

# UML-B State Machine Diagrams

May, 2010

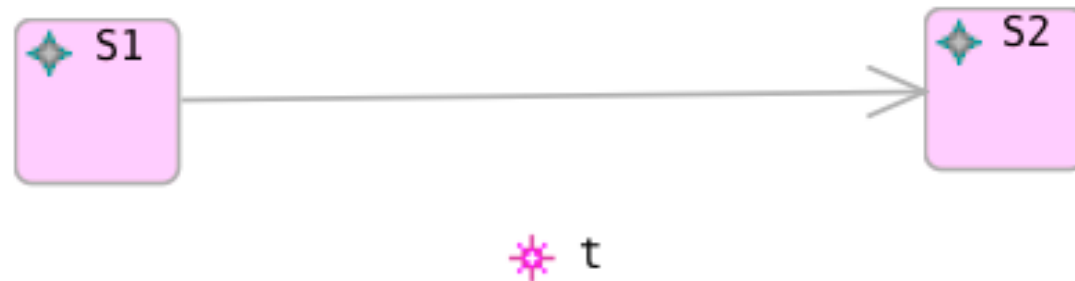
Colin Snook University of Southampton

# State Machines

State machines provide a way to model behaviour (transitions)

Constrained by some data (source state)

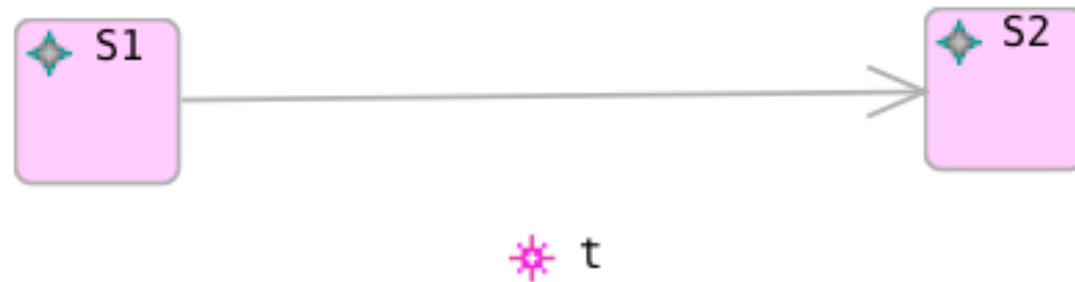
The transition's behaviour is to change the data (to target state)



Transition **t** can only fire when the state is **S1**  
when **t** fires it changes the state to **S2**

*How could we represent this in Event-B?*

# State Machines to Events



EVENTS

$t \hat{=} \text{WHEN } \langle \text{in } S1 \rangle \text{ THEN } \langle \text{becomes } S2 \rangle \text{ END}$

*where,  $\langle \text{in } S1 \rangle$  and  $\langle \text{becomes } S2 \rangle$  depend on the data that represents state*

# State machine as a type

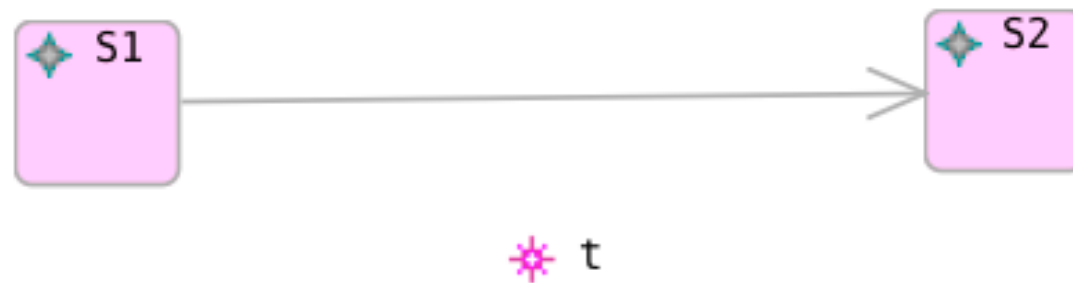
We could treat the whole statemachine as an enumerated type  
the current state is given by a variable  
(called **state\_function** translation in UML-B)

## VARIABLES

sm : sm\_STATES

## SETS

sm\_STATES = {S1,S2}



## EVENTS

$t \hat{=} \text{WHEN } \langle \text{in } S1 \rangle \text{ THEN } \langle \text{becomes } S2 \rangle \text{ END}$

*what are  $\langle \text{in } S1 \rangle$  and  $\langle \text{becomes } S2 \rangle$  in this case?*

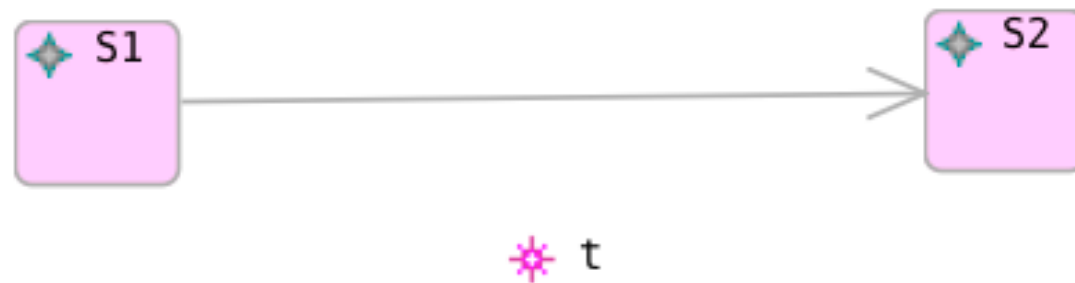
# State machine as a type

## VARIABLES

sm : sm\_STATES

## SETS

sm\_STATES = {S1,S2}



## EVENTS

t  $\hat{=}$  WHEN sm = S1 THEN sm := S2 END

# State machine collection of variables

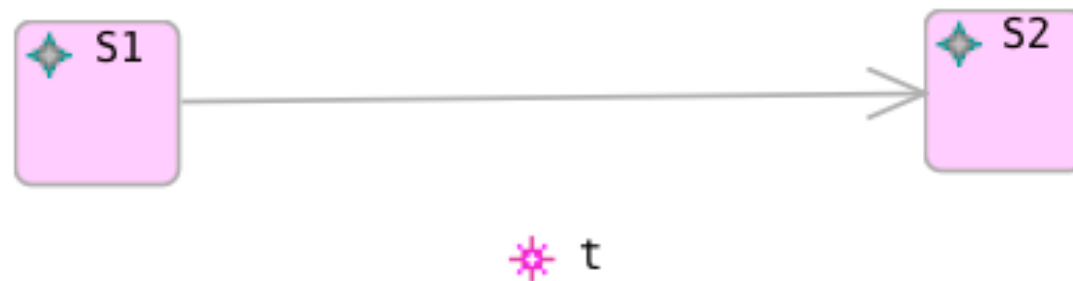
Or we could treat each state as a separate variable  
(called *state\_sets* translation in UML-B)

## VARIABLES

S1 : BOOL

S2 : BOOL

*where, one of S1, S2 is TRUE at any moment*



## EVENTS

$t \hat{=} \text{WHEN } \langle \text{in } S1 \rangle \text{ THEN } \langle \text{becomes } S2 \rangle \text{ END}$

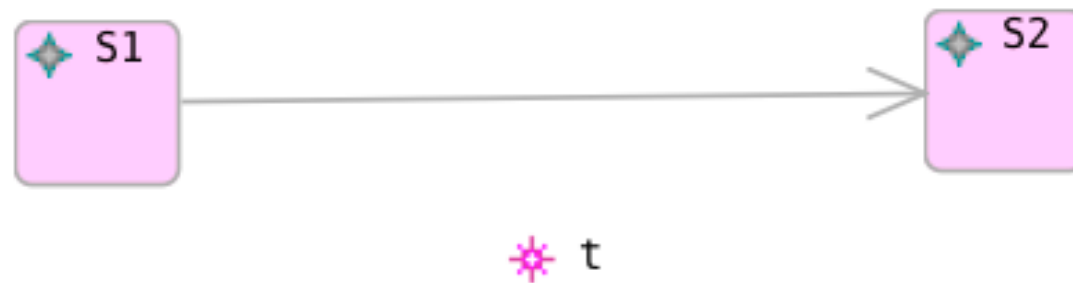
*what are  $\langle \text{in } S1 \rangle$  and  $\langle \text{becomes } S2 \rangle$  in this case?*

# State machine collection of variables

## VARIABLES

S1 : BOOL

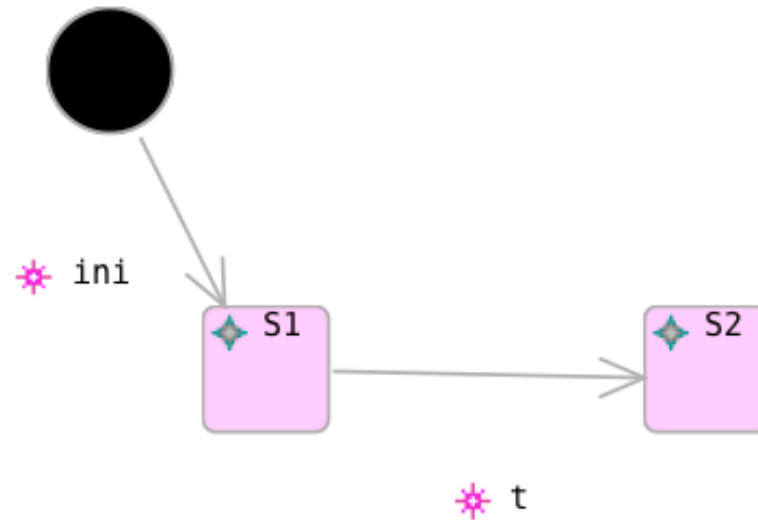
S2 : BOOL



## EVENTS

```
t ≐ WHEN  S1 = TRUE
        THEN S1 := FALSE
           S2 := TRUE
        END
```

# Initial transition



Statemachine as type

**INITIALISATION**

`sm := S1`

States as variables

**INITIALISATION**

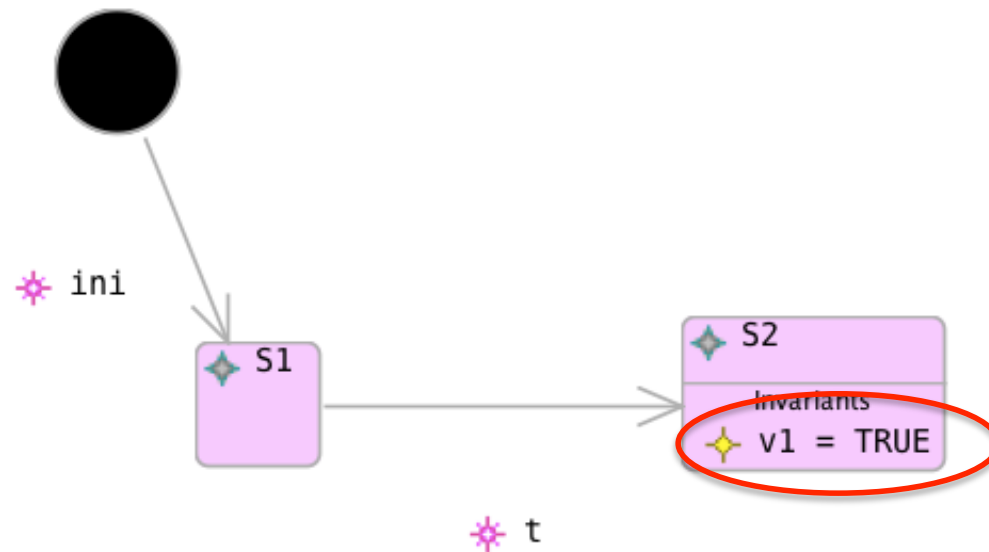
`S1 := TRUE`

`S2 := FALSE`



# State Invariant

Something that must be true whenever the system is in that state.



Translations:

$$(sm = S2) \Rightarrow (v1 = TRUE)$$

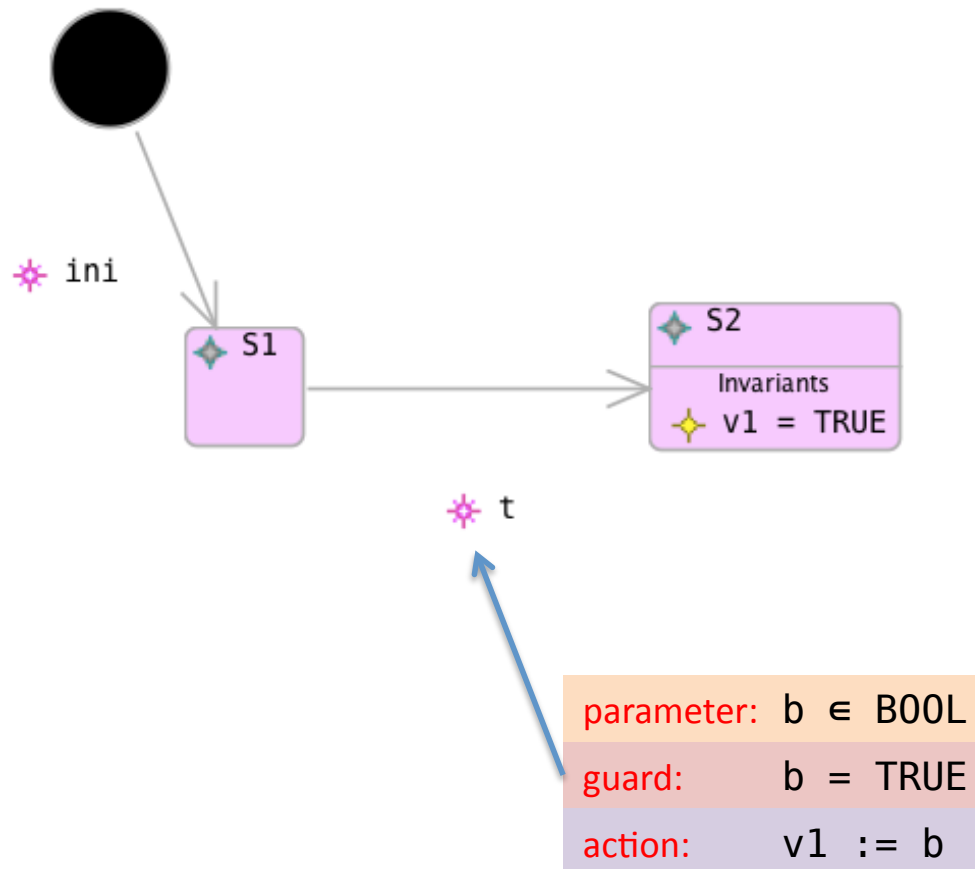
or

$$(S2=TRUE) \Rightarrow (v1 = TRUE)$$

# Transition parameters, guards and actions

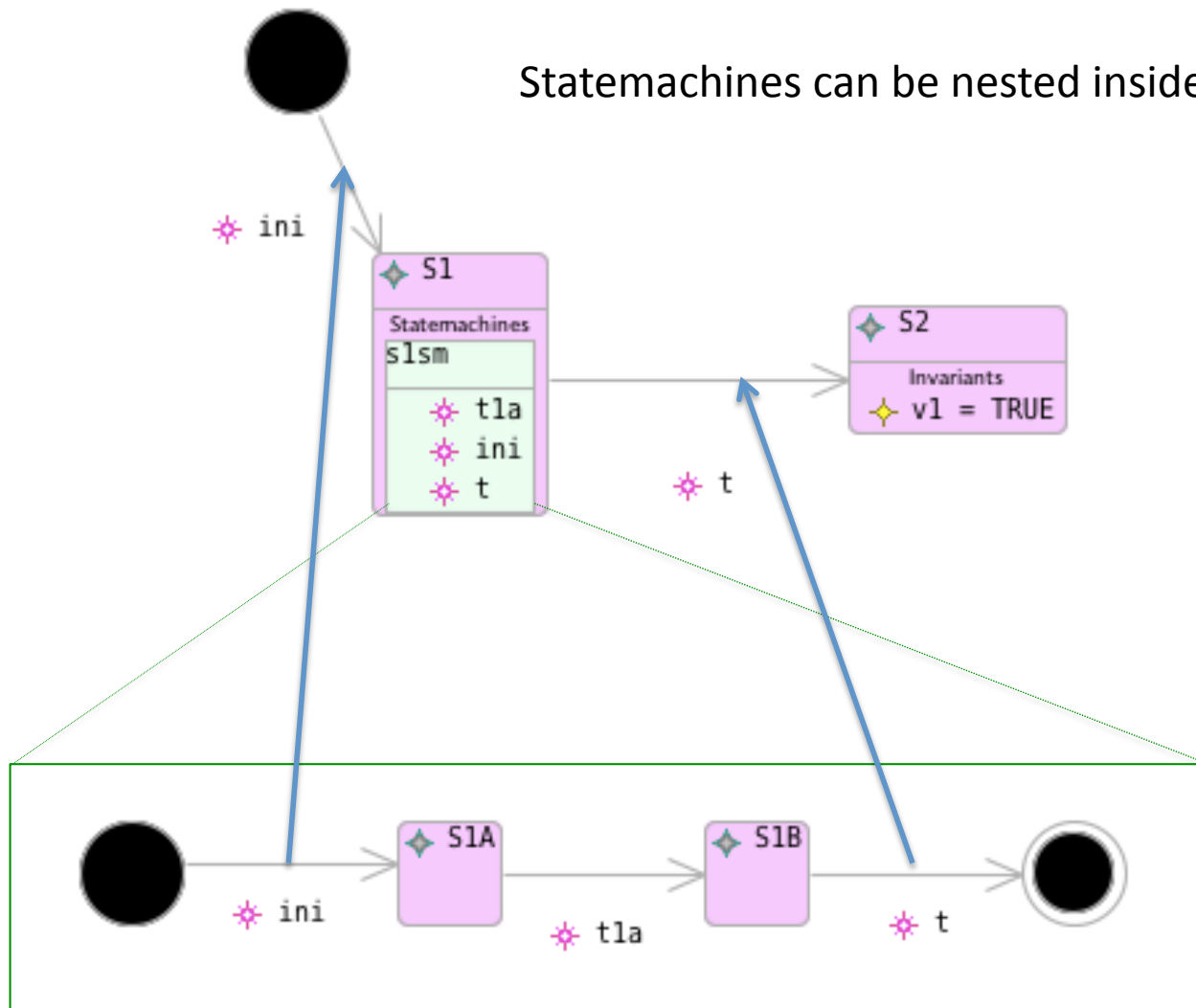
Transitions are events.

So you can give them parameters, guards and actions etc.



# Nested Statemachines

Statemachines can be nested inside states



Entry and Exit transitions must align with parent state using **Elaborates** property

# Example

A factory machine can be switched on and off.  
When it is on it can then be started and becomes active.  
When it is active it can run repeatedly until it is stopped.

A separately controlled guard mechanism can be opened and closed.  
[The guard is opened automatically when the machine is stopped]

Safety Requirement:

The machine should never be in the active state (where runs can occur) with the guard in the open position.

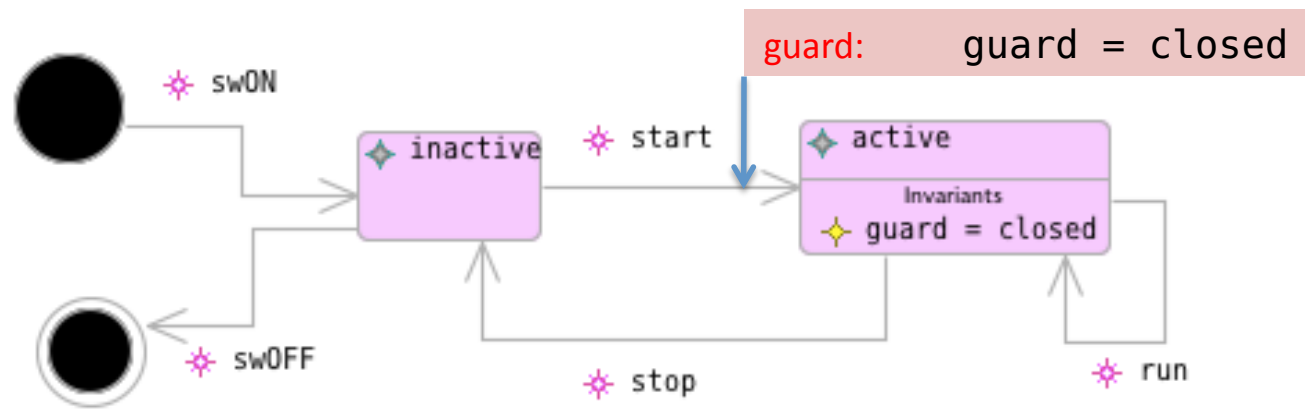
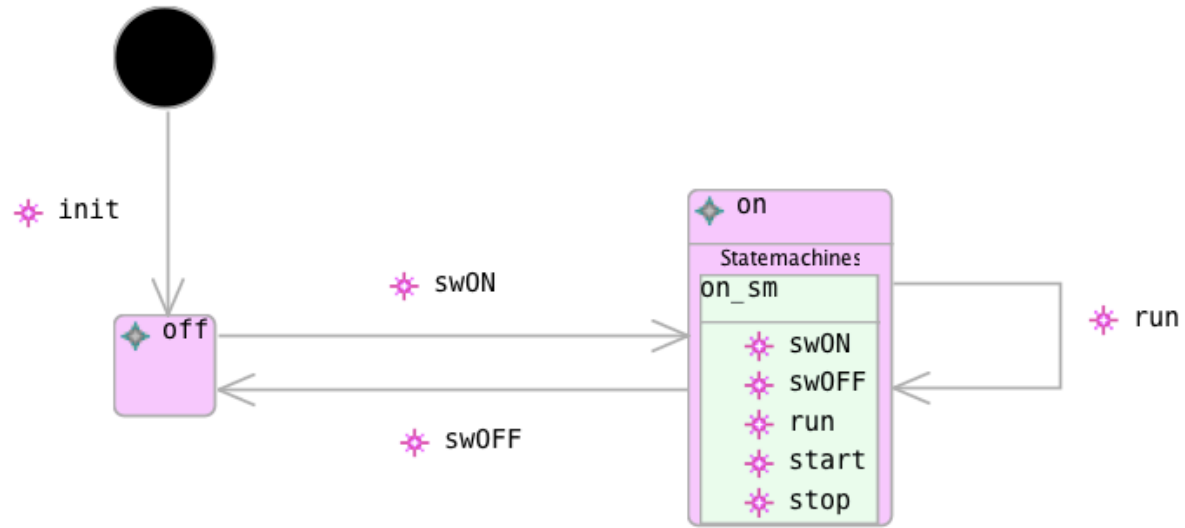
Model the machine and guard as separate statemachines.

Add an invariant to model the safety requirement.

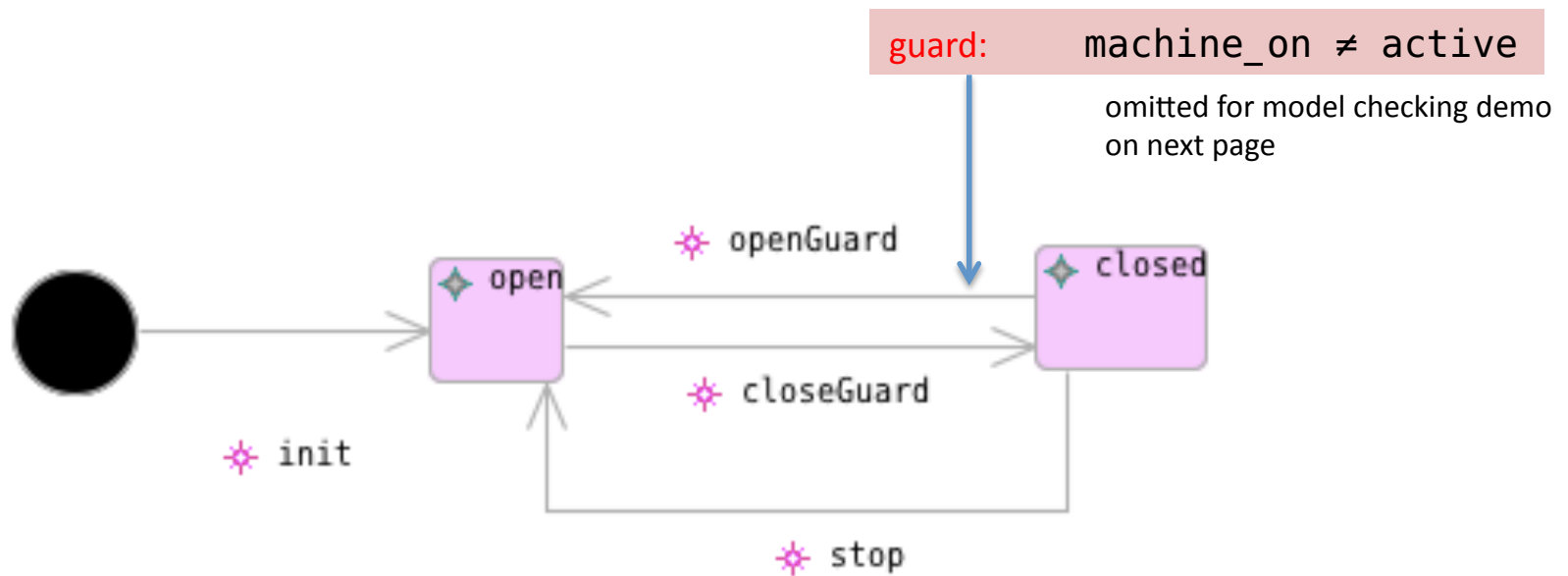
Determine the transitions guards needed to represent the interlocks between the machine and the guard controller.

Use the Pro-B model checker/Animator to ensure that the safety invariant is never violated

# Example – Factory Machine



# Example – Factory Machine Safety Guard



# Statemachine Animation showing invariant violation

The image displays three UML State Machine Diagrams (stateDiag.anim\_diag) and two panels: State and History.

**Diagram 1: C\_Statemachines.m1.machine.stateDiag.anim\_diag**  
 Shows a state machine with two states: **off** (initial state) and **on**.  
 Transitions:  
 - **off** to **on** on event **swON**.  
 - **on** to **off** on event **swOFF**.  
 - **on** to **on** (self-loop) on event **run**.  
 - **init** operation leads to the **off** state.

**Diagram 2: C\_Statemachines.m1.machine.on.machine\_on.stateDiag.anim\_diag**  
 Shows a state machine with two states: **inactive** and **active**.  
 Transitions:  
 - **inactive** to **active** on event **start**.  
 - **active** to **inactive** on event **stop**.  
 - **active** to **active** (self-loop) on event **run**.  
 - **swON** operation leads to the **inactive** state.  
 - **swOFF** operation leads to the **inactive** state.

**Diagram 3: C\_Statemachines.m1.guard.stateDiag.anim\_diag**  
 Shows a state machine with two states: **open** and **closed**.  
 Transitions:  
 - **open** to **closed** on event **closeGuard**.  
 - **closed** to **open** on event **openGuard**.  
 - **stop** operation leads to the **open** state.  
 - **init** operation leads to the **open** state.

**State Panel:**  
 Table showing the current state of the machine:

| Name       | Value  | Pre...alue |
|------------|--------|------------|
| ▼ m1       |        |            |
| guard      | open   | closed     |
| machine    | on     | on         |
| machine_on | active | active     |

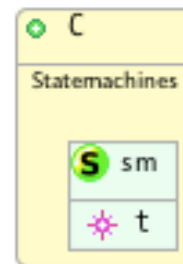
**History Panel:**  
 Operations performed:  
 openGuard  
 start  
 closeGuard  
 swON  
 INITIALISATION(FALSE,TRUE,TRUE,FALSE,  
 (root)

**Footer:**  
 invariant violated! no event errors detect

# Statemachines in Classes

Statemachines can be added to classes.

Effectively, each class instance has a “copy” of the statemachine





# State machines in Classes

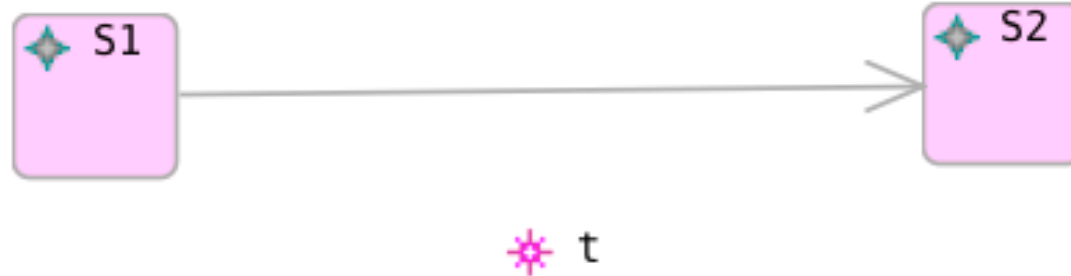
## State machine as a type (state\_function)

VARIABLES

$sm \in C \rightarrow sm\_STATES$

SETS

$sm\_STATES = \{S1, S2\}$



EVENTS

```
t (self)  $\hat{=}$  WHERE sm(self) = S1
      THEN sm(self) := S2
      END
```

# State machines in Classes

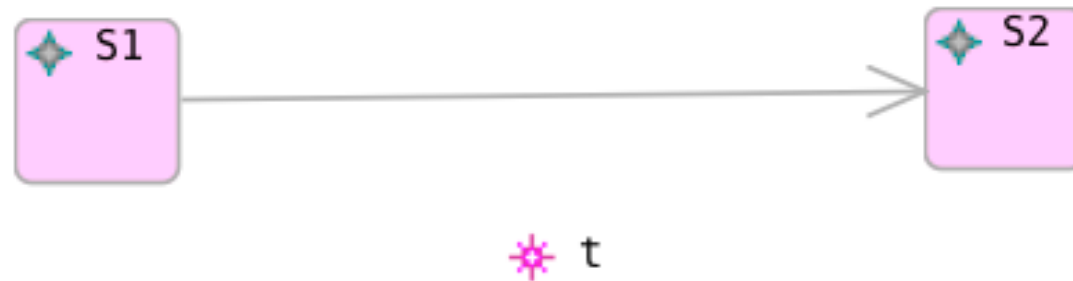
States as variables (state\_sets)

## VARIABLES

$S1 \in \mathcal{P}(C)$

$S2 \in \mathcal{P}(C)$

where  $S1$  and  $S2$  (and....) are *disjoint*



## EVENTS

$t(\text{self}) \triangleq$  WHERE  $\text{self} \in S1$   
THEN  $S1 := S1 \setminus \{\text{self}\}$   
 $S2 := S2 \cup \{\text{self}\}$   
END

## Initial transition (state\_sets)

For variable instance classes, initial transition is treated as a constructor

```
ini ≐  
STATUS  
  ordinary  
ANY  
  self // constructed instance of class C  
WHERE  
  self.type : self ∈ C_SET \ C  
THEN  
  C_constructor : C := C ∪ {self}  
  sm_enterState_S1 : S1 := S1 ∪ {self}  
END
```

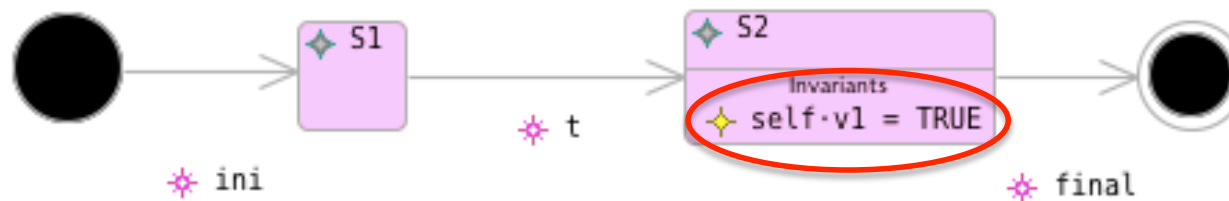
## Final transition (state\_sets)

For variable instance classes, final transition is treated as a destructor

```
final ≐  
STATUS  
  ordinary  
ANY  
  self // contextual instance of class C  
WHERE  
  self.type : self ∈ C  
  sm_isin_S2 : self ∈ S2  
THEN  
  sm_leaveState_S2 : S2 = S2 \ {self}  
  C_destructor : C = C \ {self}  
END
```

# State Invariant (state\_sets)

Something that must be true whenever an instance of the class is in that state.



Translation:

$$\forall \text{self} \cdot ((\text{self} \in C) \Rightarrow ((\text{self} \in S2) \Rightarrow (v1(\text{self}) = \text{TRUE})))$$

# Summary

## Statemachines for modelling behaviour

- ▶ nested – statemachines in states
- ▶ invariants in states
- ▶ transitions are events (with parameters, guards, actions)

## Choice of 2 translations

## Can be lifted to classes

- ▶ initial/final transition as constructor/destructor (variable instance classes)

## State-machines can be animated and model checked

- ▶ (front-end for Pro-B)