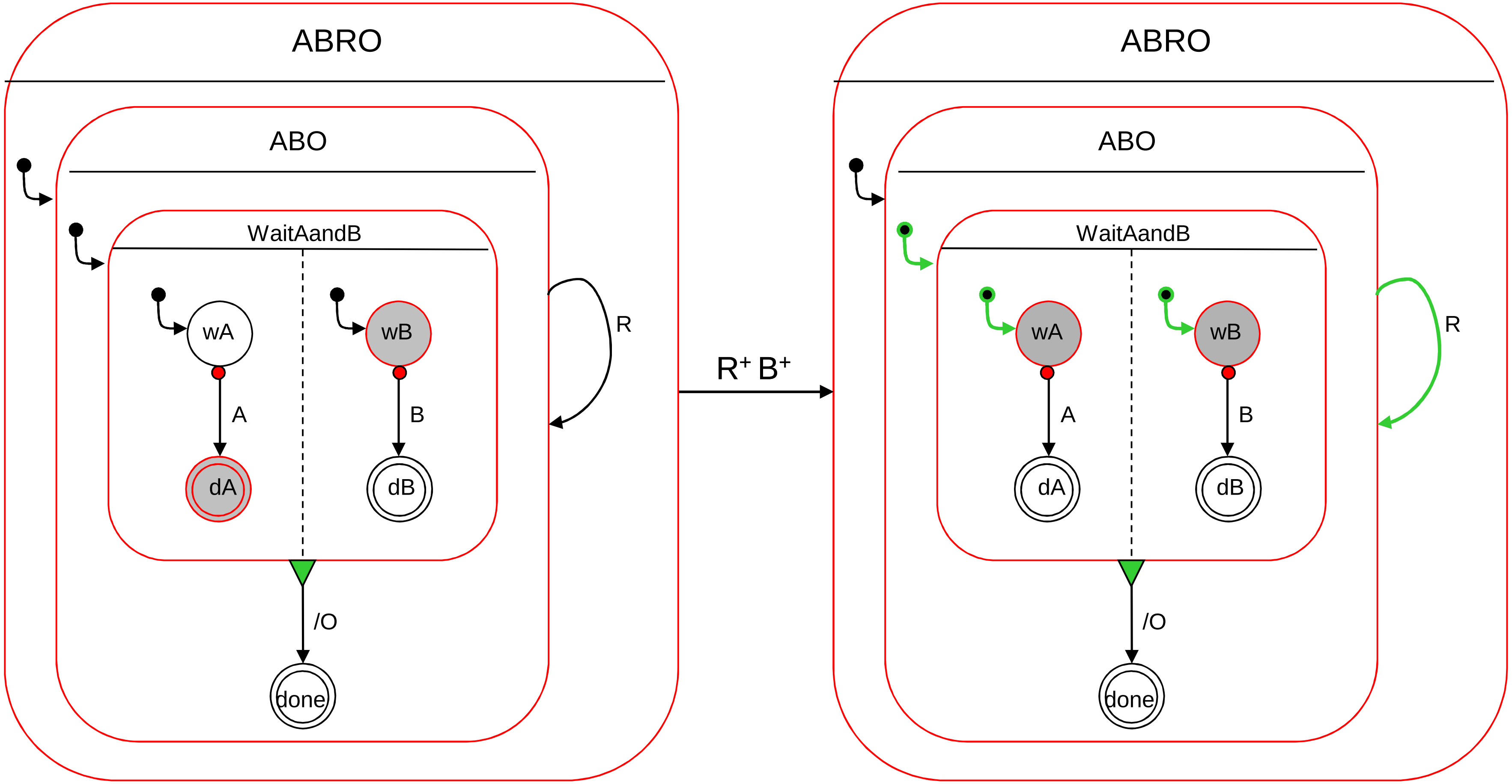
Microstep 1

Concurrency and Weak Abort



Microstep 2

Concurrency and Weak Abort



Microstep 3

Computing the reaction of an SSM

- Reaction of an SSM. Given a stable configuration, a reaction is computed by:
  1. Read input signals (presence status of all input signals is known)
  2. Set all output signals to the unknown presence status (⊥)
  3. Compute reaction of the top macrostate:
     react(T)
     /* yields emitted signals and the next stable configuration */

State Reaction

- Reaction of macrostate M:
  1. Set all local signals to ⊥
  2. For each STG g directly contained in M.G do in parallel
     - Compute reaction of STG g and store the termination code in c(g): c(g) = react(g)
  3. When all parallel executions are done
     - Compute C=maximum of c(g) for all STGs g in M
  4. Return C

- Reaction of a simple state S
  1. If S is a final state return DEAD.
  2. If an effect is associated with S, then emit all signals of the effect.
  3. return PAUSE

State Transition Graph Reaction

- Reaction of a STG g
  1. If there is no current state in g then set current to the initial reactive cell: g.current = g.ini;
  2. Compute the reaction of the reactive cell C=(M,t) whose body M is the current state (M=g.current):
     r = react(C)
  3. If r =DONE then G.current = C.nextState; goto 2.
  4. return r /* here r cannot be DONE */

- Comments:
  - When entering a macrostate the current state of each STG is undefined. If the STG is already active, the current state is the (unique) currently active state.
  - Reactions of all STGs from a macrostate are computed in parallel (fork).

Reactive Cell Reaction

- Reaction of a reactive cell C:
  1. if (!firstInstant) Strong abort test:
     - If a strong abort transition is enabled then take this transition
  2. Execute the body of the reactive cell
     - If it is a macrostate M, then recursive call: B = react(M)
     - If it is a simple state S, then terminal call: B = react(S)
  3. if (!firstInstant) Weak abort test:
     - /* Note: body has completed execution. */
     - If a weak abort transition is enabled then take this transition
  4. Normal termination test:
     - If (B == DEAD) then take normal termination transition
  5. End of reaction:
     - Set C.status=ACTIVE
     - return PAUSE

Reactive Cell Reaction

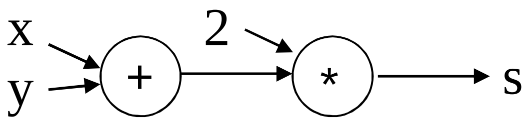
- The triggers are not tested at the first instant when the reactive cell is activated (strong/weak abort test).
- If the presence status of a triggering signal is unknown, the execution is suspended till another concurrent execution thread will fix the status of the tested signal.
- Taking a transition t (strong/weak abort, or normal termination) means:
  - Recursively "kill" the body of the reactive cell C:
    - set k.status=IDLE for all transitively contained reactive cells k
    - reset G.current for all transitively contained STGs G
  - Execute the effect associated with t and set C.nextstate = t.target;
  - Set C.status = IDLE;
  - return DONE;

Naming Confusion

- Scade Suite: model based development IDE
  - Graphical modelling language
    - ✓Control part: SyncCharts (SSM)
    - Data flow part: SCADE graphical syntax
  - Textual representation: SCADE textual language. Semantics comprise
    - constructive semantics of ESTEREL
    - data flow semantics of LUSTRE

LUSTRE

- Programs are structured into nodes:
  - Node: subprogram defining its output parameters as functions of its input parameters.
  - Definition given by unordered set of equations (→ declarative language)
- Based on synchronous data-flow model:
  - Functional: no side effects.
  - All nodes work simultaneously, ie at the same speed.
  - No broadcasting of signals; sequencing and synchronization only from data dependences.
  - Each variable takes a value at every cycle of the program.



At any cycle n:
$s_n = 2*(x_n+y_n)$

  - Basis of SCADE.
- Example:

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                                else pre(count)+incr;
tel
```

Required Properties

- Causality: The output at any instant *t* may only depend upon input received before or at *t*.
- Bounded memory: There must be a finite bound such that, at each instant the number of past input values necessary to produce a new output value remains smaller than that bound.
- Efficient code generation.
- Execution time predictability: no unbounded loops, no recursion.

Flows and Clocks

- Any variable and expression denotes a flow, ie a pair made (x, b_x) of
  - a possibly infinite sequence x of values of a given type
  - a clock b_x, representing a sequence of times.
- x is defined at instant i iff $b_x(i) = true$.
- A flow takes its n-th value in the n-th time of its clock.
- Input variables are defined at every instant: their clock is called the basic clock.
- Example: Let x run on the basic clock C, y on a slower clock. This gives the following time scales:

Basic time-scale	1	2	3	4	5	6	7	8
b_x	t	t	t	t	t	t	t	t
x time-scale	1	2	3	4	5	6	7	8
b_y	t	f	t	f	t	f	t	f
y time-scale	1		2		3		4	

Types, Equations, Assertions

- Variables are declared with their type:
  - Basic types: boolean, integer, real.
  - Type constructor tuple.
    - Semantics is Cartesian product.
  - Abstract types via import.
- Equations:
  - Variables are defined via equations, e.g. X=E with variable X and expression E.
  - Substitution principle: X can be substituted to E anywhere in the program and vice versa.
  - Definition principle: The behavior of X must be completely specified by this equation.
- Assertions:
  - Assertions assert(E): E must hold during execution.
  - Used to optimize code generation, for simulation and for verification.

Variables and Expressions

- Operators only operate on operands sharing the same clock.
- As variables and expressions are streams, operators also produce streams. Example: With x = (0,1,2,3,4,...) and y = (2,4,6,8,10,...): x+y=(2,5,8,11,14,...)
- Expressions are build from variables, constants and operators.
- Three types of operators:
  - Data operators:
    - arithmetic, boolean and relational expressions
    - conditional expressions: *if E then X else Y*
  - Imported operators:
    - functions imported from host language
  - Temporal (sequence) operators.

Temporal Operators

- 'previous' operator pre:
  - $(pre(E))_0 = \perp$ (undefined, also denoted nil)
  - $(pre(E))_n = E_{n-1}$
- 'followed by' operator ->
  - $(E\text{->}F)_0 = E_0$
  - $(E\text{->}F)_n = F_n$
- (Down-)Sampling: when
  - Let E be an expression and B a boolean expression with the same clock: (E when B) is the sequence of values of E when B is true.
- Upsampling/Interpolation/Projection: current
  - Let E be an expression and B a boolean expression defining the clock of E: Then *current E* has the same clock as B; and *(current E)* is the sequence of values of E at the last time when B was true.

Example

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y = X when B								
Z = current Y								

Example

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y = X when B		x_2		x_4			x_7	x_8
Z = current Y								

Note: `gaps´ are not filled.

Example

B	false	true	false	true	false	false	true	true
X	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
Y = X when B		x_2		x_4			x_7	x_8
Z = current Y	$\perp$	x_2	x_2	x_4	x_4	x_4	x_7	x_8

Clock Rules

- Let a clock environment ω be a function from identifiers to clocks.
- Let $clk(E,\omega)$ be the clock of the expression E in the environment ω.
- For an equation $X=E$ holds $\omega(X)=clk(E,\omega)$.
- Let $\perp$ be the undefined clock and T the erroneous clock. Then

$$ck \leq ck' \Leftrightarrow (ck = \perp \vee ck' = \mathrm{T} \vee ck = ck')$$

- Let $\cup$ denote the least upper bound operator.
- Constants: For any constant k, $clk(k,\omega)=true$ (the basic clock).
- Variables: For any identifier X, $clk(X, \omega)= \omega(X)$.
- Synchronous operators:

$$clk(op(E_{1,}E_{2,}\ldots,E_n),\varpi) = \bigcup_{i=1}^{n} clk(E_i,\varpi)$$

Clock Rules

- Downsampling: The operands of the *when* operator must be on the same clock: $clk(E)=clk(ck) \Leftrightarrow clk(E\ when\ ck,\omega)=ck$.
- Upsampling: Let ck be the clock of E, $ck \neq true$: $clk(current(E),\omega)=clk(ck)=clk(clk(E))$.
- Clock of a node instance: clock of its effective inputs.
- Initialization problem: $current(X\ when\ C)$ exists but is undefined ($\perp$) until C becomes *true* for the first time.
- Solution: activation conditions
  - Not an operator, rather a macro.
  - $y = CONDACT(OP, C, X, dflt)$ equivalent to
    $Y = if\ C\ then\ current(OP(X\ when\ C))$
    $else\ (dflt \text{ -> } pre(Y))$
  - Provided by SCADE (not part of LUSTRE).

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)							
e = (1->not pre(e))							
n when e							
current(n when e)							
Counter((1,1,false) when C)							
Counter(1,1,false) when C							

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                            else pre(count)+incr;
tel
```

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)	0	1	2	3	4	5	6
e = (1->not pre(e))							
n when e							
current(n when e)							
Counter((1,1,false) when C)							
Counter(1,1,false) when C							

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                        else pre(count)+incr;
tel
```

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)	0	1	2	3	4	5	6
e = (1->not pre(e))	1	0	1	0	1	0	1
n when e							
current(n when e)							
Counter((1,1,false) when C)							
Counter(1,1,false) when C							

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                         else pre(count)+incr;
tel
```

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)	0	1	2	3	4	5	6
e = (1->not pre(e))	1	0	1	0	1	0	1
n when e	0		2		4		6
current(n when e)							
Counter((1,1,false) when C)							
Counter(1,1,false) when C							

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                                else pre(count)+incr;
tel
```

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)	0	1	2	3	4	5	6
e = (1->not pre(e))	1	0	1	0	1	0	1
n when e	0		2		4		6
current(n when e)	0	0	2	2	4	4	6
Counter((1,1,false) when C)							
Counter(1,1,false) when C							

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                         else pre(count)+incr;
tel
```

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)	0	1	2	3	4	5	6
e = (1->not pre(e))	1	0	1	0	1	0	1
n when e	0		2		4		6
current(n when e)	0	0	2	2	4	4	6
Counter((1,1,false) when C)	1	2			3		4
Counter(1,1,false) when C							

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                        else pre(count)+incr;
tel
```

Example

C	true	true	false	false	true	false	true
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
n=(0->pre(n)+1)	0	1	2	3	4	5	6
e = (1->not pre(e))	1	0	1	0	1	0	1
n when e	0		2		4		6
current(n when e)	0	0	2	2	4	4	6
Counter((1,1,false) when C)	1	2			3		4
Counter(1,1,false) when C	1	2			5		7

```
node Counter (init, incr: int; reset: bool)
        returns (count:int);
let
        count = init -> if reset then init
                                else pre(count)+incr;
tel
```

Program Structure

- Nodes are LUSTRE subprograms. General structure:

  $\mathit{node}\ N\ (\ x_1\!:\tau_1;\ x_2\!:\tau_2;\ \ldots;\ x_p\!:\tau_p\ )$
  $\mathit{returns}\ (\ y_1\!:\theta_1;\ y_2\!:\theta_2;\ \ldots;\ y_p\!:\theta_q\ )$
  $\mathit{var}\ z_1\!:\gamma_1\ ;\ z_2\!:\gamma_2\ ;\ \ldots;\ z_k\!:\gamma_k$
  let
  $\quad z_1=E_1;\ \ldots;\ z_k=E_i\ ;$
  $\quad y_1=E_j;\ \ldots;\ y_p=E_m\ ;$
  tel

- Node instantiation: if N is the name of a node with above signature and if $E_1,\ldots,E_p$ are expressions of type $\tau_1,\ldots,\tau_p$, then $N(E_1,\ldots,E_p)$ is an expression of type $\mathit{tuple}(\theta_1,\ldots,\ \theta_q)$.
- Conditional and sequence operators are polymorphic and can be applied to tuples.

Arrays and Recursion

- Let n be a constant. Given type τ, τ_n defines an array with n entries of type τ.
- Example: $x{:}bool_n$
- The bounds of an array must be known at compile time; the compiler transforms an array of n values into n different variables.
- $X[i]$ denotes i th element.
- $X[i..j]$ denotes the array made of elements i to j of X.

- LUSTRE only allows static recursion: the recursion is completely unrolled.
- Attention: if the recursion is not bounded the compiler will not stop.

Compilation of LUSTRE Programs

- Static compiler checks:
  - Definition checking: any local and output variable must have exactly one definition.
  - No recursive node calls.
  - Clock consistency.
  - Absence of uninitialized expressions (yielding $\perp$).
  - Absence of cyclic definitions.

- Compilation to
  - single-loop code
  - automata code.

Causality Problems

- LUSTRE only allows acyclic equation systems.
  Note: acyclic equations have a unique solution.
- $X{=}E$ is acyclic if X does not occur in E unless as subterm of the pre operator.
- Examples:
  - $X = X \mathit{\ and\ pre}(X)$ is cyclic
  - $X = Y \mathit{\ and\ pre}(X)$ is acyclic

- Also structural deadlocks which are not true ones are rejected:
  - $X = \mathit{if}\ C\ \mathit{then}\ Y\ \mathit{else}\ Z;$
    $Y = \mathit{if}\ C\ \mathit{then}\ Z\ \mathit{else}\ X;$
- Improved causality analysis in SCADE.

Clock Consistency

- Consider the following (illegal) example:

```
b = true -> not pre b;
y = x + (x when b);
```

x	x_0	x_1	x_2	x_3
b	1	0	1	0
x when b	x_0		x_2	
x+(x when b)	$x_0 + x_0$	$x_1 + x_2$	$x_2 + x_4$	$x_3 + x_6$

- The computation of the $2n$th value of y needs the $2n$th and the nth values of x.
- Problem: not possible with bounded memory.
- Consequence: only streams of the same clock can be combined.
- Problem: undecidable whether two boolean expressions denote the same flow.

Clock Consistency (c'ed)

- Thus: two boolean expressions define the same clock iff they can be syntactically unified.
- Examples:

  $x = a \textit{ when } (y>z)$

  $y = b+c$

  $u = d \textit{ when } (b+c>z)$

  $v = e \textit{ when } (z<y)$

- x and u share the same clock.
- x and v have different clocks.

LUSTRE and SCADE

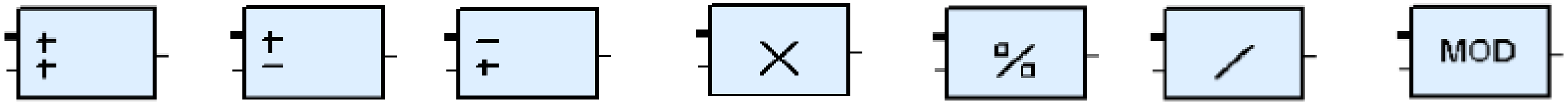
- SCADE: constant blocks:
  - delimited by keywords *let const* and *tel*;
  - Example:

```
let const PCSt1
    C2: [real,int] = [C1,3]
    C1: real = 7.2;
    imported Cimp: real;
tel;
```

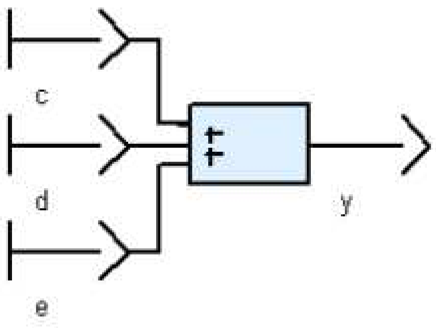
- Equation blocks delimited by keywords *let equa* and *tel*; variable blocks by *var*; global variable blocks by *let global* and *tel*; type blocks by *let type* and *tel*;
- Other additions for syntactic convenience, like “don’t care” symbol ‘_’: *_,x = [3,4];*
- For details see *SCADE Language Reference Manual* [Esterel 2006].

SCADE: The Graphical Language

- See SCADE Technical Manual, Chapter 3: SCADE Language Graphic Syntax.
- Arithmetic Operators:
    - "+" "-" "*" intdiv realdiv mod ...



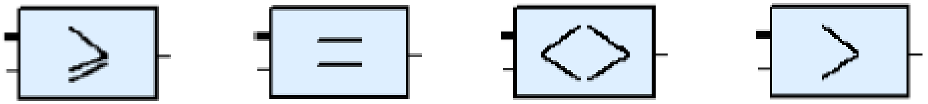
- Example: y=c+d+e



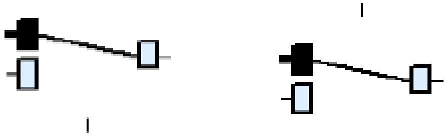
SCADE: The Graphical Language

- Logial Operators:
  - “or” “xor” “and” “not” ...

- Some Comparison Operators:

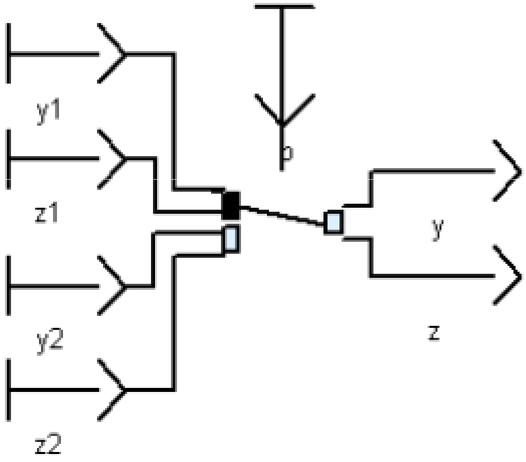


- Control Operators:
  - if ... then ... else ...



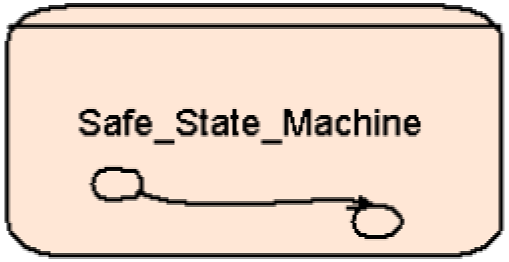
SCADE: The Graphical Language

- Example:
  - y,z=if b then (y1,z1) else (y2,z2)



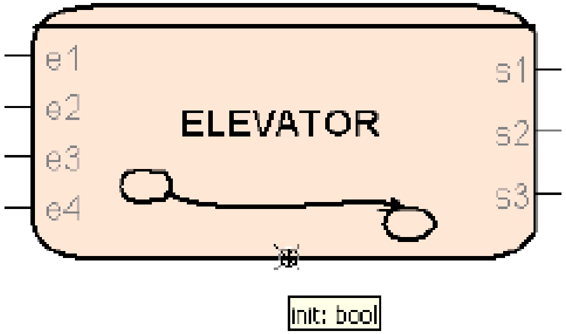
SCADE: The Graphical Language

- SSM (SyncCharts) Nodes



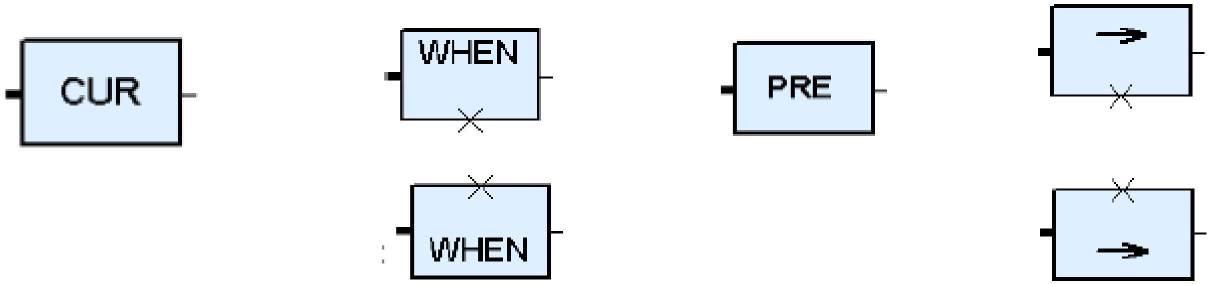
Call of the SSM node ELEVATOR with 4 inputs and 3 outputs:

- ELEVATOR (e1 : int ; e2 : int ; e3 : int ; e4 : int ; hidden input_init : bool) returns (s1 : bool ; s2 : bool ; s3 : bool)



SCADE: The Graphical Language

- Temporal Operators



- Example: o1,o2,o3 = (i1,i2,i3) when c