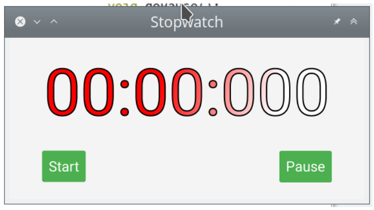
SCXML
State Chart XML

Au delà du transducteur à état fini

Stopwatch



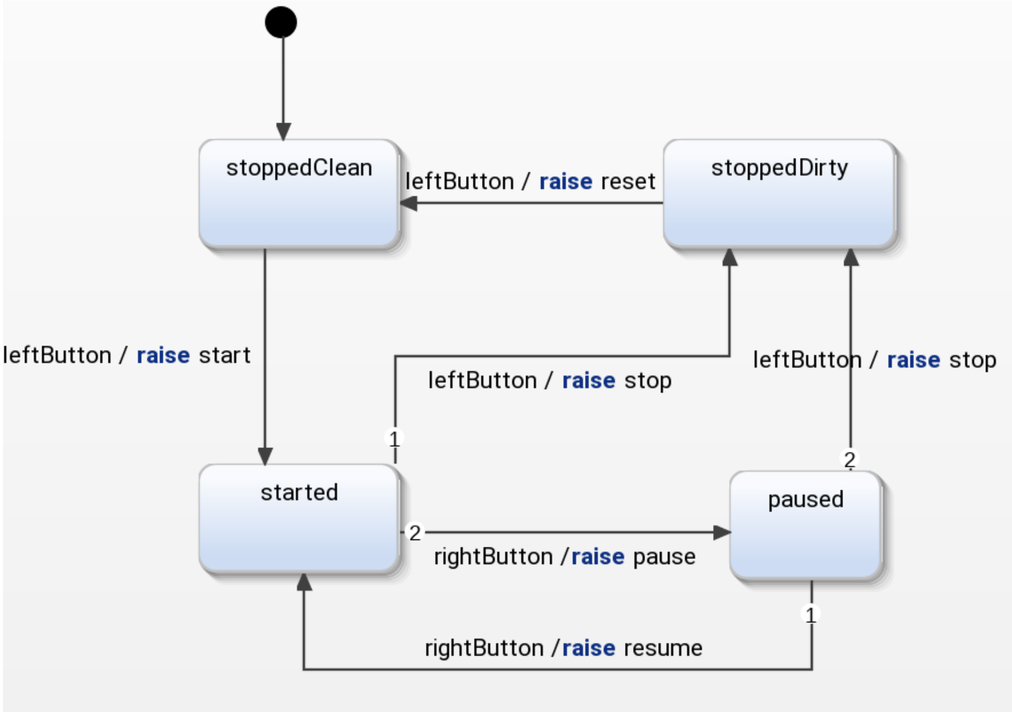
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

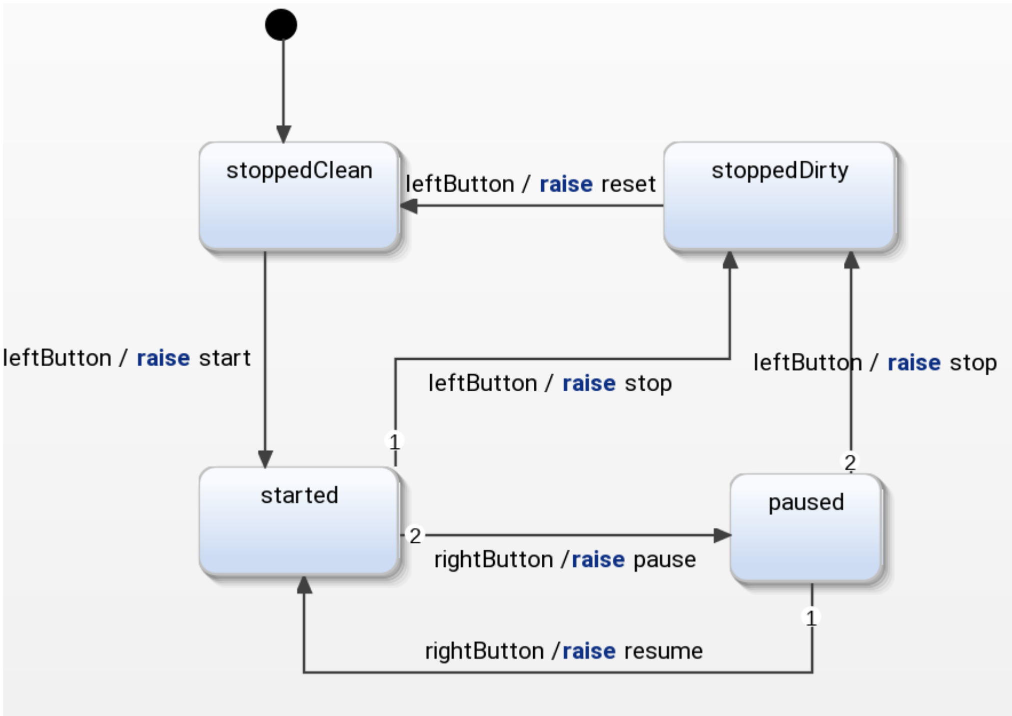
Stopwatch

interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void



Notion of behavior of the FSM



- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

$\mathcal{Q}$ is a set of State

$q_0 \in \mathcal{Q}$ is the initial state

$\mathcal{F}$ is the set of final states

Σ_I is the input alphabet

Σ_O is the output alphabet

$\delta \stackrel{\text{def}}{=} \mathcal{Q} \times \Sigma_I \times \Sigma_O \times \mathcal{Q}$

A finite state transducer is defined by $< \mathcal{Q}, q_0, \mathcal{F}, \Sigma_I, \Sigma_O, \delta >$

Consider an automaton $< \mathcal{Q}, q_0, \Sigma_I \times \Sigma_O, \delta' >$ where

$(s, (i,o), s') \in \delta'$ iff $(s, i, o, s') \in \delta$.

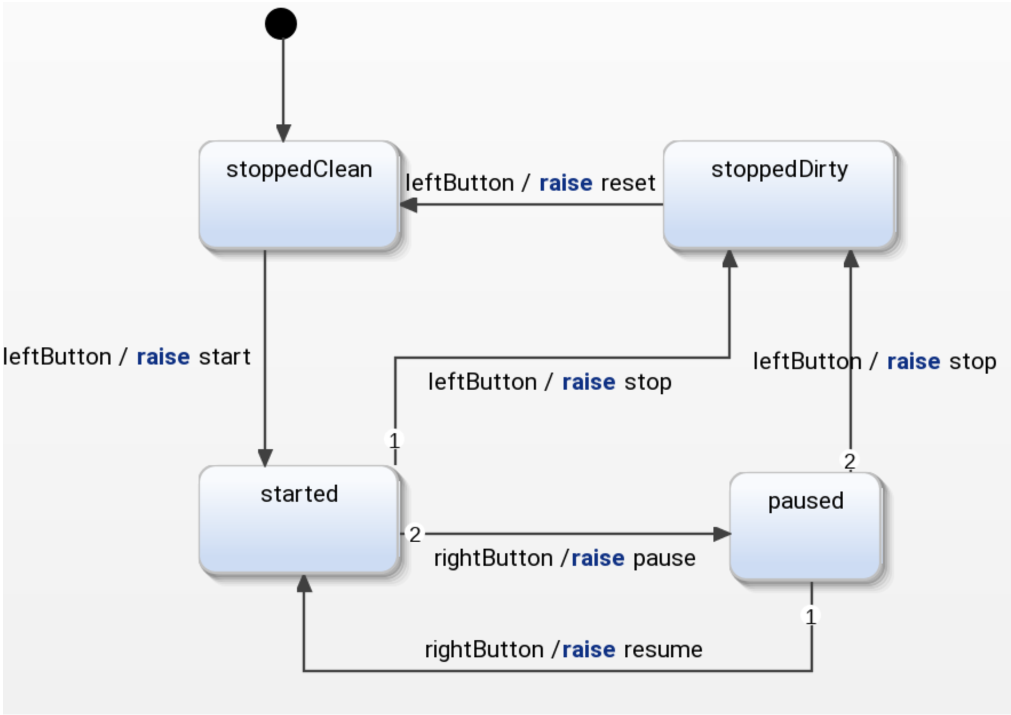
The language accepted by this automaton is the **language of the FSM at state q_0**

This language is sometimes called '**behavior**'

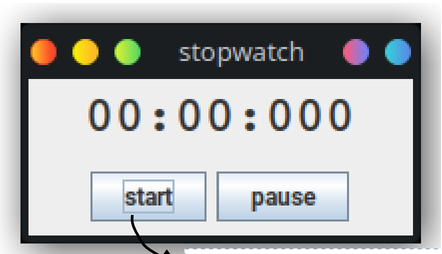
Stopwatch


```
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
```

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void



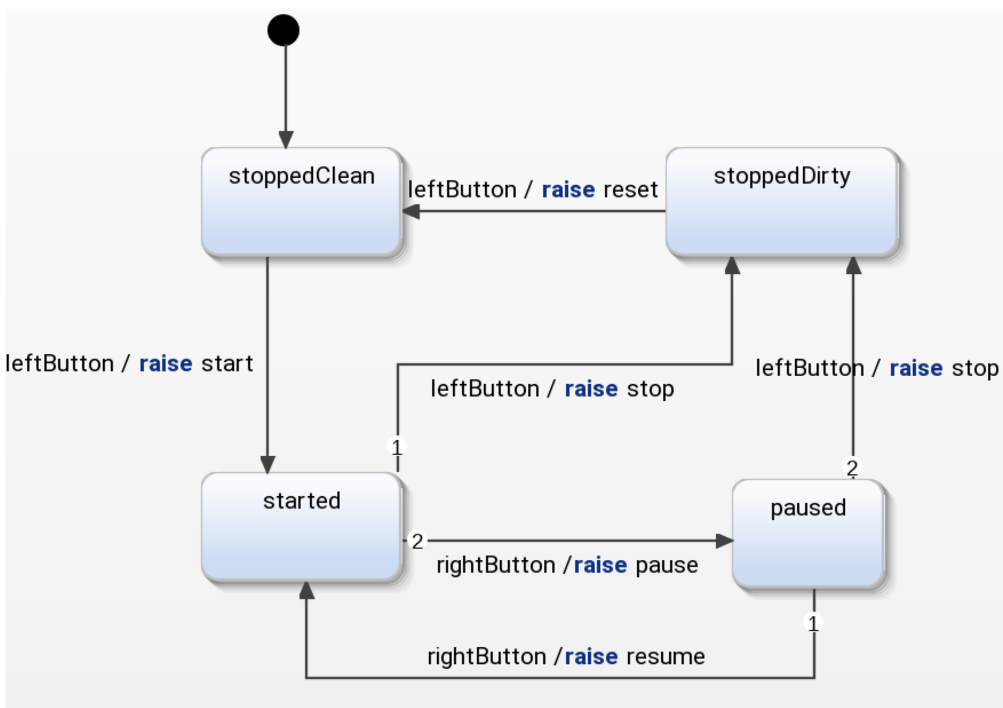
Stopwatch



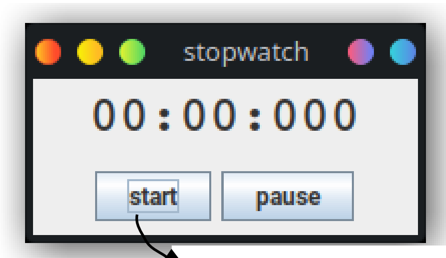
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

```
leftButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        theFSM.raiseLeftButton();
    }
});
```



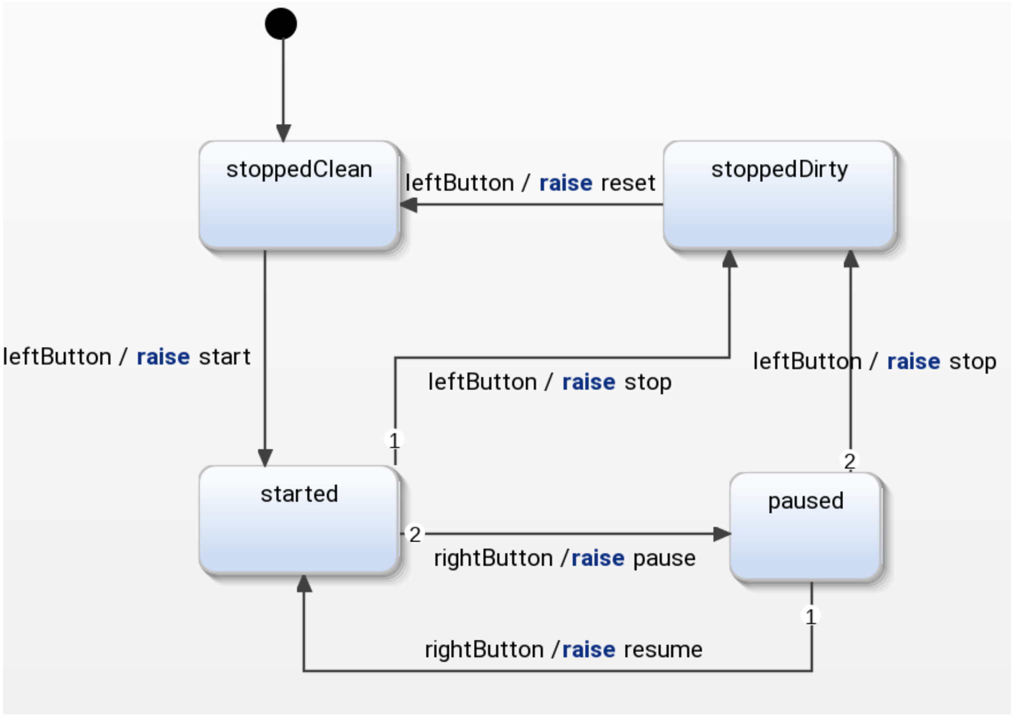
Stopwatch



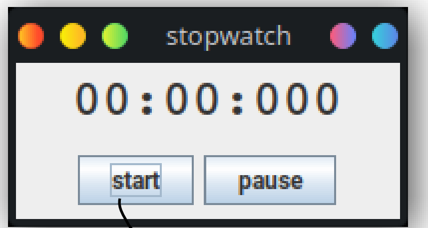
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

```
leftButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        theFSM.raiseLeftButton();
    }
});
```



Stopwatch

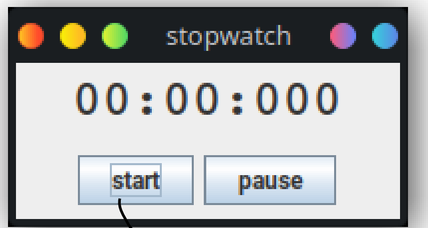


- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

```
leftButton.setOnAction(new EventHandler<ActionEvent>() {
    @Overri
    public
        the od.
    }
});
```

- setOnAction(EventHandler<ActionEvent> value) : void - ButtonBase
- setOnContextMenuRequested(EventHandler<? super ContextMenuEvent> value) : void - Node
- setOnDragDetected(EventHandler<? super MouseEvent> value) : void - Node
- setOnDragDone(EventHandler<? super DragEvent> value) : void - Node
- setOnDragDropped(EventHandler<? super DragEvent> value) : void - Node
- setOnDragEntered(EventHandler<? super DragEvent> value) : void - Node
- setOnDragExited(EventHandler<? super DragEvent> value) : void - Node
- setOnDragOver(EventHandler<? super DragEvent> value) : void - Node
- setOnInputMethodTextChanged(EventHandler<? super InputMethodEvent> value) : void - Node
- setOnKeyPressed(EventHandler<? super KeyEvent> value) : void - Node
- setOnKeyReleased(EventHandler<? super KeyEvent> value) : void - Node
- setOnKeyTyped(EventHandler<? super KeyEvent> value) : void - Node
- setOnMouseClicked(EventHandler<? super MouseEvent> value) : void - Node
- setOnMouseDragEntered(EventHandler<? super MouseDragEvent> value) : void - Node
- setOnMouseDragExited(EventHandler<? super MouseDragEvent> value) : void - Node
- setOnMouseDragged(EventHandler<? super MouseEvent> value) : void - Node
- setOnMouseDragOver(EventHandler<? super MouseDragEvent> value) : void - Node
- setOnMouseDragReleased(EventHandler<? super MouseDragEvent> value) : void - Node
- setOnMouseEntered(EventHandler<? super MouseEvent> value) : void - Node
- setOnMouseExited(EventHandler<? super MouseEvent> value) : void - Node
- setOnMouseMoved(EventHandler<? super MouseEvent> value) : void - Node
- setOnMousePressed(EventHandler<? super MouseEvent> value) : void - Node
- setOnMouseReleased(EventHandler<? super MouseEvent> value) : void - Node
- setOnRotate(EventHandler<? super RotateEvent> value) : void - Node
- setOnRotationFinished(EventHandler<? super RotateEvent> value) : void - Node
- setOnRotationStarted(EventHandler<? super RotateEvent> value) : void - Node

Stopwatch



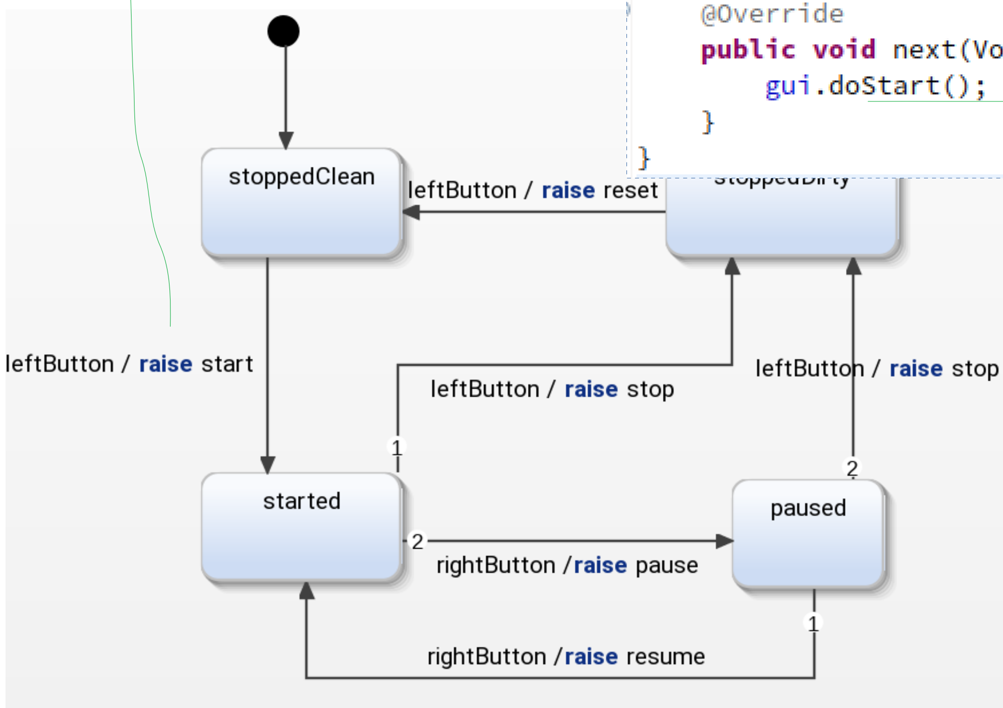

```
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
```

```
theFSM.getStart().subscribe(new StartObserver(this));
```

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

```
leftButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        theFSM.getSCInterface().raiseLeftButton();
    }
});
```

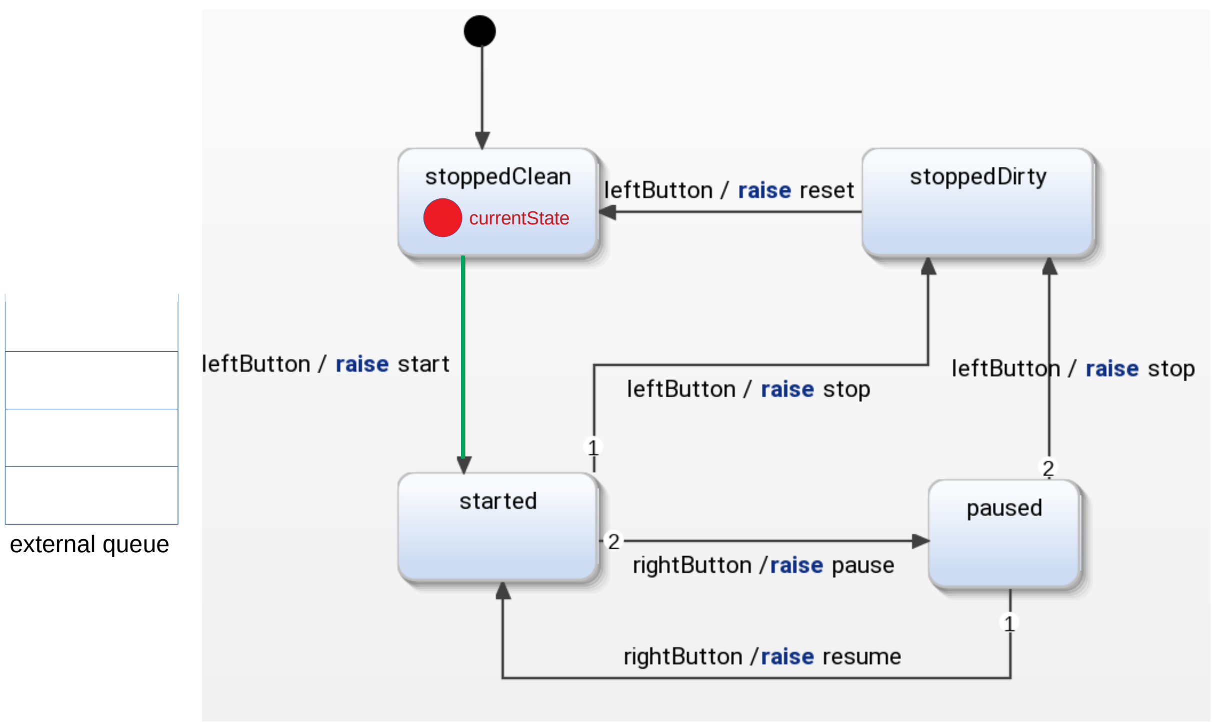
```
public class StartObserver implements Observer<Void> {
    private StopWatchGui gui;
    public StartObserver(StopWatchGui theGui) {
        gui = theGui;
    }
    @Override
    public void next(Void value) {
        gui.doStart();
    }
}
```



Stopwatch

interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

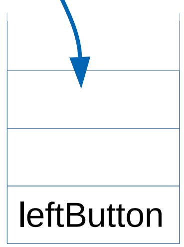


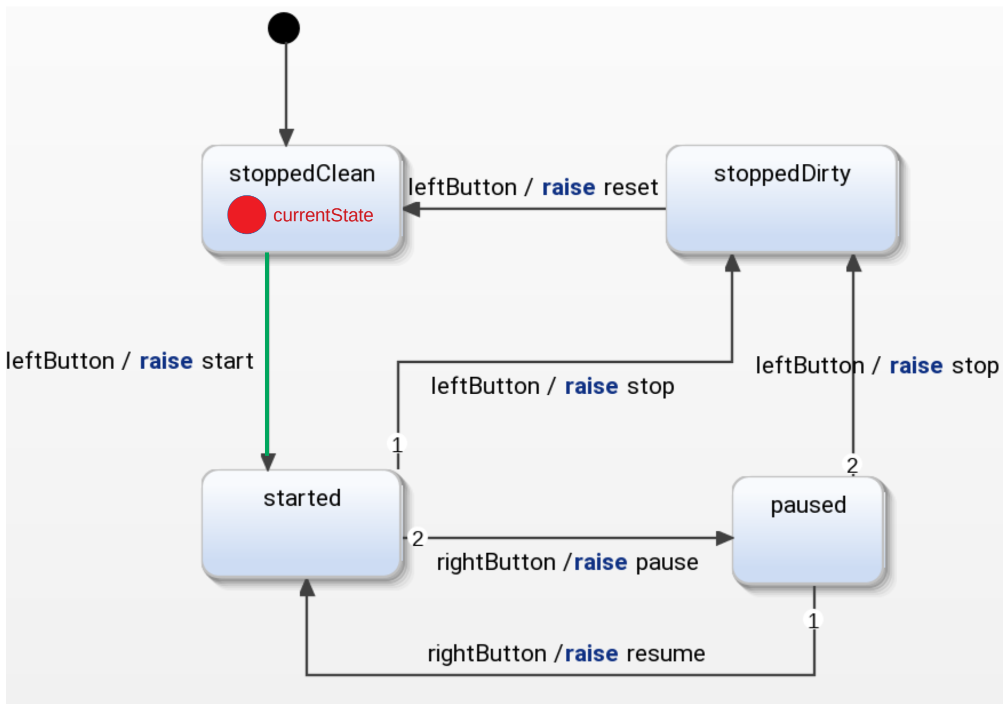
Stopwatch

```
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
```

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

Inject an event

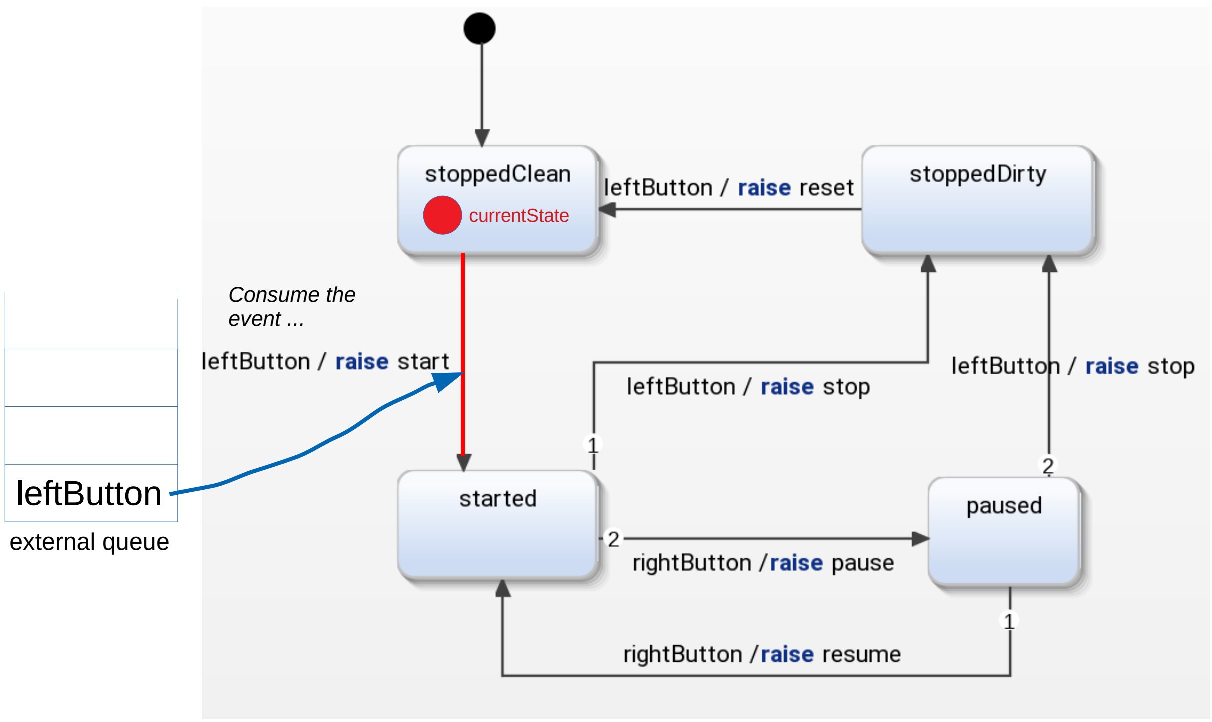




Stopwatch

interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

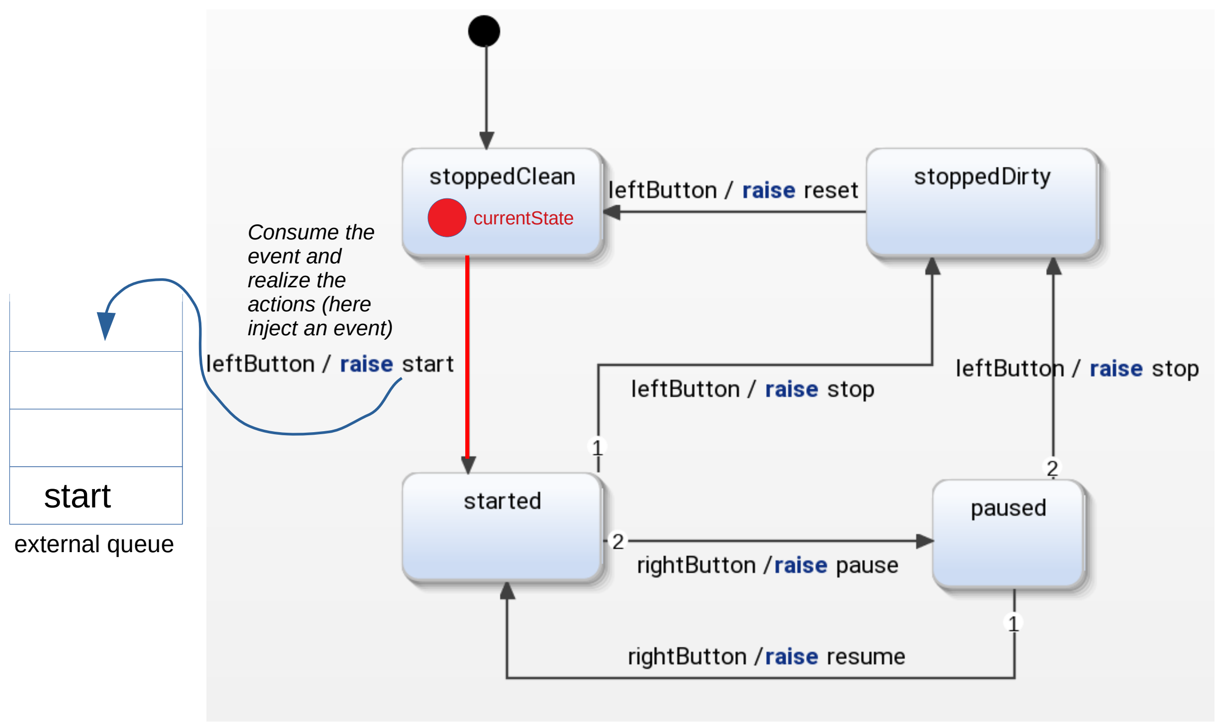
- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void



Stopwatch

interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

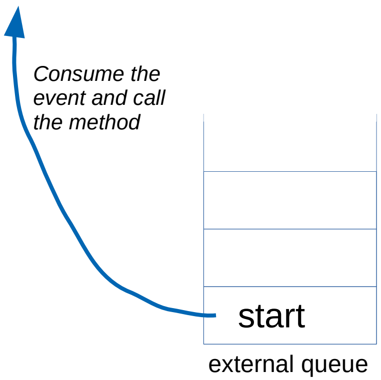


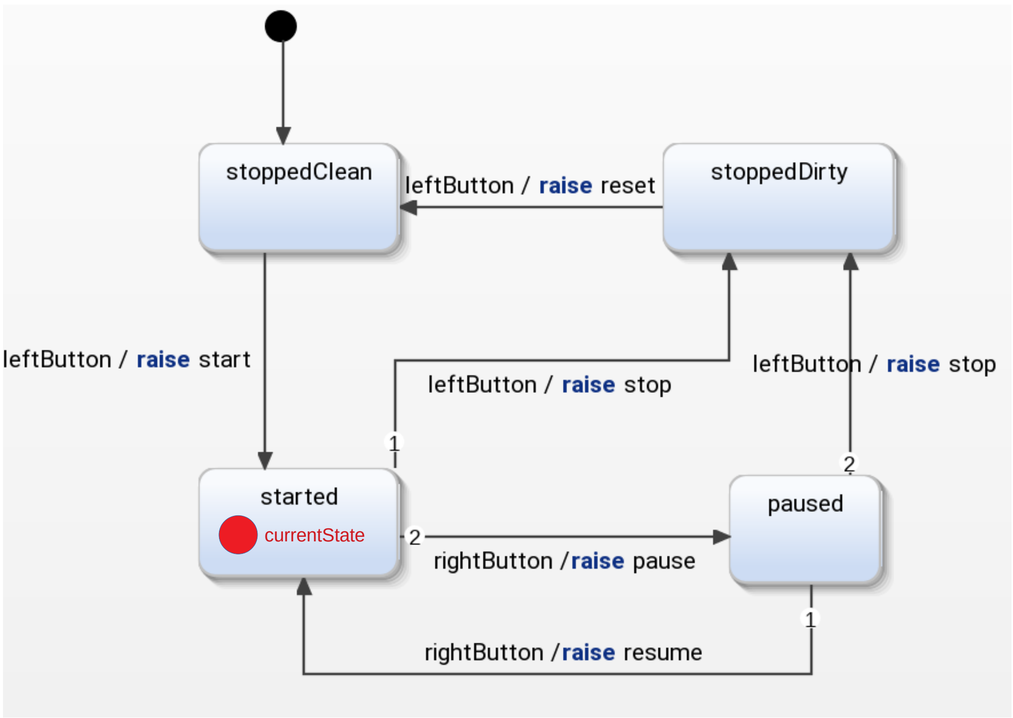
Stopwatch

interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

```
protected void doStart() {
    msTimer.start();
    updateTimer.start();
    leftButton.setText("stop");
    rightButton.setText("pause");
}
```

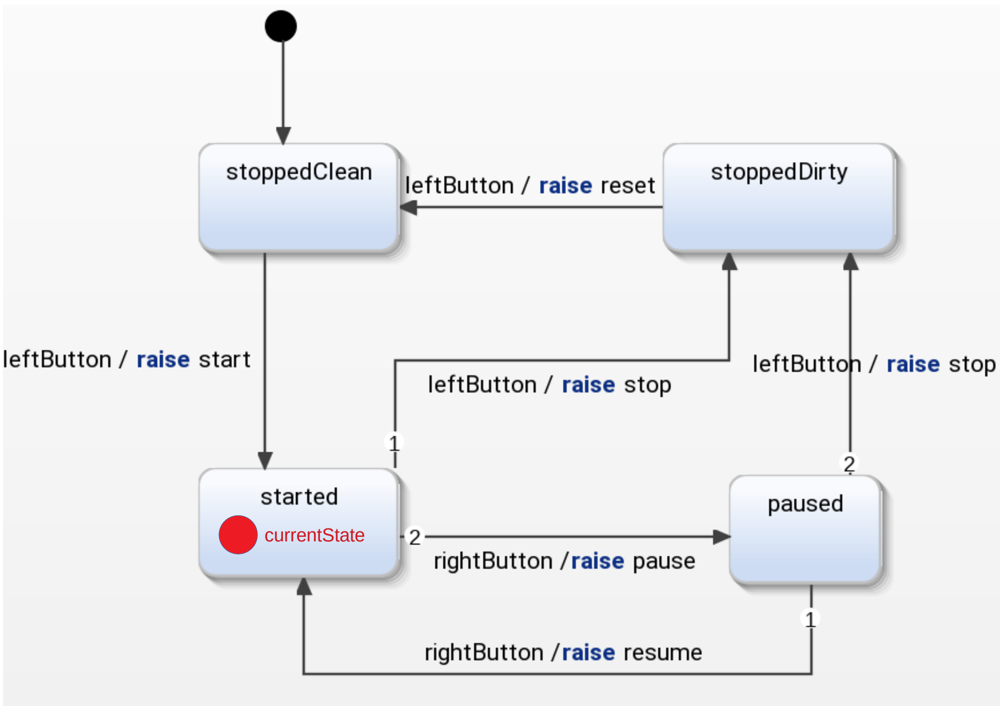




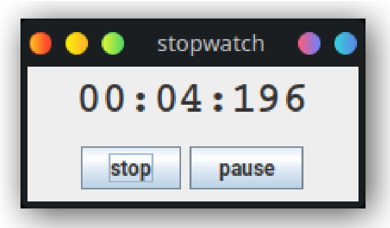
Stopwatch

interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

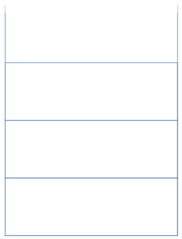


Stopwatch

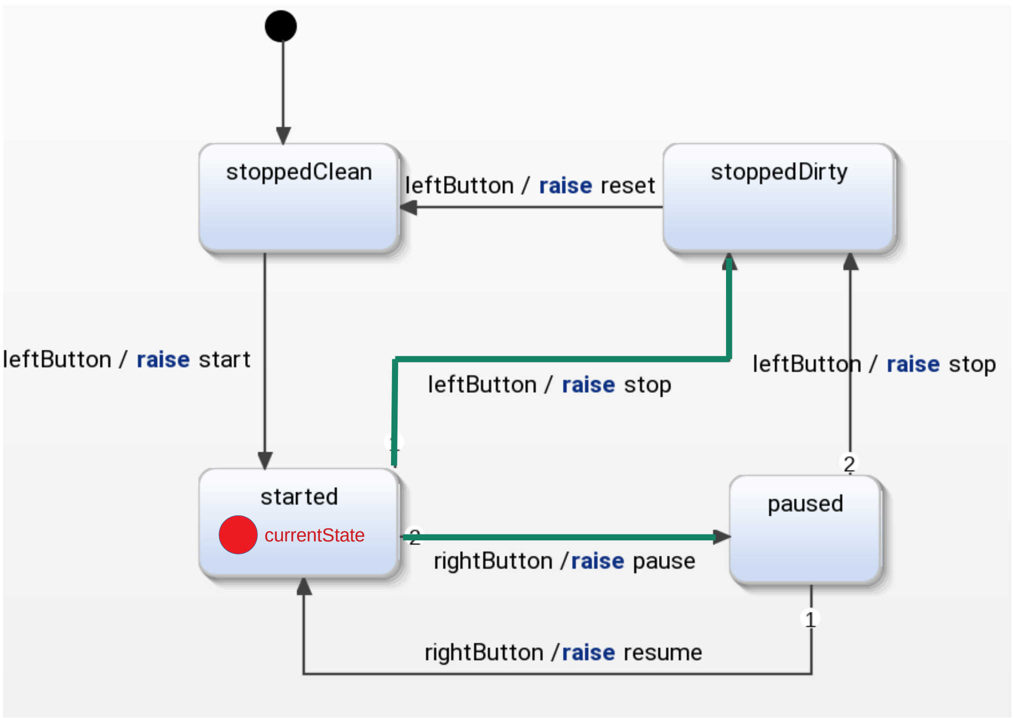


interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

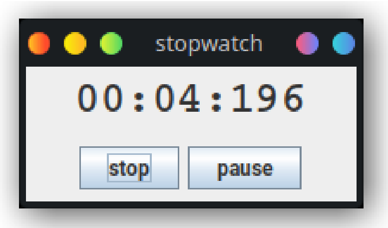
- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void



external queue

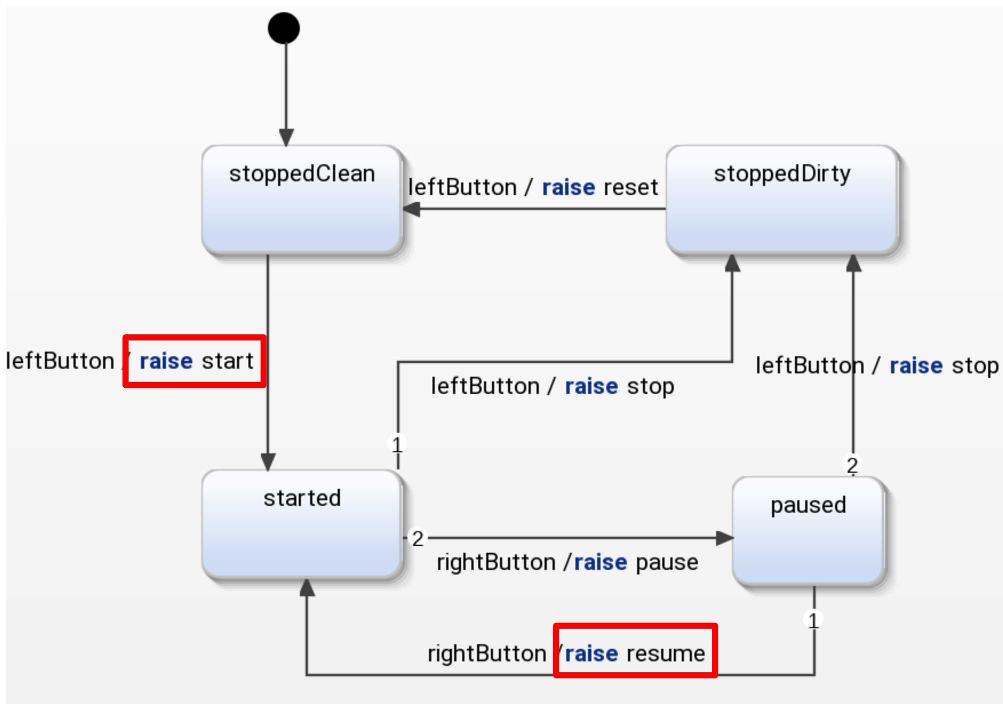


Stopwatch



interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void



Mealy

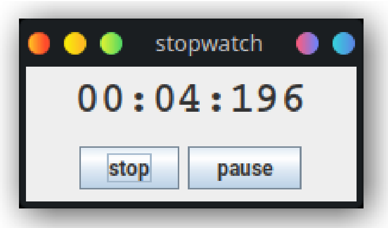
```java
protected void doStart() {
    msTimer.start();
    updateTimer.start();
    leftButton.setText("stop");
    rightButton.setText("pause");
}
```

```java
protected void doResume() {
    updateTimer.stop() ;
    rightButton.setText("pause");
}
```

Stopwatch

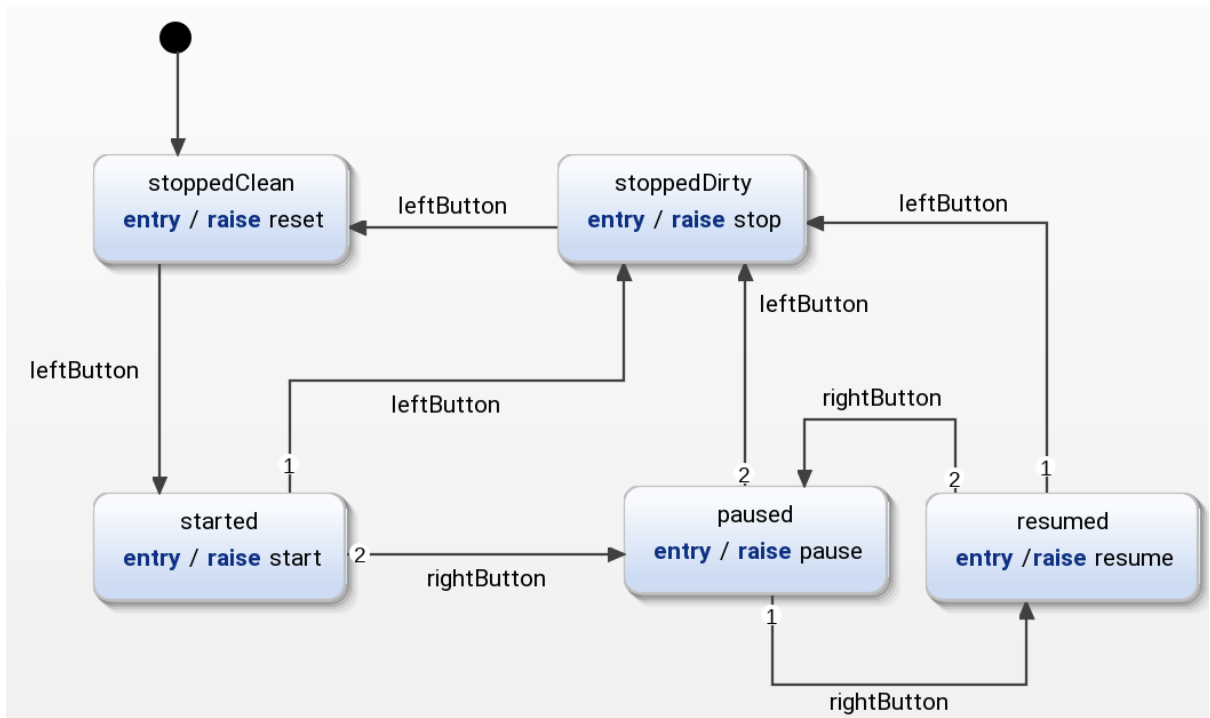
```
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
```

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void



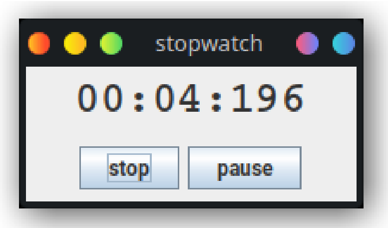

```
protected void doStart() {
    msTimer.start();
    updateTimer.start();
    leftButton.setText("stop");
    rightButton.setText("pause");
}
```

```
protected void doResume() {
    updateTimer.stop() ;
    rightButton.setText("pause");
}
```



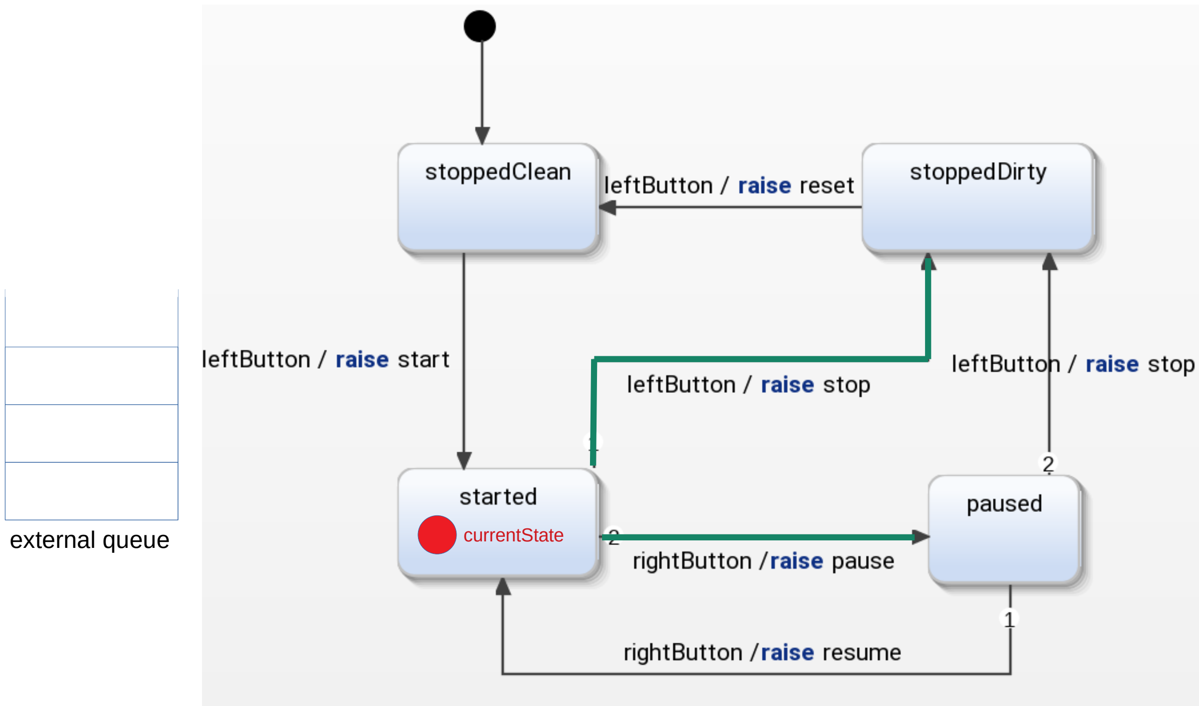
Moore

Stopwatch



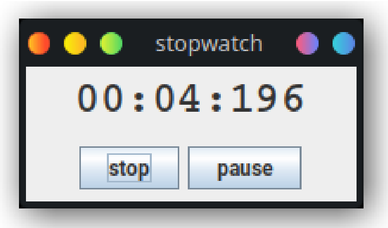
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void

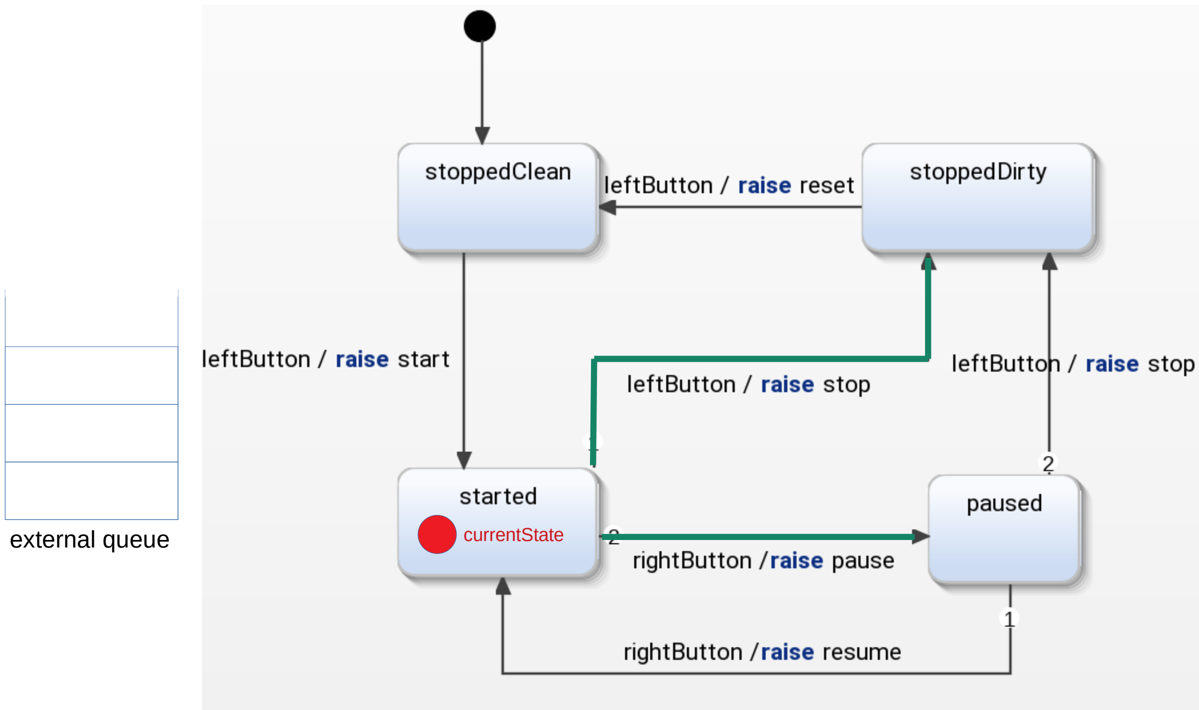


Stopwatch

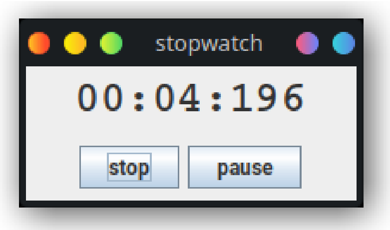
interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume



- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void
- updateText() : void

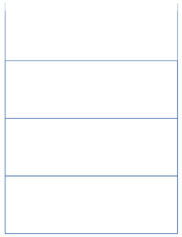


Stopwatch

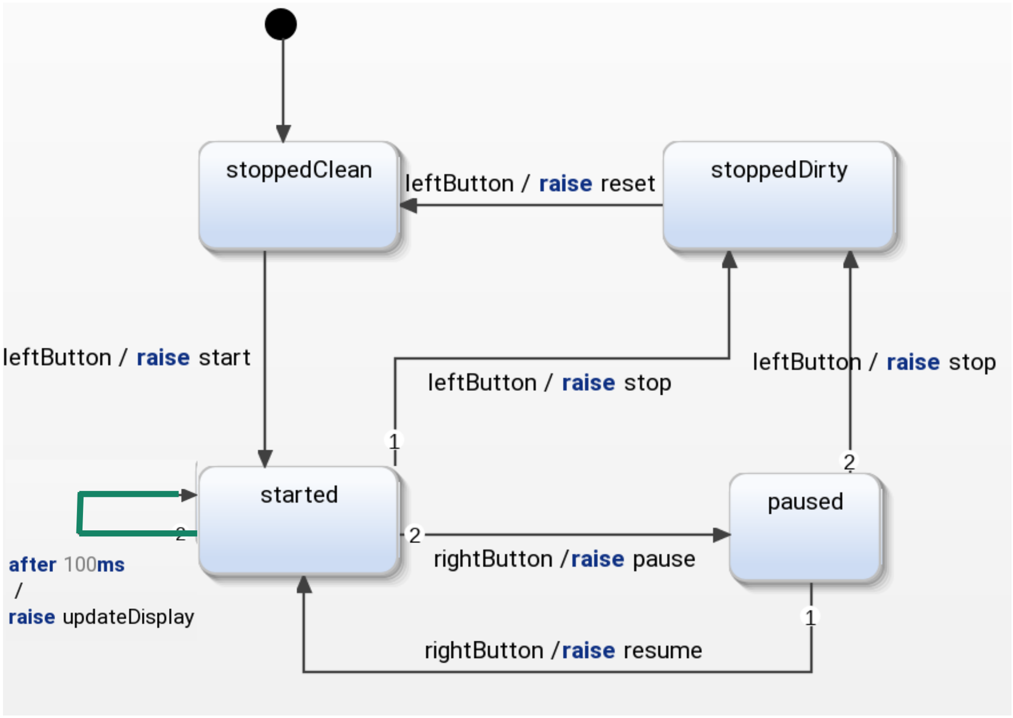


interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
out event updateDisplay

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void
- updateText() : void

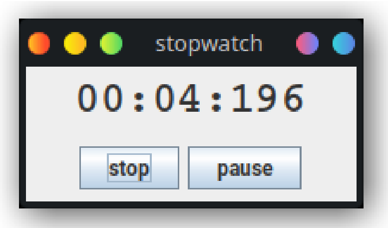


external queue



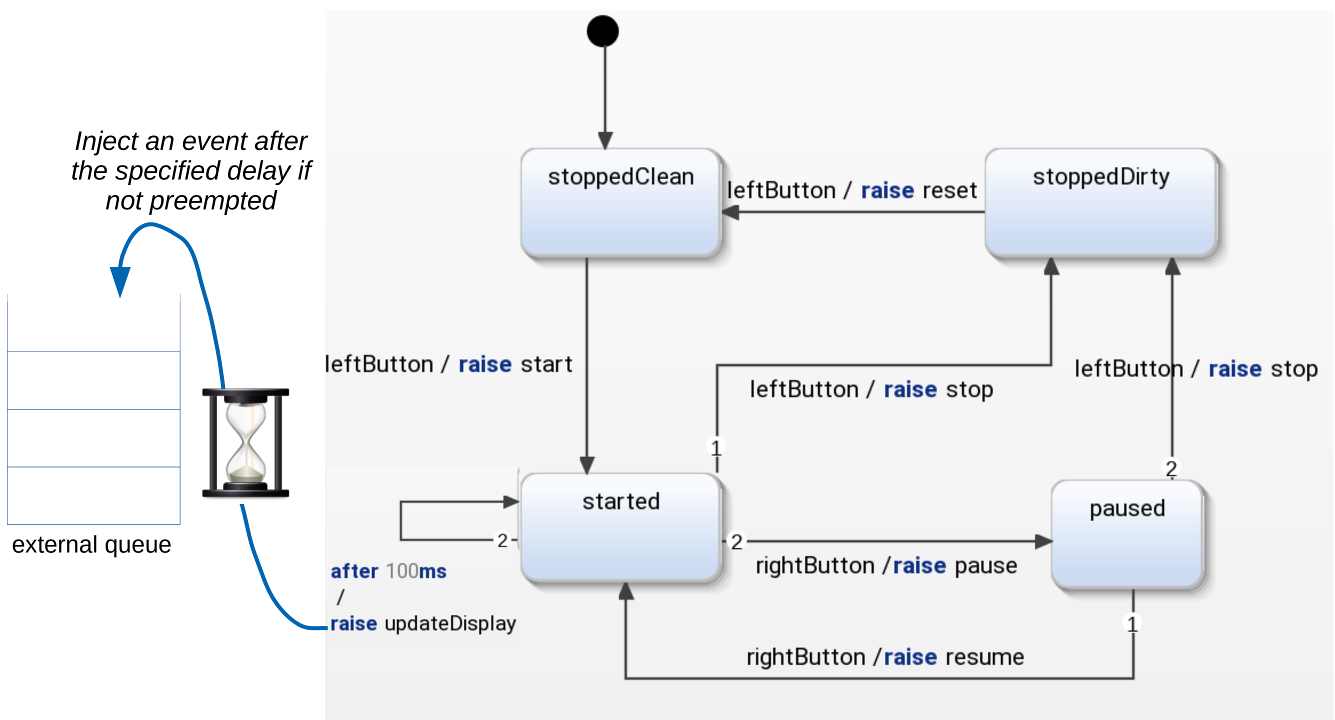
Timed Automata

Stopwatch

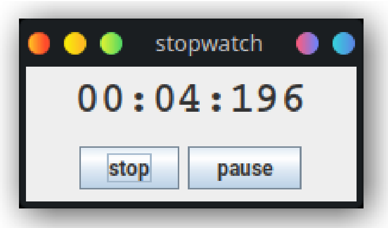


interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
out event updateDisplay

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void
- updateText() : void

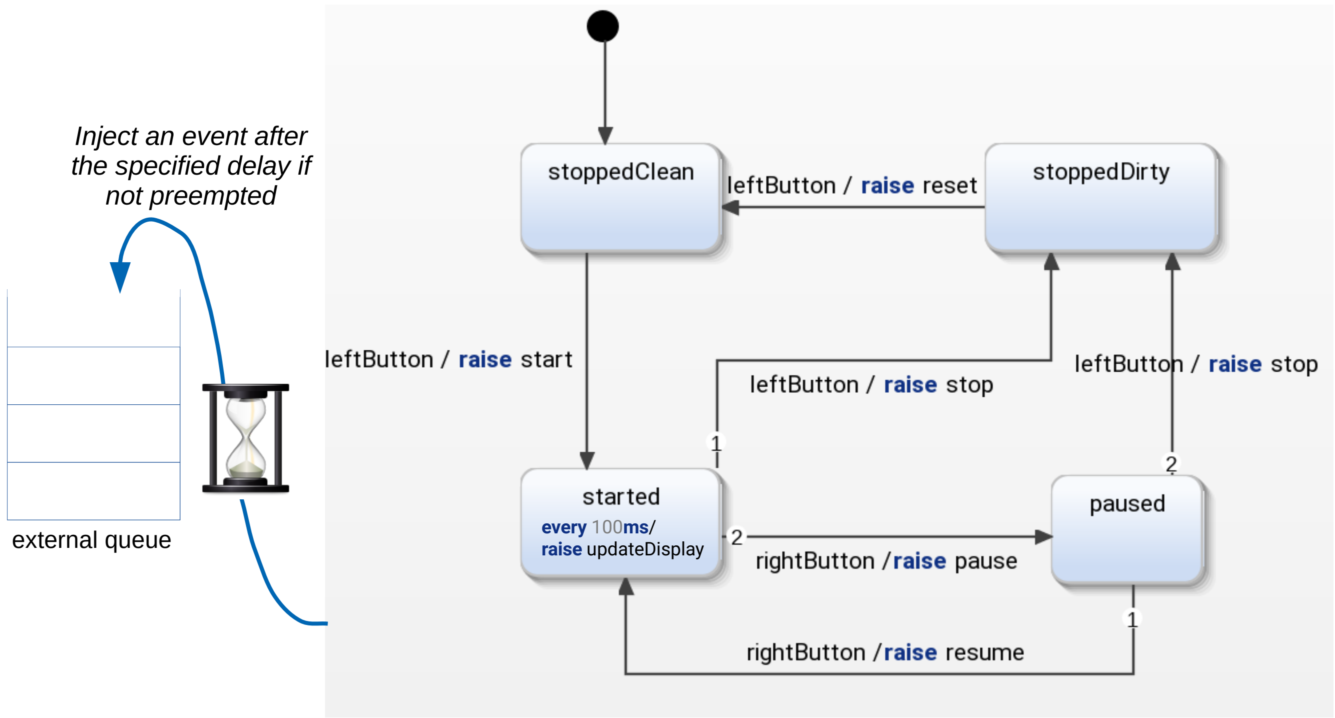


Stopwatch



interface:
in event leftButton
in event rightButton
out event start
out event stop
out event reset
out event pause
out event resume
out event updateDisplay

- doReset() : void
- doResume() : void
- doPause() : void
- doStop() : void
- doStart() : void
- updateText() : void



SCXML
State Chart XML

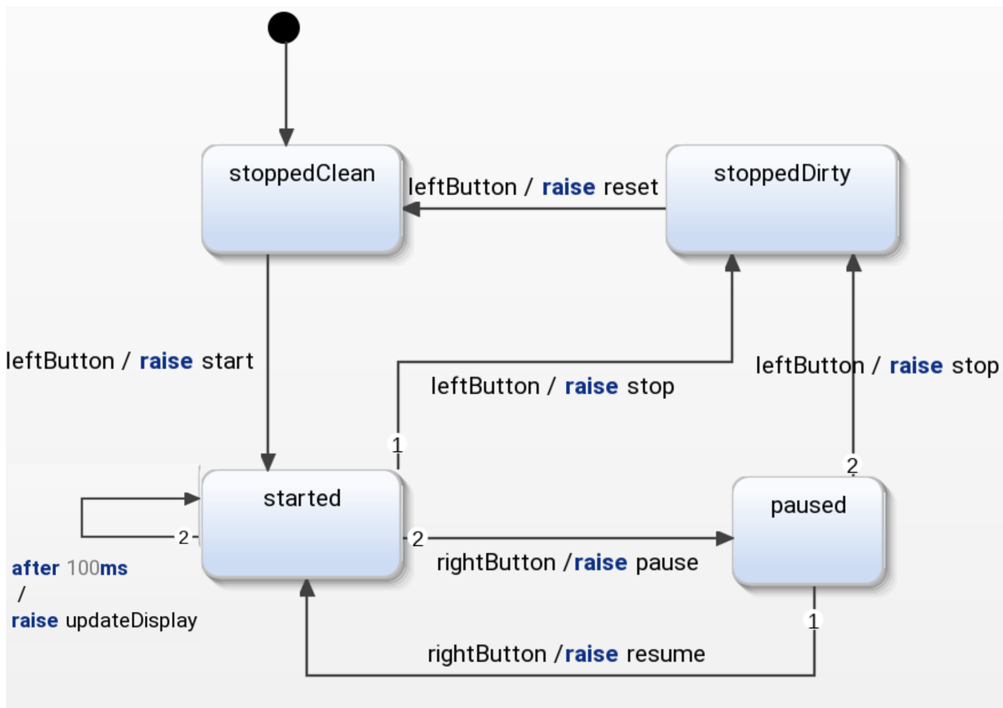
statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.

Stopwatch

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.



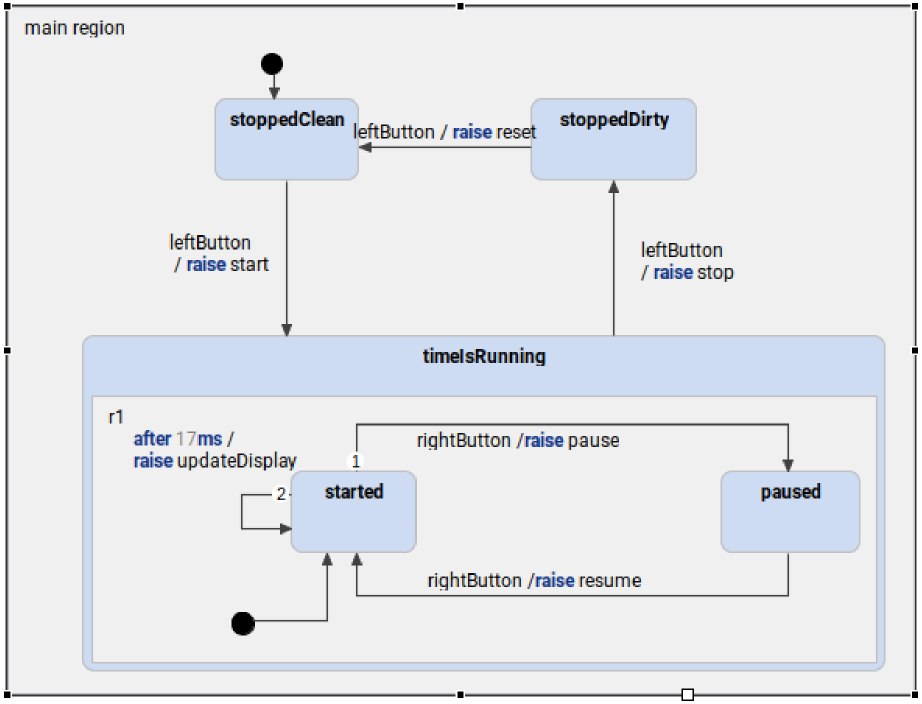
- A simple state is one which has no substructure.

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

Stopwatch

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.



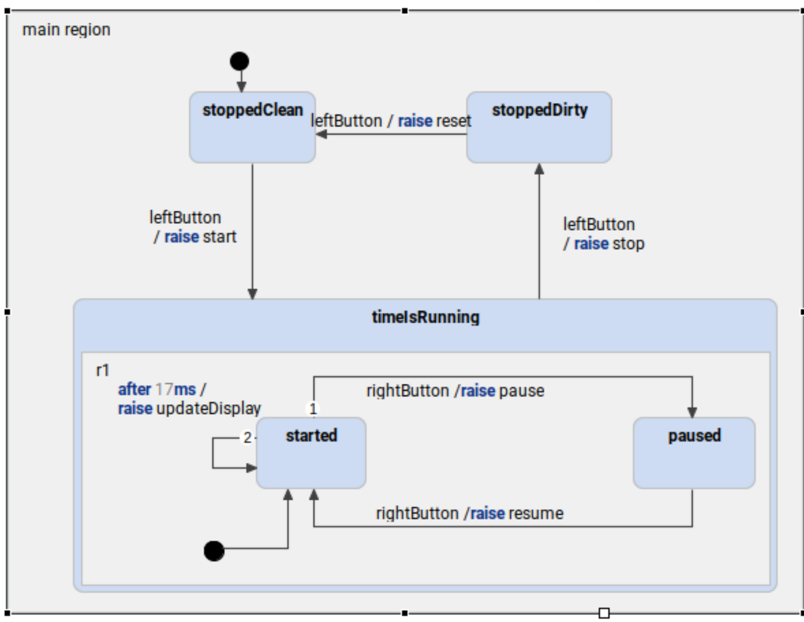
- A simple state is one which has no substructure.
- A state which has substates (nested states) is called a composite state (or compound state).
- Substates may be nested to any level. A nested state machine may have at most one initial state.
- **Substates are used to simplify complex flat state machines by showing that some states are only possible/accessible 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

Stopwatch

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.



When leftButton occurs:

1. Leave *stoppedClean*
2. Enter *timeIsRunning*
3. Enter *started*
4. *After 17ms* (no *rightButton*)
   1. Leave *started*
   2. Enter *started*

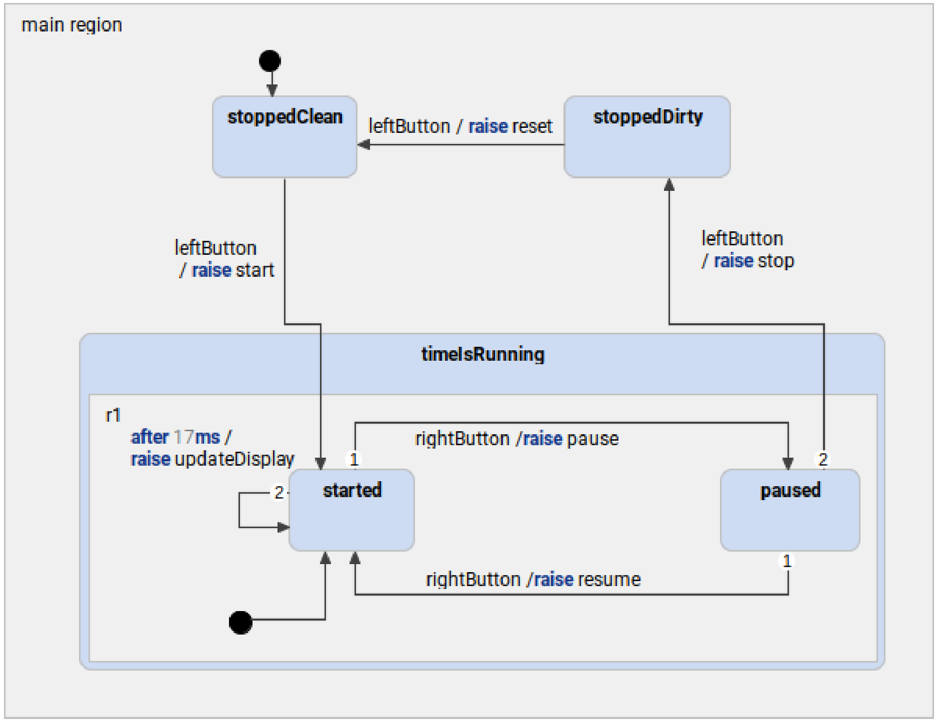
- A simple state is one which has no substructure.
- A state which has substates (nested states) is called a composite state (or compound state).
- Substates may be nested to any level. A nested state machine may have at most one initial state and one final state.
- **Substates are used to simplify complex 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

Stopwatch

statecharts = state-diagrams + depth

+ orthogonality + broadcast-communication.

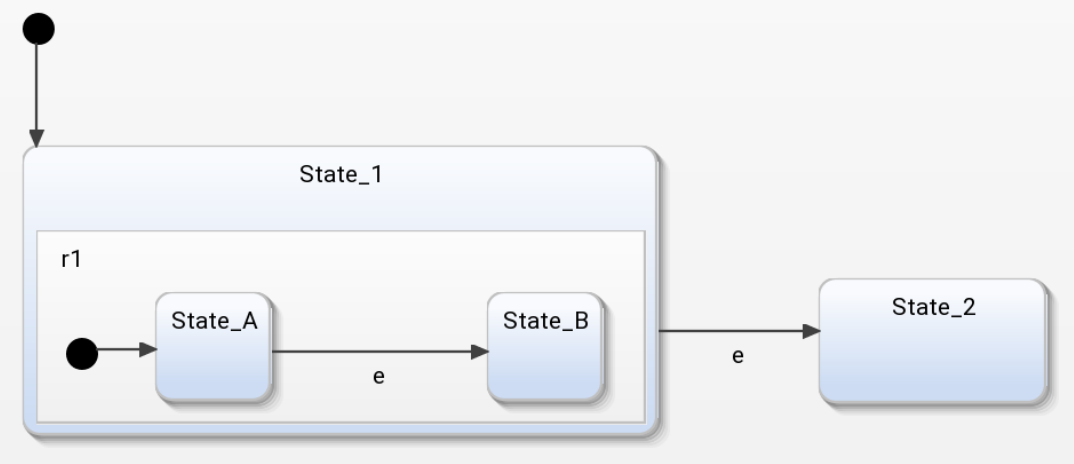


Syntactically correct but the **behavior** is not the expected one

- 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**

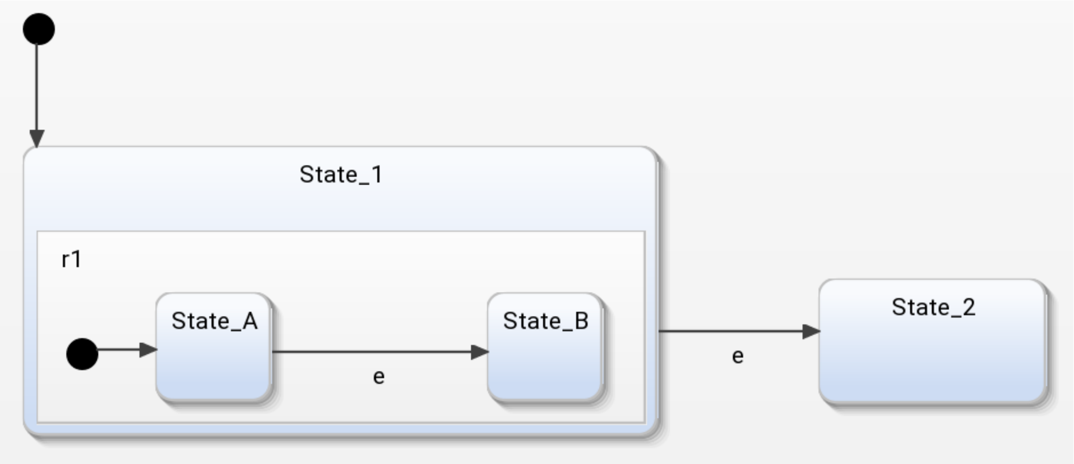
Taken and modified from http://sce.uhcl.edu/helm/rationalunifiedprocess/process/modguide/md_stadm.htm

Composite State



After initialization, '*e*' is injected. What happens and why ?

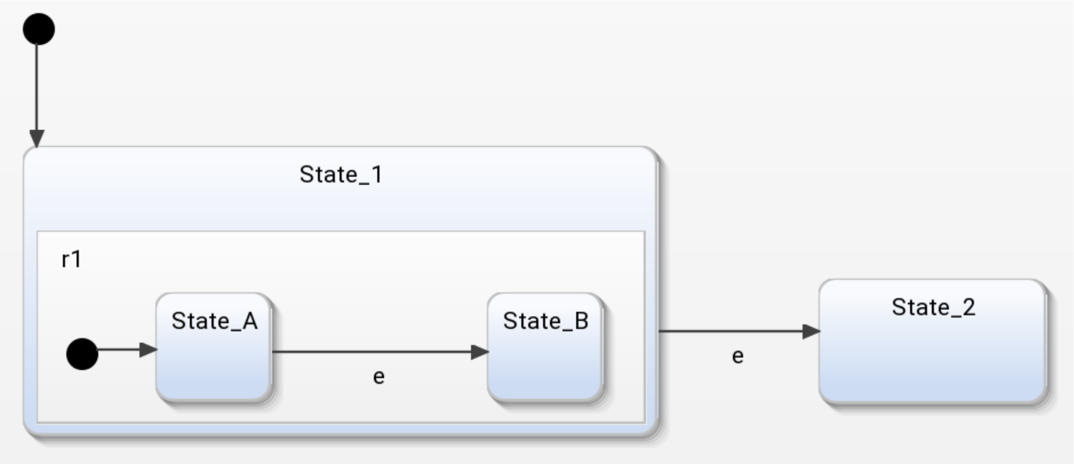
Composite State



After initialization, '*e*' is injected. What happens and why ?

- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

Composite State

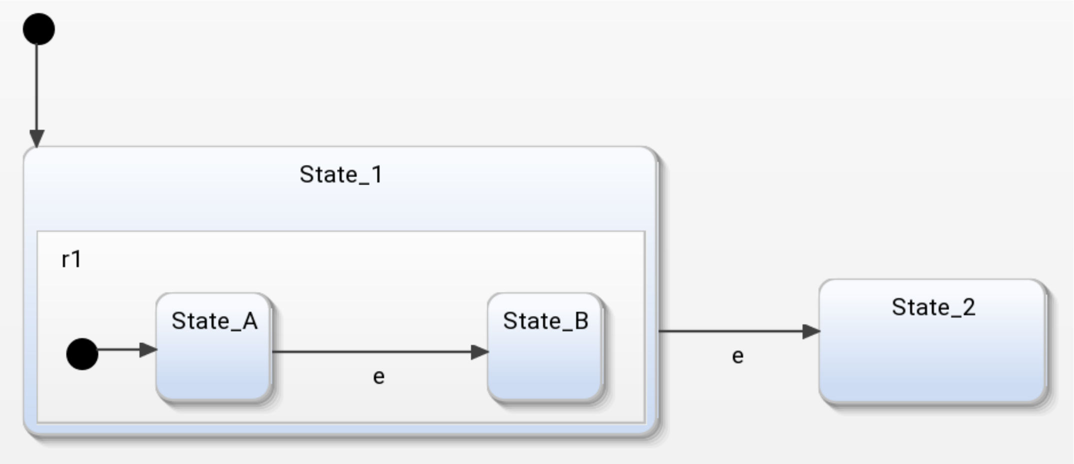


After initialization, ‘*e*’ is injected. What happens and why ?

- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

enter *State_1*;
enter *State_A*;
Inject *e*
exit *State_A*;
enter *State_B;*

Composite State

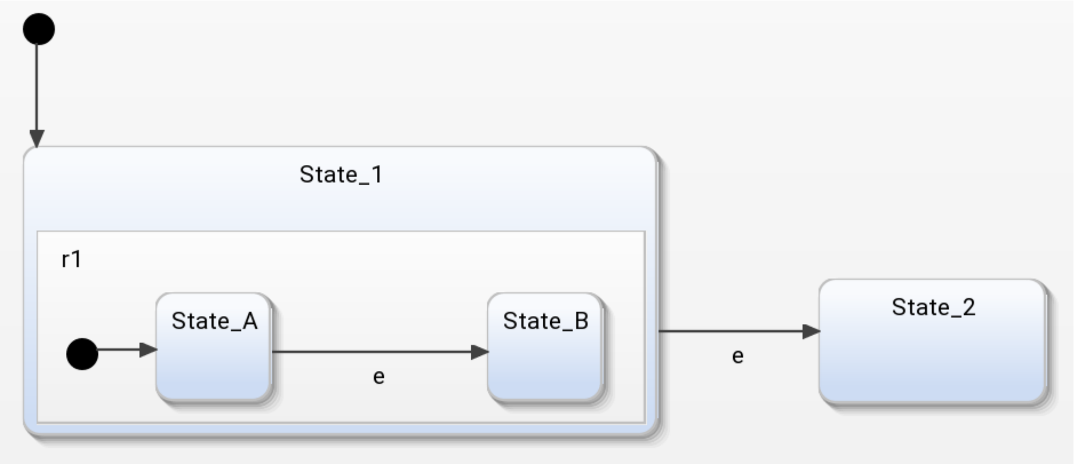


After initialization, ‘*e*’ is injected. What happens and why ?

- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

```
enter State_1;          Inject e;
enter State_A;          exit State_B;
Inject e;               exit State_1;
exit State_A;           enter State_2;
enter State_B;
```

Composite State



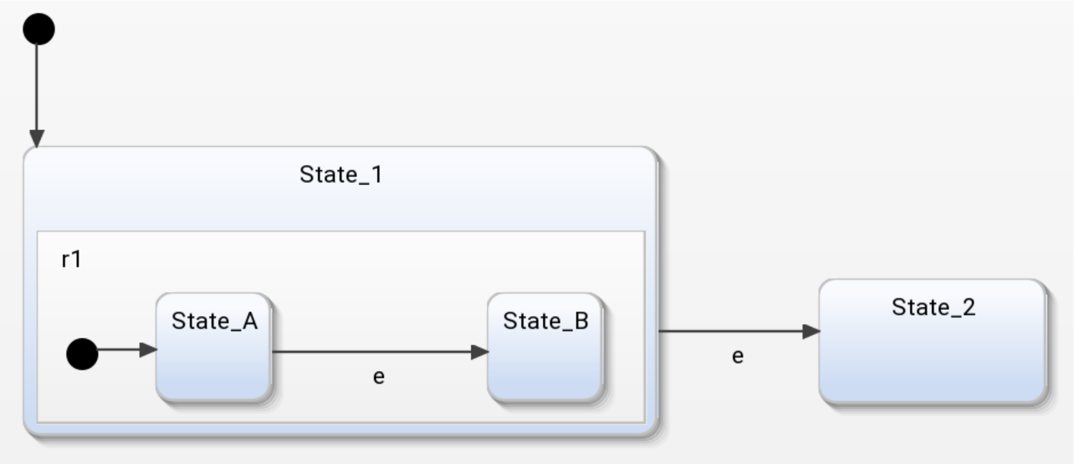
After initialization, ‘*e*’ is injected. What happens and why ?

- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

```
enter State_1;
enter State_A;
Inject e;
exit State_A;
enter State_B;
```

```
Inject e;
exit State_B;
exit State_1;
enter State_2;
Inject e;
Inject e;
```

Composite State



After initialization, ‘*e*’ is injected. What happens and why ?

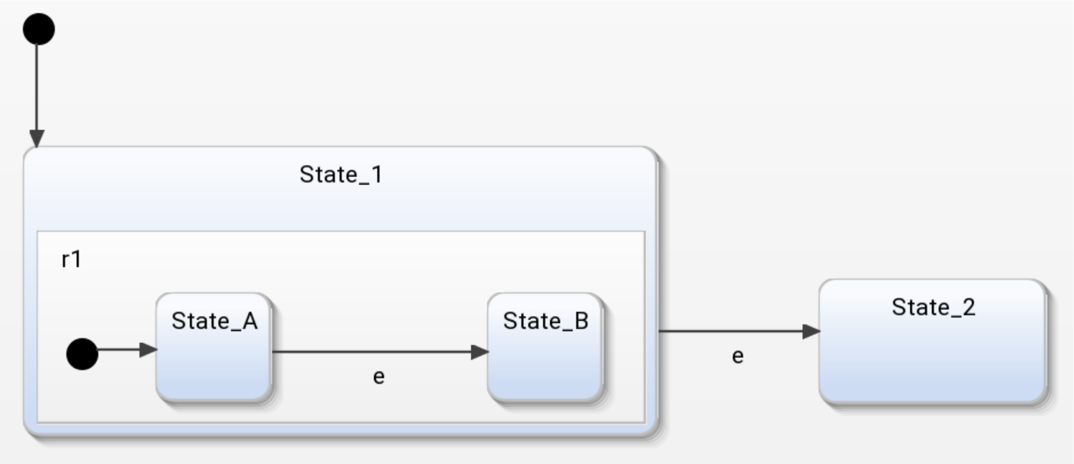
- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

enter *State_1*;
enter *State_A*;
Inject *e*
exit *State_A*;
enter *State_B*;

In Yakindu, this is a **semantic variation point**, i.e., a part of the semantics that can be adjusted by the user

@ChildFirstExecution → SCXML semantics
@ParentFirstExecution → Simulink Stateflow semantics

Composite State



After initialization, '*e*' is injected. What happens and why ?

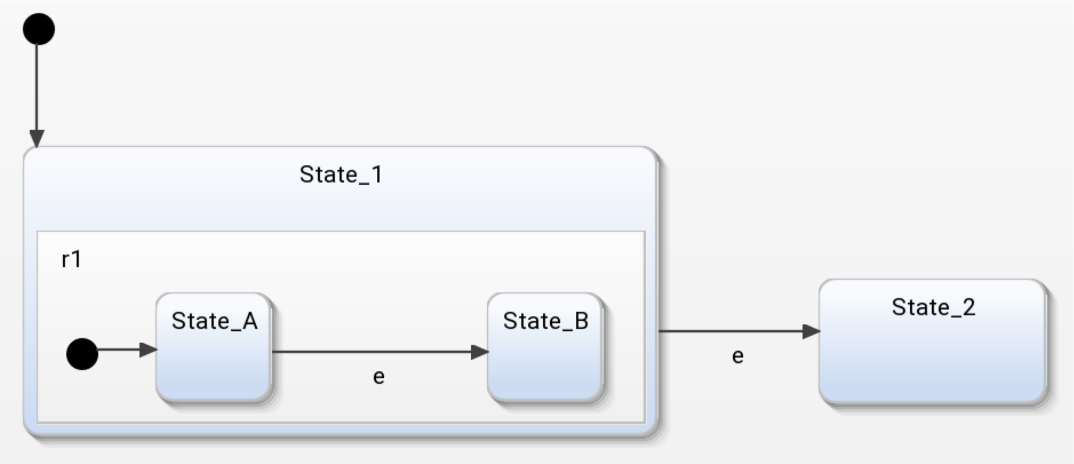
- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

```
enter State_1;
enter State_A;
Inject e
exit State_A;
enter State_B;
```

In Yakindu, this is a semantic variation point, i.e., a part of the semantic that can be adjusted by the user

@ChildFirstExecution → SCXML semantics
@ParentFirstExecution → Simulink Stateflow semantics

Composite State



After initialization, ‘*e*’ is injected. What happens and why ?

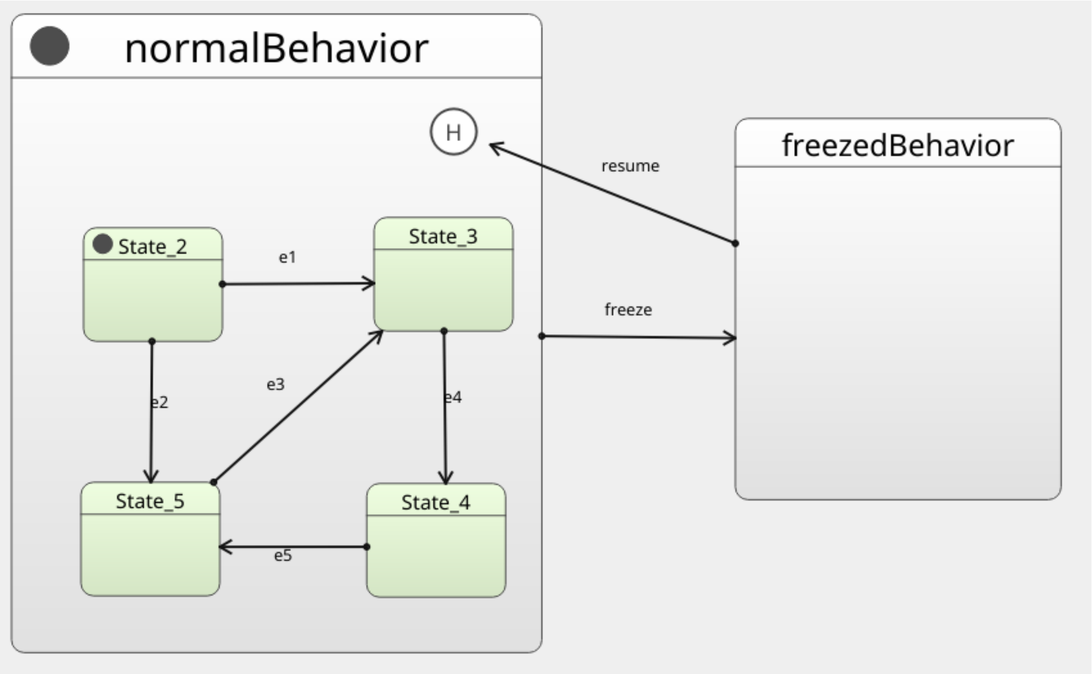
- Compound States: When looking for transitions, the state machine first looks in the most deeply nested active state(s), i.e., in the atomic state(s) that have no substates. If no transitions match in the atomic state, the state machine will look in its parent state, then in the parent's parent, etc. Thus transitions in ancestor states serve as defaults that will be taken if no transition matches in a descendant state. If no transition matches in any state, the event is discarded.

```
enter State_1;
enter State_A;
Inject e
exit State_A;
Exit State_1;
enter State_2;
```

In Yakindu, this is a semantic variation point, i.e., a part of the semantic that can be adjusted by the user

@ChildFirstExecution → SCXML semantics
@ParentFirstExecution → Simulink Stateflow semantics

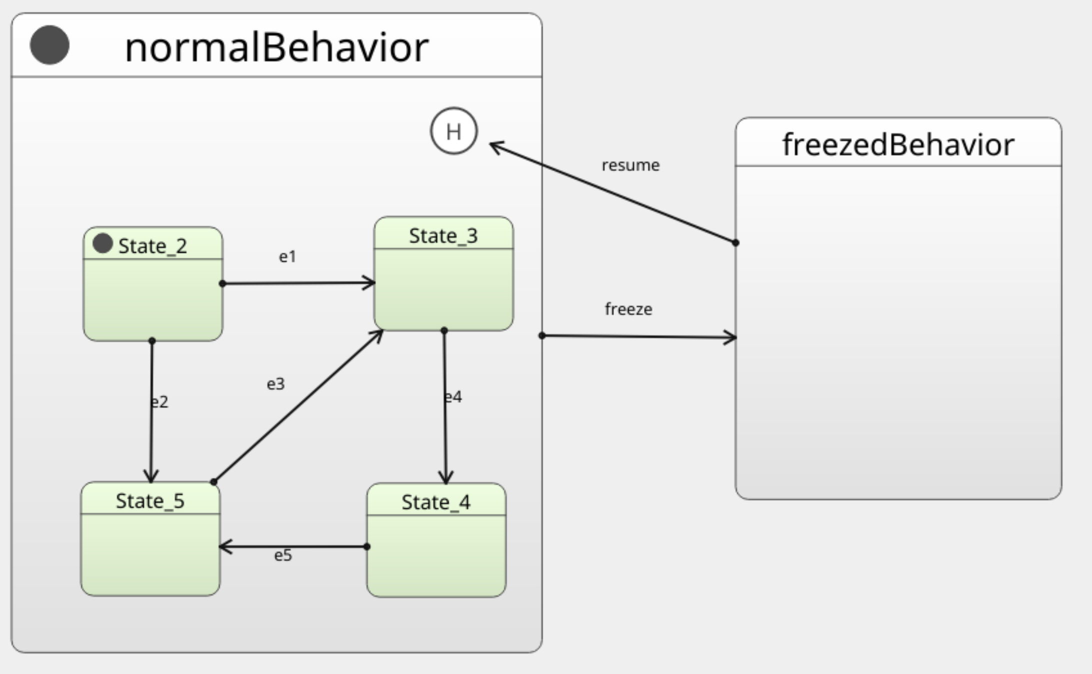
History state



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.).

History state

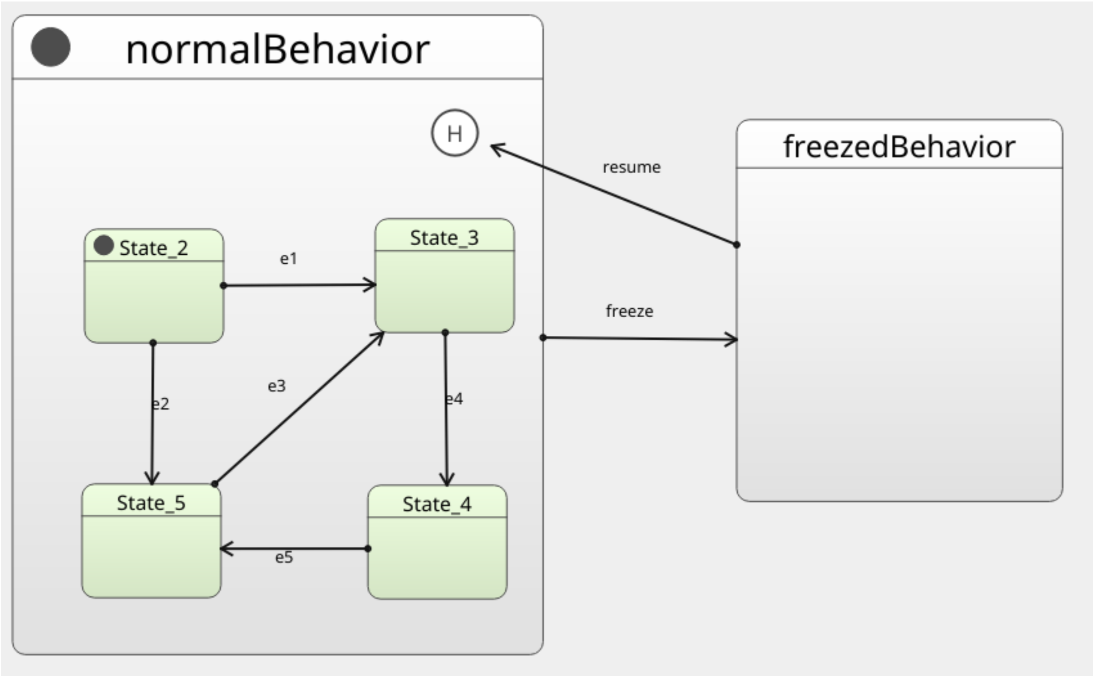


Deep or shallow...

```
h.start();
this->h.submitEvent("e1");
this->h.submitEvent("freeze");
this->h.submitEvent("resume");
```

- <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.).

History state



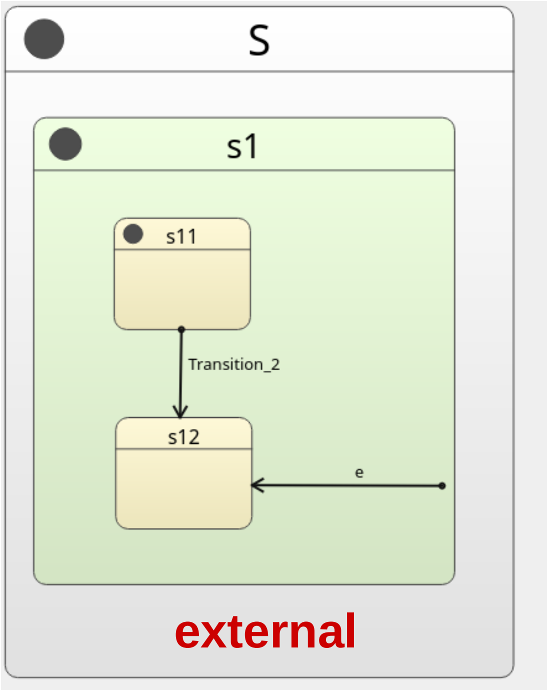

```
h.start();
this->h.submitEvent("e1");
this->h.submitEvent("freeze");
this->h.submitEvent("resume");
```

```
scxml.statemachine: “ ” : “enter normalBehavior”
scxml.statemachine: "" : "enter State_2"
scxml.statemachine: "" : "enter State_3"
scxml.statemachine: "" : "enter FreezedBehavior"
scxml.statemachine: "" : "enter State_3"
```

partial trace...

- <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.).

Type d’une transition



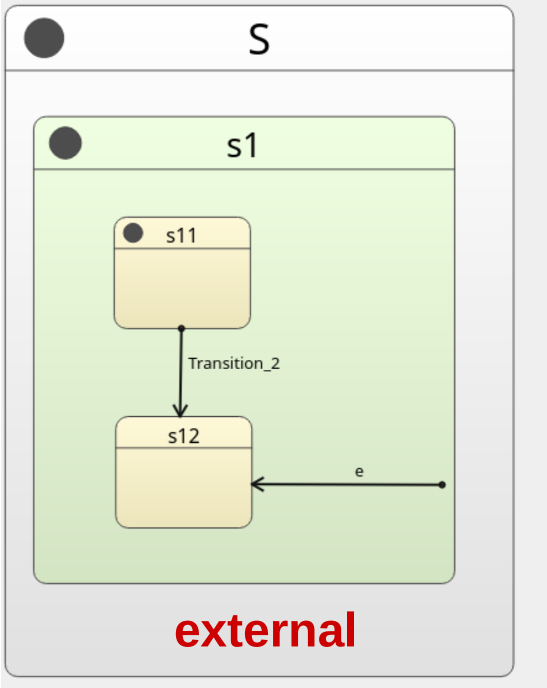

```
int_ext.start();
this->int_ext.submitEvent("e");
```

```
scxml.statemachine: "" : "entering S"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s11"
```

In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, [...]

Type d’une transition




```
int_ext.start();
this->int_ext.submitEvent("e");
```

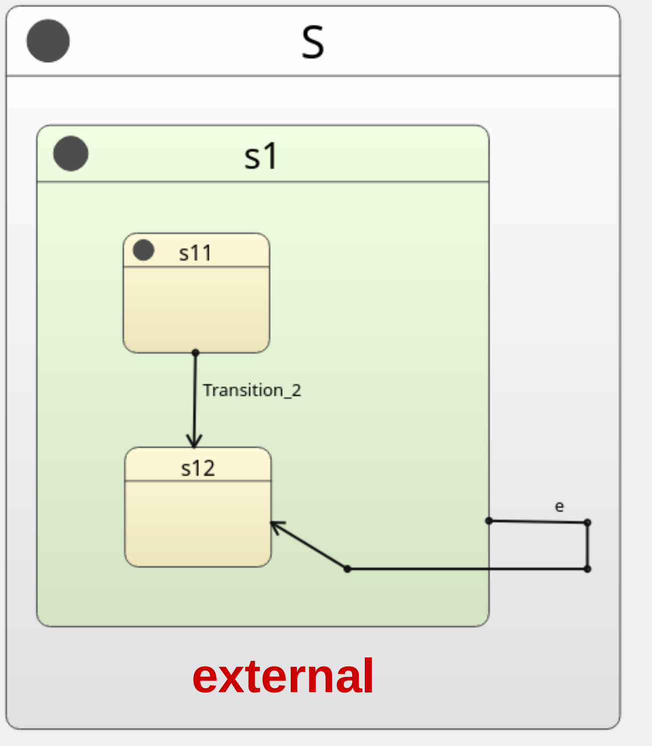
```
scxml.statemachine: "" : "entering S"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s11"
```

```
scxml.statemachine: "" : "leaving s11"
scxml.statemachine: "" : "leaving s1"
scxml.statemachine: "" : "executing transition"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s12"
```

In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, [...]

Type d'une transition




```
int_ext.start();
this->int_ext.submitEvent("e");
```

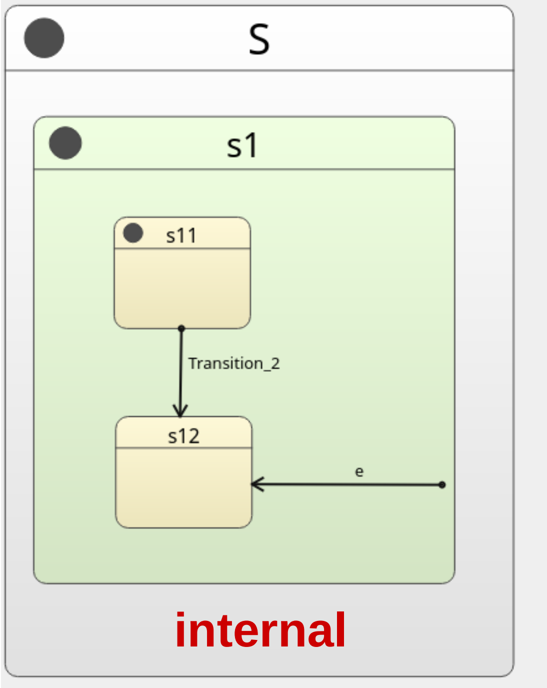
```
scxml.statemachine: "" : "entering S"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s11"
```

```
scxml.statemachine: "" : "leaving s11"
scxml.statemachine: "" : "leaving s1"
scxml.statemachine: "" : "executing transition"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s12"
```

In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, [...]

Type d’une transition



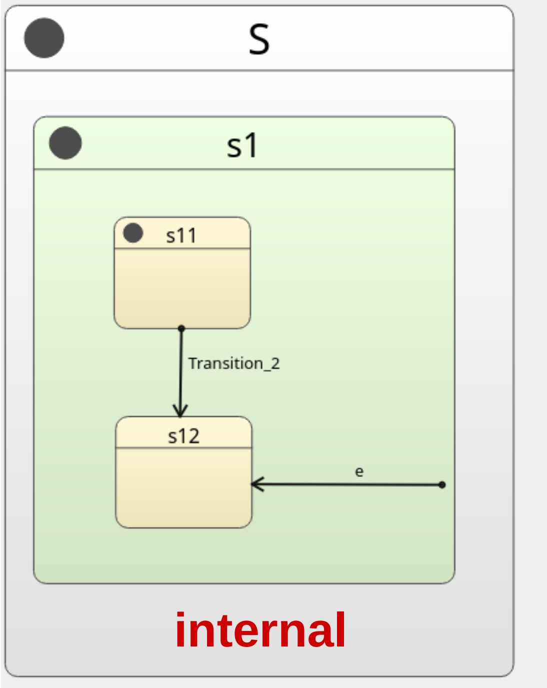

```
int_ext.start();
this->int_ext.submitEvent("e");
```

```
scxml.statemachine: "" : "entering S"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s11"
```

In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, [...]

Type d’une transition




```
int_ext.start();
this->int_ext.submitEvent("e");
```

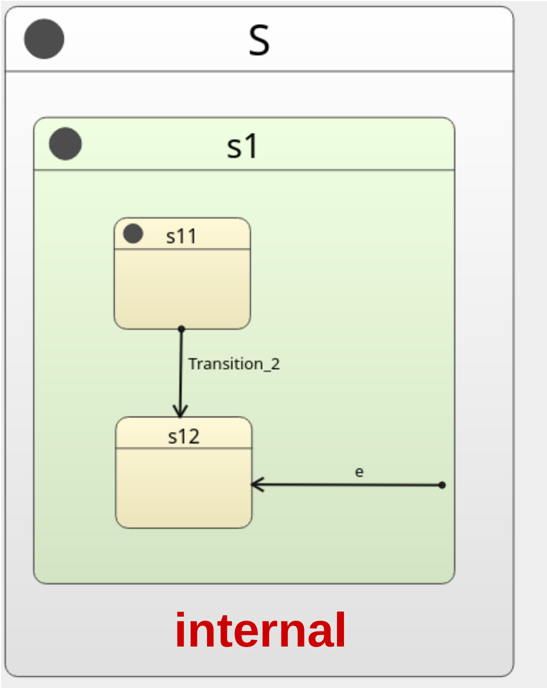
```
scxml.statemachine: "" : "entering S"
scxml.statemachine: "" : "entering s1"
scxml.statemachine: "" : "entering s11"
```

```
scxml.statemachine: "" : "leaving s11"
scxml.statemachine: "" : "executing transition"
scxml.statemachine: "" : "entering s12"
```

In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, [...]

Type d’une transition

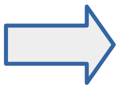


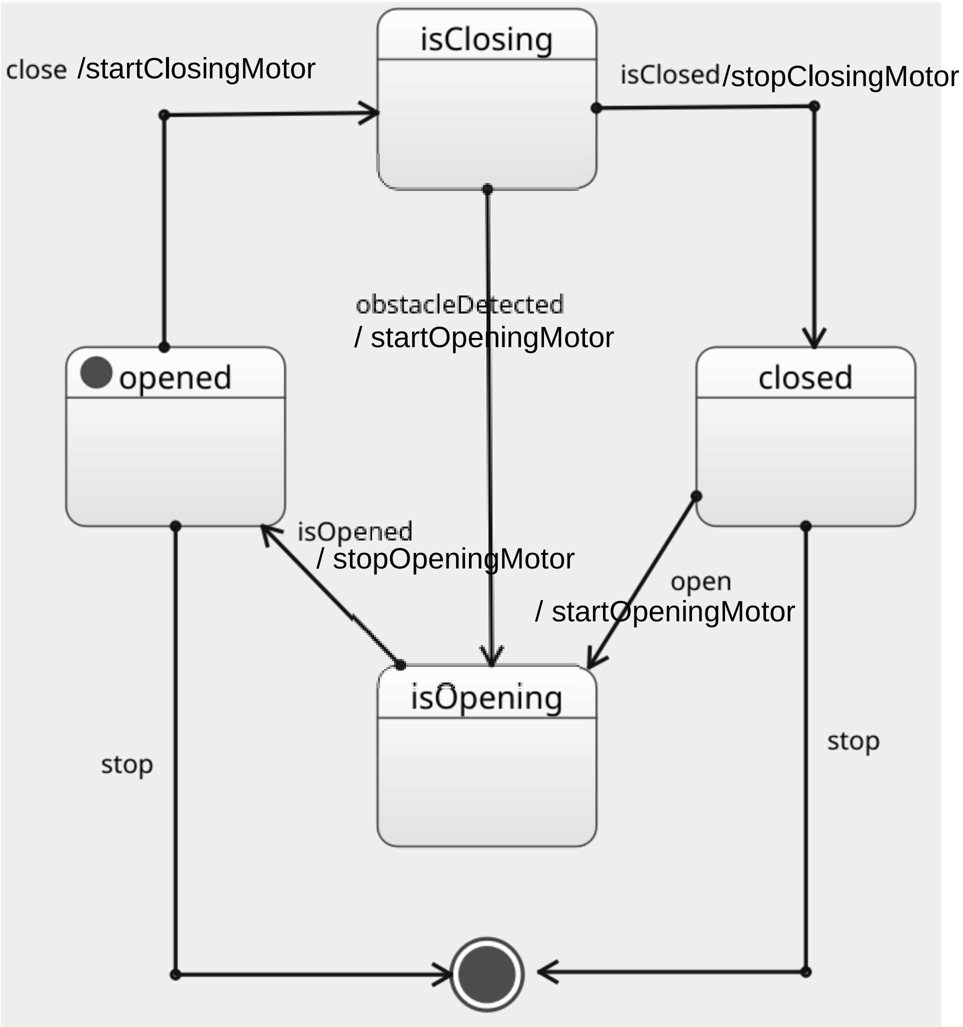
Such concept does not exist in Yakindu, even in the SCXML domain :’(

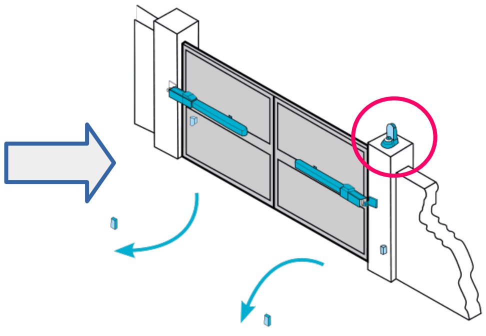
In the case of a transition located in a compound state, the 'type' attribute is significant. The behavior of a transition with 'type' of "external" (the default) is defined in terms of the transition's source state (which is the state that contains the transition), the transition's target state(or states), and the Least Common Compound Ancestor (LCCA) of the source and target states (which is the closest compound state that is an ancestor of all the source and target states). When a transition is taken, the state machine will exit all active states that are proper descendants of the LCCA, starting with the innermost one(s) and working up to the immediate descendant(s) of the LCCA. (A 'proper descendant' of a state is a child, or a child of a child, or a child of a child of a child, etc.) Then the state machine enters the target state(s), plus any states that are between it and the LCCA, starting with the outermost one (i.e., the immediate descendant of the LCCA) and working down to the target state(s). As states are exited, their <onexit> handlers are executed. Then the executable content in the transition is executed, followed by the <onentry> handlers of the states that are entered. If the target state(s) of the transition is not atomic, the state machine will enter their default initial states recursively until it reaches an atomic state(s).

The behavior of transitions with 'type' of "internal" is identical, except in the case of a transition whose source state is a compound state and whose target(s) is a descendant of the source. In such a case, an internal transition will not exit and re-enter its source state, while an external one will, [...]

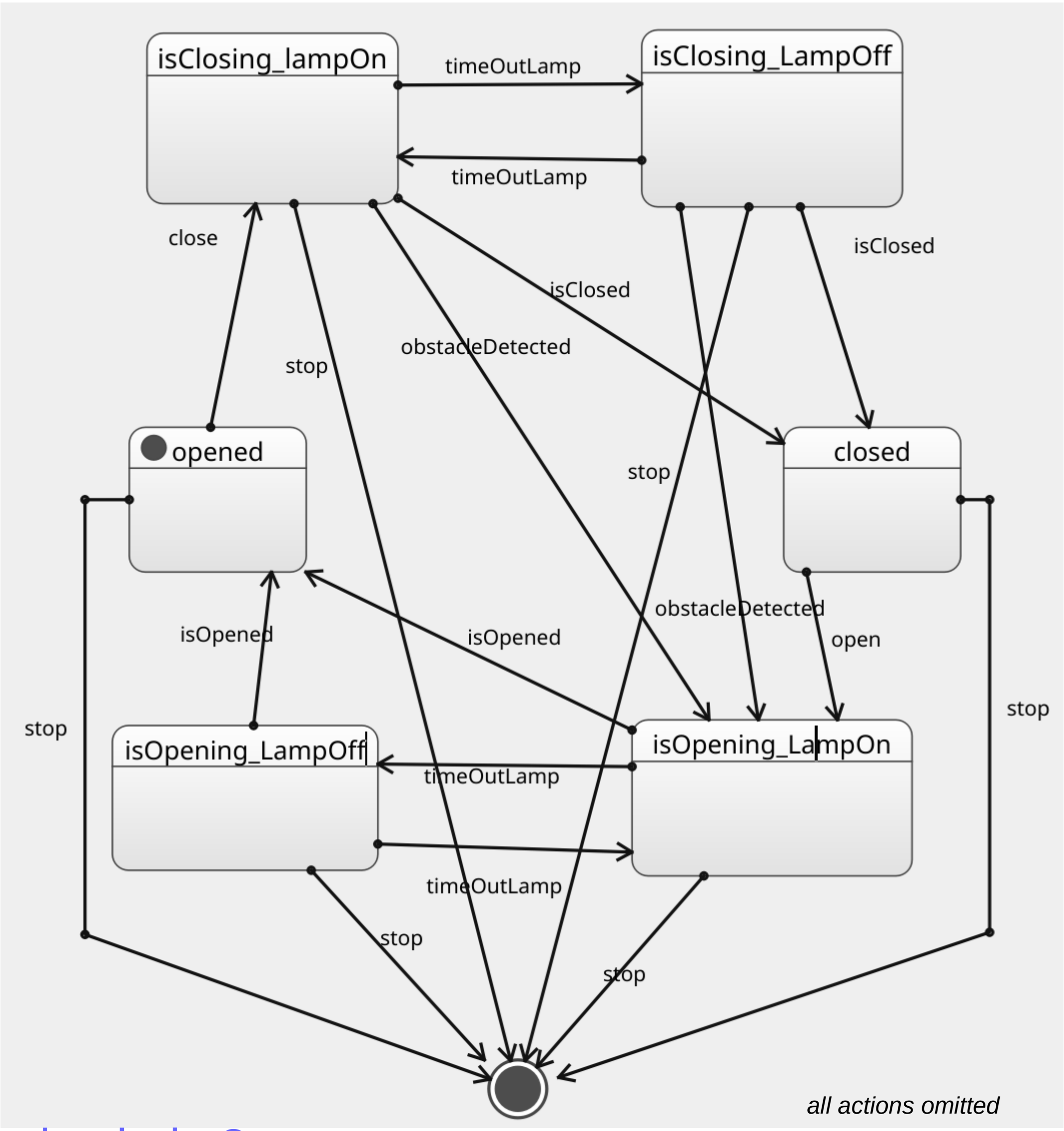
Running Example

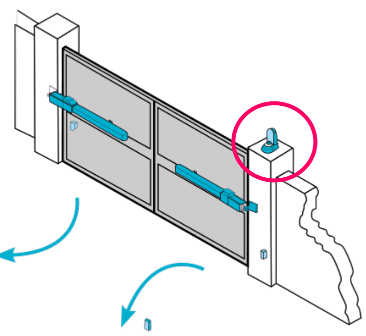






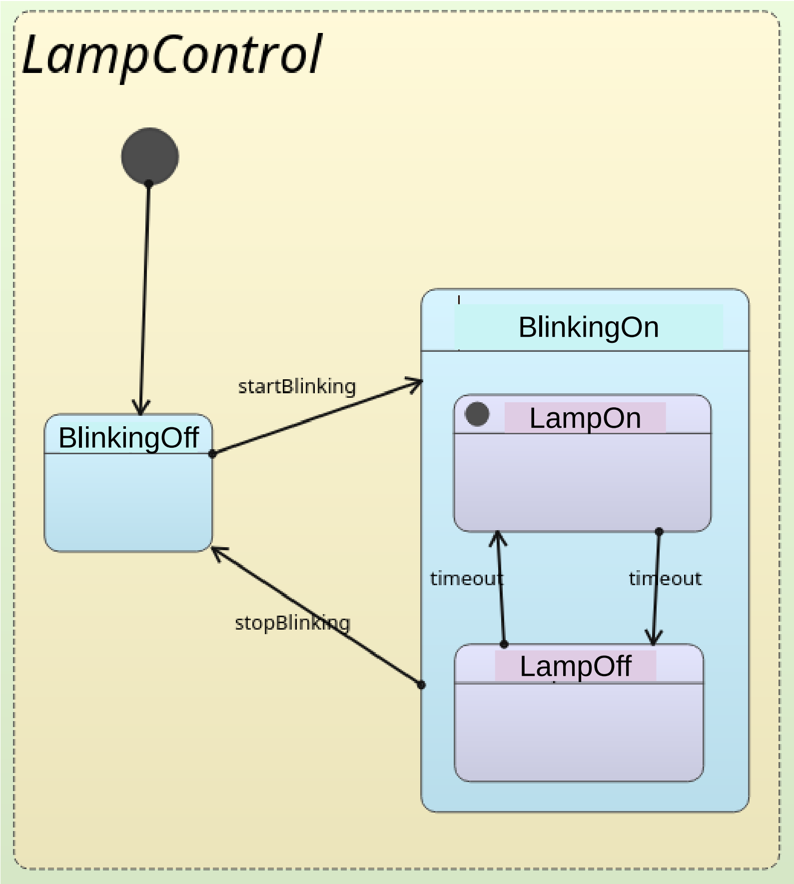
Running Example

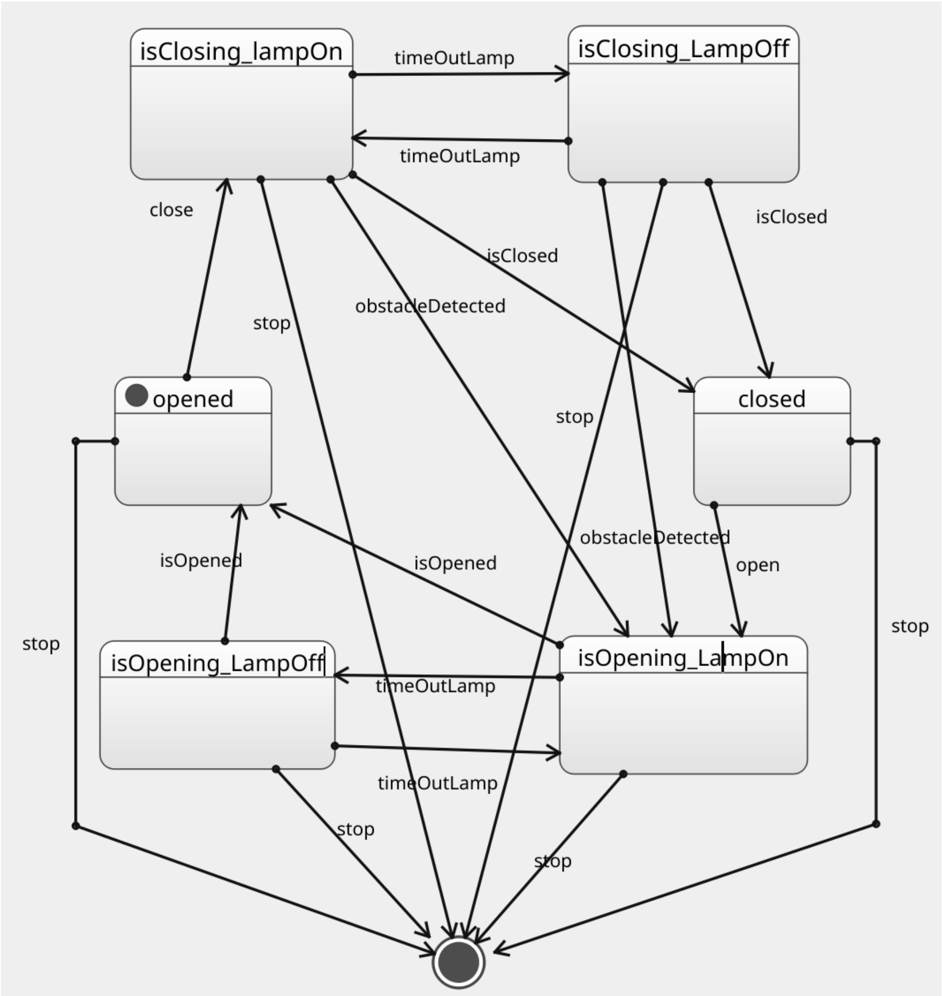




wasn't it supposed to help ?

Composite State





- 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**