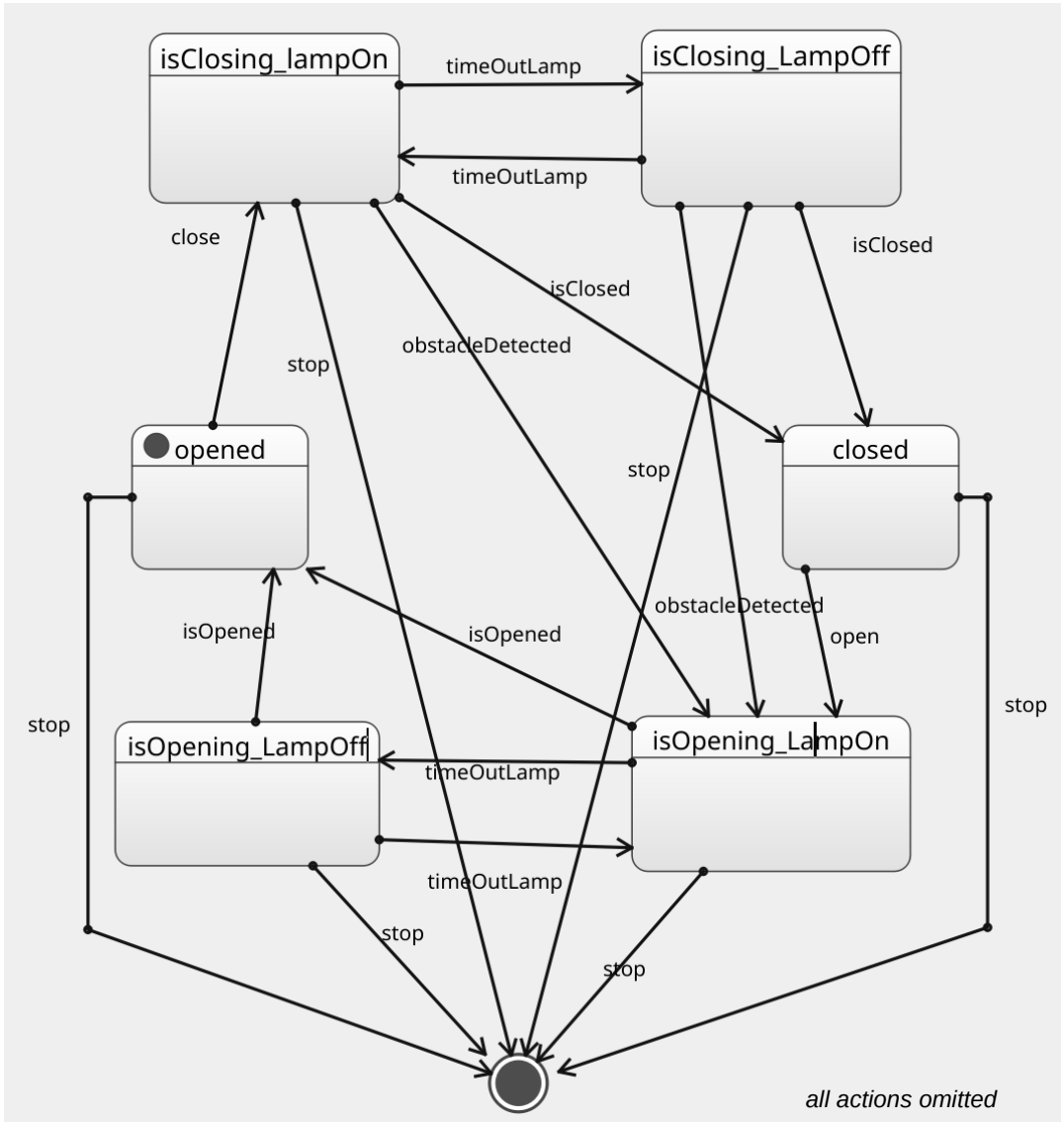
SCXML
State Chart XML

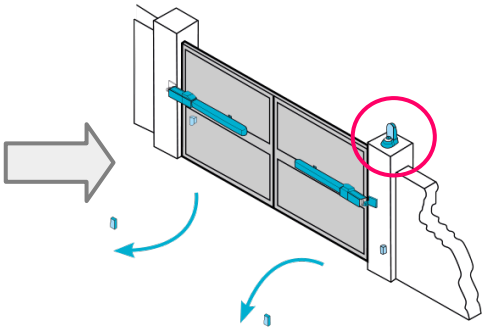
Previously, in this course...

Running Example

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.





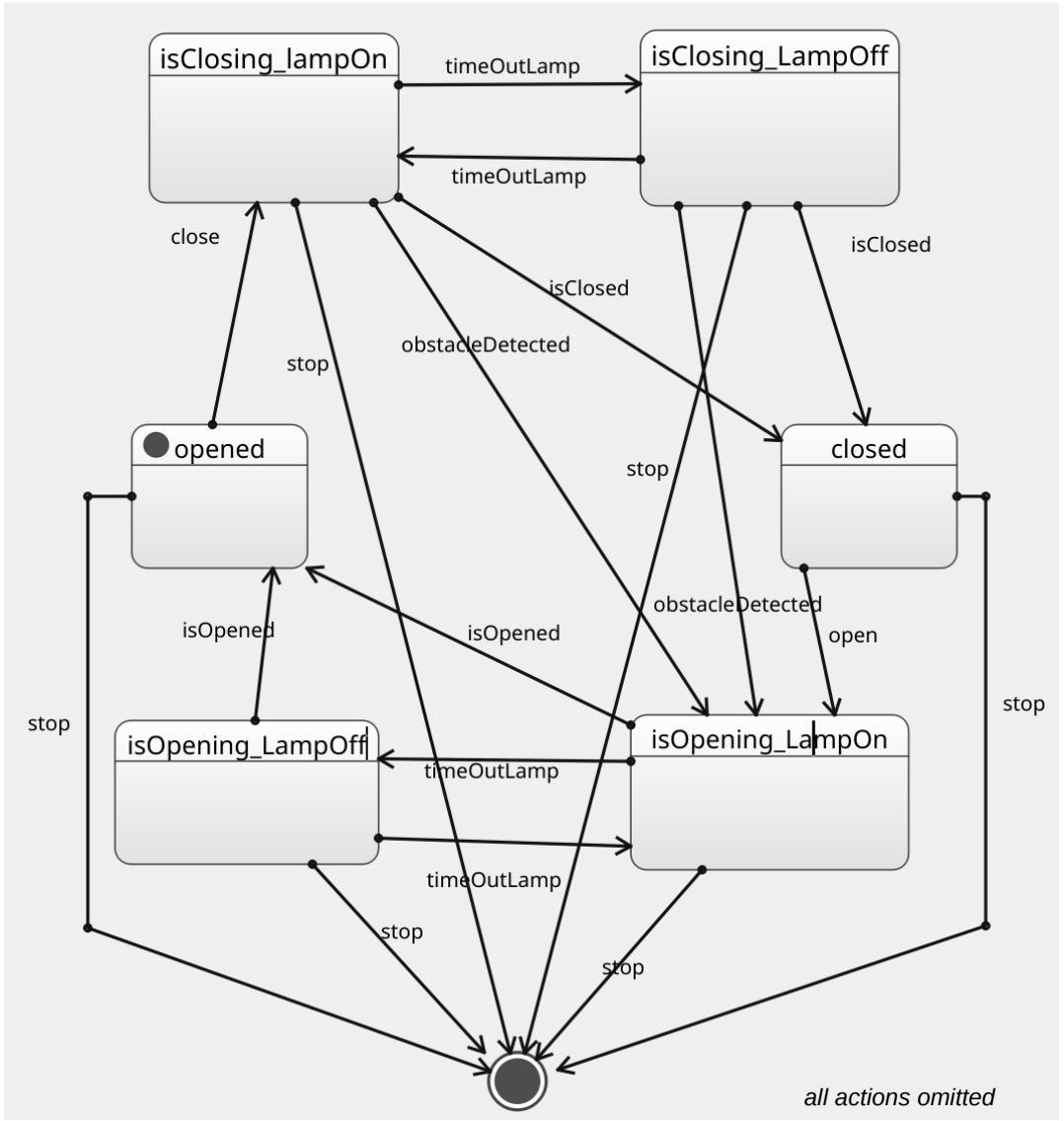
wasn't it supposed to help ?

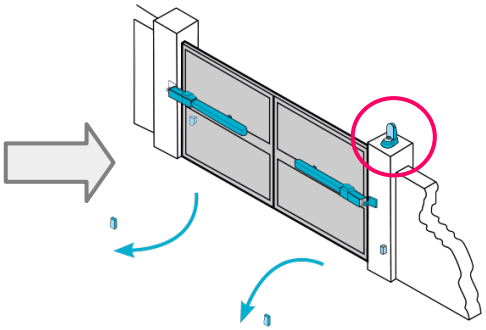
Previously, in this course...

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.

Running Example



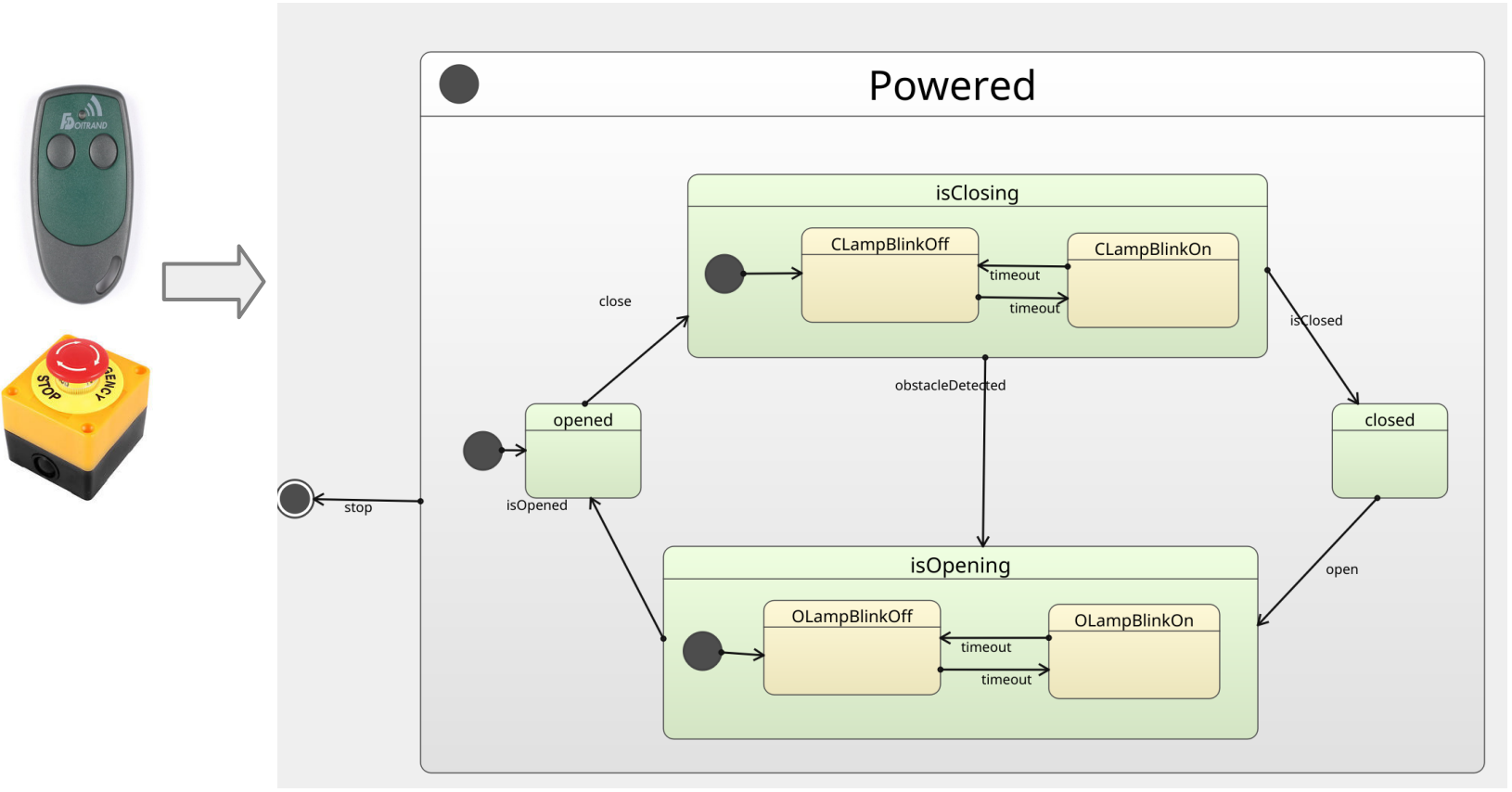


- Substates may be nested to any level. A nested state machine may have at most one initial state and one final state.
- **Substates are used to simplify complex flat state machines by showing that some states are only possible within a particular context (the enclosing state)**.
- **A composite state factorizes the possible exits from all (most of) the states**

Running Example

statecharts = state-diagrams + depth

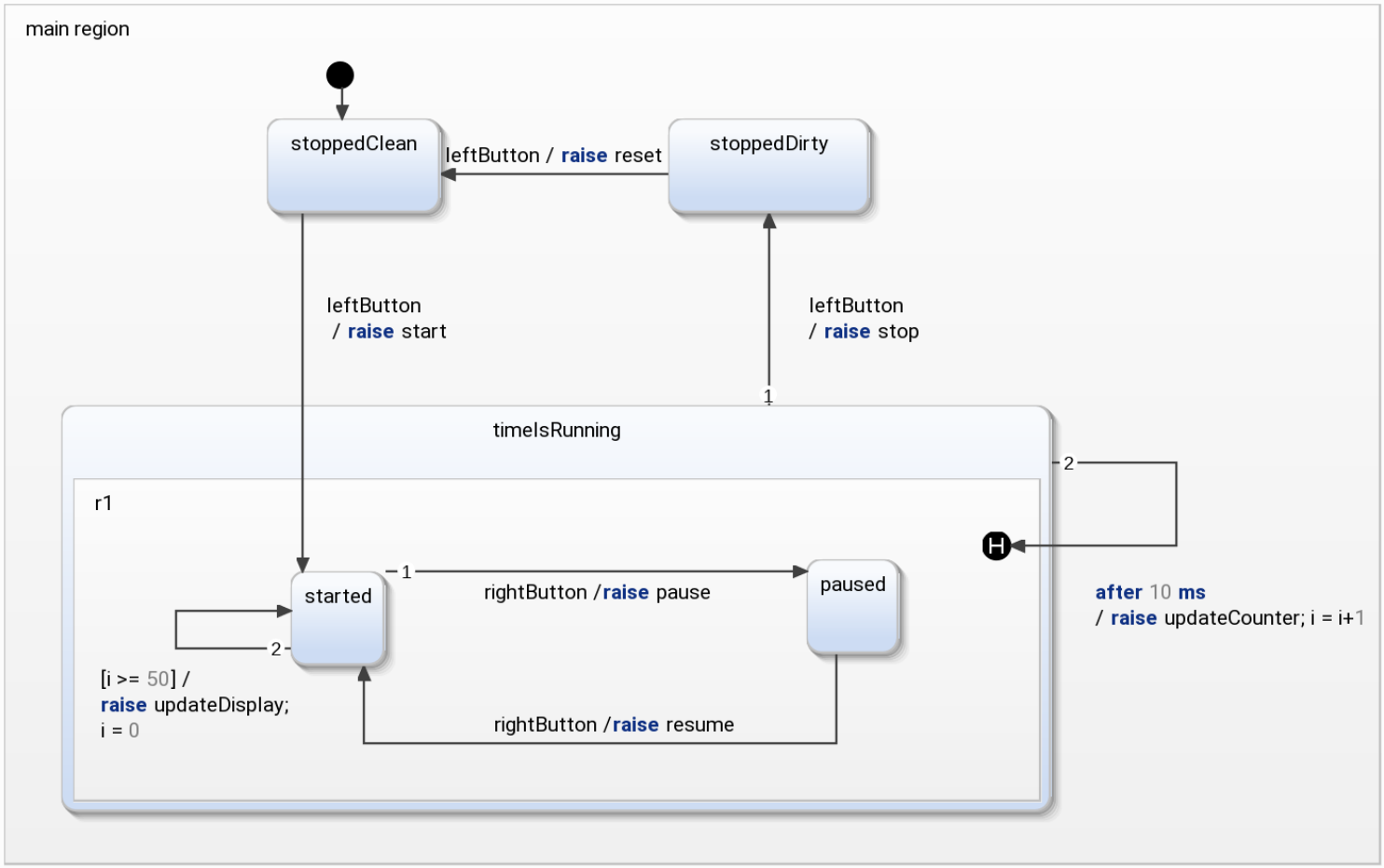
+ orthogonality + broadcast-communication.



all actions omitted

- A simple state is one which has no substructure.
- A state which has substates (nested states) is called a composite state (or compound state).
- Substates may be nested to any level. A nested state machine may have at most one initial state and one final state.
- **Substates are used to simplify complex flat state machines by showing that some states are only possible within a particular context (the enclosing state)**.
- **A composite state factorizes the possible exits from all (most of) the states**

Stopwatch

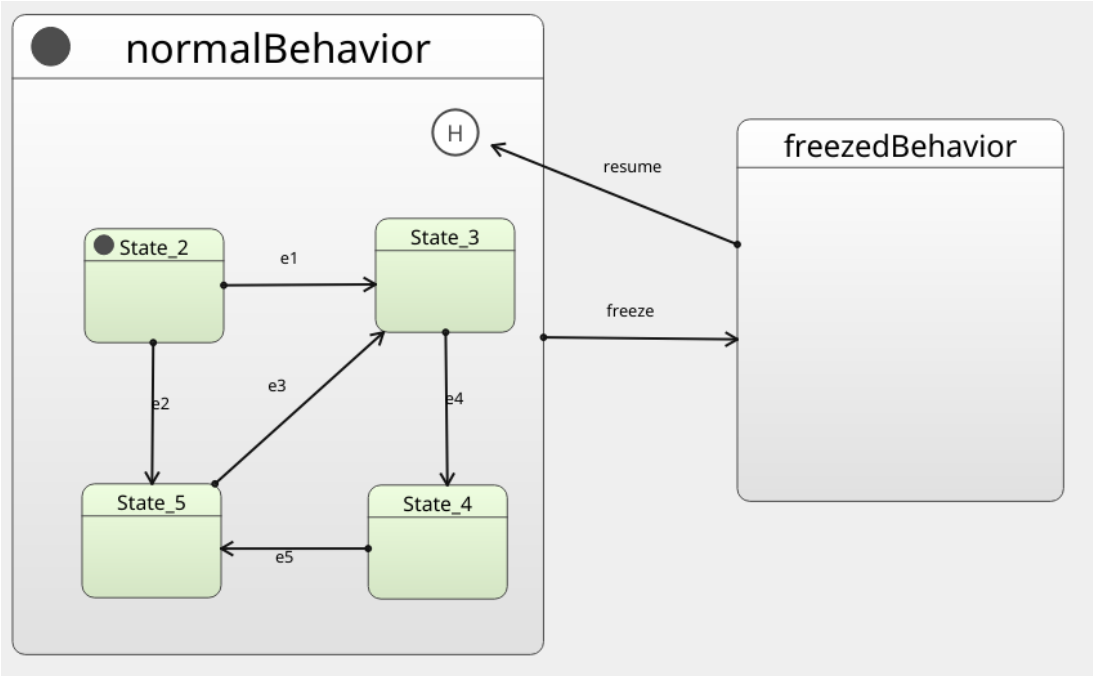


- A simple state is one which has no substructure.
- A state which has substates (nested states) is called a composite state (or compound state).
- Substates may be nested to any level. A nested state machine may have at most one initial state and one final state.
- **Substates are used to simplify complex flat state machines by showing that some states are only possible within a particular context (the enclosing state)**.
- **A composite state factorizes the possible exits from all (most of) the states**

Taken and modified from http://sce.uhcl.edu/helm/rationalunifiedprocess/process/modguide/md_stadm.htm

History state

Deep or shallow...



- **<history> allows for pause and resume semantics in compound states**. Before the state machine exits a compound state, it records the state's active descendants. If the 'type' attribute of the <history> state is set to "deep", the state machine saves the state's full active descendant configuration, down to the atomic descendant(s). If 'type' is set to "shallow", the state machine remembers only which immediate child was active. After that, if a transition takes a <history> child of the state as its target, the state machine re-enters not only the parent compound state but also the state(s) in the saved configuration. Thus a transition with a deep history state as its target returns to exactly where the state was when it was last exited, while a transition with a shallow history state as a target re-enters the previously active child state, but will enter the child's default initial state (if the child is itself compound.).

TD stopWatch

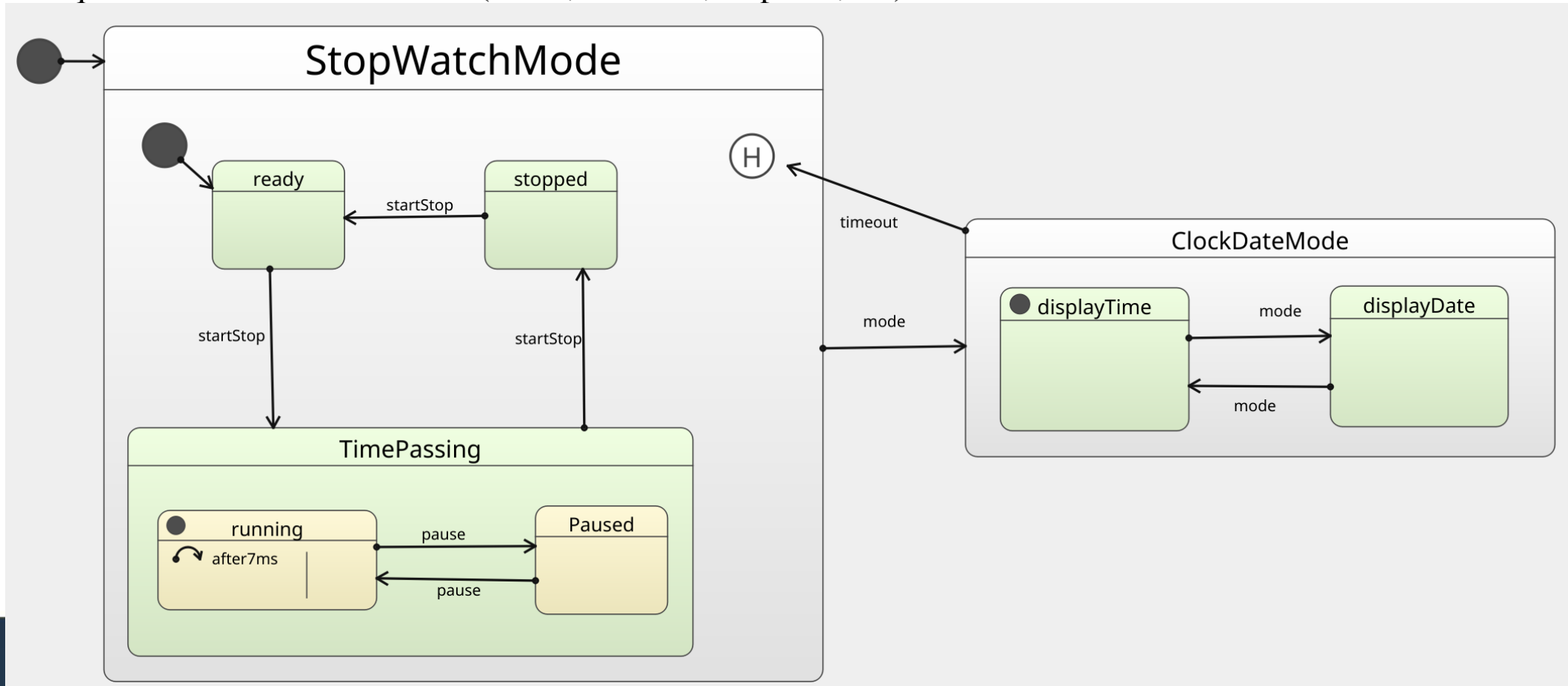
Version 3

Vous allez ajouter du comportement à la version 2. Il s'agit ici d'ajouter un troisième bouton "mode". Ce bouton permet d'afficher l'heure ou la date pendant 1 seconde. Au premier appuie sur le bouton, l'affichage devra montrer l'heure courante. Au bout d'une seconde, si le bouton n'a pas été appuyé à nouveau, le chronomètre retourne dans l'état où il était. Si le bouton est appuyé à nouveau avant 1 seconde, l'affichage montrera la date. De même, si aucun bouton n'est appuyé pendant 1 seconde on retourne au comportement initial, sinon on remontre l'heure courante. Le fonctionnement décrit sera actif peu importe l'état dans lequel se trouve le chronomètre (arrêté, démarré , en pause, etc).

TD stopWatch

Version 3

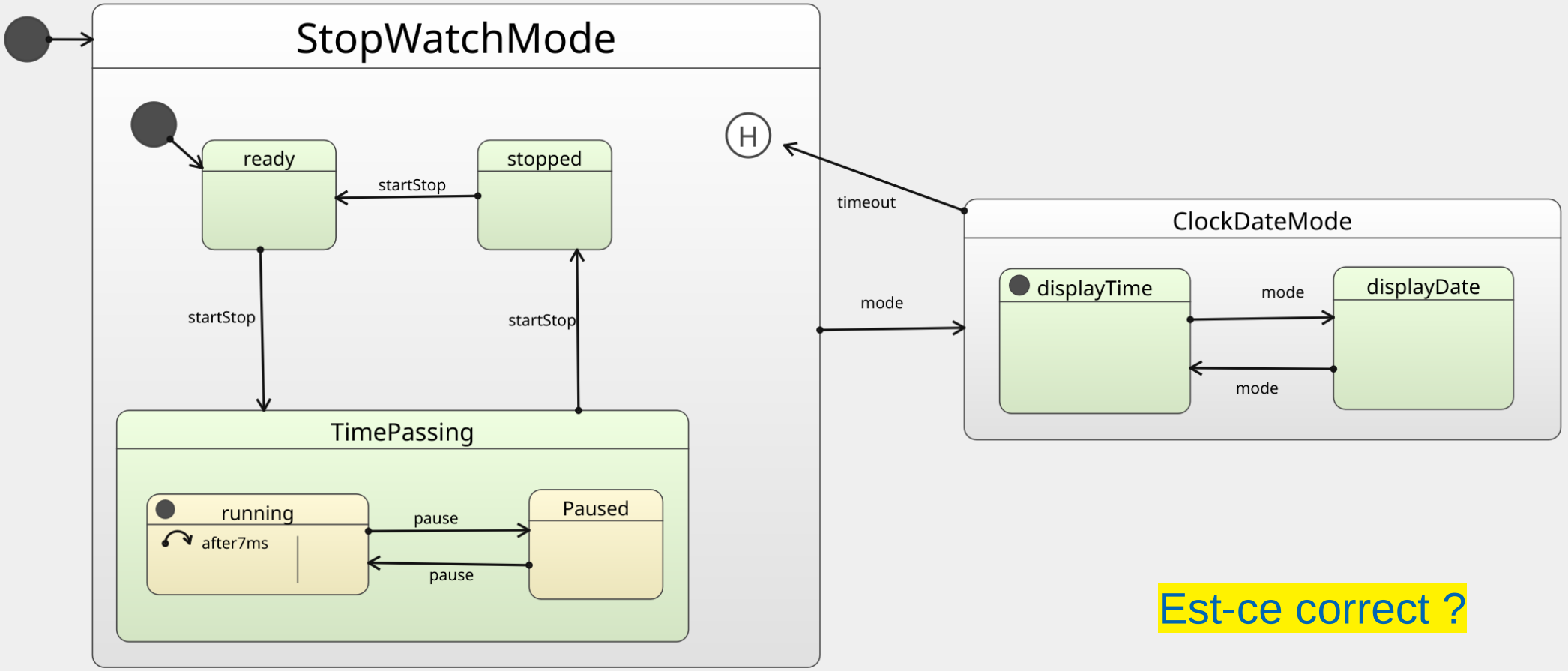
Vous allez ajouter du comportement à la version 2. Il s'agit ici d'ajouter un troisième bouton "mode". Ce bouton permet d'afficher l'heure ou la date pendant 1 seconde. Au premier appuie sur le bouton, l'affichage devra montrer l'heure courante. Au bout d'une seconde, si le bouton n'a pas été appuyé à nouveau, le chronomètre retourne dans l'état où il était. Si le bouton est appuyé à nouveau avant 1 seconde, l'affichage montrera la date. De même, si aucun bouton n'est appuyé pendant 1 seconde on retourne au comportement initial, sinon on remontre l'heure courante. Le fonctionnement décrit sera actif peu importe l'état dans lequel se trouve le chronomètre (arrêté, démarré , en pause, etc).



TD stopWatch

Version 3

Vous allez ajouter du comportement à la version 2. Il s'agit ici d'ajouter un troisième bouton "mode". Ce bouton permet d'afficher l'heure ou la date pendant 1 seconde. Au premier appuie sur le bouton, l'affichage devra montrer l'heure courante. Au bout d'une seconde, si le bouton n'a pas été appuyé à nouveau, le chronomètre retourne dans l'état où il était. Si le bouton est appuyé à nouveau avant 1 seconde, l'affichage montrera la date. De même, si aucun bouton n'est appuyé pendant 1 seconde on retourne au comportement initial, sinon on remontre l'heure courante. Le fonctionnement décrit sera actif peu importe l'état dans lequel se trouve le chronomètre (arrêté, démarré , en pause, etc).

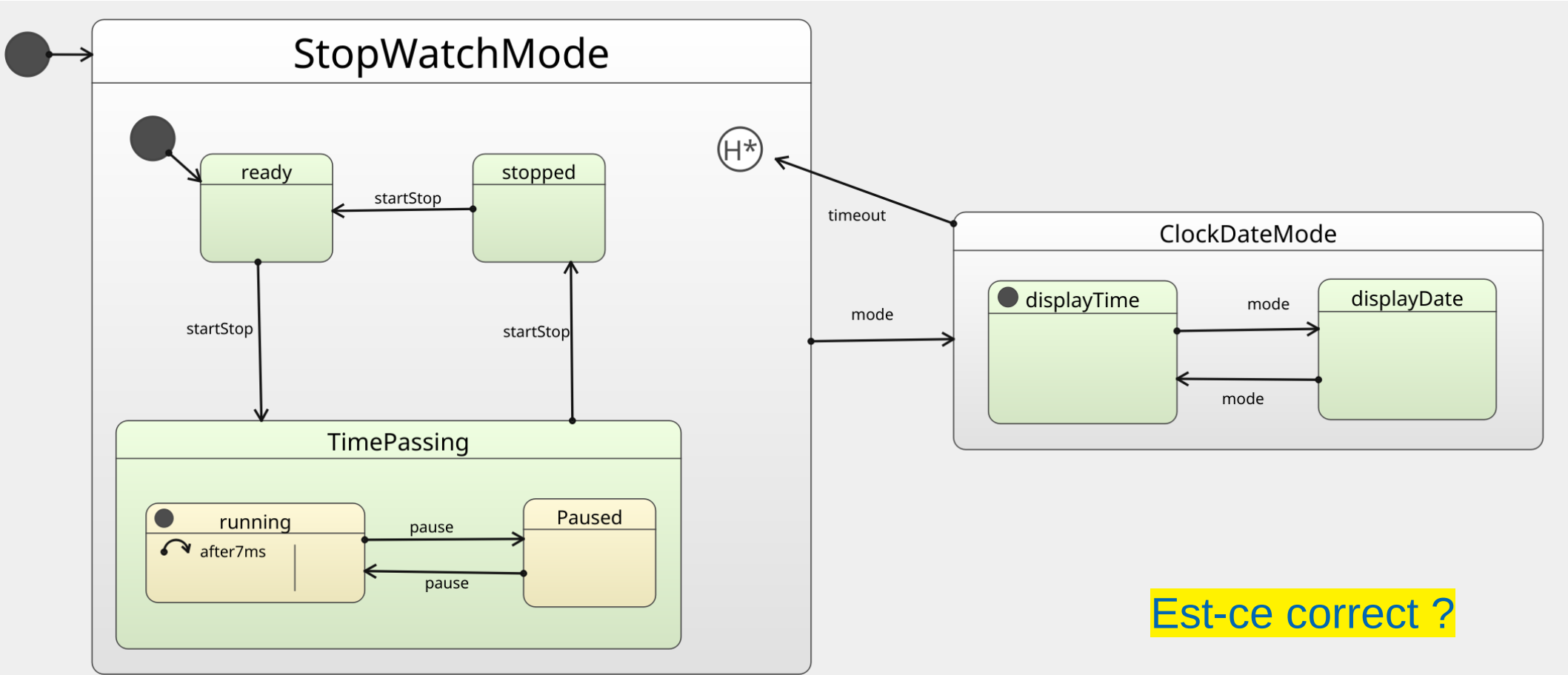


Est-ce correct ?

TD stopWatch

Version 3

Vous allez ajouter du comportement à la version 2. Il s'agit ici d'ajouter un troisième bouton "mode". Ce bouton permet d'afficher l'heure ou la date pendant 1 seconde. Au premier appuie sur le bouton, l'affichage devra montrer l'heure courante. Au bout d'une seconde, si le bouton n'a pas été appuyé à nouveau, le chronomètre retourne dans l'état où il était. Si le bouton est appuyé à nouveau avant 1 seconde, l'affichage montrera la date. De même, si aucun bouton n'est appuyé pendant 1 seconde on retourne au comportement initial, sinon on remontre l'heure courante. Le fonctionnement décrit sera actif peu importe l'état dans lequel se trouve le chronomètre (arrêté, démarré , en pause, etc).



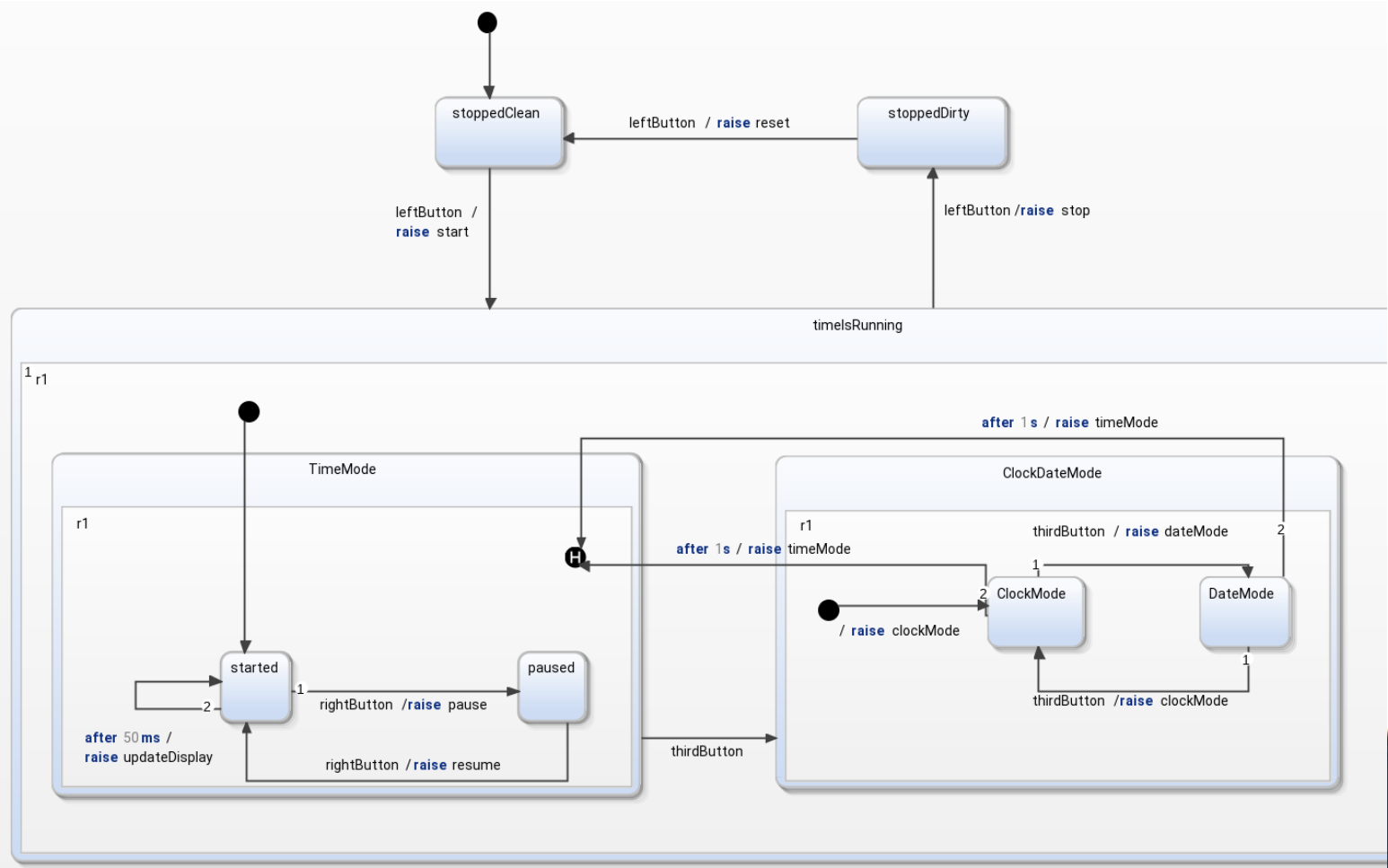
Est-ce correct ?

TD stopWatch and hierarchy !

revisited

Version 3

Vous allez ajouter du comportement à la version 2. Il s'agit ici d'ajouter un troisième bouton "mode". Ce bouton permet d'afficher l'heure ou la date pendant 1 seconde. Au premier appuie sur le bouton, l'affichage devra montrer l'heure courante. Au bout d'une seconde, si le bouton n'a pas été appuyé à nouveau, le chronomètre retourne dans l'état où il était. Si le bouton est appuyé à nouveau avant 1 seconde, l'affichage montrera la date. De même, si aucun bouton n'est appuyé pendant 1 seconde on retourne au comportement initial, sinon on remontre l'heure courante. Le fonctionnement décrit sera actif peu importe l'état dans lequel se trouve le chronomètre (arrêté, démarré , en pause, etc).



SCXML
State Chart XML

Parallel regions, communications
and a little bit more

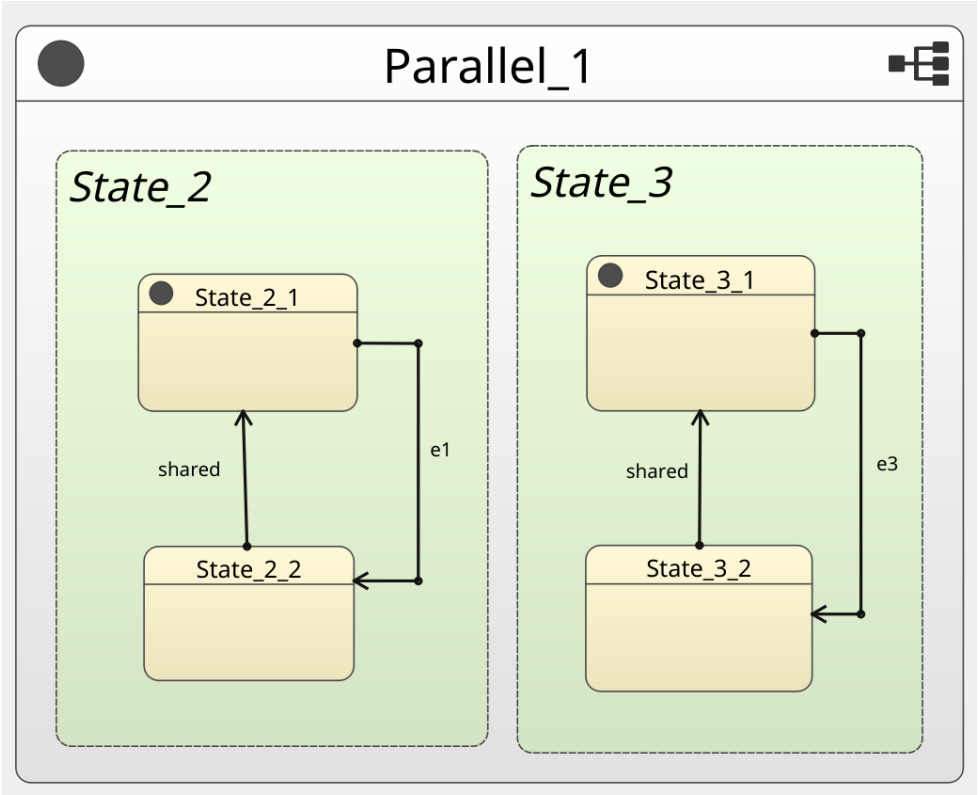
State Charts

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.

David Harel
Statecharts: A visual formalism for complex systems
Science of computer programming 8 (3), 231-274
1987

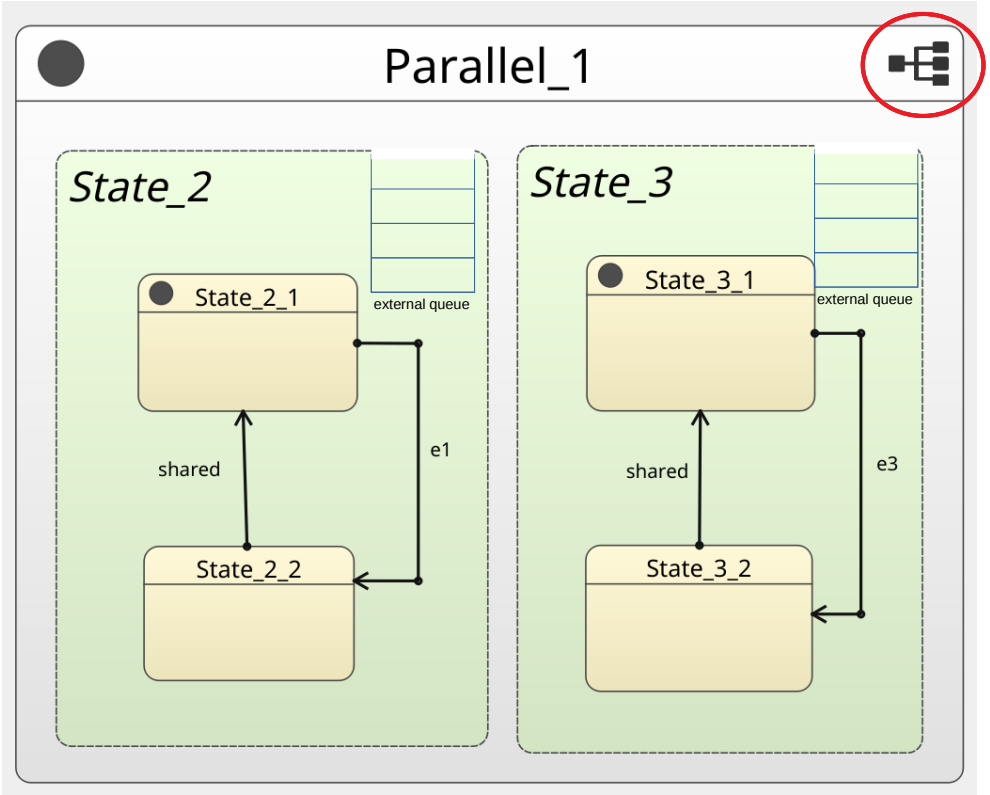
Parallel states



Also named Orthogonal regions in UML or AND-States (Since after initialization, the machine is in substate *State_2_1* AND *State_3_1.*)

Everything is working as if State_2 and State_3 were independent. When the machine is in a parallel state, all event received are broadcasted to all the parallel states. They can consume them or not like any normal state machine

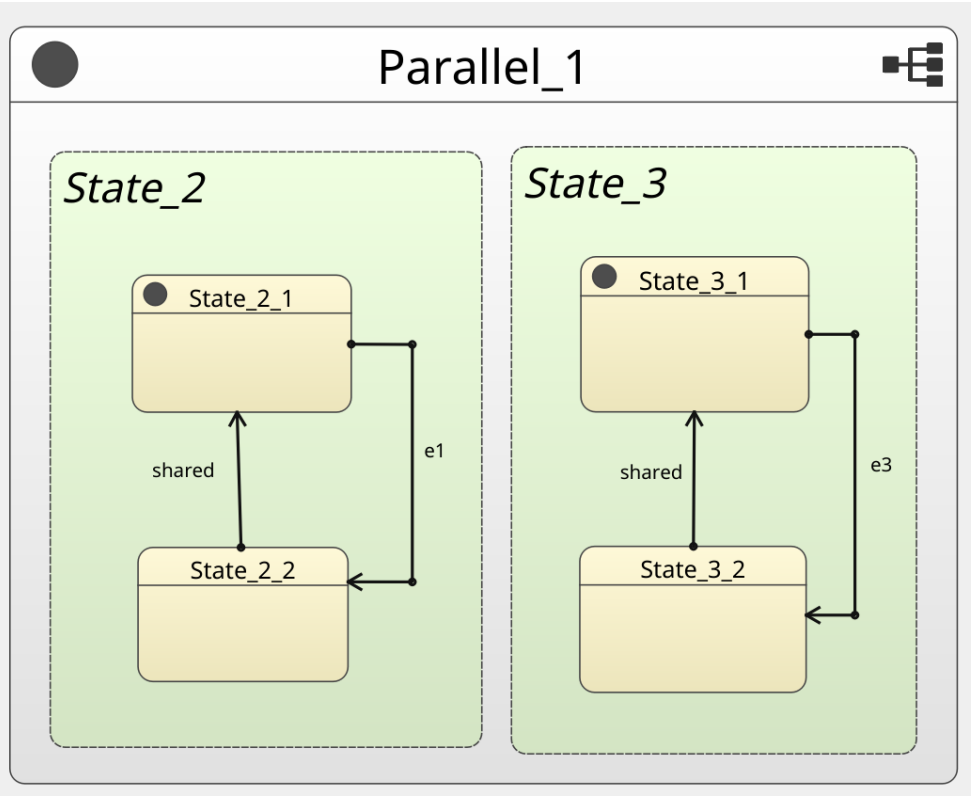
Parallel states

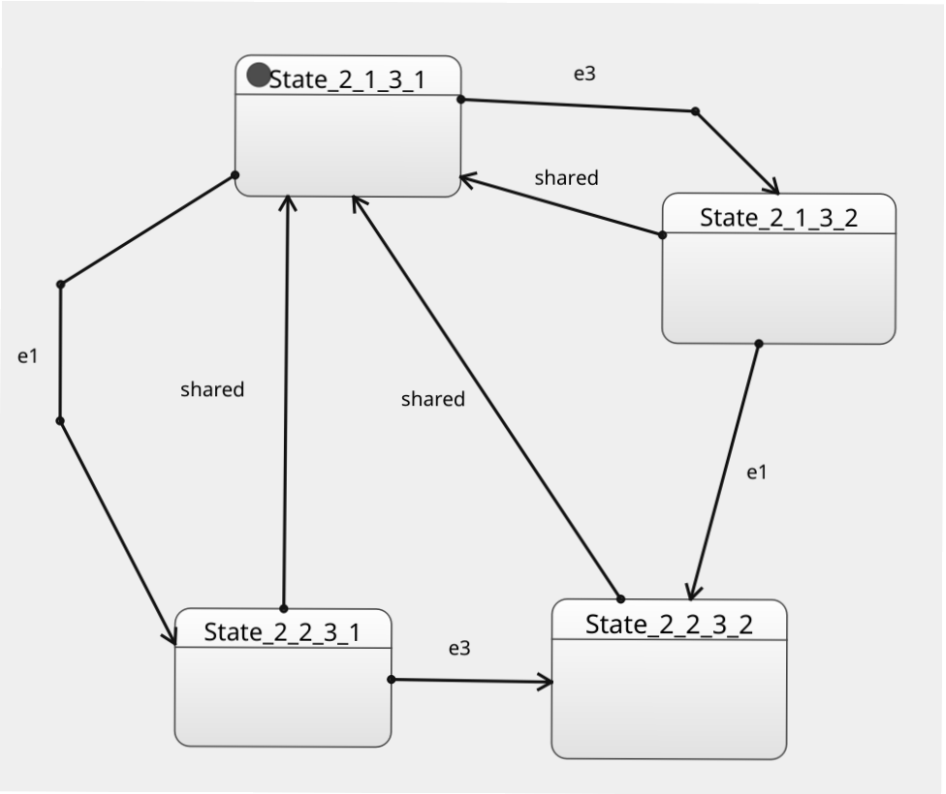


Also named Orthogonal regions in UML or AND-States (Since after initialization, the machine is in substate *State_2_1* AND *State_3_1.*)

Everything is working as if State_2 and State_3 were independent. When the machine is in a parallel state, all event received are broadcasted to all the parallel states. They can consume them or not like any normal state machine

Parallel states





The Cartesian product of the states allows for “sequential” equivalence of the input language but warning about the actions, *i.e.*, the behavior of the system !!

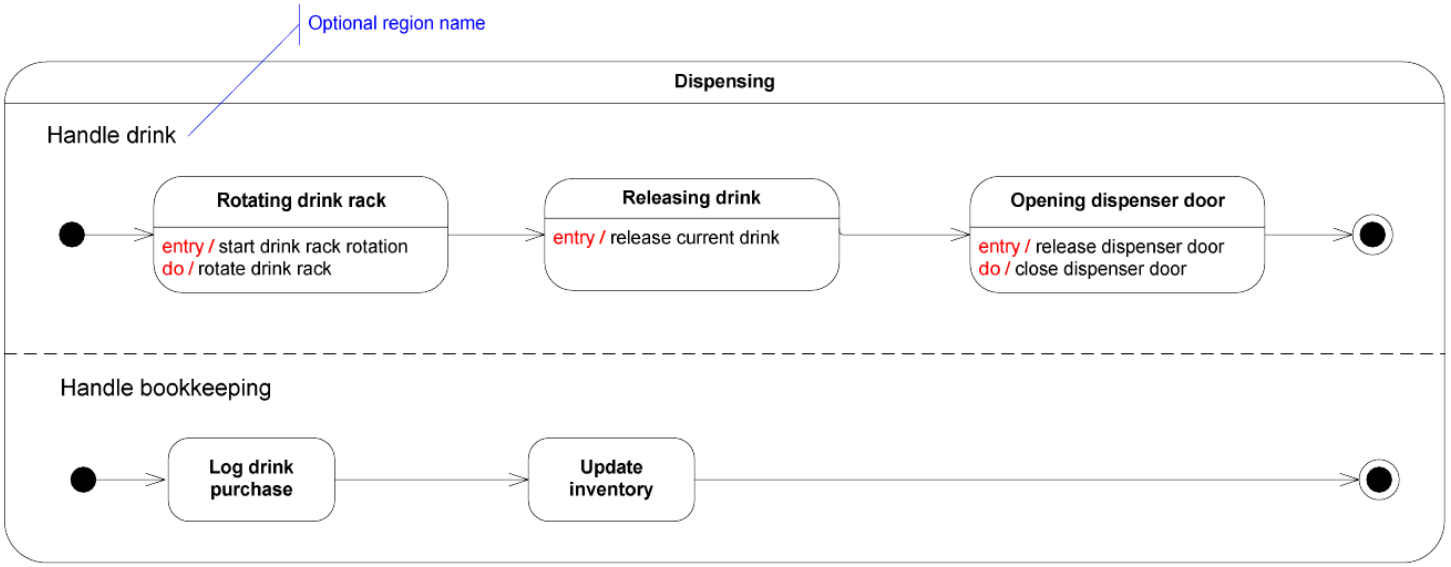
By using parallel states, the complexity in number of state is the sum of the number of states in each parallel states while otherwise it is the product of them
(if we add another parallel state with 3 states, the “sequential” result have 12 states)

Parallel states

Other representations and naming: UML

Composite states / regions

- Each region within a composite state executes in parallel.
- A transition to the final state of a region indicates completing the activity for that region.
- Once all the regions have completed, the composite state triggers a completion event and a completion transition (if one exists) triggers.



Composite state with two regions

From Charles André's slides

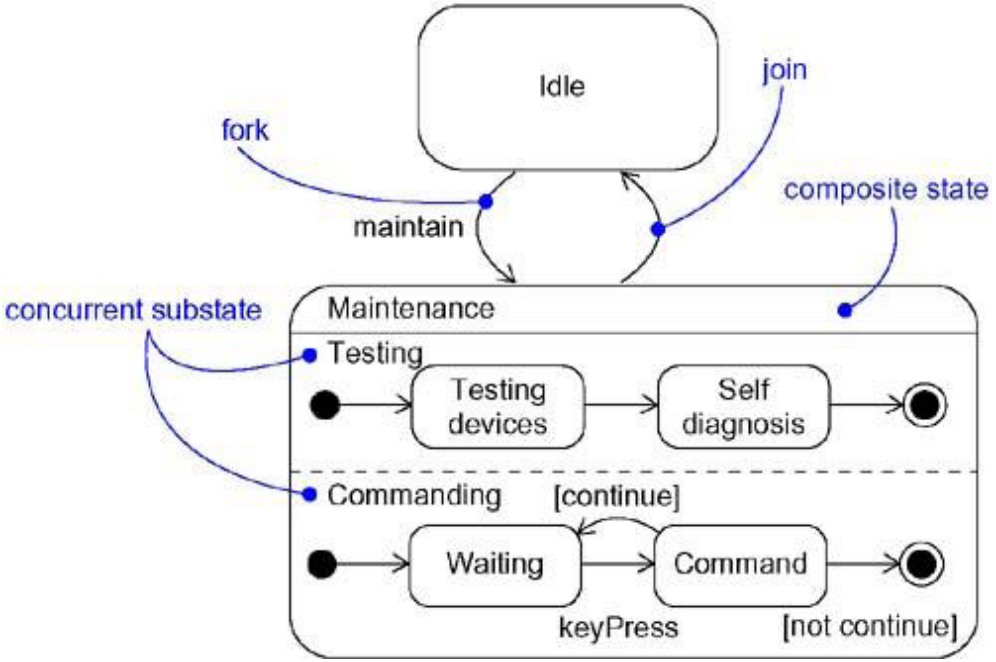
Parallel states

Yet another representation and naming

concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object

Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow

A nested concurrent state machine does not have an initial, final, or history state



Mix with the vocabulary from activity diagrams

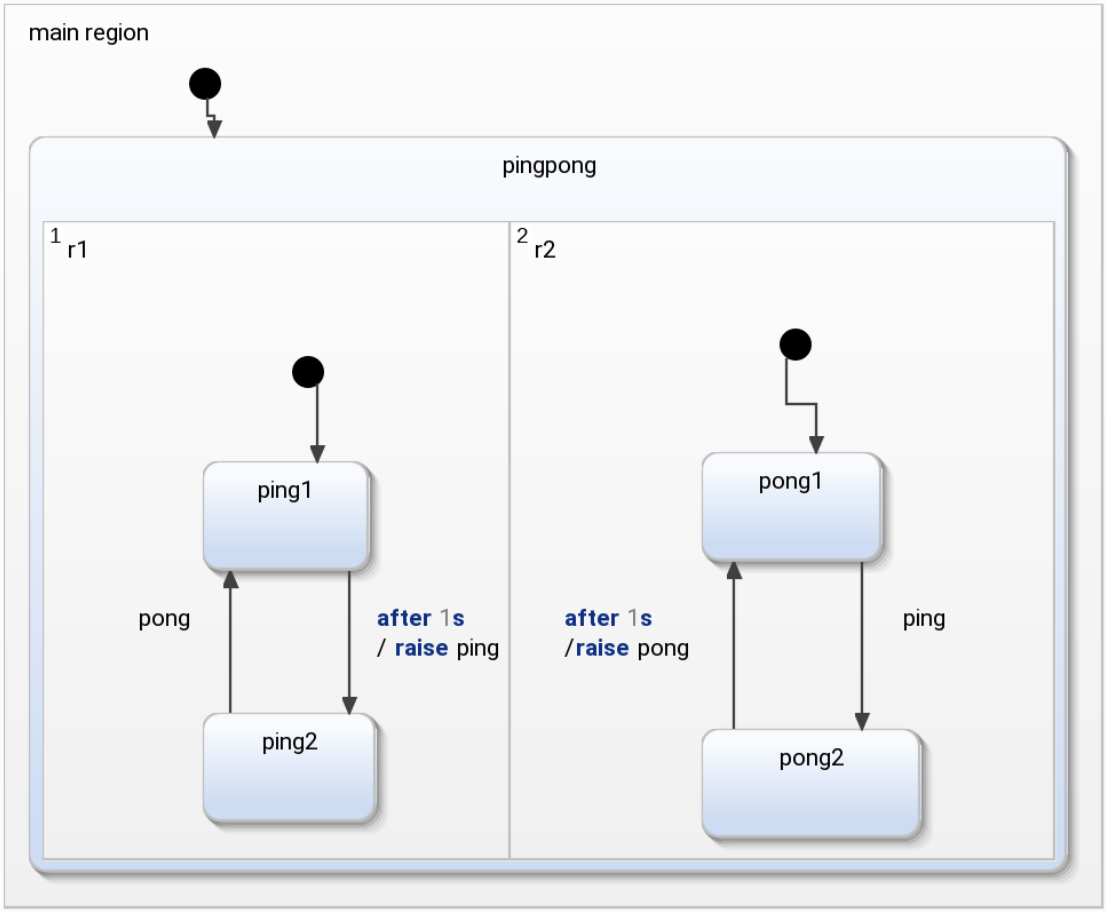
Parallel states

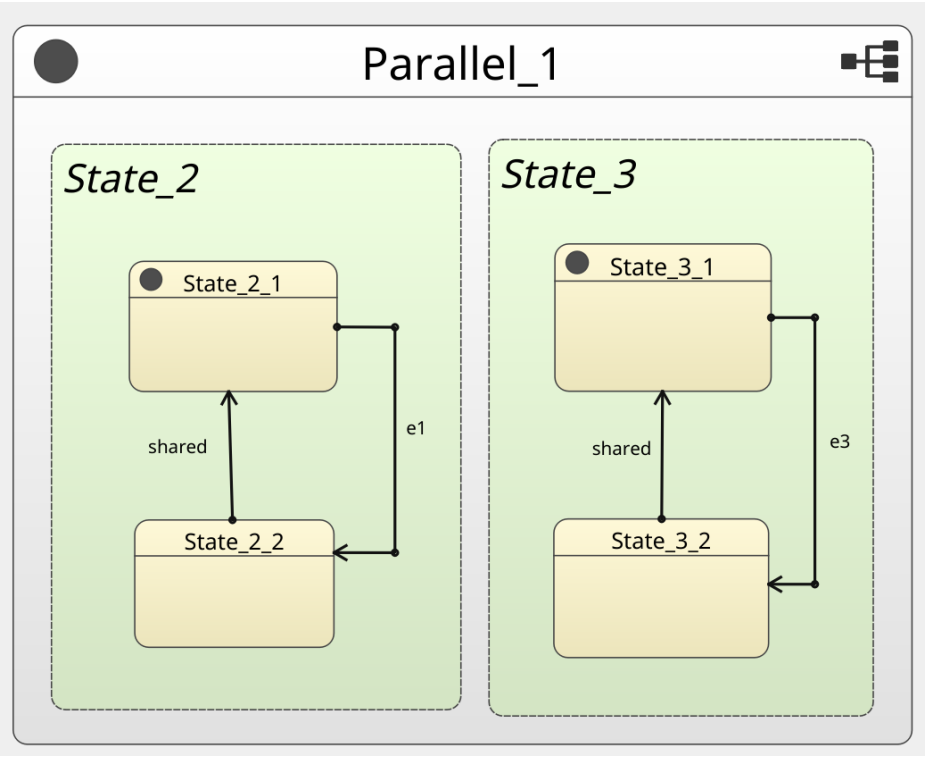
Yet another representation and naming

concurrent substates specify two or more state machines that execute in parallel in the context of the enclosing object

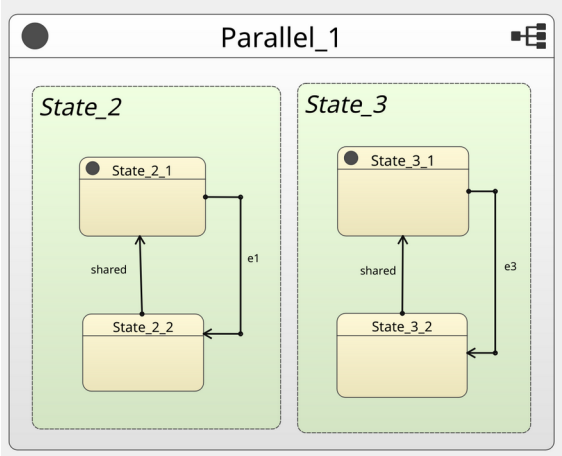
Execution of these concurrent substates continues in parallel. These substates waits for each other to finish to joins back into one flow

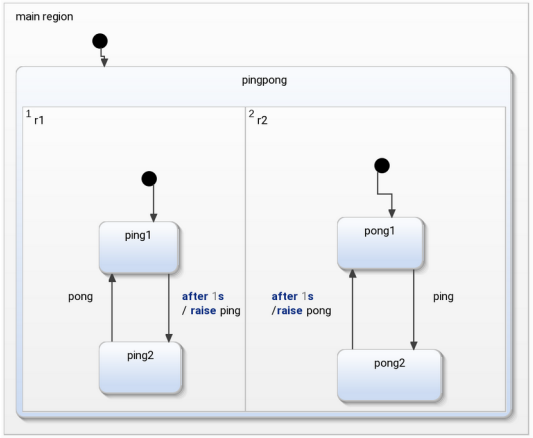
A nested concurrent state machine does not have an initial, final, or history state





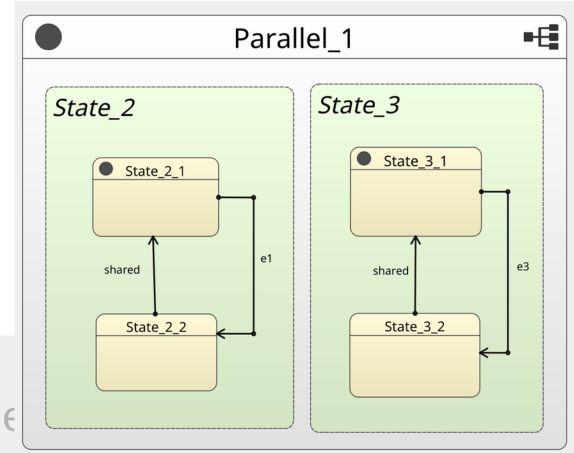
Parallel states

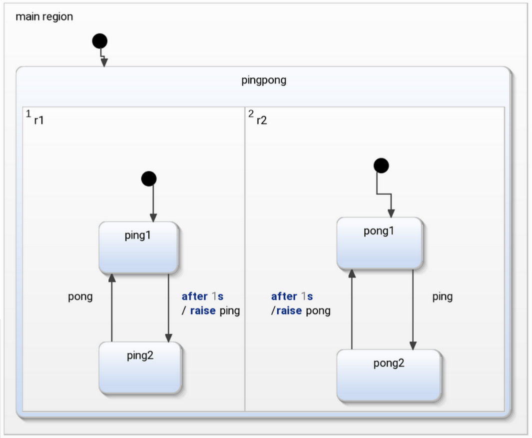




- Parallel States:
  - The child states execute in parallel in the sense that any event that is processed is processed in each child state independently, and each child state may take a different transition in response to the event. (Similarly, one child state may take a transition in response to an event, while another child ignores it.) When all of the children reach final states, the <parallel> element itself is considered to be in a final state, and a completion event done.state.id is generated, where id is the id of the <parallel> element.
  - Transitions within the individual child elements operate normally. However whenever a transition is taken with a target outside the <parallel> element, the <parallel> element and all of its child elements are exited and the corresponding <onexit> handlers are executed. The handlers for the child elements execute first, in document order, followed by those of the parent <parallel> element, followed by an action expression in the <transition> element, and then the <onentry> handlers in the "target" state.
  - Note that the semantics of the <parallel> element does not call for multiple threads or truly concurrent processing. The children of <parallel> execute in parallel in the sense that they are all simultaneously active and each one independently selects transitions for any event that is received. However, the parallel children process the event in a defined, serial order, so no conflicts or race conditions can occur. See D Algorithm for SCXML Interpretation for a detailed description of the semantics of <parallel> and the rest of SCXML.

Parallel states





- Parallel States:
  - The child states execute in paralle … that i … each child state independently, and each child state may take a different transition in response to the event. (Similarly, one child state may take a transition in response to an event, while another …ld ignores it.) When all of the children reach final states, the <parallel> element itself is …sidered to be in a final state, and a completion event done.state.id is generated, where id is

UML:

Even though orthogonal regions imply independence of execution (allowing more or less concurrency), the UML specification does not require that a separate thread of execution be assigned to each orthogonal region (although this can be done if desired). In fact, most commonly, orthogonal regions execute within the same thread*. The UML specification requires only that the designer does not rely on any particular order for event instances to be dispatched to the relevant orthogonal regions.

  - Note that the semantics of the <parallel> element does not call for multiple threads or truly concurrent processing. The children of <parallel> execute in parallel in the sense that they are all simultaneously active and each one independently selects transitions for any event that is received. However, the parallel children process the event in a defined, serial order, so no conflicts or race conditions can occur. See D Algorithm for SCXML Interpretation for a detailed description of the semantics of <parallel> and the rest of SCXML.

* Douglass, Bruce Powel (1999). Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison Wesley. p. 749. ISBN 0-201-49837-5.

SCXML
State Chart XML

Parallel regions, communications
and a little bit more

State Charts

statecharts = state-diagrams + depth

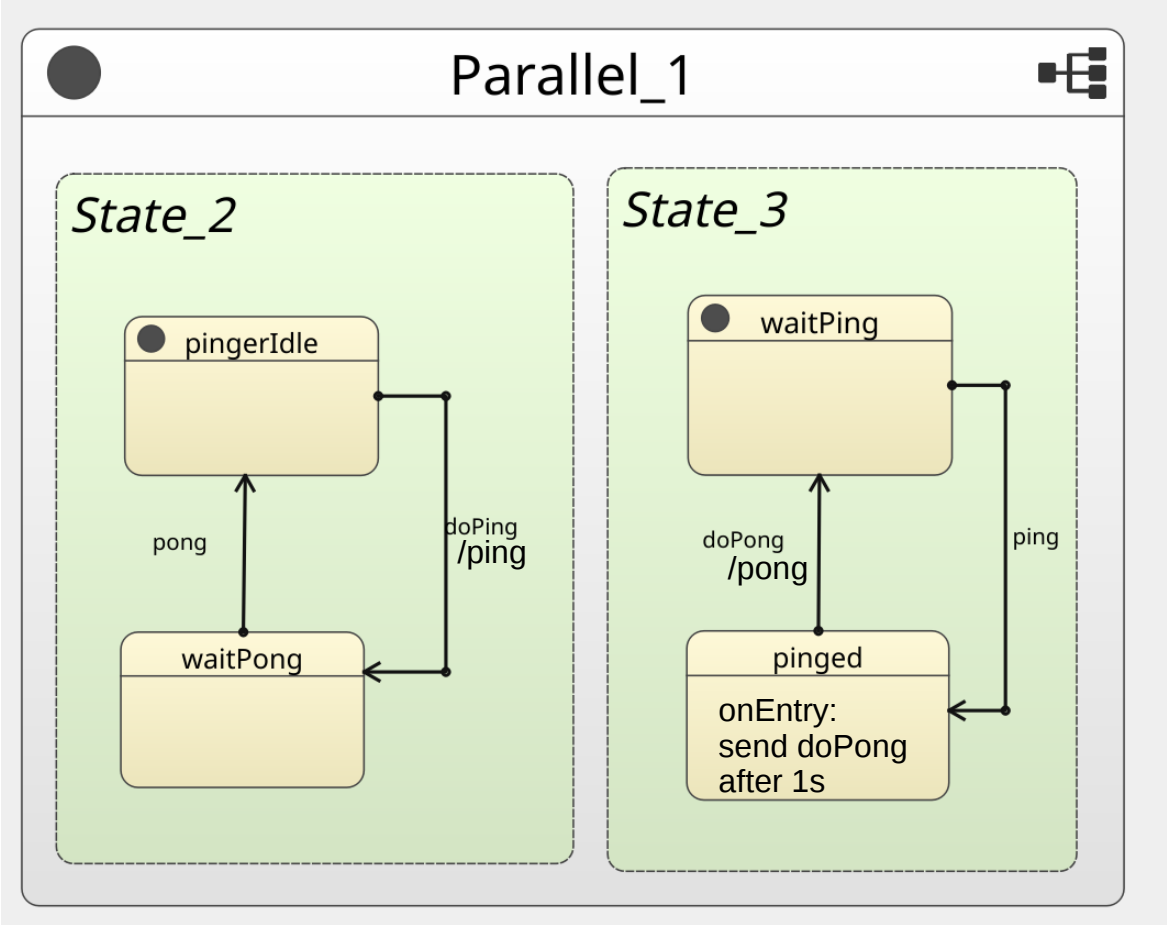
+ orthogonality + broadcast-communication.

David Harel
Statecharts: A visual formalism for complex systems
Science of computer programming 8 (3), 231-274
1987

Communication: intuition

Usually, parallel states are not truly independent. The different state machines can communicate through different mechanism (*e.g.*, shared variable or timings) to synchronize their behaviors.

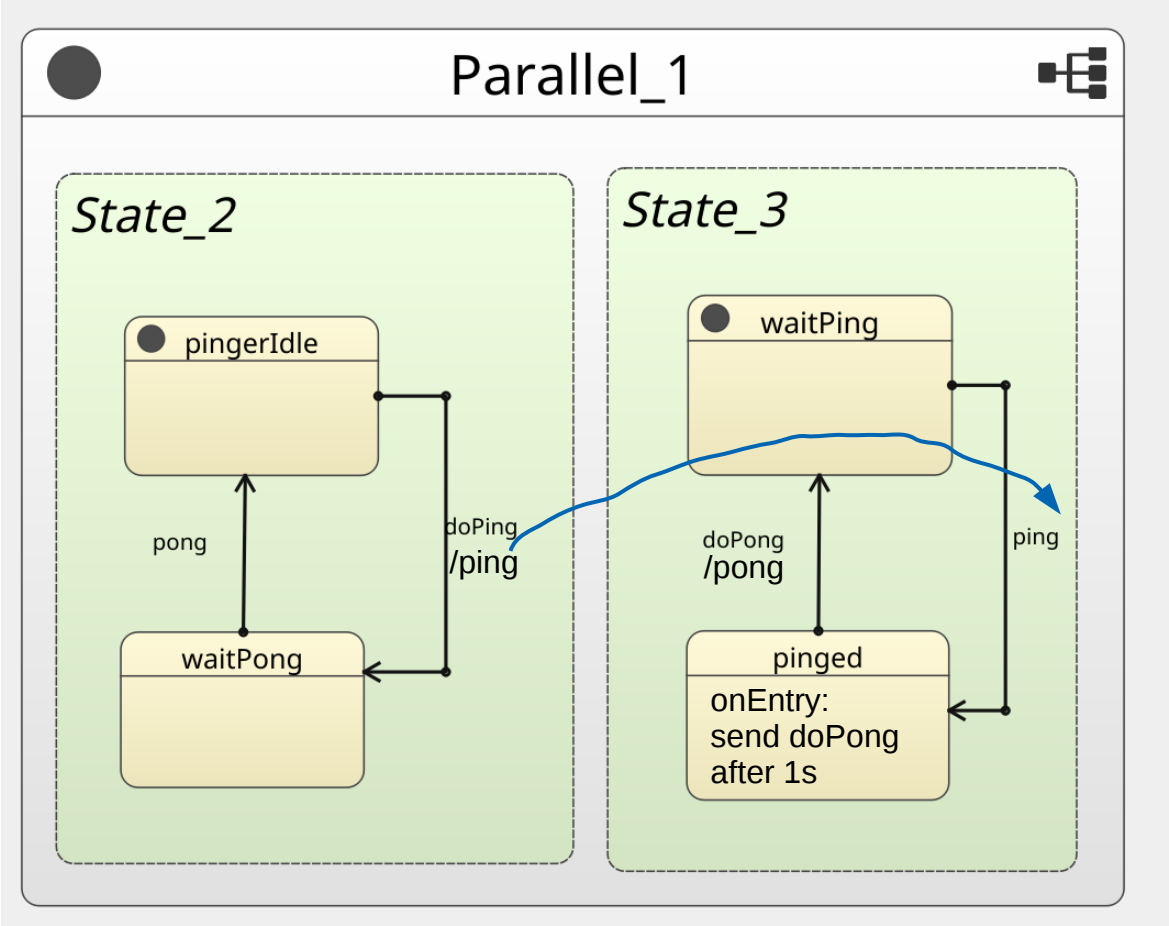
→ The most common way to coordinate behaviors from parallel states is by sending events to each others.



Communication: intuition

Usually, parallel states are not truly independent. The different state machines can communicate through different mechanism (*e.g.*, shared variable or timings) to synchronize their behaviors.

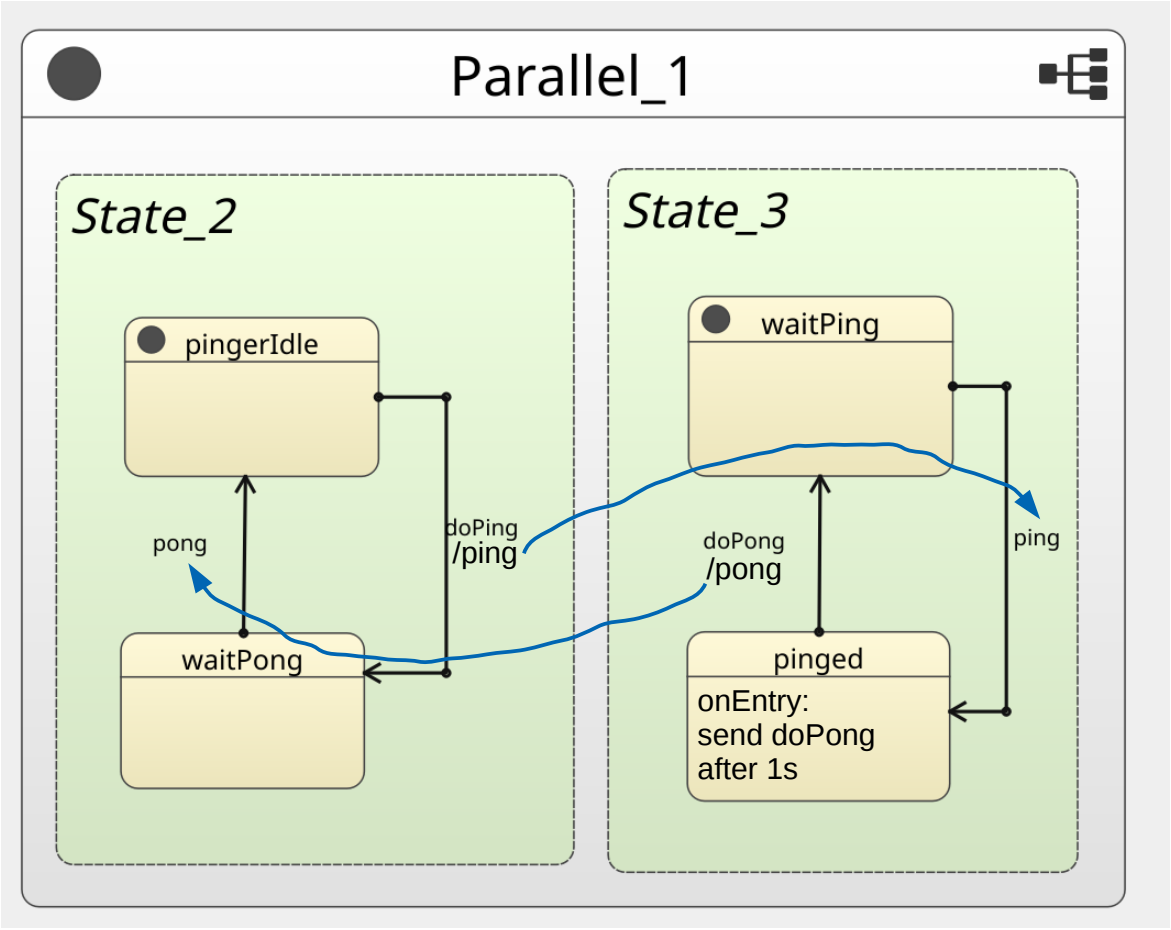
→ The most common way to coordinate behaviors from parallel states is by sending events to each others.



Communication: intuition

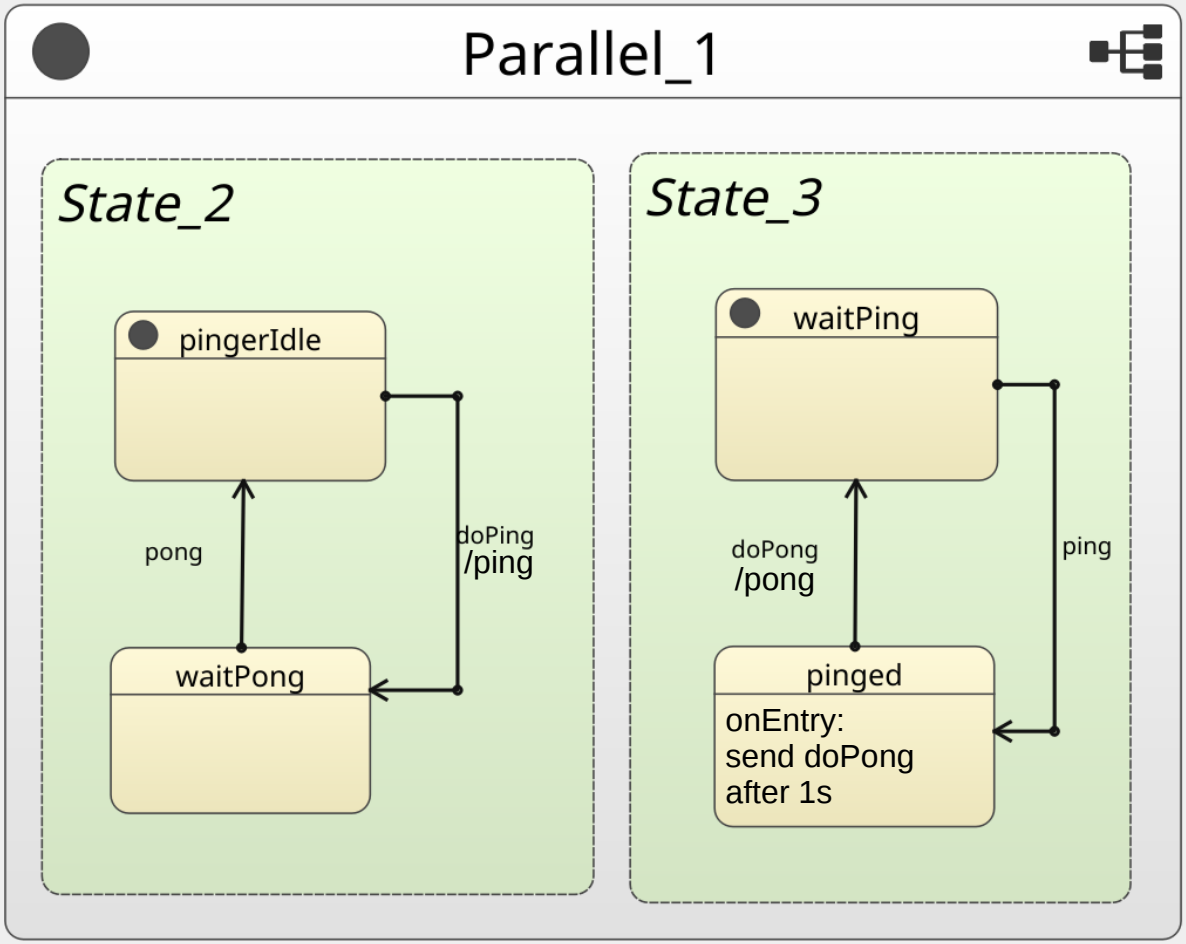
Usually, parallel states are not truly independent. The different state machines can communicate through different mechanism (*e.g.*, shared variable or timings) to synchronize their behaviors.

→ The most common way to coordinate behaviors from parallel states is by sending events to each others.



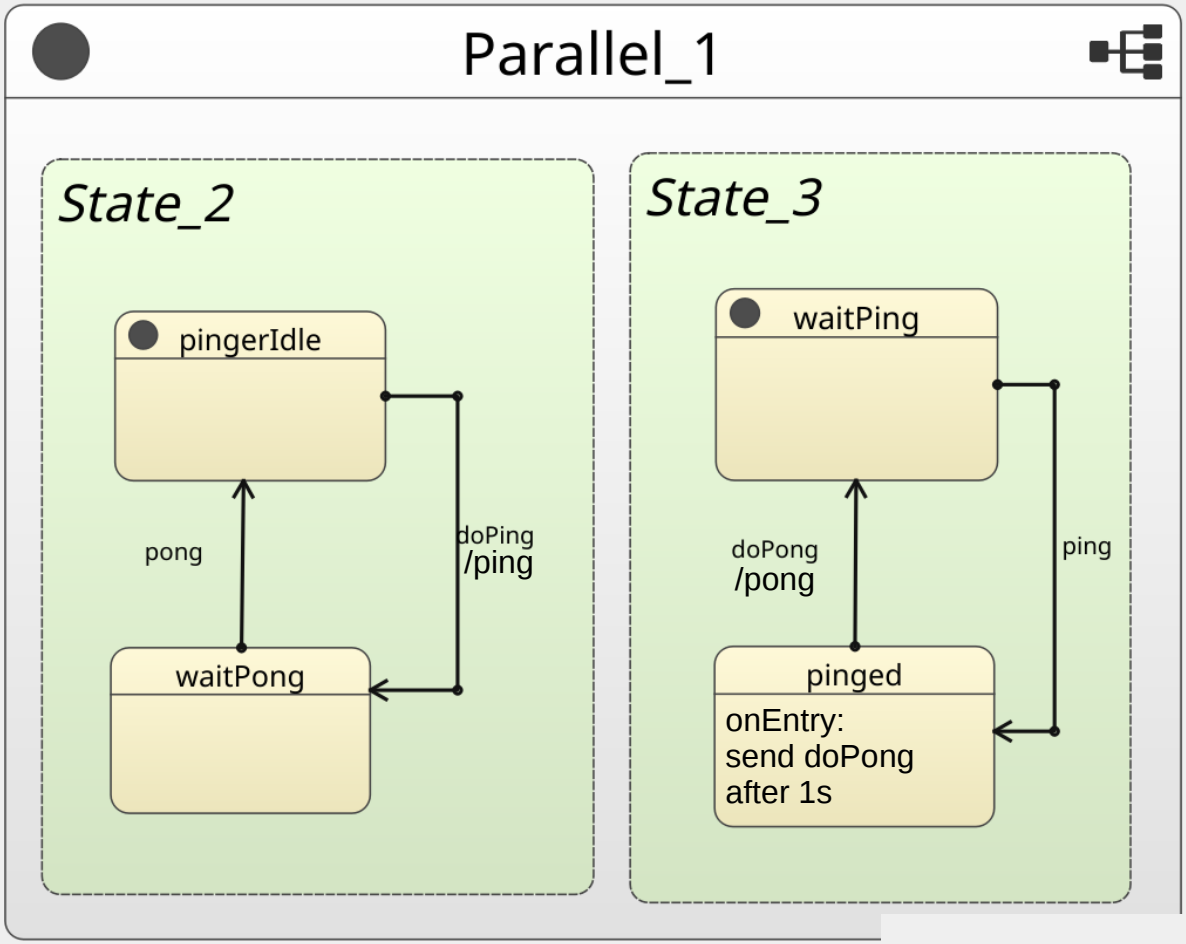
Communication: intuition

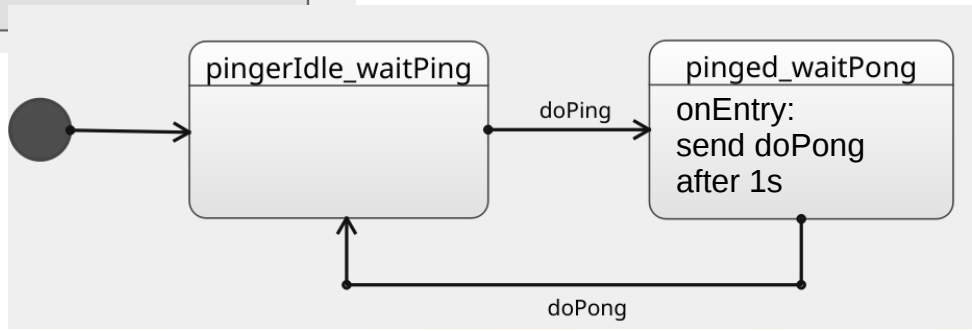
The coordination between the parallel states *can* reduce the resulting behavior



Communication: intuition

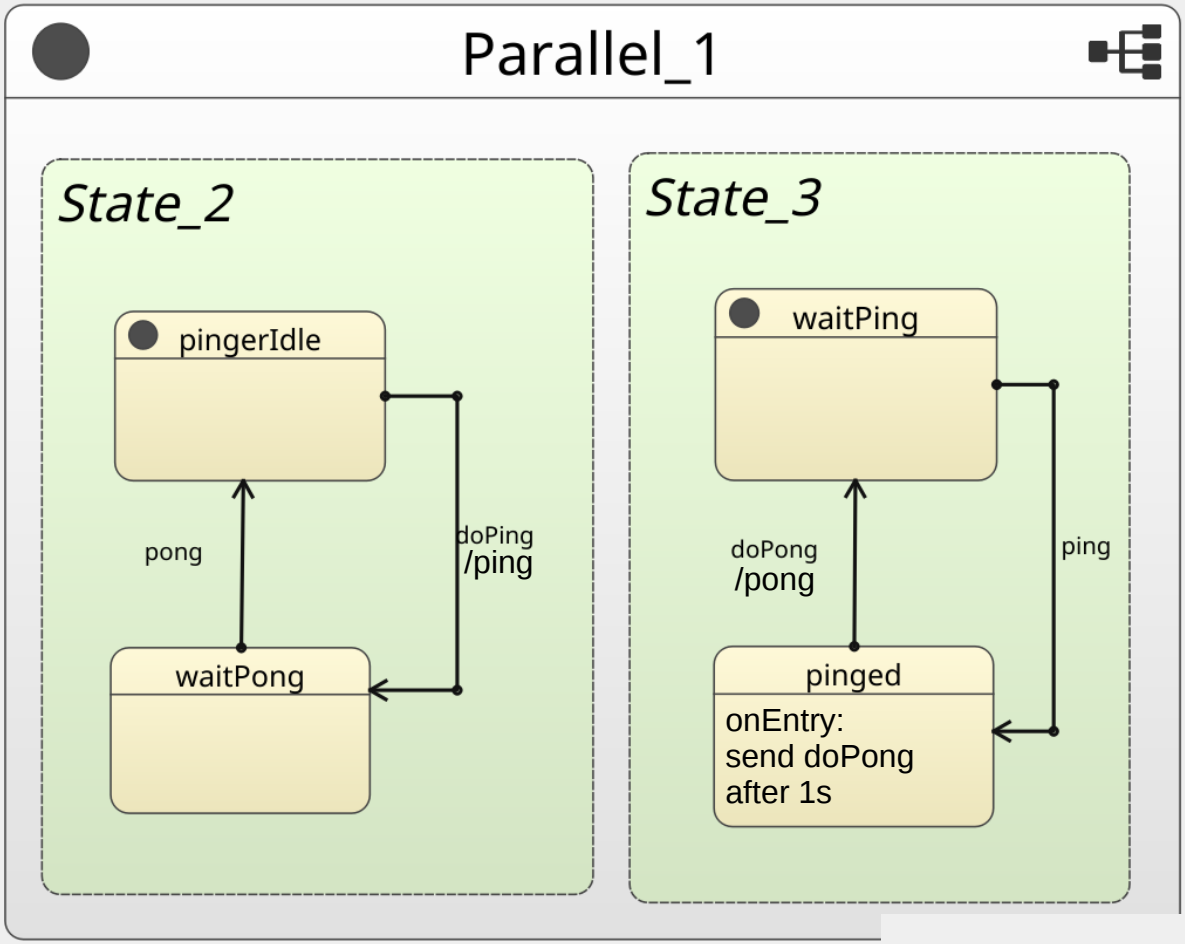
The coordination between the parallel states *can* reduce the resulting behavior



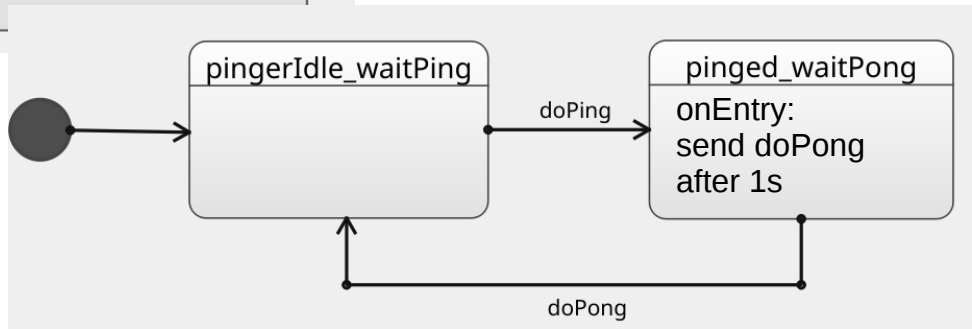


Communication: intuition

The coordination between the parallel states *can* reduce the resulting behavior



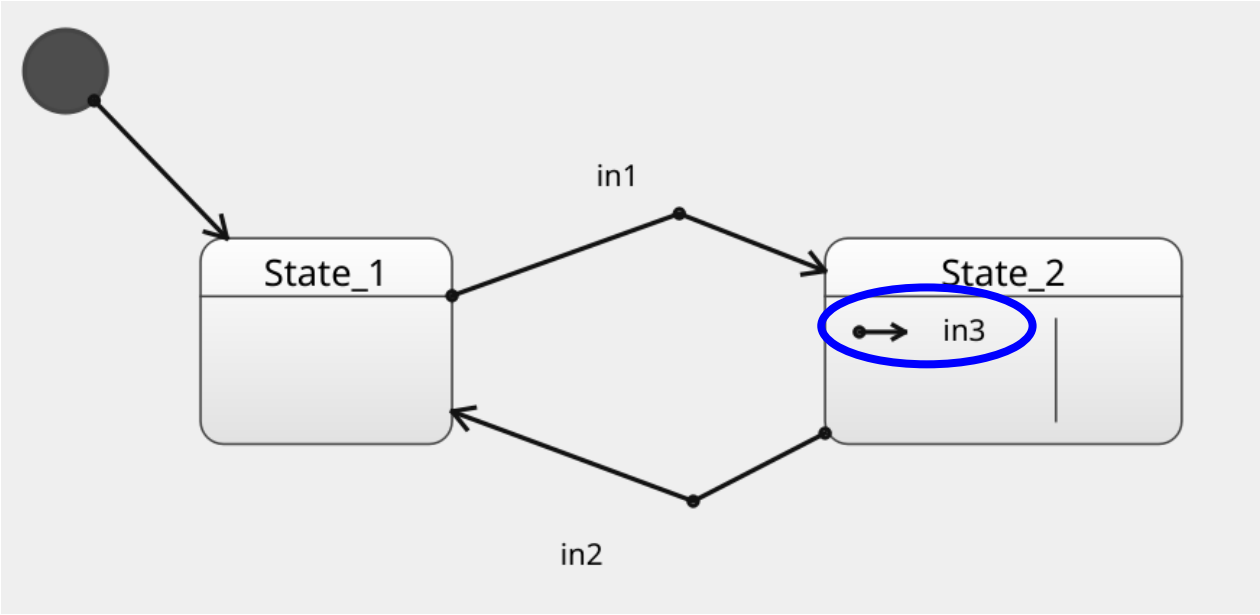
If *ping* or *pong* is not sent from the outside !



SCXML
State Chart XML

Parallel regions, communications
and a little bit more

No target transition

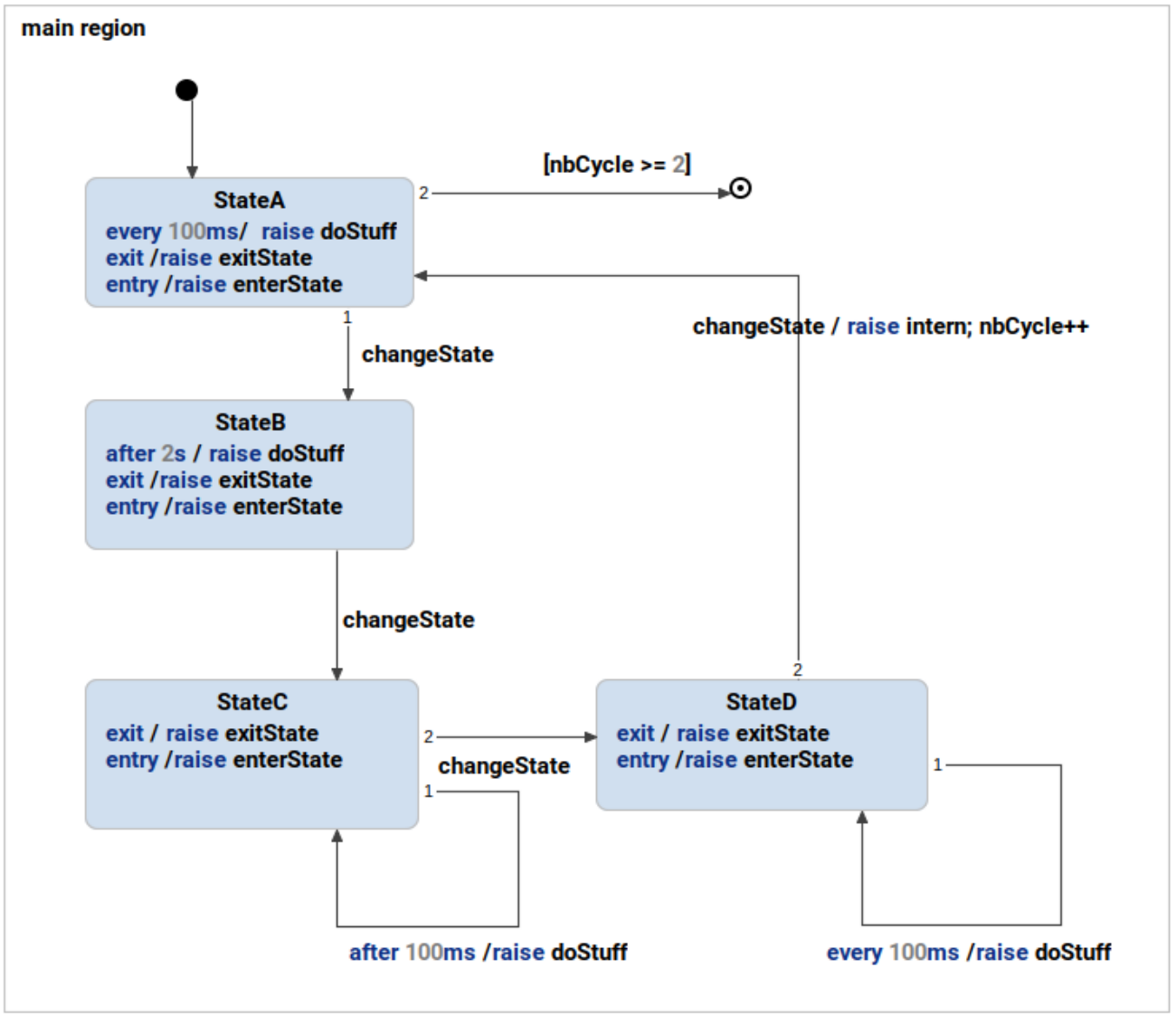


If the 'target' on a <transition> is omitted, then the value of 'type' does not have any effect and taking the transition does not change the state configuration but does invoke the executable content that is included in the transition. Note that this is different from a <transition> whose 'target' is its source state. In the latter case, the state is exited and reentered, triggering execution of its <onentry> and <onexit> executable content

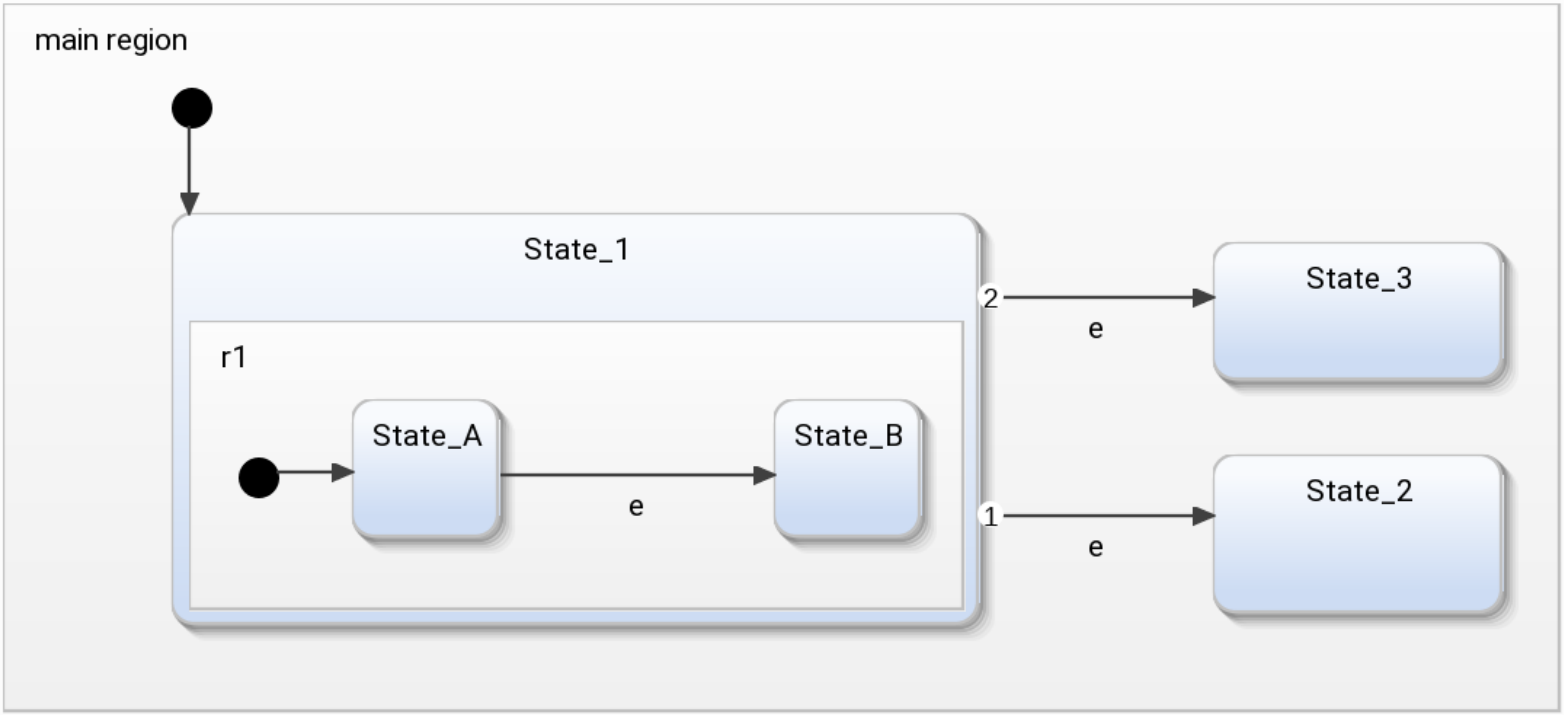
No target transition

everyTest

```
@EventDriven
// Use the event driven execution model.
// Runs a run-to-completion step
// each time an event is raised.
// Switch to cycle based behavior
// by specifying '@CycleBased(200)'
// instead.

@ChildFirstExecution
// In composite states, execute
// child states first.
// @ParentFirstExecution does the opposite.

interface:
out event doStuff
out event exitState
out event enterState
in event changeState

internal:

event intern
var nbCycle: integer = 0
```

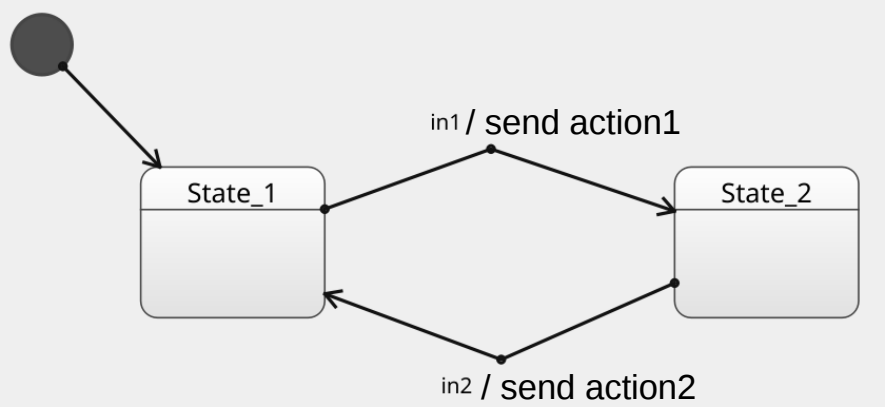


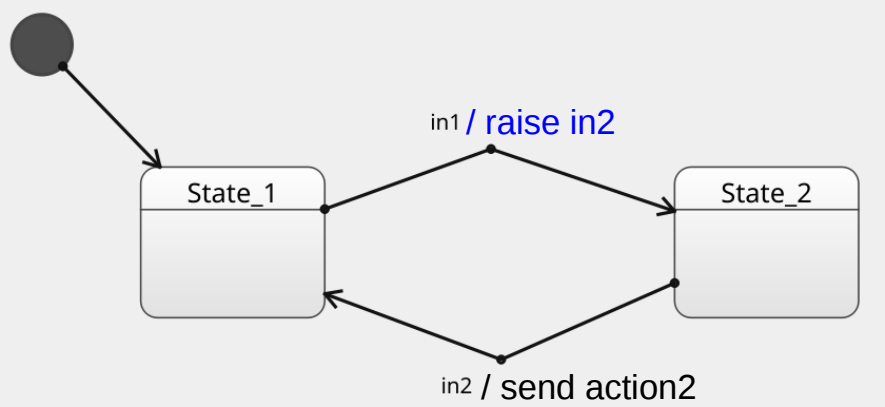
Non Determinism ?



N.B. If two transitions conflict, then taking them both may lead to an illegal configuration. Hence, only one of the transitions may safely be taken. In order to resolve conflicts between transitions, we assign priorities to transitions as follows: let transitions T1 and T2 conflict, where T1 is optimally enabled in atomic state S1, and T2 is optimally enabled in atomic state S2, where S1 and S2 are both active. We say that T1 has a higher priority than T2 if a) T1's source state is a descendant of T2's source state, or b) S1 precedes S2 in document order.

Raise vs send





The <raise> element raises an event in the current SCXML session. Note that the event will not be processed until the current block of executable content has completed and all events that are already in the internal event queue have been processed. For example, suppose the <raise> element occurs first in the <onentry> handler of state S followed by executable content elements ec1 and ec2. If event e1 is already in the internal event queue when S is entered, the event generated by <raise> will not be processed until ec1 and ec2 have finished execution and e1 has been processed.

<send> is used to send events and data to external systems, including external SCXML Interpreters, or to raise events in the current SCXML session.

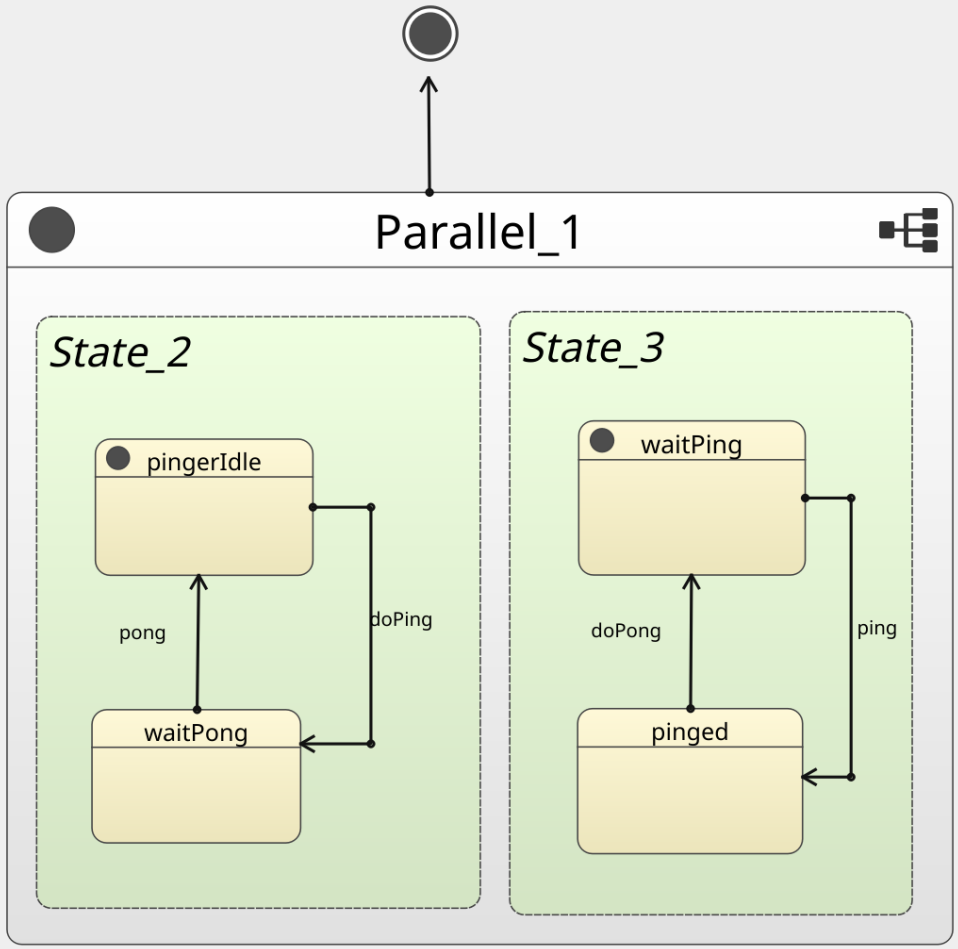
Datamodel

- It is possible to add data and data manipulation directly inside the state machine.
- SCXML supports the ecmascript data model (some sort of C++ in Qt Creator and a specific action langage in Yakindu)
- *Usually** , only data directly associated with the control is used in the state machine
  - To avoid mixing the application logic between the code and the state chart
  - To ease the V&V activities

** it strongly depends on what the state chart is used for but it seems to be a good practice, even for understanding*

Datamodel

When ecmascript data model is used, the log expr must not be null and it must conform the ecmascript syntax
→ otherwise no log is printed



Structure

- scxml
  - datamodel
    - data
    - data
  - Parallel_1
    - onentry
    - State_2
      - pingerIdle
        - doPing
          - assign
          - send
          - log
        - waitPong
          - pong
    - State_3
      - waitPing
        - ping
      - pinged
        - doPong
          - send
          - log
        - onentry
          - send
    - transition

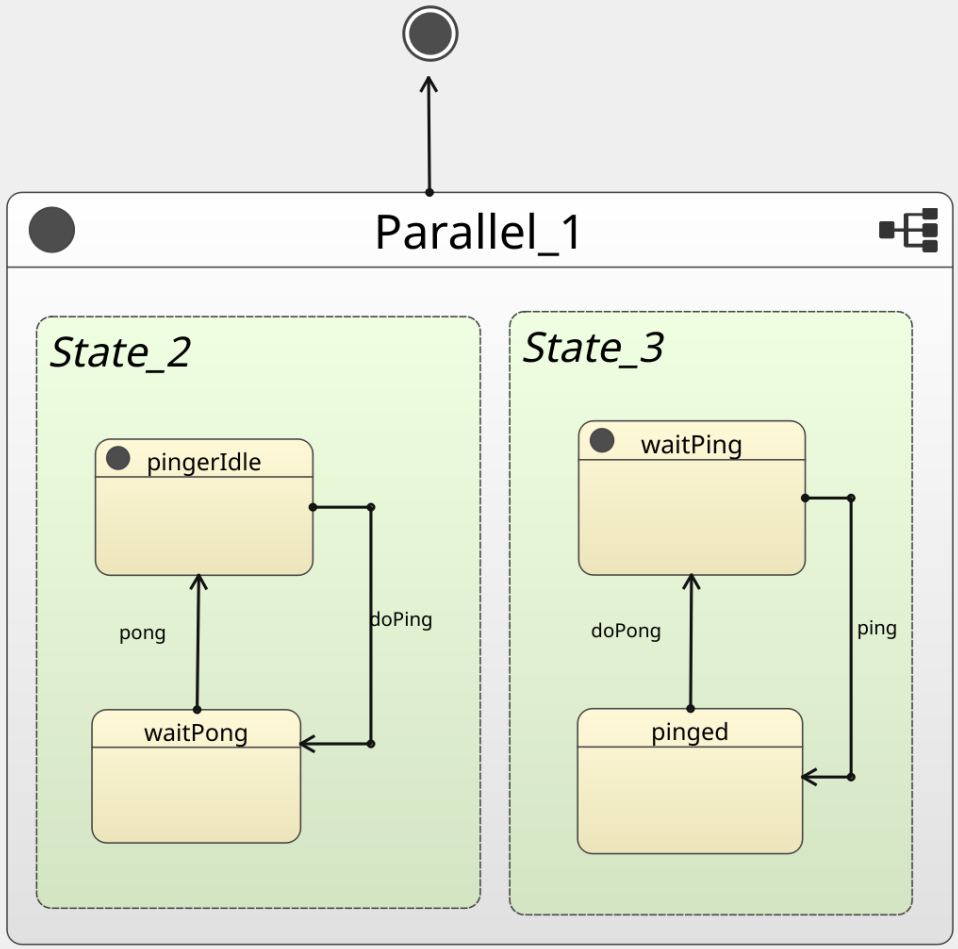
Attributes — scxml

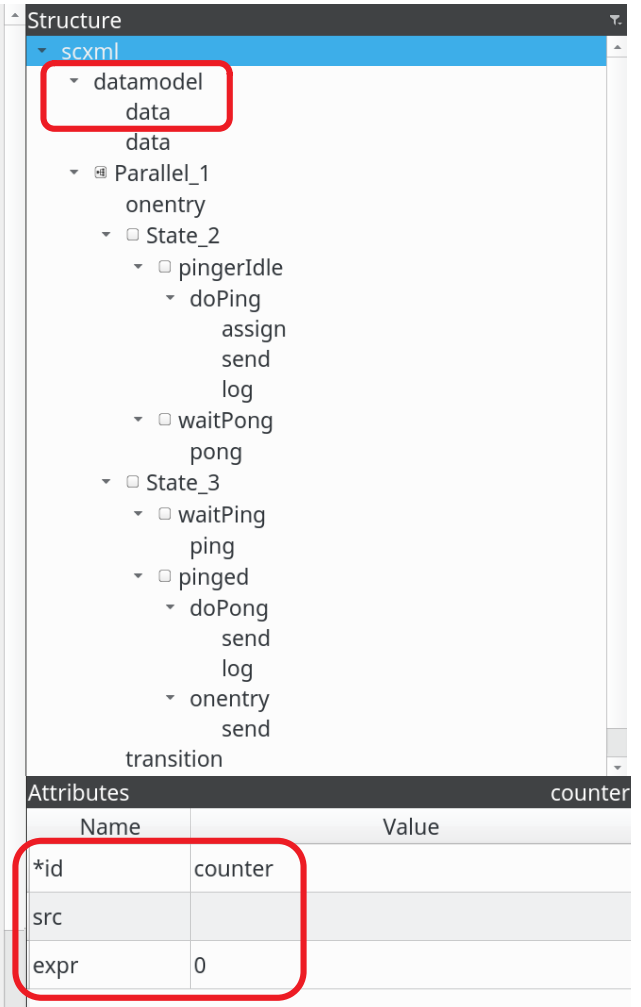
Name	Value
initial	
name	TestPingPong
*xmlns	http://www.w3.org/2005/07/scxml
*version	1.0
datamodel	ecmascript
binding	early

Datamodel

Variables are declared and initialized in a specific section

id: The name of the data item. See 3.14 IDs for details.
src: Gives the location from which the data object should be fetched. See 5.9.3 Legal Data Values and Value Expressions for details.
expr: Evaluates to provide the value of the data item. See 5.9.3 Legal Data Values and Value Expressions for details.

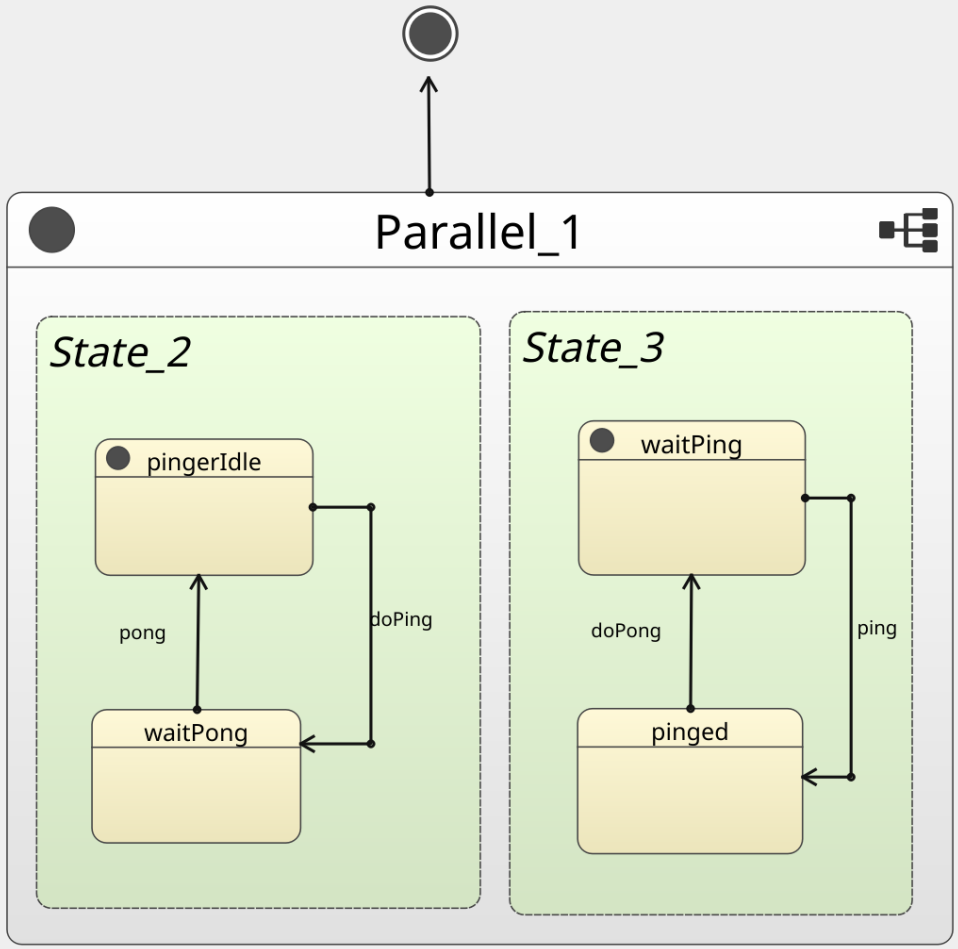


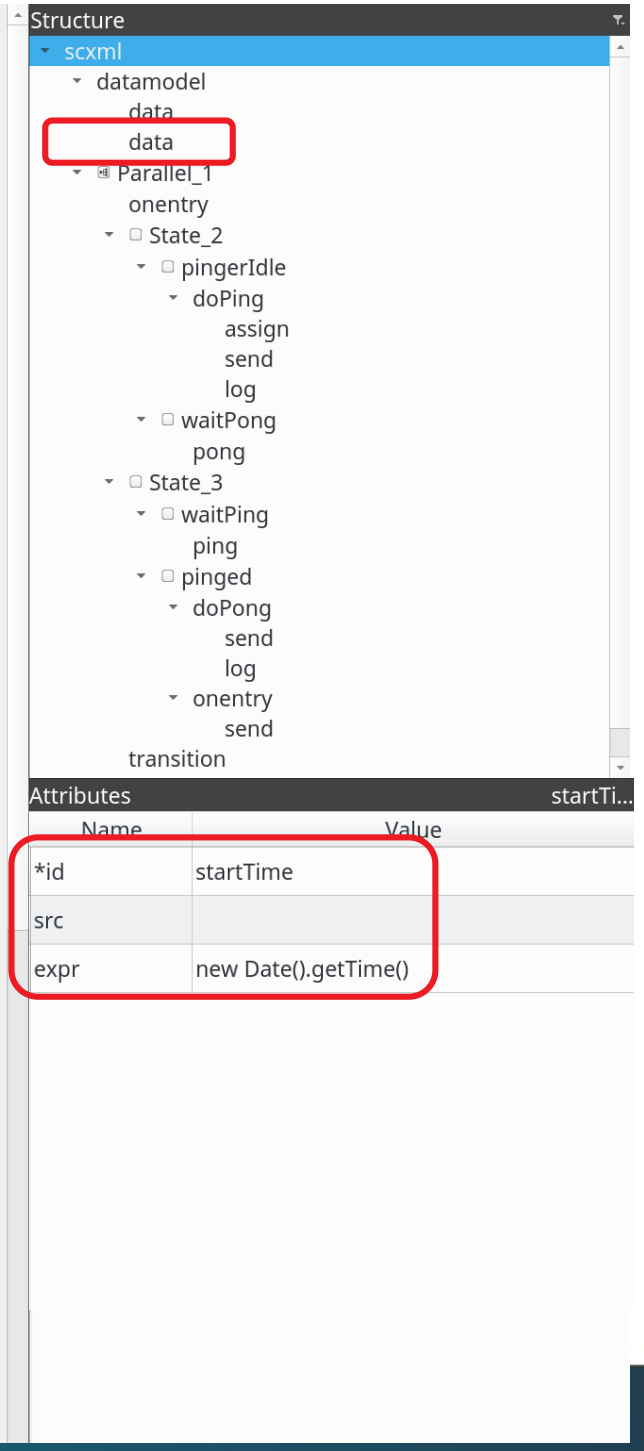


Datamodel

Variables are declared and initialized in a specific section

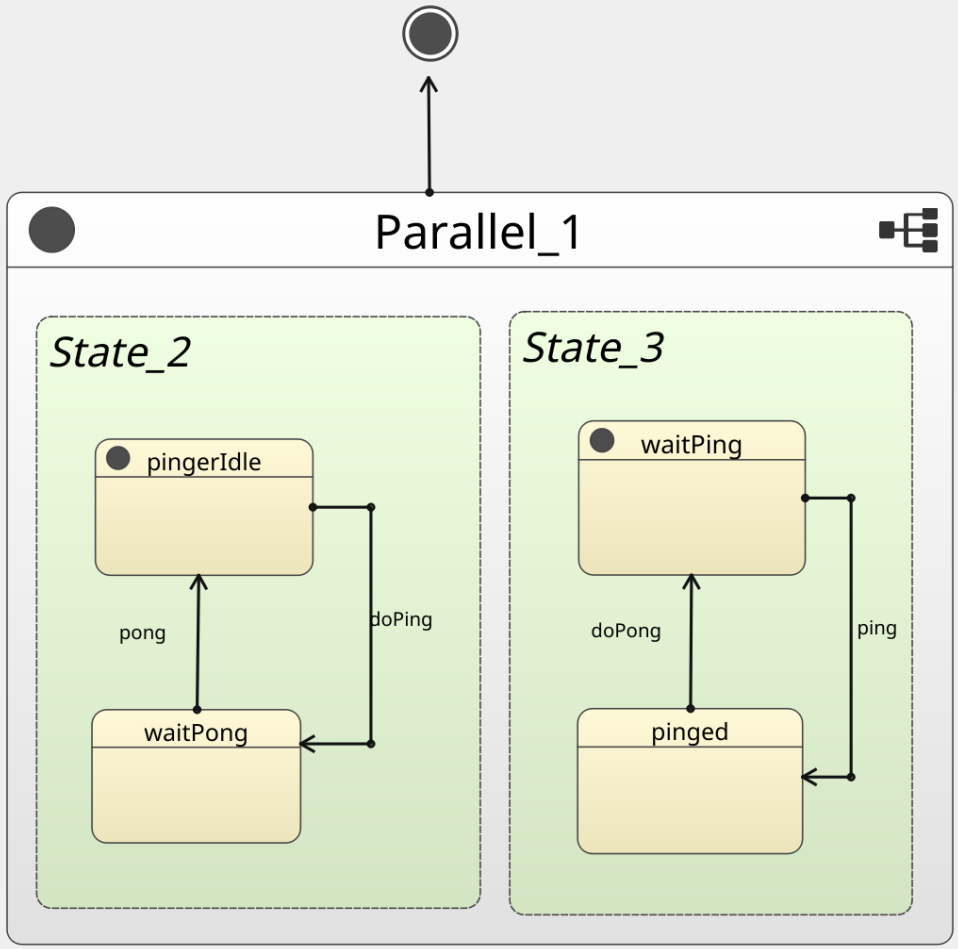
id: The name of the data item. See 3.14 IDs for details.
src: Gives the location from which the data object should be fetched. See 5.9.3 Legal Data Values and Value Expressions for details.
expr: Evaluates to provide the value of the data item. See 5.9.3 Legal Data Values and Value Expressions for details.

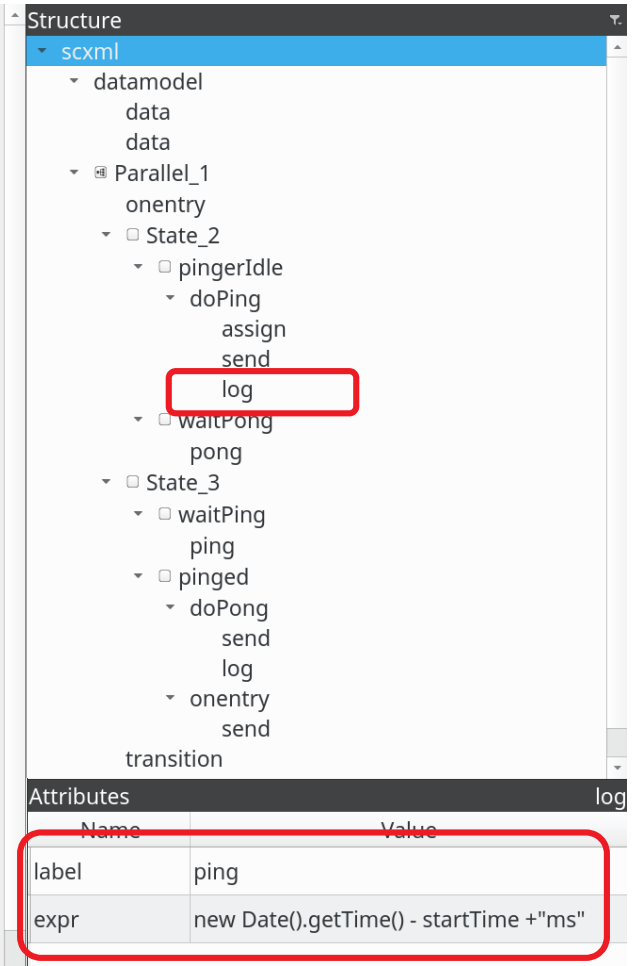




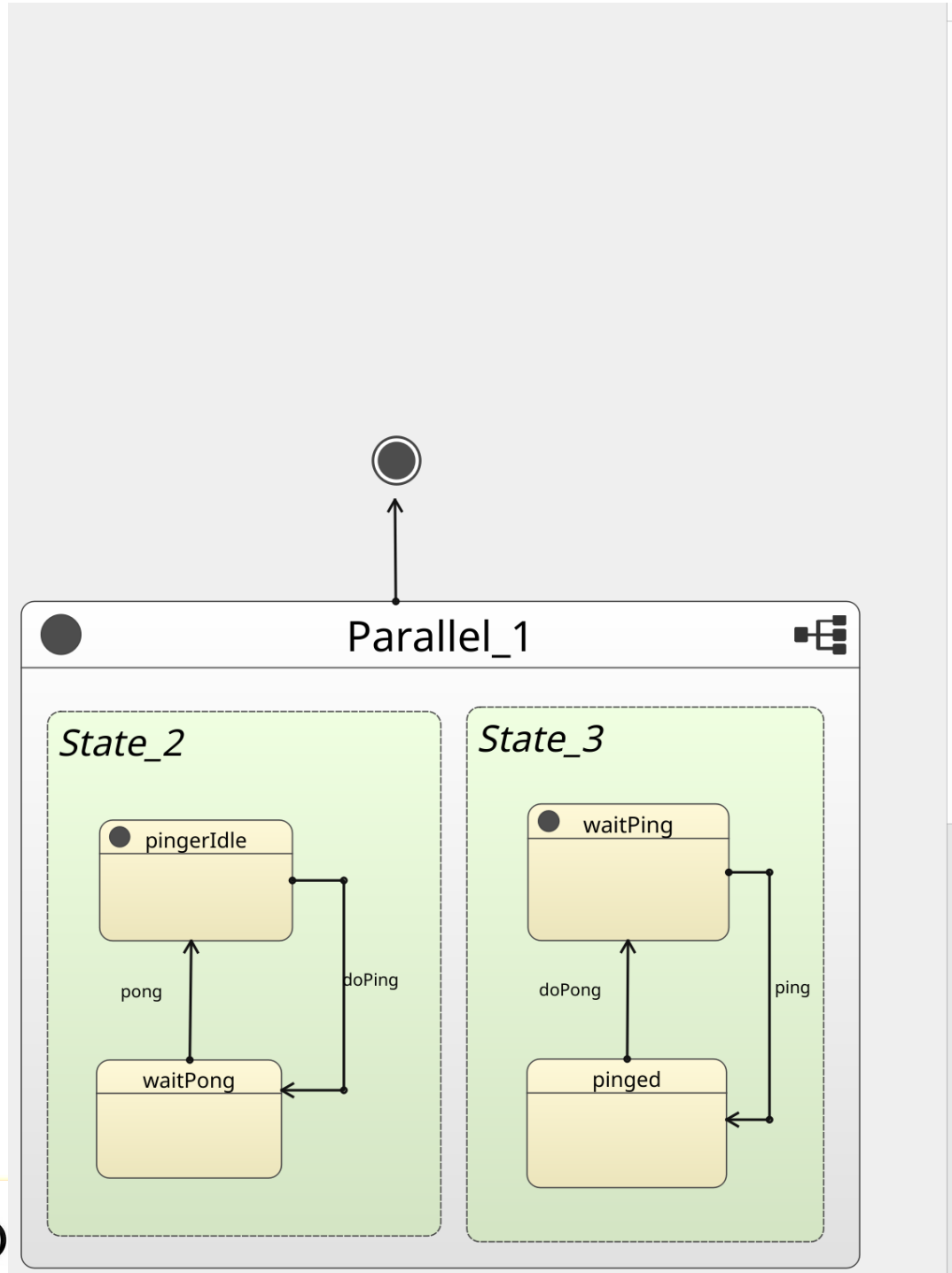
Datamodel

Label is any string and expr any legal ecmascript expression
(if no expression, needs to be '')





Datamodel



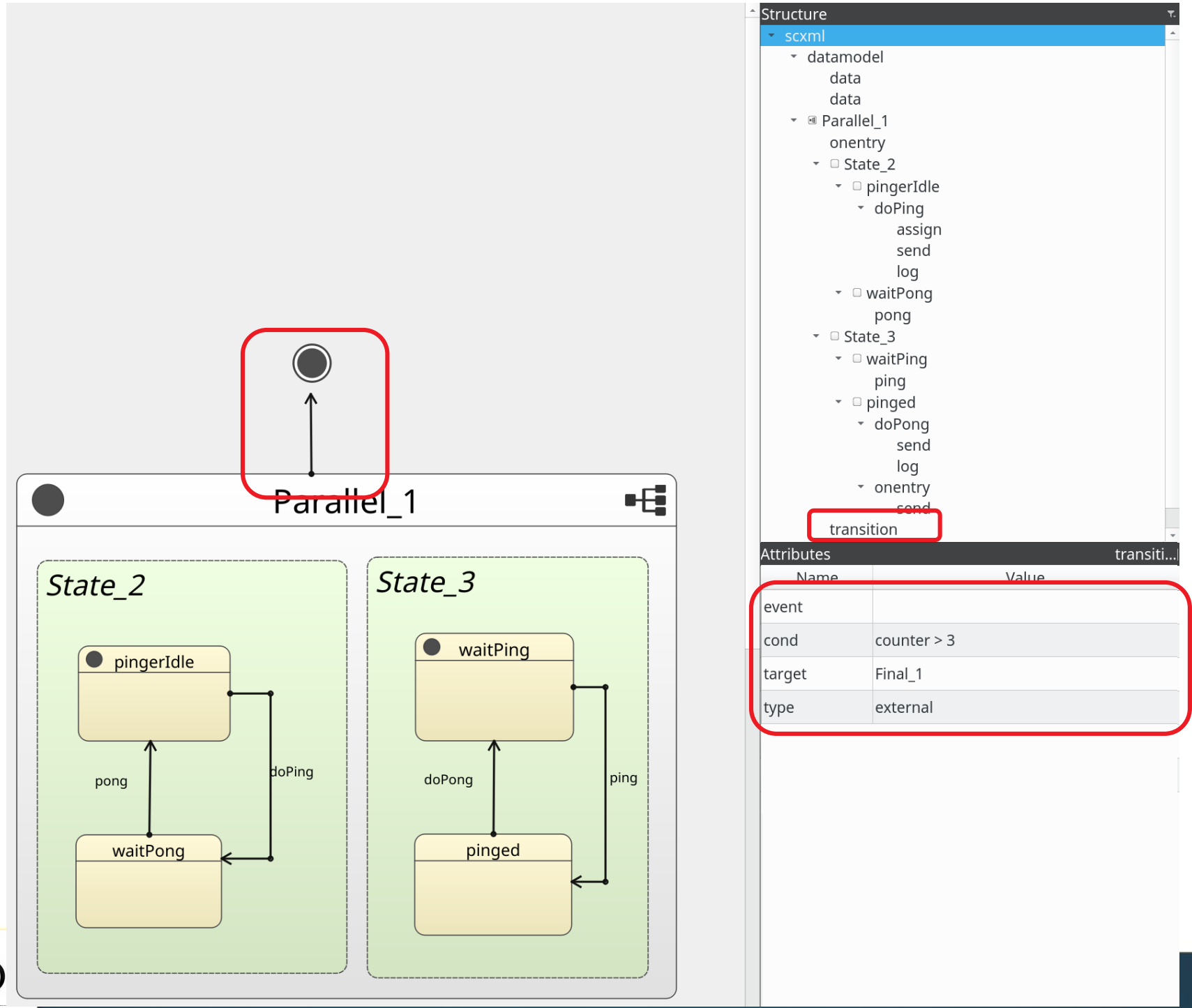
Structure

- scxml
  - datamodel
    - data
    - data
  - Parallel_1
    - onentry
    - State_2
      - pingerIdle
        - doPing
          - assign
          - send
          - log
      - waitPong
        - pong
    - State_3
      - waitPing
        - ping
      - pinged
        - doPong
          - send
          - log
        - onentry
          - send
    - transition

Attributes — transiti...

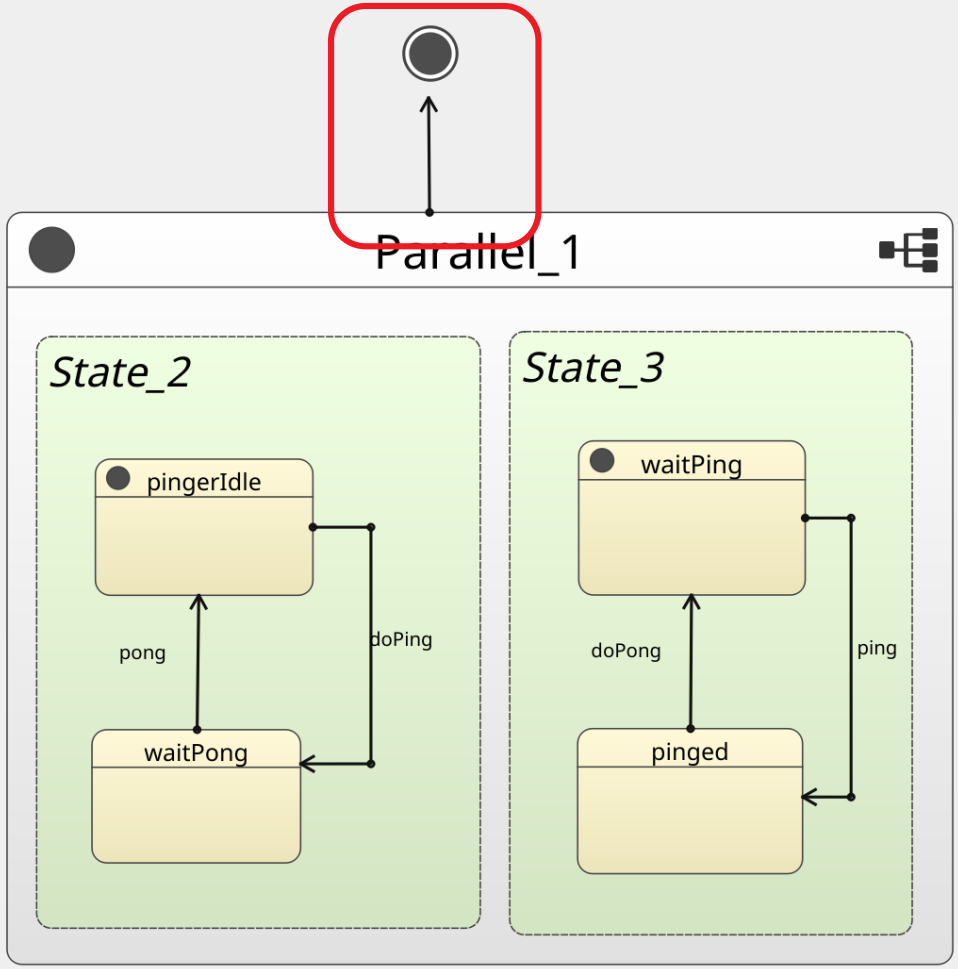
Name	Value
event	
cond	counter > 3
target	Final_1
type	external

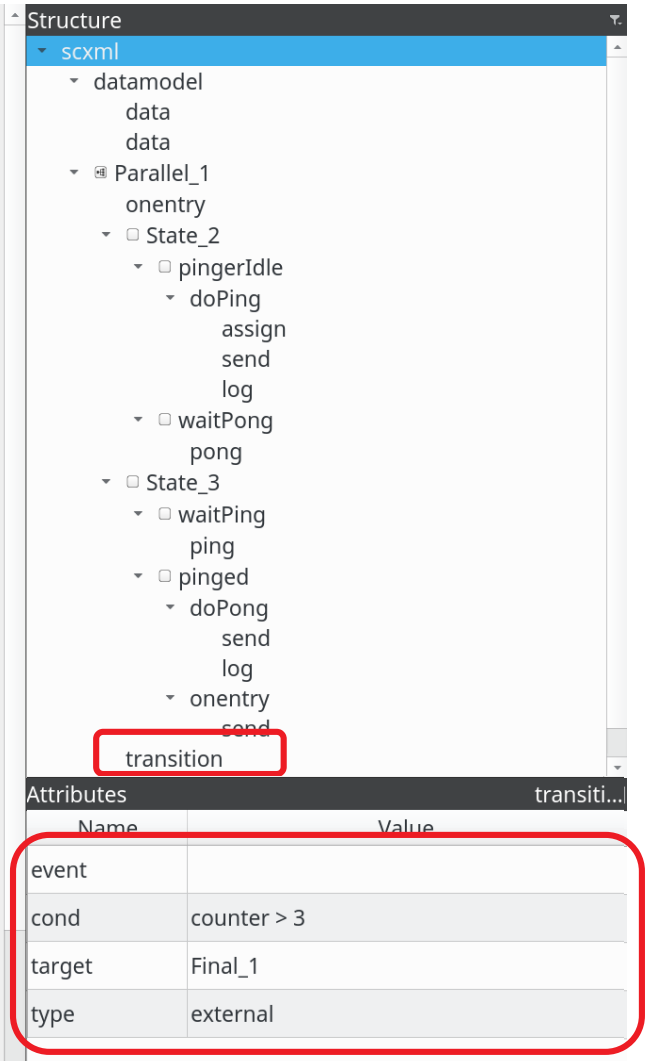
Datamodel


Structure
scxml
datamodel
data
data
Parallel_1
onentry
State_2
pingerIdle
doPing
assign
send
log
waitPong
pong
State_3
waitPing
ping
pinged
doPong
send
log
onentry
send
transition
Attributes
transiti...
Name
Value
event
cond
counter > 3
target
Final_1
type
external
Parallel_1
State_2
pingerIdle
pong
doPing
waitPong
State_3
waitPing
doPong
ping
pinged

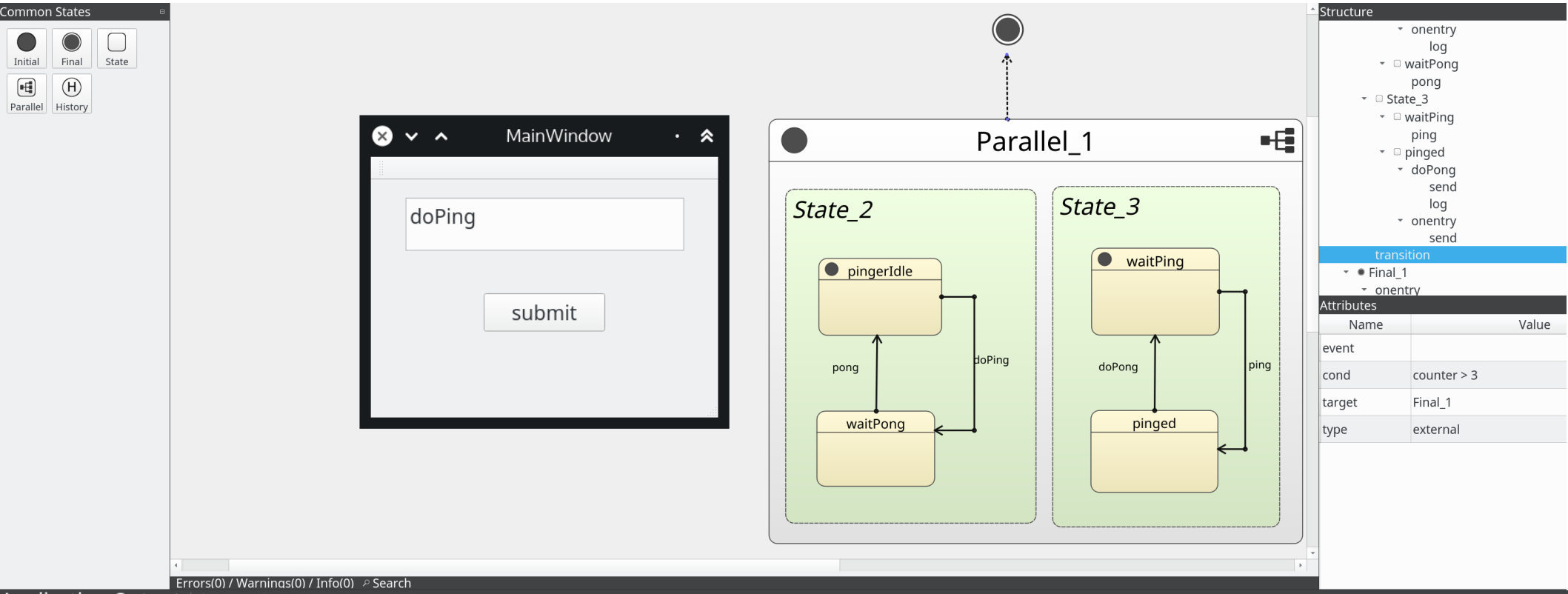
Guarded transition

cond: is evaluated after each micro step and fired when the condition holds





Guarded transition

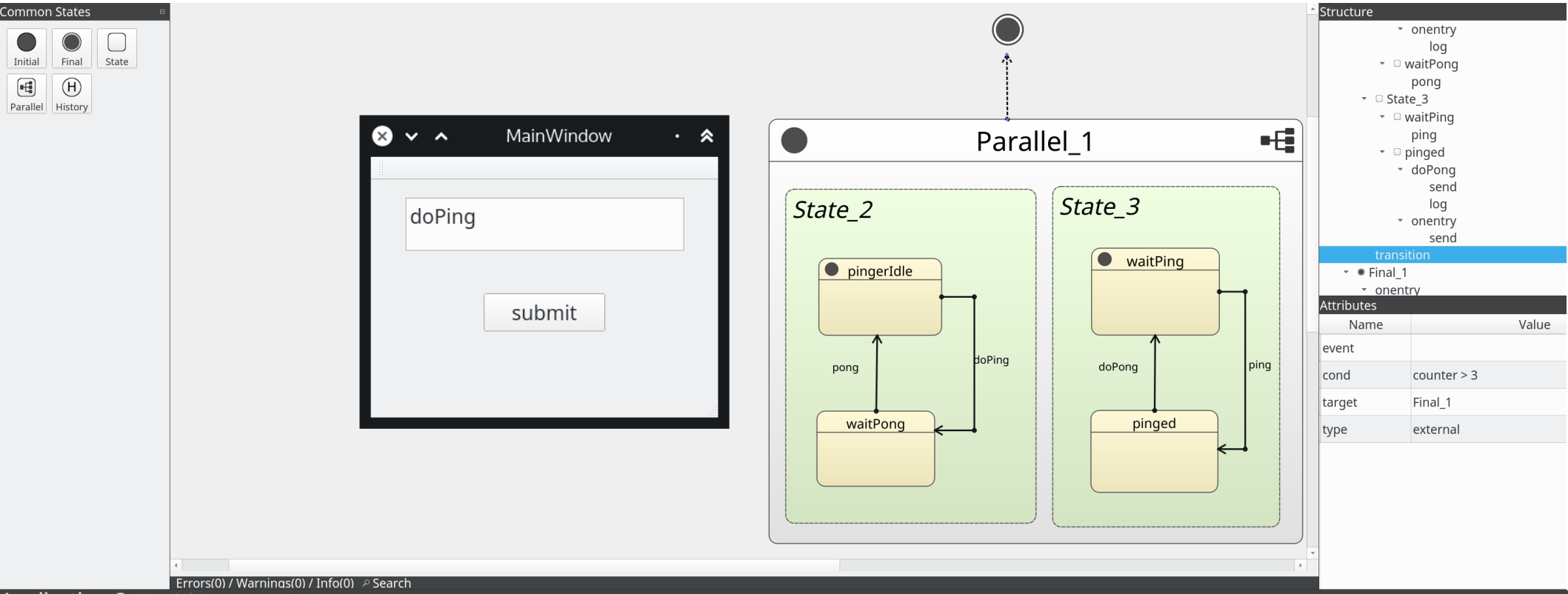


Application Output

testParallel testParallel

```
scxml.statemachine: "enter pingerIdle" : "counter = 0"
scxml.statemachine: "ping" : "35199ms"
scxml.statemachine: "pong" : "36251ms"
scxml.statemachine: "enter pingerIdle" : "counter = 1"
scxml.statemachine: "ping" : "37800ms"
scxml.statemachine: "pong" : "38753ms"
scxml.statemachine: "enter pingerIdle" : "counter = 2"
scxml.statemachine: "ping" : "39531ms"
scxml.statemachine: "pong" : "40584ms"
scxml.statemachine: "enter pingerIdle" : "counter = 3"
scxml.statemachine: "ping" : "42640ms"
scxml.statemachine: "enter final node" : "counter =4"
```

Guarded transition



Application Output

testParallel testParallel

```
scxml.statemachine: "enter pingerIdle" : "counter = 0"
scxml.statemachine: "ping" : "35199ms"
scxml.statemachine: "pong" : "36251ms"
scxml.statemachine: "enter pingerIdle" : "counter = 1"
scxml.statemachine: "ping" : "37800ms"
scxml.statemachine: "pong" : "38753ms"
scxml.statemachine: "enter pingerIdle" : "counter = 2"
scxml.statemachine: "ping" : "39531ms"
scxml.statemachine: "pong" : "40584ms"
scxml.statemachine: "enter pingerIdle" : "counter = 3"
scxml.statemachine: "ping" : "42640ms"
scxml.statemachine: "enter final node" : "counter =4"
```

https://blogs.itemis.com/en/deep-java-inte
-for-yakindu-state-machines