System Design and Methodology / Embedded Systems Design

VI. Finite State Machines

TDTS07/TDDI08
VT 2026

Ahmed Rezine

(Based on material by Petru Eles and Soheil Samii)

Institutionen för datavetenskap (IDA)
Linköpings universitet

SYNCHRONOUS FSMs & SYNCHRONOUS LANGUAGES

1. FSM and Extended FSM models
2. The State Explosion Problem
3. Hierarchical Concurrent FSMs
4. Time and Synchrony
5. Synchronous/Reactive Languages
6. How to Implement a Synchronous System? Problems.

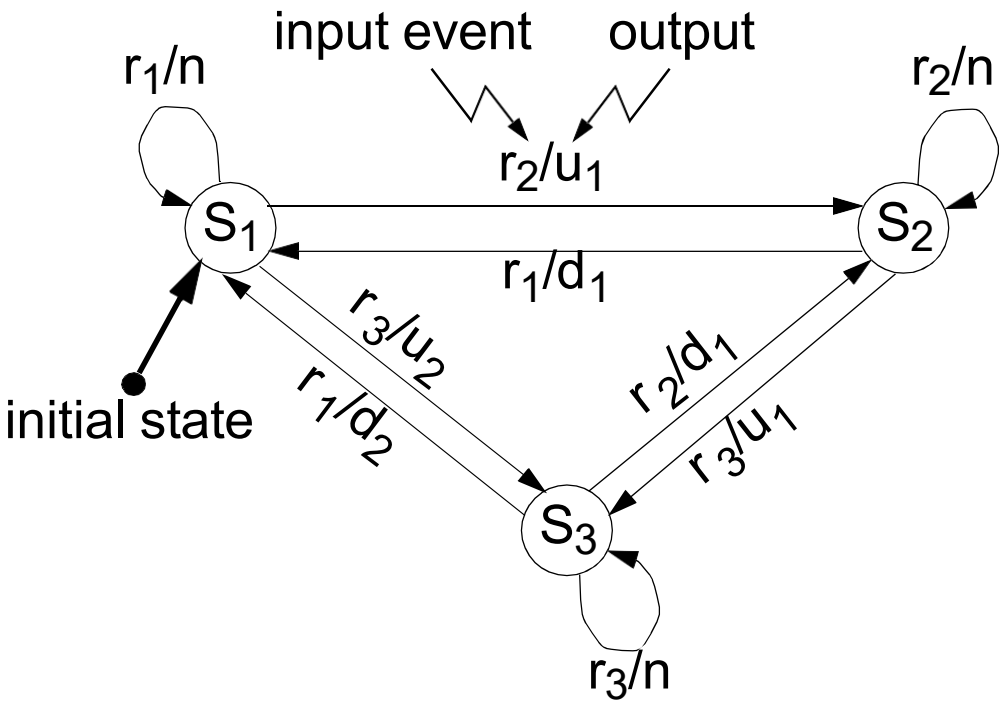
Finite State Machines

- The system is characterised by *explicitly* depicting its states as well as the transitions from one state to another.
- One particular state is specified as the initial one
- States and transitions are in a finite number.
- Transitions are triggered by input events.
- Transitions generate outputs.
- FSMs are suitable for modeling control dominated reactive systems (react on inputs with specific outputs)

FSM Example-1

Elevator controller

- Input events: {r_1, r_2, r_3}
  - r_i: request from floor i.
- Outputs: {d_2, d_1, n, u_1, u_2}
  - d_i: go down *i* floors
  - u_i: go up *i* floors
  - *n*: stay idle
- States: {S_1, S_2, S_3}
  - S_i: elevator is at floor *i*.



Extended Finite State Machines

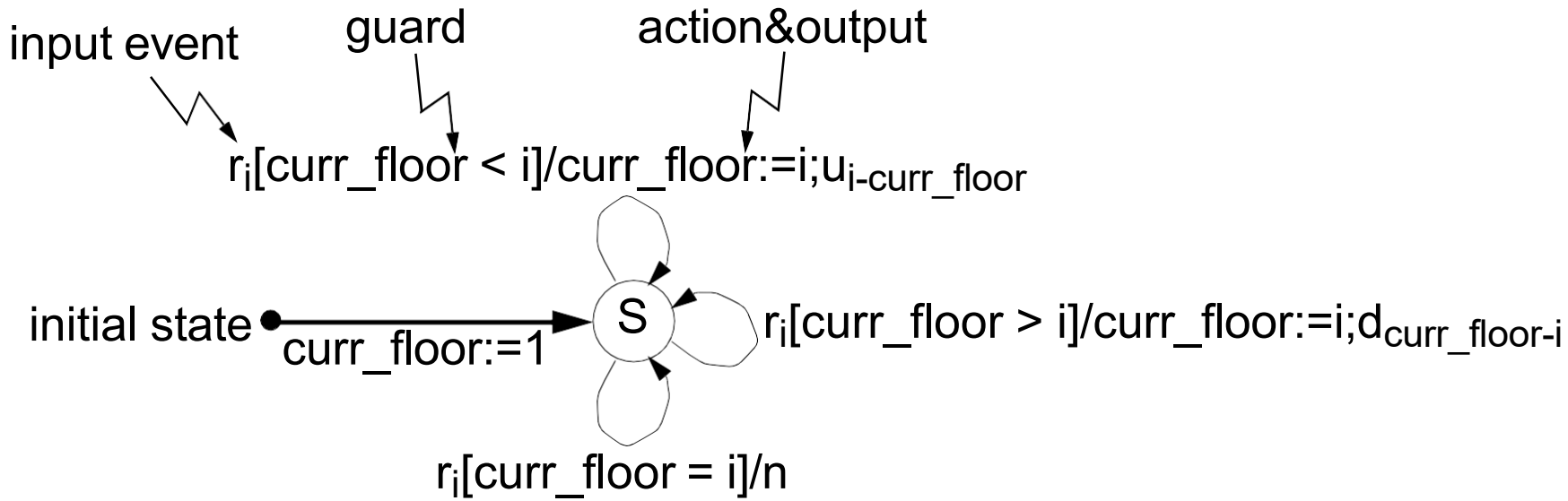
- Variables can be associated to the FSM.
  - Changes to variables specified as actions associated to transitions.
  - Extended FSMs are suitable for systems which are both control and computation intensive.

- *Guards* (expressed as conditions) may be specified for transitions: The transition is performed when the associated event(s) occur and if the associated guard is true

FSM Example-1 Modified

Elevator controller with extended FSM

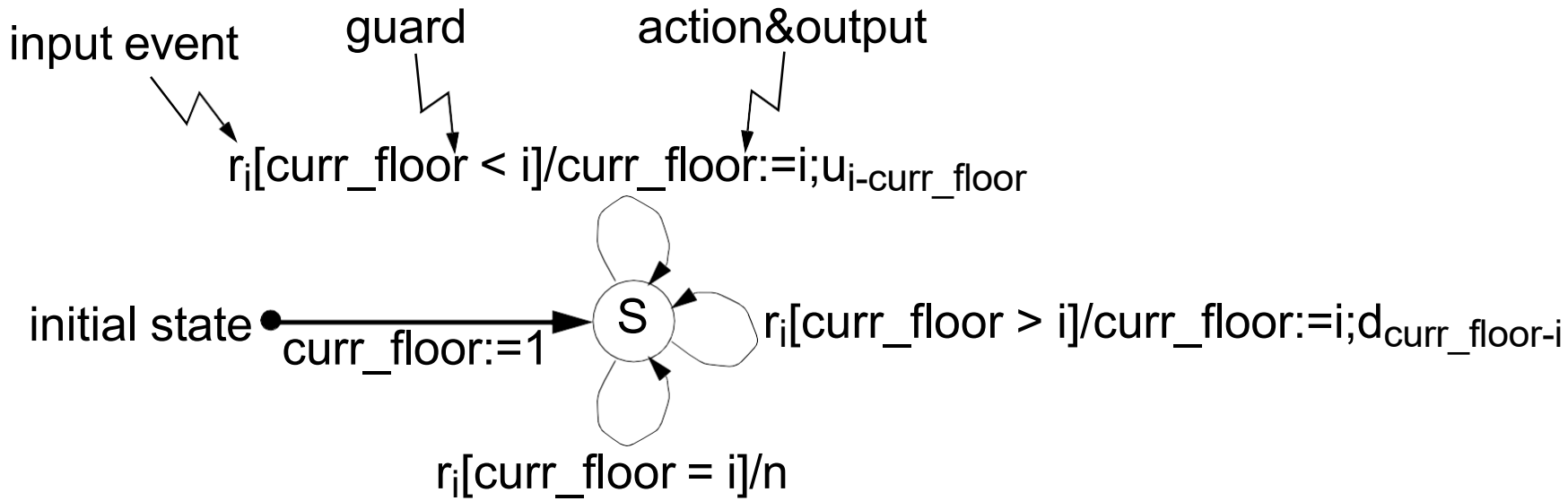
- We associate to the FSM a variable storing the current floor.



FSM Example-1 Modified

Elevator controller with extended FSM

- We associate to the FSM a variable storing the current floor.

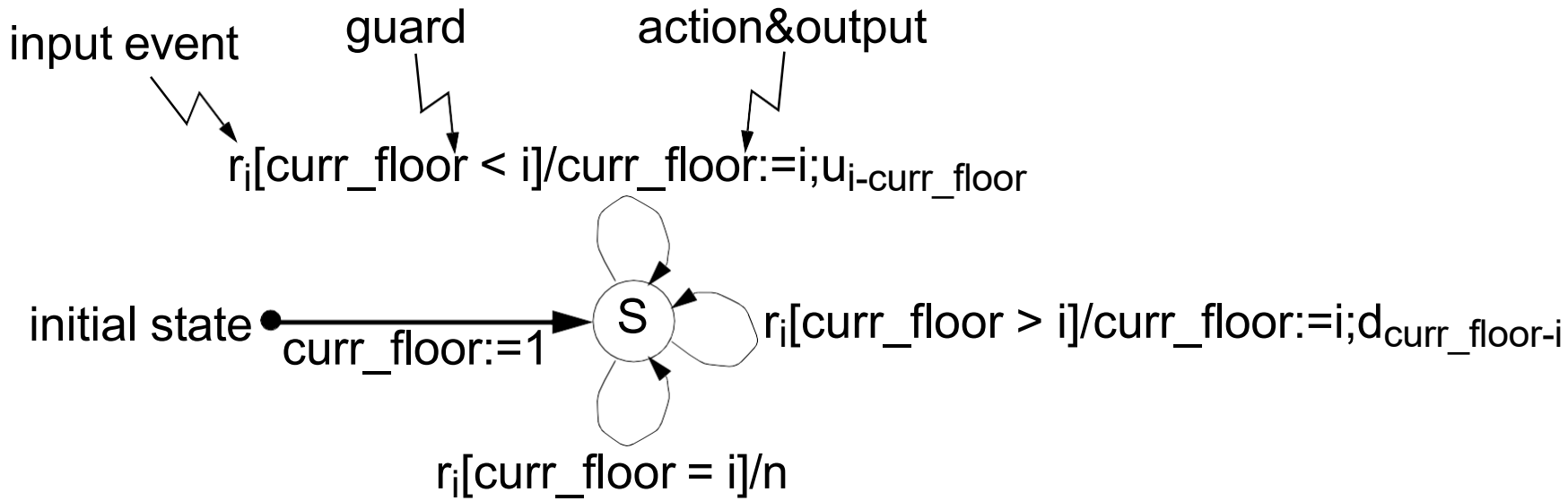


- You might wonder: *Do we really have one single state of the system?*

FSM Example-1 Modified

Elevator controller with extended FSM

- We associate to the FSM a variable storing the current floor.



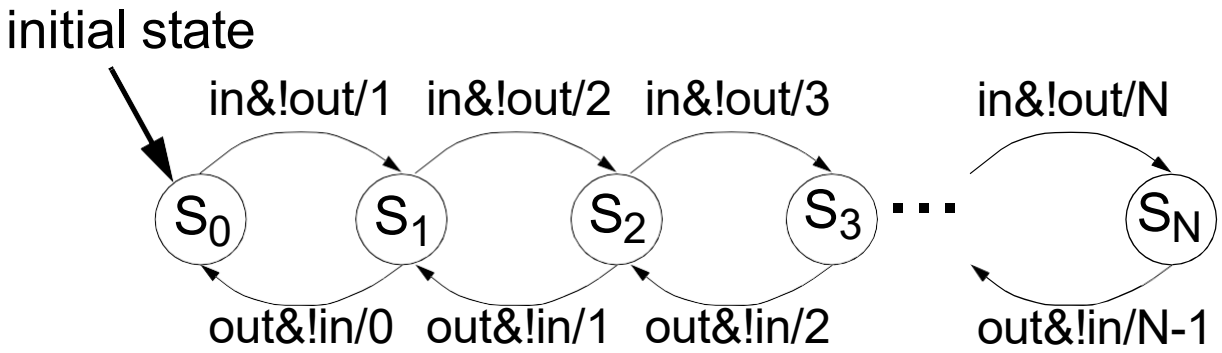
- You might wonder: *Do we really have one single state of the system? Of course not!*

  The global system state is now encoded in the FSM state and the value of the associated variable.

FSM Example-2

Parking counter

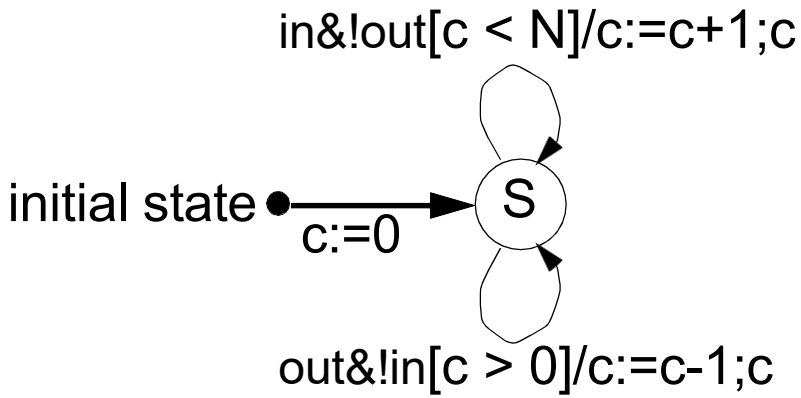
- Input events: {in, out}
  - *in*: car enters;
  - *out*: car leaves.
- Outputs: {1, 2, 3, ... N}
  - *i: display value i*
- States: {S_0, S_1, S_2, ... S_N}
  - S_i: *i* cars in the parking.



FSM Example-2

Parking counter with extended FSM

- We associate to the FSM a variable *c* storing the number of cars.



State Explosion

- Complex systems tend to have very large number of states. This particularly is the case in the presence of concurrency.
  The phenomenon is called *state explosion*.

- Every global state of a concurrent system must be represented individually
  $\Rightarrow$ interleaving of independent actions leads to exponential number of states.

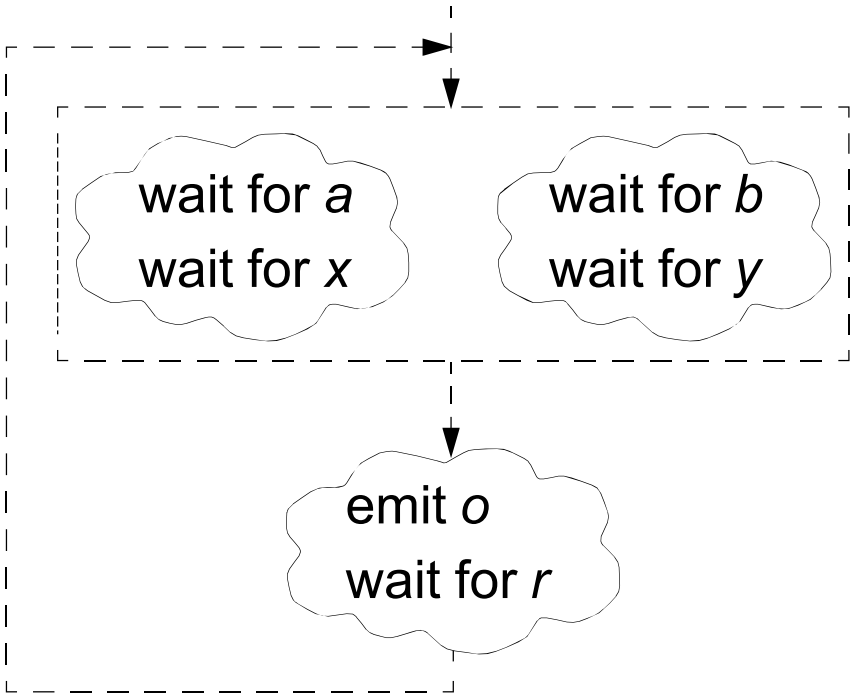
- Expressing such a system as a FSM (or extended FSM) is very difficult.

State Explosion

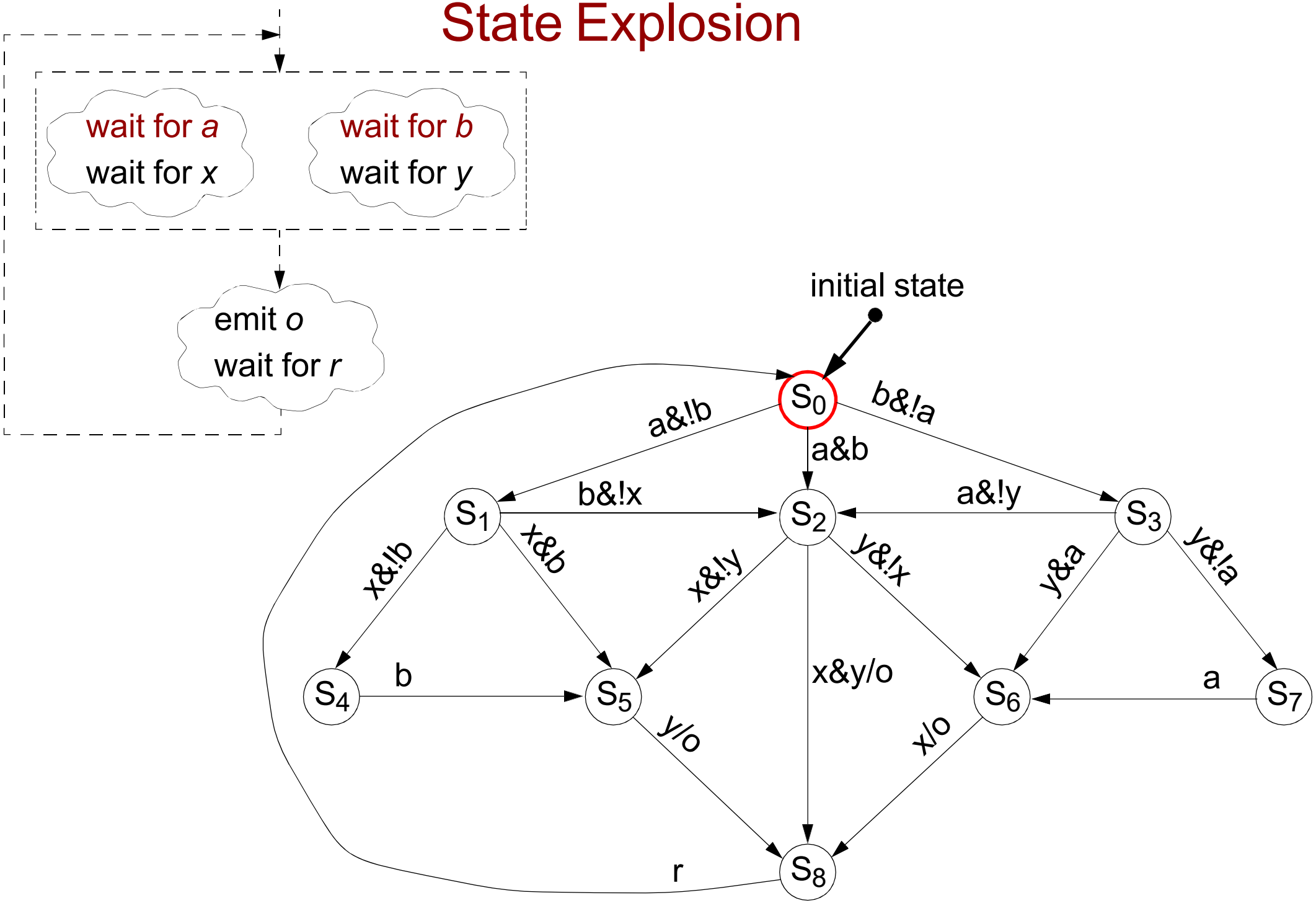
Example

After starting the system, it waits simultaneously for event *a* followed by *x*, and event *b* followed by *y*. Events can arrive in any order, except that *x* follows *a* and *y* follows *b*. Once the events are received, output *o* is emitted. Then the system waits for the reset signal *r* to return into the initial state.

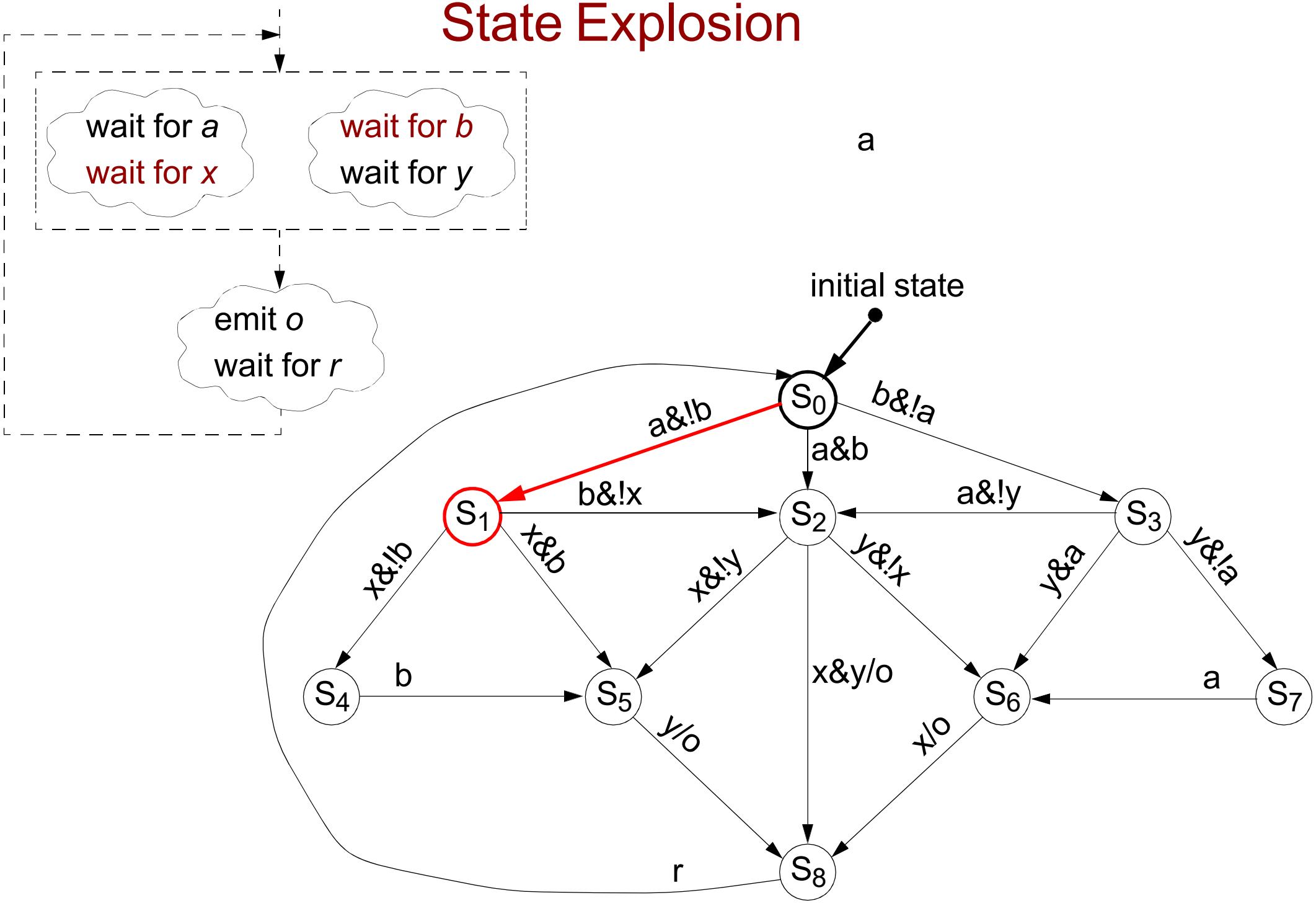
- Input events: {a, b, x, y, r}
- Output: {o}
- States: {S_0, S_1, ..., S_8}



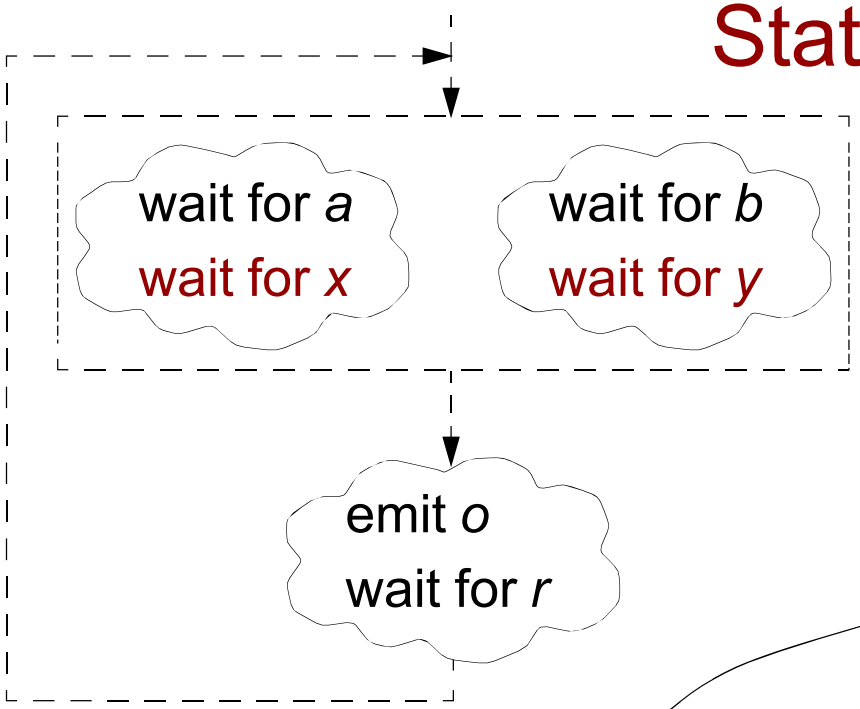
State Explosion



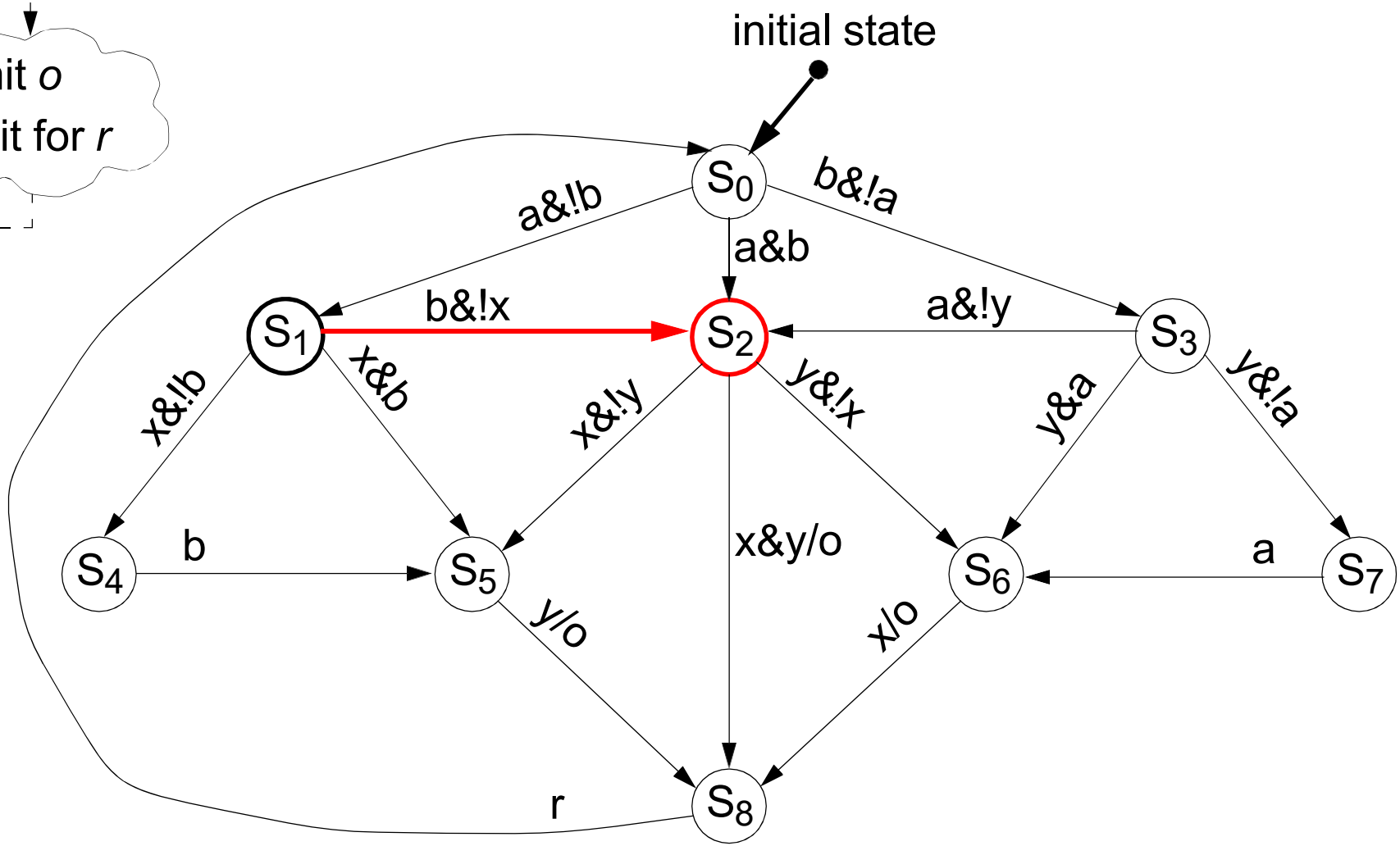
State Explosion



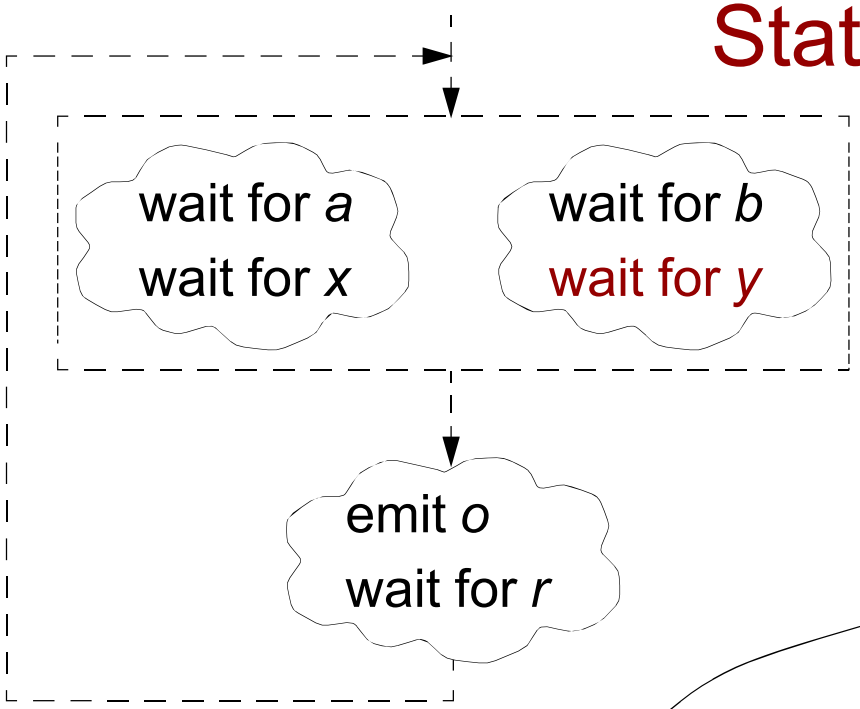
State Explosion



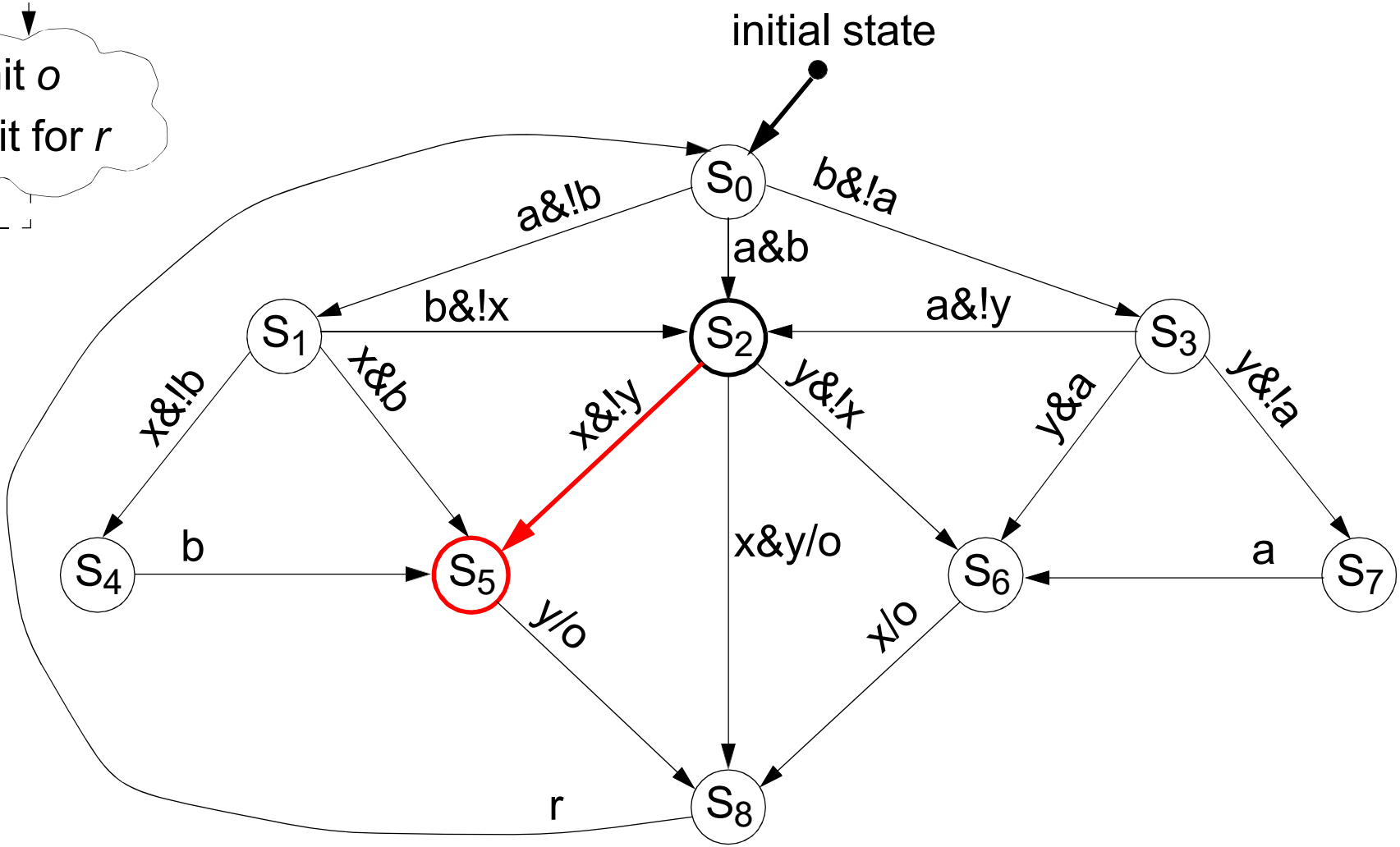
a b



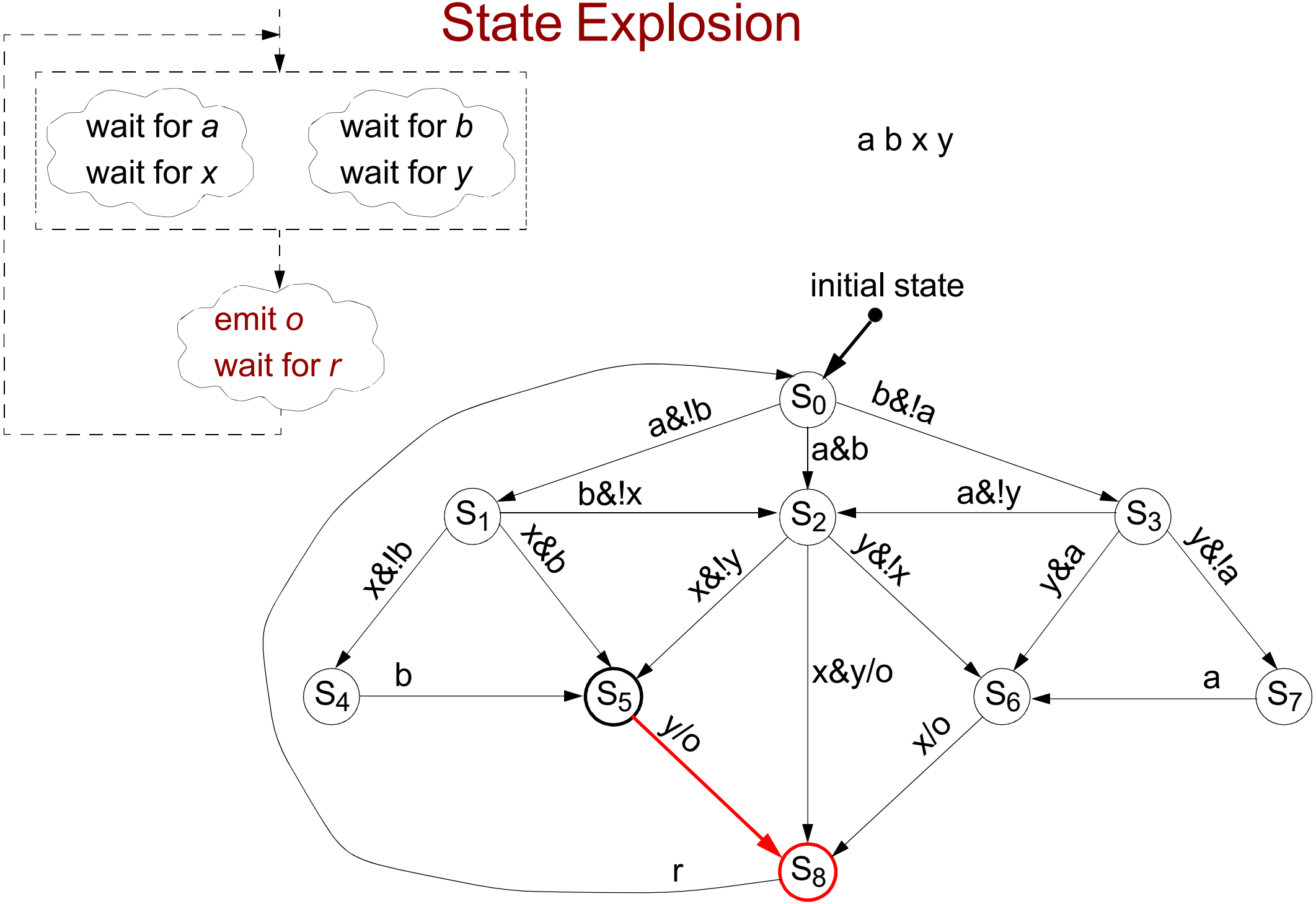
State Explosion



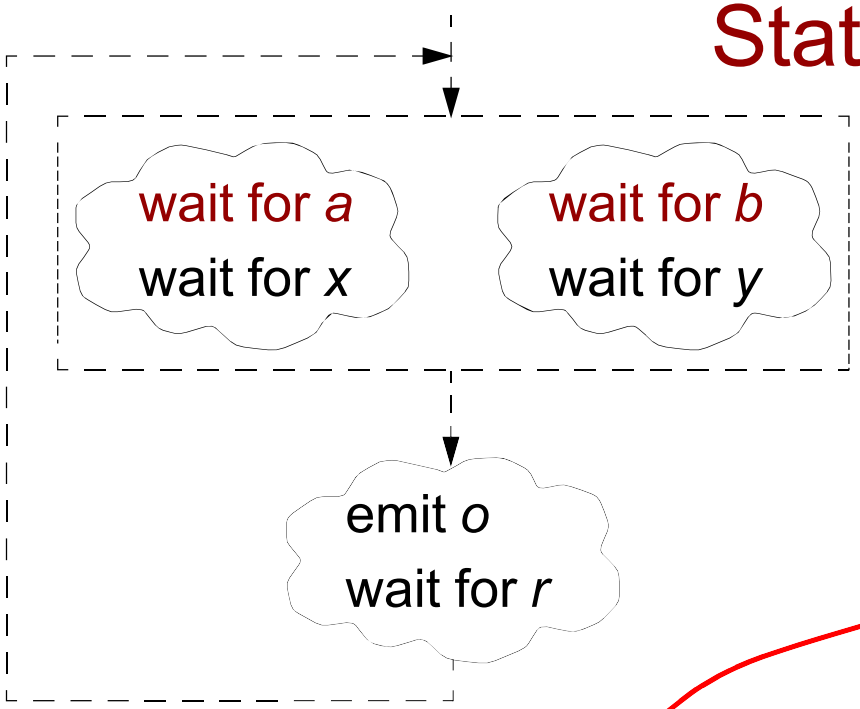
a b x



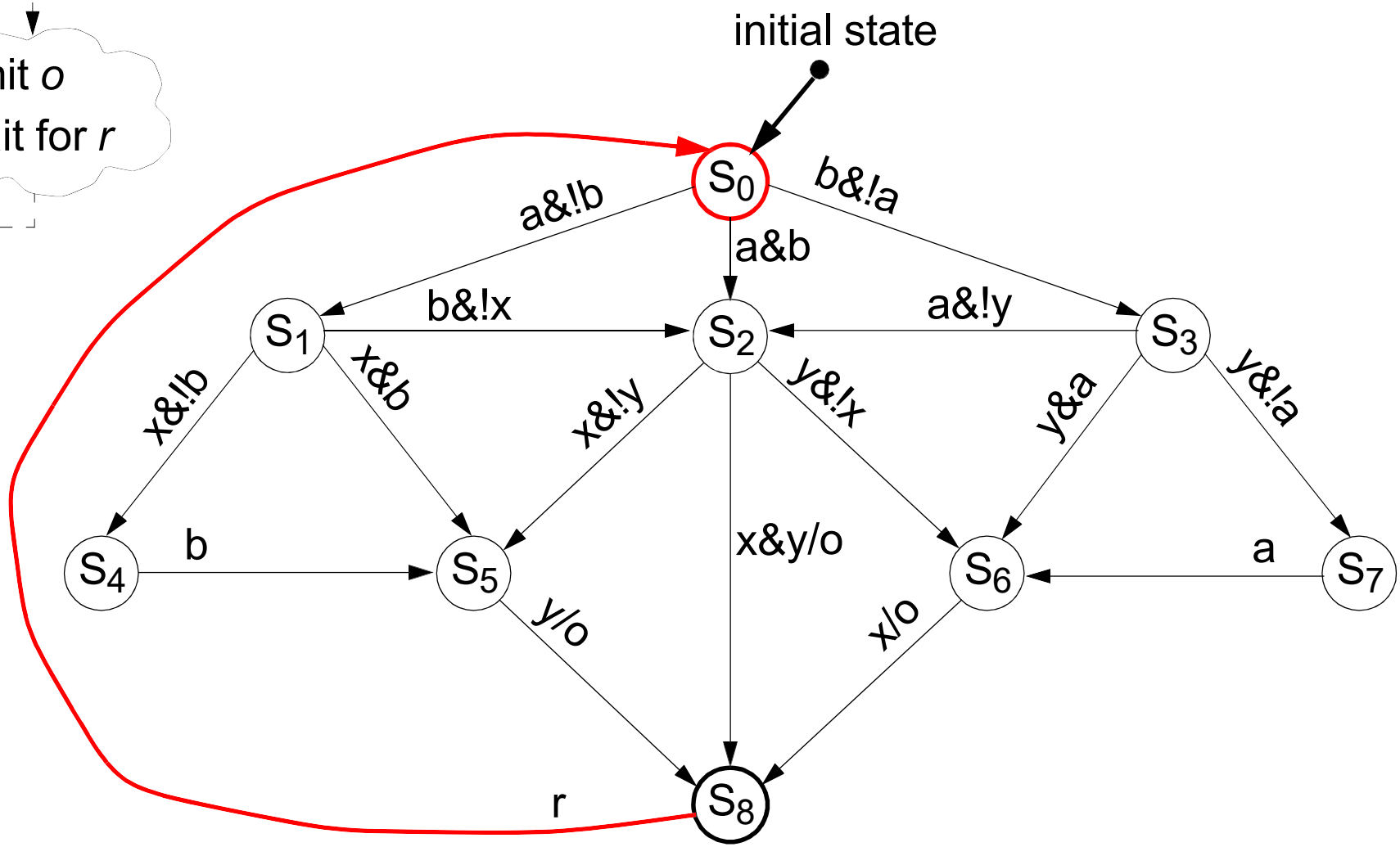
State Explosion



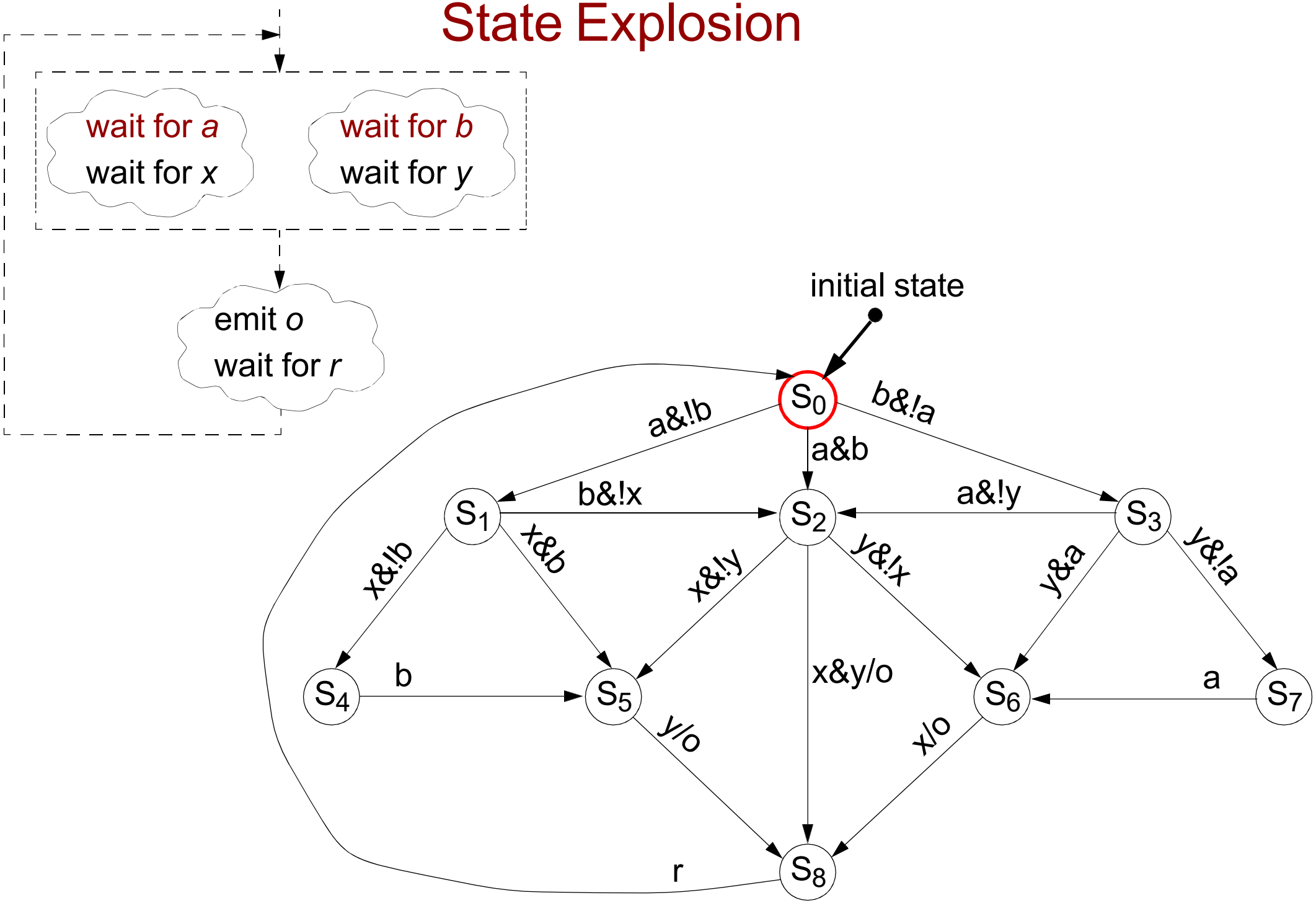
State Explosion



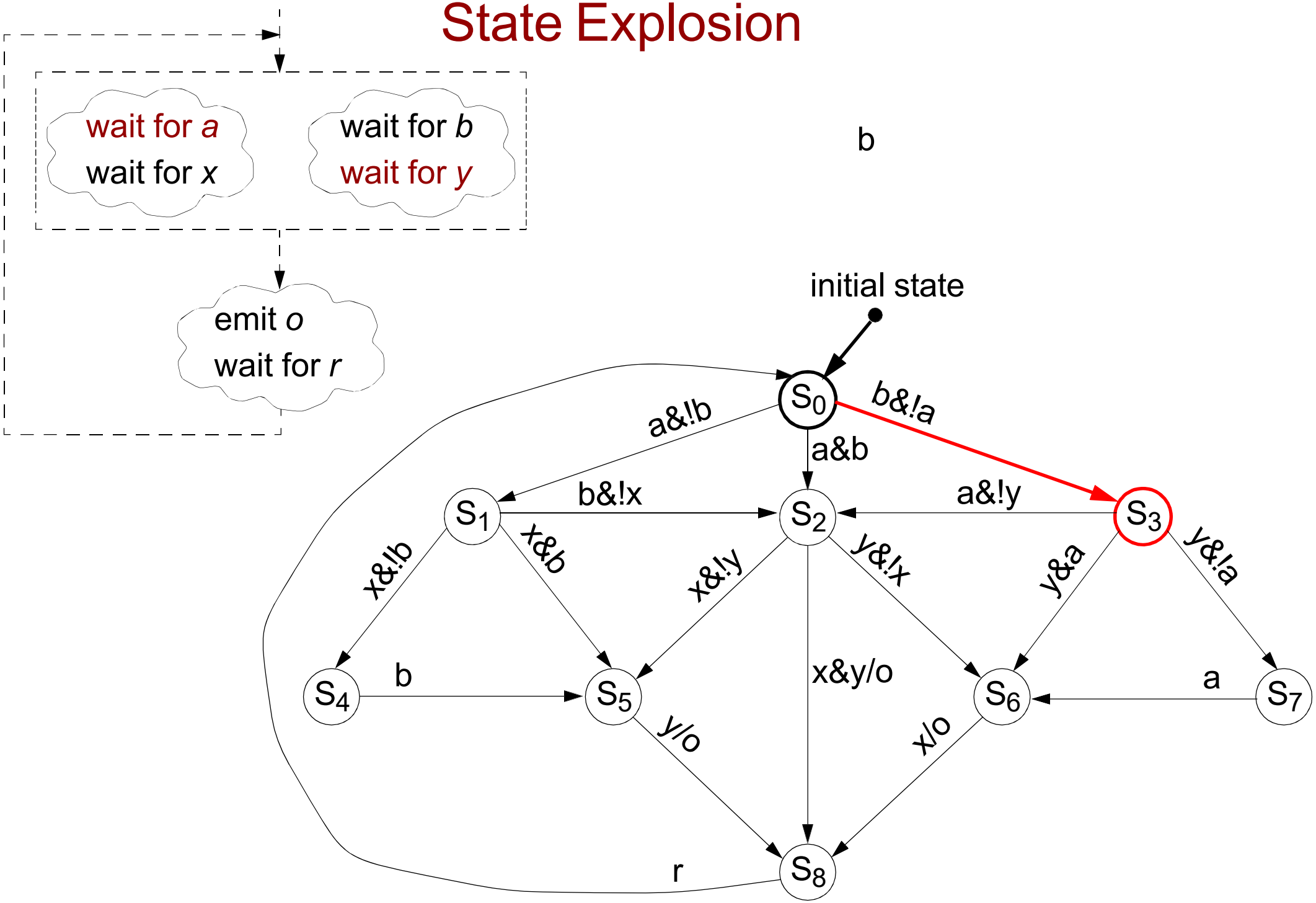
a b x y r



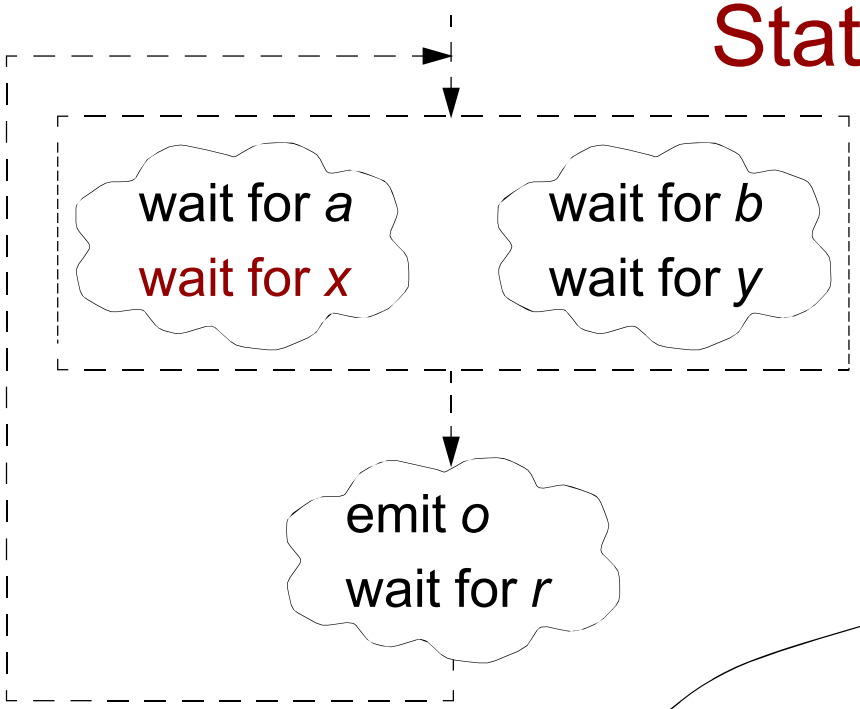
State Explosion



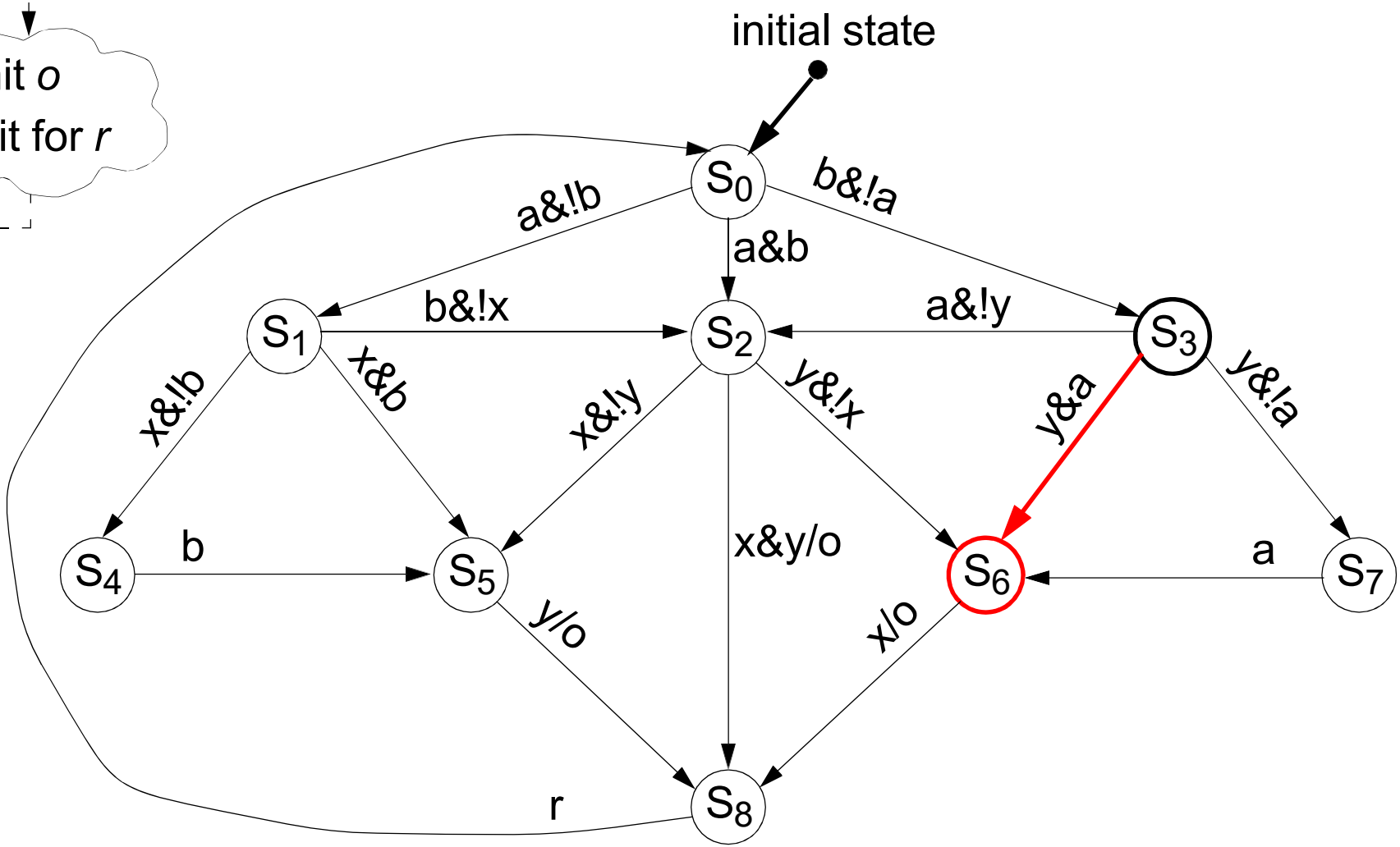
State Explosion



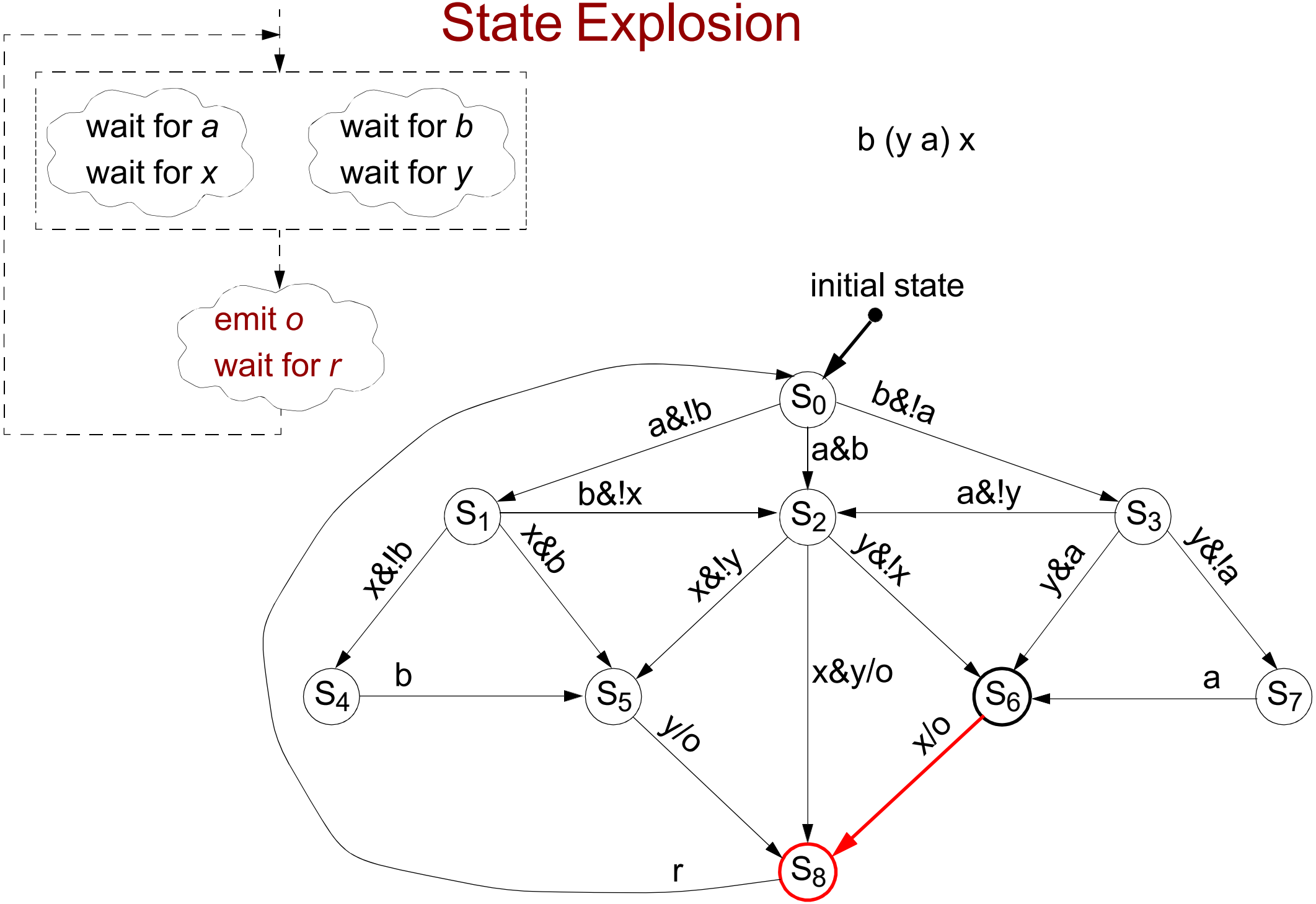
State Explosion



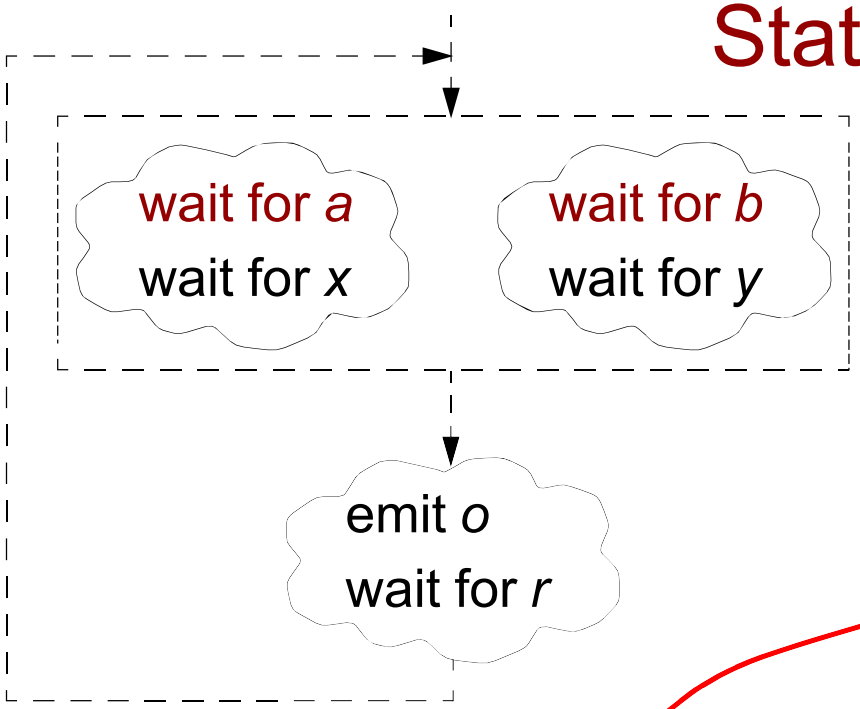
b (y a)



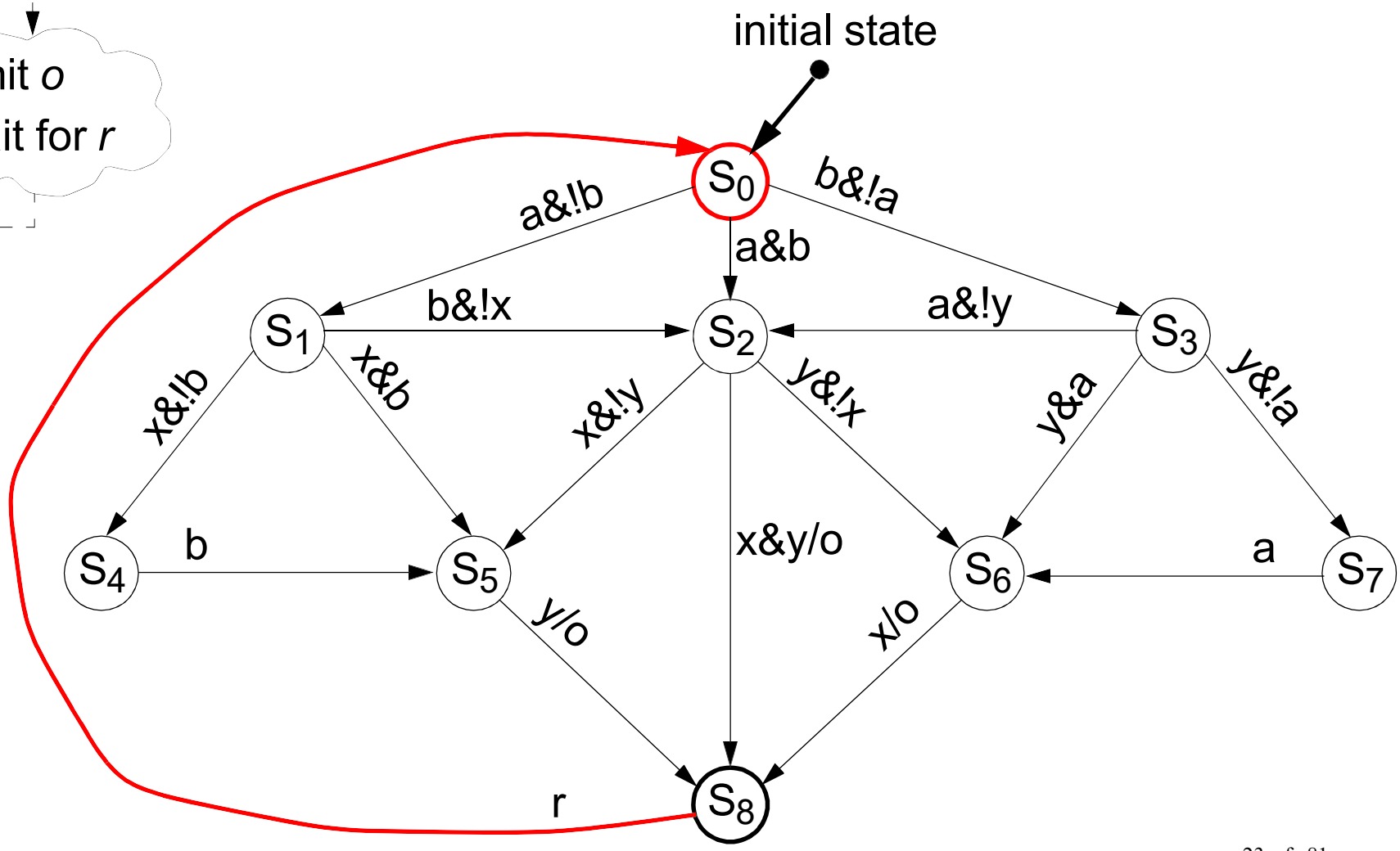
State Explosion



State Explosion



b (y a) x r



Hierarchical Concurrent Finite State Machines

- There are two important mechanisms that reduce the size of an FSM model:
  1. Hierarchy
  2. Concurrency

Important

- Using Hierarchy and concurrency we only reduce the size of the graphical or textual model; the intrinsic complexity - the number of states of the actual system - cannot be reduced.
- However, the difficulty of realising the model is drastically reduced.

Hierarchical Concurrent Finite State Machines

Hierarchy

- A single state S can represent an enclosed state machine F:

  Being in state S means that state machine F is active $\Rightarrow$ the system is in one of the states of the state machine F (*or states*).

Hierarchical Concurrent Finite State Machines

Hierarchy

- A single state S can represent an enclosed state machine F:

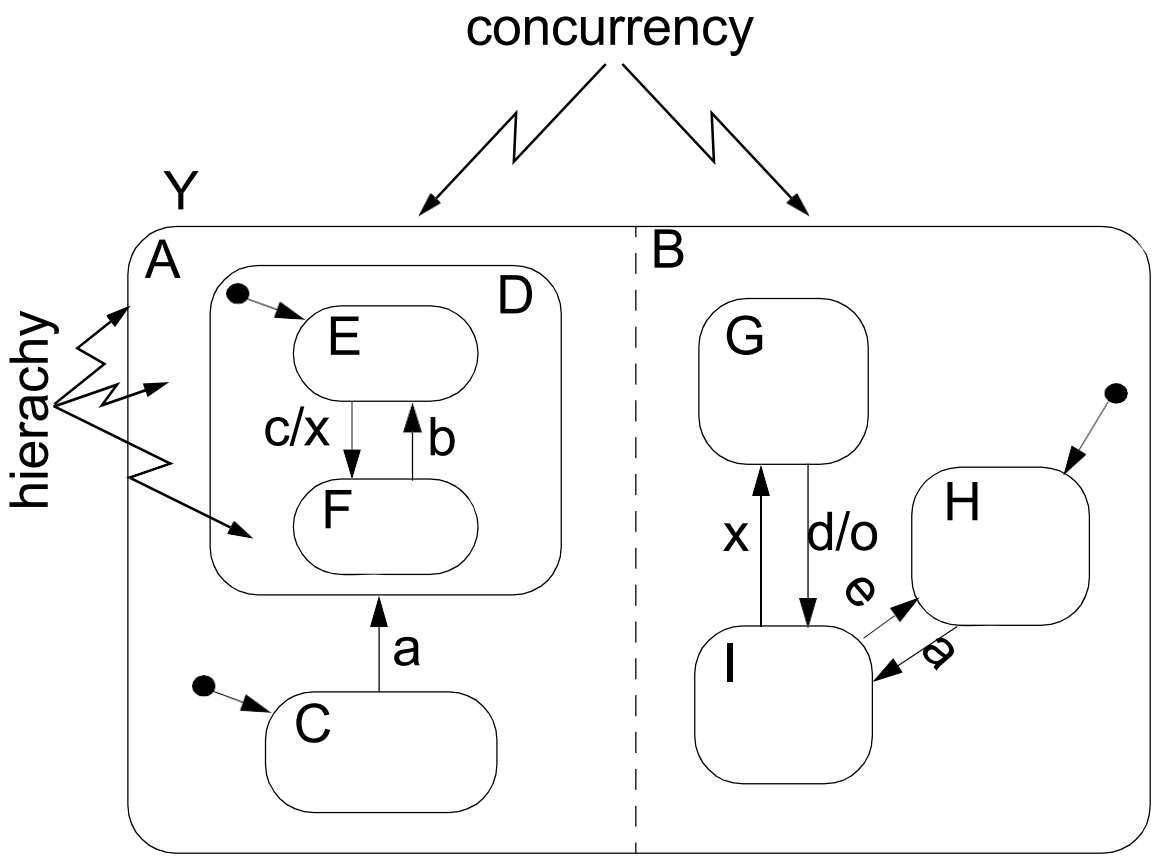
  Being in state S means that state machine F is active $\Rightarrow$ the system is in one of the states of the state machine F (*or states*).

Concurrency

- Two or more state machines are viewed as being simultaneously active $\Rightarrow$ the system is in one state of each parallel state machine simultaneously (*and* states).

Hierarchical Concurrent Finite State Machines

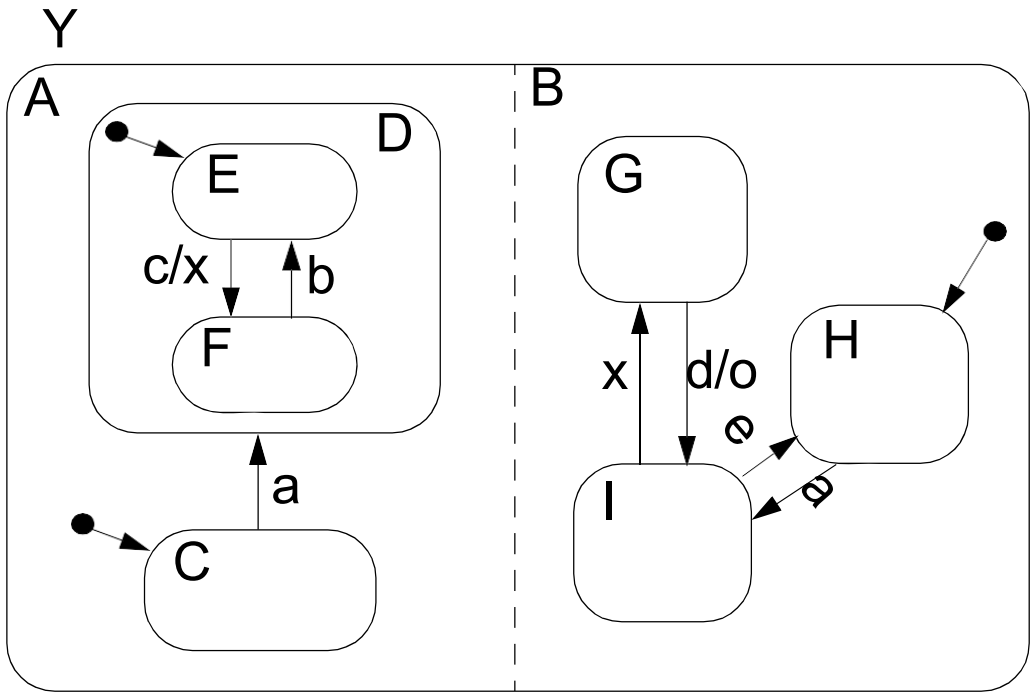
Statecharts is a graphical language for hierachical concurrent FSMs



Hierarchical Concurrent Finite State Machines

Statecharts is a graphical language for hierachical concurrent FSMs

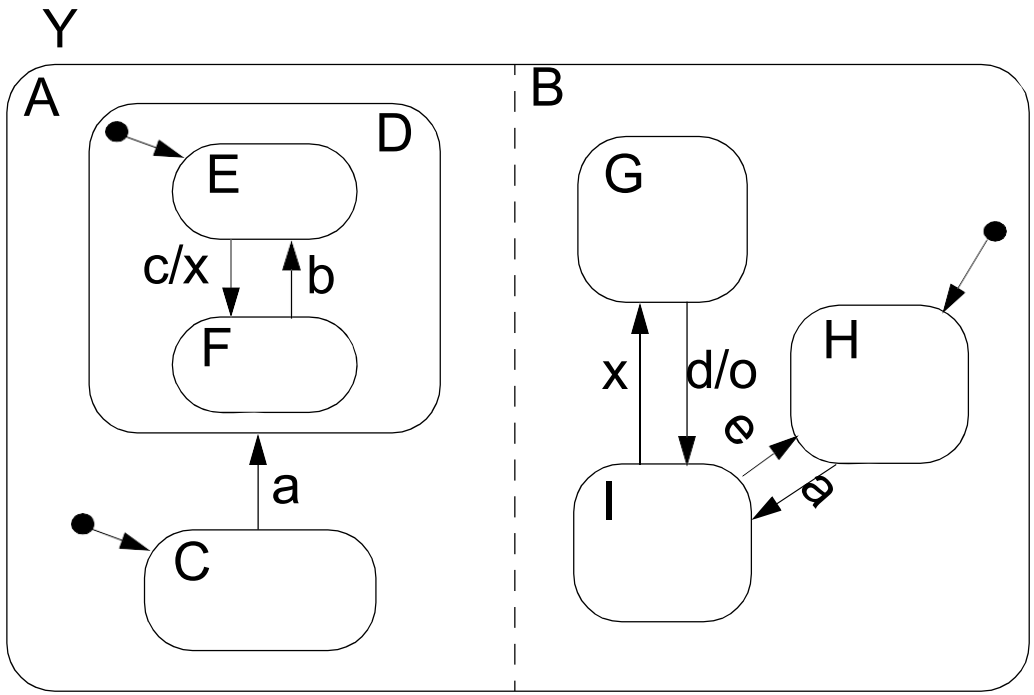
- System enters state Y $\Rightarrow$ it will be in both A and B.



Hierarchical Concurrent Finite State Machines

Statecharts is a graphical language for hierachical concurrent FSMs

- System enters state Y $\Rightarrow$ it will be in both A and B.
- A consists of D and C; C is initial state for A. D consists of E and F; E is initial state for D.
- B consists of G, I, and H; H is initial state for B.

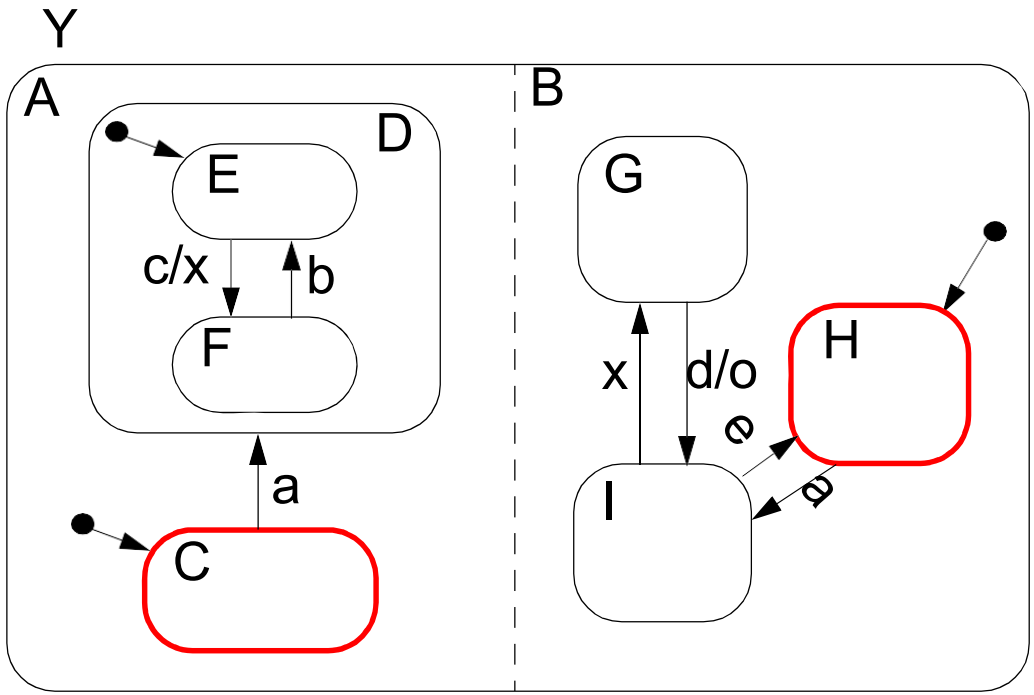


Hierarchical Concurrent Finite State Machines

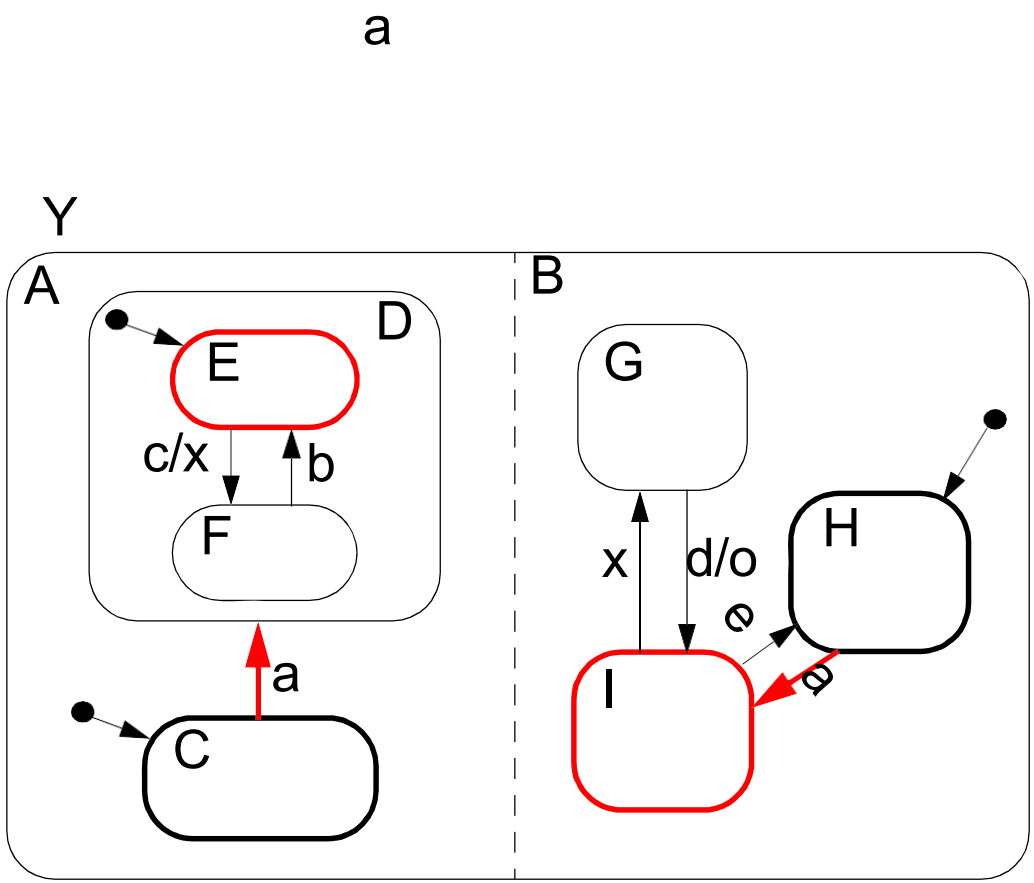
Statecharts is a graphical language for hierachical concurrent FSMs

- System enters state Y $\Rightarrow$ it will be in both A and B.
- A consists of D and C; C is initial state for A. D consists of E and F; E is initial state for D.
- B consists of G, I, and H; H is initial state for B.

Entering Y, the system will be simultaneously in C and H;

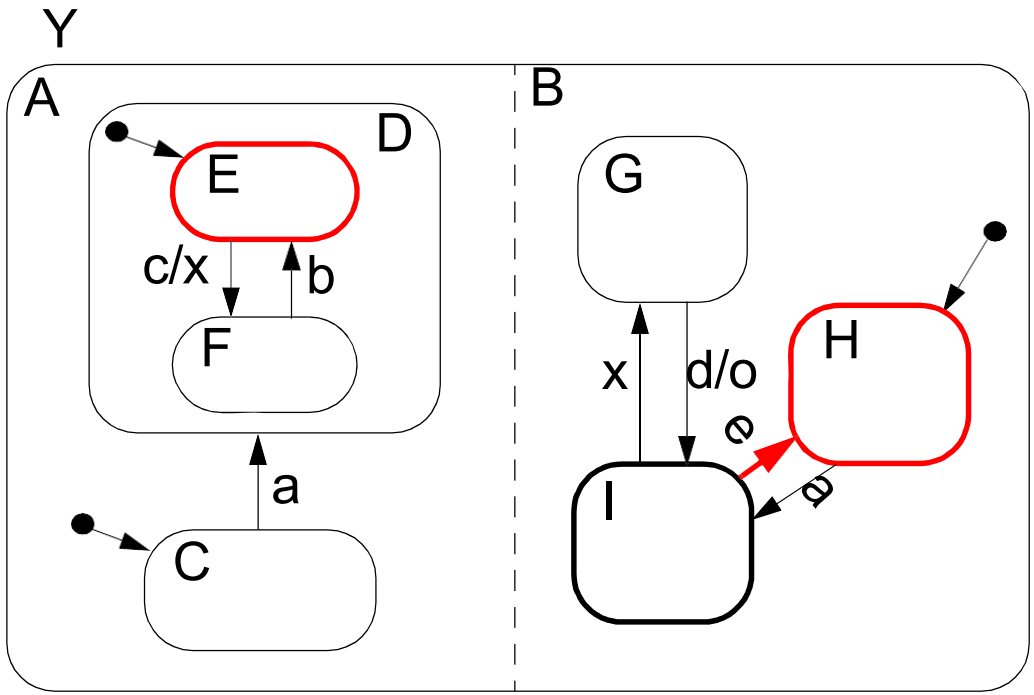


Hierarchical Concurrent Finite State Machines



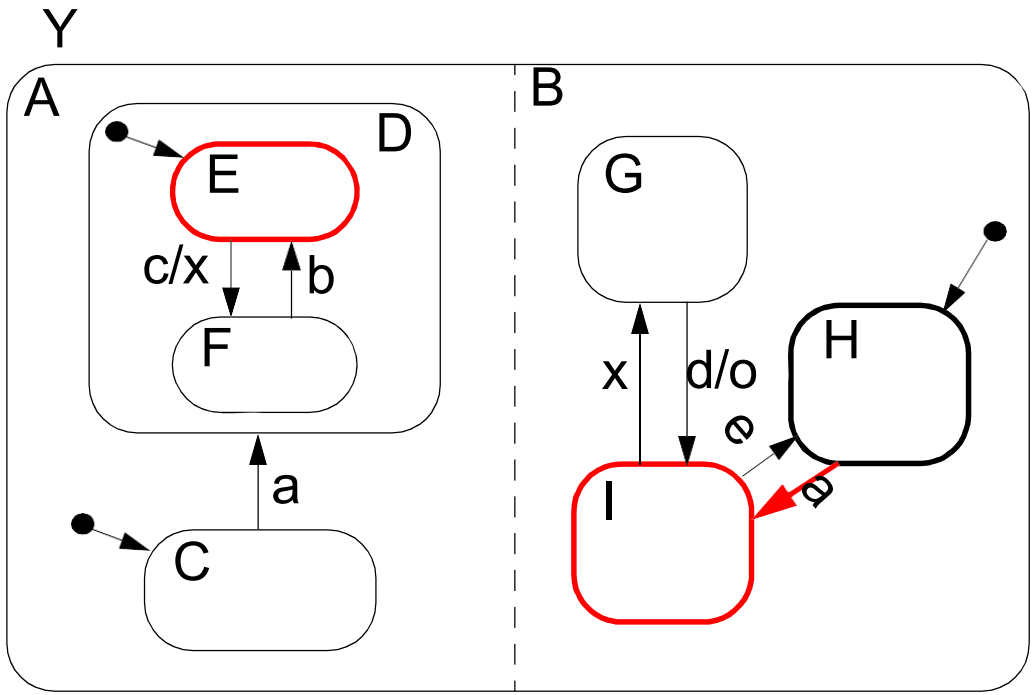
Hierarchical Concurrent Finite State Machines

a e



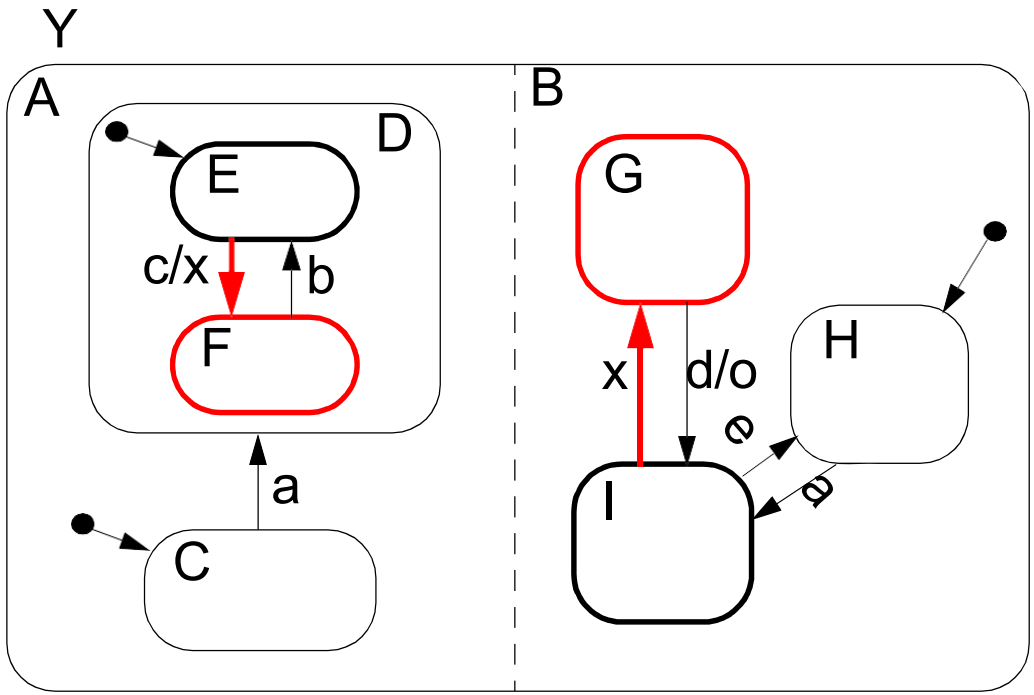
Hierarchical Concurrent Finite State Machines

a e a



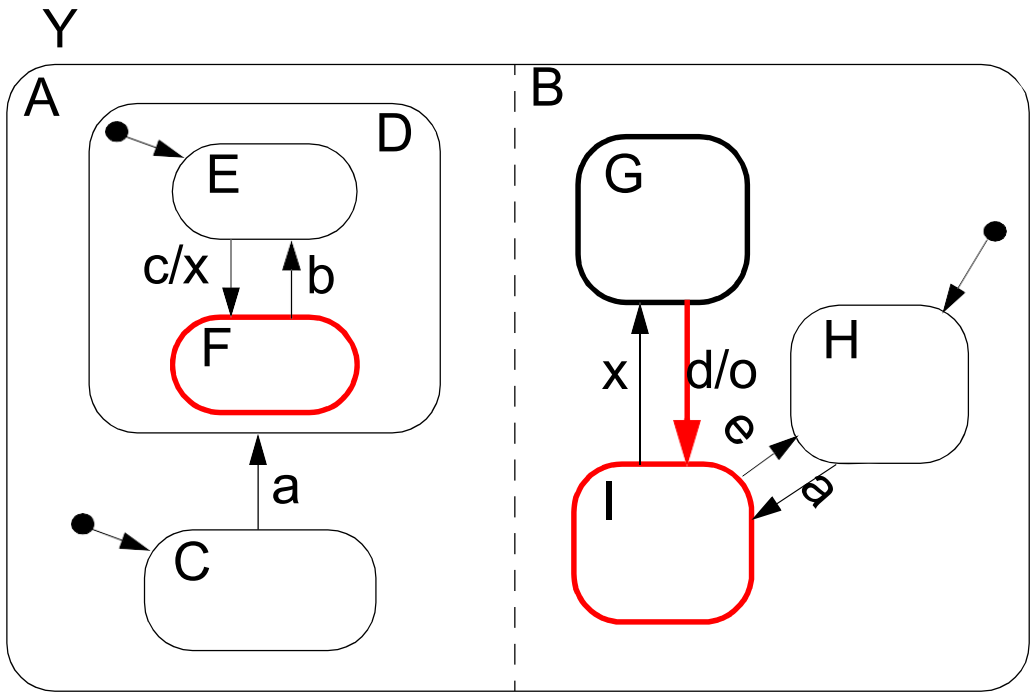
Hierarchical Concurrent Finite State Machines

a e a c



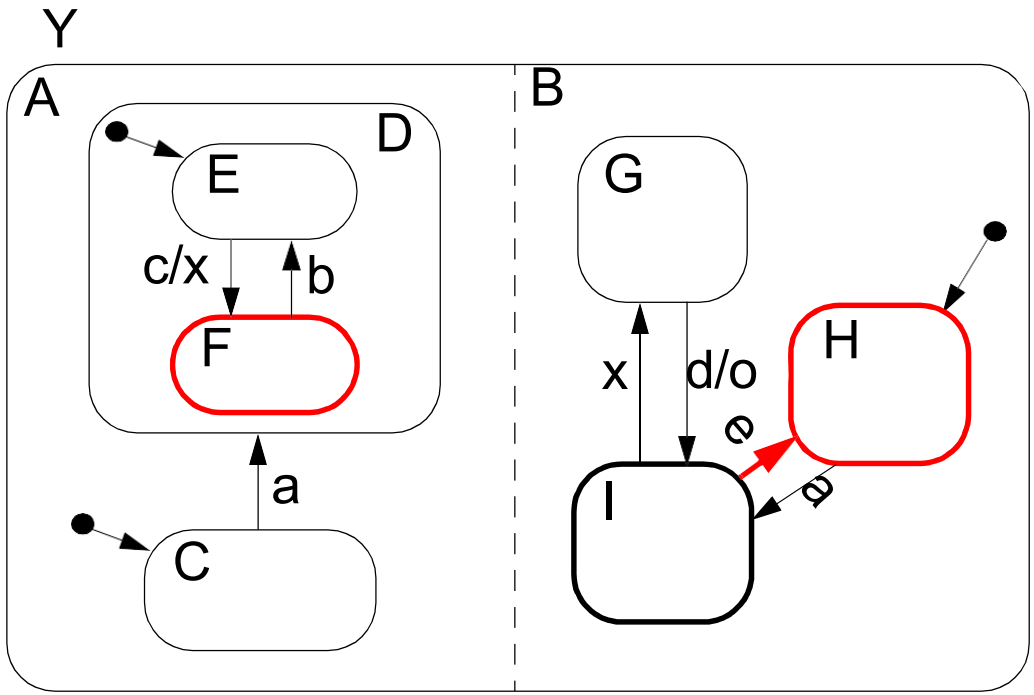
Hierarchical Concurrent Finite State Machines

a e a c d



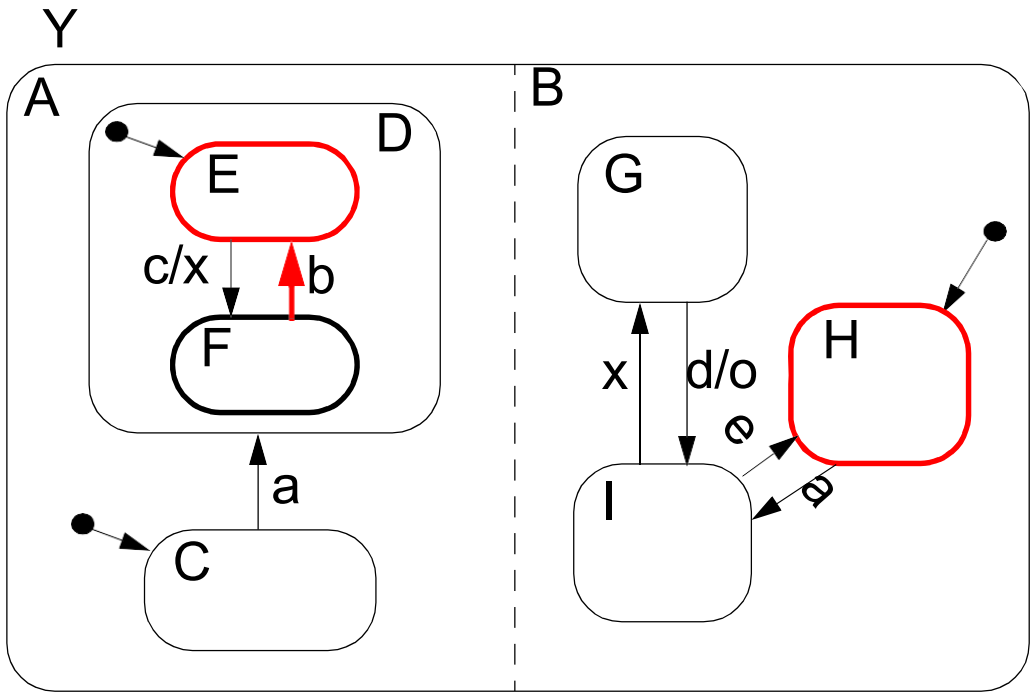
Hierarchical Concurrent Finite State Machines

a e a c d e



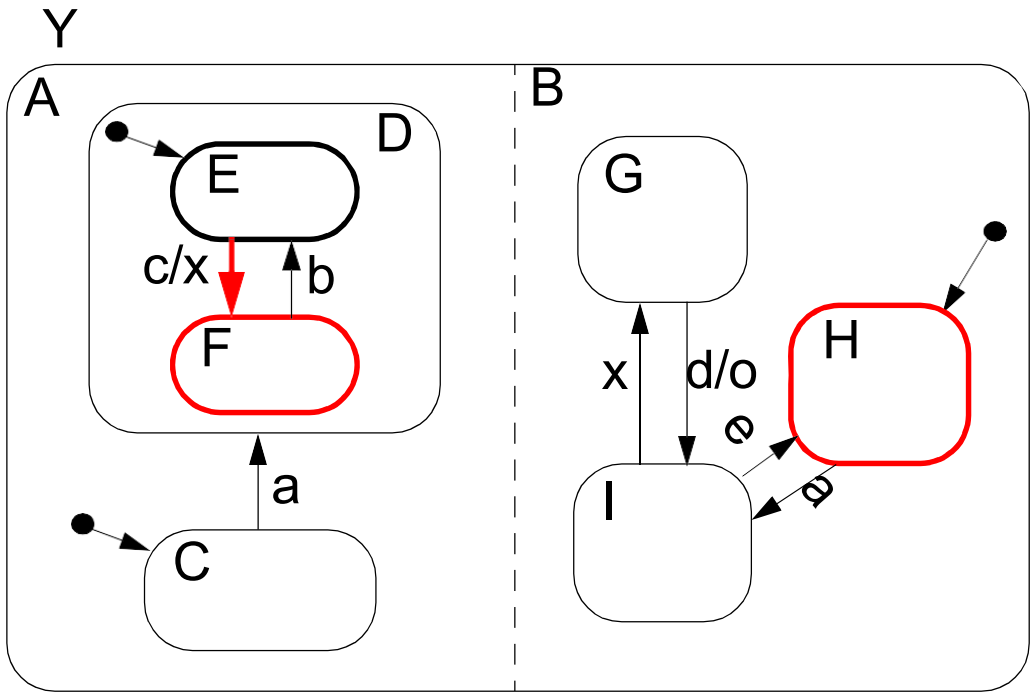
Hierarchical Concurrent Finite State Machines

a e a c d e b



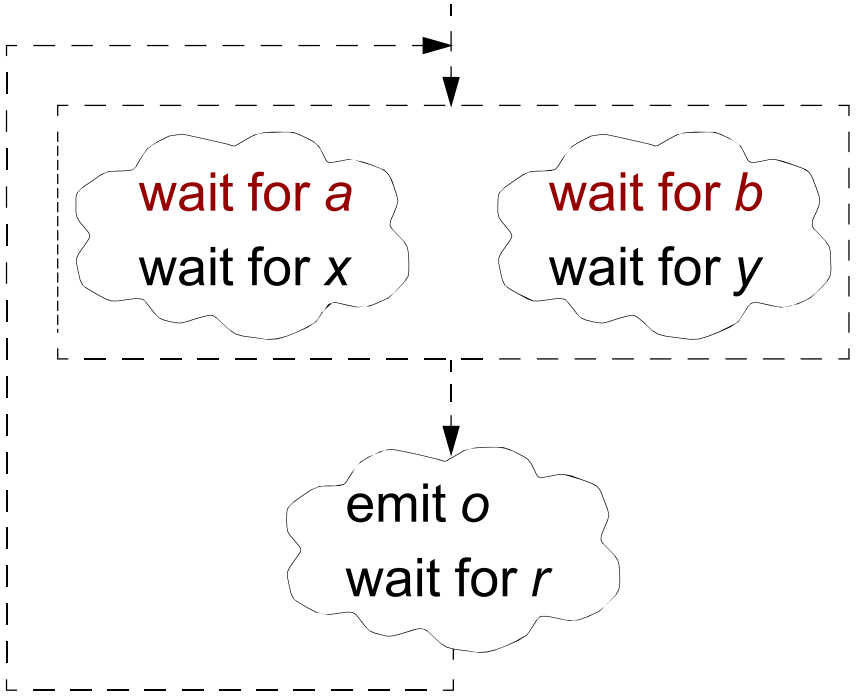
Hierarchical Concurrent Finite State Machines

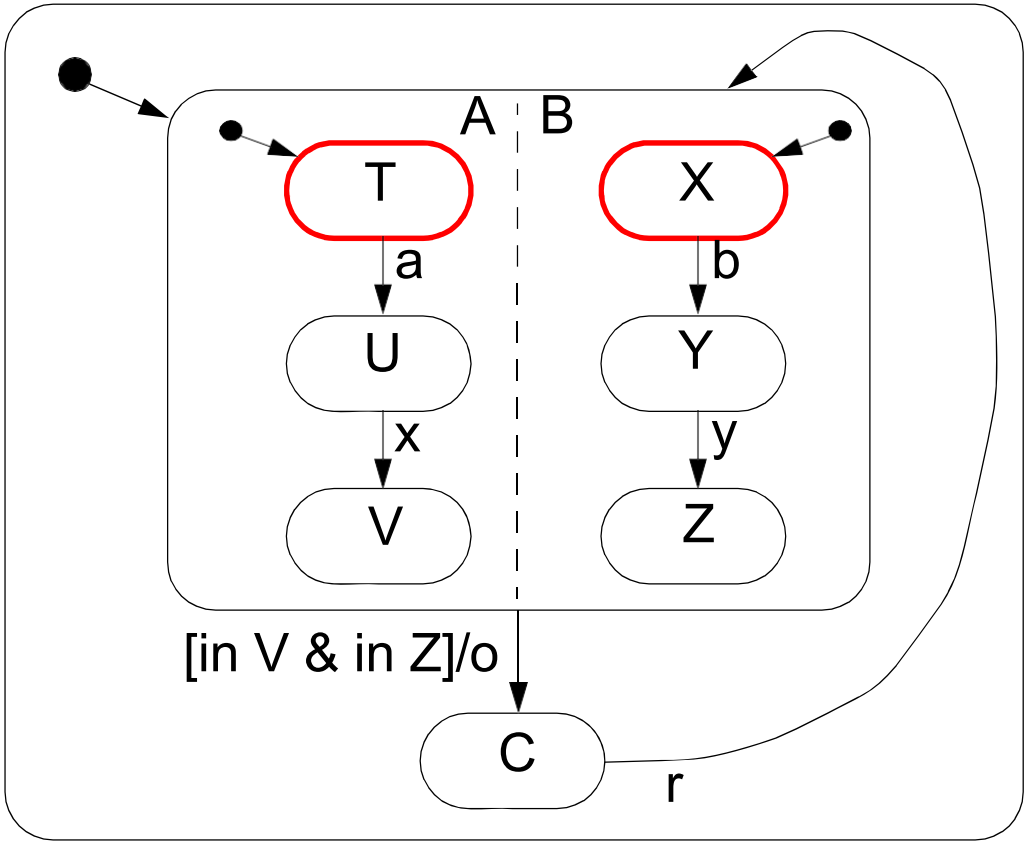
a e a c d e b c



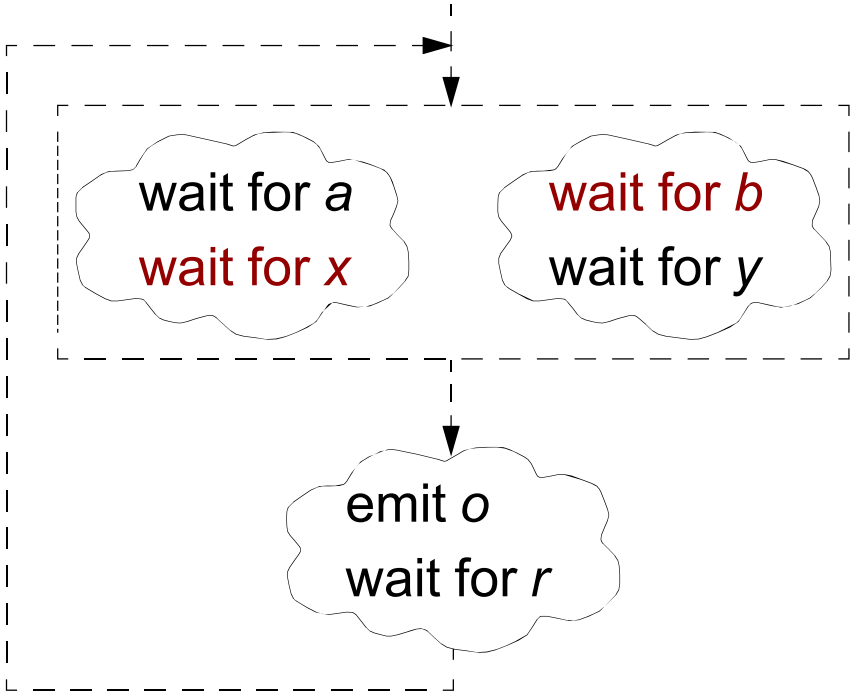
Hierarchical Concurrent Finite State Machines

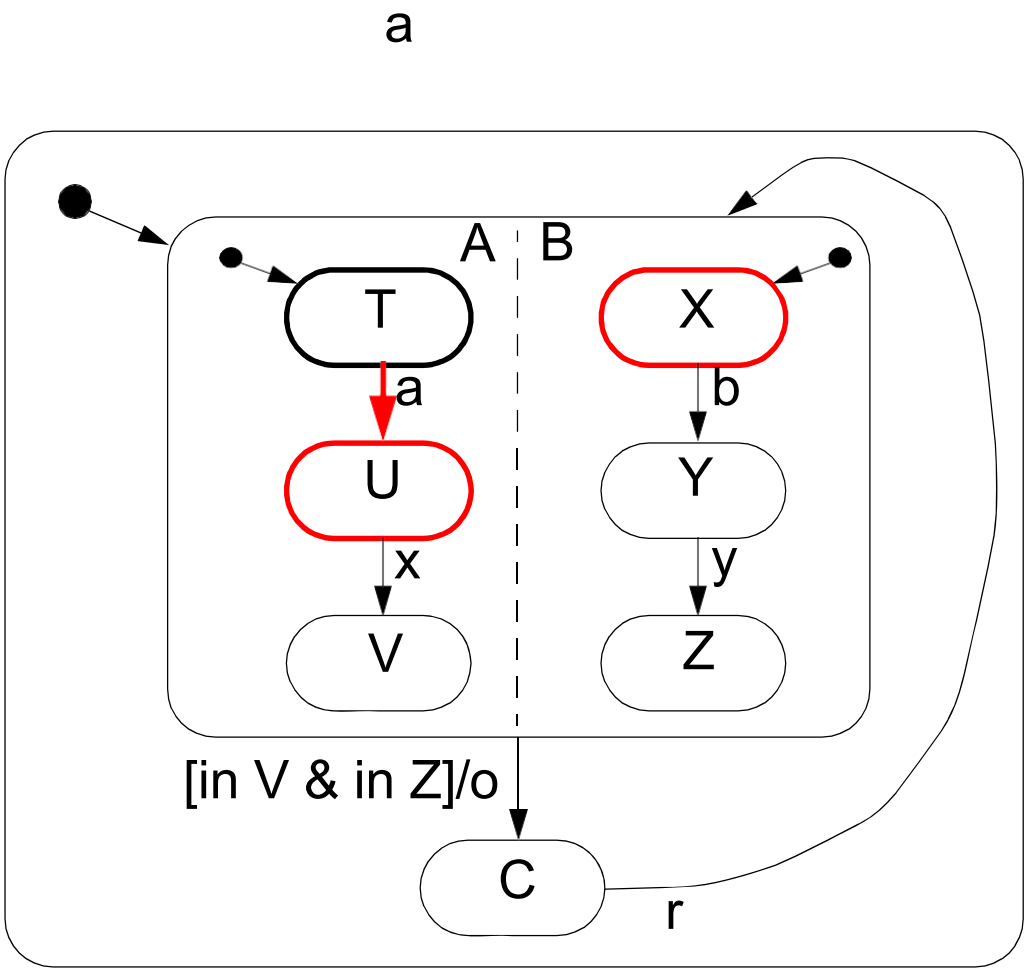
Our earlier example, now using Statecharts:



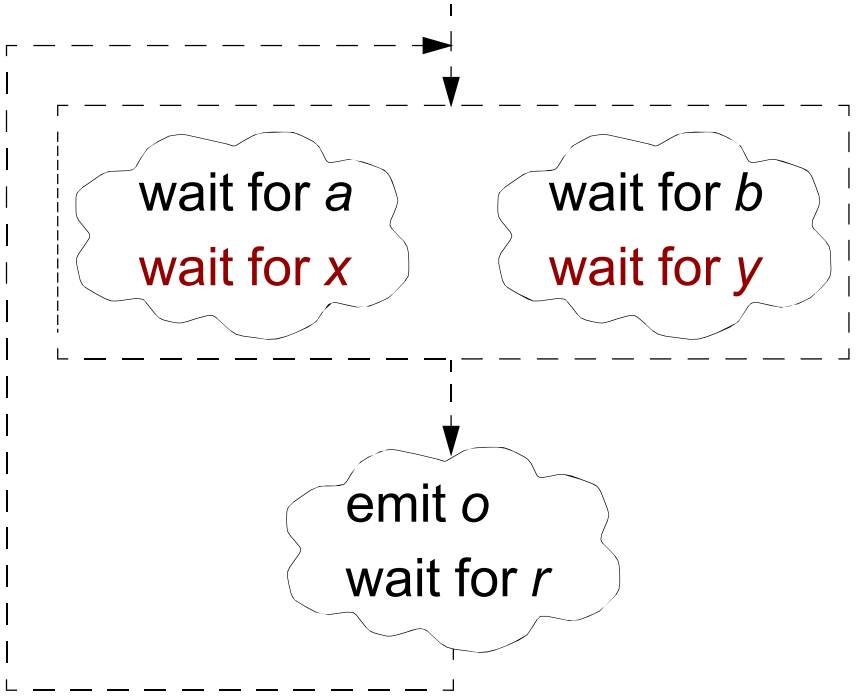


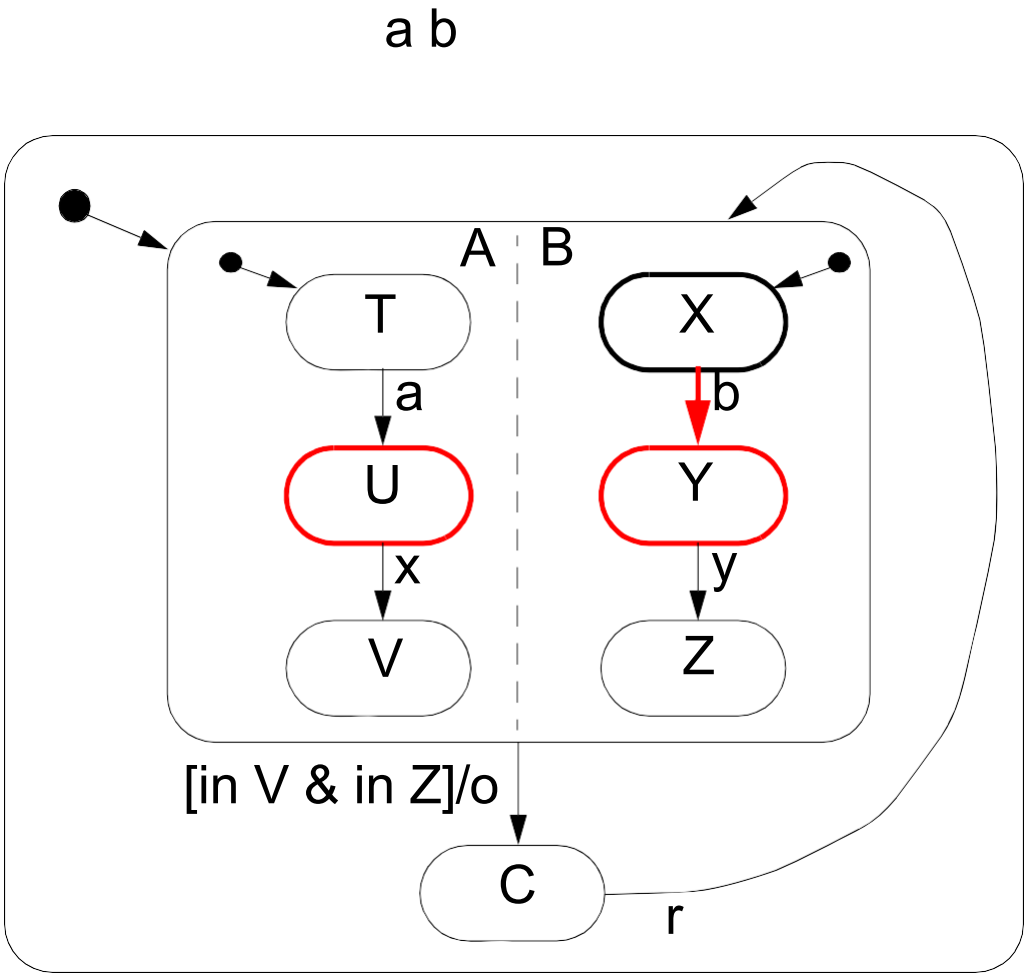
Hierarchical Concurrent Finite State Machines



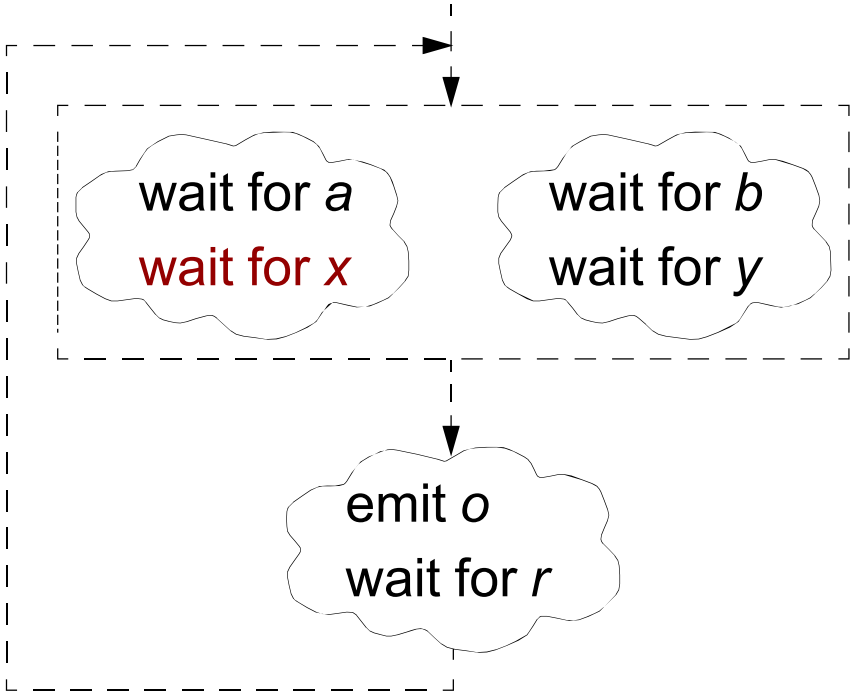


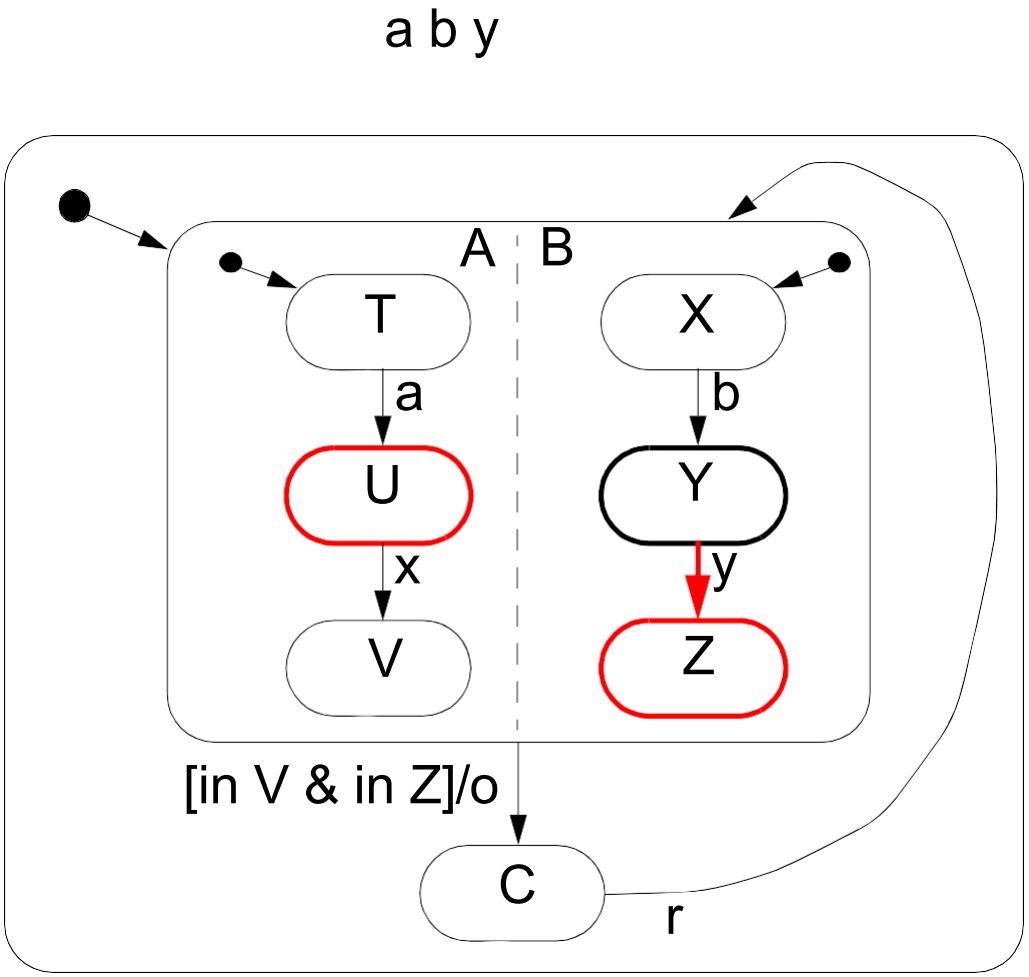
Hierarchical Concurrent Finite State Machines



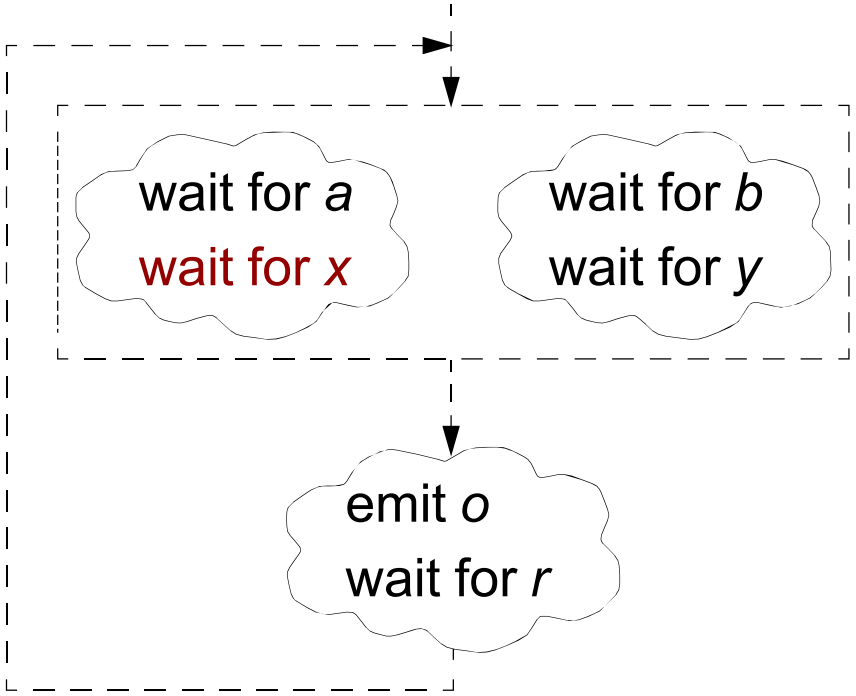


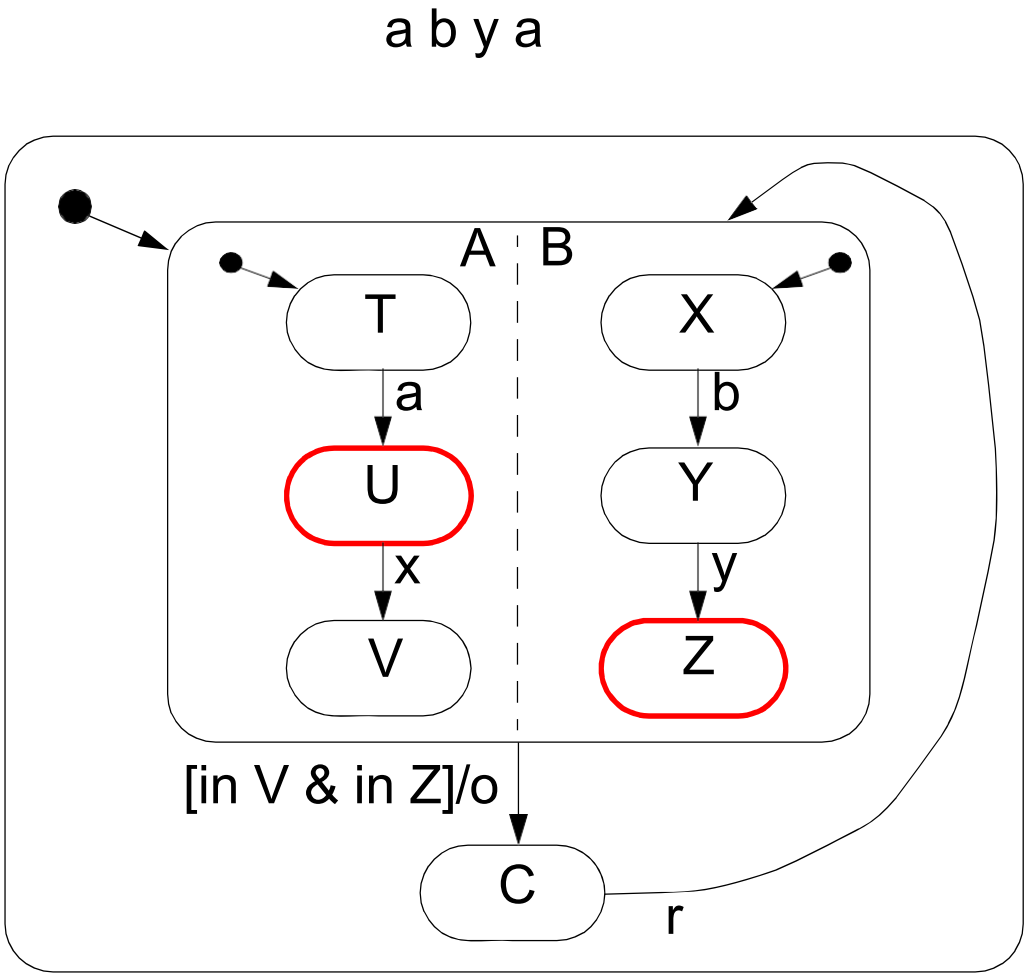
Hierarchical Concurrent Finite State Machines



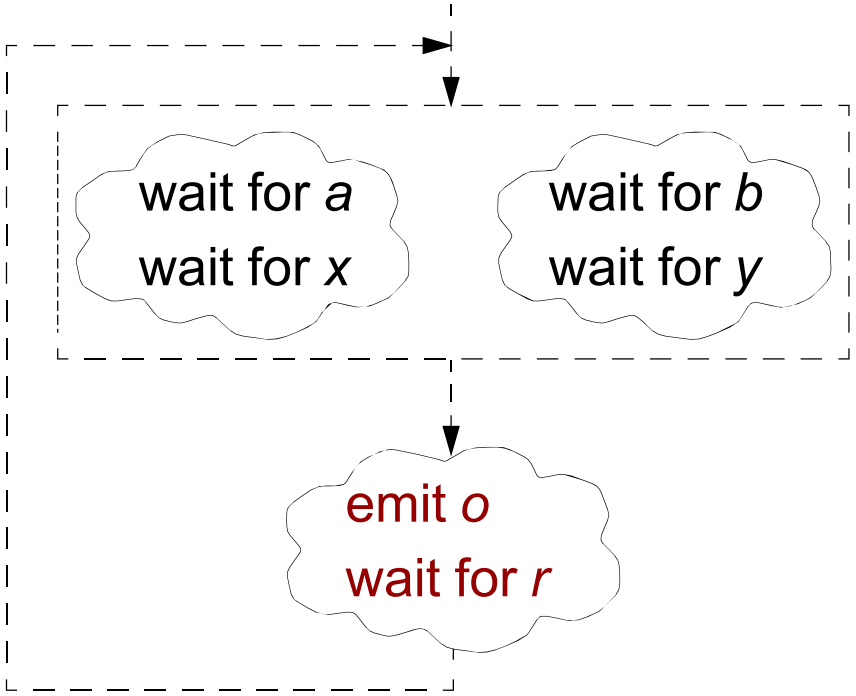


Hierarchical Concurrent Finite State Machines

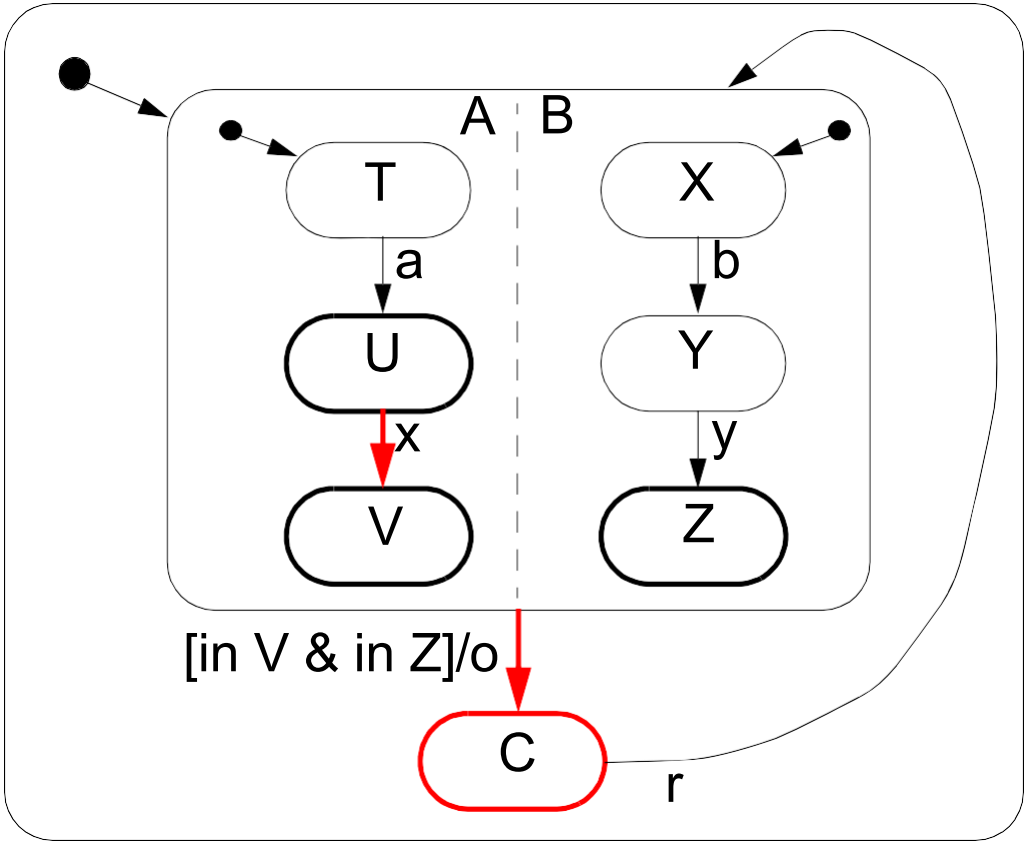




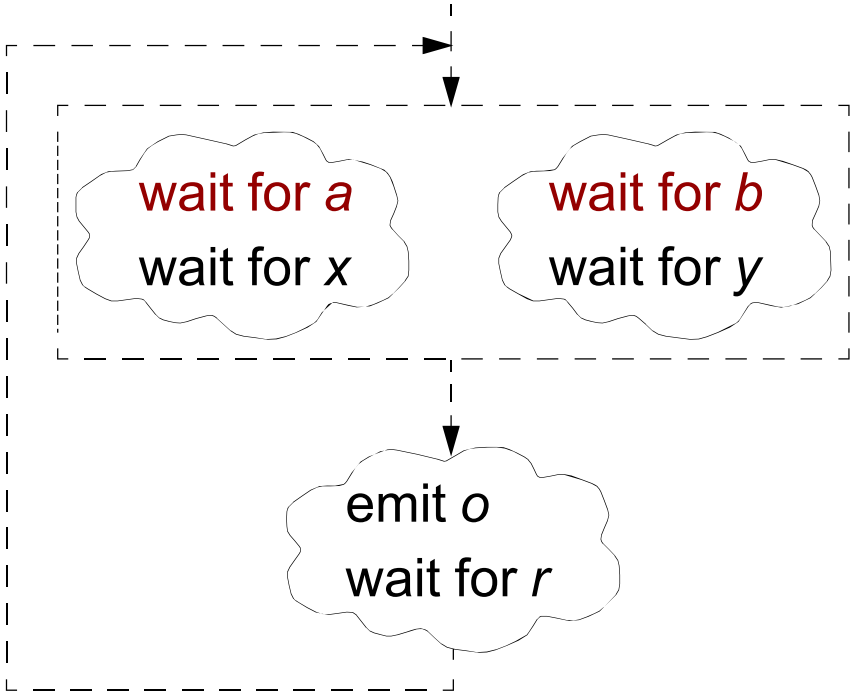
Hierarchical Concurrent Finite State Machines



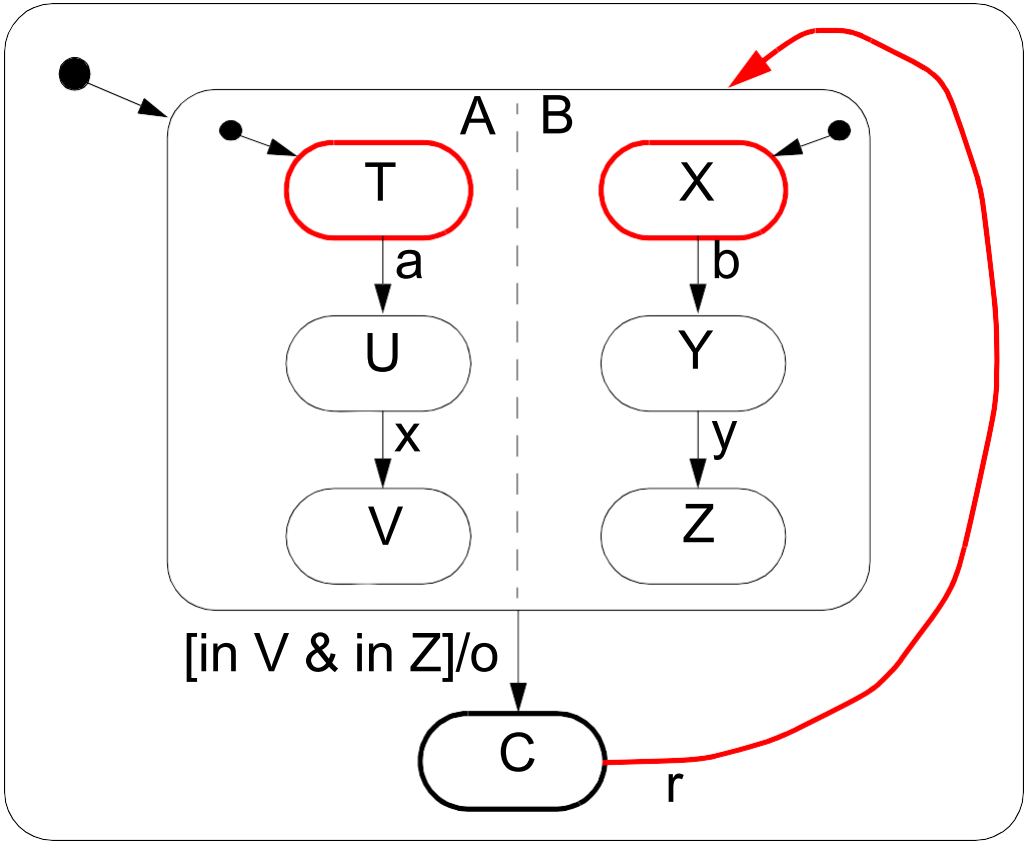
a b y a x



Hierarchical Concurrent Finite State Machines



a b y a x r



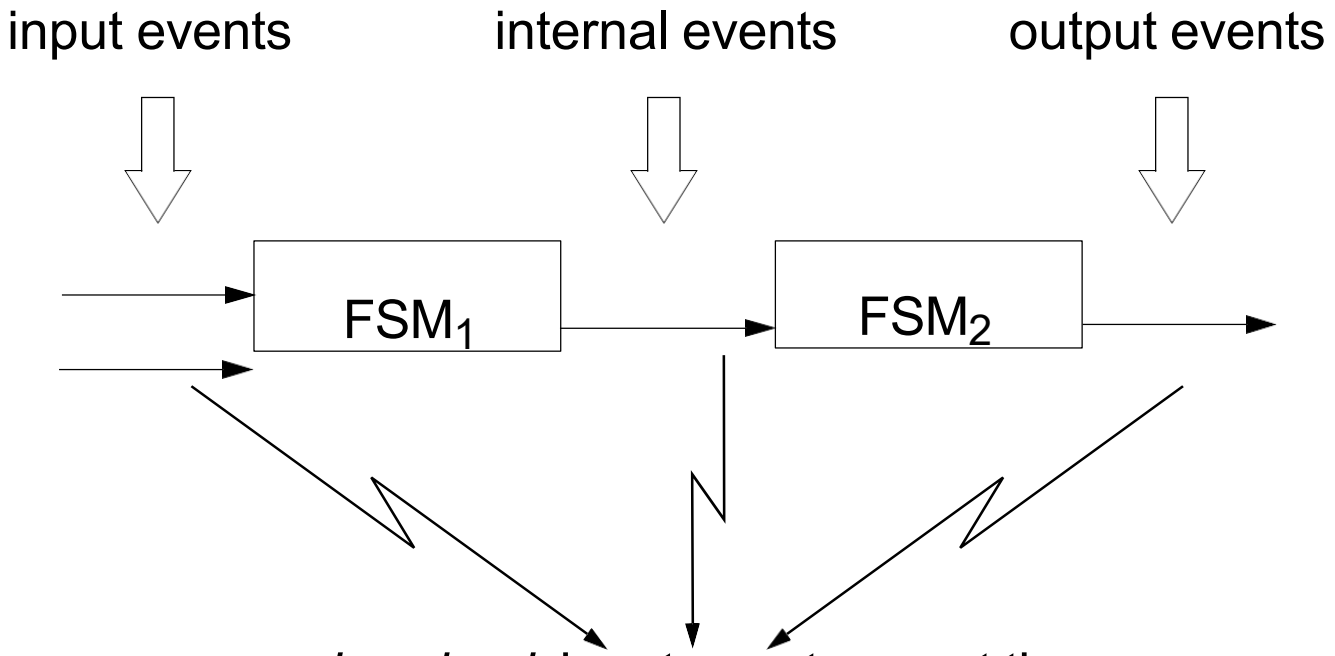
FSMs: Time and Synchrony

- (hierarchical concurrent) FSMs are *synchronous models*.
  - The synchrony hypothesis:

    The time is a sequence of instants (clock ticks) between which nothing interesting occurs. In each instant, some events (inputs) occur in the environment, and a reaction (output) is computed *instantly* by the modelled design.

    - Computation and internal communication (between the FSMs composing the system) take no time (compare to Discrete Event, where components can have arbitrary delays!).
    - Events are either simultaneous (occur at the same clock tick) or one strictly precedes the other (as opposed to dataflow and Petri Nets where we only have a partial order of events).

FSMs: Time and Synchrony



synchronized: input events are at the same time with the internal and output events generated as response.

FSMs: Time and Synchrony

Question

Is the synchronous model sufficiently realistic to be used in practice?

FSMs: Time and Synchrony

Question

Is the synchronous model sufficiently realistic to be used in practice?

Answer

For some applications yes!

It is the case when the following assumption is true:

The reaction time of the system (including internal communication) is neglectable compared to the rate of external events.

Why Do We Like Synchronous Models?

- A set of communicating, concurrent FSMs behaves exactly like one equivalent FSM.

Models are deterministic.
It is possible to formally reason about models and to formally check properties of the model. This is important for safety critical applications.

- It is possible to efficiently synthesise (compile) synchronous models to hardware or software.

Why Do We Not Like Synchronous Models?

- Reasoning, verification and synthesis based on synchronous models are meaningful and correct only as long as:
  1. A completely synchronous implementation of the whole system is possible.
  2. We are sure that for the *implemented system* the assumption is true: *The reaction time of the system (including internal communication) is neglectable compared to the rate of external events*.

- Implementing *large* models as synchronous systems is expensive and often technically impossible.

Synchronous/Reactive Languages

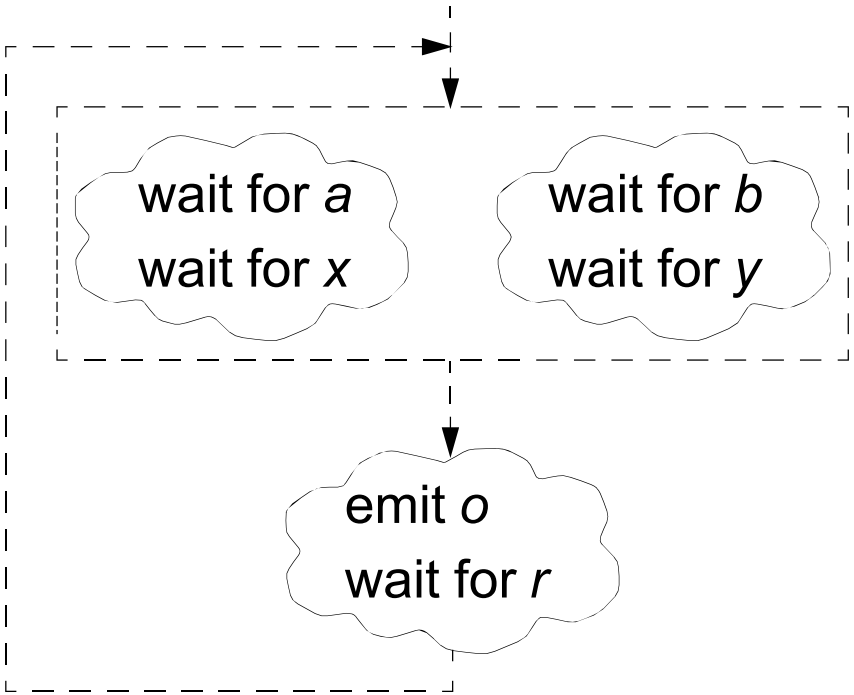
- Synchronous/reactive languages describe systems as a set of concurrently executing synchronized activities.
  - Communication is through signals.
  - Signals are either present or absent at a certain *tick*.
  - The presence of a signal is called an event.

- *These language are semantically equivalent to the (extended hierarchical concurrent) FSM model !!!*

- *Esterel* is a well known synchronous/reactive language. Every Esterel model can be compiled to an extended FSM.

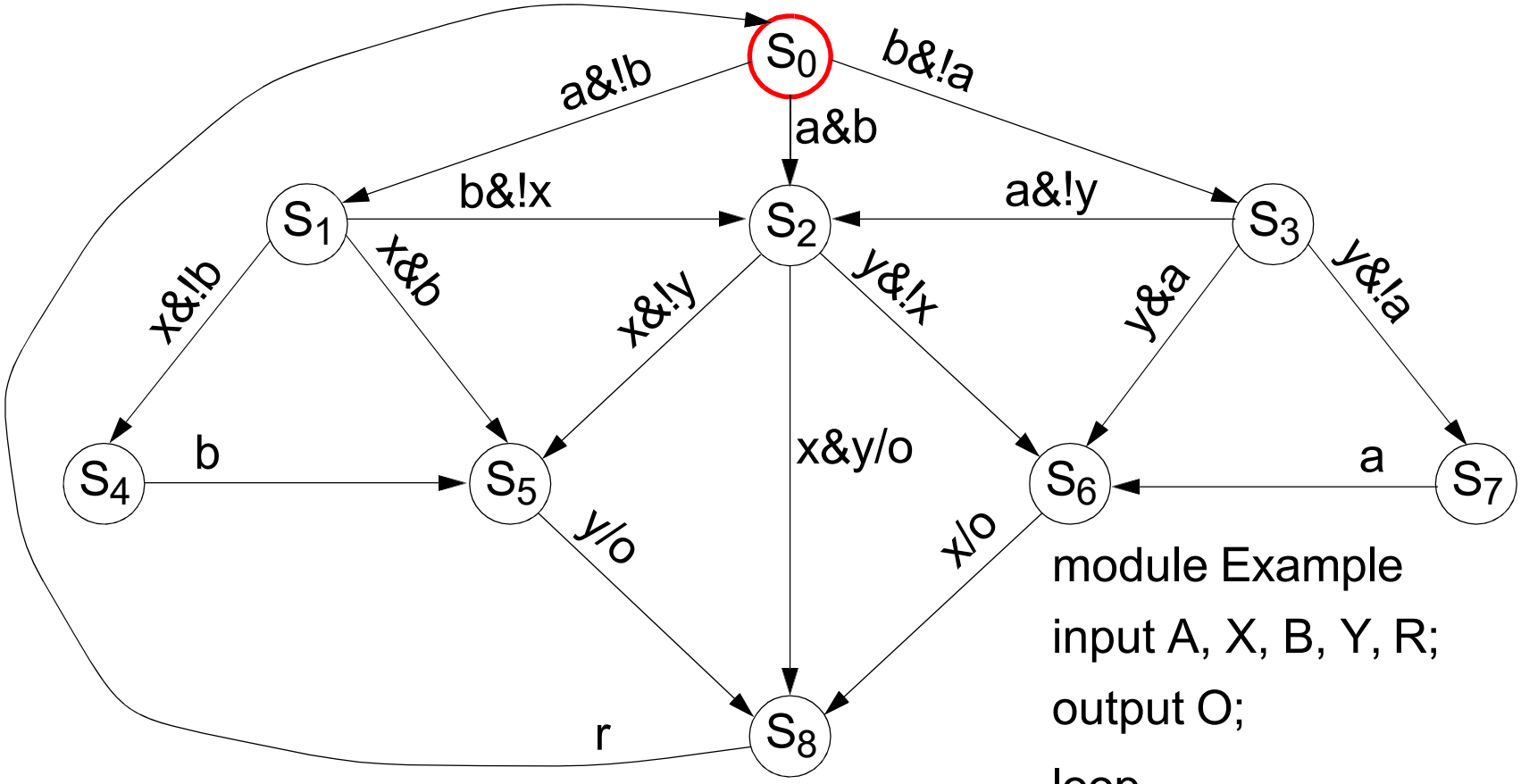
Esterel Example

The *Esterel* program corresponding to the example described earlier as a FSM and, in Statecharts, as a hierarchical concurrent FSM:



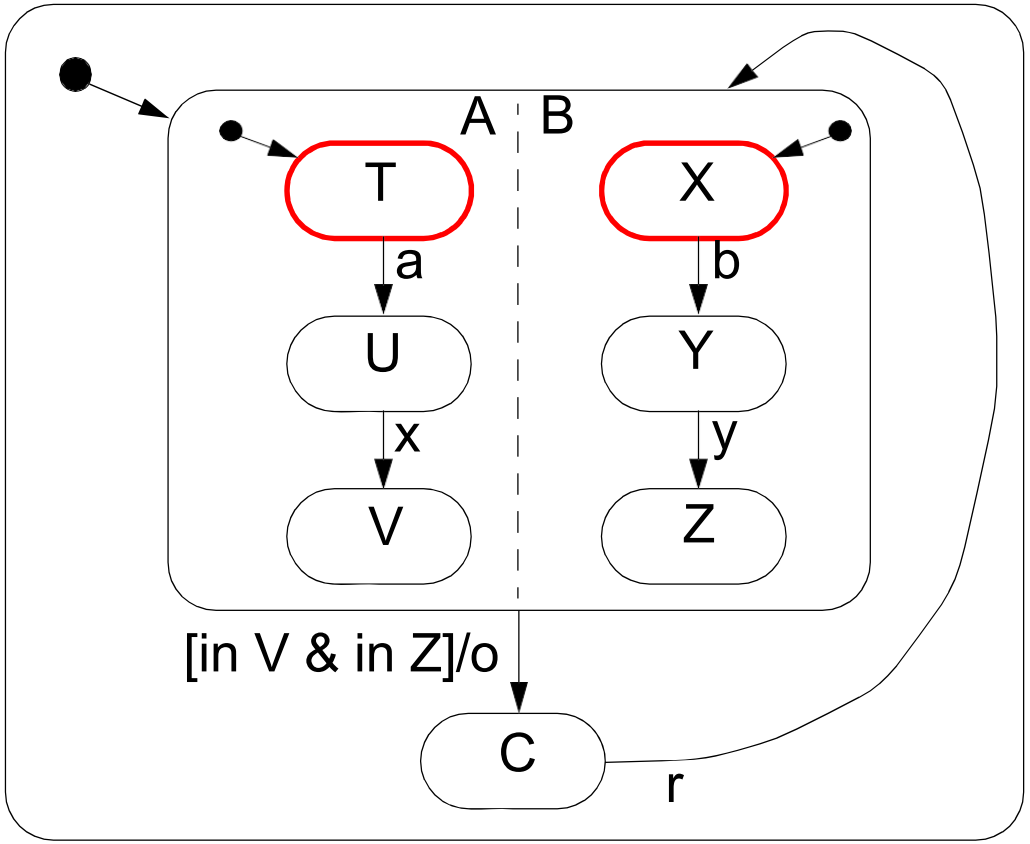

```
module Example
input A, X, B, Y, R;
output O;
loop
    [await A; await X || await B; await Y]
    emit O;
    await R
end loop
end module
```

Esterel Example




```
module Example
input A, X, B, Y, R;
output O;
loop
    [await A; await X || await B; await Y]
    emit O;
    await R
end loop
end module
```

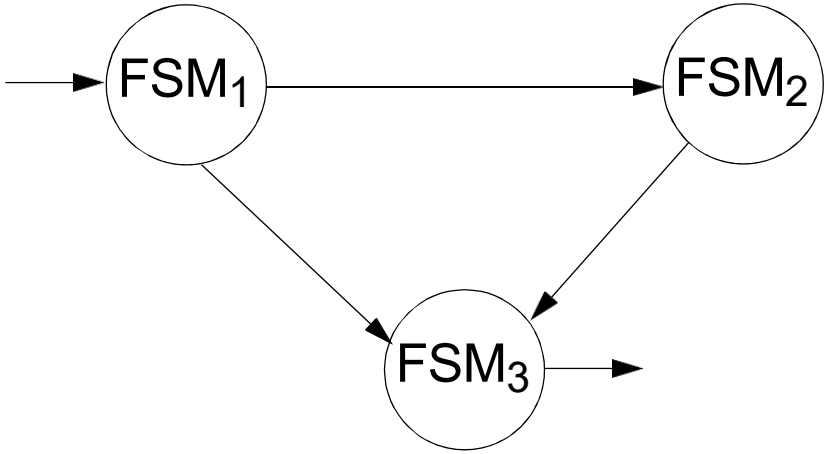
Esterel Example




```
module Example
input A, X, B, Y, R;
output O;
loop
    [await A; await X || await B; await Y]
    emit O;
    await R
end loop
end module
```

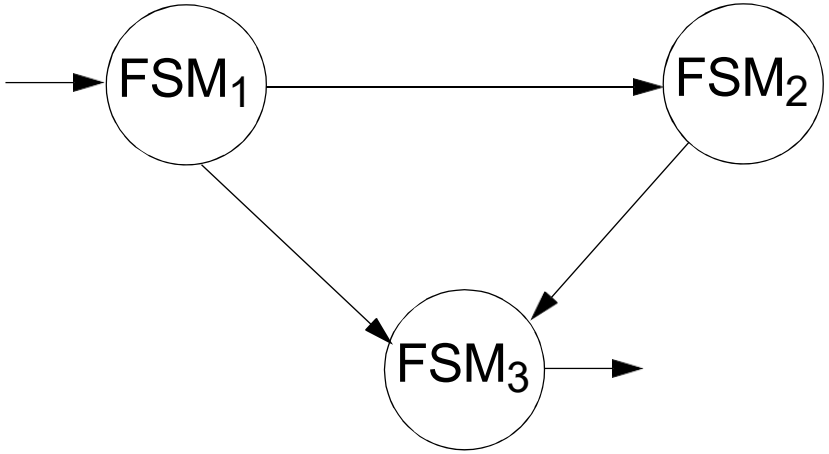
How to implement a synchronous system?

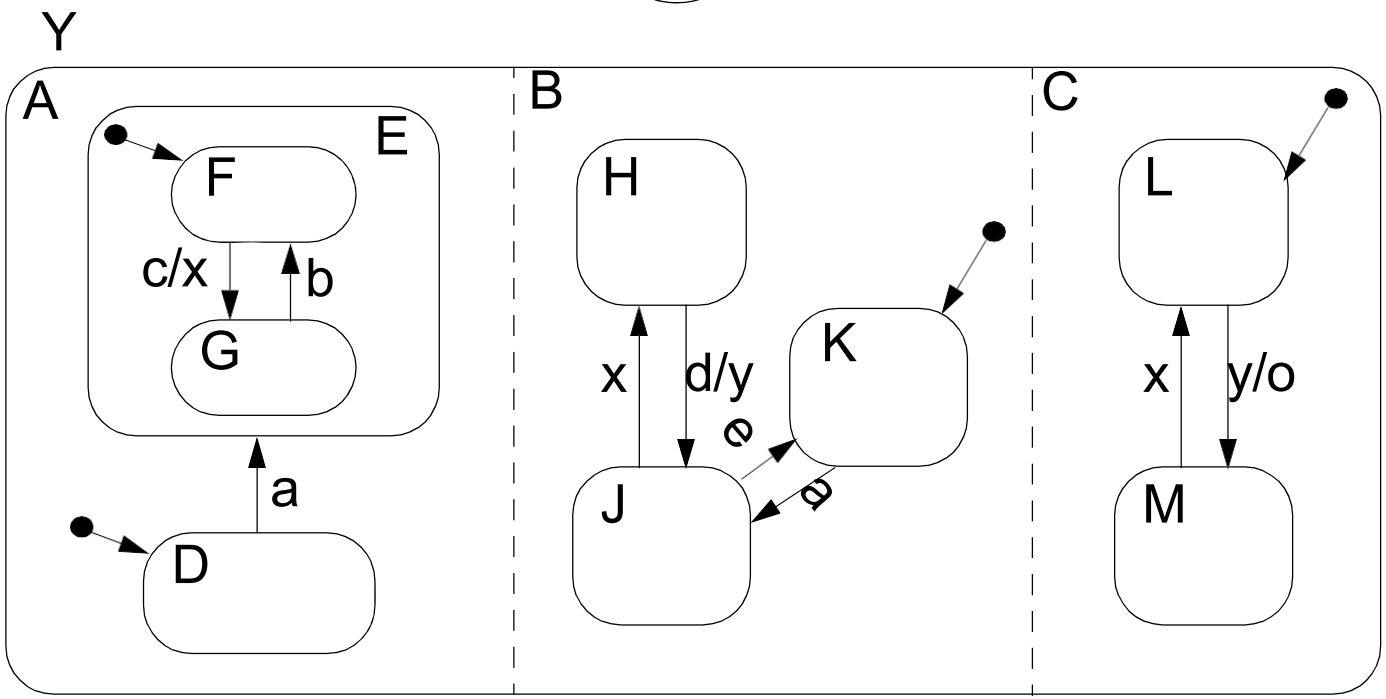
A synchronous model (concurrent FSMs):



How to implement a synchronous system?

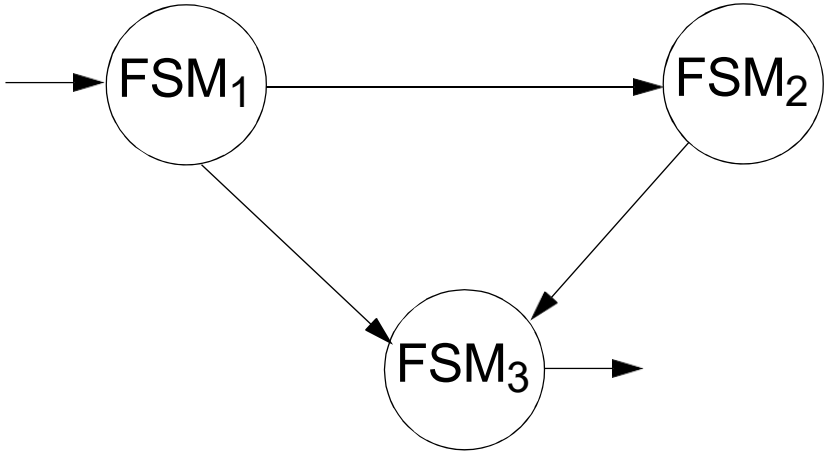
A synchronous model (concurrent FSMs):



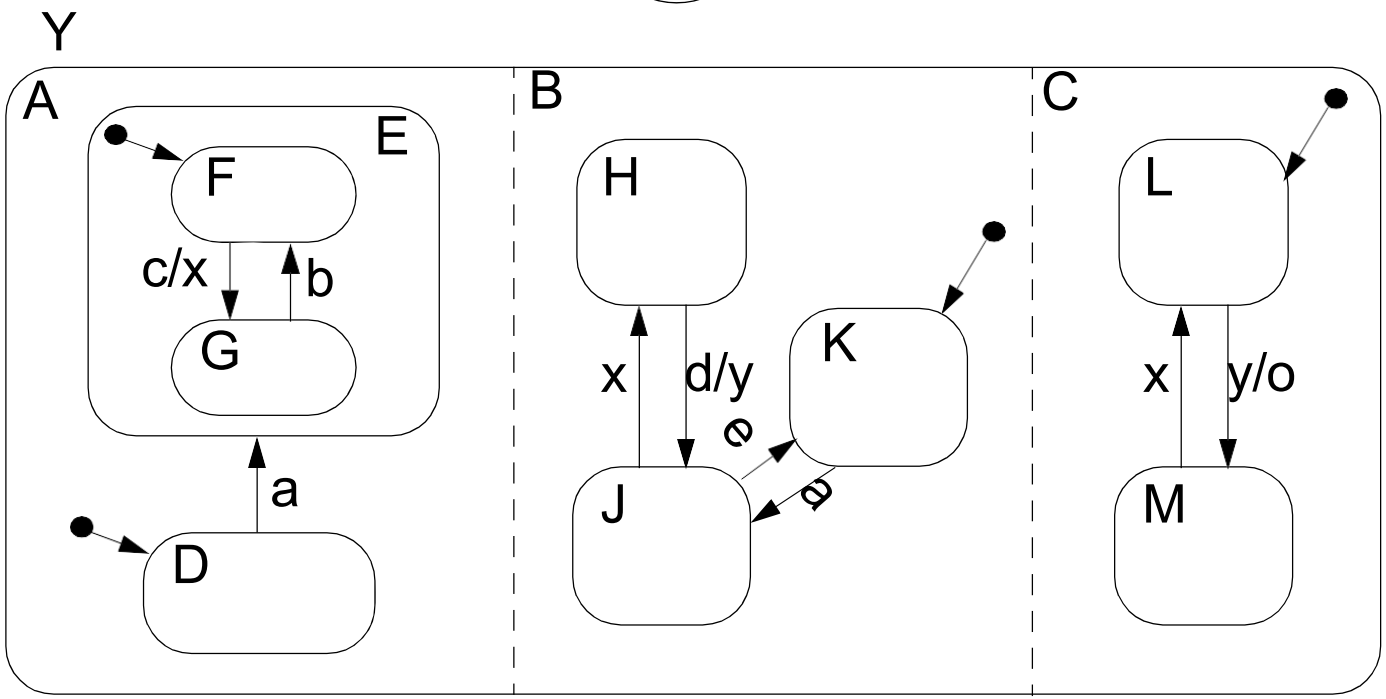


How to implement a synchronous system?

A synchronous model (concurrent FSMs):



- Signals are propagated instantaneously through the system.
- all FSMs react instantaneously to events.
- No buffering.



How to implement a synchronous system?

- In hardware:
  - System described as single FSM:
    - implementation as a state machine.
  - System described as several FSMs:
    - several communicating synchronous state machines or
    - implement the equivalent single (very large) state machine

How to implement a synchronous system?

- In hardware:
  - System described as single FSM:
    - implementation as a state machine.
  - System described as several FSMs:
    - several communicating synchronous state machines or
    - implement the equivalent single (very large) state machine

But if the system is large:

- How do you distribute the clock signal on a large chip, in order to keep synchrony?
- If there are several chips, keeping synchrony is even more difficult.

How to implement a synchronous system?

- In software:
  - One single FSM or several FSMs:

    Generate a sequential program which *emulates the state machine*.

Problems:

- Large concurrent systems $\Rightarrow$ state explosion $\Rightarrow$ very large programs.
- It is practically impossible to implement the software on a large multiprocessor/distributed system (extremely inefficient to keep the global synchrony of such a multiprocessor/distributed software).

How to implement a synchronous system?

- If the model is impossible (or very difficult and expensive) to implement, there is no use that it is elegant, simple, and can be formally verified. *We get a correct verified model but we cannot implement it correctly*!

Synchronous models are very good for relatively small systems implemented in hardware or software.

- For larger systems *we have* to *give up the assumption of global synchrony*.

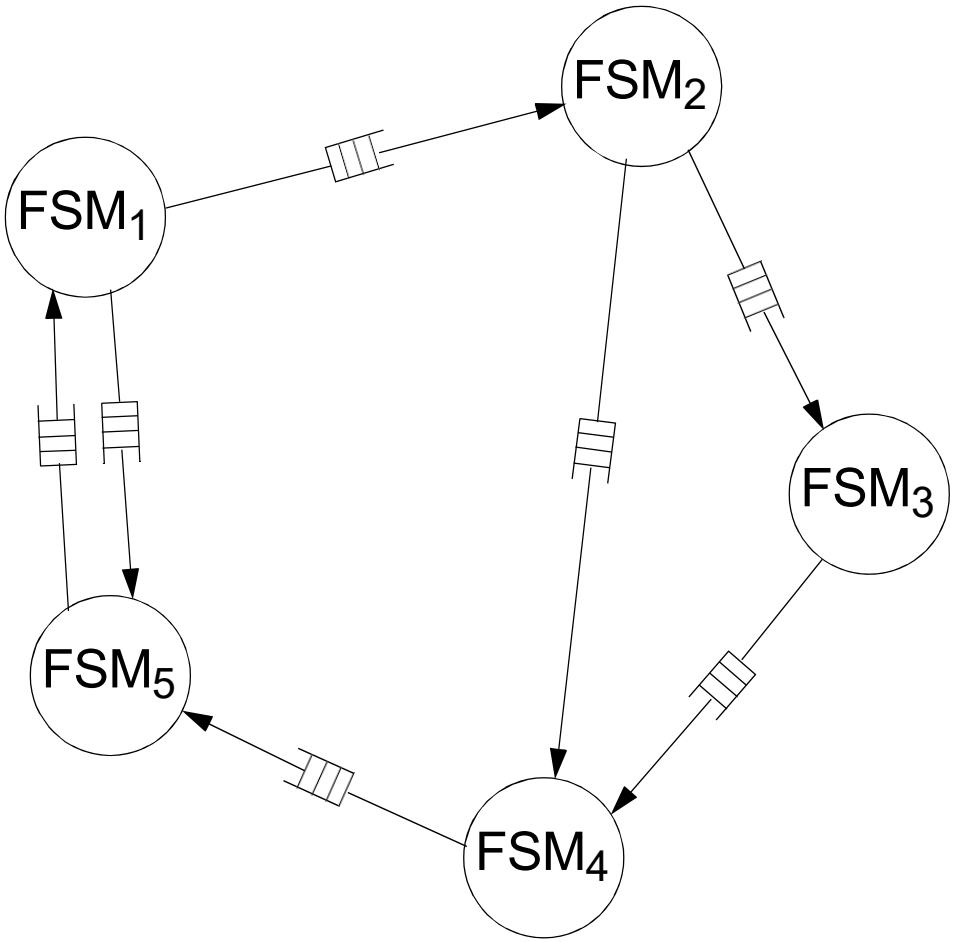
GLOBALLY ASYNCHRONOUS LOCALLY SYNCHRONOUS SYSTEMS

1. Globally Asynchronous Locally Synchronous Systems

2. Globally Asynchronous Locally Synchronous System Models

GALS Systems

Globally asynchronous and locally synchronous (GALS) models:



- Each FSM individually behaves like a synchronous systems ⇒ reacts instantaneously on a set of available inputs and generates output.
- The global system is asynchronous ⇒ communication time is finite and non-zero; reaction time of each FSM, as viewed by other FSMs is finite and non-zero.
- With global asynchrony, buffering of signals could be needed.

GALS Systems

- With a GALS model, the set of FSMs is not any more equivalent with a single FSM (as was the case for the synchronous model).

Several nice features are gone:

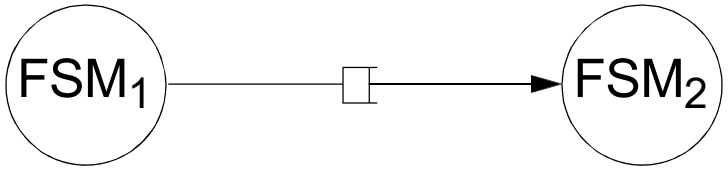
  - With synchronous FSMs we had the nice theoretical background and the possibility of formal verification of the whole system. *Not the case with GALS*.
  - Every implementation of a synchronous FSM model is guaranteed to be functionally equivalent to the initial model and behave exactly and deterministically like the model (in the case we are able to produce an implementation!). *Not the case with GALS*.

GALS Systems

- The GALS model is not deterministic, in the sense that its behavior depends on the amount of time taken for a certain communication or transition.

Two different implementations of the same GALS model can behave differently.

GALS Systems



- A GALS model: FSM_1 and FSM_2 communicate through a single-slot buffer.

- FSM_1 outputs a signal (writes into the buffer) every 2 ms (we neglect communication time).

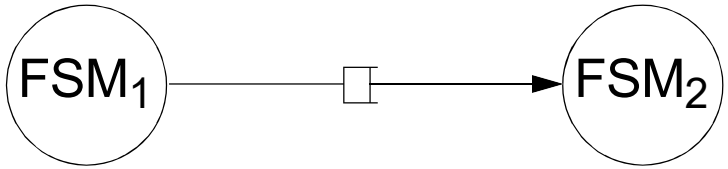
    1. If the reaction time of FSM_2 is 6ms, every third signal from FSM_1 will be reacted on.
    2. If we have a faster implementation of FSM_2, with reaction time 2ms, every signal from FSM_1 will be captured.

GALS Systems

- Each individual FSM can still be verified and formal methods can be used.
- However, individual correctness of each FSM does not guarantee the correctness of the whole system. The system behaves correctly only if, in addition, certain assumptions regarding the timing of components and of communications are satisfied.

GALS Systems

- Each individual FSM can still be verified and formal methods can be used.
- However, individual correctness of each FSM does not guarantee the correctness of the whole system. The system behaves correctly only if, in addition, certain assumptions regarding the timing of components and of communications are satisfied.



  - Each FSM can be functionally verified individually.
  - The global system will be correct (no signal is lost) if FSM_2 has a reaction time which is smaller than the production rate of FSM_1.
  - Estimation and simulation can be used in order to verify that a certain implementation (like FSM_1 as software on a certain μprocessor, and FSM_2 as an ASIC) satisfies this assumption.

GALS System Models

- A GALS system is modelled as a *network of FSMs*:
  - Each FSM has a *locally synchronous* behavior: it executes a transition by producing a single output reaction based on a single, snap-shot input assignment in zero time.
  - A System has a globally asynchronous behavior: each FSM reads inputs, executes a transition, and produces outputs in a finite amount of time as seen by the rest of the system.

GALS System Models

- FSMs communicate through signals.
  - A signal, in general, carries an event and associated data.
  - A signal is communicated between two FSMs via a connection that has an associated input buffer.
  - A sender can communicate a signal to several receivers; each receiver buffers the signal in its own input buffer (of a certain size) associated to the connection.
  - Communication is asynchronous and has undefined (finite) delays. Each input buffer stores the most recently received events and values.
  - In general, the transmitter sends without waiting for the receiver; nothing prevents the transmitter from sending a new event before the last one was consumed and, thus, potentially, overwriting it.

GALS System Models

- A FSM reacts when at least one event is available on any of its inputs; in this case the FSM
  - reads and consumes the available input signal(s);
  - identifies the matching transition and performs the corresponding state transition with the associated action set;
  - writes the outputs associated to the transition.

GALS System Models

- A FSM reacts when at least one event is available on any of its inputs; in this case the FSM
  - reads and consumes the available input signal(s);
  - identifies the matching transition and performs the corresponding state transition with the associated action set;
  - writes the outputs associated to the transition.

- The reaction takes a certain, finite, amount of time.

  After executing a transition, the FSM will be ready to react to new inputs.

  Question: When? Immediately, just after it finished the current transition?

GALS System Models

- A FSM reacts when at least one event is available on any of its inputs; in this case the FSM
  - reads and consumes the available input signal(s);
  - identifies the matching transition and performs the corresponding state transition with the associated action set;
  - writes the outputs associated to the transition.

- The reaction takes a certain, finite, amount of time.

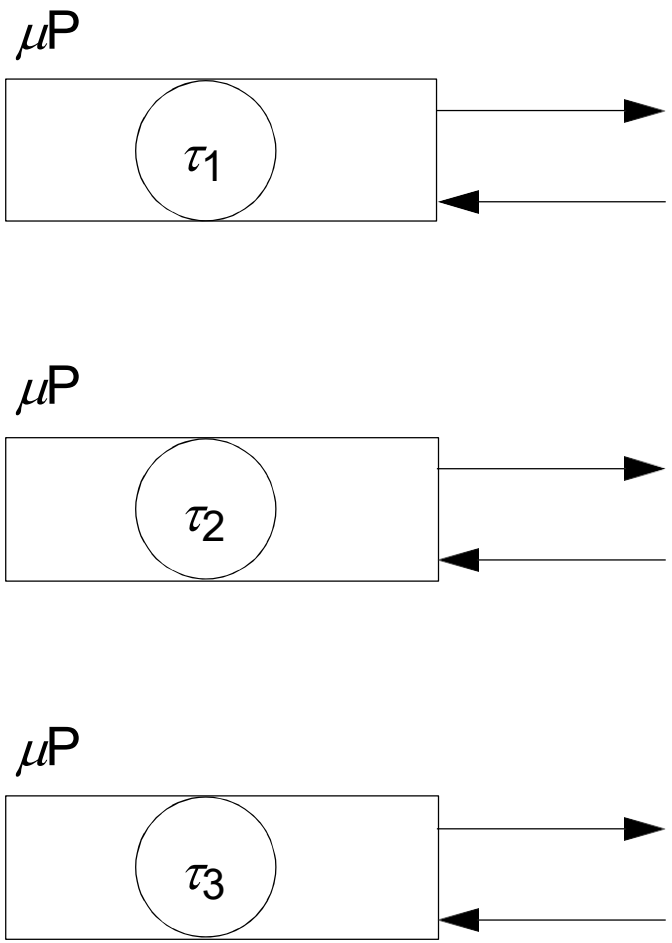
  After executing a transition, the FSM will be ready to react to new inputs.

  Question: When? Immediately, just after it finished the current transition?

  Answer: Not necessarily!

  When a certain FSM is ready to check inputs and react, depends on the, execution platform, the execution times, periods, and the scheduling policy used at implementation.

GALS System Models



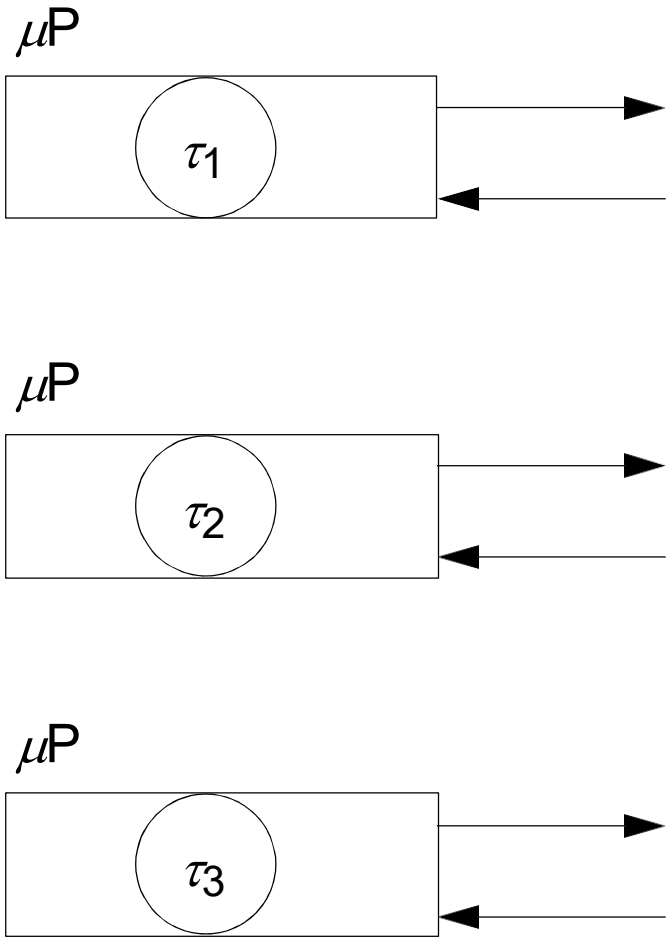
Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

Each task implements an FSM (in software).

GALS System Models



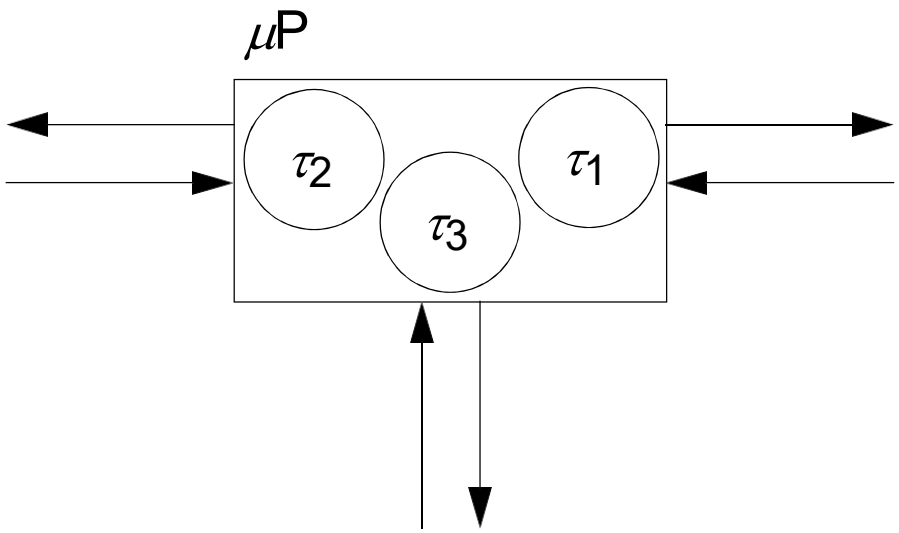
Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

Works! No problem!

GALS System Models

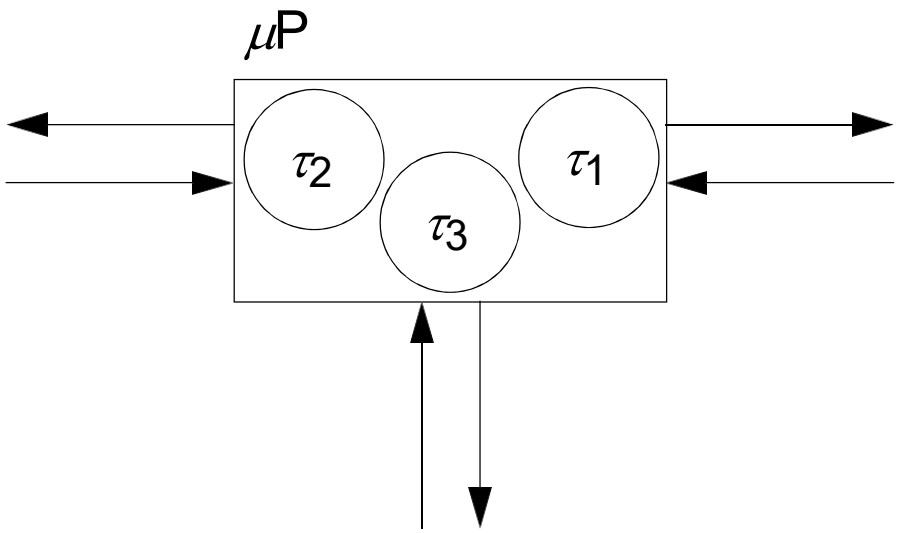


Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

GALS System Models



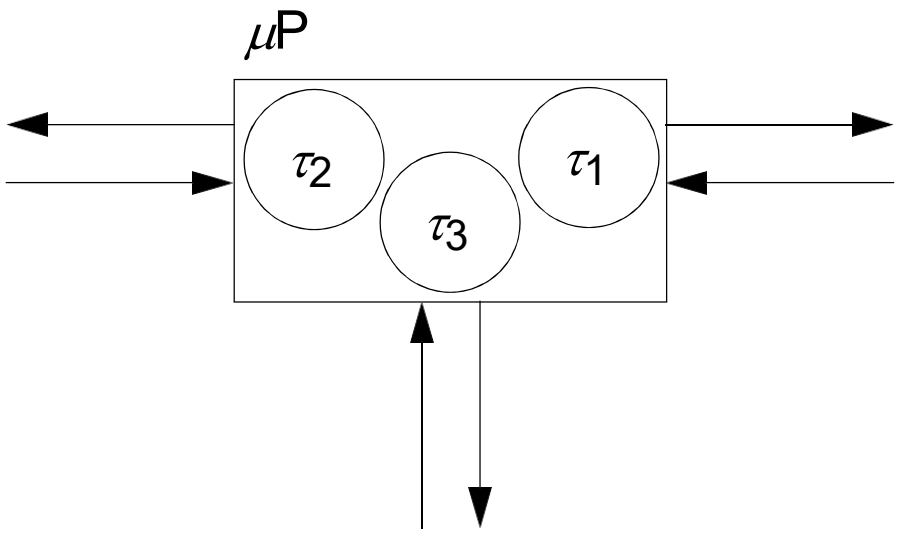
Does this work?
Can each of the tasks work at the required rate (period)?

Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

GALS System Models



Task τ_1 { Period T_1 = 100 μs; WCET C_1 = 40 μs

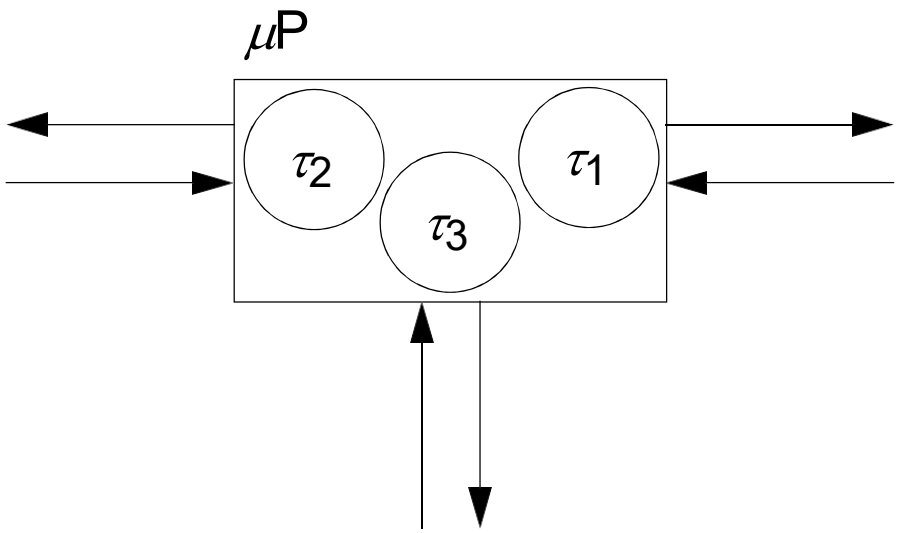
Task τ_2 { Period T_2 = 30 μs; WCET C_2 = 10 μs

Task τ_3 { Period T_3 = 25 μs; WCET C_3 = 10 μs

Does this work?
Can each of the tasks work at the required rate (period)?

- τ_1 needs to run for 40 μs every 100 μs: 40% of CPU
- τ_2 needs to run for 10 μs every 30 μs: 33% of CPU
- τ_3 needs to run for 10 μs every 25 μs: 40% of CPU

GALS System Models



Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

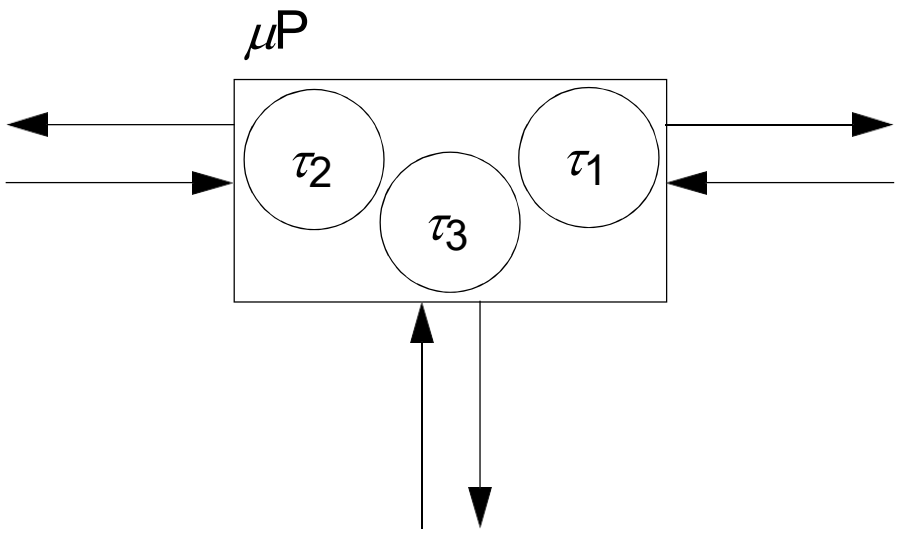
Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

Does this work?
Can each of the tasks work at the required rate (period)?

- τ_1 needs to run for 40 μs every 100 μs: 40% of CPU
- τ_2 needs to run for 10 μs every 30 μs: 33% of CPU
- τ_3 needs to run for 10 μs every 25 μs: 40% of CPU

Total: 113%

GALS System Models



Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

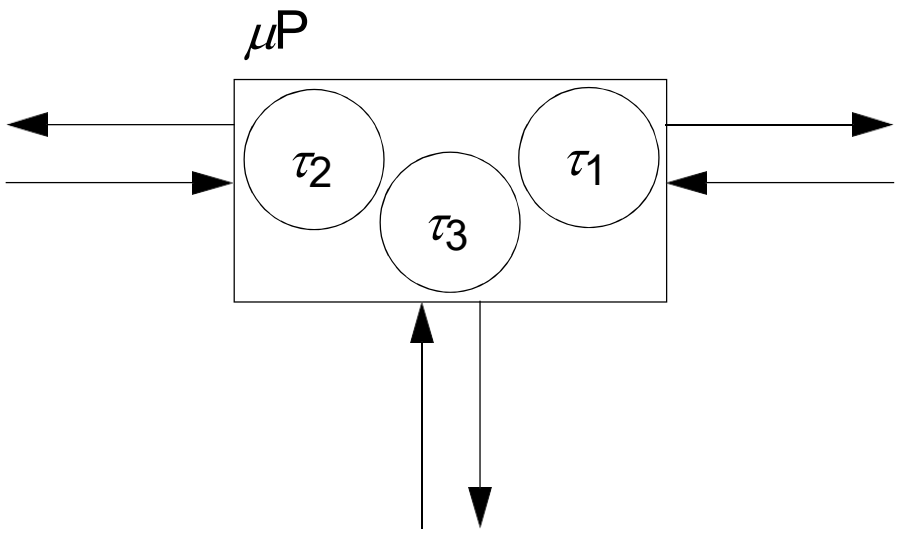
Does this work?
Can each of the tasks work at the required rate (period)?

- τ_1 needs to run for 40 μs every 100 μs: 40% of CPU
- τ_2 needs to run for 10 μs every 30 μs: 33% of CPU
- τ_3 needs to run for 10 μs every 25 μs: 40% of CPU

Total: 113%

This will not work!

GALS System Models



Task τ_1
- Period T_1 = 100 μs
- WCET C_1 = 40 μs

Task τ_2
- Period T_2 = 30 μs
- WCET C_2 = 10 μs

Task τ_3
- Period T_3 = 25 μs
- WCET C_3 = 10 μs

Does this work?
Can each of the tasks work at the required rate (period)?

If the total utilisation is not larger than 100% it is possible to implements the tasks!