Il nostro progetto è implementato con il Linguaggio LUA. Abbiamo creato una Libreria per la definizione e l'utilizzo di *macchine a stati finiti* (i cui elementi fondamentali sono Stati, Eventi, Transizioni).
Permette di testare *macchine a stati finiti (FSM)* per capirne il loro comportamento e sapere quando si avranno dei cambiamenti di stato o loop.
Diamo modo di procedere con la simulazione sia settando di volta in volta i valori dei sensori e sia cambiandoli random.
Lua è un linguaggio multiparadigma (scripting, imperativo, funzionale); le caratteristiche principali sono:

- funzioni→ viste come pseudo oggetti e quindi assegnabili a variabili;
- tabelle→ utilizzate come strutture dati generiche.

Queste, permettono al linguaggio di implementare funzionalità viste nei linguaggi orientati agli oggetti, pur non essendo tale.
D'ora in poi, quando utilizzeremo il termine "classe" o "oggetto" sappiamo che non ci riferiamo agli oggetti veri e propri che vengono utilizzati nella Programmazione OO.

Una MACCHINA A STATI FINITI o AUTOMA A STATI FINITI è un modello computazionale che permette di descrivere con precisione ed in modo formale il comportamento di molti sistemi.
Possono essere usate per modellare problemi in diversi campi, come la matematica, l'intelligenza artificiale, videogiochi, linguistica.

Ve ne sono di due tipi:

- DETERMINISTICA;
- NON DETERMINISTICA.

MACCHINA A STATI DETERMINISTICA

automa a stati finiti dove, per ogni coppia di stato e simbolo in ingresso, c'è una ed una sola transizione allo stato successivo.

È descritta da una quintupla $(\Sigma, S, \delta, s_0, F)$:

- $\Sigma = \{a_0, a_1, \ldots, a_n\}$ insieme finito di simboli, chiamato alfabeto;
- $S = \{s_0, s_1, \ldots, s_n\}$ insieme finito di stati;

- δ: S x Σ → P(S) funzione di transizione;
- $s_0 \in S$ stato di partenza;
- $F \subseteq S$ insieme degli stati finali.

MACCHINA A STATI NON DETERMINISTICA

automa a stati finiti dove, per ogni coppia di stato e simbolo in input, possono esserci più stati di destinazione.

È descritta da una quintupla (Σ, S, δ, s_0, F):

- $\Sigma = \{a_0, a_1, \ldots, a_n\}$ insieme finito di simboli, chiamato alfabeto;
- $S = \{s_0, s_1, \ldots, s_n\}$ insieme finito di stati;
- δ: S x Σ → P(S) funzione di transizione in cui P(S) è l'insieme delle parti di S;
- $s_0 \in S$ stato di partenza;
- $F \subseteq S$ insieme degli stati finali.

Per poter rappresentare il concetto di *Macchina a Stati finiti*, abbiamo utilizzato il formalismo grafico del DIAGRAMMA DEGLI STATI fornito dall'UML.
Il diagramma degli stati serve *per:*

- *rappresentare il comportamento di un artefatto*, composto da un numero finito di stati;
- *descrivere il comportamento di un elemento di un modello* e *possibili sequenze di stati e le azioni* attraverso cui l'elemento può procedere durante la sua vita, come risultato di reazioni ad eventi discreti.

Gli *elementi fondamentali dei Diagrammi degli Stati*, sono:

- *Stato* → è una situazione nella vita di un oggetto, in cui esso soddisfa una condizione, esegue un'attività o è in attesa di un qualche evento;

- *Transizione* → è una relazione che lega uno stato di partenza ad uno stato d'arrivo. Descrive come l'entità modellata "reagisce" ad eventi generati dall'esterno o da se stesso.

La sintassi dello STATO consiste in un rettangolo con angoli arrotondati, un nome, un elenco opzionale di operazioni da svolgere al suo interno.
Le operazioni sono:

- *Entry*: azione eseguita ogni volta che si entra nello stato;
- *Do*: azione associata allo stato, può prevedere un lasso di tempo considerevole e può essere interrotta da un evento;
- *Exit*: azione eseguita ogni volta che si lascia lo stato.

Nel diagramma degli Stati, sono presenti due particolari Stati:

- Stato Iniziale→ rappresentato da un disco nero, che marca l'inizio dell'esecuzione;
- Stato Finale→ rappresentato da un disco nero bordato, che indica che l'esecuzione è terminata.

La sintassi della TRANSIZIONE consiste di una freccia che collega due stati ed è etichettata con tre elementi (Evento [Condizione]/ Azione → ECA):

- *Evento* → è un avvenimento che può innescare una transizione tra stati;
- *Condizione* → è un'espressione booleana che deve essere soddisfatta affinché la transizione possa essere eseguita (se da uno stato escono più transizioni, con medesimo evento, le loro condizioni devono essere mutuamente esclusive, per poter implementare una macchina a STATI DETERMINISTICA);
- *Azione* → è associata alla transizione ed è considerata un processo rapido, non interrompibile da un evento.

==========

Abbiamo implementato un modulo *StateMachineSimulator* in cui vi sono 3 concetti di classe, in maniera procedurale:

- *State* per definire il concetto di Stato;

- *Sensor* per definire il concetto di Sensore: dispositivi hardware che raccolgono dati dall'ambiente del sistema;
- *Actuator* per definire il concetto di Attuatore: dispositivi hardware che modificano l'ambiente del sistema;

le funzioni di utilità, la funzione run che è quella fondamentale per l'esecuzione della macchina a stati, la funzione simulator che ci permette di effettuare delle simulazioni con valori di sensori random.

Un modulo *IsEqual* che viene richiamato come libreria; in esso viene implementata una funzione che ci permette di comparare le possibili configurazioni della macchina, inserite in una tabella (configurations).

STATE MACHINE SIMULATOR

Richiama il modulo "IsEqual" e crea una tabella, "configurations" (conterrà le configurazioni cioè delle triple composte da Stato, Sensori, Attuatori, che mostrano la condizione della FSM in un determinato momento), una variabile "currentState" che verrà aggiornata ogni qual volta si cambia di stato.

STATE

Abbiamo dichiarato una tabella "State".
Una tabella "states" che conterrà tutti gli stati della FSM.
La funzione "new", per creare un nuovo oggetto Stato con campi *name* (nome stato), *entry* (azioni di ingresso nello stato), *doo* (azioni interne allo stato, volontariamente con due "o" per distinguerlo dalla parola chiave, in LUA, "do"), *exitt* (azioni di uscita dallo stato, volontariamente con due "t" per distinguerlo dalla funzione standard di LUA "exit()"), *transitions* (tabella che conterrà tutte le possibili transizioni a partire dallo stato in questione), *counter* (valore numerico che ci indicherà, durante la fase di test, il conteggio delle volte in cui lo stato è stato attraversato).
La funzione "add" che inserisce lo stato nella tabella "states".
La funzione "setCurrentState" che ci permette di aggiornare il valore dello stato corrente durante la simulazione.
La funzione "addTransition" che prende una transizione e quindi gli eventi, la condizione, le azioni e lo stato successivo per inserirla nella tabella delle transizioni dello stato in questione.

Tre funzioni che permettono di settare l'entry, doo ed exitt dello stato e tre funzioni che ci restituiscono se tali campi sono vuoti o meno.
Funzione "executeEde" che esegue le funzioni, eventualmente associate in input, ai campi entry, doo, exitt.
Funzione "printStateCounter" che stampa nel file di log delle stringhe con i nomi degli stati della FSM ed i relativi counter.
Funzione "printTransitions" che stampa nel log file una stringa contenente l'ECA della transizione di uno stato.

SENSOR

Abbiamo dichiarato una tabella "Sensor".
Una tabella "sensors" che conterrà tutti i sensori della FSM.
La funzione "new", per creare un nuovo oggetto Sensor con campi *name* (nome sensore), *typee* (numerico o booleano), *val* (valore che assume il sensore), *min* (valore minimo che può assumere), *max* (valore massimo che può assumere), *change* (booleano che indica se il valore del sensore è cambiato ed aiuta a comprendere se il sensore può causare una transizione), *counterVal* (contatore di quante volte è cambiato il valore del sensore), *counterCha* (contatore di quante volte il cambio del valore sensore ha causato transizioni).
La funzione "add" che inserisce il sensore nella tabella "sensors".
La funzione "setValue" che verifica se esiste ed è aperto il log file (abbiamo implementato il Simulatore che produce un log file solo nel caso in cui utilizziamo la funzione simulator per simulazioni random), va a settare il nuovo valore del sensore e scrive nel log file, qualora è richiesto, delle stringhe con il nome dello stato iniziale da cui parte la simulazione, il sensore modificato ed il valore di questo.
Setta anche il campo change a true qualora il valore dato è diverso da quello precedente. Fa partire la funzione run e restituisce in output il nome dello stato corrente.
Funzione "getValue" che restituisce il valore del sensore.
Funzione "randomSensorValue" che prende due valori numerici (range), randomicamente sceglie dalla tabella dei sensori, un sensore e se questo è booleano assegna random true o false, mentre se è numerico assegna random un valore compreso nel range.
Function "printSensorCounters" che stampa nel file di log la stringa con i nomi dei sensori della tabella ed i relativi contatori.

ACTUATOR

Abbiamo dichiarato una tabella "Actuator".
Una tabella "actuators" che conterrà tutti gli attuatori della FSM.
La funzione "new", per creare un nuovo oggetto Actuator con campi *name* (nome attuatore), *typee* (numerico o booleano), *val* (valore che assume l'attuatore), *min* (valore minimo che può assumere), *max* (valore massimo che può assumere), *change* (booleano che indica se il valore dell'attuatore è cambiato), *counterVal* (contatore di quante volte è cambiato il valore dell'attuatore), *counterCha* (contatore di quante volte il cambio del valore attuatore ha causato transizioni).
La funzione "add" che inserisce l'attuatore nella tabella "actuators".
La funzione "setValue" che setta il campo "val" con il valore immesso. Setta anche il campo change a true qualora il valore dato è diverso da quello precedente. Restituisce in output il nome dello stato corrente.
Function "printActuatorsCounters" che stampa nel file di log la stringa con i nomi degli attuatori della tabella ed i relativi contatori.
Function "executeActions" che esegue le funzioni, eventualmente associate in input, al campo "actions" di una transizione.

Funzioni utilità:

- "countEvents" che conta i sensori inseriti negli eventi di una transizione che hanno il campo "change" uguale a true;
- "setChangeFalse" setta i campi "change" dei sensori/attuatori coinvolti a false, nel momento in cui la transizione è stata effettuata;
- "incrementCounterCha" incrementa il campo "counterCha" dei sensori/attuatori nel momento in cui questi hanno causato una transizione;
- "resetStateMachine" resetta i campi degli Stati, Sensori ed Attuatori della FSM ai valori di default ed elimina il file di log precedentemente creato.
- "setConfiguration" inserisce nella tabella "configurations" la configurazione data in input;
- "resetConfiguration" al termine di ogni simulazione, svuota la tabella delle configurazioni;

- "checkConfiguration" confronta le varie configurazioni (usando isEqual) e se ne trova due uguali restituisce true (caso di loop) e false altrimenti.

Funzione "run", dichiara una variabile locale "l", inizializzata a true, serve ad interrompere il ciclo for successivo ad avvenuta transizione (evita il non determinismo ed effettua un'unica transizione legata allo stato corrente). Il ciclo for, scorre la tabella delle transizioni di uno stato, controlla se "l" è uguale a true, fin quando trova la transizione che ha tra gli eventi tutti i sensori con campo "change" uguale a true, controlla se la transizione per essere eseguita deve soddisfare una condizione ed in tal caso verifica che sia avvenuta; controlla se lo stato corrente ha una funzione di exit ed in tal caso, la esegue, setta a false i campi "change" degli eventi della transizione e se questa ha delle azioni le esegue. Stampa nel log file, se esiste, una stringa contenente l'ECA della transizione, aggiorna lo stato corrente, aumenta il contatore di questo e se il log file è aperto, stampa lo stato di arrivo della transizione; incrementa i contatori dei sensori che hanno generato la transizione, salva la configurazione attuale e se nella tabella delle configurazioni è presente una uguale a questa ferma l'esecuzione, notificando la presenza di un loop e lo stampa nel log file qualora questo esiste, mentre se non vi è un loop, setta "l" a false (per impedire l'esecuzione di eventuali transizioni a partire da un medesimo stato), richiama se stessa. Esegue, se presenti, le funzioni di entry e doo dello stato corrente. Procede per la medesima esecuzione anche se la condizione non esiste.
Al termine, restituisce il nome dello stato corrente.

Funzione "simulator" prende in input il numero di volte che si vuole simulare ed un'eventuale tabella "simTable" (perché si può optare per l'inserimento del valore "nil" e simulare, quindi, con tutti i sensori presenti) contenente un sottoinsieme di sensori con cui si vuole effettuare la simulazione e due numeri che rappresentano il minimo ed il massimo del range di valori che il simulatore può assegnare ai sensori. Apre il file di log, per il numero di simulazioni scelte controlla se vi è un loop; stampa nel file di log i dati statistici riferiti ai Sensori, Stati ed Attuatori e chiude tale file, se non vi è, resetta la tabella delle configurazioni. Scrive nel file di log il numero della simulazione corrente e controlla se vi è come argomento il sottoinsieme dei sensori ed in tal caso esegue la funzione "randomSensorValue2" che modifica solo i sensori indicati, altrimenti esegue la funzione

"randomSensorValue", che modifica tutti i sensori della Macchina a Stati definita. Al termine delle simulazioni, scrive i dati statistici di Stati, Sensori ed Attuatori, chiude il file di log e restituisce lo stato corrente.

==========

Abbiamo implementato due esempi: Alarm e Lamp.

Alarm è una macchina a stati che rappresenta il funzionamento di un *sistema di allarme domestico*. Prevede l'utilizzo di una tastiera numerica per l'inserimento della password (per attivare o disattivare il sistema di allarme); è dotato di un sensore di movimento, che rileva la presenza di un qualcosa che si muove nella stanza, ed un sensore per finestra, che rileva l'apertura di quest'ultima.
Quando almeno uno dei due sensori si attiva, il sistema passerà nello stato di allarme con simultanea attivazione della sirena.

Alarm è composto da tre STATI:

- *sleep*→ è la condizione di "standby", non vi è quindi alcun monitoraggio della stanza;
- *arm*→ è la condizione in cui il sistema di allarme è attivo ed i sensori sono in fase di rilevamento;
- *alarm*→ è la condizione in cui uno dei due sensori o entrambi hanno rilevato un'intrusione.

In LUA abbiamo usato questa sintassi, per creare uno STATO:

```
sleep= State.new("sleep")
sleep: add()
```

nella prima riga, si crea l'oggetto "sleep" di tipo "State" a cui gli si assegna un nome; i campi "entry", "doo" ed "exitt" vengono settati a nil, al campo "transitions" viene assegnata una tabella vuota ed al campo "counter" il valore zero.
Nella seconda riga, si aggiunge alla tabella degli Stati ("states") tale oggetto.

Di seguito abbiamo creato altri due stati "arm" ed "alarm".

```
arm= State.new("arm")
arm: add()

alarm= State.new("alarm")
alarm: add()
```

Alarm ha quattro SENSORI:

- *key*→ rappresenta la password memorizzata;
- *insKey*→ rappresenta l'inserimento sul tastierino numerico di una password;
- *move*→ rileva il movimento nella stanza;
- *window*→ rileva la finestra aperta.

In LUA abbiamo usato questa sintassi, per creare un SENSORE:

```
key= Sensor.new("key", "number", 1234, 1234, 1234)
key: add()
```

nella prima riga, si crea l'oggetto "key" di tipo Sensore, con nome "key", tipo "number", valore di default 1234 e gli si assegna un range di possibili valori (in tal caso, sono identici al valore di default per asserire il valore costante della password); il campo "change" inizializzato a false ed i campi "counterVal" e "counterCha" settati a zero.
Nella seconda riga, si aggiunge alla tabella dei Sensori ("sensors") tale oggetto.

```
insKey= Sensor.new("insKey", "number", 1233, 1233, 1236)
insKey: add()
```

per la creazione di "insKey", il range di valori non è identico al valore di default, per poter simulare i tentativi di inserimento della password.

```
move= Sensor.new("move", "boolean", false)
move: add()
```

il sensore "move", è di tipo booleano e di conseguenza non ha un range di valori da specificare, in quanto nella simulazione si randomizza il valore che può essere true o false, il valore di default è settato a false.
Analogamente, si crea "window".

```
window= Sensor.new("window", "boolean", false)
window: add()
```

Alarm ha cinque ATTUATORI:

- *siren*→ rappresenta la sirena d'allarme;
- *redLed*→ è un led rosso;
- *greenLed*→ è un led verde;
- *lamp*→ è una lampada;
- *beep*→ è un suono.

In LUA abbiamo usato questa sintassi, per creare un ATTUATORE:

```
siren= Actuator.new("siren", "boolean", false)
siren: add()
```

nella prima riga, si crea l'oggetto "siren" di tipo Attuatore con nome "siren", tipo "boolean" e valore di default uguale a false); il campo "change" inizializzato a false ed i campi "counterVal" e "counterCha" settati a zero.
Analogamente, si creano gli altri attuatori:

```
redLed= Actuator.new("redLed", "boolean", false)
redLed: add()

greenLed= Actuator.new("greenLed", "boolean", false)
greenLed: add()

beep= Actuator.new("beep", "boolean", false)
beep: add()

lamp= Actuator.new("lamp", "boolean", false)
```

```
lamp: add()
```

Le funzioni da utilizzare come azioni nelle transizioni o nei campi entry, doo ed exitt si creano così:

```
turnOnBeep= {name= "turnOnBeep", functionn= function() beep: setValue(true) print("turnOnBeep") end}
turnOffBeep= {name= "turnOffBeep", functionn= function() beep: setValue(false) print("turnOffBeep") end}

turnOnRedLed= {name= "turnOnRL", functionn= function() redLed: setValue(true) print("turnOnRedLed") end}
turnOffRedLed= {name= "turnOffRedLed", functionn= function() redLed: setValue(false) print("turnOffRedLed") end}

turnOnGreenLed= {name= "turnOnGreenLed", functionn= function() greenLed: setValue(true) print("turnOnGreenLed") end}
turnOffGreenLed= {name= "turnOffGreenLed", functionn= function() greenLed: setValue(false) print("turnOffGreenLed") end}

turnOnSiren= {name= "turnOnSiren", functionn= function() siren: setValue(true) print("turnOnSiren") end}
turnOffSiren= {name= "turnOffSiren", functionn= function() siren: setValue(false) print("turnOffSiren") end}

turnOnLamp= {name= "turnOnLamp", functionn= function() lamp: setValue(true) print("turnOnLamp") end}
turnOffLamp= {name= "turnOffLamp", functionn= function() lamp: setValue(true) print("turnOffLamp") end}
```

Sono dichiarate in forma tabellare con un campo "name" ed un campo "functionn" (appositamente scritto con doppia "n" per evitare conflitti con la parola chiave, in LUA, "function") che conterrà la vera e propria funzione da eseguire.

In tal modo, le funzioni precedentemente create, vengono assegnate ai campi "entry", "doo" ed "exitt" degli stati:

```
sleep: setExitt({turnOnBeep})
arm: setEntry({turnOnGreenLed})
arm: setExitt({turnOffGreenLed})
alarm: setEntry({turnOnRedLed})
alarm: setDoo({turnOnSiren})
alarm: setExitt({turnOnBeep})
```

Le condizioni, da inserire nelle transizioni, si dichiarano nel modo seguente, come funzione, perché il valore deve poter essere aggiornato contestualmente ai cambiamenti dei sensori:

```
c1= function() return insKey: getValue() == key: getValue() end
c2= function() return move: getValue() == true end
c3= function() return window: getValue() == true end
```

Le transizioni si dichiarano così:

```
sleep: addTransition({insKey}, {c1}, {turnOffBeep}, arm)

arm: addTransition({insKey}, {c1} , nil, sleep)
arm: addTransition({move}, {c2}, {turnOnLamp}, alarm)
arm: addTransition({window}, {c3}, {turnOnLamp}, alarm)

alarm: addTransition({insKey}, {c1}, {turnOffSiren, turnOffLamp, turnOffBeep}, sleep)
```

in questo modo, si riempiono le tabelle "transitions" degli stati che compaiono prima dei ":" con le tabelle contenenti le transizioni.
Ogni transizione ha come primo campo una tabella con sensori e/o attuatori (il cui cambiamento corrisponde all'evento della transizione), come secondo campo una tabella con la condizione, come terzo campo una tabella con una o più azioni e per ultimo campo, lo stato di arrivo della transizione.

Per l'esecuzione del simulatore si deve inizialmente settare lo stato di partenza:

sleep: setCurrentState()

Si possono eseguire due tipi di simulazioni:

MANUALE→ l'utente setta di volta in volta i valori dei sensori che vuole testare ed il risultato, quindi un eventuale cambiamento di stato e/o esecuzione di azioni, appare a video.

Ad esempio, in Alarm, se si setta:

insKey: setValue(1235)

appare a video

"sleep"

perché il cambiamento del sensore "insKey" (immessa password errata) non ha causato un cambiamento di stato.

Se si setta:

insKey: setValue(1234)

appare a video

turnOnBeep
turnOffBeep
turnOnGreenLed
"arm"

la prima riga perché è l'azione di exitt dello stato "sleep";
la seconda riga è l'azione della transizione dallo stato di "sleep" a quello di "arm";
la terza riga è l'azione di entry dello stato "arm";
la quarta riga è il nome dello stato "arm" perché è avvenuta una transizione.

Se si setta:

window: setValue(true)

stampa a video:

turnOffGreenLed
turnOnLamp
turnOnRedLed
turnOnSiren
"alarm"

la prima riga perché è l'exitt dello stato "arm";
la seconda riga è l'azione della transizione dallo stato "arm" allo stato "alarm";
la terza riga è l'entry dello stato "alarm";
la quarta riga è il doo di "alarm";
la quinta riga è il nome dello stato "alarm" in cui si è passati.

Per far notare il tentativo di inserimento errato della password, si continua l'esempio seguente:

si setta:

key: setValue(1236)

stampa a video:

"alarm"

perché si rimane nello stato di "alarm".

Si setta:

key: setValue(1234)

stampa a video:

turnOnBeep

turnOffSiren
turnOffLamp
"sleep"

è avvenuto un cambiamento di stato, perciò la prima riga è l'azione dell'exitt dello stato "alarm", la seconda e terza riga sono le azioni della transizione.

SIMULAZIONE RANDOM→ l'utente inserisce il numero di simulazioni che vuole effettuare e come secondo campo, inserisce "nil" se vuole che queste siano effettuate su tutti i sensori, invece inserisce una tabella con uno o più nomi dei sensori con il relativo range se vuole restringere la simulazione a pochi sensori scelti.
I risultati di tale simulazione, li stampa in un file chiamato "logFile" dove per ogni simulazione è indicato lo stato di partenza, il sensore modificato, il valore di questo sensore, eventuale transizione (ECA) e l'eventuale stato di arrivo. Alla fine stampa i DATI STATISTICI per ogni Stato (occurrences→ numero di volte in cui si è andati in quello stato), Sensore (change occurrences→ numero di volte in cui il sensore è stato modificato; transitions produced→ numero di volte in cui il cambiamento del sensore ha causato una transizione), Attuatore (change occurrences→ numero di volte in cui l'attuatore è stato modificato; transitions produced→ numero di volte in cui il cambiamento dell'attuatore ha causato una transizione)

simulator(100, nil)

simulator(50, {{insKey, 1233, 1235}, {move, nil, nil}, {window, nil, nil}})

Per mostrare l'esempio di un'errata progettazione della Macchina a Stati, che causa quindi, il cosiddetto "loop generato dalla Macchina", si aggiungono tali transizioni:

arm: addTransition({beep}, nil, {turnOnBeep}, sleep)
sleep: addTransition({beep}, nil, nil, arm)

Queste generano un loop infinito tra gli stati di "arm" e "sleep".

Di conseguenza, in aggiunta a quanto sopra descritto, nel "logFile", si stampano tutte le transizioni coinvolte nel loop e la stringa "THERE IS A LOOP" che precede l'interruzione delle simulazioni ed è seguita dalla stampa dei dati statistici e chiusura del file.

Lamp è una macchina a stati che simula il comportamento di una lampada che si accende quando un sensore si luminosità (lux) non rileva luce nella stanza.
Tale esempio serve a mostrare la differenza sostanziale tra i due tipi di "loop" che possono verificarsi:

- generato dall'ambiente;
- generato dalla macchina.

STATI

```
on= State.new("on")
on: add()

off= State.new("off")
off: add()
```

SENSORI

```
lux= Sensor.new("lux", "boolean", false)
lux: add()
```

ATTUATORI

```
lamp= Actuator.new("lamp", "boolean", false)
lamp: add()
```

FUNZIONI

```
turnOnLamp= {name= "turnOnLamp", functionn= function() lamp: setValue(true) print("turnOnLamp") end}
turnOffLamp= {name= "turnOffLamp", functionn= function() lamp: setValue(false) print("turnOffLamp") end}
```

CONDIZIONI

```
c1= function() return lux: getValue() == true end
c2= function() return lux: getValue() == false end
c3= function() return lamp: getValue() == true end
c4= function() return lamp: getValue() == false end
```

LOOP GENERATO DALL'AMBIENTE

```
off: addTransition({lux}, {c2}, {turnOnLamp}, on)
on: addTransition({lux}, {c1}, {turnOffLamp}, off)
```

si setta:

```
off: setCurrentState()

simulator(50, nil)
```

in tal modo, nel "logFile" si ha una continua alternanza tra gli stati di "on" ed "off". Se "lux" ha il valore settato a false, la lampadina si accende e si va nello stato di "on", mentre se "lux" ha il valore settato a true, la lampadina si spegne e si passa nello stato di "off".

LOOP GENERATO DALLA MACCHINA

```
off: addTransition({lux}, {c2}, {turnOnLamp}, on)
off: addTransition({lamp}, {c4}, {turnOnLamp}, on)
on: addTransition({lamp}, {c3}, {turnOffLamp}, off)
```

in tal caso, nel "logFile", si ha la stampa di un loop tra gli stati di "on" ed "off" causato non dall'ambiente (cambiamento dei sensori), ma dalla macchina stessa (cambiamento degli attuatori).