Université Libre de Bruxelles

INFO-F410 - Embedded Systems Design

Electric-Bike Project (Strix)

Student (MA1 Computer Science)
SWIRYDOWICZ Szymon
000477108

Lecturer / TAs
RASKIN Jean-François
BALACHANDER Mrudula
CHAKRABORTY Debraj
KHALIMOV Ayrat

April - May 2022

Contents

Chapter 1

Strix

1.1 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]

1.1.1 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

1.1.2 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/.

1.2 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

1.3 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 not 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))
```

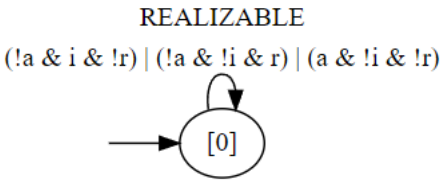
However this does not really enforce the motor to have either "a", "r" or "i" positive.

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.

1.4 Task A2: Safety Properties

1.4.1 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)
```

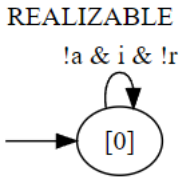
Which means that every time we encounter assist, the next state is not recharge and it hold until we reach idle (and respectively for recharge).

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.



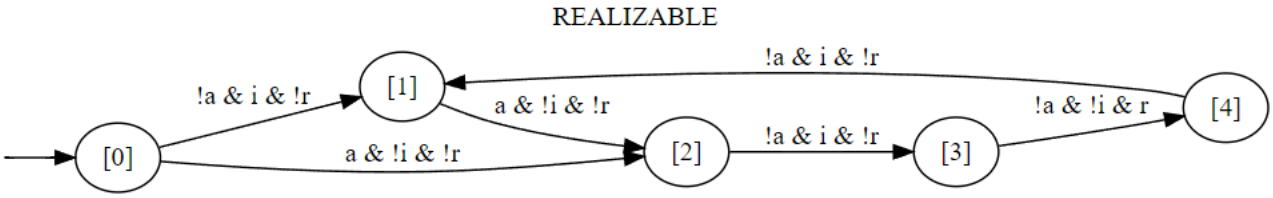
Task A1 Mealy/Moore machine.

1.4.2 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.

1.5 Task B1

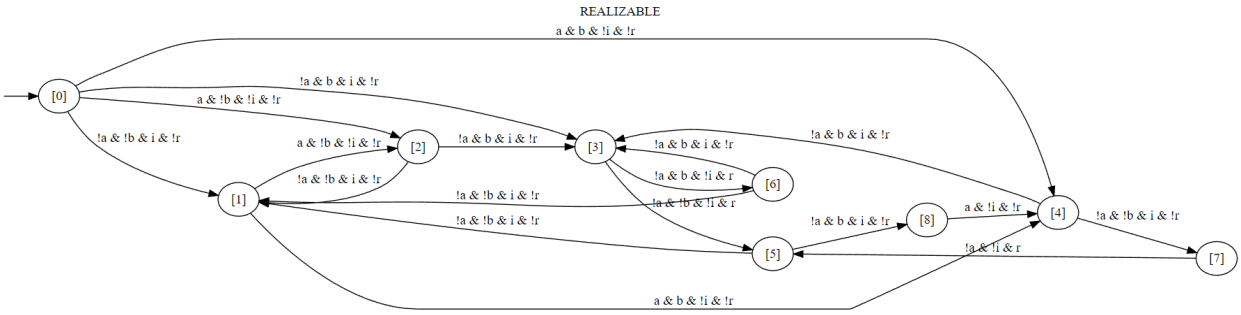
1.5.1 Part 1

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.

1.5.2 Part 2

Model behaviour

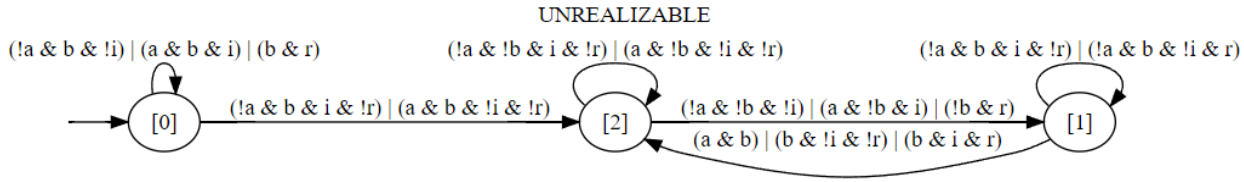
Actually, I am overall satisfied with the behaviour of the model. It respects the demanded guarantees, moreover it is realisable.

Brake and assistance

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

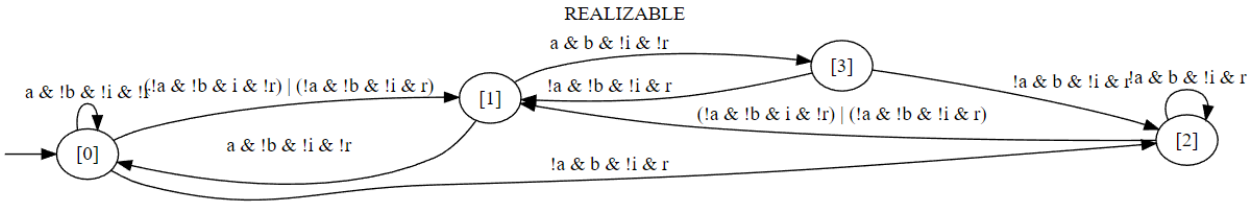
Which is simply modeled with:

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



Task B1.2A Minimized Mealy/Moore machine.

Guarantee removal



Task B1.2B Mealy/Moore machine.

1.6 Task B2

1.6.1 Part 1

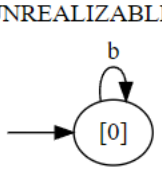
"You must guarantee that once the braking starts and the motor is in the state recharge, the motor must continue to remain in state recharge until the cyclist stops braking"

```
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.

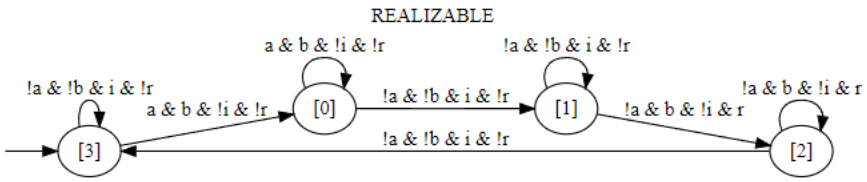
1.6.2 Part 2

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.

1.6.3 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.

1.7 Task B3

1.7.1 Part 1

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

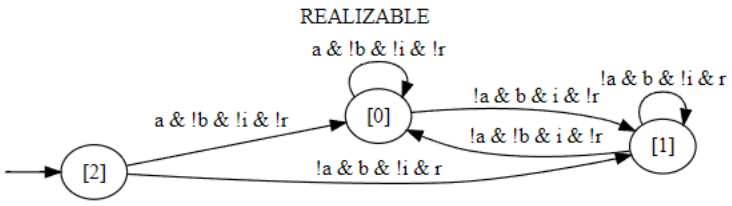
My first idea was to implement the following:

```
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.

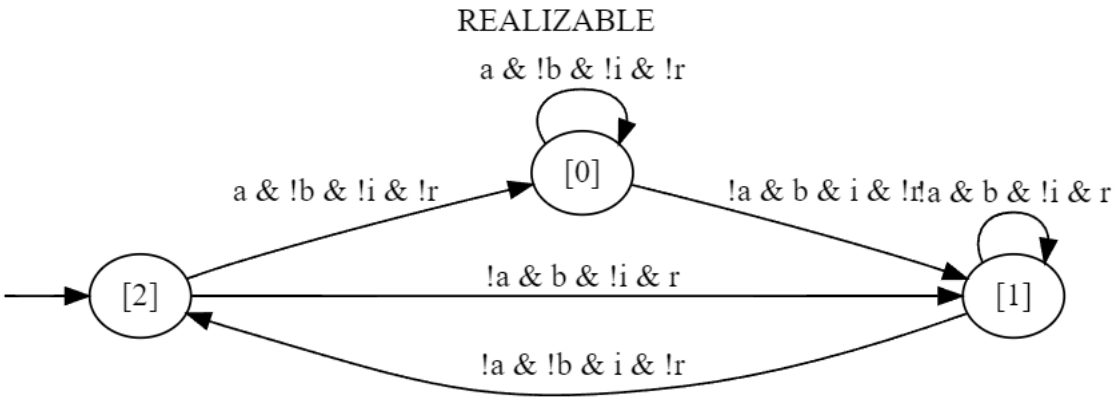
3 time-steps Please note that I was not sure if I had to interpret 3 times-steps as at least three brakes between two releases or rather that the change should happen at least at the 3rd time (at least 2 brakes). I have implemented the latest, and moreover, it is realisable. If we wanted to add one more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

1.7.2 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))
```



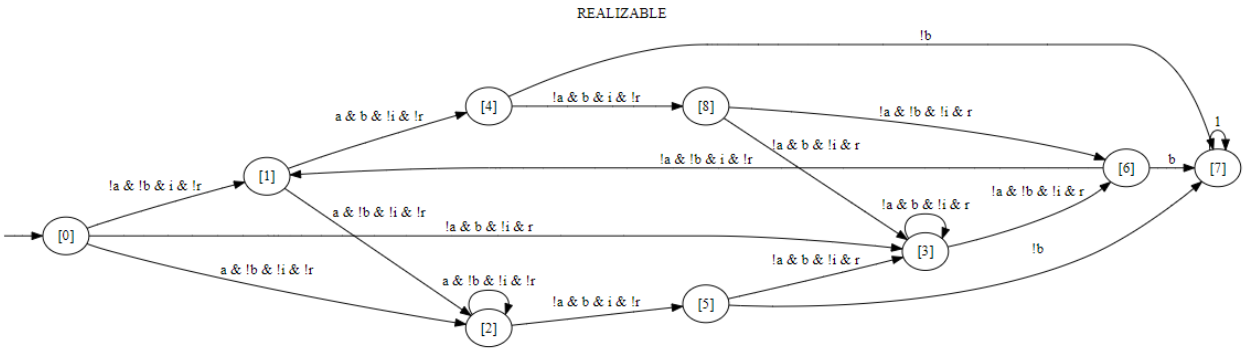
Task B3.2 Minimized Mealy/Moore machine.

1.8 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)
```

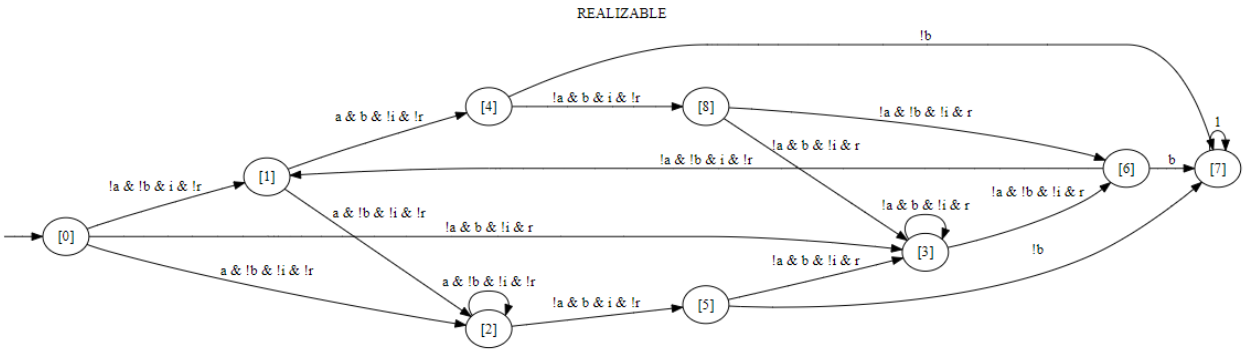
We obtain that the model is REALISABLE with 4 time-steps. It only becomes UNREALISABLE once we reach 1 time-step.



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.

1.9 Final machine

Input AP:

```
b (brake)
```

Output APs

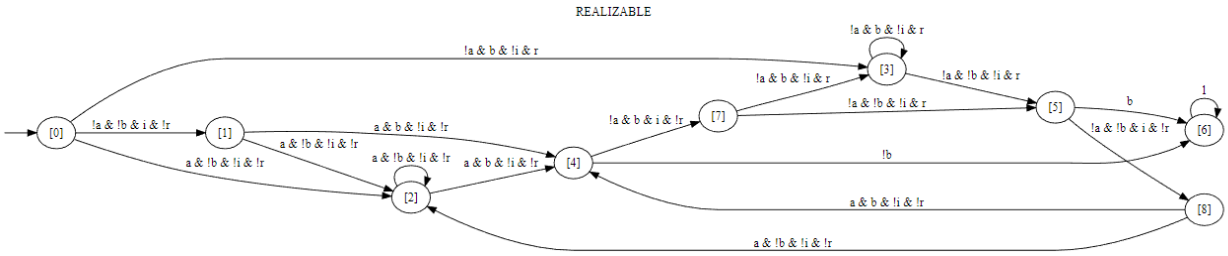
```
a (assist)
r (recharge)
i (idle)
```

Assumptions:

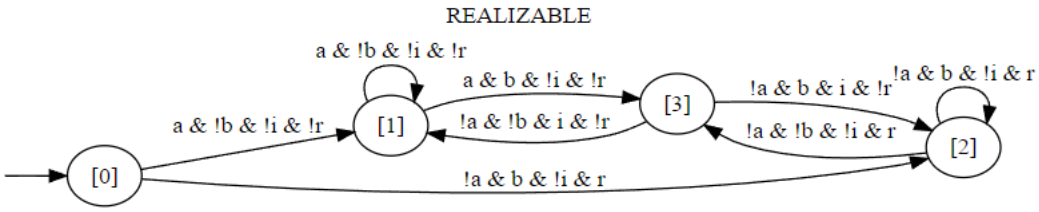
```
G NOT (!b & (X b) & (X X !b))
G NOT (b & (X !b) & (X X b))
```

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 Minimized (2ts).