

# AltaRica Examples

## The lift<sup>\*</sup>

A. Griffault, G. Point  
LaBRI - CNRS - Université Bordeaux

April 22, 2024

### Contents

<b>1</b>	<b>Informal specification of the lift</b>	<b>2</b>
<b>2</b>	<b>Roadmap to the model</b>	<b>2</b>
2.1	The number of floors . . . . .	2
2.2	Users . . . . .	3
2.3	What kind of modelling to use ? . . . . .	3
<b>3</b>	<b>The model</b>	<b>4</b>
3.1	Buttons . . . . .	4
3.2	Doors . . . . .	5
3.3	Floors . . . . .	5
3.4	The cage . . . . .	6
3.5	The whole system . . . . .	7
<b>4</b>	<b>Analysis and results of the system</b>	<b>8</b>
4.1	Safety properties . . . . .	9
4.2	Liveness property . . . . .	11
4.3	Checking the model . . . . .	11

---

This modelling example is inspired from Laroussinie’s thesis[1]. Starting from informal specifications, this “exercise” consists to build a model of a lift that satisfies given requirements.

---

<sup>\*</sup>This example is extracted from AltaRica Handbook.

# 1 Informal specification of the lift

The studied lift serves  $n$  floors. The cage is equipped with  $n$  light buttons that permit to select one or more destinations; when a button is lighted there exists a request for the corresponding floor. Each floor has a similar button used to ring for the lift. When the cage stops at a floor its door opens automatically.

At each time, a software controller chooses the next thing to do between : open a door, close a door, go up, go down or nothing. The owner of the building wants that following requirements have been proved.

1. When a button is push, it lights.
2. When the corresponding service is done, it lights off.
3. At each floor, the door is close if the lift is not here.
4. The software opens the door at some floor only if there is some requests for that floor.
5. If there is no request, the lift must stay at the same floor.
6. When the lift moves, it must stop where there is a request.
7. When there are several requests, the software must (if necessary) continue in the same direction than its last move.
8. Each request must be honored a day.

# 2 Roadmap to the model

## 2.1 The number of floors

With finite model-checking we cannot prove a property with parameters. For that, we need theorem proving method. So we need to fix the number of floors. 1000 seems a good choice since no building in the world have so much floors, but no model checker in the world can deal with such model. On the opposite, every model checker can deal with a building with only one floor, but a lift is not usefull in such a building. In addition, most of the properties are tautology for a one floor building.

The model assumes the lift serves 4 floors. This choice is justified as follows:

1. Two floors are not enough because behaviours of the lift are, in this case, necessarily fair. Actually, with only two floors, the cage passes to one floor as many times as the other one.
2. Three floors could be sufficient, but in this case, every traversal starts and/or ends at the ground-floor or at the last one. This specificity may impact results related to fairness properties.

3. Four floors seem enough.
4. Floors beyond the fourth seem redundant. If  $N \geq 4$  is the number of floors, we consider the following partition of floor numbers as follows:  $\{1\}$ ,  $\{2\}$ ,  $\{3, \dots, N-1\}$ ,  $\{N\}$ . Then we associate to each subset the number of a floor for a lift serving 4 floors:  $\{1\} \rightarrow 1$ ,  $\{2\} \rightarrow 2$ ,  $\{3, \dots, N-1\} \rightarrow 3$ ,  $\{N\} \rightarrow 4$ . We are prone to believe that any property satisfied by a 4-floors lift should be satisfied by a  $N$ -floors lift for  $N > 4$ .

Actually above arguments are not proofs but are just remarks imposed by common sense.

In the sequel we will use the following definitions.

```
const NB_FLOORS = <?php echo $NB_FLOORS ?>;
const GROUND_FLOOR = <?php echo $GROUND_FLOOR ?>;
const TOP_FLOOR = GROUND_FLOOR + NB_FLOORS - 1;
domain CageLocation = [GROUND_FLOOR, TOP_FLOOR];
```

Note the use of PHP code. In the sequel of this example, `arc` requests PHP preprocessing for each loaded file. `php` is invoked using following settings in `arc`:

```
set arc.shell.preprocessor.default.command \
"php -d short_open_tag=On -r '$NB_FLOORS = 4; $GROUND_FLOOR = 1; \
    $TOP_FLOOR = $GROUND_FLOOR+$NB_FLOORS-1; \
    include ($argv[1]); ?>'"
```

## 2.2 Users

The environment is not addressed by our model. Actually the environment is composed by an unbounded number of users. In the model, such environment can not be described because, actually, it requires to bound this number of users. In absence of users, requests are described using uncontrollable actions on buttons (whose operator is ignored). Surprisingly, this method permits to handle behaviours of an unbounded number of users.

## 2.3 What kind of modelling to use ?

There is a choice to do regarding the abstraction level used to build the model. Do we take a functional point of view or do we have to go further down to implementation level ?

**Functional level:** Both buttons related to a same floor can be merged: indeed, it does not matter that one button is  $n$ -th floor and the other in the cage.

**Implementation level:** The buttons, at a floor and in the cage, should no use the same means to communicate the control software: the first is at a static location while the second can move.

As it has been mentioned in introduction, we face a modelling *exercise*. We do not try to model an actual lift in order to study its behaviours but, we rather try to build one with some expected properties.

Thus, the model should adopt the functional point of view.

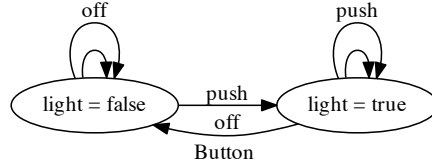


Figure 1: Semantics of the Button node.

### 3 The model

The model is based on the modelling of four *physical* objects: the buttons, the door, the floors and the cage. Obviously the model also describes their “straightforward” interactions. The model is composed with:

- Four floors and a cage.
- A door and four buttons in the cage.
- A door and a button at each floor.

#### 3.1 Buttons

There are many ways to model buttons. For this system a switch, i.e., a button that alternates between two positions, is not appropriate because users may press several times the buttons without alternation. The model of button informs its environment of its state (lit or not) using a flow variable `light`.

Another choice is to decide if a button can be putted off (by the controller) while being already off. We decide to allow these behaviours.

```

node Button
  flow light : bool;
  state on : bool;
  init on := false;
  event push : public;
  off;
  trans
    true |- push -> on := true;
    true |- off -> on := false;
  assert
    light = on;
edon

```

The reader should have notice the use of `public` attribute on `push` event. We use this trick to ensure the strict asynchronism of the events. Actually, two buttons could be pressed simultaneously but it does not impact the behaviours of the whole system.

The semantics of this model is depicted on figure 1.

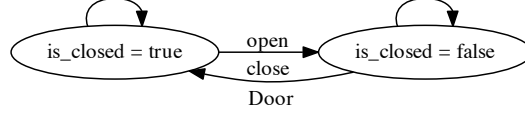


Figure 2: Semantics of the Door node

### 3.2 Doors

The model of the door makes no surprise. The reader should note that the initial state of the door is already set to *closed* and its internal state is shared with its environment via a flow variable.

The semantics of this node is given on figure 2.

```

node Door
  flow is_closed : bool;
  state closed : bool;
  init closed := true;
  event open, close : public;
  trans
    closed |- open -> closed := false;
    not closed |- close -> closed := true;
  assert
    is_closed = closed;
edon

```

### 3.3 Floors

A floor is made of a button and a door. The floor puts the button off when the request for this floor has been fulfilled. To model this phenomenon we have to define when the service is done at the floor; it can be either at the opening or at the closing of the door. We choose the latter case.

We use two local events, **open** and **close**, to expose opening and closing of the door at the floor.

A flow variable **requested** is used to inform the controller that this floor has been requested via its button. Obviously the variable is set to the state of the button.

The semantics of this node is depicted on figure 3.

```

node Floor
  flow requested : bool : public;
  sub
    B : Button;
    D : Door;
  event open, close;
  trans true |- open, close ->;
  sync <close, D.close, B.off>;
  sync <open, D.open>;
  assert
    requested = B.light;
edon

```

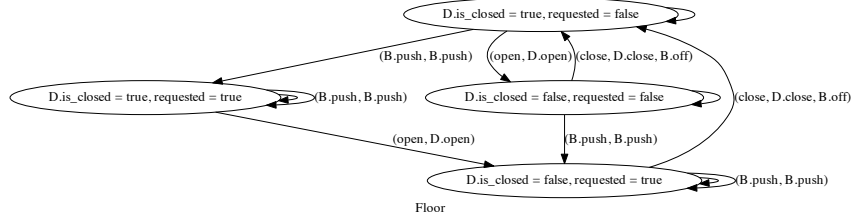


Figure 3: Semantics of the Floor node.

### 3.4 The cage

A state variable `loc` models the location of the cage; the controller knows this location via the flow variable `floor`. We assumed the cage is at ground floor in the initial configuration. `loc` variable is modified by two events, `up` and `down`, that model moves of the cage. The cage moves only if its door is closed thus, `up` and `down` are guarded according to the state of the door.

We can send the off signal to the appropriate button when the corresponding request is satisfy. We made the same choice as for the floor regarding the meaning of the service is done; buttons are put off when the door is closing.

An array of Booleans, `request`, is used to indicate the controller recorderd requests from the cage.

```

node Cage
  flow
    request : bool[NB_FLOORS];
    floor : CageLocation;
  state
    loc : CageLocation;
  init
    loc := GROUND_FLOOR;
  sub
    D : Door;
    B : Button[NB_FLOORS];
  event
    up, down, close[NB_FLOORS], open;
  trans
    D.is_closed | up -> loc := loc + 1;
    D.is_closed | down -> loc := loc - 1;
    D.is_closed | open ->;

  assert floor = loc;
  sync <open, D.open>;

  // for i = 0 ... N-1:
  //   assert request[i] = B[i].light
  //   trans ~D.is_closed & floor = i | close[i] -> ;
  //   sync <close[i], D.close, B[i].off>;

  <?php for ($i = 0 ; $i < $NB_FLOORS; $i++) { ?>
    assert request[<?=$i?>] = B[<?=$i?>].light;
    trans ~D.is_closed & floor = <?=$GROUND_FLOOR+$i?> |
      close[<?=$i?>] -> ;
    sync <close[<?=$i?>], D.close, B[<?=$i?>].off>;
  <?php } ?>
edon

```

The semantics of the node is too large to be depicted on a figure; ARC tool tells us:

```
/*
 * Properties for node : Cage
 * # state properties : 1
 *
 * any_s = 128
 *
 * # trans properties : 1
 *
 * any_t = 864
 */
```

### 3.5 The whole system

The model that describes the whole system consists of a **Cage** and four **Floors**.

We use **private** flow variables to make local computation:

- **requestDown** indicates if there exists a request to a floor that is down the current position of the cage;
- **requestUp** indicates if there exists a request to a floor that is up the current position of the cage;
- **request** is an array of  $N$  Booleans; **request**[ $i$ ] is **true** if there exists a request for the  $i$ -th floor.

The opening is allowed at some floor only if there is some request to that floor. The controller has  $N$  events **open**[ $i$ ] guarded by this condition. Synchronization vectors are used to model the synchronous opening or closing of doors of the cage and of the floors.

```
<open[i], C.open, F[i].open>;
<C.close[i], F[i].close>;
```

We use PHP preprocessor to generate synchronization vectors and assertions which makes the model not so easy to read but comments should help.

```
node Main1
sub
  F : Floor[NB_FLOORS];
  C : Cage;

flow
  requestUp, requestDown : bool : private;
  request : bool[NB_FLOORS] : private;

event down, up, open[NB_FLOORS];

trans
  requestDown |- down  -> ;
  requestUp   |- up    -> ;

sync <up,      C.up>;
    <down,    C.down>;
```

```

// For i = 0 ... N - 1:
//   assert request[i] = (C.request[i] | F[i].requested);
<?php for ($i = 0; $i < $NB_FLOORS; $i++) { ?>
    assert request[<?=$i?>] = (C.request[<?=$i?>] |
        F[<?=$i?>].requested);
<?php } ?>

//   assert requestUp =
//   For i = 0 ... N-2,
//   (C.floor = i &
//   (for i < j < N, C.request[j] | F[j].requested))
//   assert requestUp = (
<?php for ($f = 0; $f < $NB_FLOORS-1; $f++) { ?>
    (C.floor = <?=$GROUND_FLOOR + $f?> & (
<?php for ($r = $f + 1; $r < $NB_FLOORS; $r++) { ?>
        request[<?=$r?>]
<?php if ($r != $NB_FLOORS-1) echo "|"; } ?>
    ))
<?php if ($f != $NB_FLOORS-2) echo "|"; } ?>
);

//   assert requestDown =
//   For i = 1 ... N-1,
//   (C.floor = i &
//   (for 0 <= j < i, C.request[j] | F[j].requested))
//   assert requestDown = (
<?php for ($f = 1; $f < $NB_FLOORS; $f++) { ?>
    (C.floor = <?=$GROUND_FLOOR + $f?> & (
<?php for ($r = 0; $r < $f; $r++) { ?>
        request[<?=$r?>]
<?php if ($r != $f-1) echo "|"; } ?>
    ))
<?php if ($f != $NB_FLOORS-1) echo "|"; } ?>
);

// For i = 0 ... N - 1:
//   trans (C.floor=i) & request[i] |- open[i] -> ;
//   sync <open[i], C.open, F[i].open>;
//   <C.close[i], F[i].close>;

<?php for ($i = 0; $i < $NB_FLOORS; $i++) { ?>
    trans (C.floor=<?=$GROUND_FLOOR+$i?>) & request[<?=$i?>] |-
        open[<?=$i?>] -> ;

    sync <open[<?=$i?>], C.open, F[<?=$i?>].open>;
        <C.close[<?=$i?>], F[<?=$i?>].close>;
<?php } ?>
edon

```

## 4 Analysis and results of the system

The analysis of this model is realized using ARC model-checker. We have to prove the truth of eight properties. The first seven are *safety* properties while the last one is a *liveness* property.



## 4.1 Safety properties

Safety properties describe invariants i.e. properties that must be true in all configurations of the system. To prove these properties using **arc** we compute sets of configurations that do not satisfy the specified invariant and we check if this set is empty or not. If not, the model does not satisfy the property.

### 4.1.1 When a button is push, it lights

This property is falsified if after a **push** event on a button  $B$  its state is off. We compute the set **notP1** that is the union for each button  $B$  of reachable configurations that are the target configuration of  $B.push$  but with  $B.light$  equal to **false**.

For each floor  $i$  the following set should be empty:

$$(\text{tgt}(\text{label } F[i].B.\text{push}) \ \& \ [\text{not } F[i].B.\text{light}]) \mid \\ (\text{tgt}(\text{label } C.B[i].\text{push}) \ \& \ [\text{not } C.B[i].\text{light}])$$

Specifications passed to **arc** is the following:

```
notP1 :=
<?php for ($i = 0; $i < $NB_FLOORS; $i++) { ?>
    (tgt(label F[<?=$i?>].B.push) & [not F[<?=$i?>].B.light]) |
    (tgt(label C.B[<?=$i?>].push) & [not C.B[<?=$i?>].light])
<?php if ($i != $NB_FLOORS-1) echo "|"; } ?>
```

### 4.1.2 When the corresponding service is done, it lights off.

Throughout the modelling process we have assumed that the service is done at floor  $i$  when its door becomes closed. Configurations that do not satisfy this invariant are the target of a closure event from some floor  $i$  but with a button for  $i$ -th floor that is yet lit.

For each floor  $i$  the following set should be empty:

$$(\text{any\_s} \ \& \ \text{tgt}(\text{label } F[<i>].\text{close}) \ \& \ [\text{request}[i]])$$

Note the use of predefined set **any\_s** that contains reachable configurations. Here its use is mandatory because some unreachable configurations do not respect invariant  $P_2$ .

Specifications passed to **arc** is the following:

```
notP2 :=
<?php for ($i = 0; $i < $NB_FLOORS; $i++) { ?>
    (any_s & tgt(label F[<?=$i?>].close) & [request[<?=$i?>]])
<?php if ($i != $NB_FLOORS-1) echo "|"; } ?>
```

### 4.1.3 At each floor, the door is close if the lift is not here.

This property is straightforward. A configuration does not satisfy invariant  $P_3$  if the door of some floor  $i$  is open while the cage is at some other floor  $j \neq i$ .

For each floor  $i$  the following set should be empty:

(any\_s & [C.floor != i & not F[i].D.closed])

Specifications passed to `arc` is the following:

```
notP3 :=
  <?php for ($i = 0; $i < $NB_FLOORS; $i++) { ?>
    (any_s & [C.floor != <?=$i+$GROUND_FLOOR?> & not
      F[<?=$i?>].D.closed])
  <?php if ($i != $NB_FLOORS-1) echo "|"; } ?>;
```

#### 4.1.4 The software opens the door at some floor only if there is some requests for that floor.

Here we describe a property on transitions.  $P_4$  is falsified if there exists a transition that open the door of some floor  $i$  from a configuration where no request exists for  $i$ -th floor. This is specified as follows:

(label F[i].D.open - rsrc([request[i]]))

Specifications passed to `arc` is the following:

```
notP4 :=
  <?php for ($i = 0; $i < $NB_FLOORS; $i++) { ?>
    (label F[<?=$i?>].D.open - rsrc([request[<?=$i?>]]))
  <?php if ($i != $NB_FLOORS-1) echo "|"; } ?>;
```

#### 4.1.5 If there is no request, the lift must stay at the same floor.

Similarly to  $P_4$  we prove  $P_5$  with the emptiness of a set of transitions. This set of *bad* transitions should move the cage while there is no request. Transitions that move the lift are those labelled with controller's events `up` or `down`; we check there is no such transition triggered from a configuration where there exists a request.

We write this set of bad transitions:

(label C.up | label C.down) - rsrc([request[0] | ... | request[N-1]])

Specifications passed to `arc` is the following. We have constraint the set to be in `any_t` to be sure to consider only transitions between reachable configurations:

```
notP5 := any_t & ((label C.up | label C.down) -
  rsrc([request[0]
  <?php for ($i = 1; $i < $NB_FLOORS; $i++) { ?>
    | request[<?=$i?>]
  <?php } ?> ]));
```

#### 4.1.6 When the lift moves, it must stop where there is a request.

$P_6$  is not satisfied if there exists a transitions that moves the cage while it is at some floor  $i$  and there exists a request for this floor. Yet, if such bad transition exists, it is labelled with `up` or `down` and is triggered from a configuration where, for some  $i$ , `C.floor` equals  $i$  and `request[i]` is `true`. We compute the set of transitions:

```
(label C.up | label C.down) &
  rsrc([C.floor[0] \& request[0]] |
    ...
    [C.floor[N-1] \& request[N-1]])
```

Specifications passed to `arc` is the following:

```
notP6 := any_t &
  ((label C.up | label C.down) &
    rsrc([C.floor = <?=$GROUND_FLOOR?> & request[0]]
  <?php for ($i = 1; $i < $NB_FLOORS; $i++) { ?>
    | [C.floor = <?=$i+$GROUND_FLOOR?> & request[<?=$i?>]]
  <?php } ?> ));
```

#### 4.1.7 When there are several requests, the software must (if necessary) continue in the same direction than its last move.

Property  $P_7$  specifies that in any configuration the cage can moves in only one direction. We compute the set of bad transitions that represent moves of the cage (i.e labelled with `up` or `down`) and that are triggered from a configuration from which it is also possible to move but in the opposite direction.

Specifications passed to `arc` is the following:

```
notP7 := any_t & (label C.up & rsrc(src(label C.down)) |
  label C.down & rsrc(src(label C.up))) ;
```

## 4.2 Liveness property

The last property  $P_8$  is different from previous ones because it specifies a liveness condition. The model satisfies  $P_8$  if there is a request for some floor  $i$  (via one of both buttons for this floor) then, eventually, the cage will serve the  $i$ -th floor.

In order to check this property we leave Dicky's logic to use CTL\* module of `arc`. This change is due to the use of symbolic data structure that not permit the use of the well-known `loop` operator usually used to prove liveness properties. Unfortunately this operator works only with explicit representation of state graphs.

The model satisfies  $P_8$  if for all floor  $i$  its initial state fulfils the CTL formula:

$$AG([request[i]] \Rightarrow AF([not request[i]]))$$

Recalls about CTL formulas:

- A state  $s$  satisfies  $AG\phi$  if all states reachable from  $s$  satisfy  $\phi$ .
- A state  $s$  satisfies  $AF\phi$  if on all paths originating from  $s$  there exist a state that satisfies  $\phi$ .

To check  $P_8$  we have use `chkctl` command of `arc`. We could have used `ctlspec` to integrate it with others properties but `chkctl` offers a better counter-example generator specialized for CTL formulas.

## 4.3 Checking the model

If we consider only safety properties, our model does not satisfy properties  $P_6$  and  $P_7$ . `arc` script tells us:

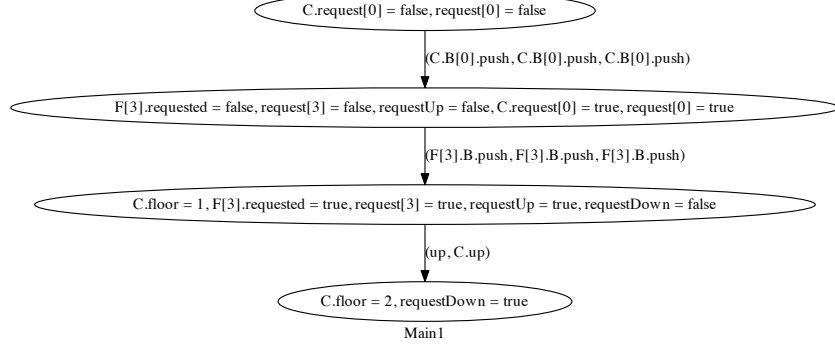


Figure 4: Counter example for  $P_6$  property.

```

/*
 * Properties for node : Main1
 * # state properties : 1
 *
 * any.s = 1792
 *
 * # trans properties : 1
 *
 * any.t = 19032
 */
TEST(initial,1) [PASSED]
TEST(notP1,0) [PASSED]
TEST(notP2,0) [PASSED]
TEST(notP3,0) [PASSED]
TEST(notP4,0) [PASSED]
TEST(notP5,0) [PASSED]
TEST(notP6,0) [FAILED] actual size = 1026
TEST(notP7,0) [FAILED] actual size = 720

```

#### 4.3.1 Fixing $P_6$ : When the lift moves, it must stop where there is a request.

In order to fix  $P_6$  we request **arc** to generate a counter-example to a configuration that is the source of a faulty transition i.e. that moves the cage out of floor  $i$  while there is actually a request for  $i$ . Commands passed to **arc** are the following:

```

traceP6 := trace(initial, any.t, src(notP6));
ceP6 := reach(src(traceP6), traceP6|notP6);
dot(ceP6, (traceP6|notP6)) > 'lift-$NODENAME-P6.dot';

```

The counter-example generated by **arc** is depicted on figure 4. On the figure one can see that the cage moves instead of opening of the door at the current floor. Actually moves of the cage and opening of of the door may occur in same configuration. To fix the problem we have to forbid moves if the door can be open (i.e. if there is a request for the current floor).

The fix consists simply in the specification of priorities between events

```

// add priorities to fix P6
event {down, up} < open[NB_FLOORS];

```

After applying this patch, **arc** tells us that the new model satisfies  $P_6$  but not yet  $P_7$ :

```

/*
 * Properties for node : Main2
 * # state properties : 1
 *
 * any.s = 1792
 *
 * # trans properties : 1
 *
 * any.t = 18006
 */
TEST(initial,1) [PASSED]
TEST(notP1,0) [PASSED]
TEST(notP2,0) [PASSED]
TEST(notP3,0) [PASSED]
TEST(notP4,0) [PASSED]
TEST(notP5,0) [PASSED]
TEST(notP6,0) [PASSED]
TEST(notP7,0) [FAILED] actual size = 180

```

#### 4.3.2 Fixing $P_7$ : When there are several requests, the software must (if necessary) continue in the same direction than its last move.

As previous we request **arc** to generate a counter-example. We uses the following commands to get the result depicted on figure 5.

```

traceP7 := trace(initial, any.t, src(notP7));
ceP7 := reach(src(traceP7), traceP7|notP7);
dot(ceP7, (traceP7|notP7)) > 'lift-$NODENAME-P7.dot';

```

Actually the lift has no mean to favor the current direction since it does not know it. We modify the model in order to make the controller remember the direction of the last move. To this aim we add a Boolean state variable **climb** use to store last direction. Transitions labelled with **up** and **down** events are modified accordingly:

```

state climb : bool;
init climb := false;

trans
  climb & requestUp          |→ up   → ;
  ~climb & requestDown       |→ down → ;
  ~climb & ~requestDown & requestUp |→ up   → climb:=true;
  climb & ~requestUp & requestDown |→ down → climb:=false;

/*
 * Properties for node : Main3
 * # state properties : 1
 *
 * any.s = 2688
 *
 * # trans properties : 1
 *
 * any.t = 26874
 */
TEST(initial,1) [PASSED]
TEST(notP1,0) [PASSED]
TEST(notP2,0) [PASSED]
TEST(notP3,0) [PASSED]

```

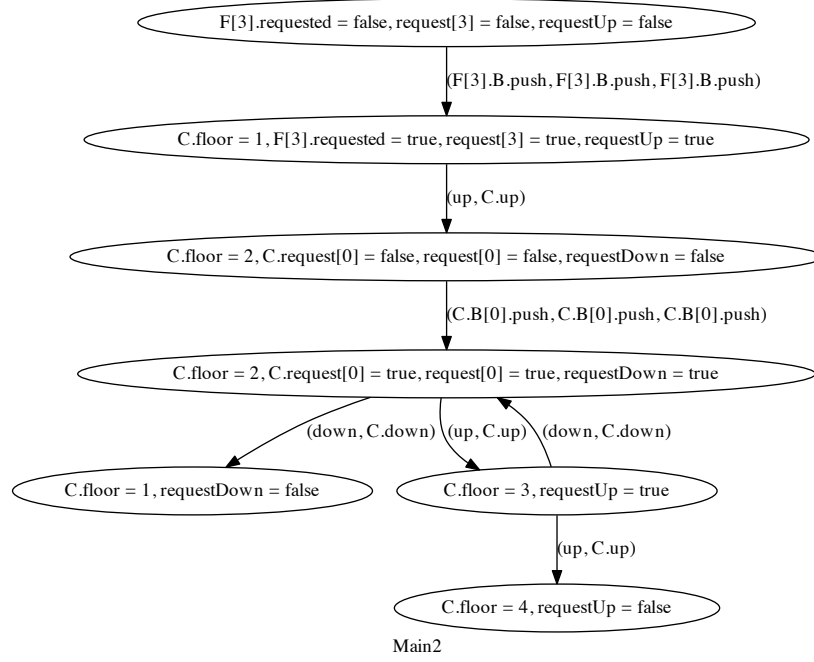


Figure 5: Counter example for  $P_7$  property.

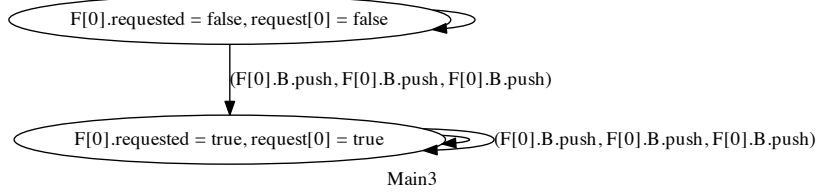


Figure 6: Counter example for  $P_8$  property.

```
TEST(notP4, 0)    [PASSED]
TEST(notP5, 0)    [PASSED]
TEST(notP6, 0)    [PASSED]
TEST(notP7, 0)    [PASSED]
```

With this new fix, the model passes all safety properties. Unfortunately the liveness property is not satisfied as shown by counter-example returned by the following `arc` command:

```
chkctl -to-dot=lift-Main3-P8.dot Main3 "AG([request[0]] => AF([not
request[0]])) <?php for ($i = 1; $i < $NB_FLOORS; $i++) { ?>
and AG([request[<?=$i?>]] => AF([not request[<?=$i?>]])) <?php
} ?>"
```

#### 4.3.3 Fixing $P_8$ : Each request must be honored a day.

The counter-example shown figure 6 is quite disappointing but it is really a counter-example of formula specified section 4.2. Nothing in the model forbids a user to continuously push on a button which has the consequence to maintain the system in the same state. An other counter-example that is more realistic, could have been a user that press the button, the doors open and then close and the user press the button and so on. Even if these are annoying behaviours it is not necessary to forbid them.

Actually  $P_8$  is incomplete and should be: *In a normal use*, each request must be honored a day. What is a normal use of a lift ? In a normal use a lift should always be able to move, eventually; in others words, in any configuration the system should be reach a configuration from which the cage moves. With this setting, we just have to change specification to restrict  $P_8$  to paths describing *normal* uses of the lift. To impose this constraint on  $P_8$  we write the following CTL\* formula for each floor  $i$ :

```
A [(G F ([C.floor = i] and X [C.floor != i]))
=> G([request[i]] => F([not request[i]]))]
```

Recalls on CTL\* formulas:

- a state  $s$  satisfies  $A[\phi]$  if all paths originating in  $s$  satisfy  $\phi$
- a path  $p$  satisfies  $G \phi$ , if all suffixes of  $p$  satisfy  $\phi$ .
- a path  $p$  satisfies  $F \phi$ , if there exists a suffix of  $p$  that satisfies  $\phi$ .
- a path  $p = s.w$  satisfies  $X \phi$  if the suffix  $w$  of  $p$  satisfies  $\phi$ .

Patched  $P_8$  property is satisfied by our last model.

## References

- [1] François Laroussinie. *Logique temporelle avec passé pour la spécification et la vérification des systèmes réactifs*. PhD thesis, Institut National Polytechnique de Grenoble, novembre 1994.