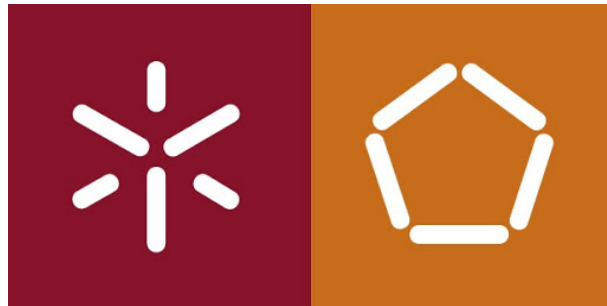


Software Architecture and Calculi - Problem 1



Luís Ribeiro (A85954)
Mestrado em Engenharia Informática
Universidade do Minho

1 | Exercise 1

There are plenty of languages to express process algebra, defined by some properties such as, **forcing a synchronization** between actions by assigning complementary actions, meaning that those processes will evolve by a **non-observable action**. In this case, since you want to force synchronization between actions, typically you want to prevent these actions from 'running' as one, so blocking actions individually is another add-on. These properties are the main ones that you want to keep in mind before introducing mCRL2.

mCRL2 is a formal specification language with an associated toolset. This **toolset** provides a generic process algebra, based on ACP, where we can specify parametric multi-actions, sequential processes (sequential composition) with non deterministic behaviour (meaning that choice $+$, is included) and parallelism within processes (Parallel composition).

Since there is no concept of complementary action in this tool, there is a predefined function, named Communication (comm) that allows us to synchronize actions between parallel processes. We can also block, allow and hide (declaring as non observable action τ) actions. This tool also provides some signatures and definitions, that will be mentioned in the following section.

Since mCRL2 can also be used for **verification**, μ -calculus and properties written in modal logic are supported as well.

In order to really show how this tool works, we're going to analyse a well know problem known as "The River Crossing Problem".

1.1 The River Crossing Problem

There are a farmer with a fox, a goose and a pack of beans standing at one side of a river. He wants to cross and for this purpose he has a small boat that can only contain him and one item (2 beings at most). If the fox is left with the goose, the fox will eat the goose. If the goose is left with the beans, the beans is consumed. The question is to find a strategy to bring the man to the other shore (right shore) without losing any of his items.

To fully understand my resolution of this problem, I separated my resolution in **3 separated steps**. First we will go through the declarations of types and functions used. Then, we will see how we declare each action and each process. At last, parallelism and actions properties (mentioned before), will be shown as well.

1.1.1 Declaration of Types and Functions

```
sort
  Position = struct left | right;
  Being = struct fox | goose | beans;
  RiverShores = struct rshores(Set(Being), Set(Being));
```

The keyword **sort** is used to declare data-types, and the **struct** keyword combined with sort reference is to define an alias to a structured sort. The **river has 2 shores** or margins, each defined by its position, thus I created a data-type **Position** that will be either **Right** or **Left**. Then I declared which **beings** are involved in the problem (**Beings data-type**). Since the farmer will dictate how the system evolves, it will be consider as an individual process (like an outside watcher).

For **purposes of controlling which beings are at each shore**, I created **RiverShores data-type** with a pair of Set of Beings for each shore. This last data-type will also be used to check if every being reached the other side of the river, the fox is left with the goose or if the goose is left with the beans.

```
map
  oPos: Position -> Position;
  beings: RiverShores # Position -> Set(Being);
  being2right: RiverShores # Being -> RiverShores;
  being2left: RiverShores # Being -> RiverShores;
var
  l,r: Set(Being);
  b: Being;

eqn
  oPos(left) = right; oPos(right) = left;
  beings(rshores(l, r), left) = l; beings(rshores(l, r), right) = r;
  being2right(rshores(l,r), b) = rshores(l - {b}, r + {b});
  being2left(rshores(l,r), b) = rshores(l + {b}, r - {b});
```

The keyword **map** is reserved to declare the functions types and parameters. The keyword **var** is used to declare variables. **Eqn** is the keyword for functions definitions. Consequently, 3 different auxiliary functions are defined to the processes yet to be explained.

Function **oPos** returns the opposite Position of the one given as argument. The Left Shore is the opposite of Right Shore, and vice-versa. Function **beings** returns the set of beings in a given shore/position. If Position equals left, then we return the first Set of Beings. If Position equals right, then we return the second Set of Beings. The remaining functions are used to indicate when the farmer crosses the river with a being, and when that happens, the beings at each river shore must be updated. Meaning, if a being is moved from the left shore to the right shore, then we must remove it from the first set of beings (Left Shore related) and add it to the second set of beings (Right Shore related). Notice that we commonly use the data-type **rshores(RiverShores)** declared above as our main struct.

1.1.2 Declaration of Actions and Processes

```
act
  crossBeing, farmerBeing: Being # Position; % Beings actions
  crossFarmer: Position; % Farmer
  cross: Being # Position;
  is_eaten: Being; gg;

proc
  Farmer(s: RiverShores, p:Position) =
    (beings(s, right) == {fox,goose,beans}) -> gg . delta
    + ({fox,goose} <= beings(s, oPos(p))) -> is_eaten(goose) . delta
    + ({goose,beans} <= beings(s, oPos(p))) -> is_eaten(beans) . delta
  + (!({fox,goose} <= beings(s, oPos(p)))
    && !({goose,beans} <= beings(s, oPos(p)))
    && (beings(s, right) != {fox,goose,beans})) ->
  (
    crossFarmer(oPos(p)) . Farmer(s,oPos(p)) % Crossing the river alone.
    + sum i:Being. (i in beings(s, p) &&
      (beings(s, right) != {fox,goose,beans}))
      -> farmerBeing(i,oPos(p)) .
    (p == left) ->
      Farmer(being2right(s,i),oPos(p))
    <>
      Farmer(being2left(s,i),oPos(p)));

  Being(i: Being, p: Position) =
    crossBeing(i,oPos(p)) . Being(i,oPos(p));
```

In order to explain why I declared each action mentioned in the keyword `act`, I'll show you my thought process on each Process declared above in the keyword `proc`.

First I assumed that the **Farmer** would be consider as an "outside controller", that watches and controls every cross between shores. So the process **Farmer(s: RiverShores, p:Position)** must receive the struct defined above, `rshores`, that acknowledges which beings are at each shore. The Position where the Farmer is is also relevant to this process. In this Process, I must check if all beings crossed the river to the right shore, meaning "Job's finished", so we proceed to take the **action gg followed by delta, this being the definition of a deadlock**. On top of that, it is also needed to check whether the fox and the goose or the goose and the beans are at the opposite shore of the Farmer, this means that the goose or the beans **will be eaten** (the action `is_eaten(b:Being)` is followed by delta, meaning deadlock). Finally, **if none of those conditions above are verified**, the Farmer can cross the river alone (`crossFarmer(oPos(p:Position))`) or with another being besides him (`farmerBeing(i:Being,oPos(p:Position))`). The farmer can cross with one of the 3 beings declared above, so it must be declared a **sum** (for all beings), **leading to Non-Determinism**. Note that, this Process must be called after an action is performed, with the Position of the farmer getting updated by the function `oPos(p:Position)`.

The other declared Process, named **Being(i:Being, p:Position)**, receives a being and a position as argument. This one is continuously recursive after performing the action `crossBeing(i:Being,oPos(p:Position))`, where the Position of the being is updated.

There is an action that I didn't referred above, which is the action `cross(i:Being,p:Position)`. **This action is needed for synchronization purposes.**

1.1.3 Process Synchronization and Parallelism

```
init allow({gg, is_eaten, crossFarmer, cross},
  comm({farmerBeing | crossBeing -> cross},
  Being(fox,left) || Being(goose,left) || Being(beans,left)
  || Farmer(rshores({fox,goose,beans}, {}), left)));
```

As I said before, there must be an **action which synchronizes both processes** when the Farmer crosses the river with a Being besides him, meaning that in the process Being, `crossBeing` must be executed. So, the action `farmerBeing`, that takes place in **Farmer**, must be **synchronized** with **Being's** action `crossBeing`, to ensure that only one being crosses the river at time. Both actions receive a being as parameter, so this is restrict to one being per action. Therefore, to **force this synchronization**, we use the predefined function **comm**, declaring that those 2 actions are synchronized, resulting in a new action named `cross`. Since we don't want to let these actions to be executed separately, we use the predefined function **allow**, **filtering those actions that can be 'executed' individually.**

Finally, we need to declare which Processes are parallel to each other and how they initially are composed. So we have **3 Beings** and **one Farmer**. All these must be **initially located at the left shore**. The struct `rshores` must be **initialized with all beings at the left shore**, and none at the right shore.

1.2 mCRL2 IDE and running commands

If you are new to this tool, I suggest using the mCRL2 IDE. since its aimed at new users to use the mCRL2 tool-set and its language. With this tool the user can write and validate an mCRL2 specification, simulate the mCRL2 specification, view the transition graph, validate μ -calculus properties and much more.

```
$ mcrl2ide farmer.mcrl2
```

Here you have every tool available for mCRL2: [mCRL2 Tools](#).

2 | Exercise 2

Consider the following description of a two-position buffer with acknowledgements. Note the process is built from copies of a 1-position buffer also with acknowledgements: it acknowledges in \bar{r} the reception of a message and waits in t the confirmation that a message sent was arrived to its destination.

$$B_s \triangleq (B(in, mo, mi, r) | B(mo, out, t, mi)) \setminus_{\{mo, mi\}} \\ B(in, out, t, r) \triangleq in.\overline{out}.t.\bar{r}.B$$

Use MCRL2 to help answering the following questions.

- (a) Check whether the behaviour of B_s is the intended one (drawing, for this purpose, the corresponding transition graph).
- (b) Find a solution to the problem detected (if any) and draw the corresponding transition graph.
- (c) Explain how the specification given (or your new solution) can be adapted to describe buffers with an arbitrary, but fixed number of positions.
- (d) Formulate safety and liveness properties that process B_s may verify and use MCRL2 to check them.

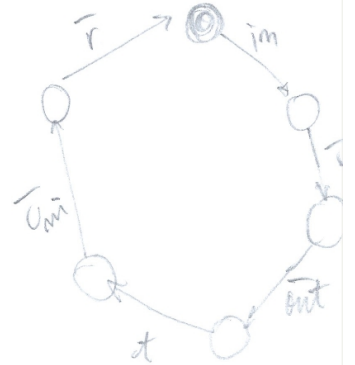
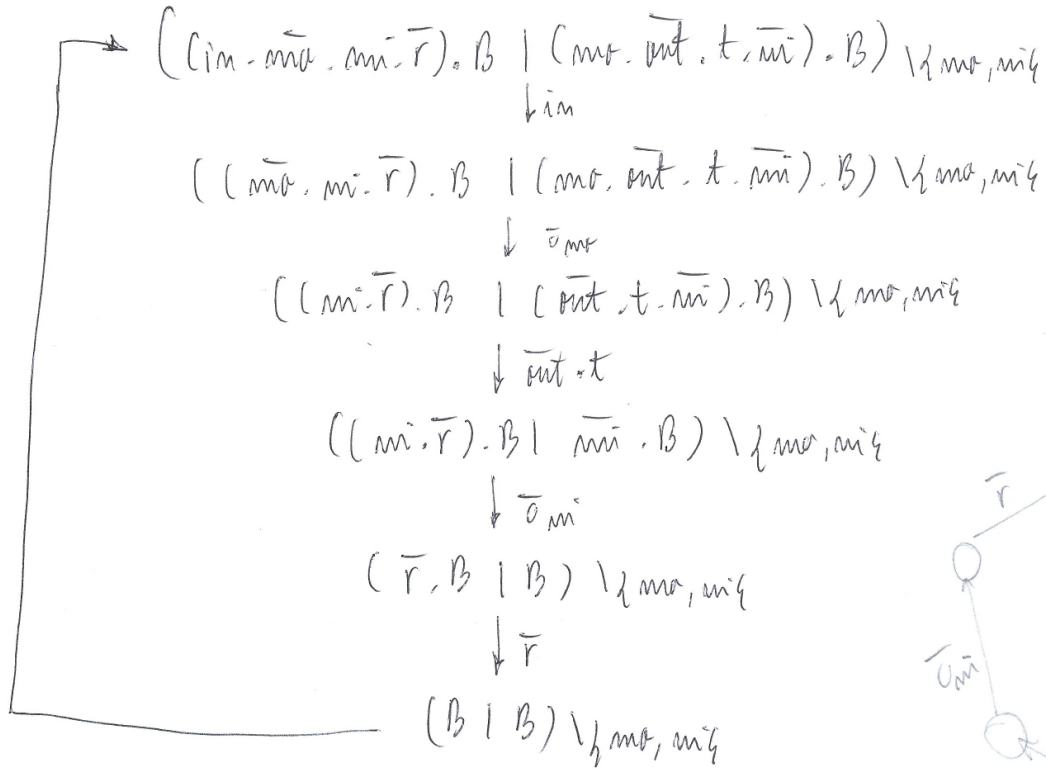
2.1 Question a

Check whether the behaviour of B_s is the intended one (drawing, for this purpose, the corresponding transition graph).

To check the behaviour of B_s , I tried to see how both B_s would evolve in parallel. As you can see right below, there's my resolution to this problem.

$$B_s \triangleq (B(in, mo, m, r) \mid B(mo, out, t, mi)) \setminus \{mo, mi\}$$

$$B(in, out, t, r) \triangleq in.out.t.\bar{r}.B$$



I also used mCRL2 for the purpose of validating my resolution and I obtained the same transition graph. Here is my mCRL2 specification:

```
act in1, in2, out1, out2, t1, t2, r1, r2, m1, m2;

proc B1 = in1 . out1 . t1 . r1 . B1;
     B2 = in2 . out2 . t2 . r2 . B2;
```

```

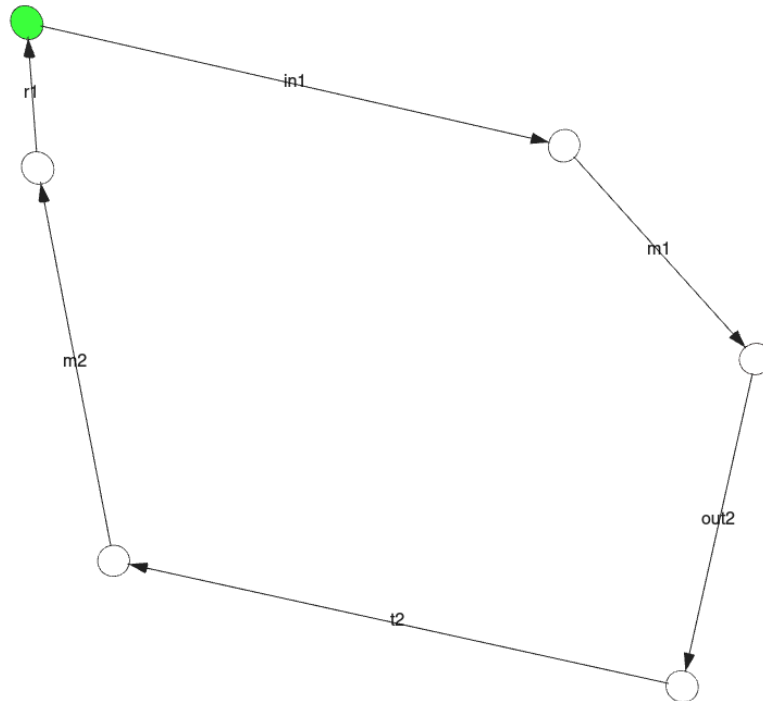
init block({out1, in2, t1, r2},
  comm({ out1 | in2 -> m1, t1 | r2 -> m2 },
  B1 || B2 % Bs
));

```

Since processes can't receive actions as arguments, I need to declare different actions for each process B_1 and B_2 , and some must be synchronized (forcing action synchronization). Also, the actions that have been synchronized can't be executed separately, so they must be denied/blocked.

Note that, $m1$ and $m2$ correspond to an internal synchronized (non-observable) actions τ_{mo} and τ_{mi} , respectively. I could have used the predefined action named **hide**, declaring both $m1$ and $m2$ as τ , but for readability purposes, I haven't.

Here is the corresponding transition graph:



After analysing the behaviour of B_s and the corresponding transitions graphs, we can conclude that it does not correspond to what we were expecting. Although action t happens after the message being sent to its destination (\overline{out}), \bar{r} , because of its misplacement, doesn't acknowledge the reception of the message. In order to do so, \bar{r} should happen right after the action in .

2.2 Question b

Find a solution to the problem detected (if any) and draw the corresponding transition graph.

As I said in the previous question, \bar{r} should happen after the action in and before the first internal message between Processes (τ_{mo}), so as to achieve the acknowledgement of the reception of a message.

$$B(in, out, t, r) \triangleq in.\bar{r}.\overline{out}.t.B$$

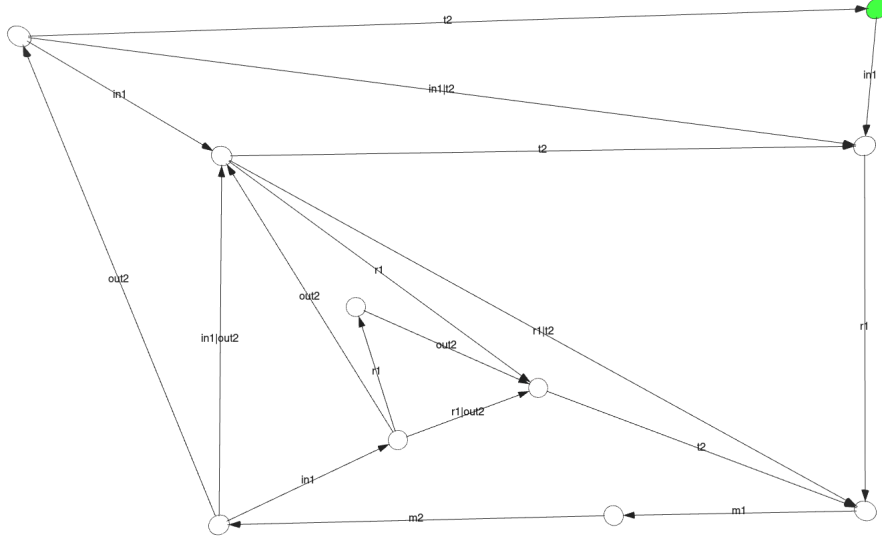
Here is my new mCRL2 specification:

```
act in1, in2, out1, out2, t1, t2, r1, r2, m1, m2;

proc B1 = in1 . r1 . out1 . t1 . B1;
    B2 = in2 . r2 . out2 . t2 . B2;

init block({out1, in2, t1, r2},
    comm({ out1 | in2 -> m1, t1 | r2 -> m2 },
    B1 || B2
));
```

By using mCRL2 and its tools, the corresponding transition graph was obtained.



As we can see now, the behaviour of B_s is not as linear or as simple as the last one. Although both processes in parallel must be synchronized using τ_{mi} and τ_{mo} (internal, non-observable actions), there's the possibility of receiving 2 different messages (**message 1 and message 2**), and before totally processing message 1 ($\overline{out}.t$), message 2 is already being received(in) and checked(\bar{r}). However, message 2 can't be sent to its destination, while message 1 isn't totally processed, adding the possibility of skipping some sequential steps.

2.3 Question c

Explain how the specification given (or your new solution) can be adapted to describe buffers with an arbitrary, but fixed number of positions.

For the resolution of this question, we must consider that we need to create as many processes as the number of positions that the buffer has. By that, if we have a 3-position buffer, we must declare 3 parallel processes, each one with their set of actions, because, as explained before, processes can't receive actions as arguments.

For synchronization, we must be aware that the action *out* of the process B_i must be synchronized with the action *in* of the process B_{i+1} .

2.4 Question d

Formulate safety and liveness properties that process B s may verify and use MCRL2 to check them.

SAFETY

It's impossible to send a message from B_1 to B_2 , after a prior message from B_1 to B_2 where action t wasn't executed (meaning there was a message that didn't reach its destination). Even though we can receive a new message and acknowledge it while the previous one isn't totally processed (t , same as t_2 , action wasn't executed), we can't send the new message to the other parallel process (B_1 to B_2) by action τ_{mo} (same as m_1).

```
[m1 . !t2* . m1 ] false
```

LIVENESS

After a message is received, at least one message will arrive to its destination. It's referred as t_2 , which is the process right after sending the message to its destination.

```
[in1]<true* . t2> true
```

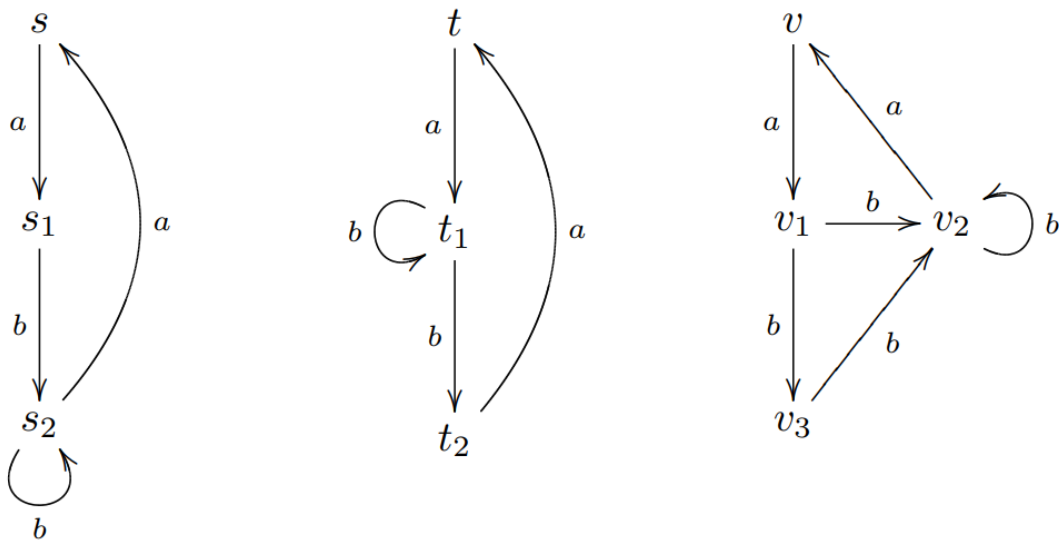
Since mCRL2 provides a verification tool, I tested both properties and obtained the result **true**, which was what we wanted. To check these properties, you must execute the following commands:

```
$ mcrl22lps e2b.mcrl2 e2b.lps
$ lps2pbcs --formula=property.mcf e2b.lps | pbcs2bool
```

The `property.mcf` mentioned above, must be one of the properties defined, separated in an individually *mcf* file.

3 | Exercise 3

Consider the following labelled transition systems.

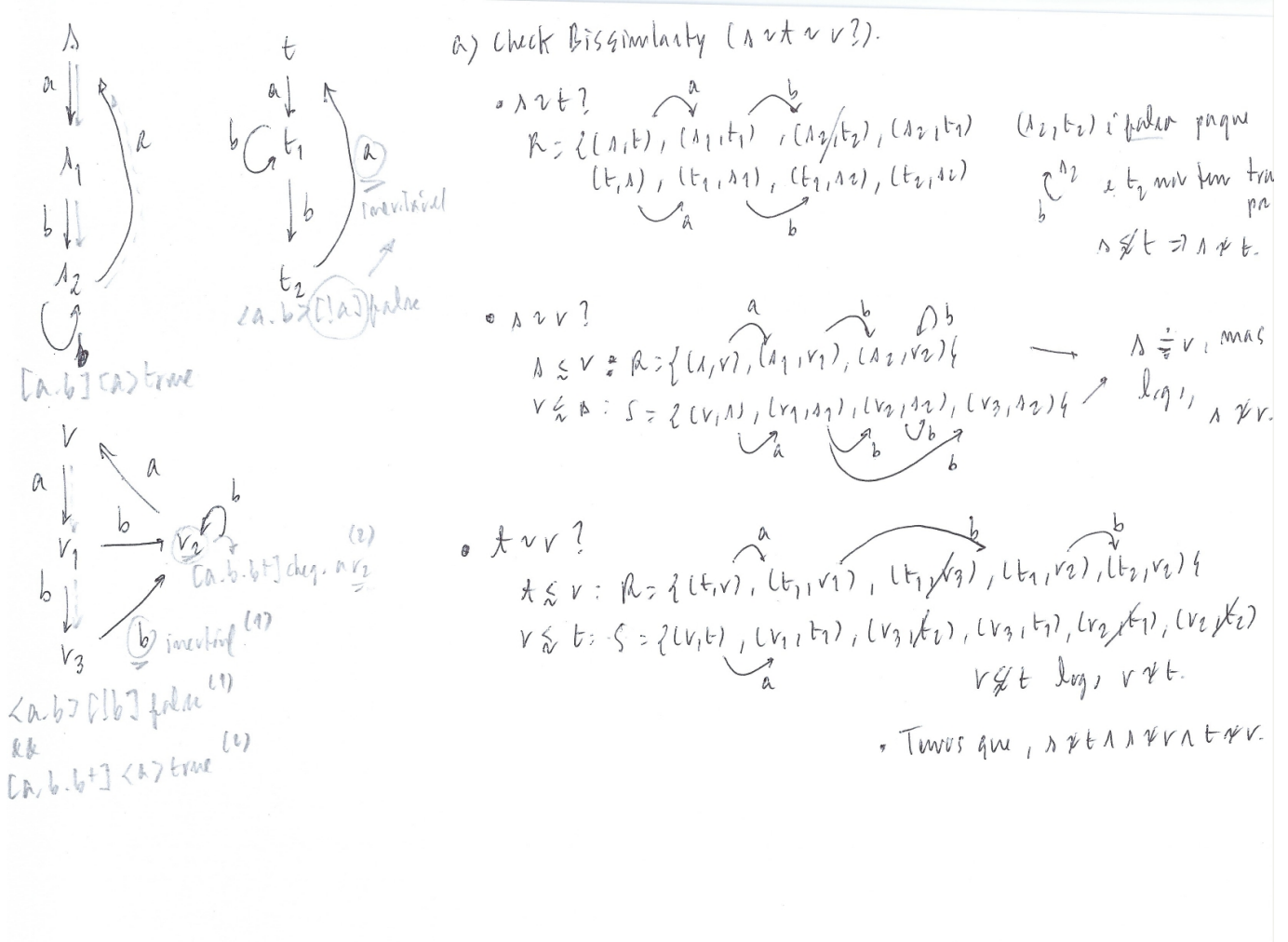


- Show that states s , t and v are not bisimilar and determine the modal properties which distinguish between them.
- Use MCRL2 to verify your answer to the previous question.

3.1 Question a

Show that states s , t and v are not bisimilar and determine the modal properties which distinguish between them.

Bellow this, we can see my resolution to this problem:



$s \sim t?$

Note that, s isn't simulated by t , because the state s_2 doesn't have a corresponding t state. So, we conclude that $s \not\sim t$.

$s \sim v?$

LTS s is equal to the upper side of LTS v , meaning that s is clearly simulated by v (Relation R). v is also simulated by s (Relation S). Although both states simulate each other, they are not bisimilar, because $S \neq R^\circ$. By that, we conclude that s and v are equisimilar ($s \doteq v$, but $s \not\sim v$).

$t \sim v$?

It's clear that v isn't simulated by t , because there is no corresponding state on t for state v_2 , since this last can evolve through action a and action b .

Why is Bisimilarity so important?

If 2 LTS, s and s' , are bisimilar means that no modal property can distinguish them. For that reason, when we are trying to specify properties that **restrictively represents** the LTS s , we must guarantee that s and s' are not bisimilar.

Since we proved that there's no bisimulation relation between them, we can now look at the modal properties which distinguish each from the remaining.

Modal Property of s :

If you noticed, s wasn't simulated by t because s_2 had the possibility to transit to s via action a or looping via b , where no state t could do that. We identify state s_2 by some state s that can transit either through b or through a , and until we reached s_2 , there is no non-determinism in the previous states. If we go through action a ($s \rightarrow s_1$) and then through b ($s_1 \rightarrow s_2$), we reach s_2 . Whereas in t and v this is not true at all, because going over a then b in t , we might reach t_2 or we can loop in t_1 , and in v we can reach v_2 or v_3 .

Note that only v_2 has the same non-determinism property as s_2 . However, v_3 only transits via b , thereby not all states on v , reached from taking action a and then b , have the ability of transiting via action a .

So, this is **the modal property that uniquely identifies s** :

$[a \ . \ b] \langle a \rangle \text{ true}$

This means that every state reached from the composition of actions a and b , has the possibility of transit over a .

Modal Property of t :

As mentioned, s wasn't simulated by t , because neither t_1 or t_2 had the possibility of transit via actions a and b , as s_2 could. Note that, t_1 inevitably transits through b , and t_2 inevitably transits through a . Both s_1 and v_1 have the same properties as t_1 , where transition through b is inevitable, and are reached via action a . Whereas, no state in s and in v , that are reached through $a \cdot b$, necessarily transits via a , as t_2 does.

So, this is **the modal property that uniquely identifies t** :

$\langle a \ . \ b \rangle [!a] \text{ false}$

This means that we can reach a state where action a is inevitable, after taking action a and then b . This state is t_2 .

Modal Property of v :

To fully understand how v differs from s and t , we must analyse how v_1 transits towards to 2 distinct states by b , those being v_2 and v_3 . As explained before, v_2 has the same behaviour as s_2 . However, the modal property defined to express s is false on v , because on this last, not all transitions through $a \cdot b$ reach a state where a is possible, as v_3 can only evolve over b . But, v_3 evolves towards v_2 by b , where transaction over a is possible. Since v_2 can loop to itself through b , we conclude that the following property is **true** on v :

```
[a . b . b] <a> true
```

We need to find another property, as this last is still true on LTS s , since v_2 can loop to itself via b . As mentioned, v_3 can only evolve through b , therefore b is inevitable on that state, and its reached by composing $a \cdot b$. There is no corresponding state in s . By that, we have that there's a state reached by $a \cdot b$ where action b is inevitable.

```
<a . b> [!b] false
```

The result of the logic conjunction of both properties is **the modal property that uniquely identifies v** :

```
<a . b> [!b] false && [a . b . b] <a> true
```

3.2 Question b

Use MCRL2 to verify your answer to the previous question.

First, we need to create a mCRL2 specification for each LTS presented.

For that, I created 3 separated files, where each one represents a LTS. To check the model properties determined in the last question, I also created 3 files, in order to check each property on its corresponding specification.

Here are my specifications and properties:

```
s.mclr2:
  act a,b;
  proc S = a . b . S2;
    S2 = (b . S2) + (a . S);
  init S;

s.mcf:
  [a . b] <a> true

t.mcrl2:
  act a,b;
  proc T = a . T1;
    T1 = (b . T1) + (b . a . T);
  init T;

t.mcf:
  <a . b> [!a] false
```

```

v.mcrl2:
  act a,b;
  proc V = a . V1;
    V1 = (b . V2) + (b . V3);
    V2 = (b . V2) + (a . V);
    V3 = b . V2;
  init V;

v.mcf:
  <a . b> [!b] false && [a . b . b] <a> true

```

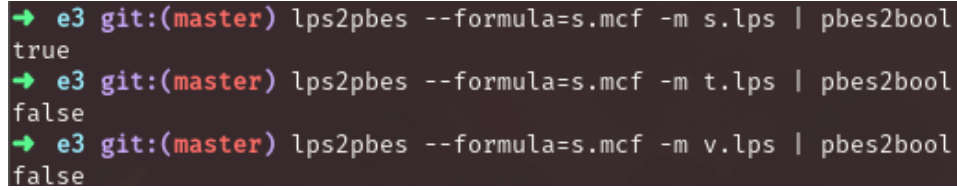
Now that we declared each property and specification, we need to use the verification tool that mCRL2 provides. To check these properties, you must execute the following commands:

```

$ mcrl22lps lts.mcrl2 lts.lps
$ lps2pbcs --formula=property.mcf lts.lps | pbcs2bool

```

The `property.mcf` mentioned above, must be one of the properties defined, separated in an individually *mcf* file. Same for `lts`, which must be replaced for one of the LTS specifications defined.

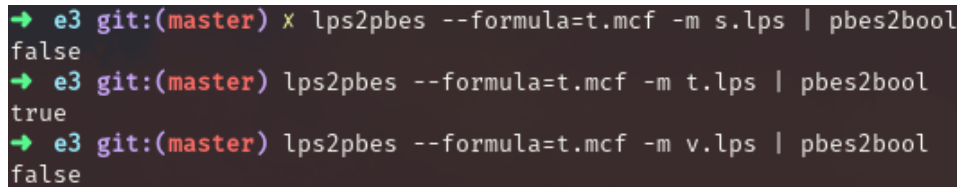


```

→ e3 git:(master) lps2pbcs --formula=s.mcf -m s.lps | pbcs2bool
true
→ e3 git:(master) lps2pbcs --formula=s.mcf -m t.lps | pbcs2bool
false
→ e3 git:(master) lps2pbcs --formula=s.mcf -m v.lps | pbcs2bool
false

```

Figura 3.1: Verification result of `s.mcf` on each LTS.

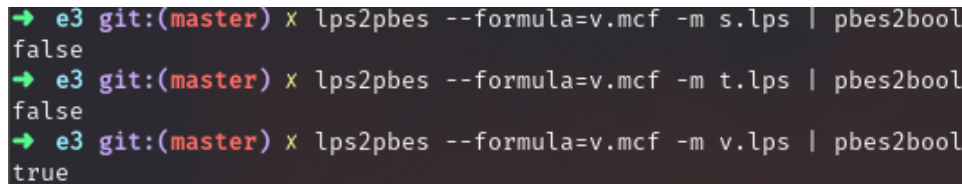


```

→ e3 git:(master) x lps2pbcs --formula=t.mcf -m s.lps | pbcs2bool
false
→ e3 git:(master) lps2pbcs --formula=t.mcf -m t.lps | pbcs2bool
true
→ e3 git:(master) lps2pbcs --formula=t.mcf -m v.lps | pbcs2bool
false

```

Figura 3.2: Verification result of `t.mcf` on each LTS.



```

→ e3 git:(master) x lps2pbcs --formula=v.mcf -m s.lps | pbcs2bool
false
→ e3 git:(master) x lps2pbcs --formula=v.mcf -m t.lps | pbcs2bool
false
→ e3 git:(master) x lps2pbcs --formula=v.mcf -m v.lps | pbcs2bool
true

```

Figura 3.3: Verification result of `v.mcf` on each LTS.

Every property is **true** to its corresponding LTS specification, and **false** to the others, as it was supposed to.