# Projet NuSMV - Traffic light

Aurélien Spinelli - Junior Fernandes

25 mai 2018

# Introduction

Nous avons réaliser un projet sur l'implémentation d'un feu de voiture et piéton où un bouton d'appel intervient. Ce projet a été encadré par Pr. Frédéric Mallet dans le cadre du Master Informatique.
Notre projet consistera dans un premier temps à faire réaliser un « *Simple traffic light* » qui sera composé uniquement d'un feu de voiture et de piéton, puis nous réaliserons une extension de ce dernier en rajoutant un bouton d'appel pour le feu de piéton.

# Table des matières

# Chapitre 1

# Simple traffic light

Dans ce chapitre nous allons décrire la façon dont nous avons implémenté les éléments de notre *Simple traffic light* . Nous expliquerons également les propriétés de suretés et vivacités et les tests réalisés pour vérifier leurs bien-fondés. (cf. simple-traffic-light.smv)

## 1.1 Module

Notre projet sera réalisé à l'aide de module définissant le comportement de nos feux de manière indépendante, pour pouvoir avoir un système extensible.

### 1.1.1 Feu de piéton

Notre feu de piéton sera composé uniquement d'une variable d'état qui définira si le piéton peut oui ou non traverser (resp. vert ou rouge). On définira également les transitions d'état suivant l'entrée qui nous indique si notre piéton peut traverser. Cette entrée TRUE ou FALSE si les voitures sont entrain de rouler ou non. On passe donc de l'état rouge à vert s'il n'y a aucune voiture, et de l'état vert à rouge s'il y a des voitures. Les autres transitions ne sont pas utile à spécifier, d'où le « TRUE : state ; ».

```
1  MODULE pedestrianLight(youcango)
2      VAR
3          state : { red, green };
4
5      ASSIGN
6          next(state) :=  case
7                              state = red & next(youcango) : green;
8                              state = green & next(!youcango) : red;
9                              TRUE : state;
10                         esac;
```

### 1.1.2 Feu de voiture

Notre feu de voiture est composé de trois variables dont une variable d'état qui aura 3 états possibles : vert, jaune, ou rouge (la valeur initiale sera rouge). Elle aura ses transitions définies comme étant un cycle. Une variable tick qui définira le temps passé dans chaque état qui variera entre 0 et 10, car 10 est la plus grande valeur atteinte. Et également une variable, isRed, booléenne qui sera utilisée pour indiquer si aucune voiture ne roule et donc utilisé pour le feu de piéton. Cela permet de ne pas avoir le module de piéton dépendant de celui de voiture.
Les transitions dépendent donc de l'état actuel et du nombre de tick, les ticks sont décrémenté à chaque étape et lorsqu'on atteint la valeur 0 c'est que l'on a passé le temps nécessaire dans un état, on peut donc passer à l'état suivant. Nous prenons le cas où tick atteint 0 pour ne pas à avoir à modifier les valeurs utiles pour les transitions de state et tick.
Pour finir la valeur de tick est donné par le temps que l'on va passer dans un état.
Exemple : Si le feu était rouge et que le tick a atteint la valeur 0, c'est-à-dire que doit passer à l'état vert, la valeur du tick va prendre celle du temps que l'on va passer dans ce dernier.

```
12
13  MODULE carLight
14      VAR
15          state : {red, yellow, green};
16          tick : 0..10;
17          isRed : boolean;
18
19      ASSIGN
20          init(state) := red;
21          next(state) := case
22                              state = red & tick = 0 : green;
23                              state = green & tick = 0 : yellow;
24                              state = yellow & tick = 0 : red;
25                              TRUE : state;
26                          esac;
27
28          init(tick) := 0;
29          next(tick) := case
30                              tick = 0 & state = red : 10;
31                              tick = 0 & state = green : 2;
32                              tick = 0 & state = yellow : 5;
33                              TRUE : tick - 1;
34                          esac;
35
36          init(isRed) := TRUE;
37          --Droit de passage au tick suivant ! Donc anticipé le coup suivant
38          next(isRed) := case
39                              next(state = red): TRUE;
40                              TRUE : FALSE;
41                          esac;
42
```

## 1.2 Exécution

Voici l'exécution d'un cycle de notre système :

```
Trace Type: Simulation
  -> State: 1.1 <-
    car_l.state = red
    car_l.tick = 0
    car_l.isRed = TRUE
    ped_l.state = red
  -> State: 1.2 <-
    car_l.state = green
    car_l.tick = 10
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.3 <-
    car_l.state = green
    car_l.tick = 9
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.4 <-
    car_l.state = green
    car_l.tick = 8
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.5 <-
    car_l.state = green
    car_l.tick = 7
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.6 <-
    car_l.state = green
    car_l.tick = 6
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.7 <-
    car_l.state = green
    car_l.tick = 5
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.8 <-
    car_l.state = green
    car_l.tick = 4
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.9 <-
    car_l.state = green
    car_l.tick = 3
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.10 <-
    car_l.state = green
    car_l.tick = 2
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.11 <-
    car_l.state = green
    car_l.tick = 1
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.12 <-
    car_l.state = green
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.13 <-
    car_l.state = yellow
    car_l.tick = 2
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.14 <-
    car_l.state = yellow
    car_l.tick = 1
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.15 <-
    car_l.state = yellow
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
  -> State: 1.16 <-
    car_l.state = red
    car_l.tick = 5
    car_l.isRed = TRUE
    ped_l.state = green
  -> State: 1.17 <-
    car_l.state = red
    car_l.tick = 4
    car_l.isRed = TRUE
    ped_l.state = green
  -> State: 1.18 <-
    car_l.state = red
    car_l.tick = 3
    car_l.isRed = TRUE
    ped_l.state = green
  -> State: 1.19 <-
    car_l.state = red
    car_l.tick = 2
    car_l.isRed = TRUE
    ped_l.state = green
  -> State: 1.20 <-
    car_l.state = red
    car_l.tick = 1
    car_l.isRed = TRUE
    ped_l.state = green
  -> State: 1.21 <-
    car_l.state = red
    car_l.tick = 0
    car_l.isRed = TRUE
    ped_l.state = green
  -> State: 1.22 <-
    car_l.state = green
    car_l.tick = 10
    car_l.isRed = FALSE
    ped_l.state = red
```

## 1.3 Propriété

Notre système doit également répondre à deux propriétés, celle de sureté et vivacité, qui respectivement assure que le piéton ne peut traverser lorsque les voitures circulent, et qu'à n'importe quel moment notre piéton pourra traversé. Ces propriétés seront exprimer à l'aide de formule CTL et LTL.

### 1.3.1 CTL

#### 1.3.1.1 Propriété de sureté

Cette propriété sera exprimé de la manière suivante :

SPEC AG ! ( ! ( car_l.state = red ) & ( ped_l.state = green ) ) ;

qui se traduit littéralement par : ≪ Il n'existe pas de cas où le feu n'est pas rouge (c-à-d est jaune ou vert) et que le feu de piéton est vert ≫

#### 1.3.1.2 Propriété de vivacité

Cette propriété sera décomposé en deux parties simples :

SPEC AF (car_l.state = green) ;

SPEC AF (ped_l.state = green) ;

Ces propriétés expriment le fait que depuis n'importe quelle situation il existe un moyen d'avoir le feu de voiture et de piéton qui passera au vert.

### 1.3.2 LTL

Nous remarquons que contrairement aux propriétés CTL, les propriétés LTL n'ont pas de quantificateur de chemin, car ces propriétés sont évaluées sur des ≪ linear paths ≫, et une propriété est considérée comme étant vraie à un moment donné si elle est vraie pour tous les chemins commençant par ce dernier.

#### 1.3.2.1 Propriété de sureté

Cette propriété sera exprimé de la manière suivante :

LTLSPEC G ! ( ! ( car_l.state = red ) & ( ped_l.state = green ) ) ;

qui se traduit de la même façon que la propriété CTL, mais on remarque comme vu précédemment que nous ne précisons pas le quantificateur de chemin.

#### 1.3.2.2 Propriété de vivacité

Cette propriété sera décomposé en deux parties simples comme en CTL :

$$\text{LTLSPEC F } (\text{car\_l.state} = \text{green})\,;$$

$$\text{LTLSPEC F } (\text{ped\_l.state} = \text{green})\,;$$

## 1.4 Test

### 1.4.1 CTL & LTL

Les propriétés sont exprimés de deux façon différentes, CTL et LTL, mais en ce qui concerne les tests ils sont identiques.

#### 1.4.1.1 Propriété de sureté

Pour tester la propriété de sureté on va créer des systèmes plus simplistes pour mettre en avant des situations qui ne répondent pas à cette dernière. Nous avons donc mis en place un système où l'initialisation met les deux feux à vert et nous obtenons le résultat suivant :

```
-- specification AG !(!(car_l.state = red) & ped_l.state = green)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    car_l.state = green
    ped_l.state = green
```

Nous voyons ici que dès l'initialisation la propriété est fausse.
Nous avons également fait un test où l'initialisation était correct :

```
-- specification AG !(!(car_l.state = red) & ped_l.state = green)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    car_l.state = red
    ped_l.state = green
  -> State: 1.2 <-
    car_l.state = yellow
```

Nous voyons ici qu'à la première étape la propriété était vraie mais le changement de valeur du feu de voiture étant aléatoire, ce dernier est passé à jaune ce qui fait que notre propriété n'est plus vérifiée.
De plus, nous avons ici les trois différentes situations possibles lorsque le feu de piéton est vert. Notre propriété est donc validée.
Dans le dossier simple-traffic-light-test/CTL/P1 nous avons aussi ajouté deux tests supplémentaires qui mettent en avant un compteur décalé, et un test où le feu suit son cycle normalement mais le feu de piéton change d'état aléatoirement.

#### 1.4.1.2 Propriété de vivacité

Nous testons les deux propriétés séparément car elles sont indépendantes néanmoins elles devront être vraies en même temps dans notre système ce qui permettra d'assurer la vivacité des piétons et des voitures.
Pour ces tests nous mettons en place un système composé uniquement d'un feu de piéton ou de voiture, suivant la partie de la propriété testé, qui reste à rouge. Ces tests indique bien que nos propriétés sont fausses dans ces systèmes, et vraies dans des systèmes restant à vert. Nos propriétés sont donc vérifiées, et étant donné que dans notre système « *Simple traffic light* » les vérifies également la vivacité de notre système est assuré.

# Chapitre 2

# Traffic light extension

Dans ce chapitre nous réalisons une extension de notre « *Simple traffic light* » pour y ajouter un nouveau module qui représente un bouton d'appel pour les piétons. (cf. traffic-light-extension.smv)

## 2.1 Module

### 2.1.1 Buton d'appel

Notre bouton a une variable d'état qui indique si notre bouton est pressé ou non à l'instant t, cette variable dépend d'une entrée booléenne qui représente le fait qu'un piéton appuie sur le bouton. Si personne n'appuie le bouton est dit « relâché ». Mais pour sauvegarder le fait que quelqu'un ait appuyé, nous utilisons une autre variable qui joue ce rôle. Cette dernière repassera à FALSE lorsque l'entrée « isRed » sera vraie, pour indiquer que le feu de voiture est passé au rouge c'est-à-dire que l'appel à été pris en compte et a été traité.

```
5  MODULE button(someonePress, isRed)
6  VAR
7      state : {pressed, released};
8      isPressed : boolean;
9  ASSIGN
10     init(state) := released;
11     next(state) := case
12                         state = released & next(someonePress) : pressed;
13                         state = pressed & next(!someonePress) : released;
14                         TRUE : state;
15                     esac;
16
17     init(isPressed) :=FALSE;
18     next(isPressed) := case
19                         isRed : FALSE;
20                         next(state) = pressed : TRUE;
21                         TRUE : isPressed;
22                     esac;
```

### 2.1.2 Feu de voiture

Les variables du feu de voiture n'ont pas changé mais les transitions des variables oui. Maintenant le feu reste dans l'état vert jusqu'à ce que nous reçevions un appel. Cet appel est pris en compte seulement quand nous sommes dans l'état vert, car dans les autres états il n'a aucune influence. Si nous n'avons aucun appel le feu reste dans l'état vert.

```
38 MODULE carLight(buttonHasBeenPress)
39     VAR
40         state : {red, yellow, green};
41         tick : 0..10;
42         isRed : boolean;
43 
44     ASSIGN
45         init(state) := red;
46         next(state) := case
47                             state = red & tick = 0 : green;
48                             state = green & tick = 0 & buttonHasBeenPress : yellow;
49                             state = yellow & tick = 0 : red;
50                             TRUE : state;
51                         esac;
52 
53 
54 
55         init(tick) := 0;
56         next(tick) := case
57                             tick = 0 & state = red : 10;
58                             tick = 0 & next(state = yellow) : 2;
59                             tick = 0 & state = green : 0;
60                             tick = 0 & state = yellow : 5;
61                             TRUE : tick - 1;
62                         esac;
63 
64         init(isRed) := TRUE;
65         --Droit de passage au tick suivant ! Donc anticipé le coup suivant
66         next(isRed) := case
67                             next(state = red) : TRUE;
68                             TRUE : FALSE;
69                         esac;
```

## 2.2 Exécution

Nous allons réaliser l'exécution de notre système dans deux cas : sans appel de piéton pour montrer que le feu reste indéfiniment au vert, et une exécution avec appel pour mettre en avant les transitions.

### 2.2.1 Sans appel

```
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    car_l.state = red
    car_l.tick = 0
    car_l.isRed = TRUE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.2 <-
    car_l.state = green
    car_l.tick = 10
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.3 <-
    car_l.state = green
    car_l.tick = 9
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.4 <-
    car_l.state = green
    car_l.tick = 8
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.5 <-
    car_l.state = green
    car_l.tick = 7
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.6 <-
    car_l.state = green
    car_l.tick = 6
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.7 <-
    car_l.state = green
    car_l.tick = 5
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.8 <-
    car_l.state = green
    car_l.tick = 4
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.9 <-
    car_l.state = green
    car_l.tick = 3
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.10 <-
    car_l.state = green
    car_l.tick = 2
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.11 <-
    car_l.state = green
    car_l.tick = 1
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.12 <-
    car_l.state = green
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.13 <-
    car_l.state = green
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.14 <-
    car_l.state = green
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
```

### 2.2.2 Avec appel

```
Trace Description: Simulation Trace
Trace Type: Simulation
  -> State: 1.1 <-
    car_l.state = red
    car_l.tick = 0
    car_l.isRed = TRUE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.2 <-
    car_l.state = green
    car_l.tick = 10
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = FALSE
  -> State: 1.3 <-
    car_l.state = green
    car_l.tick = 9
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.4 <-
    car_l.state = green
    car_l.tick = 8
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.5 <-
    car_l.state = green
    car_l.tick = 7
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = TRUE
  -> State: 1.6 <-
    car_l.state = green
    car_l.tick = 6
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.7 <-
    car_l.state = green
    car_l.tick = 5
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = TRUE
  -> State: 1.8 <-
    car_l.state = green
    car_l.tick = 4
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.9 <-
    car_l.state = green
    car_l.tick = 3
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.10 <-
    car_l.state = green
    car_l.tick = 2
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.11 <-
    car_l.state = green
    car_l.tick = 1
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.12 <-
    car_l.state = green
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.13 <-
    car_l.state = yellow
    car_l.tick = 2
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.14 <-
    car_l.state = yellow
    car_l.tick = 1
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.15 <-
    car_l.state = yellow
    car_l.tick = 0
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = TRUE
  -> State: 1.16 <-
    car_l.state = red
    car_l.tick = 5
    car_l.isRed = TRUE
    ped_l.state = green
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = TRUE
  -> State: 1.17 <-
    car_l.state = red
    car_l.tick = 4
    car_l.isRed = TRUE
    ped_l.state = green
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.18 <-
    car_l.state = red
    car_l.tick = 3
    car_l.isRed = TRUE
    ped_l.state = green
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = FALSE
  -> State: 1.19 <-
    car_l.state = red
    car_l.tick = 2
    car_l.isRed = TRUE
    ped_l.state = green
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.20 <-
    car_l.state = red
    car_l.tick = 1
    car_l.isRed = TRUE
    ped_l.state = green
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 1.21 <-
    car_l.state = red
    car_l.tick = 0
    car_l.isRed = TRUE
    ped_l.state = green
    isPushed = TRUE
    ped_b.state = pressed
    ped_b.isPressed = FALSE
  -> State: 1.22 <-
    car_l.state = green
    car_l.tick = 10
    car_l.isRed = FALSE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
```

## 2.3 Propriété

Dans notre nouveau système, la propriété de sureté reste la même, nous devons continuer à assurer que le piéton ne se fera pas assuré. Mais la propriété de vivacité se voit changé.

### 2.3.1 CTL

#### 2.3.1.1 Propriété de vivacité

Notre propriété de vivacité est maintenant dépendante du l'appel, car le feu de piéton ne passera plus au vert si personne ne fait d'appel. Néanmoins du coté des voitures la propriété n'a pas changé car le feu respecte un cycle équivalent à la version précédente. Nous allons donc nous attarder sur la propriété pour les piétons :

```
--CTL P2: A chaque occurence de button.isPressed (coté piéton)
SPEC AG (ped_b.isPressed -> AF ped_l.state = green);
```

La propriété indique que pour tout les états où le bouton d'appel a été pressé, nous avons dans le future le feu de piéton qui passe au vert.

### 2.3.2 LTL

#### 2.3.2.1 Propriété de vivacité

La propriété en LTL est équivalente à celle en CTL sans le quantificateur de chemin comme vu précédemment :

```
--LTL P2: A chaque occurence de button.isPressed (coté piéton)
LTLSPEC G (ped_b.isPressed -> F ped_l.state = green);
```

## 2.4 Test

### 2.4.1 CTL & LTL

#### 2.4.1.1 Propriété de vivacité

Pour montrer la cohérence de notre système nous testons également la propriété de vivacité du système précédant :

$$\text{SPEC AF } (\text{ped\_l.state} = \text{green})\,;$$

$$\text{SPEC AF } (\text{ped\_l.state} = \text{green})\,;$$

Et notre test montre que la propriété est fausse dans le cas où personne ne fait d'appel, car cet appel est géré de manière aléatoire dans notre test. Nous voyons donc que sans appel le feu reste au rouge d'où la réfutation de nos propriétés.
Nous allons tester les propriétés énoncées dans les points précédant. Nous allons changer le système de tel sorte à ce que le feu de piéton ne passe plus au vert même après un appel. Nos propriétés seront fausses dans ce système.

```
-- specification  G (ped_b.isPressed ->  F ped_l.state = green)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
  -- Loop starts here
  -> State: 2.1 <-
    car_l.state = red
    car_l.tick = 0
    car_l.isRed = TRUE
    ped_l.state = red
    isPushed = FALSE
    ped_b.state = released
    ped_b.isPressed = FALSE
  -> State: 2.2 <-
    car_l.state = green
    car_l.tick = 10
    car_l.isRed = FALSE
    isPushed = TRUE
    ped_b.state = pressed
  -> State: 2.3 <-
    car_l.tick = 9
    ped_b.isPressed = TRUE
  -> State: 2.4 <-
    car_l.tick = 8
  -> State: 2.5 <-
    car_l.tick = 7
  -> State: 2.6 <-
    car_l.tick = 6
  -> State: 2.7 <-
    car_l.tick = 5
  -> State: 2.8 <-
    car_l.tick = 4
  -> State: 2.9 <-
    car_l.tick = 3
  -> State: 2.10 <-
    car_l.tick = 2
  -> State: 2.11 <-
    car_l.tick = 1
  -> State: 2.12 <-
    car_l.tick = 0
  -> State: 2.13 <-
    car_l.state = yellow
    car_l.tick = 2
  -> State: 2.14 <-
    car_l.tick = 1
  -> State: 2.15 <-
    car_l.tick = 0
  -> State: 2.16 <-
    car_l.state = red
    car_l.tick = 5
    car_l.isRed = TRUE
  -> State: 2.17 <-
    car_l.tick = 4
    ped_b.isPressed = FALSE
  -> State: 2.18 <-
    car_l.tick = 3
  -> State: 2.19 <-
    car_l.tick = 2
  -> State: 2.20 <-
    car_l.tick = 1
  -> State: 2.21 <-
    car_l.tick = 0
    isPushed = FALSE
    ped_b.state = released
```

On voit ici que le feu de piéton reste au rouge. Le test pour la propriété CTL donne le même résultat.

# Conclusion

Nous avons donc réussi à exposer des cas particuliers pour que nos propriétés soient fausses, et donc montrer que si ces dernières sont vraies dans nos systèmes c'est que ces situations ne se produisent jamais. De plus nos systèmes ont des modules indépendants se qui nous permet par exemple de simuler des carrefours facilement.