# AltaRica
# Conception d'un ascenseur

Alain Griffault

8 février 2008

# Table des matières

# Chapitre 1

# Le cahier des charges

## 1.1 Introduction

Cet exercice est inspiré de la thèse de François Laroussinie [**?**]. Il consiste, en partant d'une spécification informelle d'un ascenseur à construire un modèle d'ascenseur qui satisfait cet ensemble de propriétés.

## 1.2 Spécifications informelles d'un ascenseur

L'ascenseur que nous allons étudier dessert $n$ étages. La cabine comporte $n$ boutons lumineux (c-à-d lorsqu'un bouton est allumé, il témoigne de l'existence d'une requête pour cet étage) qui permettent de choisir la (ou les) destination(s). De plus, à chaque étage se trouve un bouton de même type qui permet d'appeler l'ascenseur. Lorsque la cabine s'arrête à un étage, la porte s'ouvre alors automatiquement.

Le comportement de cet ascenseur est contrôlé par logiciel. Les changements d'états du contrôleur sont liés aux actions sur les boutons lumineux. Le donneur d'ordre choisira le constructeur qui pourra lui prouver que les comportements suivants sont *"vrais"*.

1. (a) Quand un bouton est enfoncé, le voyant correspondant s'allume.
   (b) Quand la requête correspondante est satisfaite, le voyant s'éteint.
2. A chaque étage, la porte n'est jamais ouverte si la cabine n'est pas là.
3. Chaque requête doit être satisfaite "un jour".
4. L'ascenseur ne dessert que les étages pour lesquels existe une requête.
5. Lorsqu'il n'y a pas de requête, la cabine reste à l'étage où elle se trouve.
6. Lorsque la cabine se déplace, elle doit s'arrêter aux étages par lesquels elle passe si une requête les concerne.
7. Lorsqu'il existe plusieurs requêtes, l'ascenseur doit traiter prioritairement celle(s) permettant de continuer dans la même direction que le dernier déplacement.

# Chapitre 2

# Méthodologie de conception

## 2.1 Les choix initiaux de modélisation

### 2.1.1 Le nombre d'étages

L'ascenseur modélisé dessert 4 étages pour les raisons suivantes :

1. Deux étages ne sont pas suffisant, car le comportement de l'ascenseur sera obligatoirement équitable. Il passera autant de fois à chacun des étages.
2. Trois étages sont peut-être suffisant, mais avec trois étages, tout déplacement a pour départ ou pour arrivée, une extrémité. Cette particularité risque de perturber les résultats pour les propriétés d'équité.
3. Quatre étages semblent suffisant.
4. Cinq et plus ne semblent pas nécessaires. Si l'on partitionne les étages de la manière suivante ({1},{2},{3,...,N-1},{N}) et qu'à chaque sous-ensemble on associe un numéro d'étage d'un ascenseur à quatre étages ({1},1), ({2},2), ({3,4},3), ({5},4), il semble que toutes les propriétés que l'on veut vérifier sur un ascenseur ayant 5 étages ou plus, peuvent l'être sur un ascenseur ayant 4 étages.

Les points précédents ne sont pas des preuves, mais seulement des remarques de bon sens.

### 2.1.2 Les utilisateurs

L'environnement (les personnes) n'est pas modélisé pour la raison suivante :

– L'environnement d'un ascenseur est composé d'un nombre illimité de personnes utilisatrices. Dans le modèle, cet environnement ne peut donc pas être décrit, car il faudrait alors borner le nombre d'utilisateurs. Paradoxalement, ne pas représenter l'environnement (c'est à dire modéliser les requêtes par des actions incontrôlables sur les boutons dont l'émetteur (la personne) est ignorée) permet de représenter les comportements d'un nombre illimité de personnes.

## 2.2 Quel type de modélisation ?

Il y a ici un choix à effectuer concernant la description en AltaRica. Doit-elle se situer au niveau fonctionnel, ou bien au niveau implémentation ?

**niveau fonctionnel :** les deux boutons qui concernent un même étage doivent être fonctionnellement regroupés, le fait que l'un soit sur le palier et l'autre dans la cabine n'a ici que peu d'importance.

**niveau implémentation :** les deux boutons précédant n'utiliseront certainement pas les mêmes moyens pour communiquer avec le logiciel contrôleur du fait que l'un est statique (le palier ne bouge pas) et l'autre non.

*L'exercice*, comme le titre du chapitre l'indique est un exercice de conception. Nous ne cherchons pas ici à reproduire un ascenseur existant afin d'étudier ses propriétés, mais cherchons à en construire un ayant certaines propriétés.

Néanmoins, du fait qu'il sera sans doute plus facile de convaincre le client avec le modèle séparant les boutons, la description de l'ascenseur respecte l'architecture d'une implémentation possible.

## 2.3 Méthode de travail

### 2.3.1 Modélisation des composants de base

La décomposition hiérarchique du système doit faire apparaître des composants de base que l'on décrit en AltaRica si possible indépendemment du contexte d'utilisation.

Ces composants sont généralement suffisemment *petits* pour pouvoir être validés *graphiquement*.

### 2.3.2 Spécification logique du système

Les proriétés attendues du système doivent être précisées.

## 2.4 Conception du système

Tant que les propriétés spécifiées comme devant être correctes ne le sont pas, un nouveau modèle est proposé.

L'examen des traces conduisant dans une situation invalidant les propriétés permet généralement de proposer des modifications.

# Chapitre 3

# Les composants

## 3.1 Quelques constantes

```
const  DernierEtage = 3 ;
domain Etages       = [0,DernierEtage] ;
```

## 3.2 Le nœud Button

### 3.2.1 Le source AltaRica

```
/* les boutons réagissent à :
 *   - une action de l'utilisateur
 *   - un signal d'extinction
*/
node Button
  state
    light : bool : public;
  event
    push : public;
    off;
  trans
    true |- push -> light := true;
    true |- off  -> light := false;
  init
    light := false;
edon
```

### 3.2.2 La sémantique

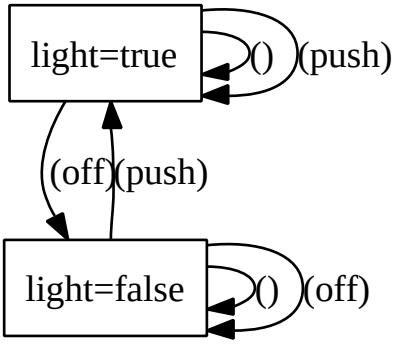


FIG. 3.1 – Le nœud Button

### 3.2.3 Les propriétés

**Les spécifications**

```
with Button do
 quot()                    >  '$NODENAME.dot';
 dead                      := any_s - src(any_t - self_epsilon);
 notCFC                    := any_t - loop(any_t,any_t);
 light                     := [light];
 push                      := label push;
 off                       := label off;
 pushImpliesLight          := tgt(push) - light;
 show(all)                 >  '$NODENAME.prop';
 test(dead,0)              >  '$NODENAME.res';
 test(notCFC,0)            >> '$NODENAME.res';
 test(pushImpliesLight,0) >> '$NODENAME.res';
done
```

**Les résultats**

```
/*
 * # state properties : 5
 *
 * light = 1
 * pushImpliesLight = 0
 * dead = 0
 * any_s = 2
 * initial = 1
 *
 * # transition properties : 8
 *
 * push = 2
 * off = 2
 * notCFC = 0
 * self = 4
 * epsilon = 2
 * not_deterministic = 0
 * any_t = 6
 * self_epsilon = 2
 */
```

**Interprétations**

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
TEST(pushImpliesLight=0) [PASSED]
```

## 3.3 Le nœud Door

### 3.3.1 Le source AltaRica

```
/* les portes réagissent à :
 *   - un signal d'ouverture
 *   - un signal de fermeture
*/
node Door
  state
    closed : bool : public;
  event
    open, close : public;
  trans
    true |- open  -> closed := false;
    true |- close -> closed := true;
  init
    closed := true;
edon
```

### 3.3.2 La sémantique

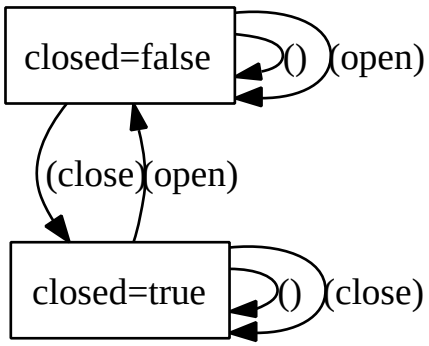


FIG. 3.2 – Le nœud Door

### 3.3.3 Les propriétés

**Les spécifications**

```
with Door do
 quot()                >  '$NODENAME.dot';
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 closed                := [closed];
 open                  := label open;
 close                 := label close;
 show(all)             >  '$NODENAME.prop';
 test(dead,0)          >  '$NODENAME.res';
 test(notCFC,0)        >> '$NODENAME.res';
```

```
done
```

**Les résultats**

```
/*
 * # state properties : 4
 *
 * closed = 1
 * dead = 0
 * any_s = 2
 * initial = 1
 *
 * # transition properties : 8
 *
 * open = 2
 * close = 2
 * notCFC = 0
 * self = 4
 * epsilon = 2
 * not_deterministic = 0
 * any_t = 6
 * self_epsilon = 2
 */
```

**Interprétations**

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

## 3.4 Le nœud Cage

### 3.4.1 Le source AltaRica

```
/* la cabine comprend une porte et un bouton par étage (ici 4)
 * La modélisation doit donner un sens à "la requete est satisfaite"
 * Elle doit choisir entre :
 *   - lors de l'ouverture de la porte
 *   - lors de la fermeture de la porte
 * Nous choisissons la seconde solution (comme dans l'étage)
*/
node Cage
  state
    floor : Etages : parent;
  sub
    D : Door;
    B0, B1, B2, B3 : Button;
  event
    up, down, close0, close1, close2, close3;
  trans
    D.closed |- up   -> floor := floor + 1;
    D.closed |- down -> floor := floor - 1;
    floor = 0 |- close0 -> ;
    floor = 1 |- close1 -> ;
    floor = 2 |- close2 -> ;
    floor = 3 |- close3 -> ;
  sync
```

```
    <close0, D.close, B0.off>;
    <close1, D.close, B1.off>;
    <close2, D.close, B2.off>;
    <close3, D.close, B3.off>;
edon
```

### 3.4.2 La sémantique

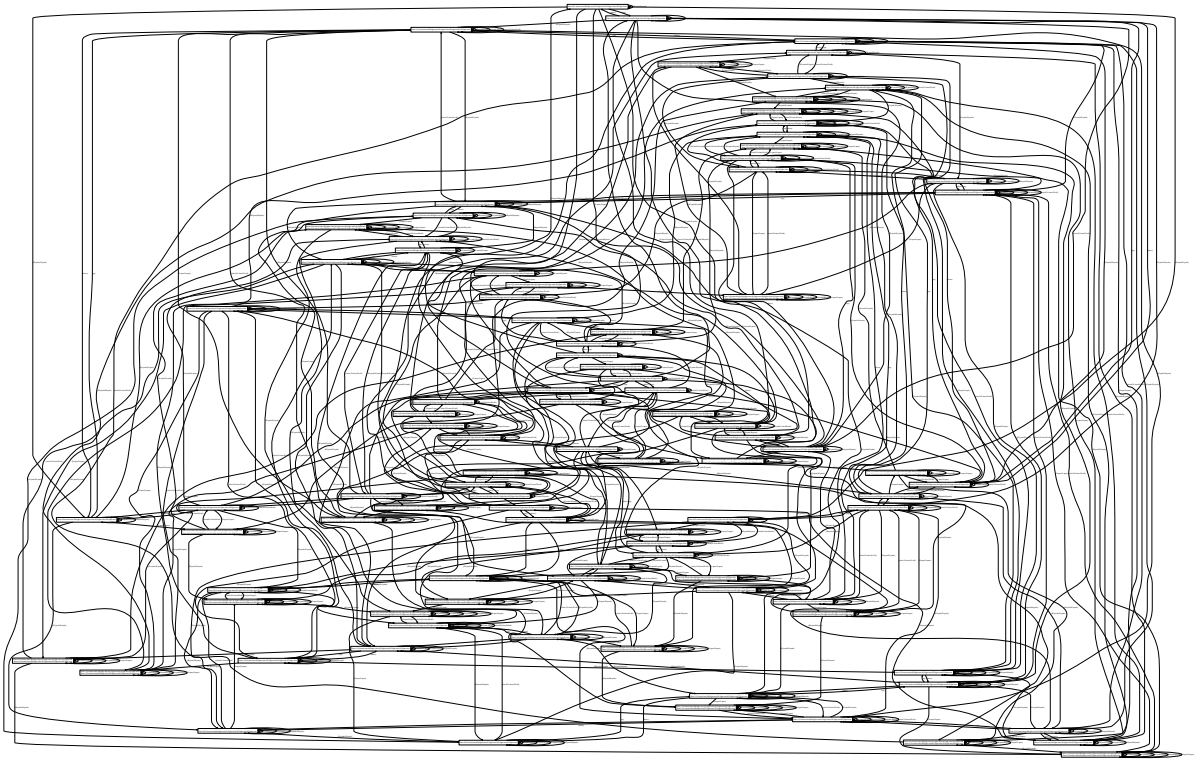

FIG. 3.3 – Le nœud Cage

### 3.4.3 Les propriétés

**Les spécifications**

```
with Cage do
 quot()                > '$NODENAME.dot';
 floor0                := [floor = 0];
 floor1                := [floor = 1];
 floor2                := [floor = 2];
 floor3                := [floor = 3];
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 close0                := label close0;
 close1                := label close1;
 close2                := label close2;
 close3                := label close3;
 open                  := label D.open;
 up                    := label up;
 down                  := label down;
 show(all)             > '$NODENAME.prop';
 test(dead,0)          > '$NODENAME.res';
```

```
 test(notCFC,0)        >> '$NODENAME.res';
done
```

**Les résultats**

```
/*
 * # state properties : 7
 *
 * floor1 = 32
 * floor2 = 32
 * floor3 = 32
 * dead = 0
 * any_s = 128
 * initial = 4
 * floor0 = 32
 *
 * # transition properties : 13
 *
 * open = 128
 * up = 48
 * notCFC = 0
 * down = 48
 * close0 = 32
 * close1 = 32
 * close2 = 32
 * close3 = 32
 * self = 480
 * epsilon = 128
 * not_deterministic = 0
 * any_t = 992
 * self_epsilon = 128
 */
```

**Interprétations**

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

## 3.5 Le nœud Floor

### 3.5.1 Le source AltaRica

```
/* les étages comprennent une porte et un bouton
 * La modélisation doit donner un sens à "la requete est satisfaite"
 * Elle doit choisir entre :
 *   - lors de l'ouverture de la porte
 *   - lors de la fermeture de la porte
 * Nous choisissons la seconde solution
*/
node Floor
  sub
    B : Button;
    D : Door;
  event
    close;
  trans
```

```
    ~D.closed |- close -> ;
  sync
    <close, D.close, B.off>;
edon
```

### 3.5.2 La sémantique

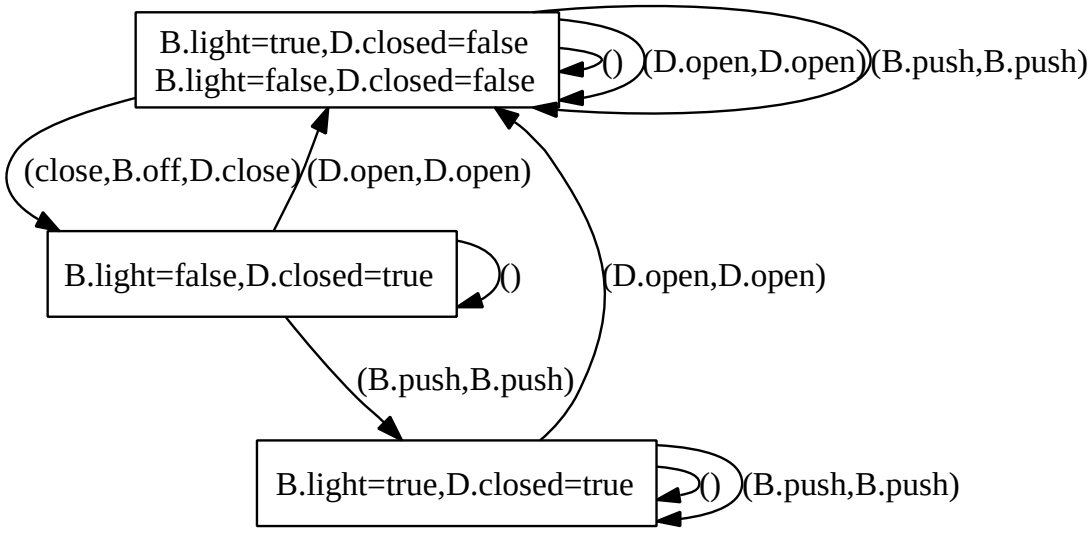


FIG. 3.4 – Le nœud Floor

### 3.5.3 Les propriétés

**Les spécifications**

```
with Floor do
 quot()                > '$NODENAME.dot';
 dead                  := any_s - src(any_t - self_epsilon);
 notCFC                := any_t - loop(any_t,any_t);
 close                 := label close;
 open                  := label D.open;
 show(all)             > '$NODENAME.prop';
 test(dead,0)          > '$NODENAME.res';
 test(notCFC,0)        >> '$NODENAME.res';
done
```

**Les résultats**

```
/*
 * # state properties : 3
 *
 * dead = 0
 * any_s = 4
 * initial = 1
 *
 * # transition properties : 8
 *
 * open = 4
 * close = 2
 * notCFC = 0
```

```
 * self = 8
 * epsilon = 4
 * not_deterministic = 0
 * any_t = 14
 * self_epsilon = 4
 */
```

### Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
```

# Chapitre 4

# Les spécifications du système

## 4.1 Composition d'un système

Un modèle d'ascenseur est composé :
- d'une cabine notée C,
- d'un certain nombre d'étages notés F0, F1, ....

## 4.2 Les propriétés vérifiées

# Chapitre 5

# Le premier système

## 5.1 Le nœud Version1

### 5.1.1 Le source AltaRica

```
/* L'immeuble comprend une cabine et quatre étages
 *  - l'ouverture et la fermeture des portes sont synchronisées
 *  - l'ouverture n'est possible que si une requete existe à cet étage.
 *  - la cabine se déplace si il existe une requete qui le necessite
*/
node Version1
  flow
    mayGoUp, mayGoDown, request0, request1, request2, request3 : bool : private;
  sub
    F0, F1, F2, F3 : Floor;
    C : Cage;
  event
    open0, open1, open2, open3, up, down;
  trans
    (C.floor = 0) & request0 |- open0 -> ;
    (C.floor = 1) & request1 |- open1 -> ;
    (C.floor = 2) & request2 |- open2 -> ;
    (C.floor = 3) & request3 |- open3 -> ;
    mayGoDown |- down    -> ;
    mayGoUp   |- up      -> ;
  sync
    <up,    C.up>;
    <down,  C.down>;
    <open0, C.D.open, F0.D.open>;
    <open1, C.D.open, F1.D.open>;
    <open2, C.D.open, F2.D.open>;
    <open3, C.D.open, F3.D.open>;
    <C.close0, F0.close>;
    <C.close1, F1.close>;
    <C.close2, F2.close>;
    <C.close3, F3.close>;
  assert
    request0 = (C.B0.light | F0.B.light);
    request1 = (C.B1.light | F1.B.light);
    request2 = (C.B2.light | F2.B.light);
    request3 = (C.B3.light | F3.B.light);
    mayGoUp   = ((C.floor=0 & (request3 | request2 | request1)) |
```

```
                (C.floor=1 & (request3 | request2)) |
                (C.floor=2 & (request3)));
    mayGoDown = ((C.floor=3 & (request0 | request1 | request2)) |
                (C.floor=2 & (request0 | request1)) |
                (C.floor=1 & (request0)));
  init
    C.floor := 0;
edon
```

### 5.1.2 Les propriétés

#### Les résultats

```
/*
 * # state properties : 6
 *
 * dead = 0
 * any_s = 1792
 * P1A = 0
 * P1B = 0
 * P2 = 0
 * initial = 1
 *
 * # transition properties : 15
 *
 * notCFC = 0
 * P4 = 0
 * P5 = 0
 * P6 = 1026
 * traceP1A = 0
 * traceP1B = 0
 * traceP2 = 0
 * traceP4 = 0
 * traceP5 = 0
 * traceP6 = 2
 * self = 9984
 * epsilon = 1792
 * not_deterministic = 0
 * any_t = 19800
 * self_epsilon = 1792
 */
```

#### Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
TEST(P1A=0) [PASSED]
TEST(P1B=0) [PASSED]
TEST(P2=0) [PASSED]
TEST(P4=0) [PASSED]
TEST(P5=0) [PASSED]
TEST(P6=0) [FAILED] actual size = 1026
```

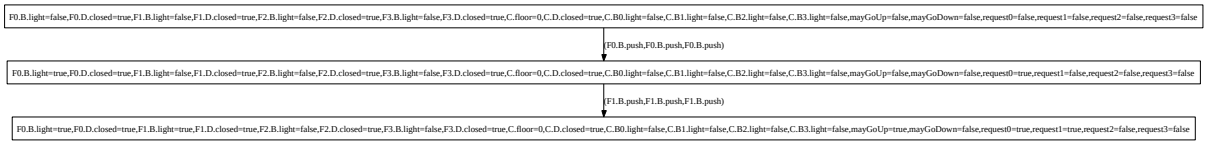


FIG. 5.1 – Le nœud Version1, contre exemple pour P6

# Chapitre 6

# Le second système

## 6.1 Le nœud Version2

### 6.1.1 Le source AltaRica

```
/* L'immeuble comprend une cabine et quatre étages
 *  - l'ouverture et la fermeture des portes sont synchronisées
 *  - l'ouverture n'est possible que si une requete existe à cet étage.
 *  - la cabine se déplace si il existe une requete qui le necessite
 *  - la cabine se déplace si il n'y a pas de requete à cet étage.
*/
node Version2
  flow
    mayGoUp, mayGoDown, request0, request1, request2, request3 : bool : private;
  sub
    F0, F1, F2, F3 : Floor;
    C : Cage;
  event
    open0, open1, open2, open3;
    up < {open0, open1, open2, open3};
    down < {open0, open1, open2, open3};
  trans
    (C.floor = 0) & request0 |- open0 -> ;
    (C.floor = 1) & request1 |- open1 -> ;
    (C.floor = 2) & request2 |- open2 -> ;
    (C.floor = 3) & request3 |- open3 -> ;
    mayGoDown |- down    -> ;
    mayGoUp   |- up      -> ;
  sync
    <up,    C.up>;
    <down,  C.down>;
    <open0, C.D.open, F0.D.open>;
    <open1, C.D.open, F1.D.open>;
    <open2, C.D.open, F2.D.open>;
    <open3, C.D.open, F3.D.open>;
    <C.close0, F0.close>;
    <C.close1, F1.close>;
    <C.close2, F2.close>;
    <C.close3, F3.close>;
  assert
    request0 = (C.B0.light | F0.B.light);
    request1 = (C.B1.light | F1.B.light);
```

```
    request2 = (C.B2.light | F2.B.light);
    request3 = (C.B3.light | F3.B.light);
    mayGoUp   = ((C.floor=0 & (request3 | request2 | request1)) |
                 (C.floor=1 & (request3 | request2)) |
                 (C.floor=2 & (request3)));
    mayGoDown = ((C.floor=3 & (request0 | request1 | request2)) |
                 (C.floor=2 & (request0 | request1)) |
                 (C.floor=1 & (request0)));
  init
    C.floor := 0;
edon
```

### 6.1.2 Les propriétés

**Les résultats**

```
/*
 * # state properties : 7
 *
 * dead = 0
 * any_s = 1792
 * P1A = 0
 * P1B = 0
 * P2 = 0
 * initial = 1
 * P7 = 90
 *
 * # transition properties : 16
 *
 * notCFC = 0
 * P4 = 0
 * P5 = 0
 * P6 = 0
 * traceP1A = 0
 * traceP1B = 0
 * traceP2 = 0
 * traceP4 = 0
 * traceP5 = 0
 * traceP6 = 0
 * traceP7 = 3
 * self = 9984
 * epsilon = 1792
 * not_deterministic = 0
 * any_t = 18774
 * self_epsilon = 1792
 */
```

**Interprétations**

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
TEST(P1A=0) [PASSED]
TEST(P1B=0) [PASSED]
TEST(P2=0) [PASSED]
TEST(P4=0) [PASSED]
TEST(P5=0) [PASSED]
TEST(P6=0) [PASSED]
```

```
TEST(P7=0) [FAILED] actual size = 90
```

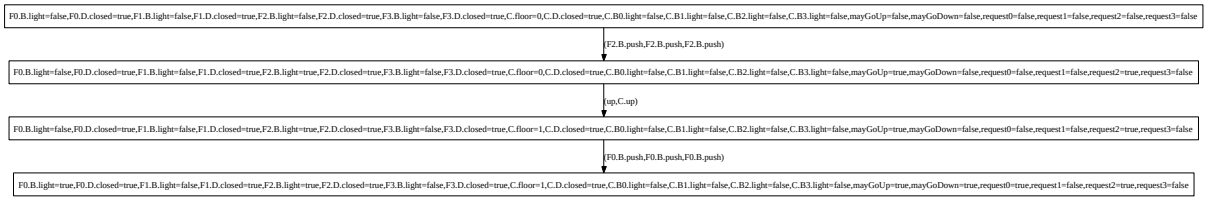


FIG. 6.1 – Le nœud Version2, contre exemple pour P7

# Chapitre 7

# Le troisième système

## 7.1 Le nœud Version3

### 7.1.1 Le source AltaRica

```
/* L'immeuble comprend une cabine et quatre étages
 *  - l'ouverture et la fermeture des portes sont synchronisées
 *  - l'ouverture n'est possible que si une requete existe à cet étage.
 *  - la cabine se déplace si il existe une requete qui le necessite.
 *  - la cabine se déplace si il n'y a pas de requete à cet étage.
 *  - le dernier mouvement est mémorisé pour pouvoir décider du suivant.
 *  - l'initialisation doit etre cohérente.
*/
node Version3
  flow
    mayGoUp, mayGoDown : bool : private;
    request0, request1, request2, request3 : bool : private;
  state
    climb : bool;
  sub
    F0, F1, F2, F3 : Floor;
    C : Cage;
  event
    open0, open1, open2, open3;
    up < {open0, open1, open2, open3};
    down < {open0, open1, open2, open3};
  trans
    (C.floor = 0) & request0 |- open0 -> ;
    (C.floor = 1) & request1 |- open1 -> ;
    (C.floor = 2) & request2 |- open2 -> ;
    (C.floor = 3) & request3 |- open3 -> ;
    climb & mayGoUp               |- up   -> ;
    ~climb & ~mayGoDown & mayGoUp |- up   -> climb := true;
    ~climb & mayGoDown            |- down -> ;
    climb & ~mayGoUp & mayGoDown  |- down -> climb := false;
  sync
    <up,    C.up>;
    <down,  C.down>;
    <open0, C.D.open, F0.D.open>;
    <open1, C.D.open, F1.D.open>;
    <open2, C.D.open, F2.D.open>;
    <open3, C.D.open, F3.D.open>;
```

```
    <C.close0, F0.close>;
    <C.close1, F1.close>;
    <C.close2, F2.close>;
    <C.close3, F3.close>;
  assert
    request0 = (C.B0.light | F0.B.light);
    request1 = (C.B1.light | F1.B.light);
    request2 = (C.B2.light | F2.B.light);
    request3 = (C.B3.light | F3.B.light);
    mayGoUp   = ((C.floor=0 & (request3 | request2 | request1)) |
                 (C.floor=1 & (request3 | request2)) |
                 (C.floor=2 & (request3)));
    mayGoDown = ((C.floor=3 & (request0 | request1 | request2)) |
                 (C.floor=2 & (request0 | request1)) |
                 (C.floor=1 & (request0)));
  init
    C.floor :=0, climb := false;
edon
```

### 7.1.2 Les propriétés

**Les résultats**

```
/*
 * # state properties : 8
 *
 * dead = 0
 * any_s = 2688
 * P1A = 0
 * P1B = 0
 * P2 = 0
 * initial = 1
 * P7 = 0
 * poeleP3A = 8
 *
 * # transition properties : 28
 *
 * notCFC = 0
 * P4 = 0
 * P5 = 0
 * P6 = 0
 * traceP1A = 0
 * traceP1B = 0
 * traceP2 = 0
 * traceP4 = 0
 * traceP5 = 0
 * traceP6 = 0
 * traceP7 = 0
 * attenteServiceB0 = 5904
 * self = 14976
 * attenteServiceB1 = 5952
 * attenteServiceB2 = 5952
 * attenteServiceB3 = 5904
 * attenteServiceF0 = 5904
 * attenteServiceF1 = 5952
 * attenteServiceF2 = 5952
 * attenteServiceF3 = 5904
```

```
 * P3A = 4536
 * P3B = 0
 * traceP3A = 1
 * traceP3B = 0
 * epsilon = 2688
 * not_deterministic = 0
 * any_t = 28026
 * self_epsilon = 2688
 */
```

### Interprétations

```
TEST(dead=0) [PASSED]
TEST(notCFC=0) [PASSED]
TEST(P1A=0) [PASSED]
TEST(P1B=0) [PASSED]
TEST(P2=0) [PASSED]
TEST(P4=0) [PASSED]
TEST(P5=0) [PASSED]
TEST(P6=0) [PASSED]
TEST(P7=0) [PASSED]
TEST(P3A=0) [FAILED] actual size = 4536
TEST(P3B=0) [PASSED]
```

FIG. 7.1 – Le nœud Version3, contre exemple pour P3A