CPS3233 – Verification Techniques

Part 4 – Model Checking

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

Table of Contents

Exercise 1 – Heater

To ensure that the heater does not switch on momentarily, a **heaterOnTime** variable keeps track of the time that the heater has been on. It starts as 0 and increments up to an assumed minimum on-time of 3 time units. The **heating** assignment was modified to ensure that if the heater is on (heaterOnTime > 0) and it has not been on for 3 time units (heaterOnTime < 3), it does not turn off. Listing 1 includes (in bold red) the new code:

```
MODULE main
    VAR
        heating: boolean;
        temperature: 10..40;
        lowTempTime: 0..4;
        heaterOnTime: 0..3;
    ASSIGN
        init(lowTempTime) := 0;
        next(lowTempTime) :=
            temperature < 23 ?
                (lowTempTime=4? 4: lowTempTime+1):0;
        init(heaterOnTime) := 0;
        next(heaterOnTime) :=
            heating ?
                (heaterOnTime=3? 3: heaterOnTime+1):0;
        heating := lowTempTime=4
                | (heaterOnTime > 0 & heaterOnTime < 3);
    LTLSPEC
        G(!(heating & X !heating & X X heating));
    LTLSPEC
        G(!(!heating & X heating & X X !heating));
```

Listing 1 – Exercise 1 Solution

Running NuSMV on the solution gives the following output, which indicates that the solution is correct:

```
-- specification  G !((heating &  X !heating) &  X ( X heating)) is true
-- specification  G !((!heating &  X heating) &  X ( X !heating)) is true
```

Listing 2 – Exercise 1 Output

Exercise 2 – Resource Lock

2.1 – Implementation is Not Fair

To show that one of the parties may obtain the lock infinitely often, while the other never gets it despite polling infinitely for it, the LTL property in Listing 3 was added to the *main* module. For each user u_A, this property checks that (i) if u_A is requesting access to the critical section and (ii) if u_B will not be in the critical section forever starting from the next step, then (iii) u_A will eventually gain access.

```
LTLSPEC
    G ((u2.req & !(X G u1.mode=CRITICAL)) -> X F u2.mode=CRITICAL) &
    G ((u1.req & !(X G u2.mode=CRITICAL)) -> X F u1.mode=CRITICAL)
```

Listing 3 – Exercise 2.1 Fairness Checks

When NuSMV was run, this property turned out to be true. On further inspection, it was discovered that the implementation may be fair after all. Table 1 shows a sequence steps where both users manage to obtain the lock, even though user 1 attempts to obtain it again immediately after releasing it.

Description	User 1			User 2			Lock		
	Mode	Req	Rel	Mode	Req	Rel	Mode	Ack1	Ack2
Initially, both idle and lock available	IDLE	F	F	IDLE	F	F	AVAIL	F	F
Both users need to enter critical section	ENTR	**T**	F	ENTR	**T**	F	AVAIL	**T**	F
User 1 is given priority over User 2	CRIT	F	F	ENTR	**T**	F	LOCK	F	F
User 1 still in the critical section	...	…	…	…	…	…	…	…	…
User 1 finished	EXIT	F	**T**	ENTR	**T**	F	LOCK	F	F
Only User 2 is requesting the lock	IDLE	F	F	ENTR	**T**	F	AVAIL	F	**T**
User 2 manages to obtain the lock	ENTR	**T**	F	CRIT	F	F	LOCK	F	F
User 2 still in the critical section	…	…	…	…	…	…	…	…	…

Table 1 – Implementation is Fair

However, one can appreciate that if three users were competing for the lock using a similar approach, two of the users may alternatively obtain the lock and the third user never manages to obtain the lock.

2.2 – Improving the Manager

Since the manager was found to be fair, no changes were necessary. However, to give alternating priority, rather than always prioritizing User 1, some changes were still applied. A **priority** variable is used to indicate which user will be given priority if the users happen to simultaneously request the lock. A user loses prioritisation upon obtaining the lock. If only one user requests the lock, the current priority value is ignored.

```
MODULE manager(req1,req2,rel)
    VAR
        ack1: boolean;
        ack2: boolean;
        mode: {AVAILABLE, LOCKED};
        priority: 1..2;
    ASSIGN
        ack1 := req1 & (!req2 | priority=1) & mode=AVAILABLE;
        ack2 := req2 & (!req1 | priority=2) & mode=AVAILABLE;
        init(mode) := AVAILABLE;
        next(mode) :=
            case
                ack1 | ack2: LOCKED;
                rel: AVAILABLE;
                TRUE: mode;
            esac;
        init(priority) := 1;
        next(priority) :=
            case
                ack1 & priority=1: 2;
                ack2 & priority=2: 1;
                TRUE: priority;
            esac;
```

Listing 4 – Improving the Manager

Running NuSMV on the modified script gives the same result as that of the script for Exercise 2.1.

```
-- specification  G !(u1.mode = CRITICAL & u2.mode = CRITICAL)  is true
-- specification (
G ((u2.req & !( X ( G u1.mode = CRITICAL))) ->  X ( F u2.mode = CRITICAL)) &
G ((u1.req & !( X ( G u2.mode = CRITICAL))) ->  X ( F u1.mode = CRITICAL))) is true
```

Listing 5 – Exercise 2.2 Output

Exercise 3

To start with, the code in the slides was copied:

```
MODULE main
    VAR
        reqLift: array 1..3 of boolean;
        liftNow: 1..3;
        liftDst: 0..3;
    ASSIGN
        init(liftNow) := 1;
        next(liftNow) :=
            case
                liftDst=liftNow | liftDst=0: liftNow;
                liftDst > liftNow: liftNow + 1;
                TRUE: liftNow - 1;
            esac;
        init(liftDst) := 0;
        next(liftDst) :=
            case
                liftDst=liftNow: 0;
                liftDst!=0: liftDst;
                TRUE:
                    reqLift[1]?
                        (reqLift[2]?
                            (reqLift[3]? {1,2,3}:{1,2}):
                            (reqLift[3]? {1,3}:1)):
                        (reqLift[2]?
                            (reqLift[3]? {2,3}:2):
                            (reqLift[3]? 3:0));
            esac;
```

Listing 6 – Exercise 3 Original Script

3.1 – First LTL Property

The LTL property stating that for "Once no more requests are received, the lift destination remains 0." was written in the main module as shown in Listing 7. This LTL formula ensures that, starting from any point, if the requests are all false forever, the lift destination is immediately 0 and remains 0 forever.

```
LTLSPEC
    G ( G(!(reqLift[1] | reqLift[2] | reqLift[3])) -> G liftDst=0 )
```

Listing 7 – First LTL Property

Running NuSMV on the script with the LTL formula, the property turns out to be false, as expected. The output, presented overleaf (Listing 8), also gives a counterexample, which shows the case where `reqLift[1]` becomes false immediately after being recognized by the lift. The lift thus has to acknowledge the request, which means its destination is set to 1, even though the lift is already at floor 1.

If the property is modified by prefixing '`X`' to '`G liftDst=0`', the property still fails because `reqLift[2]` can become false immediately after being recognized by the lift, causing the lift to move from floor 1 to floor 2. Similarly, if another '`X`' is prefixed to '`X G liftDst=0`', the property fails because of `reqLift[3]`. The property finally becomes true if a third '`X`' is added but would be violated again if a fourth floor was added.

```
-- specification  G ( G !((reqLift[1] | reqLift[2]) |
reqLift[3]) ->  G liftDst = 0)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    reqLift[1] = TRUE
    reqLift[2] = FALSE
    reqLift[3] = FALSE
    liftNow = 1
    liftDst = 0
  -> State: 1.2 <-
    reqLift[1] = FALSE
    liftDst = 1
  -- Loop starts here
  -> State: 1.3 <-
    liftDst = 0
  -> State: 1.4 <-
```

Listing 8 – Exercise 3.1 Output

3.2 – Modifying the LTL Property

A general solution so that "Once no more requests are received, then (i) the lift destination will eventually become 0; and (ii) once it does so, it will remain 0 forever." is to use the *eventually* operator as shown in Listing 9. This ensures that *eventually* (`F`), the lift destination remains 0 forever (`G`).

```
LTLSPEC
    G ( G(!(reqLift[1] | reqLift[2] | reqLift[3])) -> F G liftDst=0 )
```

Listing 9 – Modified LTL Property

The output from running NuSMV on the resultant script indicates that the property is now satisfied:

```
-- specification  G ( G !((reqLift[1] | reqLift[2]) | reqLift[3]) ->
F ( G liftDst = 0)) is true
```

Listing 10 – Exercise 3.2 Output

3.3 – Strengthening the LTL Property

To ensure that "(once no more requests are received) the first time the lift destination turns 0, it will remain so", the property is strengthened as shown in Listing 11. The newly added '`liftDst=0`' condition ensures that, starting from the point that no more requests are received, once the destination is zero, all future destinations are also 0. The '`F liftDst=0`' ensures that the lift's destination does indeed eventually become 0.

```
LTLSPEC
    G ( G(!(reqLift[1] | reqLift[2] | reqLift[3])) ->
    (F liftDst=0 & (liftDst=0 -> G liftDst=0)) )
```

Listing 11 – Strengthened LTL Property

The output from running NuSMV on the resultant script indicates that the stronger property is also satisfied:

```
-- specification  G ( G !((reqLift[1] | reqLift[2]) | reqLift[3])
-> ( F liftDst = 0 & (liftDst = 0 ->  G liftDst = 0))) is true
```

Listing 12 – Exercise 3.3 Output

3.4 – CTL Property

The requirement that "sometimes the lift will non-deterministically be able to head towards level 1 or 2" was interpreted with the following assumptions in mind:

- "head towards level 1 or 2" implies that the lift starts off from level 3.
- "sometimes" translates to an existential quantification over all paths from when level 3 is reached.
- "level 1 or 2" is an exclusive 'or'.

Considering the above assumptions, the following is the CTL property for the specified requirement, where, for all possible paths, once level 3 is reached, there is at least one path along which level 1 or level 2 is reached.

```
CTLSPEC
    AG ( liftNow=3 -> EF (liftNow < 3) )
```

Listing 13 – The CTL Property

The output from running NuSMV on the resultant script indicates that the CTL property is satisfied:

```
-- specification AG (liftNow = 3 -> EF liftNow < 3) is true
```

Listing 14 – Exercise 3.4 Output

3.5 – Extended CTL Property

The CTL property for the requirement that "this non-determinism [Exercise 3.4] never arises if there never are concurrent requests from levels 1 and 2" is presented in Listing 15. The property from exercise 3.4 was reused but negated, since "this non-determinism never arises…". For "…there never are concurrent requests from levels 1 and 2", the condition that `reqLift[1]` and `reqLift[2]` are in no case both true was added.

```
CTLSPEC
    (AG !(reqLift[1] & reqLift[2])) -> !(AG ( liftNow=3 -> EF (liftNow < 3) ))
```

Listing 15 – Extended CTL Property

The output from running NuSMV on the resultant script indicates that the extended CTL property is satisfied:

```
-- specification (AG !(reqLift[1] & reqLift[2]) ->
!(AG (liftNow = 3 -> EF liftNow < 3))) is true
```

Listing 16 – Exercise 3.5 Output

Exercise 4 – Alternative Model

Listing 17 presents the original script of the given alternative model with the five properties from exercise 3 (shown in bold red) adapted for the new model. Running NuSMV on the script results in the violation of properties 1, 2 and 3, and the satisfaction of properties 4 and 5. From exercise 3, property 1 should be violated and properties 2 to 5 should be satisfied, and thus the results for properties 2 and 3 are incorrect.

```
MODULE main
    VAR
        req: array 1..3 of boolean;
        dst: destination(req,pos.out);
        pos: position(req,dst.out);
    LTLSPEC
        G ( G(!(req[1]|req[2]|req[3])) -> G dst.out=0 )
    LTLSPEC
        G ( G(!(req[1]|req[2]|req[3])) -> F G dst.out=0 )
    LTLSPEC
        G ( G(!(req[1]|req[2]|req[3])) -> (F dst.out=0 & (dst.out=0 -> G dst.out=0)) )
    CTLSPEC
        AG ( pos.out=3 -> EF (pos.out < 3) )
    CTLSPEC
        (AG !(req[1] & req[2])) -> !(AG ( pos.out=3 -> EF (pos.out < 3) ))

MODULE position(req,dst)
    VAR
        out: 1..3;
    INIT
        out = 1
    TRANS
        ((dst=out | dst=0) -> next(out)=out)
      & ((dst > out) -> next(out)=out + 1)
      & ((dst < out & dst > 0) -> next(out)=out - 1)

MODULE destination(req,pos)
    VAR
        out: 0..3;
    INIT
        out = 0
    TRANS
        (out=pos -> next(out)=0)
      & ((out!=pos & out>0) -> next(out)=out)
      & ( (!req[1] & !req[2] & !req[3])
        | (req[1] & next(out)=1)
        | (req[2] & next(out)=2)
        | (req[3] & next(out)=3)
        );
```

Listing 17 – Exercise 4 Original Script

The debugging will start with the counterexample (Listing 18 overleaf) given for the violation of property 2, which states that “Once no more requests are received, then (i) the lift destination will eventually become 0; and (ii) once it does so, it will remain 0 forever.” The counterexample shows the scenario where the destination goes from 0, to 1, and back to 0, even though all of the requests are false.

```
-- specification  G ( G !((req[1] | req[2]) | req[3]) ->
F ( G dst.out = 0)) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 2.1 <-
    req[1] = FALSE
    req[2] = FALSE
    req[3] = FALSE
    dst.out = 0
    pos.out = 1
  -> State: 2.2 <-
    dst.out = 1
  -> State: 2.3 <-
    dst.out = 0
```

Listing 18 – Counterexample for Violation of Property 2

The fact that the issue has to do with the lift's destination hints that we should focus on the destination module. Since $dst.out$ is 0 and $pos.out$ is 1 in state 2.1 (in Listing 18), then we know that **out!=pos**, and thus the first case in the TRANS is not satisfied (in Listing 17). Additionally, since $dst.out$ is 0, we know that **out>0** is not satisfied, and thus the change of the destination to 1 is due to the remainder of the TRANS.

```
TRANS
    (out=pos -> next(out)=0)
  & ((out!=pos & out>0) -> next(out)=out)
  & ( (!req[1] & !req[2] & !req[3])
    | (req[1] & next(out)=1)
    | (req[2] & next(out)=2)
    | (req[3] & next(out)=3)
    );
```

Listing 19 – Remainder of the TRANS

In Listing 19, one can notice that in the case of a request, the next value of the destination is deterministically 1, 2, or 3. However, in the case that all requests are false, no value is specified for the destination. This is most likely why it is set to 1 incorrectly. After it becomes 1, **out=pos** is satisfied, and thus **next(out)=0**. Thus, we can apply the following fix to ensure that the destination is set to 0 if there are not requests:

```
TRANS
    (out=pos -> next(out)=0)
  & ((out!=pos & out>0) -> next(out)=out)
  & ( (!req[1] & !req[2] & !req[3] & next(out)=0)
    | (req[1] & next(out)=1)
    | (req[2] & next(out)=2)
    | (req[3] & next(out)=3)
    );
```

Listing 20 – Modified TRANS Section

Running NuSMV on the modified script results in the violation of property 1 and the satisfaction of all other properties. Thus, the solution for the satisfaction of property 2 has also caused property 3 to be satisfied and thus no further changes are necessary. Listing 21 presents the output with added property identification tags.

```
(Property 4)
-- specification AG (pos.out = 3 -> EF pos.out < 3) is true
(Property 5)
-- specification (AG !(req[1] & req[2]) -> !(AG (pos.out =
3 -> EF pos.out < 3))) is true
(Property 1)
-- specification  G ( G !((req[1] | req[2]) | req[3]) ->  G
dst.out = 0) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample

        …identical to counterexample in Listing 8…

(Property 2)
-- specification  G ( G !((req[1] | req[2]) | req[3]) ->  F
( G dst.out = 0)) is true
(Property 3)
-- specification  G ( G !((req[1] | req[2]) | req[3]) -> (
F dst.out = 0 & (dst.out = 0 ->  G dst.out = 0))) is true
```

Listing 21 – Exercise 4 Output

Exercise 5 – Single Lift Model Generator

The form chosen for this exercise is that of exercise 4. Throughout the exercise, it was assumed that the number of floors is 3 or more, specifically because of the CTL properties which would otherwise not make sense.

5.1 – Main Module

In the Main module, the generalisations involve the *req* arrau and the properties. In the case of the *req* array, this has to be of size n and so it was changed as shown in Listing 22 below:

```
req: array 1..n of boolean;
```

Listing 22 – Generalised Requests Array

As for the properties, the CTL properties were assumed to be independent of the number of floors. For the three LTL properties, the part of the expressions that checks for the absence of requests was changed so that it considers all of the floors as presented in Listing 23 below:

```
LTLSPEC
    G ( G(!(req[1]|req[2]|...|req[n])) -> G dst.out=0 )
LTLSPEC
    G ( G(!(req[1]|req[2]|...|req[n])) -> F G dst.out=0 )
LTLSPEC
    G ( G(!(req[1]|req[2]|...|req[n])) -> (F dst.out=0 & (dst.out=0 -> G dst.out=0)) )
```

Listing 23 – Generalised LTL Properties

5.2 – Position Module

The position module requires only its out variable to be generalised to n instead of 3 as the maximum. Its remaining parts are essentially independent of the number of floors since they depend on the destination.

5.3 – Destination Module

Lastly, the destination module requires a change to the out variable similar to the out variable of the position module, but also a change in the *TRANS* statement, which was generalised as shown in Listing 24.

```
TRANS
    (out=pos -> next(out)=0)
  & ((out!=pos & out>0) -> next(out)=out)
  & ( (!req[1] & !req[2] & ... & !req[n] & next(out)=0)
    | (req[1] & next(out)=1)
    | (req[2] & next(out)=2)
    ...
    | (req[n] & next(out)=n)
    );
```

Listing 24 – Generalised TRANS Statement

5.4 – Experiments with Different Numbers of Floors

NuSMV was run on the generated scripts for $n \in \{10,12,14,16,18\}$. Up to $n = 18$, the properties all behaved as expected and there were no false positives or false negatives. However, as can be observed in Figure 1, the execution time increases exponentially as the number of floors is increased. For $n = 10$, the execution time was of 1.3 seconds, but for $n = 14$, this increased to 3.1 minutes and for $n = 18$ this jumped to 10.3 hours. Due to the exceedingly high execution time, no further greater numbers of floors were attempted.

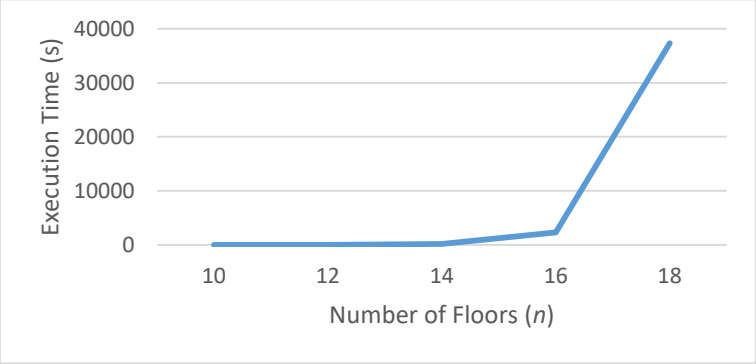

Figure 1 – Chart of Number of Floors and Execution Time

Exercise 5 (cont.) – Multiple Lifts Model Generator

5.1 – Module Overview

The following is a description of how the automatically generated model is organised, where m is the number of lifts and n is the number of floors:

- **`MODULE main`**: consists of just a building variable:
  - `building: building;`

- **`MODULE building`**: consists of a control panel, memorised user requests, and m lifts. This provides to the request memory module (i) the control panel inputs, so that it knows what inputs to memorise, and (ii) the lifts, so that the memory of any floor at which a lift opens its doors can be erased (`NONE`). Additionally, the lifts are provided with the user requests, so that they can move around accordingly.
  - `ctrl: controlPanel;`
  - `req: requestMemory(ctrl.up, ctrl.down, lift1, …, liftm);`
  - `lift1: lift(req.reqs);`
  - …
  - `liftm: lift(req.reqs);`

- **`MODULE controlPanel`**: consists of the user lift request buttons, one for each of the n floors:

  - `up: array 1..FLOORS of boolean;`
  - `down: array 1..FLOORS of boolean;`

- **`MODULE requestMemory(up, down, lift1,…, lift`m`)`** : consists of the memorised user requests, one for each of the n floors, with the values UP, DOWN, and NONE, as specified in the slides:
  - `reqs: array 1..FLOORS of {UP, DOWN, NONE};`

- **`MODULE lift(reqs)`** : consists of n door (obstacle) sensors (one per floor), a single weight sensor, an array of $n + 1$ user (goto) buttons (one per floor with 0 for no input), n doors (one per floor) with four possible values (open, opening, closed, closing), and the current level and destination of the lift:

  - `oSensor: array 1..FLOORS of boolean;`
  - `wSensor: boolean;`
  - `goto: 0..FLOORS;`
  - `door: array 1..FLOORS of {OPEN, OING, CLOS, CING};`
  - `level: 1..FLOORS;`
  - `dest: 0..FLOORS;`

5.2 – Variable Values

What remains to be discussed is how the changing of the memorised user requests and the lifts' doors, level, and destination variables, as the model is executed, was implemented. Since the door sensors, weight sensors, lift-request buttons, and user buttons in the lifts are all (non-deterministic) inputs, these will not be discussed.

5.2.1 – The Lift Doors

For the *door* array, n ASSIGN statements were used. Listing 25 presents the assignment for second floor door, assuming $n > 2$. The door is initially closed and its next state depends on its current and various other factors. The last three cases are straightforward; (2) when opening, it can open or remain opening, (3) once open, it remains open if there are no objects and (4) once closing, it *can* only close if there is no obstacle in the way. The first case, i.e. when the door is closed, is somewhat more complex since it depends on the current lift level, lift direction, and memorised user requests. If one of three subcases is satisfied, the door starts opening.

Subcase 1. The lift's destination is the door's level and it has arrived;
Subcase 2. The lift, which is going down, is one level above its destination or above a user requested level;
Subcase 3. The lift, which is going up, is one level below its destination or above a user-requested level.

```
ASSIGN
    init(door[2]) := CLOS;
    next(door[2]) :=
        case
            door[2]=CLOS: (level=2 & dest=2)                                -- already there
                        | (level=3 & goingDn & (dest=2 | reqs[2]=DOWN)) -- from above
                        | (level=1 & goingUp & (dest=2 | reqs[2]=UP))   -- from below
                          ? OING : CLOS;
            door[2]=OING: {OPEN, OING};
            door[2]=OPEN: (!oSensor[2]? OPEN : CING);
            door[2]=CING: (oSensor[2]? CING : {CLOS, CING});
        esac;
```

Listing 25 – Second Floor Door Assignment for $n > 2$

In the case of floor 1, the third subcase is not included, since there are no further floors below, whereas in the case of floor n, the second subcase is not included, since there are no further floors above. The lift's movement direction (presented in Listing 26) is defined based on whether the lift is below or above its destination, *if any*.

```
goingUp := dest != 0 & level < dest;
goingDn := dest != 0 & level > dest;
```

Listing 26 – Definitions for 'goingUp' and 'goingDown'

5.2.2 – The Lift Level

For the *level*, a single pair of INIT and TRANS statements was used. The lift starts off at level 1. It remains at its current level if it has no destination or if it is going up/down and there is a request to go up/down. Otherwise, the lift moves towards the destination if there are no request for the same direction at the current level.

```
INIT level = 1;
TRANS
    ((dest=0
    | (goingUp & reqs[level]=UP)
    | (goingDn & reqs[level]=DOWN)) -> next(level) = level)
  & ((goingDn & reqs[level] != DOWN) -> next(level) = level - 1)
  & ((goingUp & reqs[level] != UP  ) -> next(level) = level + 1)
```

Listing 27 – General Lift Level Initialisation and Transitions

5.2.3 – The Destination

For the *destination*, an ASSIGN statement was used. The destination is initialised to none (i.e. 0). The first case in the *next* part is straightforward: if the lift has a person inside it and no destination, its destination becomes the *goto* value. Otherwise, each level is checked, starting from the current and then alternating between the next below (which is prioritised) and next above. The first request detected, if any, becomes the destination.

```
ASSIGN
    init(dest) := 0;
    next(dest) :=
        case
            wSensor & dest=0: goto;
            (reqs[level] != NONE): level;
            ((level)-1 >  0)      & (reqs[(level)-1] != NONE): (level)-1;
            ((level)+1 <= FLOORS) & (reqs[(level)+1] != NONE): (level)+1;
            ((level)-2 >  0)      & (reqs[(level)-2] != NONE): (level)-2;
            ((level)+2 <= FLOORS) & (reqs[(level)+2] != NONE): (level)+2;
            TRUE: 0;
        esac;
```

Listing 28 – Destination Assignment for $n = 3$

5.2.4 – The Memorised User Requests

For the memorised user requests *reqs*, a collection of n INIT statements and a single TRANS statement were used. Listing 26 presents these for $m = 2$ and $n = 3$. For each floor, the requests are initialised to NONE. If an *up* or *down* button is pressed for a particular floor, the respective request is set accordingly (and immediately), giving priority to the *up* button. A request is set to NONE if the door of any lift opens up at the particular floor.

```
INIT reqs[1] = NONE;
INIT reqs[2] = NONE;
INIT reqs[3] = NONE;
TRANS
    (up[1] -> reqs[1]=UP)
  & (up[2] -> reqs[2]=UP)
  & (up[3] -> reqs[3]=UP)
  & (!up[1] & down[1] -> reqs[1]=DOWN)
  & (!up[2] & down[2] -> reqs[2]=DOWN)
  & (!up[3] & down[3] -> reqs[3]=DOWN)
  & ((lift1.door[1]=OPEN | lift2.door[1]=OPEN) -> reqs[1]=NONE)
  & ((lift1.door[2]=OPEN | lift2.door[2]=OPEN) -> reqs[2]=NONE)
  & ((lift1.door[3]=OPEN | lift2.door[3]=OPEN) -> reqs[3]=NONE)
```

Listing 29 – Memorised User Request Initialisation and Transitions for $m = 2$ and $n = 3$

Exercise 6

6.1 – Explaining the Specifications

```
G ((weight sensor[1] & goto[1]>0) -> X (destination[1]>0))
```

The above LTL specification states that if the weight sensor shows the presence of a person and a button is pressed inside the lift, then the destination cannot be zero in the next step. In fact, the destination is expected to become the value of *goto*, unless it was already some non-zero value.

```
G (goto[1]>0 -> F (door[1][goto[1]]=OPEN))
```

The above LTL specification states that if a user button is pressed in the lift, then the lift's doors at the specified floor will eventually open (presumably after the lift reaches that level – but this is not being checked). However, this property is erroneous in that the value of $goto$ may be zero, thus causing an array bounds exception. Despite the $goto[1] > 0$ condition, the actual *goto* value used in the remainder is not necessarily the same, and it can thus become 0. In order to make the above property usable, it was changed to the following:

Modified property: `G (goto[1]=1 -> F (door[1][1]=OPEN))`

The modified property checks that if the user wishes to go to floor 1, the respective door eventually opens. To achieve the intention of the original property, the modified version can be duplicated for the remaining floors.

```
AG !(door[1][1]=OPEN & EX door[1][1]=OPENING)
```

The above CTL specification states that it is never the case that the door goes from *open* to *opening*.

```
AG !(door[1][1]=CLOSING & EX door[1][1]=OPENING)
```

The above CTL specification states that it is never the case that the door goes from *closing* to *opening*.

```
AG !(door[1][1]=CLOSED & EX door[1][1]=CLOSING)
```

The above CTL specification states that it is never the case that the door goes from *closed* to *closing*.

```
AG !(door[1][1]=OPENING & EX door[1][1]=CLOSING)
```

The above CTL specification states that it is never the case that the door goes from *opening* to *closing*.

6.2 – Applying the Specifications to the Automatically-Generated Model

To apply the specifications to the automatically-generated model, the generator was modified to generate the main module shown in Listing 30. A set of definitions were included to avoid having to modify the properties. For the second LTL property, the modified version was used rather than the given one.

```
MODULE main
    VAR
        building: building;
    DEFINE
        weight_sensor[1] := building.lift1.wSensor;
        goto[1] := building.lift1.goto;
        destination[1] := building.lift1.dest;
        door[1] := building.lift1.door;
        level[1] := building.lift1.level;
        OPENING := OING;
        CLOSING := CING;
        CLOSED := CLOS;
    LTLSPEC G ((weight_sensor[1] & goto[1]>0) -> X (destination[1]>0))
    LTLSPEC G (goto[1]=1 -> F (door[1][1]=OPEN))
    CTLSPEC AG !(door[1][1]=OPEN    & EX door[1][1]=OPENING)
    CTLSPEC AG !(door[1][1]=CLOSING & EX door[1][1]=OPENING)
    CTLSPEC AG !(door[1][1]=CLOSED  & EX door[1][1]=CLOSING)
    CTLSPEC AG !(door[1][1]=OPENING & EX door[1][1]=CLOSING)
```

Listing 30 – Main Module

To test whether the properties hold, a model with two lifts ($m = 2$) and three floors ($n = 3$) was generated. Running NuSMV on the model resulted in the violation of the two LTL properties and the satisfaction of all of the CTL properties. The remainder of the document will discuss the two violations.

6.2.1 – Violation of LTL Property 1

By analysing the counterexample given for the violation of the first LTL property, it was noticed that despite the weight sensor being activated and the *goto* set to some floor, once the lift reaches that floor, the destination becomes zero if there are no other requests, even if the weight sensor and *goto* are still the same values. This is due to the fact that the weight sensor is only considered if the lift "has no destination".

6.2.2 – Violation of LTL Property 2

By analysing the counterexample given for the violation of the second LTL property, it was discovered that the property cannot be true because the door at floor 1 may remaining in an *opening* state forever, given that it is allowed to non-deterministically keep opening or else go to the *open* state.

A secondary issue is that the system does not have a memory for the *goto* buttons, as described in the slides. Thus, if the *goto* is released immediately after it is set, the lift can forget about it and thus never service it. This means that even if the property is changed to "`G (...door[1][1]=OPENING)`", it is still violated.