CPS3233 – Verification Techniques

Part 2 – Runtime Verification

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

Table of Contents

1 – Introduction 2
2 – Implementing the Invariants 2
2.1 – Invariant 1 2
2.2 – Invariant 2 2
2.3 – Invariant 3 3
2.4 – Invariant 4 4
2.5 – Invariant 5 4
3 – Implementing the Temporal Properties 5
3.1 – Temporal 1 5
3.2 – Temporal 2 5
3.3 – Temporal 3 6
3.4 – Temporal 4 7
3.5 – Temporal 5 7
3.6 – Temporal 6 8
4 – Implementing the Real-Time Properties 8
4.1 – Real-Time 1 8
4.2 – Real-Time 2 9
5 – Testing the Monitors 10
5.1 – Testing the Invariants 10
5.2 – Testing the Temporal Properties 10
5.2.1 – Testing Temporal Property 2 10
5.2.2 – Testing Temporal Property 3 10
5.2.3 – Testing Temporal Property 4 11
5.2.4 – Testing Temporal Property 5 11
5.3 – Testing the Real-Time Properties 11
6 – Improvements 11

1 – Introduction

All properties in the written Larva script were implemented individually so that if a property is violated or is faulty, this does not affect other properties. A negative side-effect of this is that it causes an increase in duplicate code throughout the Larva script, which is expected to increase the monitoring overhead.

In the *Runner* class implemented to run the system with 6 floors and 3 lifts, the randomization of lift positions was disabled. Thus, the lifts are all assumed to start from the bottom floor, which decreases the possible non-determinism in the monitoring results. However, setting this to true is not expected to cause issues.

Assumption: In general, the lifts are assumed to start as not moving and with the doors closed. Some property-specific assumptions are included throughout the document.

Issue: an issue noticed while running the elevator system is that if the number of summons is greater than the number of lifts, the system crashes. Some property-specific issues are included throughout the document.

2 – Implementing the Invariants

2.1 – Invariant 1

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

Property 1

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. Using the *startOrStopMoving* event, the property is violated if the lift starts moving and its door is open, using *isOpen()*.

```
EVENTS{
    startOrStopMoving(boolean isNowMoving) = {Lift l1.setMoving(isNowMoving)}
        where l = l1;
    ...
}
                                        ...
PROPERTY invar1{
    STATES{ BAD { bad } STARTING { start } }
    TRANSITIONS{ start -> bad [ startOrStopMoving \isNowMoving && l.isOpen() ] }
}
```

Listing 1 – Property 1 Implementation

The above could have been extended by considering the case where the door opens while the lift is moving, i.e. by adding a transition "`start -> bad [openDoors \l.isMoving()]`". Since the system erroneously sets the lift to stationary *after* closing the doors, the property is violated if this transition is added.

2.2 – Invariant 2

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

Property 2

The implementation for this property was placed in the `FOREACH(Shaft s)` context of the script. To detect lift movement and the direction of movement, the animation methods were extracted from the shaft along with the current floor. The property is violated if an attempt is made to move the lift upwards and the current (zero-based) floor is $n - 1$ or if an attempt is made to move the lift downwards and the current floor is 0.

The first attempt at implementing this property made use of the *setFloor(int floor)* method in the Lift class to check that no illegal values are passed to the method. However, the lift would have already moved by the point that this method is called from the Shaft class. The animation methods, called before, were used instead.

```
EVENTS{
    animateUp(int currFloor) = {Shaft s1.animateUp(currFloor)} where s = s1;
    animateDn(int currFloor) = {Shaft s1.animateDown(currFloor)} where s = s1;
}
                                        ...
PROPERTY invar2{
    STATES{ BAD { bad } STARTING { start } }
    TRANSITIONS{
        start -> bad [ animateUp \currFloor == s.numFloors - 1 ]
        start -> bad [ animateDn \currFloor == 0 ]
    }
}
```

Listing 2 – Property 2 Implementation

2.3 – Invariant 3

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

Property 3

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. The property keeps track of requests and summons by using a *reqOrSumm* counter which increments whenever the *moveLift* method is called in the LiftController class. This method is used for both requests and summons. Once the lift starts moving, it *services* one of these requests or summons, and thus the counter is decremented. If the counter is zero, the property is violated because the lift in consideration moved for no reason.

The implementation above makes the assumption that when the lift moves, the intention is to service one of the requests or a summon. The original property does not require this intention to be guaranteed. Lastly, for this property to work as intended, an extra "*lift.setMoving(true);*" in the *animateLift()* method of the Shaft class was commented out since it is called twice for no obvious reason.

```
VARIABLES{
    int reqOrSumm = 0;
    ...
}
                                        ...
EVENTS{
    ...
    requestOrSummon() = {LiftController *.moveLift(Lift l1, *)} where l = l1;
    ...
}
                                        ...
PROPERTY invar3{
    STATES{ BAD { bad } STARTING { start } }
    TRANSITIONS{
        start -> start [ requestOrSummon \\reqOrSumm++; ]
        start -> bad [ startOrStopMoving \isNowMoving && reqOrSumm == 0 ]
        start -> start [ startOrStopMoving \isNowMoving \reqOrSumm--; ]
    }
}
```

Listing 3 – Property 3 Implementation

2.4 – Invariant 4

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

Property 4

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. To implement this property, the previously discussed *startOrStopMoving* event was reused. However, in this case, the property is violated if the elevator stops moving while the lift's *isBetweenFloors()* method returns true.

The *isBetweenFloors()* method was not originally present in the Lift class. However, the system specification indicates that the elevator has a sensor for detecting whether the lift is between floors. Thus, a *betweenFloors* boolean, initialised to false, was added to the Lift class, along with a *setAsBetweenFloors()* method which sets the boolean to true and an *isBetweenFloors()* getter method. Additionally, *betweenFloors* is set to false whenever the *setFloor()* method of the Lift class is invoked, since the lift is no longer between floors at that point. Lastly, the *setAsBetweenFloors()* is invoked from *animateLift()*in the Shaft class right before animating the lift.

```
PROPERTY invar4{
    STATES{ BAD { bad } STARTING { start } }
    TRANSITIONS{
        start -> bad [ startOrStopMoving \!isNowMoving && l.isBetweenFloors() ]
    }
}
```

Listing 4 – Property 4 Implementation

2.5 – Invariant 5

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

Property 5

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. This property reuses the *isBetweenFloors()* method implemented for Invariant 4. The property is considered violated if the lift's doors open and the lift is in between floors.

For this property to work as intended, calls to the *openDoors()* and *closeDoors()* methods of the Lift were added in the Shaft's *openDoors()* and *closeDoors()* methods. Otherwise, the lift's methods are never invoked, meaning that the lift effectively remains with its doors closed throughout the execution.

```
EVENTS{
    ...
    openDoors() = {Lift l1.openDoors()} where l = l1;
    ...
}
                            ...
PROPERTY invar5{
    STATES{ BAD { bad } STARTING { start } }
    TRANSITIONS{
        start -> bad [ openDoors \l.isBetweenFloors() ]
    }
}
```

Listing 5 – Property 5 Implementation

3 – Implementing the Temporal Properties

3.1 – Temporal 1

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

Property 6

Since this property is labelled as "Model checking only", it was not implemented in this part of the assignment.

3.2 – Temporal 2

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

Property 7

The implementation for this property was placed in the `GENERAL` context of the script. This property was assumed to refer to the just the summons, such that two summons from the same floor should not be serviced using more than one lift. On the other hand, pressing the in-lift buttons multiple times does not have as much of a negative consequence since the other lifts are not involved, and thus such requests were not considered.

To keep track of the summons being serviced, a HashSet stores the respective levels. For each summon, the property transitions to the *summon* state, where it assumes that a lift will be selected to service the summon (the moveLift method does not actually move the lift immediately). Once a lift is assigned, a condition checks whether the floor is already being serviced whilst it adds the floor to the summons set. The property is violated if the level was already there. Back at the *start* state, each level reached by a lift is assumed to be serviced.

```
VARIABLES{
    ...
    Set<Integer> summonsBeingServiced = new HashSet<>();
}
                                        ...
EVENTS{
    floorSummon(int floorNumber) = {LiftController *.callLiftToFloor(floorNumber)}
    moveLift(Lift lift, int floorNumber) = {LiftController *.moveLift(lift,
        floorNumber)}
    setFloor(Lift lift, int floorNumber) = {lift.setFloor(floorNumber)}
    ...
}
                                        ...
PROPERTY temp2{
    STATES{ BAD { bad } NORMAL { summon } STARTING { start } }
    TRANSITIONS{
        start  -> summon [ floorSummon ]
        summon -> bad    [ moveLift \!summonsBeingServiced.add(floorNumber) ]
        summon -> start  [ moveLift ]
        start  -> start  [ setFloor \\summonsBeingServiced.remove(floorNumber); ]
    }
}
```

Listing 6 – Property 7 Implementation

The above implementation could have been extended to also consider multiple identical in-lift requests. If the lift is given such requests, it opens the door repeatedly at the same floor. Thus, the implementation would keep track of the requests and ensure that the door does not open more than once if the requests are identical. Due to the described lift behaviour, the property ends up getting violated since the extras are not ignored.

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

The implementation for this property was placed in the `GENERAL` context of the script. To keep track of the floors that each elevator is passing through, a *destinations* HashMap with lift-floor pairs is used. This is populated with the lifts and destinations obtained from the *moveLift* event. Whenever a lift reaches a floor, if this lift-floor is present in the map, it is removed since the lift has reached its destination.

The more complex task is to ensure that the lift selected to service a summon is the optimal one. The *getClosestStationaryLifts()* method in the LiftController is assumed to be the only part of the system which makes this decision. The list of lifts returned from this method is compared to that from the implemented *getBestLifts()* method, which first considers the lifts that will pass through the floor using the *destinations* map and, if no lift is found, it considers the closest stationary lifts, similar to how *getClosestStationaryLifts()* works. If the results from these two methods are not equal, then the system is assumed to have selected an incorrect lift.

(The discussion continues overleaf)

```
VARIABLES{
    Map<Lift, Integer> destinations = new HashMap<>();
    ...
}
                                        ...
EVENTS{
    ...
    getClosest(LiftController lc, int floorNumber, Object lifts) = {execution
        LiftController lc.getClosestStationaryLifts(floorNumber)uponReturning(lifts)}
}
                                        ...
PROPERTY temp3{
    STATES{ BAD { bad } STARTING { start } }
    TRANSITIONS{
        start -> bad [ getClosest \!getBestLifts(lc.getLifts(), floorNumber,
            destinations).equals(lifts) ]
        start -> start [ moveLift \\destinations.put(lift, floorNumber); ]
        start -> start [ setFloor \floorNumber==destinations.getOrDefault(lift, -1)
            \destinations.remove(lift); ]
    }
}
                                        ...
METHODS {
    ArrayList<Lift> getBestLifts(Lift[] lifts, int floor,
            Map<Lift,Integer> destinations) {

        final ArrayList<Lift> result = new ArrayList<Lift>();

        // Search for moving lifts which will pass through the floor
        ...

        // Search for closest stationary lifts
        ...

        return result;
    }
}
```

Listing 7 – Property 8 Implementation

The *getBestLifts()* method assumes that if multiple elevators are passing through a summon, all of them stop to service it. This could have been improved by selecting the closest lift, however, it is not straightforward to detect which lift was actually selected to service a summon unless the lifts' movements are tracked. It was also assumed that, in general, once a lift (or lifts) is assigned to service a summon, this decision never changes.

3.4 – Temporal 4

```
Door opening/closing signals always alternate
each other; there should never be two consecu-
tive door opening/two consecutive door closing
```

Property 9

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. This property simply alternates between the *closed* and *open* states and if an unexpected action occurs, the property is violated. The *openDoors* event is reused from a previous property but the *closeDoors* event had to be added.

```
EVENTS{
    ...
    closeDoors() = {Lift l1.closeDoors()} where l = l1;
    ...
}
                                ...
PROPERTY temp4{
    STATES{ BAD { bad } NORMAL { open } STARTING { closed } }
    TRANSITIONS{
        closed -> open [ openDoors ]
        open -> closed [ closeDoors ]
        closed -> bad [ closeDoors ]
        open -> bad [ openDoors ]
    }
}
```

Listing 8 – Property 9 Implementation

3.5 – Temporal 5

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

Property 10

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. This property detects invocations of the *startClosingDoors()* method in the Lift class to determine when the lift is in the "closing" stage. The automaton starts with *closed* doors. Once these are open and start closing, if a summon (using the previously presented *floorSummon* event) occurs at the same floor, the lift is expected to go back to the *open* state. Otherwise, if a summon does not occur, the lift can go back to the *closed* state normally.

Since the elevator system does not implement a doors closing functionality, this was added so that the property could be checked. A dummy *startClosingDoors()* method was added to the Lift class. This is invoked from the Shaft which, rather than closing the doors after 2000ms, it was modified to call the method after 1000ms and then proceed to call the *closeDoors()* method after a further 1000ms, for an unaltered total of 2000ms. As a visual reference of this functionality at work, the doors are set to yellow while in the intermediary stage.

```
EVENTS{
    ...
    startClosingDoors() = {Lift l1.startClosingDoors()} where l = l1;
    ...
}
                                        ...
PROPERTY temp5{
    STATES{ BAD { bad } NORMAL { open closing summon } STARTING { closed } }
    TRANSITIONS{
        closed -> open [ openDoors ]
        open -> closing [ startClosingDoors ]
        closing -> closed [ closeDoors ]
        closing -> summon [ floorSummon \floorNumber == l.getFloor() ]
        summon -> open [ openDoors ]
        summon -> bad [ closeDoors ]
   }
}
```

Listing 9 – Property 10 Implementation

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

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. Since the elevator system does not implement the button holding functionality described in the property, the property was not implemented. However, a commented-out property is still presented below, indicating how the property would be implemented if this functionality was present, involving a button *press* and *release* events.

```
%%PROPERTY temp6{
%%    STATES{
%%        BAD { bad }
%%        NORMAL { open summonHeld }
%%        STARTING { closed }
%%    }
%%    TRANSITIONS{
%%        closed  -> open     [ openDoors ]
%%        open -> summonHeld [ summonPress ]
%%        summonHeld -> open [ summonRelease ]
%%        open -> closed      [ closeDoors ]
%%    }
%%}
```

Listing 10 – Property 11 Implementation

4 – Implementing the Real-Time Properties

4.1 – Real-Time 1

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

Property 12

The implementation for this property was placed in the `FOREACH(Lift l)` context of the script. For the timing aspect, the *time1Clock* clock was added along with a *clock1* event which triggers after 3 seconds. To

detect user requests, the lift-number version of the *moveLift* method, which is used for such requests, was extracted. The lift was obtained through the LiftController's getter for the lifts array.

The implemented automaton's states keep track of whether the lift's door is open and whether an outstanding request exists. The *closedPlusReq* is reached in two ways: (i) the request was received when the door was open and the door closed, or (ii) the request was received when the door was closed. In both cases, the clock is reset so that if 3 seconds elapse before the lift starts moving (*closedAndMoving*), the property is violated. Otherwise, the lift eventually stops moving and the automaton returns to the *closed* state.

Requests that are received when the lift is already moving are not considered for this property. If these should also be considered, a counter can keep track of the number of outstanding requests so that whenever the lift stops, opens its doors and closes, the clock can be reset to ensure that it starts moving in less than 3 seconds.

```
VARIABLES{
    ...
    Clock time1Clock = new Clock();
    ...
}
                                        ...
EVENTS{
    ...
    floorRequest() = {LiftController ctrl.moveLift(int liftNo, *)}
        where l = ctrl.getLifts()[liftNo];
    clock1() = {time1Clock@3}
    ...
}
                                        ...
PROPERTY time1{
    STATES{
        BAD { bad }
        NORMAL { open openPlusReq closedPlusReq closedAndMoving }
        STARTING { closed }
    }
    TRANSITIONS{
        closed -> open [ openDoors ]
        open -> closed [ closeDoors ]
        open -> openPlusReq [ floorRequest ]
        closed -> closedPlusReq [ floorRequest \\time1Clock.reset(); ]
        openPlusReq -> closedPlusReq [ closeDoors \\time1Clock.reset(); ]
        closedPlusReq -> bad [ clock1 ]
        closedPlusReq -> closedAndMoving [ startOrStopMoving \isNowMoving ]
        closed -> closedAndMoving [ startOrStopMoving \isNowMoving ]
        closedAndMoving -> closed [ startOrStopMoving \!isNowMoving ]
    }
}
```

Listing 11 – Property 12 Implementation

4.2 – Real-Time 2

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

Property 13

The implementation was placed in the `FOREACH(Lift l)` context. For the timing aspect, the *time2Clock* clock was added along with a *clock2* event which triggers after 3 seconds. The automaton for this property resets the clock whenever the door opens, which can occur both from a closed state or a closing state.

From an open state, it is assumed that the door cannot be closed manually and that the door should close automatically after at least 3 seconds elapse. The implementation does not guarantee that the door eventually

closes but this could have been added by using a secondary clock as a timeout for the automatic close. Once the door reaches the *canClose* state, it can return back to the *closingOrClosed* state by closing its doors.

```
VARIABLES{
    ...
    Clock time2Clock = new Clock();
}
                                    ...
EVENTS{
    ...
    clock2() = {time2Clock@3}
}
                                    ...
PROPERTY time2{
    STATES{
        BAD { bad }
        NORMAL { open canClose }
        STARTING { closingOrClosed }
    }
    TRANSITIONS{
        closingOrClosed -> open [ openDoors \\time2Clock.reset(); ]
        open -> bad [ startClosingDoors ]
        open -> canClose [ clock2 ]
        canClose -> closingOrClosed [ startClosingDoors ]
    }
}
```

Listing 12 – Property 13 Implementation

5 – Testing the Monitors

5.1 – Testing the Invariants

From the five invariants, none of them are expected to be violated, mainly because they deal with simple but fundamental aspects of the elevator system. Due to their simplicity, the invariants were all tested together by performing various summons and requests ranging across all of the floors, from top to bottom.

These summons and requests (i) cover invariant 1 since they involve movement and requests generated when the doors are in an open state, (ii) cover invariant 2 since they involve movement up to the top and down to the bottom floor, (iii) cover invariant 3 since they involve movement and multiple summons and requests, (iv) cover invariant 4 since they involve movement between floors, and (v) cover invariant 5 since they involve numerous opening and closing of the elevator doors. This testing resulted in no violations of the properties.

5.2 – Testing the Temporal Properties

From the six temporal properties, properties 1 and 6 were not implemented, properties 2, 3, and 5 are expected to be violated, and property 4 is not expected to be violated.

5.2.1 – Testing Temporal Property 2

This property was tested by performing two summons from the same floor before the first summon is serviced. The property fails as expected since, instead of ignoring the second summon, a second lift is sent to service it.

5.2.2 – Testing Temporal Property 3

This property was tested by performing a summon for the top floor and then performing another summon for the fifth (the floor under the top floor) before the elevator reaches the top floor. The property fails as expected because rather than the same elevator servicing the second summon, a second lift is sent to service it.

5.2.3 – Testing Temporal Property 4

This property was tested by performing various summons and request, similar to section 5.1, since a sequence of random opening and closing of doors suffices to test this property. As expected, this property is not violated.

5.2.4 – Testing Temporal Property 5

This property was tested by performing a summon for a lift, to cause its door to open and, when the door is closing, the summon button was pressed once more. As expected, the property fails because, rather than the door re-opening its doors, a second lift is sent to service the summon.

5.3 – Testing the Real-Time Properties

From the two real-time properties, the property 1 is not expected to be violated, since the lift starts moving after two seconds, while property 2 is expected to be violated, since the door closes before the 3 seconds. To test these two real-time properties it suffices to perform a sequence of summons and requests, similar to 5.1.

These summons and requests (i) cover the first property since they involve opening and closing of elevator doors and (ii) cover the second property since, once more, they involve opening and closing of elevator doors. As expected, the first real-time property is not violated whilst the second real-time property is violated.

6 – Improvements

Some possible improvements were already mentioned throughout the document. However, some general improvements are also possible. Despite the fact that all properties were intentionally kept separate, as mentioned in the introduction, trivial parts of the script which are unlikely to be faulty could still have been made common to the properties. For example, a single property could have handled all of the opening and closing of doors and passed on the information to other properties using channels.

Additional improvements are perhaps also possible for the second and third temporal properties, which store information in a HashMap and HashSet rather than depending on real-time information from the system itself.

A final possible improvement could have been to make use of actual invariants in the Larva script. However, an attempt to use these resulted in an unexpected NullPointerException and so these were ultimately avoided.