L-Università ta' Malta
Faculty of Information & Communication Technology

CPS3233 Assignment Report

Manwel Bugeja

September 17, 2021

Contents

1 Elevator System Specification **2**
1.1 Finite State Automata 2
1.2 Regular Expressions 3
1.3 Timed Automata 3
1.4 Duration Calculus 5

2 Runtime Verification **6**
2.1 Testing 8
2.2 Larva script improvements 9

3 Model-Based Testing **12**
3.1 BasicLiftModel 12
3.2 LiftTimeModel 15

4 Runtime Verification and Testing **15**

1 Elevator System Specification

In this section, the specifications of the elevator system are expressed in several different formal notations. The notations are Finite State Automata (FSAs), Regular Expressions (RE), Timed Automata (TAs) and Duration Calculus (DC). From the ones listed, TAs and DC are the capable of expressing timed events.

1.1 Finite State Automata

A finite state automata was designed to describe the elevator system. The formal definition of the state machine can be seen in listing 1. The lift can be in one of three states: Idle, Loading or Moving. Idle is when the lift is not moving and has its door closed. Loading when the lift is not moving but has the doors open. Moving is when the lift is moving either up or down. Note that a 'bad' state could be added for when the lift is moving and has the doors open as the lift should never be in this state.

This FSA describes the flow of the lift lifetime in a very basic nature for example it does not consider elevator summoning or floor requests. This can be seen visually in figure 1. A more detailed automaton considering more actions and time constraints is discussed in section 1.3.

Listing 1: FSA definition

```
M = {
{Idle, Loading, Moving},
{openDoor, closeDoor, stop, goUp, goDown},
{
    Idle, openDoor -> Loading,
    Idle, goUp -> Moving,
    Idle, goDown -> Moving,
    Loading, closeDoor -> Idle,
    Moving, stop -> Idle
},
Idle,
{}
}
```

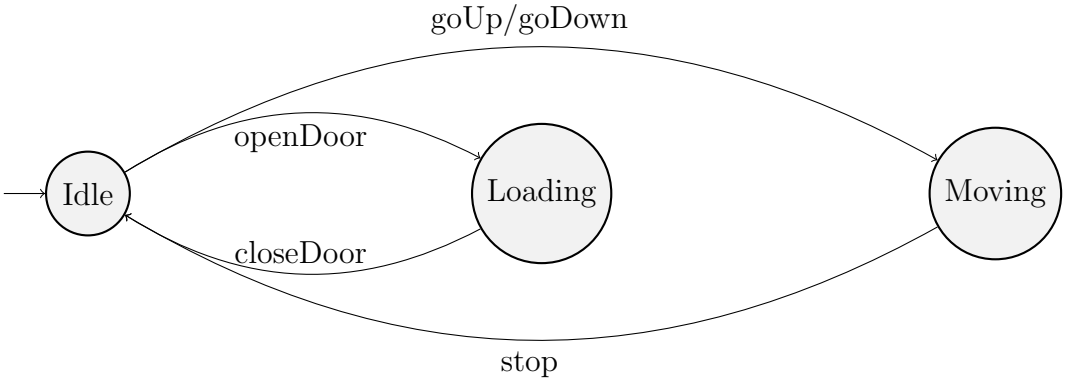


Figure 1: Basic FSA

1.2 Regular Expressions

The lift system was also expressed via regular expressions. This RE can be seen in listing 2. The RE starts of with a series of *openDoor* and *closeDoor* actions which must occur consecutively. This is done so that when the lift moves to the next stage in the RE, the door is always closed, since the lift cannot move with its door open.

Next, the RE verifiers that the lift goes an arbitrary number of steps up or down (depending on what the user decided) followed by a stop. The *stop* is there to make sure the lift is not moving when the progress move back to the open/close door part of the expression.

An example run of the elevator is: `openDoor > closeDoor > goUp > goUp > stop > openDoor > closeDoor > openDoor > closeDoor > goDown > stop`. This run through is correct according to the RE provided.

Listing 2: RE definition

```
((openDoor.closeDoor)* (goUp* + goDown*).stop )*
```

1.3 Timed Automata

For the timed automata (figure 2, the same state machine above was used with some clocks and states added to verify time-constrained events. In the formal definition of the automaton, the transition functions where modified to accommodate such changes. The formal specification can be seen in listing 3. The transition functions follow the format `[from, to, letter, clocks`

`reseted, condition]`.

The first time property that is handled says that "upon a request, after the door closes, the elevator starts moving in less than 3 seconds". When adapted to the provided automaton, this translates to "when the lift is *Idle* (because a floor button was pressed so the door closed), it goes either up or down in less than 3 seconds". This is verified by clock x.

The second time property expresses that "after the door has been open for 3 seconds, it closes automatically". This implies that when the lift has its doors opened i.e. its in the *Loading* state, it must go to the *Idle* in less than 3 seconds. This is because if a user presses the button in less than 3 seconds, the door will close and if he does not press any button, the door closes automatically. This is verified by clock y.

The two new states are `Requested` and `LoadingRequested`. A new event, `requested` is also added to the alphabet. These new additions concern the summon button present at each floor. The `Requested` state signifies when the lift a floor was requested while the lift was moving or idle. The state `LoadingRequested` is the same except that before the request, the lift was in the `Loading` state.

The state `Requested` was added for the time property that states that the lift must start moving in less than 3 seconds i.e., the time property handles by clock x. Therefore, every transition that leads to the `Requested` state resets clock x. Meanwhile, `LoadingRequested`) is needed for the close-door time property handled by clock y.

Listing 3: TA definition

```
M = {
{openDoor, closeDoor, stop, goUp, goDown},
{Idle, Loading, Moving},
{idle},
{x, y},
{
    [ Idle, Loading, openDoor, {y:=0}, true ]

    [ Idle, Requested, request, {x:=0}, true ]

    [ Loading, Idle, closeDoor, {x:=0}, y<3 ]
```

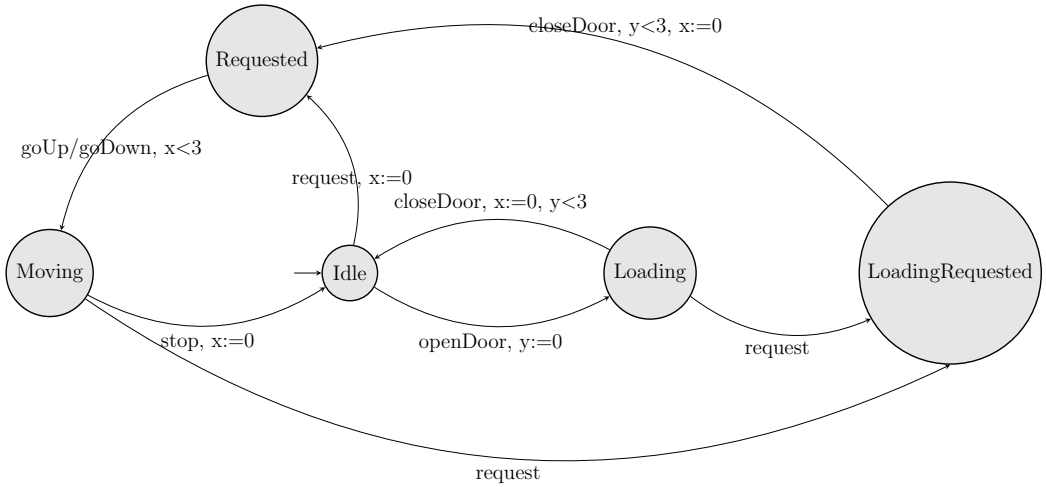


Figure 2: TA figure

```
        [Loading, LoadingRequested, request, {}, true]

        [ Moving, Idle, stop, {x:=0}, true ]

        [ LoadingRequested, Requested, closeDoor, {x:=0}, y
           <3 ]

        [ Requested, Moving, goUp/goDown, {}, x<3 ]
}
}
```

1.4 Duration Calculus

For the DC specifications, two formulas where defined, one for each time constrain. The first time constrain states that: "upon a request, after the door closes, the elevator starts moving in less than 3 seconds". This is expressed by the formula:

$$\Box[Idle];[Loading];[Moving] \wedge [idle];[Loading];[Idle] \Rightarrow \int loading < 3$$

The square means that it is checking for all subintervals. Then the pattern the formula is looking for is *Idle*, followed my *Loading*, followed by *Moving*. Then, the expression verifies that the time of loading is less than 3.

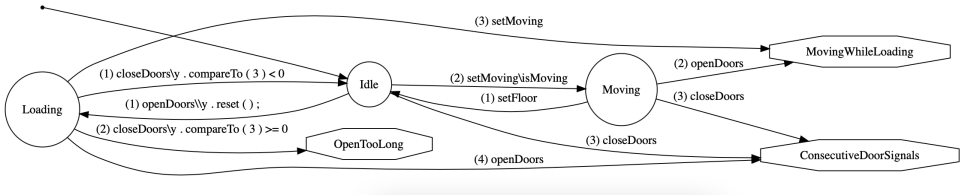


Figure 3: Generated automaton for LiftOpenTimeProperty

The second time constrain says that: after the door has been open for 3 seconds, it closes automatically". The formula describing this is as follows:

$$\Box[doorOpen]; [\neg doorOpen] \Rightarrow \int doorOpen < 3$$

The formula looks for the pattern of an open door followed by a closed door and then verifies that the door was open for less than 3 seconds.

2 Runtime Verification

For runtime verification (RV), the Larva tool was used. The automata applied where directly derived from the automata discussed in section 1.3. However, it was not implemented as a single automaton in order to check for properties individually. On top of that, several bas states where added since the ones shown do not include them to make it visually easier. The full automata including the bad states where generated and can be seen later on.

The first property investigated was the door close property (seen visually in figure 3) . First off, to make the investigation simpler, a single lift was considered. This is because the automaton designed only considers one lift. The Java code was analysed to find the methods that correspond with the designed events. These methods were present in *Lift.java*. The methods and their corresponding events is shown in table 1.

This property does not only check for the time property. It also checks for temporal properties such as receiving a `openDoor` signal while already open. It also checks for lift movement while the lift is open.

To add support for multiple lift, the `FOREACH` Larva keyword was used to iterate over lift objects. All the event methods resided in the class *Lift.java*. This meant that adding multiple lift support was done by binding the Lift

[1]*isMoving parameter must be true

Method	Event
Lift.setMoving(boolean isMoving)	goUp/goDown *
Lift.closeDoors()	closeDoor
Lift.openDoors()	openDoor
Lift.setFloor()	stop

Table 1: LiftOpenTimeProperty events table

1

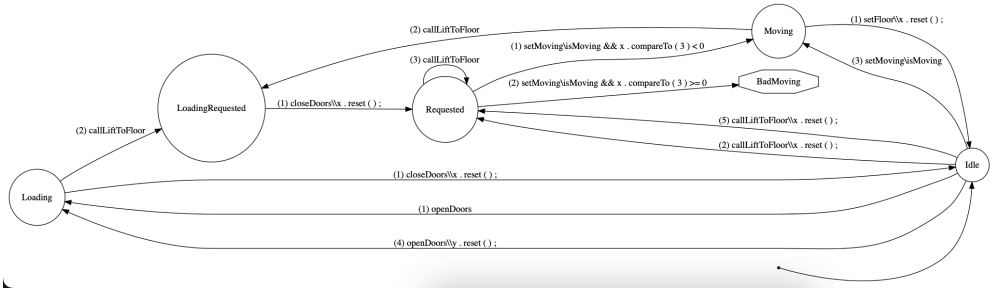


Figure 4: Generated automaton for StartMovingTimeProperty

object in the event declarations and then assigning it to the `FOREACH` object using the `WHERE` keyword.

Listing 4: Multiple lift support for LiftOpenProperty

```
setMoving(isMoving) = {Lift l.setMoving(boolean
    isMoving)} WHERE {lift = l;}
closeDoors() = {Lift l.closeDoors()} WHERE {lift = l;}
openDoors() = {Lift l.openDoors()} WHERE {lift = l;}
setFloor() = {Lift l.setFloor(*)} WHERE {lift = l;}
```

The second property (visual representation in figure 4) is called `StartMovingTimeProperty` and verifies that the lift start moving in less than 3 seconds after the door closes following a request. A new event was added for the summon request action. The method that corresponds to this event is `callLiftToFloor`.

This method provided some difficulty when switching to adding support for multiple lift verification. The reason being that this method does not reside in the *Lift.java* class. It resided in the *JavaController.java* class. When `callLiftToFloor` is called, it performs some calculations and then proceeds to execute `moveLift` with the chosen lift. To find out which lift was called via Larva, the source needed to be edited. The change done was very minimal and does not impact the design of the lift system at all.

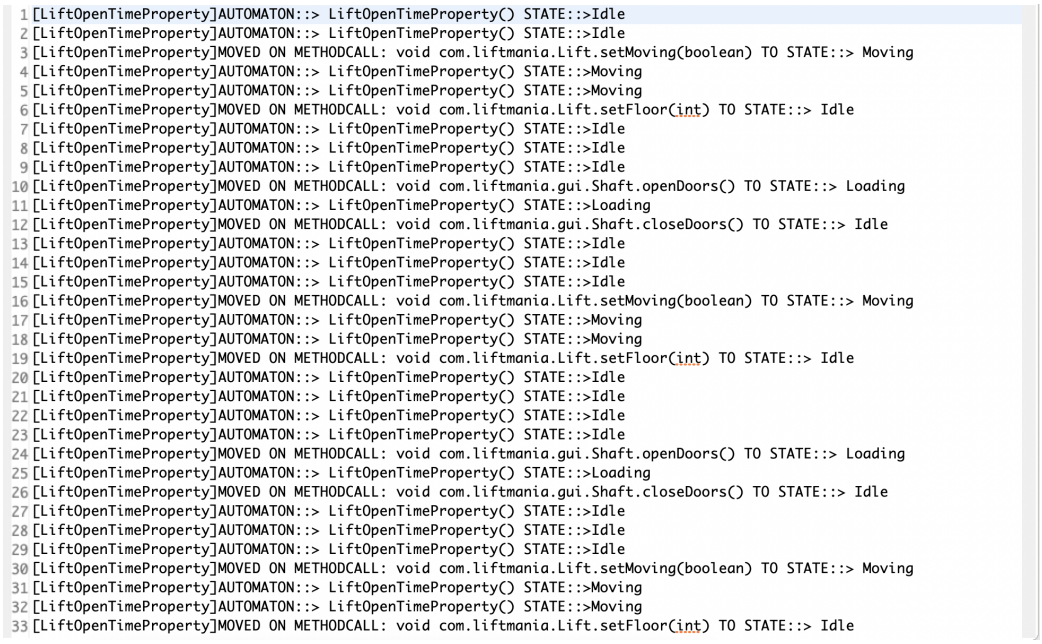


Figure 5: `LiftOpenTimeProperty` verified

The `callLiftToFloor` method signature was changed to return a lift object instead of void. This is then used by Larva by making use of the `uponReturning` keyword when declaring the event. The whole event declaration is as follows:
`callLiftToFloor(Lift l) = *callLiftToFloor(*)uponReturning(l) where lift = l;`, where 'lift' is the iterated object declared in the FOREACH loop.

2.1 Testing

In the tests shown, one property at a time was tested to make the outputs more legible. Figure 5 shows the property `LiftOpenTimeProperty` running as should. several buttons were pressed and the lift always moved as specified.

For the second test shown in figure 6, time taken to close the door automatically was changed from the java source code in file *Shaft.java*. The figure shows that verification failed because the door was closing after 3 seconds.

Referirng to figure 7 the third test shown, displays the verification being conducted on several (three) lifts. Identifiers are used to differentiate the multiple lifts being verified.

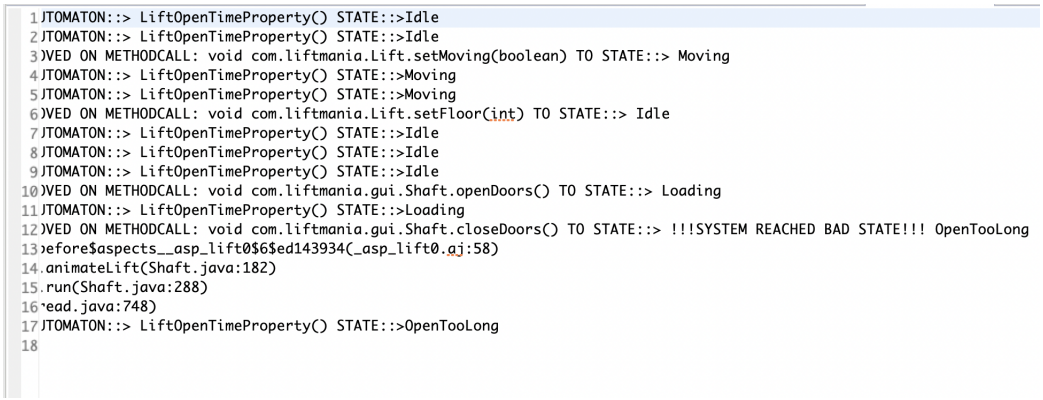


Figure 6: `LiftOpenTimeProperty` failed

Figure 8 shows the fourth documented test. This test shows the verification of the `StartMovingTimeProperty` working on more than one lift. To make the output more clear, the `LiftOpenTimeProperty` was removed from the script to focus only on the property discussed.

A failure was also induced for this property. The `animateLift` method in *Shaft.java* was edited to wait three seconds before starting to move the lift. The added code can be seen in figure 9. As expected, this caused the verification to fail. The output can be seen in figure 10

The final test showcased in this section can be seen in figure 11. This shows all the properties being run at once for several lifts.

2.2 Larva script improvements

The script developed is not perfect. One flaw with the system is that requests are only summon requests. This means that only buttons on floors are verified. The reason for this decision lies in the java source code for the Lift system. As can be seen in figure 12 was a button is pressed, it is registered as a `move` action. As opposed to using a function similar to summon requests where `callLiftToFloor` is used.

One possible way to circumvent this problem would have been to iterate over `Shaft` objects rather than `Lift` object as the script does currently. This would have most likely work since each shaft corresponds to a single that resides in it. The *Shaft.java* class also contains `openDoors()` and `closeDoors()` methods similar to the *Lift.java* class. Movement events

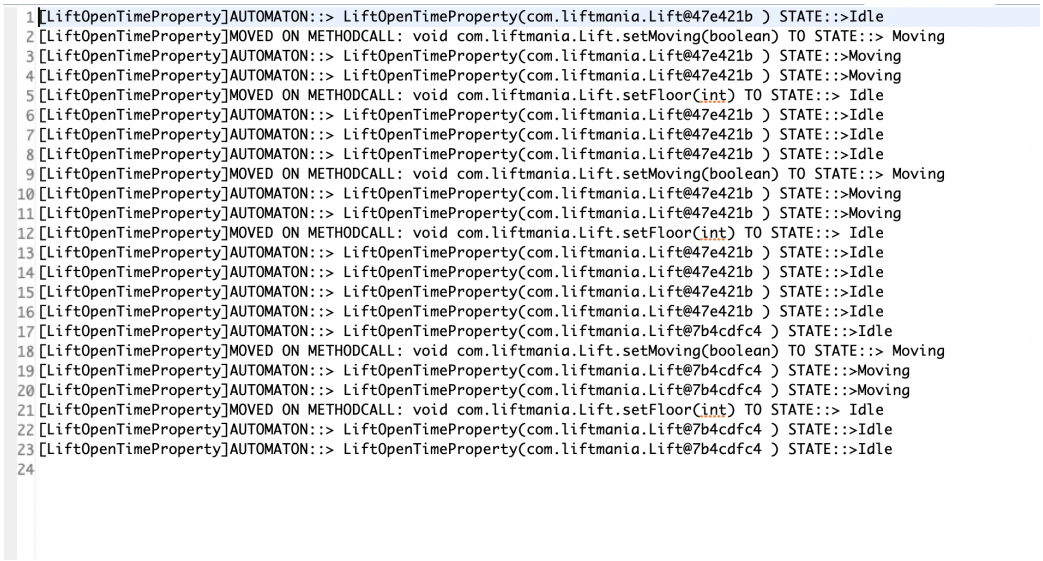


Figure 7: `LiftOpenTimeProperty` on several lifts

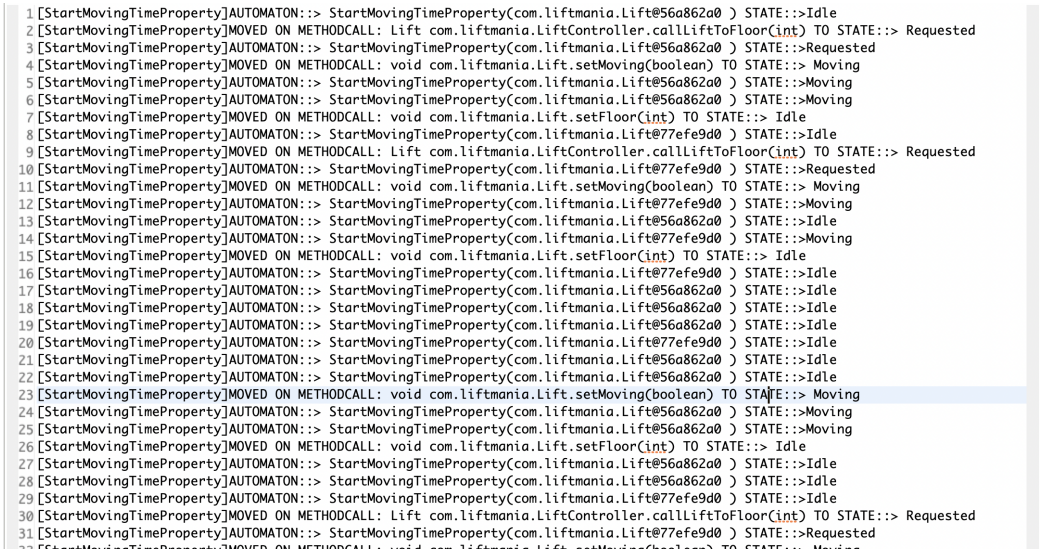


Figure 8: `StartMovingTimeProperty` on several lifts

```
153     public void animateLift(int toFloor) {
154
155         //Wait three seconds after call before moving
156         // (should cause verification)
157         try {
158             Thread.sleep(3000);
159         } catch (Exception e) {}
160
161         //Update lift state
162         lift.setMoving(true);
163
164         int fromFloor = lift.getFloor();
165         setLiftFloor(fromFloor);
166         lift.setMoving(true);
167
168         if (toFloor > fromFloor) {
169             for (int i = fromFloor; i < toFloor; i++) {
170                 animateUp(i);
171                 lift.setFloor(i);
172             }
```

Figure 9: Added code to stop for three seconds before starting to move lift in *Shaft.java*

```
1 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>Idle
2 ingTimeProperty]MOVED ON METHODCALL: Lift com.liftmania.LiftController.callLiftToFloor(int) TO STATE::> Requested
3 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>Requested
4 ift took too long to start moving
5 ingTimeProperty]MOVED ON METHODCALL: void com.liftmania.Lift.setMoving(boolean) TO STATE::> !!!SYSTEM REACHED BAD STATE!!! BadMoving
6 asp_lift1.ajc$before$aspects__asp_lift1$2$74a464b4(_asp_lift1.aj:25)
7 ania.gui.Shaft.animateLift(Shaft.java:162)
8 ania.gui.Shaft.run(Shaft.java:295)
9 .Thread.run(Thread.java:748)
10 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
11 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
12 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
13 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
14 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
15 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
16 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
17 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
18 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
19 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
20 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
21 ingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@4c4251d4 ) STATE::>BadMoving
22
```

Figure 10: `StartMovingTimeProperty` failure

```
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]MOVED ON METHODCALL: Lift com.liftmania.LiftController.callLiftToFloor(int) TO STATE::> Requested
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[LiftOpenTimeProperty]MOVED ON METHODCALL: void com.liftmania.Lift.setMoving(boolean) TO STATE::> Moving
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Requested
[StartMovingTimeProperty]MOVED ON METHODCALL: void com.liftmania.Lift.setMoving(boolean) TO STATE::> Moving
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Moving
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Moving
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Moving
[LiftOpenTimeProperty]MOVED ON METHODCALL: void com.liftmania.Lift.setFloor(int) TO STATE::> Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Moving
[StartMovingTimeProperty]MOVED ON METHODCALL: void com.liftmania.Lift.setFloor(int) TO STATE::> Idle
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[LiftOpenTimeProperty]AUTOMATON::> LiftOpenTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
[StartMovingTimeProperty]AUTOMATON::> StartMovingTimeProperty(com.liftmania.Lift@e3a71a4 ) STATE::>Idle
```

Figure 11: All properties

```java
        // Create floor buttons
        JPanel buttonsPanel = new JPanel(new GridLayout(1, numFloors));
        for (int j = 0; j < numFloors; j++) {
            JButton btn = new JButton(Integer.toString(j));
            btn.setActionCommand("move" + "," + Integer.toString(lift.getId()) + ","
                    + Integer.toString(j));
            btn.addActionListener(visualiser);
            buttonsPanel.add(btn);
        }

        add(buttonsPanel, BorderLayout.SOUTH);
```

Figure 12: Floor request button code

would have been recognised by the `animateUp()` and `animateDown()` present in *Shaft.java*. Finally, stopping would have been recognised by the `animationPause()` method. Iteration over the `Shaft` objects would enable the script to track which shaft floor buttons were pressed.

3 Model-Based Testing

For model based testing ModelJUnit was used. Importing the project with maven provided several difficulties and wasted efforts. As a result, the ModelJUnit jar file was used. Also, the Intellij IDE was used for this task instead of eclipse.

3.1 BasicLiftModel

The previous automata were used for Model-based testing. Using this model, internal variables and states are updated and compared to the system under

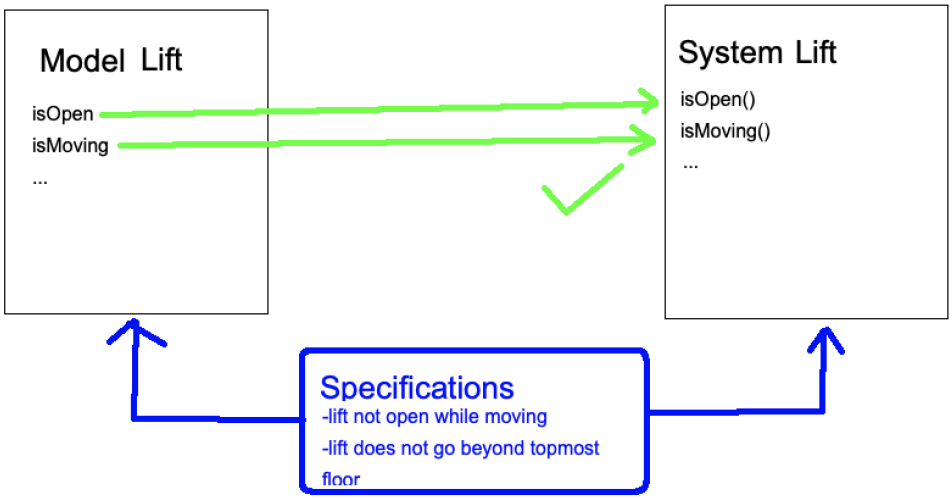


Figure 13: Model based testing diagram

test (SUT). This workflow can be seen visually in figure 13. This model also checks for temporal some temporal properties for example trying to open a lift that is already open. This is done via assertions at the start of the method to check the current state.

The first model implemented is based on the automaton seen in 1. The variables that the model keeps track of are shown in figure 14. When the flow enters a state in the model, these variables are updated and matched with the SUT's variables via the assert method. Two types of assert methods are used in this model:
`Assert.assertEquals("Model does not match SUT: Moving", isMoving, lift.isMoving());`
`Assert.assertFalse("Moved with door open", lift.isOpen());` The first one is comparing a variable of the model with a variable of the SUT while the second is comparing a verifying that a variable of SUT is in accordance with the current state.

The tests provided full coverage for this model. This can be seen in figure 15. The Lift system passes the test. The *Lift.java* was then edited with faulty code. The code added was a `openDoors()` call in the `moveLift()` method. This caused the test to fail as can be seen in figure 16.

```java
public class BasicLiftModel implements TimedFsmModel {

    private Lift lift = new Lift( id: 0);
    private LiftStates liftState = LiftStates.IDLE;
    private boolean isOpen = false;
    private boolean isMoving = false;
    private int currentFloor = 0;
    private int lastFloor = 0;
    private int lastDifferentFloor = 0;
```

Figure 14: BasicLiftModel variables

```
action coverage: 4/4
state coverage: 3/3
transition coverage: 4/4
transition-pair coverage: 6/6
```

Figure 15: BaseLiftModel test coverage

```java
public void setMoving(boolean moving) {
    this.moving = moving;
    openDoors();
}
```

(a) Code edit

```
/Library/Java/JavaVirtualMachines/openjdk8-temurin/Contents/Home/bin/java ...
Lift is closed before
Lift is closed before
done 0:(IDLE, openDoor, LOADING)
done Random reset(true)
FAILURE: failure in action moveLift from state IDLE due to junit.framework.AssertionFailedError: Moved with door open
```

(b) Output

Figure 16: BasicLiftModel test failure

3.2 LiftTimeModel

To test time properties another model was created. The provided lift system did not include a clock class so time could not be mocked easily by ModelJUnit. As a result, inline java code had to be used, for example `Thread.sleep`.

This part was not fully implemented. Due to problems found with driving the `LiftController` class. The aim was to make the SUT run `LiftController.moveLift(...)` and simulate it from the model side to match the correct times for example verifying that the door stays open for less than 3 seconds. However a problem arose where the method `LiftController.moveLift(...)` was not taking the necessary time to run. It should have taken the function at least 2000 milliseconds since it contains a `thread.sleep` however upon debugging it was shown to take only 2 milliseconds maximum. The code to time the function can be seen in the `moveLift()` action in *LiftTimeModel* class or in listing 5.

Listing 5: FSA definition tabsize

```
1  liftState = LiftStates.MOVING;
2          startMoving = System.currentTimeMillis();
3
4
5          if (currentFloor == 0 ) {
6              liftController.moveLift(0, 5);
7          } else {
8              liftController.moveLift(0, 0);
9          }
10
11         System.out.print("Time taken: ");
12         System.out.println(System.currentTimeMillis() -
               startMoving);
```

4 Runtime Verification and Testing

For this task, executing RV and model-based testing at the same time was required. This means that this task is combining the artefacts of task 2 and task 3 together. The eclipse IDE needed to be used again so that the LARVA plugin could be used. The two tasks (2 and 3) contain a different structure. This is because the provided project structure by the respective IDE was used. This resulted in task 4 containing a combination of the structure of the two tasks. Table 2 shows the package structure of each task to make

Task	Packages
Task 2	com.liftmania
Task 3	main.java.com.liftmania
	test.java.com.liftmania
Task 4	main.java.com.liftmania
	test.java.com.liftmania

Table 2: LiftOpenTimeProperty events table

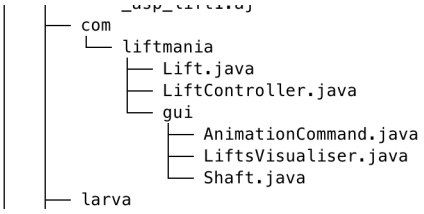
things clearer. Similarly, the directory trees can be seen in figure 17.

One difference between RV and testing is that in runtime verification, the user/tester controls the system as usual while RV is done in the background independent of what actions the user does. Meanwhile, in testing, the system is driven by the tests i.e., the tests decide what flow the system follows. In this task, RV is used but it is executed on the system driven by the tests. For example the tests run the method `moveLift()` and the RV 'catches' this and change the state accordingly. Since the Larva script and Model JUnit tests are based on the same automata, the same states should be visited. This can be seen figuratively in figure 18

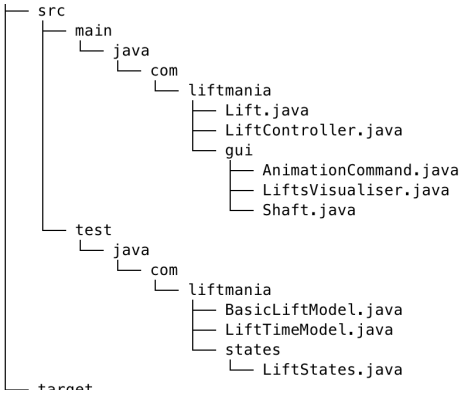
To test this setup, some faults where introduced to investigate wether larva ends in a bad state. In the first test shown in this section, the `StartMovingTimeProperty` was removed to make analysis of the Larva output less simpler. Then, the Model JUnit method that calls the `closeDoors()` was modified to wait 4 seconds before proceeding to close the doors. As expected, this resulted in a bad state being reached. The modified code and the result can be seen in figure 19.

The second test discussed in this section can be seen in figure 20. In this test the ModelJunit code was edited to open the doors before moving. Assertions that checked for this bug were commented out so that the program does not stop running. The Larva script successfully detected the errors.

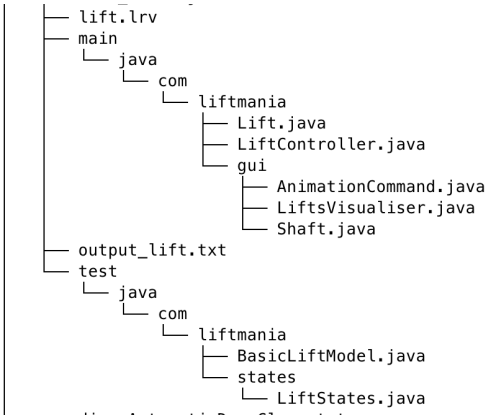
Overall this setup really makes sense since if one technique can better check for something than the other, when implemented together they check for all the setups. For example, checking for timed events proved to be easier in java with the help of implemented clocks. On top of that, tests help to check for all possible scenarios without having to do them manually. This means that the tests can be done unattended. Furthermore, the purpose of



(a) Task 2

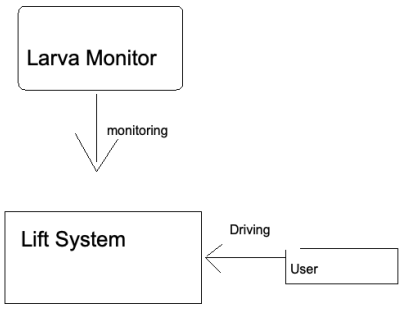


(b) Task 3

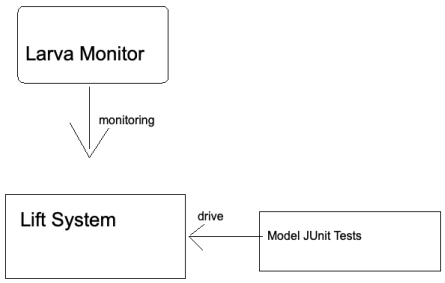


(c) Task 4

Figure 17: Directory trees for each task

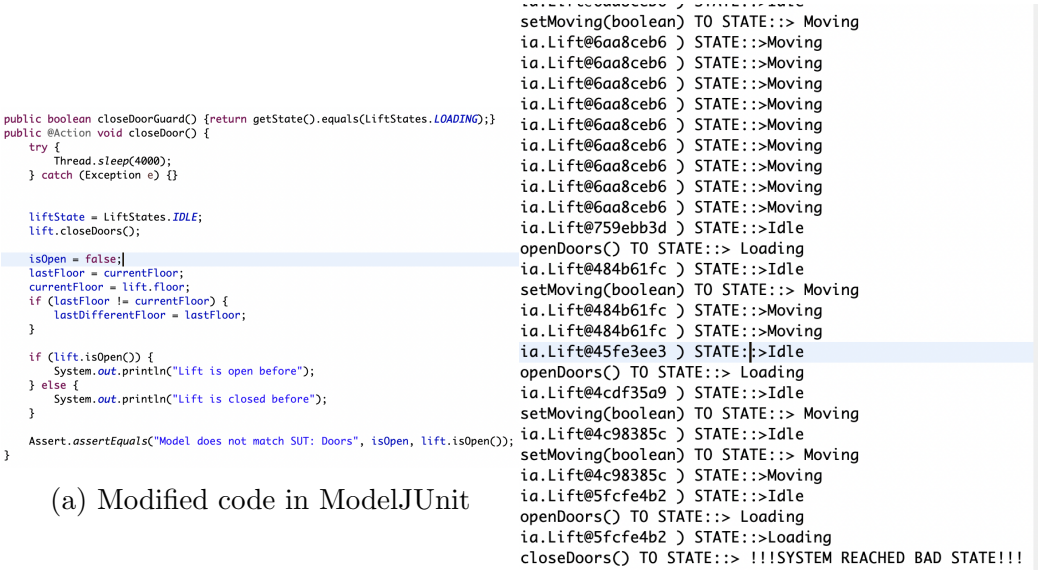


(a) Task 2



(b) Task 4

Figure 18: Larva diagram for each task



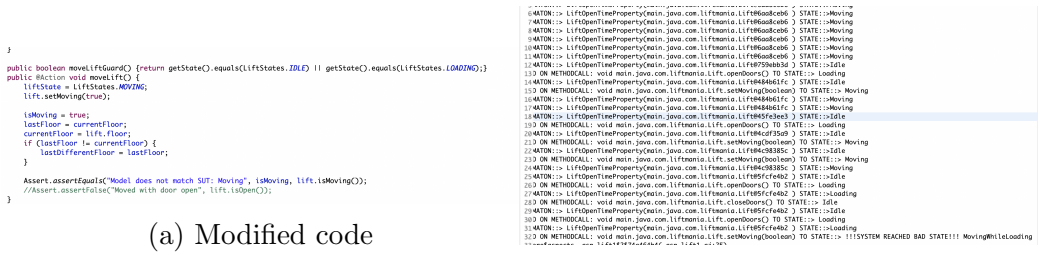
```
public boolean closeDoorGuard() {return getState().equals(LiftStates.LOADING);}
public @Action void closeDoor() {
    try {
        Thread.sleep(4000);
    } catch (Exception e) {}


    liftState = LiftStates.IDLE;
    lift.closeDoors();

    isOpen = false;
    lastFloor = currentFloor;
    currentFloor = lift.floor;
    if (lastFloor != currentFloor) {
        lastDifferentFloor = lastFloor;
    }

    if (lift.isOpen()) {
        System.out.println("Lift is open before");
    } else {
        System.out.println("Lift is closed before");
    }

    Assert.assertEquals("Model does not match SUT: Doors", isOpen, lift.isOpen());
}
```

(a) Modified code in ModelJUnit

```
setMoving(boolean) TO STATE::> Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@6aa8ceb6 ) STATE::>Moving
ia.Lift@759ebb3d ) STATE::>Idle
openDoors() TO STATE::> Loading
ia.Lift@484b61fc ) STATE::>Idle
setMoving(boolean) TO STATE::> Moving
ia.Lift@484b61fc ) STATE::>Moving
ia.Lift@484b61fc ) STATE::>Moving
ia.Lift@45fe3ee3 ) STATE::>Idle
openDoors() TO STATE::> Loading
ia.Lift@4cdf35a9 ) STATE::>Idle
setMoving(boolean) TO STATE::> Moving
ia.Lift@4c98385c ) STATE::>Idle
setMoving(boolean) TO STATE::> Moving
ia.Lift@4c98385c ) STATE::>Moving
ia.Lift@5fcfe4b2 ) STATE::>Idle
openDoors() TO STATE::> Loading
ia.Lift@5fcfe4b2 ) STATE::>Loading
closeDoors() TO STATE::> !!!SYSTEM REACHED BAD STATE!!!
```

(b) Larva output

Figure 19: RV reaches bad state from test example 1



```
}

public boolean moveLiftGuard() {return getState().equals(LiftStates.IDLE) || getState().equals(LiftStates.LOADING);}
public @Action void moveLift() {
    liftState = LiftStates.MOVING;
    lift.setMoving(true);

    isMoving = true;
    lastFloor = currentFloor;
    currentFloor = lift.floor;
    if (lastFloor != currentFloor) {
        lastDifferentFloor = lastFloor;
    }

    Assert.assertEquals("Model does not match SUT: Moving", isMoving, lift.isMoving());
    //Assert.assertFalse("Moved with door open", lift.isOpen());
}
```

(a) Modified code

```
4ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@6aa8ceb6 ) STATE::>Moving
7ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@6aa8ceb6 ) STATE::>Moving
8ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@6aa8ceb6 ) STATE::>Moving
9ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@6aa8ceb6 ) STATE::>Moving
10ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@6aa8ceb6 ) STATE::>Moving
11ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@6aa8ceb6 ) STATE::>Moving
12ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@759ebb3d ) STATE::>Idle
13D ON METHODCALL: void main.java.com.liftmania.Lift.openDoors() TO STATE::> Loading
14ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@484b61fc ) STATE::>Idle
15D ON METHODCALL: void main.java.com.liftmania.Lift.setMoving(boolean) TO STATE::> Moving
16ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@484b61fc ) STATE::>Moving
17ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@484b61fc ) STATE::>Moving
18ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@45fe3ee3 ) STATE::>Idle
19D ON METHODCALL: void main.java.com.liftmania.Lift.openDoors() TO STATE::> Loading
20ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@4cdf35a9 ) STATE::>Idle
21D ON METHODCALL: void main.java.com.liftmania.Lift.setMoving(boolean) TO STATE::> Moving
22ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@4c98385c ) STATE::>Idle
23D ON METHODCALL: void main.java.com.liftmania.Lift.setMoving(boolean) TO STATE::> Moving
24ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@4c98385c ) STATE::>Moving
25ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@5fcfe4b2 ) STATE::>Idle
26D ON METHODCALL: void main.java.com.liftmania.Lift.openDoors() TO STATE::> Loading
27ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@5fcfe4b2 ) STATE::>Loading
28D ON METHODCALL: void main.java.com.liftmania.Lift.closeDoors() TO STATE::> Idle
29ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@5fcfe4b2 ) STATE::>Idle
30D ON METHODCALL: void main.java.com.liftmania.Lift.openDoors() TO STATE::> Loading
31ATON::> LiftOpenTimeProperty(main.java.com.liftmania.Lift@5fcfe4b2 ) STATE::>Loading
32D ON METHODCALL: void main.java.com.liftmania.Lift.setMoving(boolean) TO STATE::> !!!SYSTEM REACHED BAD STATE!!! MovingWhileLoading
```

(b) Larva output

Figure 20: RV reaches bad state from test example 2

RV is to be deployed with systems at runtime, where real live users will be using (driving) the system instead of the programmer. Therefore, RV still needs to be implemented for the actual runtime of the system, considering that implementing RV for the model-testing was not to complicated, it would be an extremely smart and efficient move to implement the RV for the model-testing as well.