# CPS3233 – Verification Techniques

## Part 1 – Specifying the Elevator System

Miguel Dingli (49997M)
B.Sc. (Hons.) Computing Science

# Table of Contents

# 1 – Introduction

The following are some notes about the contents of the document:

- For properties that do not involve time, i.e. the invariants and temporal properties, timed automata were not included since they would simply be identical to the untimed finite state automata.
- For temporal property 1, labelled as 'Model checking only', only LTL/CTL specifications were included.
- Duration formulae (duration calculus) were intentionally only written for the real-time properties.
- For FSAs, it is generally assumed that the occurrence of any event that is absent from an FSA is ignored.
- LTL/CTL for temporal property 2 and FSA/RE for temporal property 3 were excluded due to difficulties.

# 2 – The Invariants

## 2.1 – Invariant 1

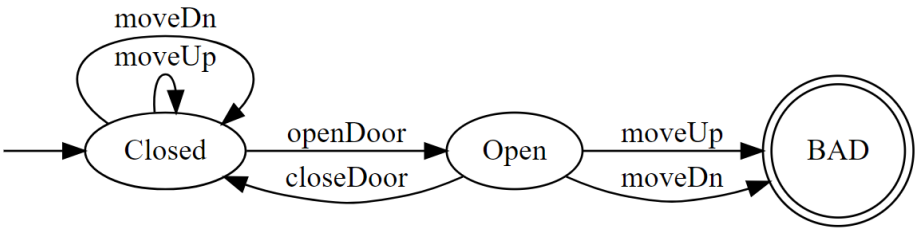
```
Elevator never moves up/down
when the door is not closed
```

*Property 1*

### 2.1.1 – Finite State Automaton

**Assumption**: lift starts with its door closed and stationary.

The reasoning behind the below automaton is that the lift is only allowed to move up or down while in a *Closed* state. Once the door opens, any attempt to move the lift results in a violation, unless the door is closed again.




### 2.1.2 – Regular Expression

The following *bad-behaviour regular* expression describes the undesired sequence of steps where the door opens and, after an optional sequence of events that do not cause the door to close, the lift move up or down.

$$?^{*}.openDoor.\left(\overline{closeDoor}\right)^{*}.(moveUp + moveDn)$$

### 2.1.3 – Linear Temporal Logic

The following LTL specification simply ensures that the *doorOpen* and *moving* sensors are never both true.

$$\boldsymbol{G}\left(\neg(doorOpen \wedge moving)\right)$$

### 2.1.4 – Computation Tree Logic

This CTL specification is simply the CTL-equivalent of the above LTL specification.

$$\boldsymbol{AG}\left(\neg(doorOpen \wedge moving)\right)$$

## 2.2 – Invariant 2

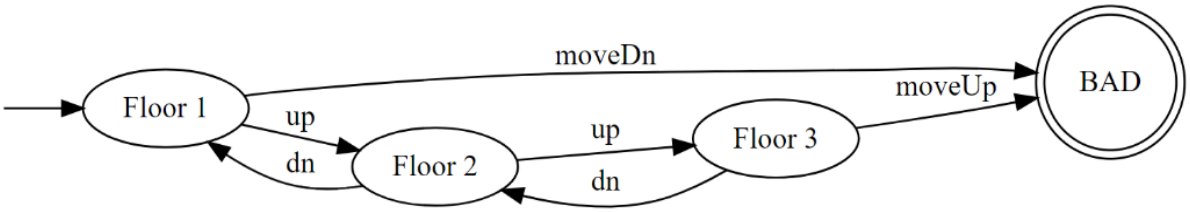
```
Elevator never attempts to go above the
topmost floor/below the lowermost floor
```

*Property 2*

### 2.2.1 – Finite State Automaton

**Assumptions**: lift starts at floor 1 and "up" and "dn" represents moving a whole floor, whereas "moveUp" and "moveDn" represent any degree of movement. Also, the number of floors is assumed to be 3.

The following finite state automaton tracks the current floor implicitly through the states. For any new floor reached using an up/down, the respective state is reached. At the state for the bottom floor (floor 1) or the top floor (in this case, floor 3), if the lift attempts to move down or up, respectively, this violates the property.




### 2.2.2 – Regular Expression

**Assumption**: the number of floors is assumed to be 3.

Regular expressions essentially cannot express counting and thus it is not possible to keep track of the current state. It was assumed that once the elevator reaches a floor, an appropriate event, such as "elevAt2" is generated. Thus, the following *good behaviour* regular expression ensures that once the elevator reaches floor 1, it is only allowed to move up until it reaches floor 2. The case for the top floor is similar.

$$\left(\left(\overline{elevAt1 + elevAt3}\right)^*.\left((elevAt1.moveUp^*.elevAt2) + (elevAt3.moveDn^*.elevAt2)\right)\right)^*$$

### 2.2.3 – Linear Temporal Logic

**Assumption**: the number of floors is assumed to be 3.

The following LTL specification guarantees that if the elevator is at the top floor, then the elevator is not allowed to move up until it moves down first. The case for the bottom floor is similar.

$$\boldsymbol{G}(elevAt3 \Rightarrow \boldsymbol{X}(\neg moveUp\ \boldsymbol{W}\ moveDn) \wedge$$
$$\boldsymbol{G}(elevAt1 \Rightarrow \boldsymbol{X}(\neg moveDn\ \boldsymbol{W}\ moveUp)$$

### 2.2.4 – Computation Tree Logic

**Assumption**: the number of floors is assumed to be 3.

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG}(elevAtN \Rightarrow \boldsymbol{AX}\ \boldsymbol{A}(\neg moveUp\ \boldsymbol{W}\ moveDn) \wedge$$
$$\boldsymbol{AG}(elevAt1 \Rightarrow \boldsymbol{AX}\ \boldsymbol{A}(\neg moveDn\ \boldsymbol{W}\ moveUp)$$

## 2.3 – Invariant 3

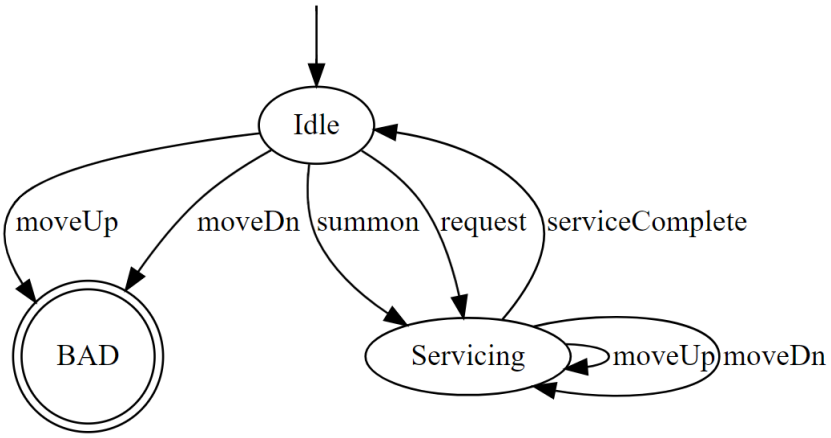
```
Elevator never moves unless a button press
occurs which has not yet been serviced
```

*Property 3*

### 2.3.1 – Finite State Automaton

**Assumptions**: lift starts with no requests, and there is only one elevator to service the summons and requests. It is also assumed that only one summon or request can occur between each service.

The following finite state automaton keep track of whether an unserviced button press exists (Servicing) or not (Idle). Once a button press occurs, the lift is allowed to move up or down. However, once the correct floor is reached, the system goes back to an idle state from which the lift cannot move up or down.




### 2.3.2 – Regular Expression

The following *good behaviour* regular expression ensures that no movement occurs until a button press (summon or request) occurs. The lift is then allowed to move up or down freely until the service is completed.

$$\left(\left(\overline{moveUp + moveDn}\right)^*.(summon + request).(moveUp + moveDn)^*.serviceComplete\right)^*$$

### 2.3.3 – Linear Temporal Logic

The following LTL specification ensures that the lift never moves *until* a request or summon is present. For just the time span that a request or summon is present, the *until* permits lift movement.

$$\boldsymbol{G}\left(\neg moving\ \boldsymbol{W}\ (request \vee summon)\right)$$

### 2.3.4 – Computation Tree Logic

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG\ A}\left(\neg moving\ \boldsymbol{W}\ (request \vee summon)\right)$$

## 2.4 – Invariant 4

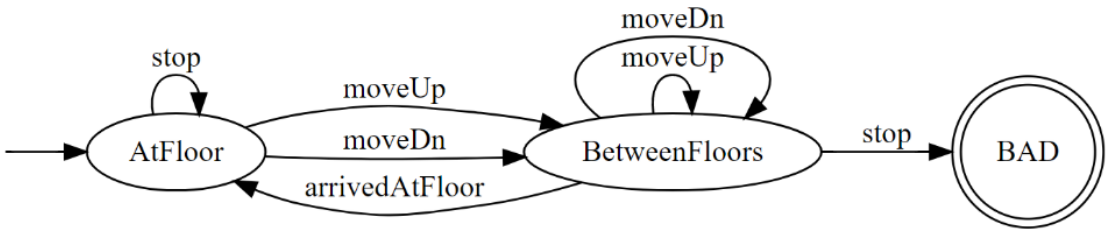
```
Elevator never stops in between floors
```

*Property 4*

### 2.4.1 – Finite State Automaton

**Assumption**: lift starts off stationary.

The following finite state automaton keeps track of whether the lift is at a floor or between floors. Once some movement occurs, the lift is assumed to be between floors (unless *arrivedAtFloor* occurs). From the state of being in between floors, the lift cannot *stop*. The lift is only allowed to *stop* while at a floor.




### 2.4.2 – Regular Expression

The following bad-behaviour regular expression describes the undesired sequence of steps where the lift starts moving up or down, and suddenly stops without the *arrivedAtFloor* event occurring. The "$\overline{arrivedAtFloor}$" part of the expression is simply meant to indicate the possibility of the occurrence of the *arrivedAtFloor* event.

$$?^* . (moveUp + moveDn)^* . (stop \ \& \ \overline{arrivedAtFloor})$$

### 2.4.3 – Linear Temporal Logic

The below LTL specification ensures that the elevator is never simultaneously between a floor and stationary.

$$\boldsymbol{G}(\neg(elevBetweenFloor \wedge stationary))$$

### 2.4.4 – Computation Tree Logic

This CTL specification is simply the CTL-equivalent of the above LTL specification.

$$\boldsymbol{AG}(\neg(elevBetweenFloor \wedge stationary))$$

## 2.5 – Invariant 5

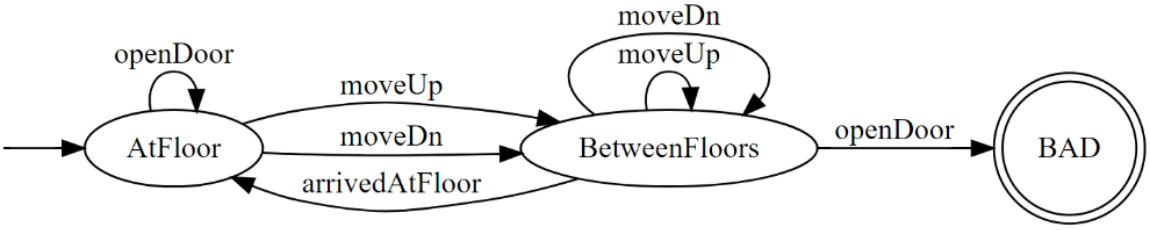
```
Elevator doors are only opened
once the elevator reaches a floor
```

*Property 5*

### 2.5.1 – Finite State Automaton

**Assumption**: lift starts off with its door closed.

Given the similarity between properties 4 and 5, the following automaton is very similar to that in 2.4.1.



### 2.5.2 – Regular Expression

Given the similarity between properties 4 and 5, the following expression is very similar to that in 2.4.2.

$$?^{*}.(moveUp + moveDn)^{*}.\left(openDoor\ \&\ \overline{arriveAtFloor}\right)$$

### 2.5.3 – Linear Temporal Logic

Given the similarity between properties 4 and 5, the following specification is very similar to that in 2.4.3.

$$\boldsymbol{G}\left(\neg(elevBetweenFloor \wedge doorOpen)\right)$$

### 2.5.4 – Computation Tree Logic

Given the similarity between properties 4 and 5, the following specification is very similar to that in 2.4.4.

$$\boldsymbol{AG}\left(\neg(elevBetweenFloor \wedge doorOpen)\right)$$

# 3 – The Temporal Properties

## 3.1 – Temporal 1 (Model Checking Only)

```
When a button (summon or floor request) is
pressed, elevator eventually services it
```

*Property 6*

### 3.1.1 – Linear Temporal Logic

For some floor $N$, the following LTL specification ensures that if a summon or a request occurs at floor $N$, the lift is eventually at floor $N$ with its door open, indicating that the summon or request was serviced.

$$\boldsymbol{G}\left((summAtN + reqForN) \Rightarrow \boldsymbol{F}\ (elevAtN \wedge doorOpen)\right)$$

### 3.1.2 – Computation Tree Logic

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG}\left((summAtN + reqForN) \Rightarrow \boldsymbol{AF}\ (elevAtN \wedge doorOpen)\right)$$

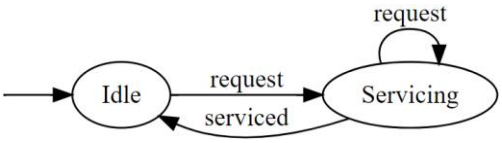
## 3.2 – Temporal 2

```
Multiple presses of the same button in between
servicing are considered as a single request
```

*Property 7*

### 3.2.1 – Finite State Automaton

The following finite state automaton represents the fact that if the lift is servicing the a request, any further requests, assumed to be for the same floor, are essentially ignored.




### 3.2.2 – Regular Expression

The following *good behaviour* regular expression shows how any one or more requests, assumed to be for the same floor, are always considered as one unservice request.

$$(request^{+}.service)^{*}$$

### 3.2.3 – Linear Temporal Logic

This task was skipped since no LTL specification for the given property could be formulated.

### 3.2.4 – Computation Tree Logic

This task was skipped since no CTL specification for the given property could be formulated.

## 3.3 – Temporal 3

> If an elevator is moving through a floor for which a summons button has been pressed, the elevator should service that floor. Otherwise, the elevator closest to the requested floor should service it

*Property 8*

### 3.3.1 – Finite State Automaton

This task was skipped since no finite state automaton for the given property could be formulated.

### 3.3.2 – Regular Expression

This task was skipped since no regular expression for the given property could be formulated.

### 3.3.3 – Linear Temporal Logic

For some floor $N$, the following LTL specification ensures that if the lift is at a floor that has the summon button pressed, the elevator will have its doors open by the next step, so that it services the floor. This specification does not consider the closest-lift selection process.

$$\boldsymbol{G}\left(atFloorN \wedge summAtN \Rightarrow \boldsymbol{X}(atFloorN \wedge doorOpen)\right)$$

### 3.3.4 – Computation Tree Logic

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG}\left(atFloorN \wedge summAtN \Rightarrow \boldsymbol{AX}(atFloorN \wedge doorOpen)\right)$$

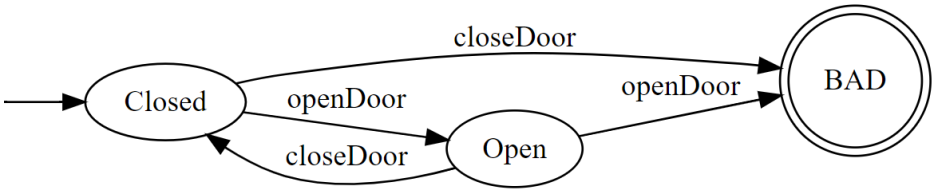
## 3.4 – Temporal 4

> Door opening/closing signals always alternate each other; there should never be two consecutive door opening/two consecutive door closing

*Property 9*

### 3.4.1 – Finite State Automaton

The following finite state automaton keeps track of whether the door is open or closed and ensures that the door never closes from a closed state and doesn't open from an open state.




### 3.4.2 – Regular Expression

The following *good behaviour* regular expression intuitively ensures that door opens and closes occur in pairs.

$$(openDoor.closeDoor)^*$$

### 3.4.3 – Linear Temporal Logic

**Assumption**: the signals for opening and closing the door cannot happen simultaneously.

The following two-part LTL specification ensures that (i) if the *closeDoor* signal occurs, then it cannot occur again *until* the *openDoor* signal occurs, and that (ii) if the *openDoor* signal occurs, it cannot occur again *until* the *closeDoor* signal occurs. Thus, this enforces the alternation of the opening and closing of the door.

$$\boldsymbol{G}(closeDoor \Rightarrow \boldsymbol{X}(\neg closeDoor\ \boldsymbol{W}\ openDoor) \wedge \\ openDoor \Rightarrow \boldsymbol{X}(\neg openDoor\ \boldsymbol{W}\ closeDoor))$$

### 3.4.4 – Computation Tree Logic

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG}(closeDoor \Rightarrow \boldsymbol{AX}\ \boldsymbol{A}(\neg closeDoor\ \boldsymbol{W}\ openDoor) \wedge \\ openDoor \Rightarrow \boldsymbol{AX}\ \boldsymbol{A}(\neg openDoor\ \boldsymbol{W}\ closeDoor))$$

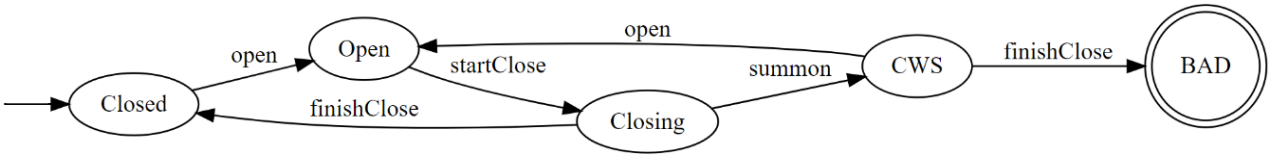
## 3.5 – Temporal 5

```
If summon button pressed for a floor where
door is closing, the door should open again
```

*Property 10*

### 3.5.1 – Finite State Automaton

The following finite state automaton keeps track of whether the door is open, closed, closing, or CWS (i.e. closing with summon). Once the lift door is open, they eventually start closing. At the *closing* state, if no summon occurs, the doors close, but if the lift reaches the CWS state, it can only re-open or the property is violated.




As such, summons are allowed to occur at any time and at any frequency of occurrence. The above finite state automaton does not aim to put restrictions on the summons, but rather on the act of closing the lift doors.

### 3.5.2 – Regular Expression

Once the doors open, the following *good behaviour* regular expression permits the possible repeated three-step process of receiving a summon, re-opening the doors, and starting to close again, in between when the door initially started to close and when the door finishes closing.

$$(open.startClosing.(summon.open.startClosing)^*.finishClosing)^*$$

As such, summons are allowed to occur at any time and at any frequency of occurrence. The above regular expression does not aim to put restrictions on the summons, but rather on the act of closing the lift doors.

### 3.5.3 – Linear Temporal Logic

The following LTL specification ensures that if the door is closing and a summon has occurred, the lift will always have its doors open again in the next step.

$$\boldsymbol{G}\big(doorClosing \wedge summonOccurred \Rightarrow \boldsymbol{X}(openDoor)\big)$$

### 3.5.4 – Computation Tree Logic

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG}\big(doorClosing \wedge summonOccurred \Rightarrow \boldsymbol{AX}(openDoor)\big)$$

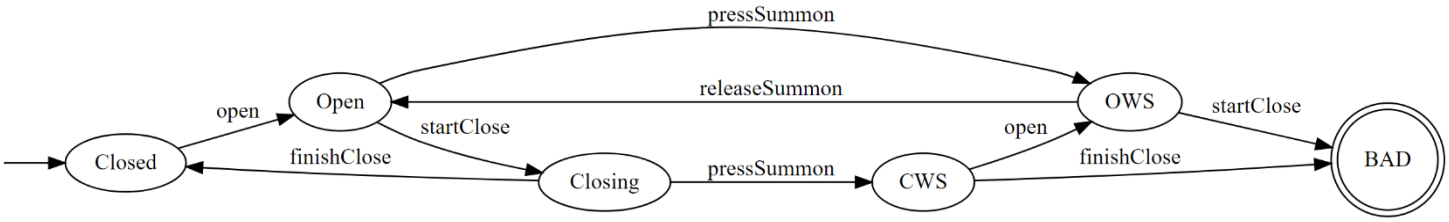
## 3.6 – Temporal 6

> If summon button held down on a floor where door is not closed, then door should open and remain open

*Property 11*

### 3.6.1 – Finite State Automaton

The following finite state automaton is similar to that in 3.5.1 but adds an OWS (open with summon) state and considers the pressing and releasing of the summon button. Once the summon button is pressed (and assumingly held) from an un-closed (open or closing) state, the door is not allowed to start closing or finish closing unless the summon button is released. In the case of closing, the door is expected to re-open (CWS to OWS).




Similar to the automaton for property 10, the above does not aim to put restrictions on the summons.

### 3.6.2 – Regular Expression

The following *good behaviour* expression ensures that after the lift doors open (which may start closing) this may be followed by a repetition of holding the summon button, causing the door to open, and releasing it, causing the door to start closing. The lift door is only allowed to finish closing after this repeated process stops.

$$\left(open.\,(startClosing + 1).\left(\begin{matrix} pressSummon.\,open. \\ releaseSummon.\,startClosing \end{matrix}\right)^{*}.finishClosing\right)^{*}$$

Similar to the regular expression for property 10, the above does not aim to put restrictions on the summons.

### 3.6.3 – Linear Temporal Logic

The following LTL specification ensures that if the door is either not closed, it must remain open from the next step until the summon button is released. It does not need to remain open if the button is not being held.

$$\boldsymbol{G}\left(\neg doorClosed \Rightarrow \boldsymbol{X}(doorOpen\ \boldsymbol{W}\ summonNotHeld)\right)$$

### 3.6.4 – Computation Tree Logic

The following CTL simply adds 'A's to the above LTL, since the specification must apply for any possible path.

$$\boldsymbol{AG}\left(\neg doorClosed \Rightarrow \boldsymbol{AX}\ \boldsymbol{A}(doorOpen\ \boldsymbol{W}\ summonNotHeld)\right)$$

# 4 – The Real-Time Properties

## 4.1 – Real-Time 1 (Timed Formalisms Only)

```
Upon a request, after the door closes, the el-
evator starts moving in less than 3 seconds
```

*Property 12*

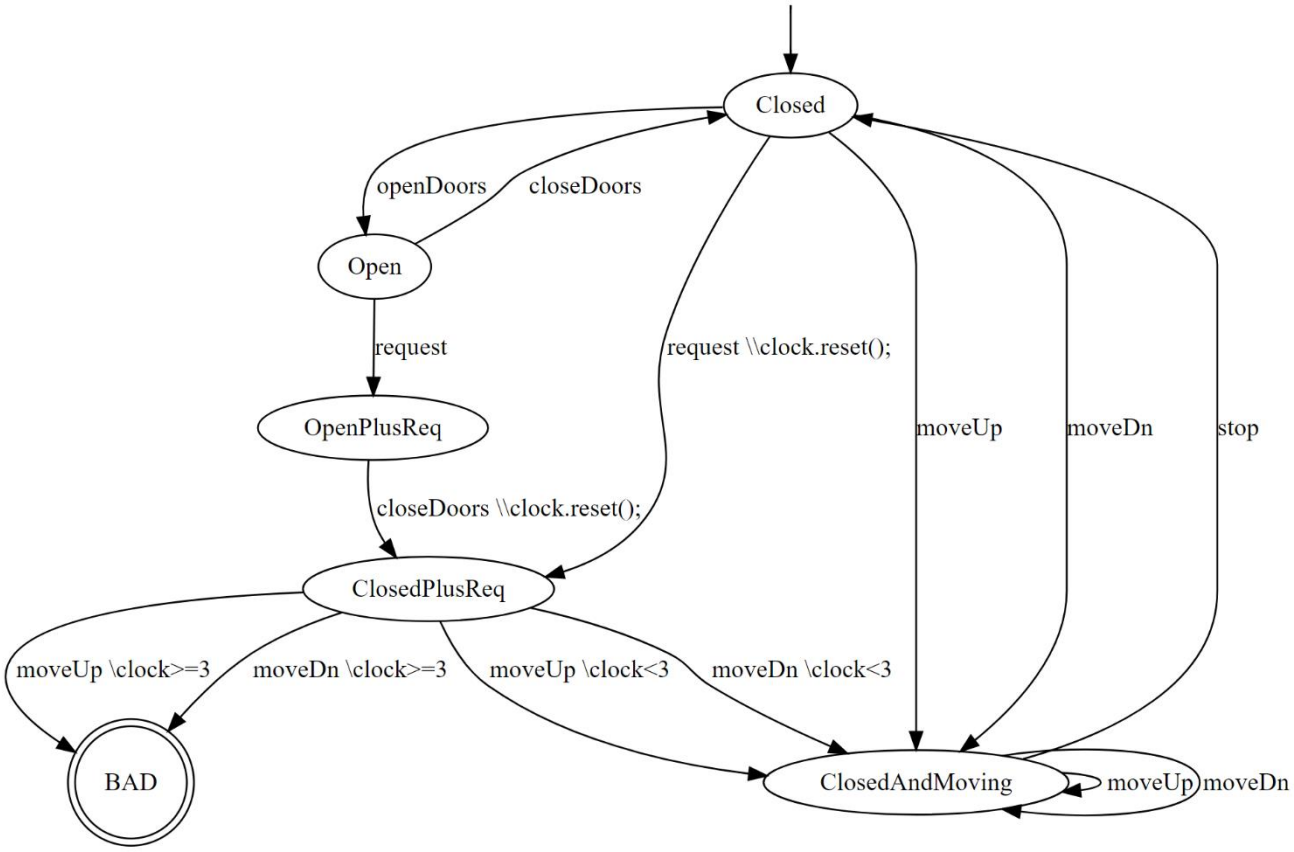
### 4.1.1 – Duration Calculus

The following duration formula indicates that the period of time consisting of the lift stationary with its doors closed and an unserviced request, which is followed by the lift starting to move, must last less than 3 seconds. It is assumed that the lift retains the *doorClosed* and *hasRequest* states while moving.

$$\Box([doorClosed \wedge \neg moving \wedge hasRequest]\,;[moving] \Rightarrow length < 3\,;[moving])$$

### 4.1.2 – Timed Automata

The following timed automaton keeps track of whether the door is closed or not and whether an unserviced request exists. If a request is received while the doors are open (OpenPlusReq), a stopwatch is started once the doors close (ClosedPlusReq). If the door was already closed when the request is received, the stopwatch is stated immediately. The lift must then start moving up or down before three seconds elapse or otherwise the property is violated. From the *Closed* state, the lift may move freely without time restrictions.




As such, summons are allowed to occur at any time and at any frequency of occurrence. The above finite state automaton does not aim to put restrictions on the summons, but rather on the timing of the lift movement.

## 4.2 – Real-Time 2 (Timed Formalisms Only)

```
After the door has been open for 3
seconds, it closes automatically
```

*Property 13*

### 4.2.1 – Duration Calculus

The following duration formula indicates that for a time interval of 3 or more seconds consisting of an open door followed by a closing door, the closing of the door starts after three seconds with the door open. It has been assumed that the property requires the door to be open for exactly 3 seconds before it starts to close.

$$\square(\text{length} > 3 \wedge [DoorOpen]\,;[DoorClosing] \Rightarrow \text{length} = 3\,;[DoorClosing])$$

### 4.2.2 – Timed Automata

The following timed automaton, using Larva DATE notation for the clock values and clock resets, ensures that the time that has elapsed from when the door opens to when it goes from an open to a (closing or) closed state is greater than three seconds. The door is not allowed to close before the three seconds elapse.