Université Libre de Bruxelles

INFO-F410 - Embedded Systems Design

Electric-Bike Project

Student (MA1 Computer Science)

SWIRYDOWICZ Szymon
000477108

Lecturer / TAs

RASKIN Jean-François

BALACHANDER
Mrudula

CHAKRABORTY
Debraj

KHALIMOV Ayrat

April - May 2022

Contents

Chapter 1

Lustre

1.1 Introduction

This part of the project focuses on Lustre, which is "a formally defined, declarative, and synchronous dataflow programming language for programming reactive systems".[1]

As the project description required adding parts of the code to this report, the full code can be found in section 1.7. Explicit comments were added to clearly separate the five given tasks. Additionally, the corresponding `main.lus` code file has been joined with this report.

The verification part was executed using the official Kind 2 VS Code extension[2], which allows not only to validate stated properties, but also allows to simulate a node's behavior. The requested verification has been added to section 1.8.

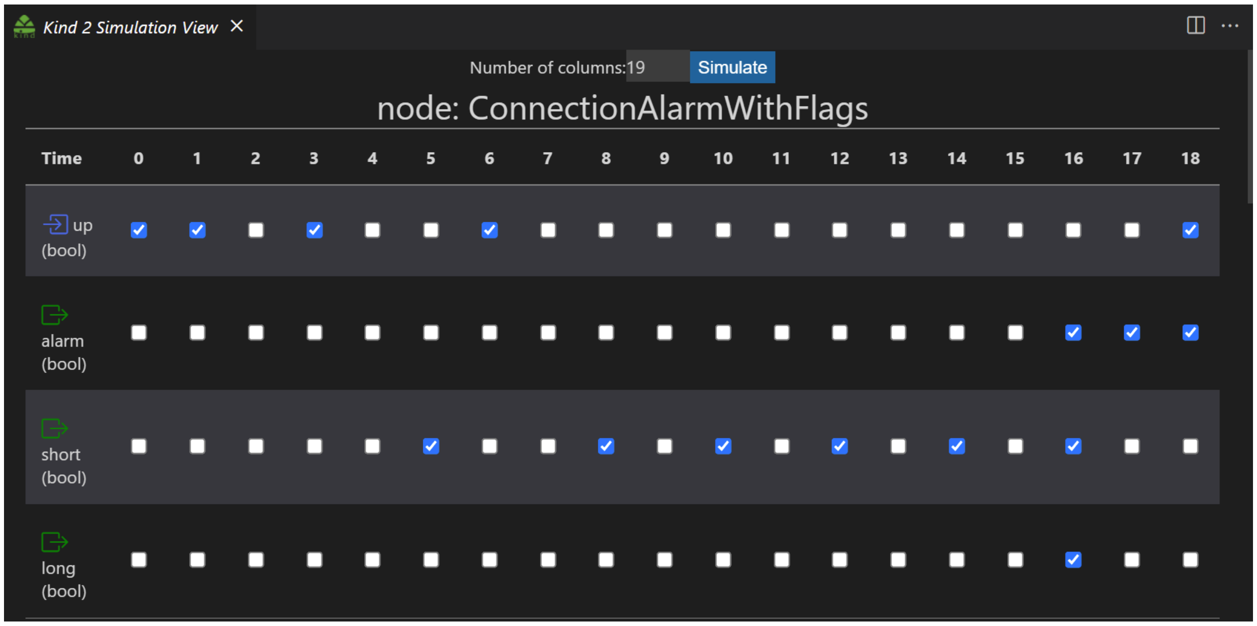
1.2 Task L1

1.2.1 Diagrams

The following diagrams were obtained through the Kind 2 Extension's (cfr. Introduction) Simulation View.

[1]Source: `https://en.wikipedia.org/wiki/Lustre_(programming_language)`

[2]Kind2 VSC extension: `https://marketplace.visualstudio.com/items?itemName=kind2-mc.vscode-kind2`



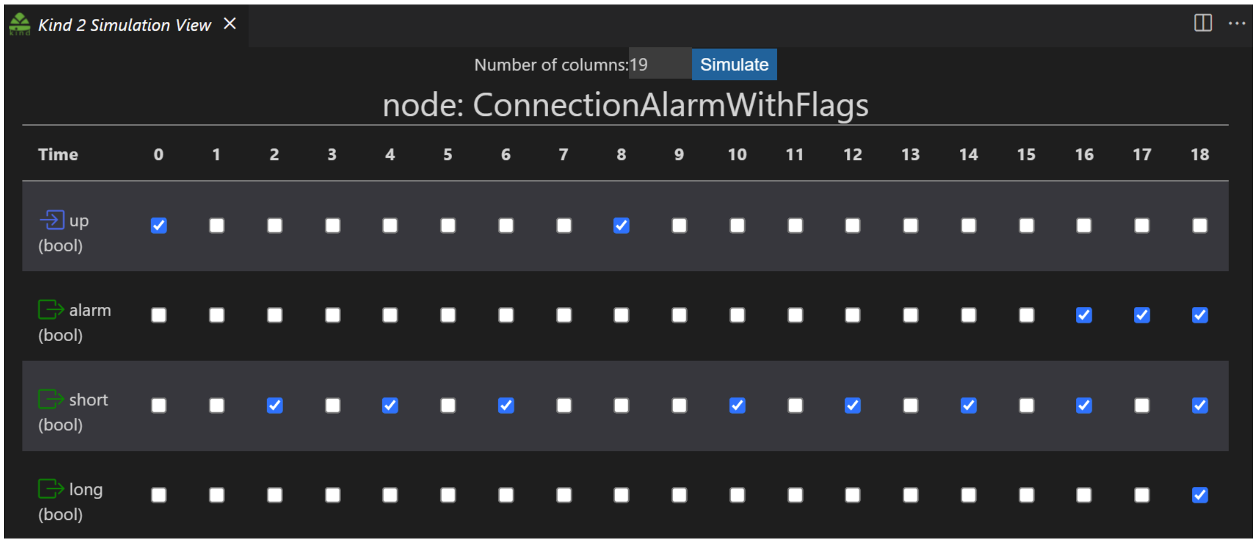
node: ConnectionAlarmWithFlags

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
up (bool)	✓	✓	☐	✓	☐	☐	✓	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓
alarm (bool)	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓	✓	✓
short (bool)	☐	☐	☐	☐	☐	✓	☐	☐	✓	☐	✓	☐	✓	☐	✓	☐	✓	☐	☐
long (bool)	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓	☐	☐

Task L1: 1 Long drop alarm.

By making `UP` true at times 0, 1, 3, 6 and 18, the system raises the SHORT flag at times 5, 8, 10, 12, 14 and 16, the LONG flag only at time 16, and in consequence of the long drop, the ALARM flag also rises at times 16, 17 and 18. This corresponds precisely, in inputs and outputs, to the diagram showcased in the Lustre project description.

Additionally, the following diagram has been added where the ALARM flag raises because of 7 short drops.



node: ConnectionAlarmWithFlags

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
up (bool)	✓	☐	☐	☐	☐	☐	☐	☐	✓	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
alarm (bool)	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓	✓	✓
short (bool)	☐	☐	✓	☐	✓	☐	✓	☐	☐	☐	✓	☐	✓	☐	✓	☐	✓	☐	✓
long (bool)	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓

Task L1: 7 Short drops alarm.

Note: Those diagrams could have been obtained using Luciole's sim2chro tool. However, for unknown reasons, that tool only raised errors on my machine. In consequence, I prefered to simply include the above simulation view as it implements the same behaviour (despite looking a bit less professional).

1.2.2 LTL

We have to translate "once the alarm is set, it stays set forever" into LTL.

I would write it in LTL using the Globally (Always) operator and the implication operator (response) as follows:

```
G ( alarm -> X alarm )
```

Which in other words means that no matter when, if alarm is true at one position, it also has to be true at the next position. Since the alarm in that next position will also be true and the global implication will still hold, basically the alarm will continue forever.

1.2.3 Verification

The property "*once the alarm is set, it stays set forever*", is described in the following way:

```
--%PROPERTY (false->pre(alarm) = true) => (alarm = true);
```

In other words, it ensures that if the alarm has been set at time $(t-1)$, it must also be set at time t. And inductively, since alarm at t must be `true`, the alarm at $(t+1)$ will also have to be `true`.

1.3 Task L2

1.3.1 Verification

Two properties had to be verified.

First, "whenever the cyclists brakes hard, the clinch should happen". This is verified using the following two lines:

```
--%PROPERTY (brk > BRK_SOFT and ConnectionAlarm(up)) => (clinch = true);
--%PROPERTY (brk > BRK_SOFT and not ConnectionAlarm(up)) => (clinch = true);
```

It simply states that when the brake is hard, a clinch must happen. The "`(not) ConnectionAlarm(up)`" part is clearly not necessary, but it has been added to better put in contrast the following two properties:

```
  --%PROPERTY (brk > 0.0 and brk <= BRK_SOFT and not ConnectionAlarm(up)) => (clinch = false);
  --%PROPERTY (brk > 0.0 and brk <= BRK_SOFT and ConnectionAlarm(up)) => (clinch = true);
```

that make sure that soft breaking is true only when the connection alarm is also true.

Next, "whenever the clinch happens, BRK is not zero".

```
  --%PROPERTY (clinch = true) => (brk <> 0.0);
```

Which simply tells that if clinch happens (is true) then it implies that breaking is not zero.

1.4 Task L3

1.4.1 Verification

Although, not demanded, the following two properties ensure that the node works as expected.

```
  --%PROPERTY (not up) => (brk_rcvd = noise);
  --%PROPERTY (up) => (brk_rcvd = brk);
```

1.5 Task L4

This part was implemented by first creating two Booleans: `assist_allowed` and `regen_allowed`, which are respectively for checking assistance and regenerative braking. After a series of conditionals setting those true or false, we determine the assistance level. If both of them happen to be true, then the regenerative braking will be prioritized, except in the case where the received `brk_rcvd` is set to 0.0, in which case assistance goes first.

Verification Different properties were added to test the correctness of the node. Those can be found in section 1.7.

1.6 Task L5

1.6.1 Verification

If the connection alarm is true and the braking happens, brake-cable clinch must happen (i.e., CLINCH must be true);

```
--%PROPERTY ((alarm = true) and (brk > 0.0)) => (clinch = true);
```

When the connection alarm is true, the regenerative braking cannot happen; (reminder: regenerative braking happens when assist level if below zero)

```
--%PROPERTY (alarm = true) => (asst_lvl >= 0.0);
```

Regenerative braking occurs only if the braking is requested by the human;

```
--%PROPERTY (asst_lvl < 0.0) => (brk > 0.0);
```

When the speed is above 7 m/s, the assistance is negative or zero;

```
 --%PROPERTY (speed > MAX_SPEED) => (asst_lvl <= 0.0);
```

The assistance level is always equal to one of the following values: 0, -BRK, HUMAN.

```
 --%PROPERTY (asst_lvl = 0.0) or (asst_lvl = -brk) or (asst_lvl = human);
```

If the braking and assisting are both requested at the same time and the connection is up and there is no alarm, then the motor should engage the regenerative braking.

```
 --%PROPERTY ((brk > 0.0) and (alarm = false) and (up = true)) => (asst_lvl < 0.0);
```

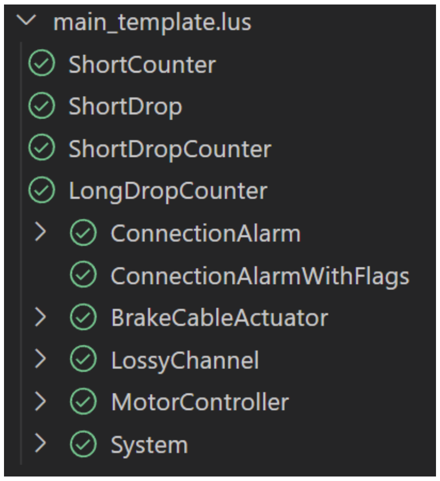
1.7 Code

```
---------------------------------------------------------------------
-- Constants:
const
  SHORT_DROP_NUM:int = 2;
  LONG_DROP_NUM:int = 10;
  MAX_NOF_SHORT_DROPS:int = 7;
  MAX_SPEED:real = 7.0;
  BRK_SOFT:real = 1.0;

---------------------------------------------------------------------

----------------------
-- ::: TASK L1 ::: ---
----------------------
-- private to the ShortDrop node
-- counts number of 'time' units since UP is false
node ShortCounter(up:bool) returns (count:int);
let
  count = ( if up then 0
          else if not up then (0 -> pre(count) + 1)
          else 0 -> pre(count)
  );
tel

node ShortDrop(up:bool) returns (drop:bool);
var can_short:bool;
let
  can_short = ShortCounter(up) mod SHORT_DROP_NUM = 0;
  -- we check if i and i-1 of UP are false,
  -- if so, set the shortDrop to true iff we didn't have a short drop before (modulo)
  drop = ( if not up and not (true->pre(up)) then can_short
          else false
  );
tel

-- counts the number of ShortDrops encountered during program execution
node ShortDropCounter(up:bool) returns (count:int);
let
  count = ( if ShortDrop(up) then (0->pre(count) + 1)
          else 0->pre(count)
  );
tel

node LongDropCounter(up:bool) returns (count:int);
let
  count = ( if up then 0
          else if not up then (0->pre(count) + 1)
          else 0->pre(count)
  );
tel

node ConnectionAlarm(up:bool) returns (alarm:bool);
let
  alarm = ( if false->pre(alarm) = true then true -- ensures alarm stays ON once set
          else if ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS then true -- looks for 7 short
              drops of 2 units
          else if LongDropCounter(up) >= LONG_DROP_NUM then true -- looks for 1 long drop of 10
              units
          else false
  );
  --%PROPERTY (false->pre(alarm) = true) => (alarm = true);
  --%PROPERTY (ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) => (alarm = true);
  --%PROPERTY (LongDropCounter(up) >= LONG_DROP_NUM) => (alarm = true);
tel

-- COPY OF THE ConnectionAlarm FOR THE DIAGRAM TASK (+= short/long flags)
node ConnectionAlarmWithFlags(up:bool) returns (alarm:bool; short:bool; long:bool);
let
  alarm = ( if false->pre(alarm) = true then true
          else if ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS then true
          else if LongDropCounter(up) >= LONG_DROP_NUM then true
          else false
  );
  short = ShortDrop(up);
  long = (not up) and LongDropCounter(up) mod LONG_DROP_NUM = 0;
tel

----------------------
-- ::: TASK L2 ::: ---
----------------------
node BrakeCableActuator(brk:real; up:bool) returns (clinch:bool);
let
  clinch = ( if ConnectionAlarm(up) and (brk <= BRK_SOFT and brk > real(0.0)) then true -- SOFT
      BREAK + ALARM
            else if brk > BRK_SOFT then true -- HARD BREAK
            else false
  );
  --%PROPERTY (clinch = true) => (brk <> 0.0);
  --%PROPERTY (brk > 0.0 and brk <= BRK_SOFT and not ConnectionAlarm(up)) => (clinch = false);
  --%PROPERTY (brk > 0.0 and brk <= BRK_SOFT and ConnectionAlarm(up)) => (clinch = true);
  --%PROPERTY (brk > BRK_SOFT and ConnectionAlarm(up)) => (clinch = true);
  --%PROPERTY (brk > BRK_SOFT and not ConnectionAlarm(up)) => (clinch = true);
tel

----------------------
-- ::: TASK L3 ::: ---
----------------------
node LossyChannel(noise,brk:real; up:bool) returns (brk_rcvd:real);
let
  brk_rcvd = (if up then brk else noise);
  --%PROPERTY (not up) => (brk_rcvd = noise);
  --%PROPERTY (up) => (brk_rcvd = brk);
tel

----------------------
-- ::: TASK L4 ::: ---
----------------------
node MotorController(brk_rcvd:real; up:bool; human,speed,battery:real) returns (asst_lvl:real);
var
  assist_allowed:bool;
  regen_allowed:bool;
let
  -- * assumption: if both assist and regen are true then regen is prioritised (exception:
      brk_rcvd = 0.0)
  assist_allowed = (if ConnectionAlarm(up) then false
                 else if speed > MAX_SPEED then false
                 else if battery = 0.0 then false
                 else true
  );
  regen_allowed = ( if ConnectionAlarm(up) then false
                 else if not up then false
                 else true
  );
  asst_lvl = ( if regen_allowed and assist_allowed and brk_rcvd = 0.0 then human
               else if regen_allowed then -brk_rcvd
               else if assist_allowed then human
               else 0.0
  );
  --%PROPERTY (not up) => (regen_allowed = false);
  --%PROPERTY (speed > MAX_SPEED) => (assist_allowed = false);
  --%PROPERTY (battery = 0.0) => (assist_allowed = false);
  --%PROPERTY ((regen_allowed = true) and (assist_allowed = false)) => (asst_lvl = -brk_rcvd);
  --%PROPERTY ((regen_allowed = false) and (assist_allowed = true)) => (asst_lvl = human);
  --%PROPERTY ((brk_rcvd = 0.0) and (assist_allowed = true) and ((regen_allowed = true))) =>
      (asst_lvl = human);
  --%PROPERTY ((brk_rcvd <> 0.0) and (assist_allowed = true) and ((regen_allowed = true))) =>
      (asst_lvl = -brk_rcvd);
tel

----------------------
-- ::: TASK L5 ::: ---
----------------------
node System(up:bool; noise,brk,human,speed,battery:real) returns (clinch:bool; asst_lvl:real);
var
  alarm:bool;
  brk_rcvd:real;
  temp_assist:real;
let
  brk_rcvd = LossyChannel(noise,brk,up);
  alarm = ConnectionAlarm(up);

  clinch = (if alarm and (brk_rcvd > 0.0) then true
           else BrakeCableActuator(brk,up)
  );

  temp_assist = MotorController(brk_rcvd,up,human,speed,battery);
  asst_lvl = ( if alarm and (temp_assist < 0.0) then 0.0
               else if (temp_assist < 0.0) and (brk <= 0.0) then 0.0
               else if (speed > MAX_SPEED) and (temp_assist > 0.0) then 0.0
               else if not (temp_assist = 0.0) and not (temp_assist = -brk) and not (temp_assist =
                   human) then 0.0
               else if (brk > 0.0) and (temp_assist > 0.0) and up and (not alarm) then -brk
               else temp_assist
  );

  --%MAIN;
  --%PROPERTY ((alarm = true) and (brk > 0.0)) => (clinch = true);
  --%PROPERTY (alarm = true) => (asst_lvl >= 0.0);
  --%PROPERTY (asst_lvl < 0.0) => (brk > 0.0);
  --%PROPERTY (speed > MAX_SPEED) => (asst_lvl <= 0.0);
  --%PROPERTY (asst_lvl = 0.0) or (asst_lvl = -brk) or (asst_lvl = human);
  --%PROPERTY ((brk > 0.0) and (alarm = false) and (up = true)) => (asst_lvl < 0.0);
tel
```

1.8 Verification - outputs

1.8.1 Summary

All the properties present in (the code of) section 1.7 were tested successfully by Kind 2.



```
main_template.lus
  ✓ ShortCounter
  ✓ ShortDrop
  ✓ ShortDropCounter
  ✓ LongDropCounter
  > ✓ ConnectionAlarm
    ✓ ConnectionAlarmWithFlags
  > ✓ BrakeCableActuator
  > ✓ LossyChannel
  > ✓ MotorController
  > ✓ System
```

Kind 2 verification of all the properties

1.8.2 Task L1

```
Analyzing ConnectionAlarm
  with First top: 'ConnectionAlarm'
             subsystems
               | concrete: ShortDropCounter, ShortDrop, ShortCounter, LongDropCounter

<Success> Property ((LongDropCounter(up) >= LONG_DROP_NUM) => (alarm = true)) is valid by property
    directed reachability after 0.082s.

<Success> Property ((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) => (alarm = true)) is valid by
    property directed reachability after 0.082s.

<Success> Property ((false -> ((pre alarm) = true)) => (alarm = true)) is valid by property
    directed reachability after 0.082s.
```

```
Summary of properties:

((LongDropCounter(up) >= LONG_DROP_NUM) => (alarm = true)): valid (at 1)
((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) => (alarm = true)): valid (at 1)
((false -> ((pre alarm) = true)) => (alarm = true)): valid (at 1)
```

1.8.3 Task L2

```
Analyzing BrakeCableActuator
  with First top: 'BrakeCableActuator'
             subsystems
               | concrete: ShortDropCounter, ShortDrop, ShortCounter, LongDropCounter,
                   ConnectionAlarm
```

```
<Success> Property (((brk > BRK_SOFT) and (not ConnectionAlarm(up))) => (clinch = true)) is valid
    by property directed reachability after 0.083s.

<Success> Property (((brk > BRK_SOFT) and ConnectionAlarm(up)) => (clinch = true)) is valid by
    property directed reachability after 0.083s.

<Success> Property ((((brk > 0.0) and (brk <= BRK_SOFT)) and ConnectionAlarm(up)) =>
  (clinch = true)) is valid by property directed reachability after 0.083s.

<Success> Property ((((brk > 0.0) and (brk <= BRK_SOFT)) and (not ConnectionAlarm(up))) =>
  (clinch = false)) is valid by property directed reachability after 0.083s.

<Success> Property ((clinch = true) => (brk <> 0.0)) is valid by property directed reachability
    after 0.083s.

<Success> Property ConnectionAlarm[l78c18].((false -> ((pre alarm) = true)) => (alarm = true)) is
    valid by property directed reachability after 0.083s.

<Success> Property ConnectionAlarm[l78c18].((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) =>
    (alarm = true)) is valid by property directed reachability after 0.083s.

<Success> Property ConnectionAlarm[l78c18].((LongDropCounter(up) >= LONG_DROP_NUM) => (alarm =
    true)) is valid by property directed reachability after 0.083s.
```

```
Summary of properties:

(((brk > BRK_SOFT) and (not ConnectionAlarm(up))) => (clinch = true)): valid (at 1)
(((brk > BRK_SOFT) and ConnectionAlarm(up)) => (clinch = true)): valid (at 1)
((((brk > 0.0) and (brk <= BRK_SOFT)) and ConnectionAlarm(up)) =>
  (clinch = true)): valid (at 1)
((((brk > 0.0) and (brk <= BRK_SOFT)) and (not ConnectionAlarm(up))) =>
  (clinch = false)): valid (at 1)
((clinch = true) => (brk <> 0.0)): valid (at 1)
ConnectionAlarm[l78c18].((false -> ((pre alarm) = true)) => (alarm = true)): valid (at 1)
ConnectionAlarm[l78c18].((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) => (alarm = true)): valid
    (at 1)
ConnectionAlarm[l78c18].((LongDropCounter(up) >= LONG_DROP_NUM) => (alarm = true)): valid (at 1)
```

1.8.4 Task L5

```
Analyzing System
  with First top: 'System'
            subsystems
              | concrete: ShortDropCounter, ShortDrop, ShortCounter, MotorController,
                  LossyChannel, LongDropCounter,
                         ConnectionAlarm, BrakeCableActuator

<Success> Property (((asst_lvl = 0.0) or (asst_lvl = (- brk))) or (asst_lvl = human)) is valid by
    inductive step after 0.144s.

<Success> Property ((speed > MAX_SPEED) => (asst_lvl <= 0.0)) is valid by inductive step after
    0.144s.

<Success> Property ((asst_lvl < 0.0) => (brk > 0.0)) is valid by inductive step after 0.144s.
```

```
<Success> Property ((alarm = true) => (asst_lvl >= 0.0)) is valid by inductive step after 0.144s.

<Success> Property ConnectionAlarm[l135c11].((false -> ((pre alarm) = true)) => (alarm = true)) is
    valid by inductive step after 0.144s.

<Success> Property ConnectionAlarm[l135c11].((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) =>
    (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property ConnectionAlarm[l135c11].((LongDropCounter(up) >= LONG_DROP_NUM) => (alarm =
    true)) is valid by inductive step after 0.144s.

<Success> Property LossyChannel[l134c14].((not up) => (brk_rcvd = noise)) is valid by inductive
    step after 0.144s.

<Success> Property LossyChannel[l134c14].(up => (brk_rcvd = brk)) is valid by inductive step after
    0.144s.

<Success> Property BrakeCableActuator[l138c18].ConnectionAlarm[l78c18].((LongDropCounter(up) >=
    LONG_DROP_NUM) => (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].ConnectionAlarm[l78c18].((ShortDropCounter(up) >=
    MAX_NOF_SHORT_DROPS) => (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].ConnectionAlarm[l78c18].((false -> ((pre alarm) =
    true)) => (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].((clinch = true) => (brk <> 0.0)) is valid by
    inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].((((brk > 0.0) and (brk <= BRK_SOFT)) and (not
    ConnectionAlarm(up))) =>
  (clinch = false)) is valid by inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].((((brk > 0.0) and (brk <= BRK_SOFT)) and
    ConnectionAlarm(up)) =>
  (clinch = true)) is valid by inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].(((brk > BRK_SOFT) and ConnectionAlarm(up)) =>
    (clinch = true)) is valid by inductive step after 0.144s.

<Success> Property BrakeCableActuator[l138c18].(((brk > BRK_SOFT) and (not ConnectionAlarm(up)))
    => (clinch = true)) is valid by inductive step after 0.144s.

<Success> Property MotorController[l141c17].ConnectionAlarm[l104c24].((LongDropCounter(up) >=
    LONG_DROP_NUM) => (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property MotorController[l141c17].ConnectionAlarm[l104c24].((ShortDropCounter(up) >=
    MAX_NOF_SHORT_DROPS) => (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property MotorController[l141c17].ConnectionAlarm[l104c24].((false -> ((pre alarm) =
    true)) => (alarm = true)) is valid by inductive step after 0.144s.

<Success> Property MotorController[l141c17].((not up) => (regen_allowed = false)) is valid by
    inductive step after 0.144s.

<Success> Property MotorController[l141c17].((speed > MAX_SPEED) => (assist_allowed = false)) is
    valid by inductive step after 0.144s.
```

```
<Success> Property MotorController[l141c17].((battery = 0.0) => (assist_allowed = false)) is valid
    by inductive step after 0.144s.

<Success> Property MotorController[l141c17].(((regen_allowed = true) and (assist_allowed = false))
    =>
  (asst_lvl = (- brk_rcvd))) is valid by inductive step after 0.144s.

<Success> Property MotorController[l141c17].(((regen_allowed = false) and (assist_allowed = true))
    => (asst_lvl = human)) is valid by inductive step after 0.144s.

<Success> Property MotorController[l141c17].((((brk_rcvd = 0.0) and (assist_allowed = true)) and
     (regen_allowed = true)) =>
  (asst_lvl = human)) is valid by inductive step after 0.144s.

<Success> Property ((((brk > 0.0) and (alarm = false)) and (up = true)) => (asst_lvl < 0.0)) is
    valid by property directed reachability after 4.250s.

<Success> Property (((alarm = true) and (brk > 0.0)) => (clinch = true)) is valid by property
    directed reachability after 4.250s.
```

```
Summary of properties:

((((brk > 0.0) and (alarm = false)) and (up = true)) => (asst_lvl < 0.0)): valid (at 1)
(((asst_lvl = 0.0) or (asst_lvl = (- brk))) or (asst_lvl = human)): valid (at 1)
((speed > MAX_SPEED) => (asst_lvl <= 0.0)): valid (at 1)
((asst_lvl < 0.0) => (brk > 0.0)): valid (at 1)
((alarm = true) => (asst_lvl >= 0.0)): valid (at 1)
(((alarm = true) and (brk > 0.0)) => (clinch = true)): valid (at 1)
ConnectionAlarm[l135c11].((false -> ((pre alarm) = true)) => (alarm = true)): valid (at 1)
ConnectionAlarm[l135c11].((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS) => (alarm = true)): valid
    (at 1)
ConnectionAlarm[l135c11].((LongDropCounter(up) >= LONG_DROP_NUM) => (alarm = true)): valid (at 1)
LossyChannel[l134c14].((not up) => (brk_rcvd = noise)): valid (at 1)
LossyChannel[l134c14].(up => (brk_rcvd = brk)): valid (at 1)
BrakeCableActuator[l138c18].ConnectionAlarm[l78c18].((LongDropCounter(up) >= LONG_DROP_NUM) =>
    (alarm = true)): valid (at 1)
BrakeCableActuator[l138c18].ConnectionAlarm[l78c18].((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS)
    => (alarm = true)): valid (at 1)
BrakeCableActuator[l138c18].ConnectionAlarm[l78c18].((false -> ((pre alarm) = true)) => (alarm =
    true)): valid (at 1)
BrakeCableActuator[l138c18].((clinch = true) => (brk <> 0.0)): valid (at 1)
BrakeCableActuator[l138c18].((((brk > 0.0) and (brk <= BRK_SOFT)) and (not ConnectionAlarm(up))) =>
  (clinch = false)): valid (at 1)
BrakeCableActuator[l138c18].((((brk > 0.0) and (brk <= BRK_SOFT)) and ConnectionAlarm(up)) =>
  (clinch = true)): valid (at 1)
BrakeCableActuator[l138c18].(((brk > BRK_SOFT) and ConnectionAlarm(up)) => (clinch = true)): valid
    (at 1)
BrakeCableActuator[l138c18].(((brk > BRK_SOFT) and (not ConnectionAlarm(up))) => (clinch = true)):
    valid (at 1)
MotorController[l141c17].ConnectionAlarm[l104c24].((LongDropCounter(up) >= LONG_DROP_NUM) =>
    (alarm = true)): valid (at 1)
MotorController[l141c17].ConnectionAlarm[l104c24].((ShortDropCounter(up) >= MAX_NOF_SHORT_DROPS)
    => (alarm = true)): valid (at 1)
MotorController[l141c17].ConnectionAlarm[l104c24].((false -> ((pre alarm) = true)) => (alarm =
    true)): valid (at 1)
MotorController[l141c17].((not up) => (regen_allowed = false)): valid (at 1)
MotorController[l141c17].((speed > MAX_SPEED) => (assist_allowed = false)): valid (at 1)
MotorController[l141c17].((battery = 0.0) => (assist_allowed = false)): valid (at 1)
```

```
MotorController[l141c17].(((regen_allowed = true) and (assist_allowed = false)) =>
  (asst_lvl = (- brk_rcvd))): valid (at 1)
MotorController[l141c17].(((regen_allowed = false) and (assist_allowed = true)) => (asst_lvl =
    human)): valid (at 1)
MotorController[l141c17].((((brk_rcvd = 0.0) and (assist_allowed = true)) and (regen_allowed =
    true)) =>
  (asst_lvl = human)): valid (at 1)
```

Chapter 2

Strix

2.1 Strix Introduction

This part of the project focuses on Strix, which is "a tool for reactive LTL synthesis combining a direct translation of LTL formulas into deterministic parity automata (DPA) and an efficient, multi-threaded explicit state solver for parity games".[1]

2.1.1 Strix and LTL

In contrast with the LTL formulas studied in the Formal Verification course, instead of using symbolic operators such as the diamond for Finally or the square for Globally, Strix relies on a more textual version of LTL operators.

Below is a small list of text symbols used in this rapport and their LTL name[2]:

- G ϕ: Globally or Always
- F ϕ: Finally or Eventually
- X ϕ: Next
- ψ U ϕ: Until
- ψ W ϕ: Weak

Useful resource giving the accepted LTL operators: `https://gitlab.lrz.de/i7/owl/-/blob/main/doc/FORMATS.md`

2.1.2 HOA format

The Hanoi Omega-Automata (HOA) format describes and allows the exchange of ω-automata. Details about HOA can be found on `https://adl.github.io/hoaf/`.

2.2 Foreword

This report will mostly include are the images generated by the Strix Demo website. The HOA format outputs are saved as text files in the `Task/` folder alongside the requested `.json` files.

[1]Source: `https://strix.model.in.tum.de/`
[2]Source: `https://en.wikipedia.org/wiki/Linear_temporal_logic`

2.3 Task A1: Consistency Properties

In this task, we must ”guarantee that motor is in one and only one state at all time instances”.

We start by defining the input and output atomic propositions that a priori will not change through this project.

The inputs will only contain the brake b,

```
"input_atomic_propositions": ["b"]
```

while the outputs will be made of the assistance a, recharge r and the idle i

```
"output_atomic_propositions": ["a", "r", "i"]
```

Note that the names have been shorten to provide better readability for further described guarantees and assumptions.

My first attempt to translate the above guarantee was to write the following:

```
G ( a & (!r) & (!i) ) OR G ( (!a) & r & (!i) ) OR G( (!a) & (!r) & i )
```

However, instead of making the states exclusive, it allows to create the motor only with either an assist, a recharge or an idle state. This is clearly not what we want.

Instead, we will write the LTL formula as follows:

```
G( ( a & (!r) & (!i) ) OR ( (!a) & r & (!i) ) OR ( (!a) & (!r) & i ))
```

Note that this time the Globally operator is outside and does <u>not</u> distribute.

Another viable way to express this guarantee would be to write :

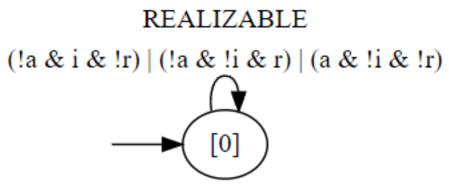
```
G(a -> (!r & !i)) AND G(r -> (!a & !i)) AND G(i -> (!r & !a))
```

However this does not really enforce the motor to have either ”a”, ”r” or ”i” positive.

Let’s execute all of the above in Strix:

```
./strix -f "true -> G( ( a & (!r) & (!i) ) OR ( (!a) & r & (!i) ) OR ( (!a) & (!r) & i ))" \
--ins="b" \
--outs="a,r,i"
REALIZABLE
HOA: v1
tool: "strix" "21.0.0"
States: 1
Start: 0
AP: 4 "b" "a" "r" "i"
controllable-AP: 1 2 3
acc-name: all
Acceptance: 0 t
--BODY--
State: 0 "[0]"
[(t) & (!(1 & (2 | 3) | !1 & (2 & 3 | !2 & !3)))] 0
--END--
```

First, it tells us that the specifications are realizable, and then it gives us the constructed machine in the HOA format. We obtain a one-state machine, with a list of 4 atomic proposition where ”a”, ”r”, and ”i” are controllable as expected. The acceptance `0 t` basically means that we have 0 accepting states with t (true) that specifies ”always accepting”. In other words, all recognized words are accepted. Next, we have a looping transition, guarded by `(!(1 & (2 | 3) | !1 & (2 & 3 | !2 & !3)))`, which converted to the respective APs, give `(!(a & (r | i) | !a & (r & i | !r & !i)))`. This respects the state exclusion as was expected. However, the purpose of "`AND (t)`" (true) in the condition, remains from my perspective unclear and superfluous.



Task A1 Mealy/Moore machine.

2.4 Task A2: Safety Properties

2.4.1 Part 1

The condition "controller must guarantee that the motor cannot jump instantaneously between states assist and recharge" must be described in this task.

This will be described using the Until LTL operator:

```
G (a -> a U i) AND G (r -> r U i)
```

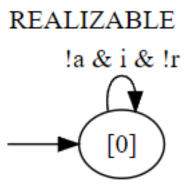
Which means that every time we encounter assist, the next state is not recharge and it hold until we reach idle (and respectively for recharge).

Another way to write the guarantee would be to use the Next operator if we wanted to explicitly oppose assist and recharge :

```
G (a -> X (!r U i)) AND G (r -> X (!a U i))
```

In other words, whenever we reach the assist state, assist must hold until we reach an idle state (i.e. we cannot get recharge in between), and respectively for the recharge state.

Strix outputs that this specification is realisable. This new machine simplifies the previous transition into `(!(1 | (2 | !3)))`, which in other words requires idle to be true and both assist and recharge to be false.



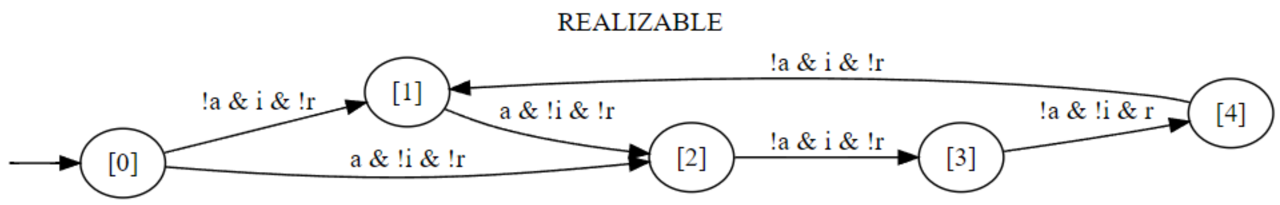
Task A1 Mealy/Moore machine.

2.4.2 Part 2

Now, we must add "a temporary guarantee that the motor must visit the states assist and recharge infinitely often".

This will be done by adding the "Always Eventually" guarantees:

```
G F (a) AND G F (r)
```



Task A1 Mealy/Moore machine.

We can observe a loop going through states 1, 2, 3, 4 and back to 1 that clearly ensures the "Always Eventually" property.

2.5 Task B1

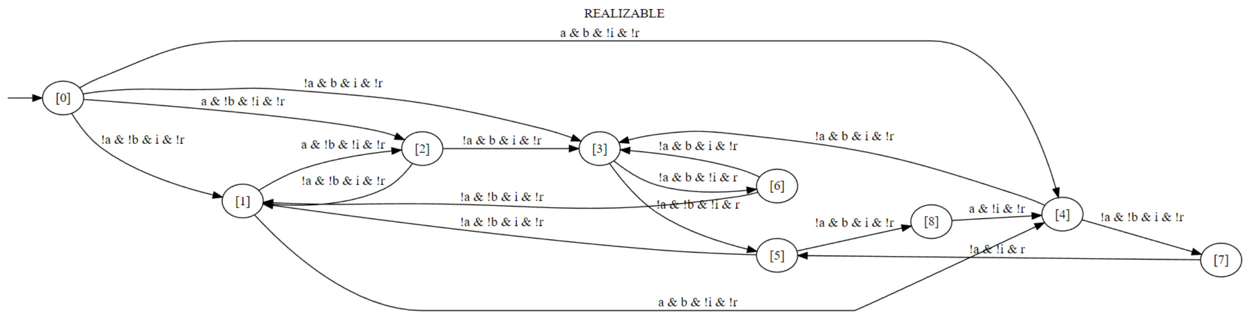
2.5.1 Part 1

We must add two new guarantees. First, "whenever the cyclist brakes, then the motor must, in the future, visit the state recharge." We model this by adding:

```
G (b -> F(r))
```

Next, "whenever the cyclist stops braking, then the motor must, in the future, visit the state assist". This is modelled in a similar way by:

```
G ((!b) -> F(a))
```



Task B1.1 Mealy/Moore machine.

2.5.2 Part 2

Model behaviour

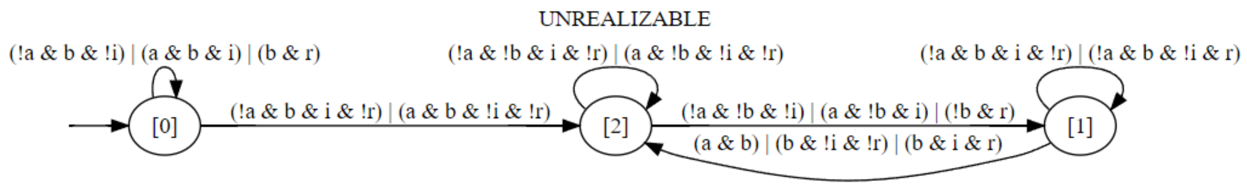
Actually, I am overall satisfied with the behaviour of the model. It respects the demanded guarantees, moreover it is realisable.

Brake and assistance

We must add a "temporary guarantee that whenever the cyclist brakes, the motor must stop assisting in the next time-step".

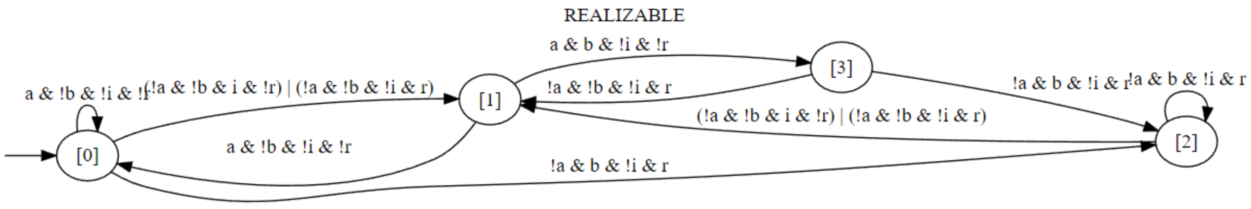
Which is simply modeled with:

```
G (b -> X !a)
```


UNREALIZABLE
(!a & b & !i) | (a & b & i) | (b & r)
[0]
(!a & b & i & !r) | (a & b & !i & !r)
[2]
(!a & !b & i & !r) | (a & !b & !i & !r)
(!a & !b & !i) | (a & !b & i) | (!b & r)
(a & b) | (b & !i & !r) | (b & i & r)
[1]
(!a & b & i & !r) | (!a & b & !i & r)

Task B1.2A Minimized Mealy/Moore machine.

Guarantee removal



Task B1.2B Mealy/Moore machine.

2.6 Task B2

2.6.1 Part 1

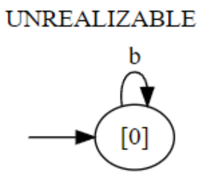
”You must guarantee that once the braking starts and the motor is in the state recharge, the motor must continue to remain in state recharge until the cyclist stops braking”

```
G ( (b & r) -> ( r U (!b) ) )
```

”Similarly, you must guarantee that when the braking is not activated and the motor is in the state assist, the motor must continue to remain in state assist until the cyclist starts braking”

```
G ( (!b & a) -> ( a U (b) ) )
```

I assumed that if the motor is not in the recharge state at the exact moment the cyclist starts braking, then ”`r U (!b)`” does not need to hold (and similarly for the assist condition).



Task B2.1 Minimized Mealy/Moore machine.

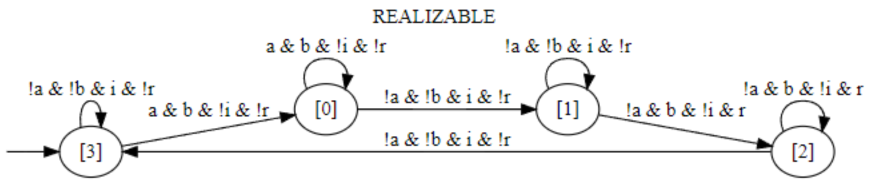
2.6.2 Part 2

We add the following temporary assumption:

```
(G F b) -> (G F NOT b) AND (G F NOT b) -> (G F b)
```

Another way to express this would be to simply write:

```
(G F b) AND (G F NOT b)
```



Task B2.2 Minimized Mealy/Moore machine.

2.6.3 Part 3

We were supposed to replace our Until operators by Weak-Until operators. Thus, we replace our current guarantees:

```
G( ( a & (!r) & (!i) ) OR ( (!a) & r & (!i) ) OR ( (!a) & (!r) & i ))
G (a -> a U i) AND G (r -> r U i)
G (b -> F(r)) AND G ((!b) -> F(a))
G ( (b & r) -> ( r U (!b) ) ) AND G ( (!b & a) -> ( a U (b) ) )
```

by the following:

```
G( ( a & (!r) & (!i) ) OR ( (!a) & r & (!i) ) OR ( (!a) & (!r) & i ))
G (a -> a W i) AND G (r -> r W i)
G (b -> F(r)) AND G ((!b) -> F(a))
G ( (b & r) -> ( r W (!b) ) ) AND G ( (!b & a) -> ( a W (b) ) )
```

As reminder, the Weak-Until operator does not require the right-hand side of the operation to be true in the future like Until does.

Since before adding the temporal assumption of Task B2 Part 2 we did not enforce that going from a brake we required the cyclist to release the brake at some point, the Until guarantee could not hold. For instance, the unrealizable machine obtained in Task B2 Part 1 clearly indicates that all the cyclist has to do is cycle forever. However, as the following guarantee

```
G ( (b & r) -> ( r U (!b) ) )
```

requires the not braking to hold at some point (once again, Until operator definition), it was one of the reasons the specifications were unrealizable.

2.7 Task B3

2.7.1 Part 1

In this task we must ”make assumption that the cyclist cannot switch between braking and not-braking within 3 time-steps”.

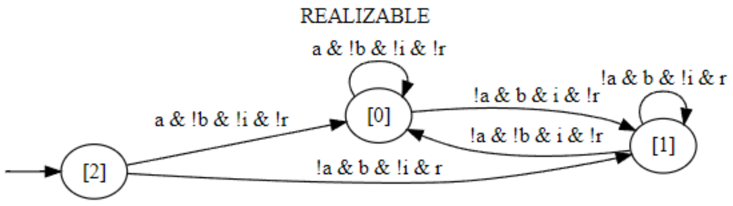
My first idea was to implement the following:

```
G ( (b & X (!b)) -> (X X (!b) & X X X (!b)) )
G ( ((!b) & X b) -> (X X b & X X X b) )
```

In other words, whenever we observe a change in braking `(b & X (!b))`, the change must preserve for two more time units. However, this is not the expected solution, as it does not ensure the condition at the start.

Instead, we will kind-of hard-code our requirement as:

```
G NOT (!b & (X b) & (X X b) & (X X X !b))
G NOT (!b & (X b) & (X X !b))
G NOT (b & (X !b) & (X X !b) & (X X X b))
G NOT (b & (X !b) & (X X b))
```



Task B3.1 Minimized Mealy/Moore machine.

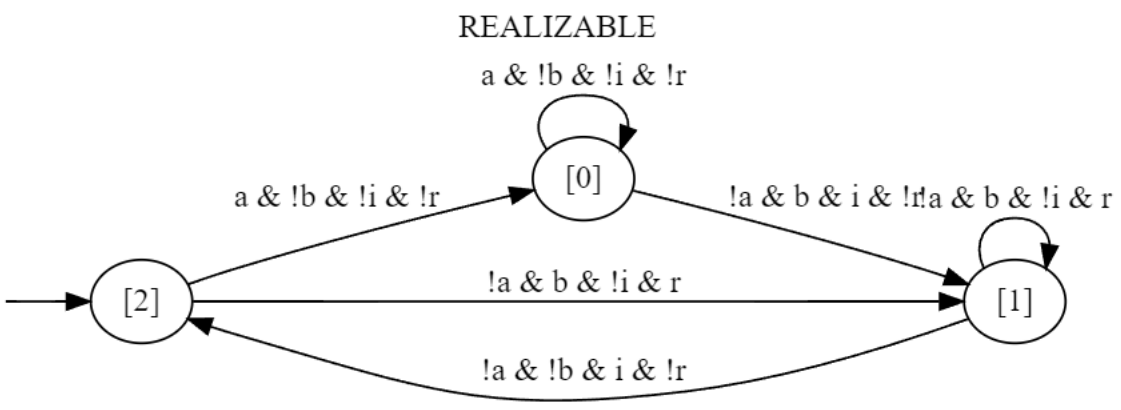
3 time-steps Please note that I was not sure if I had to interpret 3 times-steps as at least three brakes between two releases or rather that the change should happen at least at the 3rd time (at least 2 brakes). I have implemented the latest, and moreover, it is realisable. If we wanted to add one more (or even more) brake (or release) in between, we could just add the following guarantee:

```
G NOT (!b & (X b) & (X X b) & (X X X b) & (X X X X !b))
G NOT (b & (X !b) & (X X !b) & (X X X !b) & (X X X X b))
```

2.7.2 Part 2

We are required to "find out the minimum number of time steps" of Task B3 Part 1. Thus, we simply remove one level of time-stamp and see if it is still realisable.

```
G NOT (!b & (X b) & (X X !b))
G NOT (b & (X !b) & (X X b))
```



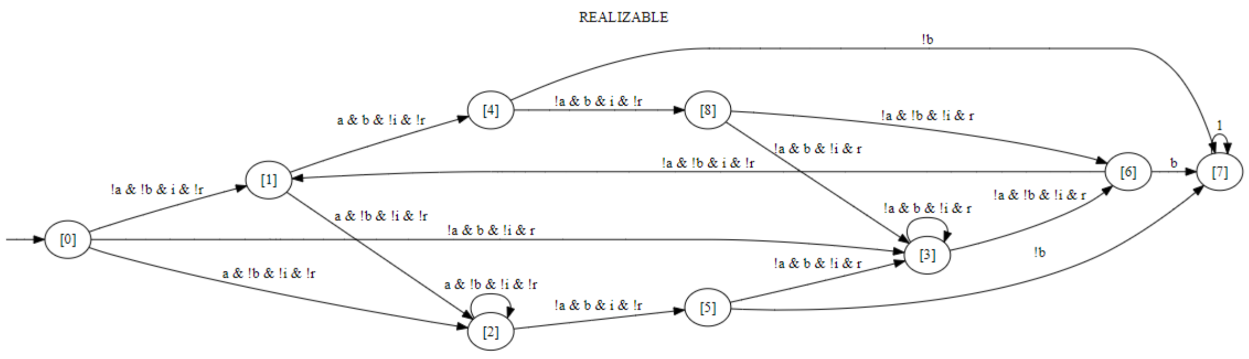
Task B3.2 Minimized Mealy/Moore machine.

2.8 Task B4

"Can you guarantee that whenever the cyclist brakes, the motor must reach the state recharge within 4 timesteps? How about 3 time-steps?".

```
G (b -> (X r | X X r | X X X r | X X X X r))  (4 time-steps)
G (b -> (X r | X X r | X X X r))              (3 time-steps)
G (b -> (X r | X X r))                        (2 time-steps)
G (b -> (X r))                                (1 time-step)
```

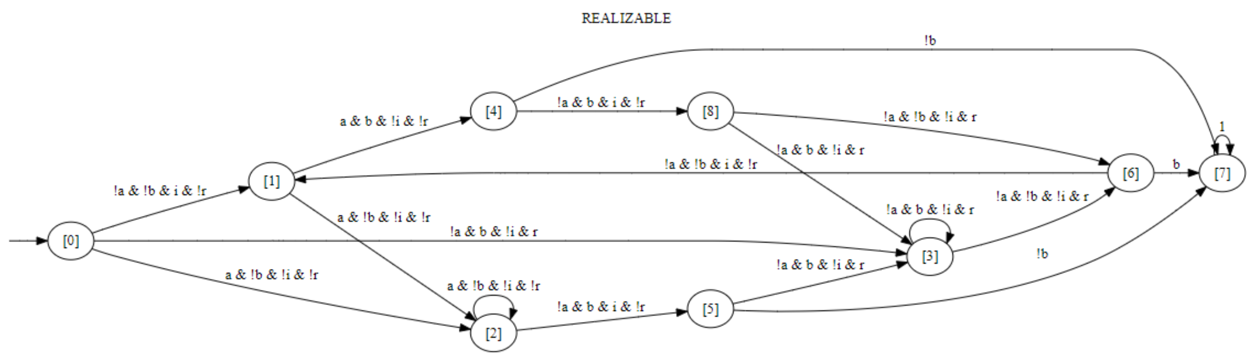
We obtain that the model is REALISABLE with 4 time-steps. It only becomes UNREALISABLE once we reach 1 time-step.



Task B4 Recharge Only (2ts).

Similarly, to the recharge, we add the assist guarantees (that also reach 2 time-steps):

```
G (!b -> (X a | X X a | X X X a | X X X X a))  (4 time-steps)
G (!b -> (X a | X X a | X X X a))              (3 time-steps)
G (!b -> (X a | X X a))                        (2 time-steps)
G (!b -> (X a))                                (1 time-step)
```



Task B4 Assist Only (2ts).

Both recharge and assist guarantees combined are given in the next section.

2.9 Final machine

Input AP:

```
b (brake)
```

Output APs

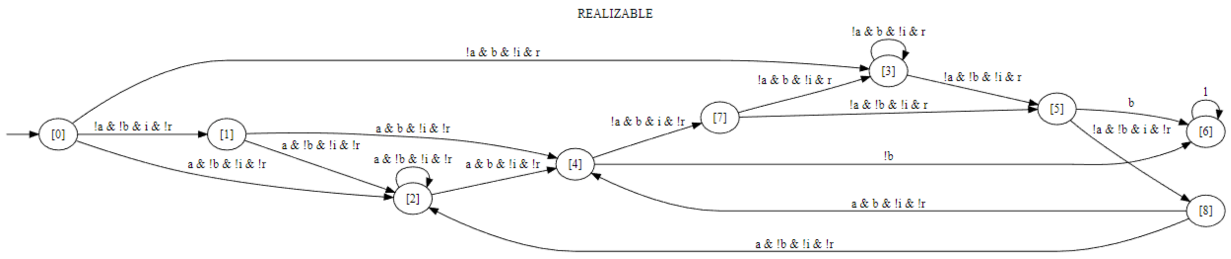
```
a (assist)
r (recharge)
i (idle)
```

Assumptions:

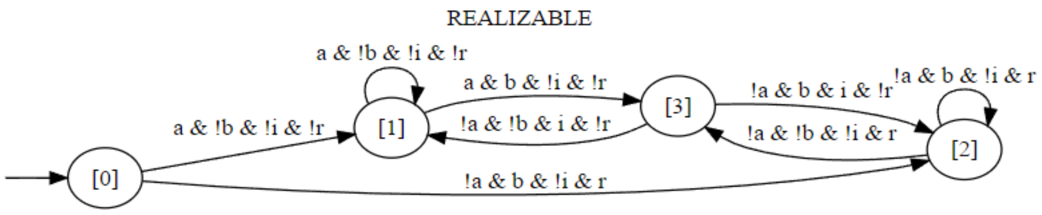
```
G NOT (!b & (X b) & (X X !b))
G NOT (b & (X !b) & (X X b))
```

Guarantees:

```
G( ( a & (!r) & (!i) ) OR ( (!a) & r & (!i) ) OR ( (!a) & (!r) & i ))
G (a -> a W i) AND G (r -> r W i)
G (b -> F(r)) AND G ((!b) -> F(a))
G ( (b & r) -> ( r W (!b) ) ) AND G ( (!b & a) -> ( a W (b) ) )
G (b -> (X r | X X r)) AND G (!b -> (X a | X X a))
```



Task B4 Recharge+Assist (2ts).



Task B4 Recharge+Assist <u>Minimized</u> (2ts).

Chapter 3

UppAal TiGa

3.1 UppAal Tiga Introduction

This part of the project focuses on Uppaal and its extension Tiga. Uppaal is "an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.)".[1] TIGA is an extension of UPPAAL "for solving games based on timed game automata with respect to reachability and safety properties".[2]

3.2 Foreword

Tasks from U1 to U6 are implemented in the uppaal_project .xml and .q files, whereas U7 is available in uppaal_project2 .xml and .q as demanded. Note that to separate the Task U7 from the rest, Tasks U8 and U9 are stored in uppaal_project3 .xml and .q.

3.3 Task U1

This task requires us to two things. First, we modify the directed edge from the state Assist to Choose by adding the `timer1` equals 0 update property. This will put the edge to 0 each time this path is traversed.

Next, we need to actually define what timer1 is. Thus, we go to `Declarations`, and we add a line for *clock timer1.*

The clock variable, as its name suggest, serves to mesure the time progress. Clocks are usually useful to synchronise different system components

3.4 Task U2

In order to add the Boolean variable, we simply go to `Declarations` and add the int [0,1] brake variable, which is simply an integer that can take values 0 and 1.

Then, we insert a new and fresh template to model the braking system. We add three states: Pressed, Choose and Released, where the latest will have the `Initial` property checked. Additionally, we will add the Commited property to the Choose state, in order to force a transition (we do not want the braking system to remain in the Choose state for long).

Then, we add directed edges between all states except between Released and Pressed, as they will have to pass through the Choose state first.

We add the brake = 0/1 update from the Choose state respectively whether it goes to the Released or the Pressed state.

And finally we uncheck the Controllable property from all the edges.

[1]Source: https://uppaal.org/
[2]Source: https://uppaal.org/features/#tiga

3.5 Task U3

As before, we add a Boolean variable named batteryLevel, and we create a new template for the Battery system with the High (init), Low and Choose states.

As the battery can only discharge/charge when the motor is assisting/recharging, there will be two types of edges going from Choose to High/Low.

The first type allows to change the battery level (i.e go from High/Low to Low/High). This transition will update the `batteryLevel` accordingly if the corresponding `motorState` guards are respected.

The second type will allow to stay on the same battery level. To achieve this, additional edges were added from Chose to High/Low with guards that only check if the `batteryLevel` variable equals 1/0. That way, we will be able to do transitions such as High - Choose - High or Low - Choose - Low.

In case the `motorState` equals 0 (i.e neither assisting or recharging), system will only be able to transit using that second type of transition.

3.6 Task U4

We are supposed to execute a loop over the 4 states every 1 unit of time, that will allow to update the Battery, the Brake and the Motor in that exact order.

For this purpose, we start by creating for states: BatteryUpdate, BrakeUpdate, MotorUpdate and Wait. They will be connected by edges such that they for a loop: Battery to Brake to Motor to Wait and once again to Battery.

The Wait to Battery transition will be guarded with a timer called *schedule_timer* and it will check if it equals exactly 1, plus it will update the timer to 0 once the transition is taken. Additionally, the Wait state will have the *schedule_timer* $\leq$ *1* invariant so the Scheduler will not be able to remain the Wait state forever. That way, we will loop every one unit of time.

All the other transitions will have the Sync property, that thanks to Channels, will allow to send signals to other automata so force them to execute. What concerns the Battery, Brake and Motor update states, the will be checked as *urgent* so they are forced to transit one to another. Those will make the time freeze until we reach the Wait state, which will allow us to ensure we loop every one whole time unit. Note that those states are **not** marked as *committed*, in which case the Scheduler would first reach the Wait state of Scheduler before executing the *committed* states of the other automata.

3.7 Task U5

The observer is implemented using a one-state automaton. Every time the Brake systems chooses to go to the Pressed state, it emits a braking! broadcast, that tells the observer to reset the timer to 0. Since we do not want the timer to reset every time the Choose $\rightarrow$ Pressed, a guard condition was added to the Observer allowing the traversal if only the brake is set to 0. This means it will reset only if the previous state was Released. It is important to note that we use a broadcast channel and not a simple channel, in which case the Brake system would not be able to go to the Pressed state more than once in row (synchronisation).

3.8 Task U6

This task requires us to write a certain number of conditions that will have to be respected. The corresponding query formula has been written as follows:

```
control: A[] (Scheduler.Wait) imply
    (((Battery.Low and (timer2 >= 5 and Brake.Pressed)) imply not Motor.Idle) and (3.b)
    ((Brake.Released and Battery.High) imply not Motor.Idle) and               (3.a)
```

```
       ((timer1 <= 2) imply not Motor.Recharge) and                        (2.c)
       (Battery.High imply not Motor.Recharge) and                         (2.b)
       (Brake.Released imply not Motor.Recharge) and                       (2.a)
       (Battery.Low imply not Motor.Assist) and                            (1.b)
       (Brake.Pressed imply not Motor.Assist))                             (1.a)
```

Along side each line of the query have been added the corresponding conditions requested in project's instructions. The `control` prefix allows to execute the query in Uppaal Tiga instead of the usual Uppaal[3], and thus look for a winning strategy instead of verifying that those conditions always hold.

3.9 Task U7

3.9.1 Queries

In order to verify that Conditions 1, 2 and 3 hold, we will test a slightly different query from the one given in Task U6:

```
   A[] (Scheduler.Wait and scheduler_timer != 1) imply
       (((Battery.Low and (timer2 >= 5 and Brake.Pressed)) imply not Motor.Idle) and (3.b)
       ((Brake.Released and Battery.High) imply not Motor.Idle) and        (3.a)
       ((timer1 <= 2) imply not Motor.Recharge) and                        (2.c)
       (Battery.High imply not Motor.Recharge) and                         (2.b)
       (Brake.Released imply not Motor.Recharge) and                       (2.a)
       (Battery.Low imply not Motor.Assist) and                            (1.b)
       (Brake.Pressed imply not Motor.Assist))                             (1.a)
```

The major difference is that we have removed the `control` prefix, which will assure that those conditions will always hold, instead of finding a winning strategy like we did in task U6.

Important assumption Please note the `schedule_timer != 1` added to the `Scheduler.Wait` check. It comes from the very design of the Scheduler and the following property from **Task U4** it must respect:

```
   At state Wait it should stay for exactly 1 time unit and go back to the state BatteryUpdate.
```

In the Scheduler, the system leaves the state **Wait** to **BatteryUpdate** only when the `schedule_timer` timer reaches 1 unit. In consequence, there might exist a really brief instant at which the `schedule_timer` equals 4, we are both in the Scheduler.Wait and in Motor.Idle (which is allowed), and just before leaving the **Wait** state at exactly `schedule_timer = 1`, the condition (3.b) instantly fails as `timer2` equals now 5 and technically we are still in the `Wait` state.

For this reason, `schedule_timer != 1` was added to the query.

What concerns the fourth Condition, it can be easily checked in Uppaal with the following query, which will check if for all possible branches we never reach a deadlock:

```
   A[] not deadlock
```

3.9.2 Guards

In order to make sure the conditions are respected, we will simply put the negation of those conditions on the corresponding motor states. For example, the condition "The motor should not assist when the brake is pressed, or the battery is low", can be implemented by adding the guard:

```
brake == 0 and batteryLevel == 1
```

[3]Source: https://people.cs.aau.dk/~adavid/tiga/manual.pdf

on the edge going from Choose to Assist.

In the same way, we add the following guard for the 2nd Condition:

```
brake == 1 and batteryLevel == 0 and timer1 > 2
```

and the below one for the 3rd Condition:

```
(brake==1 or batteryLevel==0) and (batteryLevel == 1 or (timer2 < 5 or brake == 0))
```

Note: As Uppaal can be quite capricious with negations, the guards had to be slightly rewritten and could not be wrote using the simple form *not(Condition)*.

3.9.3 Results

Using the Uppaal's verifier tool, we obtain that the two queries/properties above are successfully satisfied after adding the motor guards.

3.10 Task U8

In this task, we have to check if "Always, if the brake has been pressed for more than 1 time unit, the motor engage recharge".

This can be checked using the following query:

```
(Brake.Pressed and timer2 > 1) --> (Motor.Recharge)
```

When Verified by Uppaal, it is stated as unsatisfied.

We can find two easy counter-examples, that are due to the previously defined conditions:

2.c As first counter-example, it can be blocked by the `timer1 > 2` guard. Since the Assist state can only be reached when the cyclist is not braking, we might get a trace during which timer1 of the motor and timer2 of might evolve at the same pace (i.e `timer1 == timer2`). In consequence, we might reach Motor.Choose, and despite reaching timer2 with brake == 2 (first part of the property), we might not be able to reach enter Motor.Recharge because timer2 also equals 2, and thus we are blocked by the Condition 2.c

2.b The second counter-example, is obtained thanks to the `batteryLevel == 0` guard. No matter for how long the cyclist has been braking, the system will not be able to enter Motor.Recharge if the battery is high.

As a reminder, the battery can go to Low only if the motor is assisting, but there exist a scenario in which the battery remains indefinitely High. Additionally, no matter how long we will be braking, if the battery is High, we will not be able to enter Motor.Recharge since we would need to enter Motor.Assist to lower the battery, which in turn would require to stop braking.

3.11 Task U9

3.11.1 Rim brakes

We start by inserting a new template we will call Rimbrake. It contains two states Released (initial) and Pressed. Next we define a binary channel as `chan rimUpdate[2]`. All transitions are done by synchronisation with the Motor automaton as described in the project's instructions. Whether Motor signals `rimUpdate[0]!` we either go from Pressed to Released or we stay on Released, and similarly for `rimUpdate[1]!` and the Pressed state.

To signal `rimUpdate[0]!`, we simply added the sync property on edges Choose → Assist and Choose → Recharge. However, we can **not** only add `rimUpdate[1]!` to Choose → Idle, as we do not want to signal anything if the automaton goes to Motor.Idle while it could have gone to the Motor.Recharge state; the signal shall only be sent if we the Motor.Idle state has been reached while Motor.Recharge could have not.

A first naive solution would be to copy the guard from Choose → Idle, and add another transition that respects that guard, but that additionally makes sure that:

```
and (brake == 1)
and (batteryLevel == 1 or timer1 <= 2)
```

This however, would not make the two transitions exclusive, and so the Motor could always take the initial transition without emitting the signal. Thus, we add the following to the initial Choose → Idle to make it exclusive with the new one:

```
and ((brake == 0) or (batteryLevel == 0 and timer1 > 2))
```

3.11.2 Property check

Finally, we have to check the following property: "Always, if the brake has been pressed for more than 1 time unit, the motor either engage recharge or the rim-brakes are pressed".

It can be tested using the following query:

```
(Brake.Pressed and timer2 > 1) --> (Motor.Recharge or Rimbrake.Pressed)
```

But as expected from the Rim brakes definition, the property is not satisfied. This is due to the fact that the transitions Choose → Idle and Choose → Recharge are not exclusive. For instance, with

- `brake == 1`
- `timer1/2 == 3`
- `batteryLevel == 0`

it is allowed to reach both Motor.Idle and Motor.Recharge from Motor.Choose. In consequence, the property is not satisfied. Additionally, since Motor.Choose → Motor.Idle transition can not be taken anymore if braking happens for more than 5 time units, one might think that the Motor will eventually reach Motor.Recharge. However, after reaching Motor.Idle, the cyclist can at any moment stop braking which will restart the whole process.

Note that if we took look at this property from a strategy game point of view, it would probably be satisfied. Since the edges of the Motor automaton are controllable, we would have the choice to go to the Recharge state instead of the Idle in order to win the game.

Chapter 4

XCos

4.1 Xcos introduction

This part of the project focuses on XCos, which is a Scilab package "for modeling and simulation of explicit and implicit dynamical systems, including both continuous and discrete sub-systems".[1]

4.2 Task X0

4.2.1 Lipschitz-continuity (a)

Definition of Lipschitz-continuity In the ... book, the Lipschitz continuity is intuitively defined as a constant upper bound that limits how fast a function changes. More precisely, a function $f : \mathbf{R}^n \to \mathbf{R}^n$ is said to be Lipschitz continuous if there exists a constant K such that for all vectors u and v in $\mathbf{R}^n$,

$$||f(u) - f(v)|| \leq K||u - v||$$

To better understand this condition, we can rewrite it as follows:

$$||\frac{f(u) - f(v)}{u - v}|| \leq K$$

This actually tells us, that the slope (more precisely the absolute value of it) must be smaller than a certain constant, which in other words means that the slope must not grow indefinitely.

Bike dynamics without all the multipliers The bike dynamics function without all of the multiplies looks as follows:

$$\frac{dv}{dt} = F_e + F_h - sin(\alpha) - cos(\alpha) - v^2$$

In order to check if it actually respects the Lipschitz-condition, we will check all of the five summands above.

(i) First, we have that both $-sin(\alpha)$ and $-cos(\alpha)$ respect the condition as sinusoidal functions are bounded between -1 and 1.

(ii) Next, we have F_e and F_h that are simply the inputs.

(iii) And finally, we have $-v^2$. By definition, as the slope of a negative quadratic function over $\mathbf{R}^n$ decreases infinitely, we could never find such a constant that could limit it. However, the given variable v describes the speed of a bicycle. Even as the interval of the possible speeds is not explicitly described in the project instructions (despite being positive), we can easily suppose that the bike speed will always be lower than the speed of light. In consequence, such a constant K can be found.

As all the summands respect the Lipschitz continuity condition, the above equation also respects the condition.

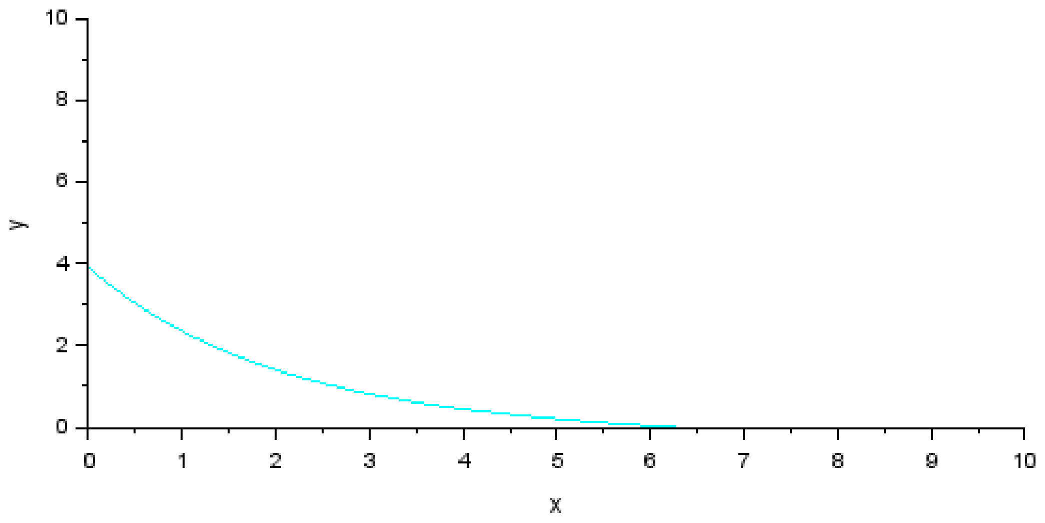
[1]Source:https://en.wikipedia.org/wiki/Scilab

4.2.2 Acceleration vs Speed (b)

In this task we are supposed to plot "derivative vs speed" on respectively the y and x axes, and then study the equilibrium of the speed variable.

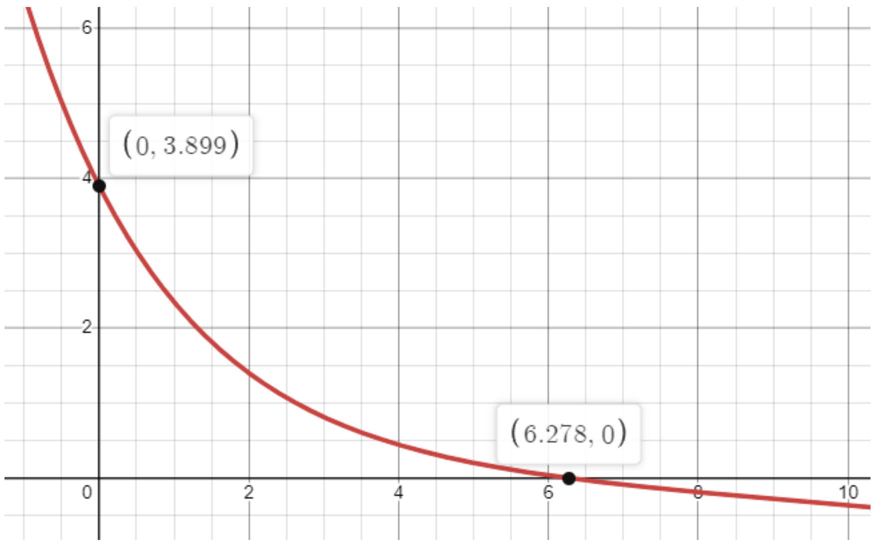
In order to obtain the inputs to our plot, the acceleration (i.e. the derivative) is simply obtained using the bike dynamics equation. The speed can be obtained by simply integrating the obtained acceleration.

This gives us the following graph:



Task X0b: Acceleration (Y) vs Speed (X)

However, as the produced plot is in my opinion hard to read, here is the equivalent plotted with the Desmos graphic calculator[2]:

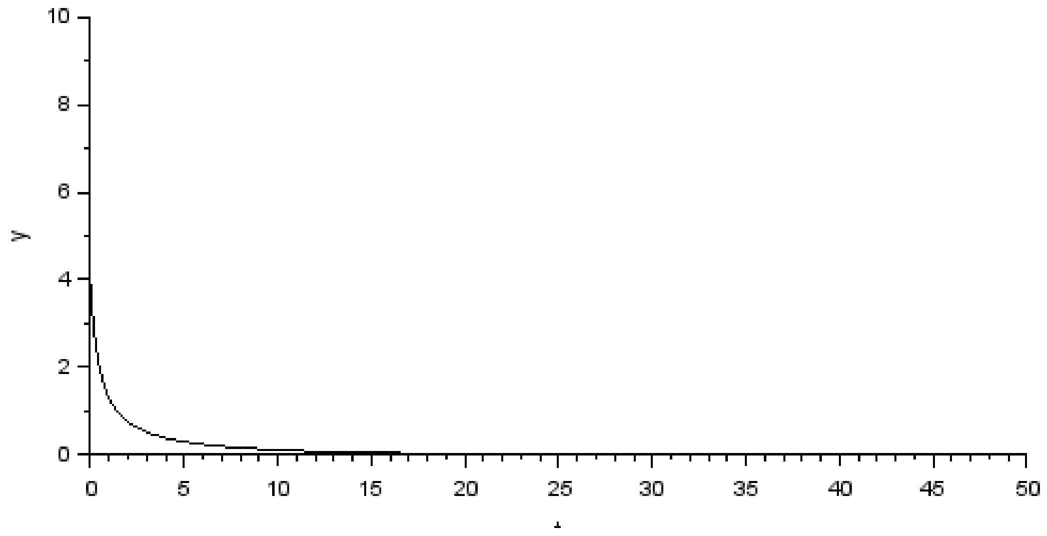


Task X0b: Acceleration (Y) vs Speed (X) (Desmos)

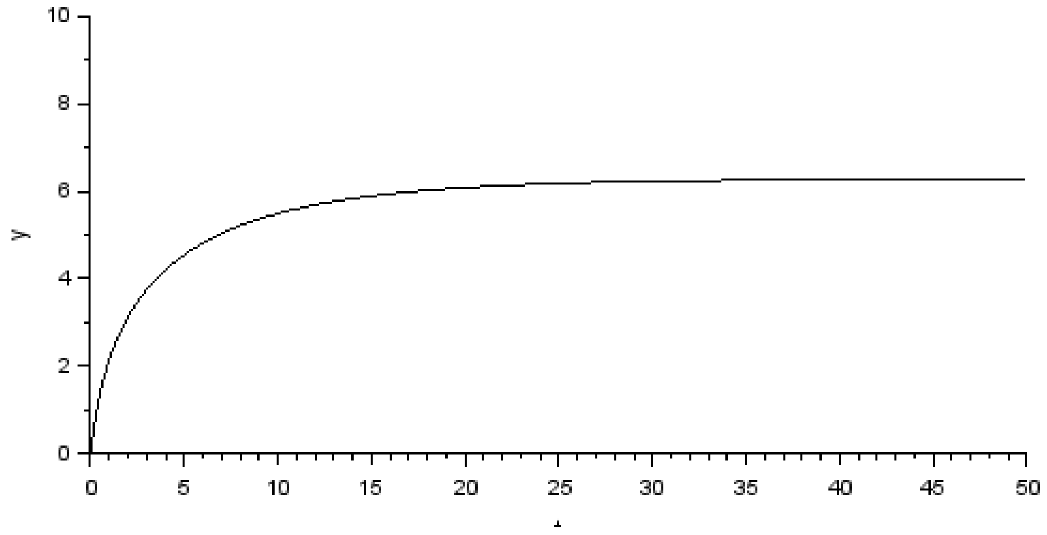
Which clearly indicates the limit value of the speed, that is of 6.278 (I suppose m/s).

[2]Image source: https://www.desmos.com/calculator

Additionally, the two plots below were added to better understand the behaviour of the acceleration and the speed over time:



Task X0b: Acceleration over time

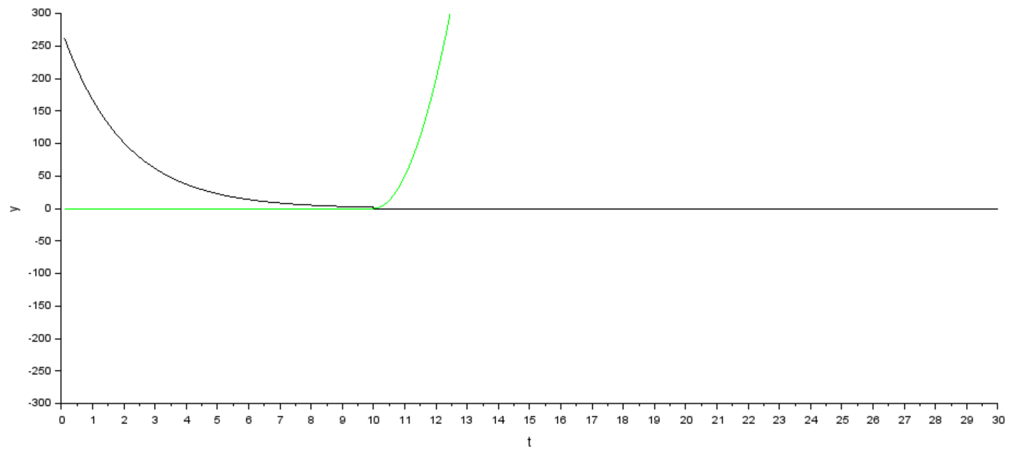


Task X0b: Speed over time

We can thus observe and confirm that as the acceleration decreases to 0, the speed tends to approximately 6.278. As the speed get closer and closer towards that value over time, we can conclude that that equilibrium is asymptotically stable.

4.3 Task X1

The below graph plots *effort vs speed*, where effort is represented on the Y axis by the human effort F_h colored in black and the braking colored in green. The speed has been modeled using the clock going from 0 to 30.

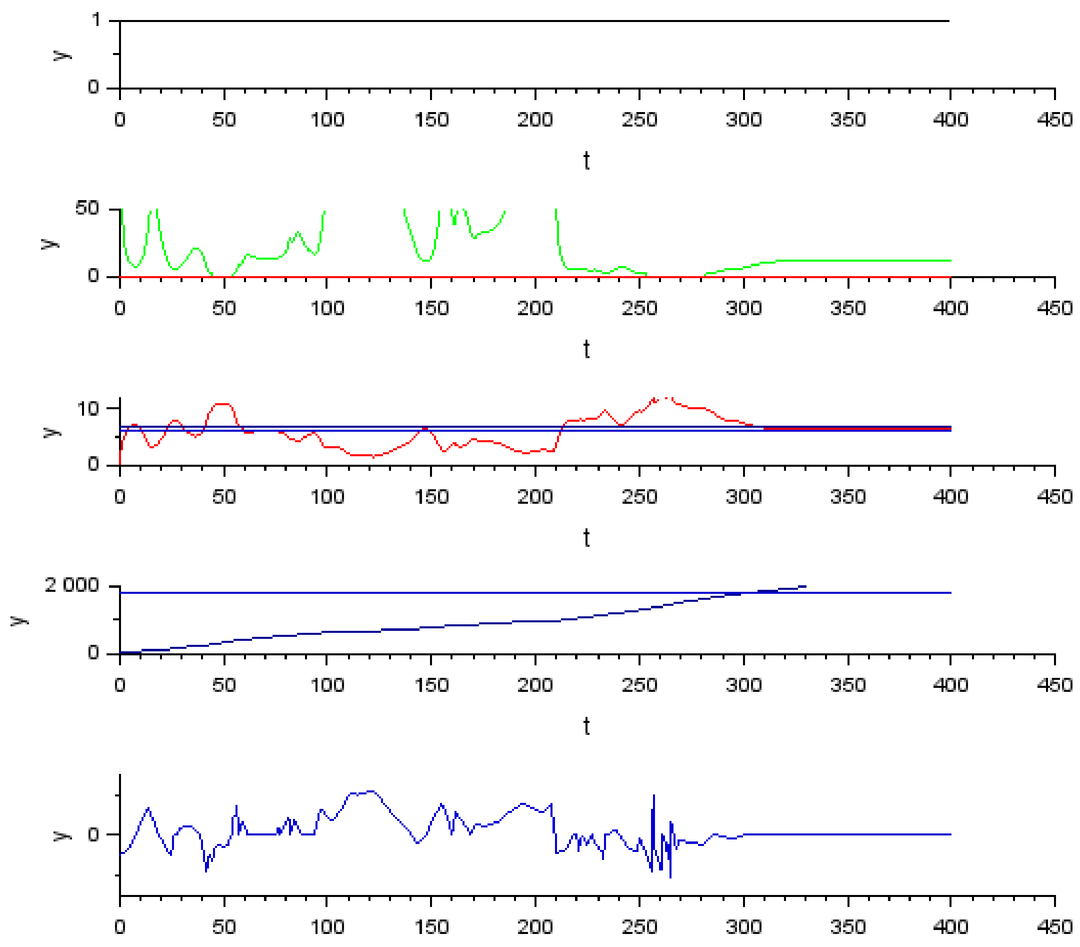


Task X1 : Black=F_h, Green=brake_request

As expected, we observe a drastic change of both functions at 10 m/s.

4.4 Task X2

This part implements the dynamics of the bike.

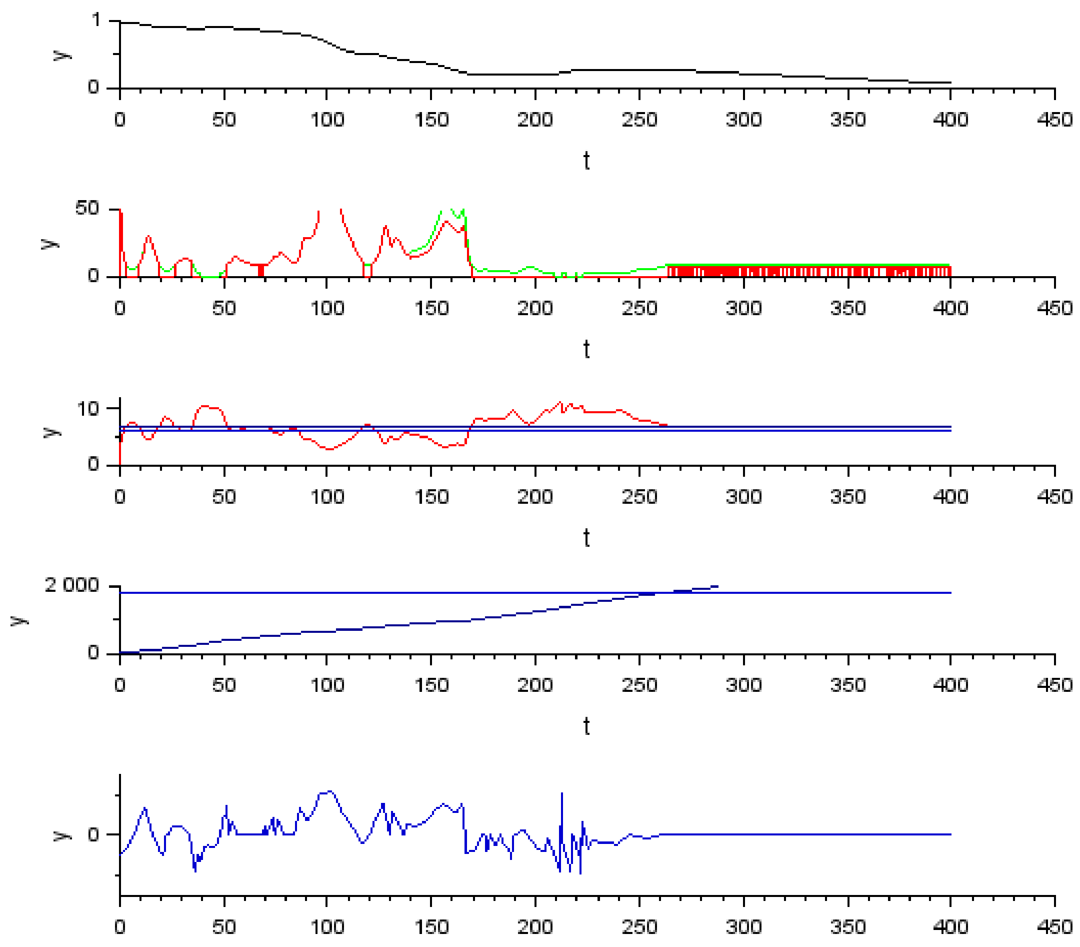


Task X2 : BL, F_h and F_e, speed, position, slope

The biker has a to travel a total distance of 1800m, as shown by the constant function in the fourth plot. This destination is reached after 300 seconds (plus some insignificant decimals), which gives an average speed of 6m/s.

4.5 Task X3

In this part we have added the basic model of electrical assistance. The simulation diagrams are given below.



Task X3 : BL, F_h and F_e, speed, position, slope

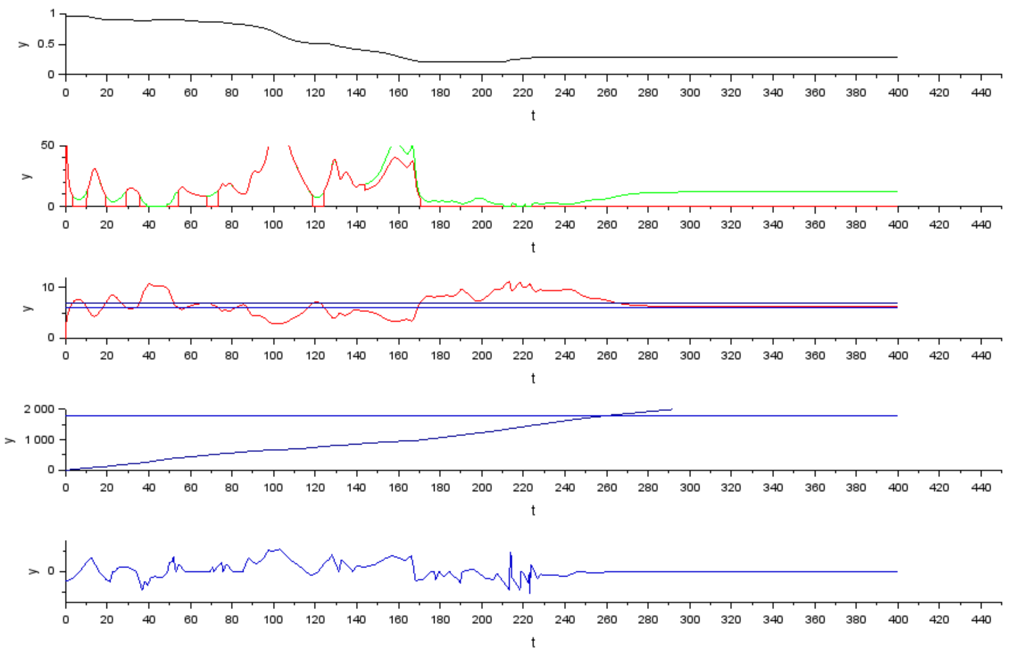
The destination is reached after 260s (around 259.97), which divided my the total distance gives the average of 6.92 m/s.

Additionally, we can observe a lot of red rectangles in the second plot after around t=260s, or even at around t=70. Those are due to the fact that the assistance is turned off on speeds above 7m/s, and that the speed is constantly fluctuating around that speed value, making the assistance frequently turning ON and OFF. This behaviour is clearly not ideal for a real-world model and it will be corrected by the advanced model implemented in the Tash X4.

4.6 Task X4

This task implements a slightly more advanced model of the electrical assistance implemented in task X3.

We have added a hysteresis behavior, which corrects the unwanted behaviour explained in the task X3.



Task X4 : BL, F_h and F_e, speed, position, slope

The travelled distance of 1800m is reached after 261S (around 261.297s), which gives an average speed of 6.89m/s.

This is slightly slower than the results obtained in in task X3: 1s total time difference and a drop of speed of around 0.03m/s. This is probably due to the fact that the assistance is off until we drop from 7m/s to 6m/s.