CPS3233 – Verification Techniques

Part 3 – Model-Based Testing

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

Table of Contents

1 – Introduction

Due to time restrictions, the model was only successfully built to handle one lift. However, since most of the properties apply to individual lifts, many of them can still be checked. Three properties were avoided:

- Temporal property 1 cannot be checked because it requires a model-checking tool to be used.
- Temporal property 3 could not be *fully* checked, since it involves all of the lifts in the system.
- Temporal property 4 was not checked because of the difficulty of detecting all door open/close signals.
- Temporal property 6 cannot be checked because no button-holding functionality is implemented.

2 – Further Changes to the System

Building on top of the system that was used for the runtime verification part of the assignment, some further changes were applied to the system:

- First of all, the startClosingDoors() method in the Lift class now actually sets a doorsClosing variable to *true*, which is then re-set to *false* once the doors close, so that the model can detect these changes.
- Secondly, the `lift.setMoving(false);` statement in the animateLift(...) method in the Shaft class was moved to before the doors are opened, since the lift would have stopped moving at that point.
- Lastly, `JFrame.EXIT_ON_CLOSE` in LiftVisualiser was changed to `JFrame.DISPOSE_ON_CLOSE` so that the GUI can be closed without stopping the whole program when the ModelJUnit model is reset.

3 – The Implemented Model

Figure 1 illustrates the model that has been implemented using ModelJUnit. The four states track whether the lift door is open or closed and whether an unserviced request or summon exists (wREQ) or not (IDLE).

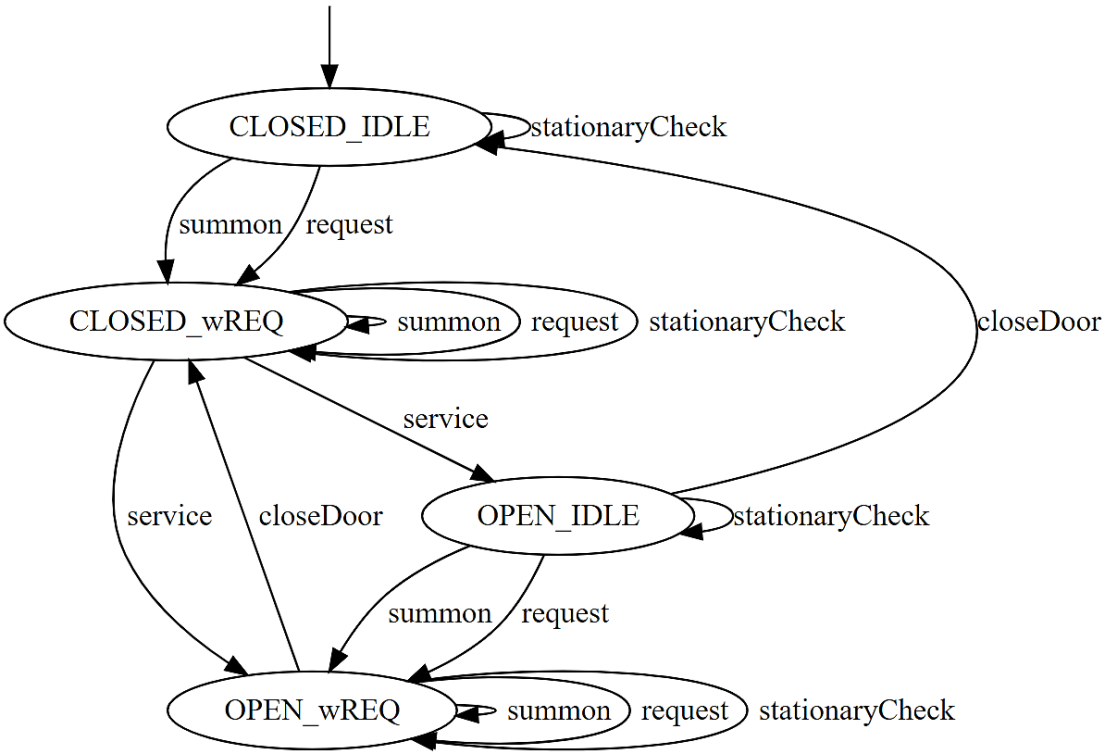

Figure 1 – The Implemented Model

The lift starts off with its door closed and without any summons or requests. A summon or request is allowed to happen from any state in the automaton. If the lift was in an idle state, it moves to the appropriate *wREQ* state, while if the lift is already in a *wREQ* state, it does not move to a new state.

Summons are only allowed to occur if $|summons| < |lifts|$ since if the number of summons exceeds the number of lifts, the system crashes. For requests, on the other hand, a limit of $2 \times |floors|$ was set, so that the model does not get overloaded with a lot of requests.

The stationaryCheck action requires that there are no unserviced summons/requests or that the lift door is open. This action does not affect the current state and simply checks that the lift is currently not moving.

The remaining actions include the service action and closeDoor action. The former is meant to consume the earliest request or summon that has been generated whereas the latter is meant to track the closing of the lift door after a request or summon has been serviced, with some assertions in the process of doing so.

3.1 – The Request Action

The request action starts off by checking that if the lift currently has no tasks. This is targeting **invariant 3**, which requires that the elevator never moves unless a button press occurs which has not yet been serviced.

A random floor is then selected for a request and some information about the request is collected:

- **requests**: list of unserviced requests;
- **setIgnoreValue(isSummon, floor)**: method that adds a true or false to a *shouldIgnore* list corresponding to whether the request (or summon) should be ignored, in accordance with **temporal property 2**.
- **taskIsSummon**: list of booleans corresponding to the ordered unserviced requests and summons.

The *sut* (an instance of LiftController) is then given the request so that the lift is instructed to move. Finally, the model's state is updated according to whether the door was previously open or closed.

```
@Action
public void request() {

    if (haveNoTasks()) {
        Assert.assertFalse(sut.lifts[LIFT0].moving); // Invariant 3
    }

    // Select random floor, collect information, and perform summon in SUT
    final int randFloor = ThreadLocalRandom.current().nextInt(0, NO_OF_FLOORS);
    requests.add(randFloor);
    setIgnoreValue(true, randFloor);
    taskIsSummon.add(false);
    sut.moveLift(LIFT0, randFloor);

    state = (state == CLOSED_IDLE || state == CLOSED_wREQ) ? CLOSED_wREQ : OPEN_wREQ;
}
```

Listing 1 – The Request Action

3.2 – The Summon Action

Similar to the request, the summon action also performs a check for **invariant 3** and proceeds by selecting a random floor and collecting some information. This is followed by the invocation of the appropriate *sut* method (in this case, the *callLiftToFloor* method). The action method now proceeds to perform two assertions:

- For **temporal property 3**: if, now that a summon has occurred, the lift is currently moving at the floor from which the summon occurred, the lift's doors should open for servicing after a small delay.
- For **temporal property 5**: if, now that a summon has occurred, the lift is currently at the summon's floor and its doors are closing, the lift's doors should open for servicing after a small delay.

Since both of these properties are expected to be violated, if the assertion passes, the testing is forced to stop. Finally, the model's state is updated according to whether the door was previously open or closed.

```
@Action
public void summon() throws InterruptedException {

    if (haveNoTasks()) {
        Assert.assertFalse(sut.lifts[LIFT0].moving); // Invariant 3
    }

    // Select random floor, collect information, and perform summon in SUT
    final int randFloor = ThreadLocalRandom.current().nextInt(0, NO_OF_FLOORS);
    summons.add(randFloor);
    setIgnoreValue(false, randFloor);
    taskIsSummon.add(true);
    sut.callLiftToFloor(randFloor);

    if (randFloor == sut.lifts[LIFT0].floor && sut.lifts[LIFT0].moving) {
        Thread.sleep(500);
        Assert.assertTrue(sut.lifts[LIFT0].doorsOpen); // Temporal 3 (expected to fail)
        Assert.fail("Temporal property 3 did not fail."); // Expected to fail and didn't
    }

    if (randFloor == sut.lifts[LIFT0].floor && sut.lifts[LIFT0].doorsClosing) {
        Thread.sleep(500);
        Assert.assertTrue(sut.lifts[LIFT0].doorsOpen); // Temporal 5 (expected to fail)
        Assert.fail("Temporal property 5 did not fail."); // Expected to fail and didn't
    }

    state = (state == CLOSED_IDLE || state == CLOSED_wREQ) ? CLOSED_wREQ : OPEN_wREQ;
}
```

Listing 2 – The Summon Action

3.3 – The Service Action

The discussion will now move to the service action, which is meant to consume the earliest summon or request that has occurred. The method first starts by checking whether the earliest task is a summon or request and obtains the expected floor from the appropriate list, i.e. the summons list or requests list. Since *shouldIgnore*s are only meant to check future summons or requests, the current *shouldIgnore* can be discarded.

Next, a downside of the model becomes apparent. Since the service action can occur after a delay, rather than immediately after a summon/request, this means that the lift might have already reached its destination, and possible opened and closed its door. Thus, a condition checks whether the lift has reached its destination.

- If the lift has not reached its destination, a loop is used to wait for this to happen. In each iteration of the loop, assertions are made to check for the violation of **invariant 4** or **invariant 5**. For the former, the lift should never stop between floors. For the latter, the lift door should not open between floors. Once the loop terminates (i.e. doors open), an approximate time that the doors opened is stored.

- If the lift has already reached its destination, the above checks cannot be performed. Additionally, since the door might have already opened, the approximate time of open may be too inaccurate.

In any case, two assertions check for the violation of **invariant 2**. These respectively check whether the floor is at the top or bottom floor and is still moving. Since the lift has just reached a new floor, this would indicate that the lift is still trying to move in the direction of arrival, which is not allowed for the top and bottom floors.

Lastly, the model's state is updated according to whether the lift has any more tasks. At this point, the lift is still assumed to have its doors open. The model's doors will change to closed through the *closeDoor* action.

```
@Action
public void service() throws InterruptedException {

    final boolean isSummon = taskIsSummon.remove(0);
    final int toFloor = isSummon ? summons.remove(0) : requests.remove(0);

    // Discard shouldIgnore for current task (since we only need the next)
    shouldIgnore.remove(0);

    if (sut.lifts[LIFT0].floor != toFloor) {
        // Wait for lift to arrive at its destination
        while (sut.lifts[LIFT0].floor != toFloor) {
            Assert.assertFalse(sut.lifts[LIFT0].betweenFloors
                && !sut.lifts[LIFT0].moving); // Invariant 4
            Assert.assertFalse(sut.lifts[LIFT0].betweenFloors
                && sut.lifts[LIFT0].doorsOpen); // Invariant 5
            Thread.sleep(200);
        }

        // Calculate approximate time that doors opened
        approxTimeOfOpen = System.currentTimeMillis();
    } else {
        // Lift might have already opened and closed
        // its door so we should skip the above checks
        approxTimeOfOpen = -1; // Dummy approximation
    }

    // Lift has arrived at its destination
    Assert.assertFalse(sut.lifts[LIFT0].floor == TOP_FLOOR
        && sut.lifts[LIFT0].moving); // Invariant 2 (top floor)
    Assert.assertFalse(sut.lifts[LIFT0].floor == BOTTOM_FLOOR
        && sut.lifts[LIFT0].moving); // Invariant 2 (bottom floor)

    // Set state
    state = haveNoTasks() ? OPEN_IDLE : OPEN_wREQ;
}
```

Listing 3 – The Service Action

3.4 – The Close Door Action

The last major action that remains to be discussed is the door-closing action. Similar to the service action, this action involves a condition that checks whether the action has already been accomplished in the *sut*. In this case, if the doors are already closed, the checks in this action are not performed.

If the doors are open, a loop waits until they close. In each iteration of the loop, for the purpose of checking **invariant 1**, an assertion makes sure that the lift is not moving with its doors open. Once the doors close, an approximation of the time that the doors closed is stored.

Next, unless the opening time approximation is invalid, to check **real-time property 2**, an assertion makes sure that at least 3 seconds ($3000ms$) have elapsed from when the door was initially opened. Next, for the purpose of testing **temporal property 2**, if the next task should be ignored, as indicated by the next value in the *shouldIgnore* list, then the lift door should not immediately re-open at the same floor. An assertion makes sure that the lift's door is closed. Note that the last two properties discussed are both expected to be violated.

The last property to be checked in this action is **real-time property 1**. If the next summon or request is not at the current floor, the lift is expected to start moving before 3 seconds ($3000ms$) have elapsed from the closing of the lift door. If the lift is not moving, a loop waits until it starts moving.

Lastly, the model's state is updated according to whether the lift has any more tasks. At this point, the lift can be assumed to have its doors closed and so the model's doors are change to closed.

```
@Action
public void closeDoor() throws InterruptedException {

    if (sut.lifts[LIFT0].doorsOpen) {

        // Wait for lift to close its doors
        while (sut.lifts[LIFT0].doorsOpen) {
            Assert.assertFalse(sut.lifts[LIFT0].doorsOpen
                && sut.lifts[LIFT0].moving); // Invariant 1
            Thread.sleep(200);
        }

        // Get time that doors closed
        final long approxTimeOfClose = System.currentTimeMillis();

        if (approxTimeOfOpen != -1) {
            Assert.assertTrue(approxTimeOfClose - approxTimeOfOpen > 3000);
            // Real-time property 2 (expected to fail)
            Assert.fail("Real-time property 2 did not fail.");
        }

        // Check that lift does not immediately re-open
        // its doors if the next task should be ignored
        if (!shouldIgnore.isEmpty() && shouldIgnore.get(0)) {
            Thread.sleep(500);
            Assert.assertFalse(sut.lifts[LIFT0].doorsOpen);
            // Temporal 2 (expected to fail)
            Assert.fail("Temporal property 2 did not fail.");
            // Expected to fail and didn't
        }

        // If next floor is not the current floor, check for real-time property 1
        final int nextFloor = nextFloor();
        if (nextFloor != -1 && sut.lifts[LIFT0].floor != nextFloor) {
            while (!sut.lifts[LIFT0].moving) {
                Thread.sleep(200);
            }
            final long approxTimeOfMove = System.currentTimeMillis();
            Assert.assertTrue(approxTimeOfMove - approxTimeOfClose < 3000);
            // Real-time property 1
        }
    } else {
      // Lift might have already closed its door so we should skip the above checks
    }

    // Set state
    state = haveNoTasks() ? CLOSED_IDLE : CLOSED_wREQ;
}
```

Listing 4 – The Close Door Action

3.5 – The Stationary Check Action

The last action to be discussed is the stationary check action. This action can occur from any state, but requires that the lift has no requests or summons and that its door is open. The aim of this action is to target once more **invariants 1 and 3**. By asserting that the lift is not moving, the facts that the lift does not move whilst the door is open and that it does not move unless an unserviced request is present are being checked.

```
@Action
public void stationaryCheck() {
    Assert.assertFalse(sut.lifts[LIFT0].moving); // Invariants 1 and 3
}
```

Listing 5 – The Stationary Check Action

4 – Running the Model

By running the model, the three main outcomes that were observed were:

- Violation of temporal property 2, which is expected;
- Violation of real-time property 2, which is expected;
- The system gets stuck in an infinite loop.

The third outcome unfortunately occurs when the *sut* and the model become too out-of-sync, such that when an infinite loop waits for the lift to reach some state or perform some action, this might never be detected since the state might have been already reached and exited or the action would have already occurred.

Fortunately, the properties that are not expected to be violated were not violated throughout the testing.