Name: Stefan Cretu
Student number: 3048438
Specialization: SE&DS

FMCS – Homework 2

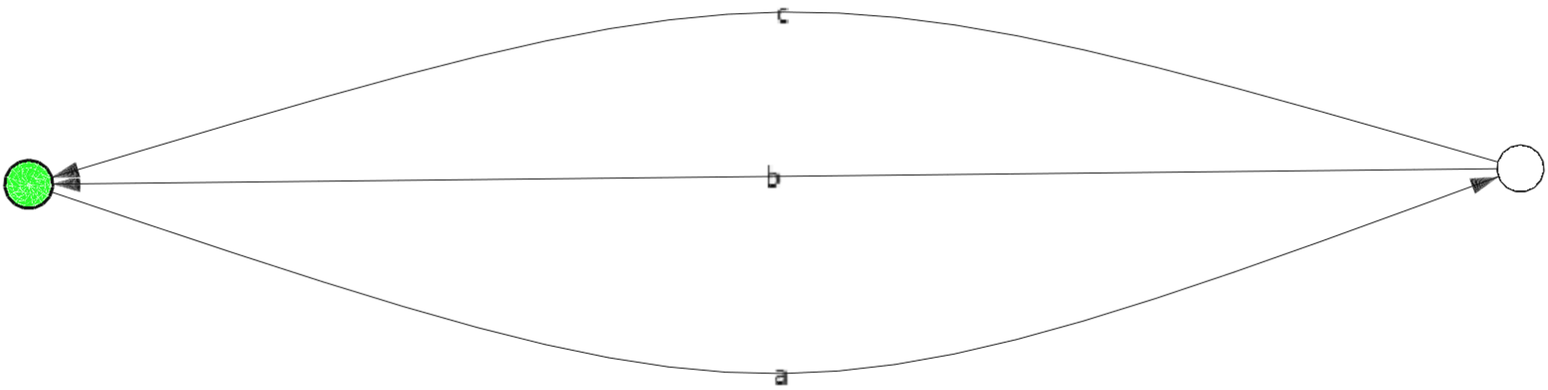
Exercise 1:

Code for process P:

```
act
    a,b,c;

proc
    P = a.P1;
    P1 = b.P + c.P;

init P;
```

```
[20:43:54 info] Parsing and type checking specification
[20:43:54 verbose] type checking process specification...
[20:43:54 info] Specification is a valid mCRL2 specification
```

The LTS graph for process P: the green dot is the P state, and by performing *a* action it goes to P1 state, the white dot. From the state P1, it can perform either *b* or *c* actions to go back to P state.

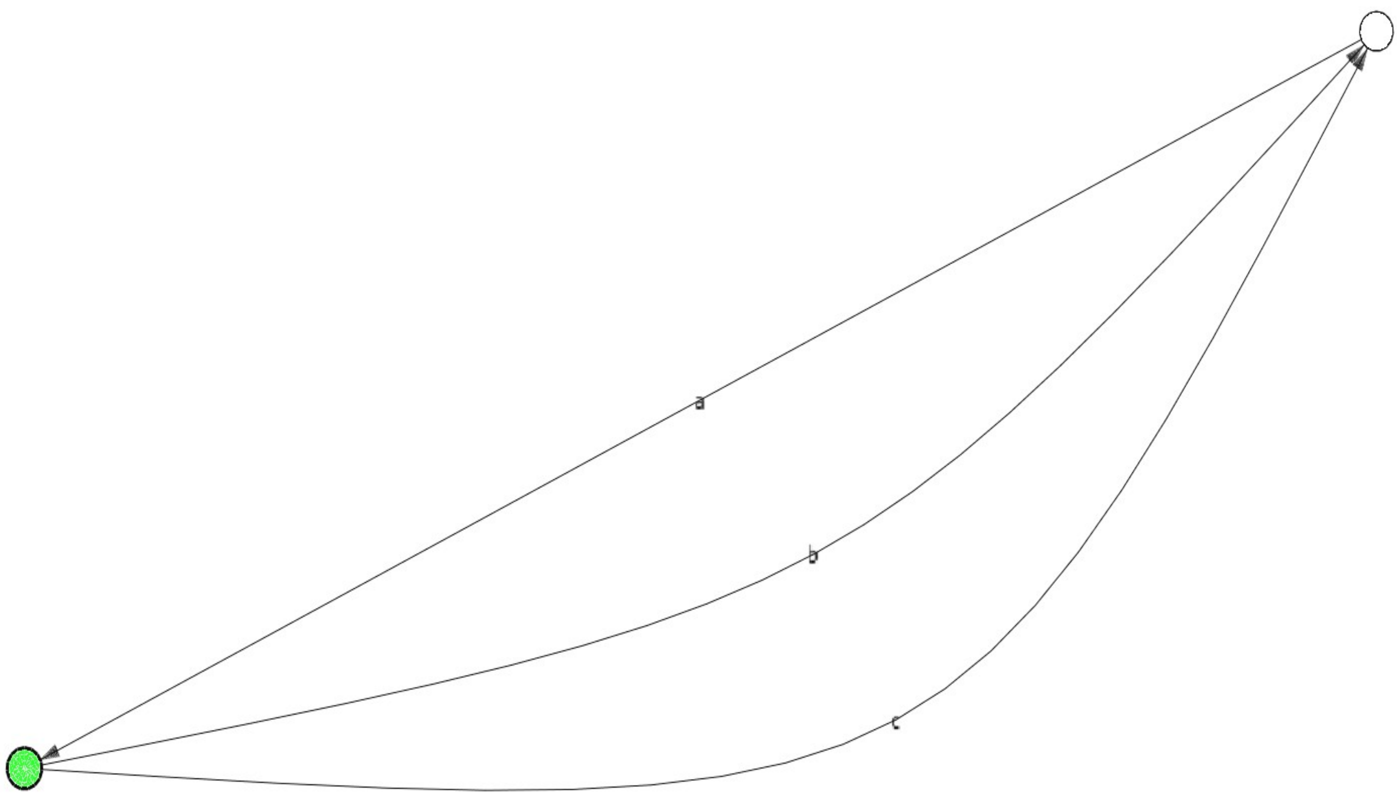


Code for process Q:

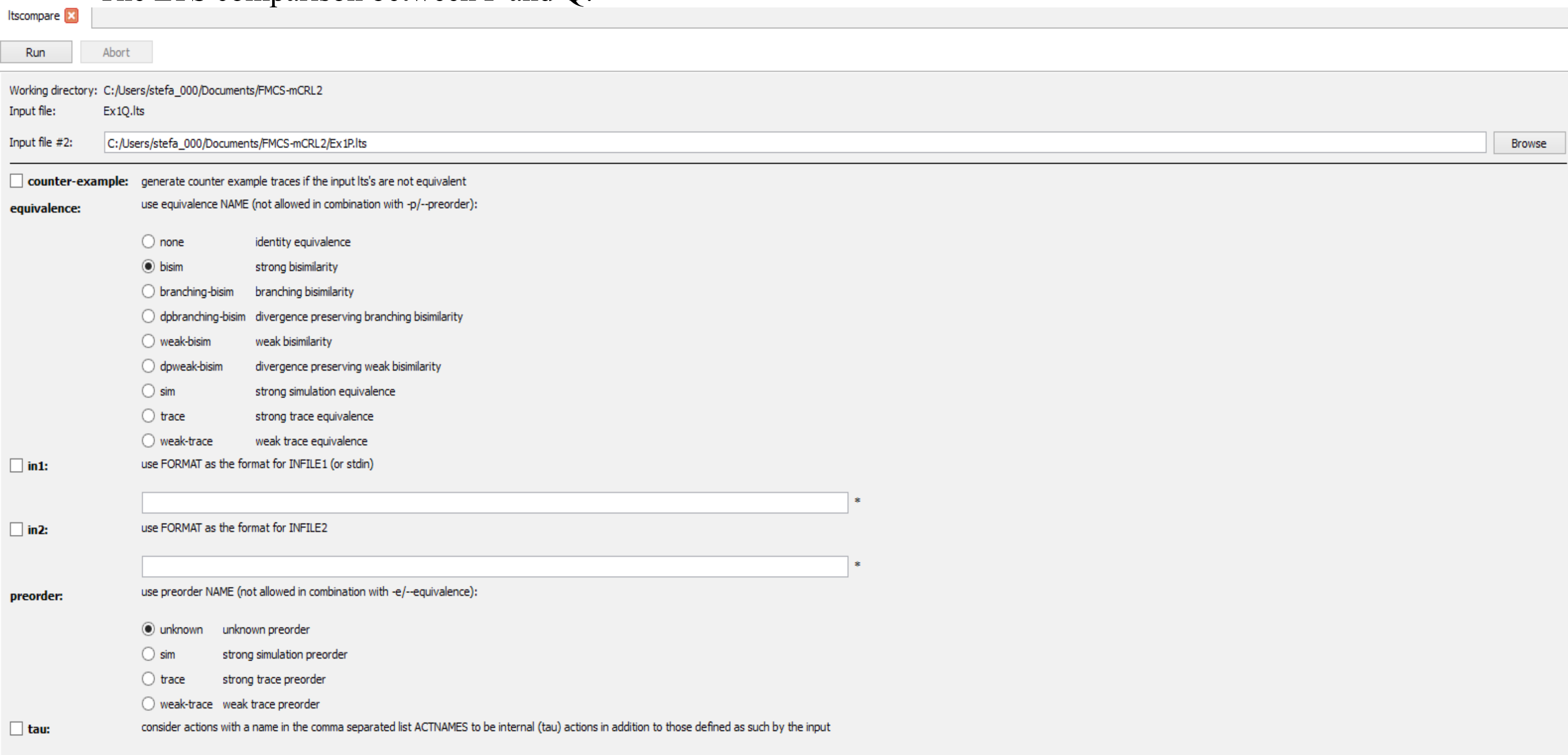
```
act
    a,b,c;
proc
    Q = a.Q1;
    Q1 = b.Q2 + c.Q;
    Q2 = a.Q3;
    Q3 = b.Q + c.Q2;

init Q;
```

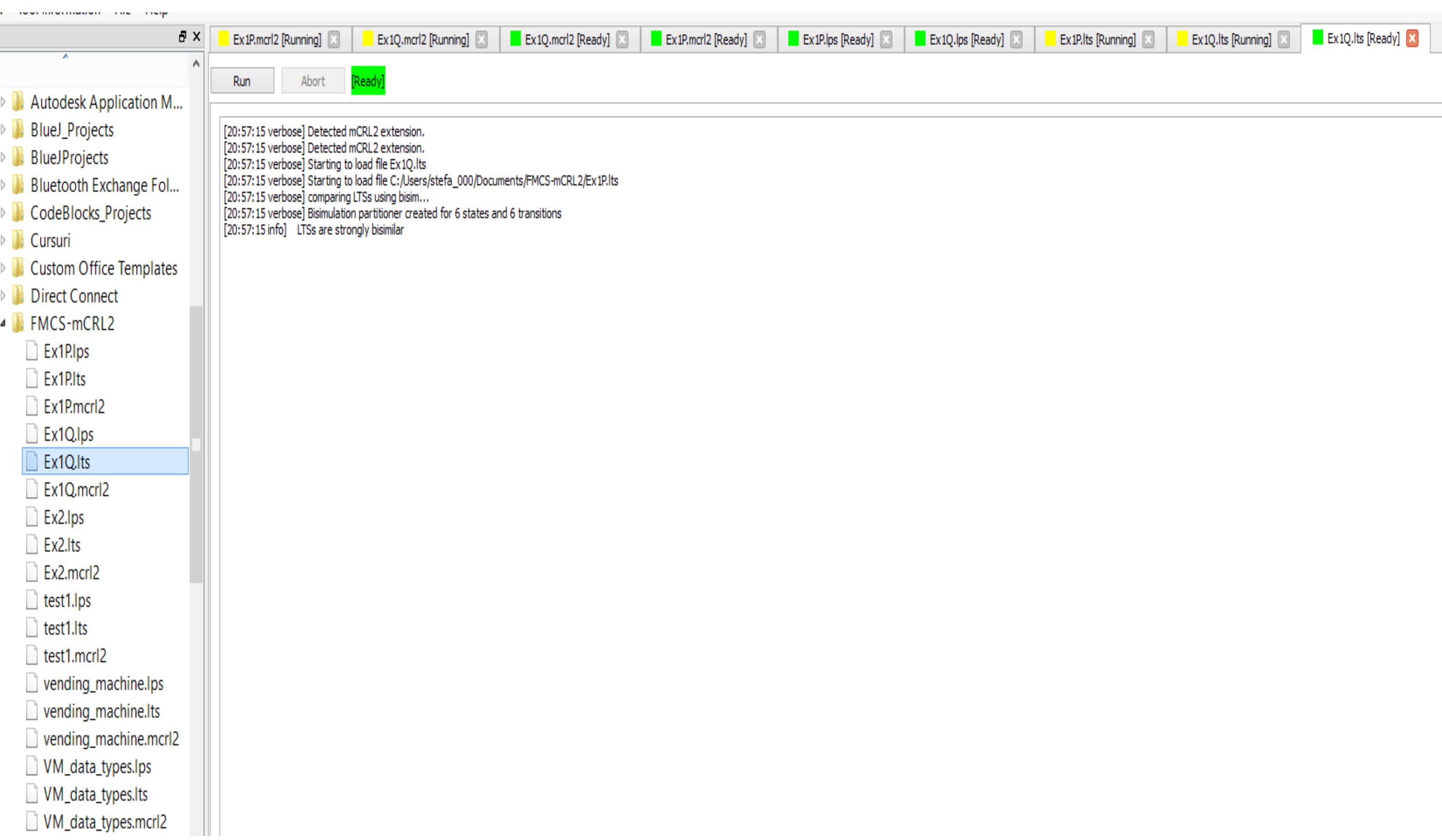
```
[20:40:41 verbose] type checking process specification...
[20:40:41 info] Specification is a valid mCRL2 specification
[20:12:23 info] Parsing and type checking specification
[20:42:23 verbose] type checking process specification...
[20:42:23 info] Specification is a valid mCRL2 specification
```

The LTS graph for process Q:



The LTS comparison between P and Q:





As expected, processes P and Q are strongly bisimilar.

Exercise 2:

Code:

Ex2

```
%define actions
act
    send, send', sendSync;
    ack, ack', ackSync;
    error, error', errorSync;
    trans, trans', transSync;
    acc, del';
%define processes Send, Rec and Med, which will run in parallel
proc
    Send = acc.Sending;
    Sending = send'.Wait;
    Wait = ack.Send + error.Sending;

    Rec = trans.Del;
    Del = del'.Ack;
    Ack = ack'.Rec;

    Med = send.Med';
    Med' = tau.Err + trans'.Med;
    Err = error'.Med;

init
    hide( [sendSync, errorSync, transSync, ackSync],
           allow( [acc, del', sendSync, errorSync, transSync, ackSync],
                  comm( [send|send' -> sendSync, ack|ack' -> ackSync, error|error' -> errorSync, trans|trans' -> transSync],
                         (Send || Med || Rec)
                      )
                )
        );
```

```
[17:42:12 info] Parsing and type checking specification
[17:42:12 verbose] type checking process specification...
[17:42:12 info] Specification is a valid mCRL2 specification
```

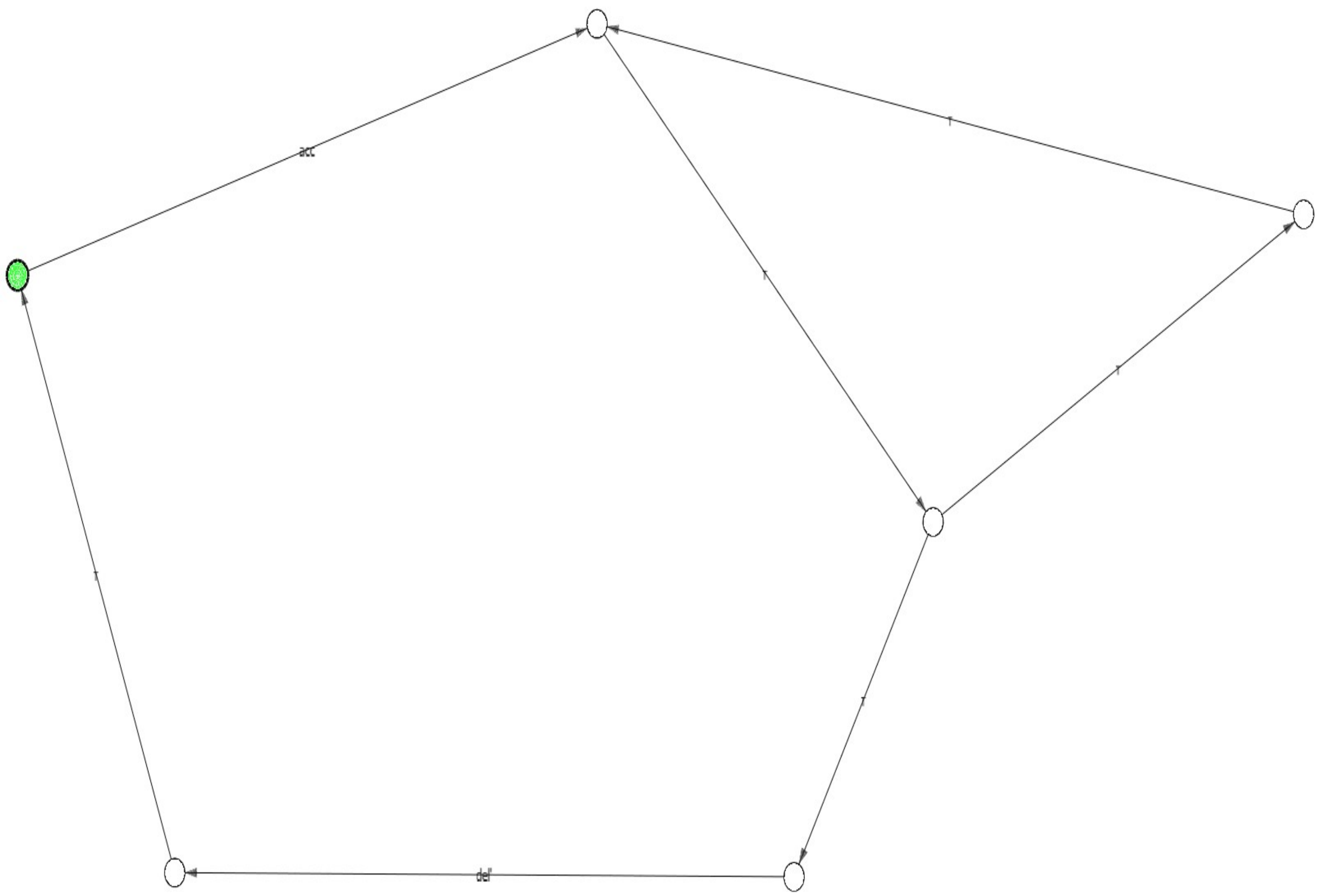
In the *act* section, I declared actions: *send, ack, error* and *trans*. The actions *send', ack', error' trans'* can be seen as the co-action of the above mentioned 4 actions, but here are defined as proper actions, because mCRL2 doesn't have co-actions like CCS. Finally, for each pair action – co-action, I declared a channel for synchronization, namely: *sendSync, errorSync, transSync* and *ackSync.* Concretely, when actions *send* and *send'* happen, their synchronization occurs on *sendSync* channel, and the same for the other 3. Giving the fact that all those 4 actions (*sendSync, errorSync, ackSync and transSync)* are synchronizations, they can be seen as *tau* actions (internal actions).

Also, there are defined *acc* and *del'* actions which can be seen as external actions.

In the *proc* section, I defined the processes as they are given in tutorial. The processes *Med, Rec* and *Send* will compute in parallel.

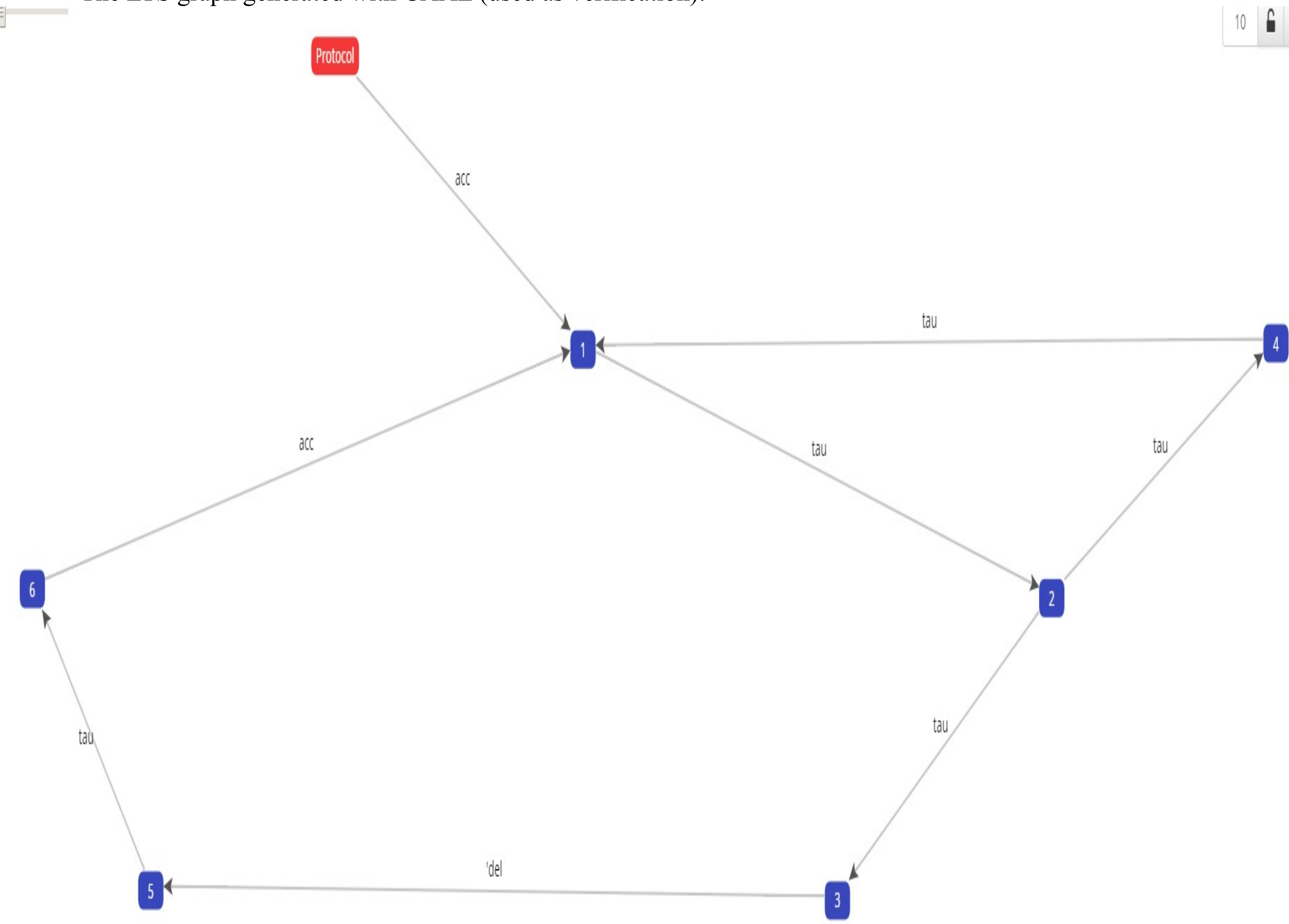
In the *init* section, I put the synchronization actions under the *hide* token, as they are restricted actions. Under the *allow* token, I wrote all those 4 synchronization actions and the external ones, namely *del'* and *acc*, as I want to let the communication to occur on those 6 channels. In the end, under the *comm* token, I specified that I want to let the communication between *send* and *send'* to occur on the *sendSync* channel, and the same for the other 3 synchronizations. Then, I specified that the processes *Med, Rec* and *Send* are running in parallel.

The LTS graph generated with mCRL2:



There are 5 *tau* actions, one is that one *tau* might be executed when there is an error in the medium, as specified in the code, and the other 4 stand for *sendSync, errorSync, ackSync and transSync.* Also, there can be seen *acc* and *del'* external actions.

The LTS graph generated with CAAL (used as verification):



Exercise 3:

→ Version 1:

The code:

Ex3V1

```
act
%VM can receive five, ten, fifteen and can send chocolate, energyBar and water
    takeFive, takeTen, takeFifteen, giveChocolate, giveEnergyBar, giveWater;
%User sends five and receives chocolate;
    giveFive, takeChocolate;
%Sync channels
    coinSync, chocolateSync;

proc
    User = giveFive.takeChocolate.User;
    VM = takeFive.giveChocolate.VM + takeTen.giveEnergyBar.VM + takeFifteen.giveWater.VM;

init
           allow( {coinSync, chocolateSync},
                   comm( {takeFive|giveFive -> coinSync, giveChocolate|takeChocolate -> chocolateSync},
                          VM || User
                   )
               );
```

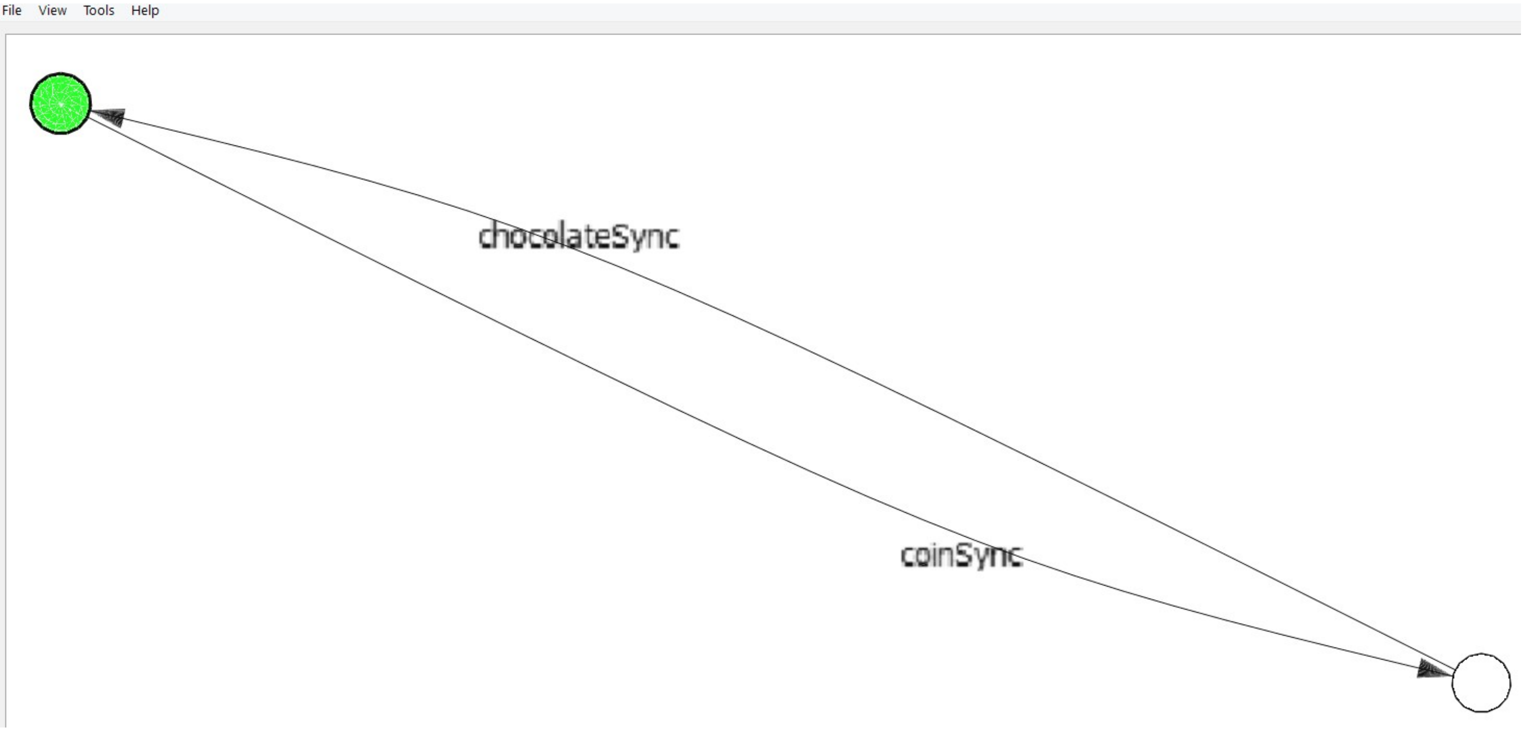
```
[12:17:59 verbose] type checking process specification...
[12:17:59 info] Specification is a valid mCRL2 specification
```

The User process does the following actions: sends a 5$ coin (modeled by *giveFive* action) , receives a chocolate (modeled by *takeChocolate* action) and then continue to do the same (acts as User).

The VM (vending machine process) can receive a 5$ coin (*takeFive)*, then sends a chocolate (*giveChocolate)* and then acts like itself, or it can receive a 10& coin (*takeTen)*, sends an energy bar (*giveEnergyBar)* and then act like itself, or it can receive a 15$ coin (*takeFifteen),* sends a bottle of water (*giveWater)* and then acts like VM.

The *takeFive* and *giveFive* actions are complementary and they synchronize on channel *coinSync.* Also, *giveChocolate* and *takeChocolate* are complementary and they synchronize on channel *chocolateSync.* Those are specified under the *comm* token. Moreover, those synchronization actions are the only ones allowed, as the targeted process, namely User|| VM, exchanges information only on those 2 channels. This is specified under the *allow* token and can be seen in the graph below.

The graph:



→ Version 2:

The code:

Ex3V3

```
act
%VM can accept five, ten, fifteen on 'acc' channel, then gives an item and acts like VM
%User can insert 5 or 10 or 15 on 'ins' channel and, depending on the inserted coin, he takes chocolate, energyBar or water
    ins, acc: Nat;
    takeChocolate, giveItem, giveChocolate, giveEnergyBar, giveWater;
%Sync channels
    coinSync: Nat;
    chocolateSync;

proc
     User(n:Nat) = (n == 5) -> ins(n).takeChocolate.User(n);

     VM = sum n:Nat.( (n == 5) -> acc(n).giveChocolate.VM
                    <> (n == 10) -> acc(n).giveEnergyBar.VM
                    <> (n == 15) -> acc(n).giveWater.VM
                  );

init
           allow( {coinSync, chocolateSync},
                   comm( {ins|acc -> coinSync, takeChocolate|giveChocolate -> chocolateSync},
                          VM || User(5)
                       )
                );
```

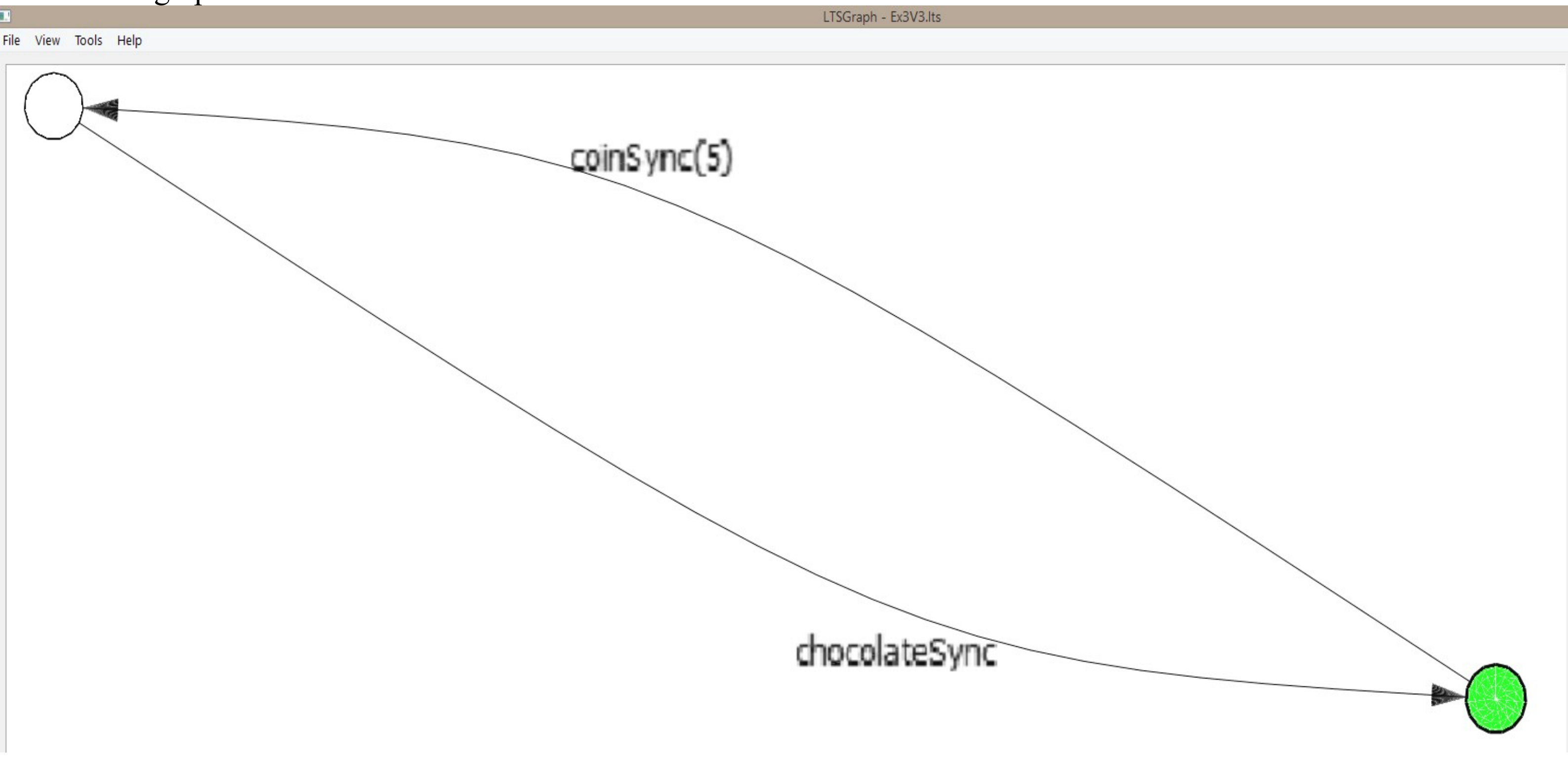
```
[12:52:48 verbose] type checking process specification...
[12:52:48 info] Specification is a valid mCRL2 specification
```

This code version is based on value passing. So, the User process is defined with a natural parameter, called *n,* he does action *ins(n)* only if the condition *n==5* is satisfied. This condition means that the User inserts a 5$ coin. Then, he receives a chocolate (*takeChocolate)* and continue to do the same thing, behaving as *User(n).*

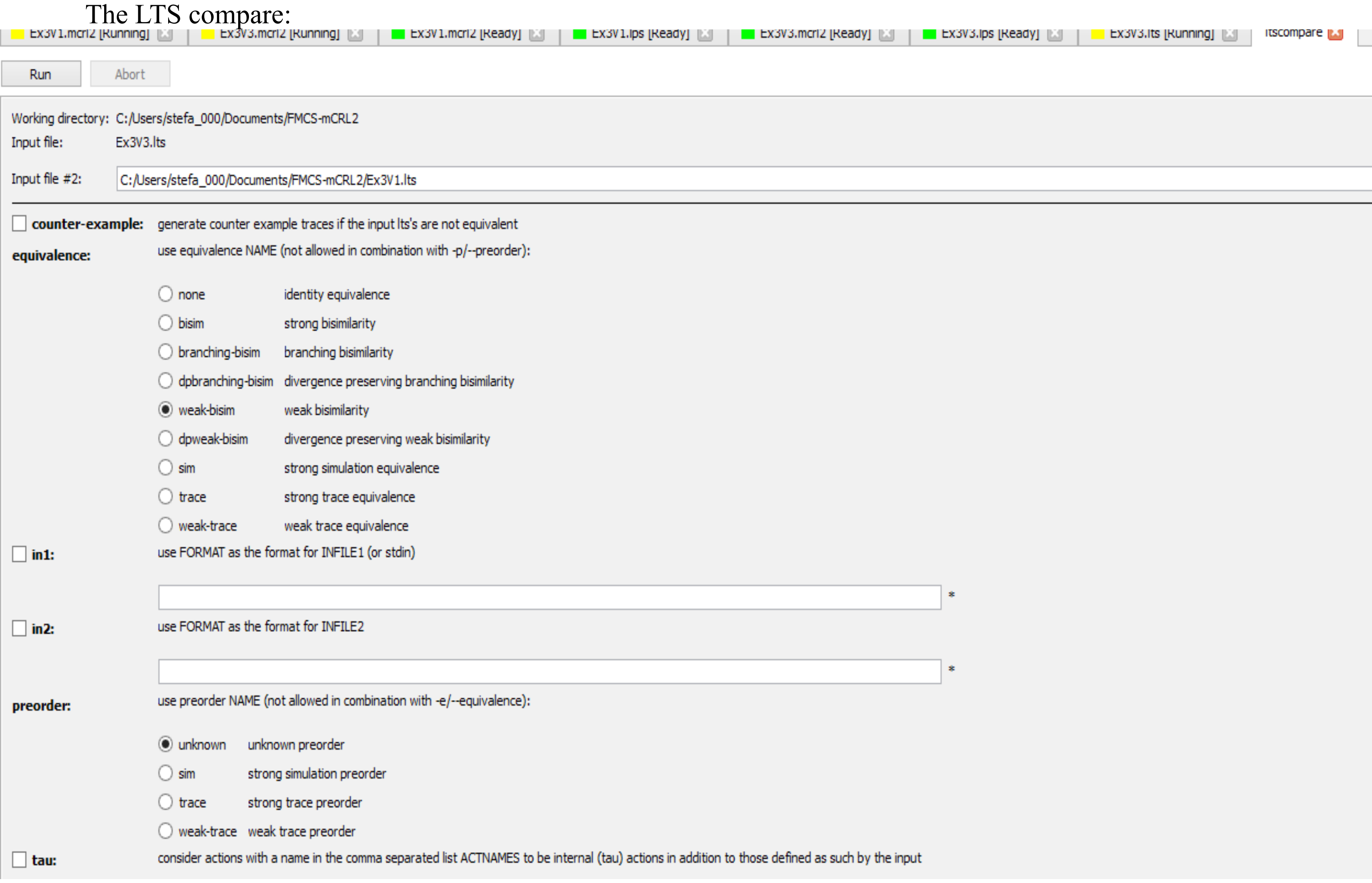
The VM process is modeled as a choice between the 3 cases described in the task for this exercise. It uses the *sum* token to specify that and uses the natural *n* to specify that it can receive natural numbers on *acc(n)* channel, which means coin receiving. Before accepting the coin, the VM verifies the coins type, then accepts it and then gives an item according to the inserted coin. In the end, it behaves like itself.

The *ins(n)* and *acc(n)* actions are complementary and they synchronize on *coinSync* channel, also defined as natural. Likewise, *giveChocolate* and *takeChocolate* are complementary and they synchronize on channel *chocolateSync.* Those are specified under the *comm* token. Moreover, those synchronization actions are the only ones allowed, as the targeted process, namely User(5) || VM, exchanges information only on those 2 channels.. This is specified under the *allow* token and can be seen in the graph below. In comparison to version 1, it is specified under the *init* section which type of coin is inserted by using *User(5)* call

The graph:



The LTS compare:



Ex3V1.mcrl2 [Running] | Ex3V3.mcrl2 [Running] | Ex3V1.mcrl2 [Ready] | Ex3V1.lps [Ready] | Ex3V3.mcrl2 [Ready] | Ex3V3.lps [Ready] | Ex3V3.lts [Running] | Ex3V3.lts [Error]

Run Abort [Error]

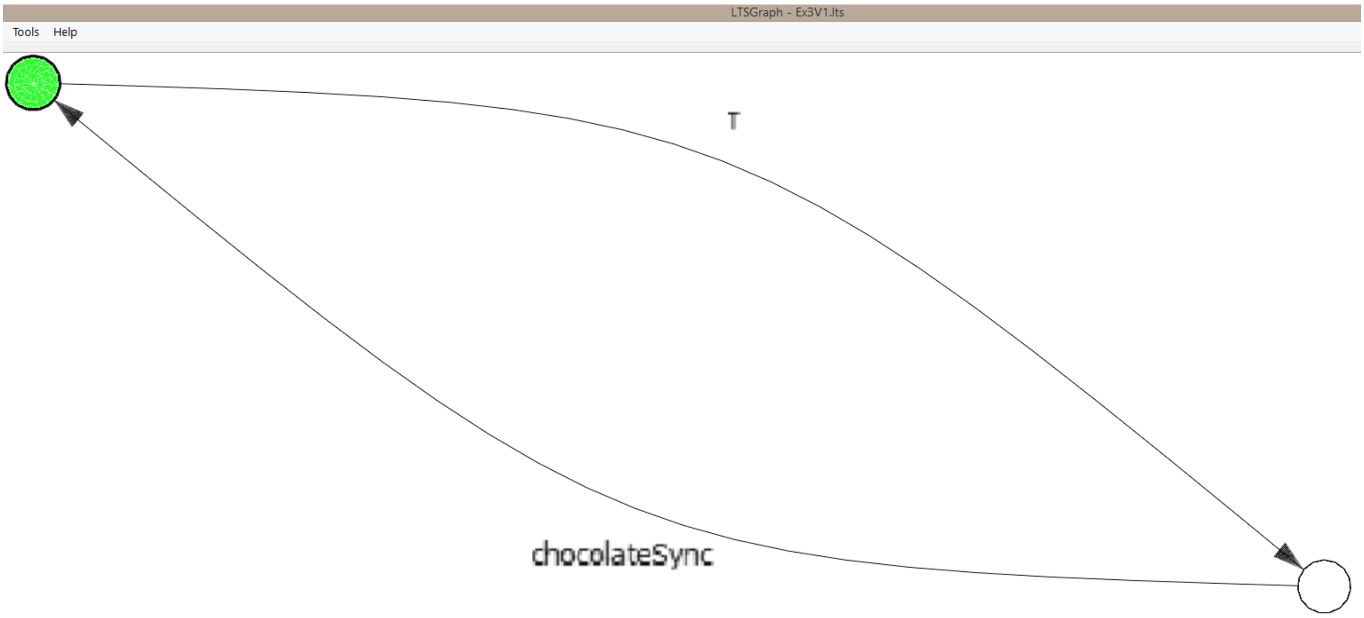
```
[13:25:38 verbose] Detected mCRL2 extension.
[13:25:38 verbose] Detected mCRL2 extension.
[13:25:38 verbose] Starting to load file Ex3V3.lts
[13:25:38 verbose] Starting to load file C:/Users/stefa_000/Documents/FMCS-mCRL2/Ex3V1.lts
[13:25:38 verbose] comparing LTSs using weak-bisim...
[13:25:38 verbose] Branching bisimulation partitioner created for 2 states and 2 transitions
[13:25:38 verbose] Bisimulation partitioner created for 2 states and 4 transitions
[13:25:38 verbose] Branching bisimulation partitioner created for 2 states and 2 transitions
[13:25:38 verbose] Bisimulation partitioner created for 1 states and 2 transitions
[13:25:38 verbose] Bisimulation partitioner created for 3 states and 3 transitions
[13:25:38 info]    LTSs are not weak bisimilar
```

The 2 systems are not weakly bisimilar as the actions *coinSync* and *coinSync(5)* don't match, because they are diferent type of actions. This can be observed, also, in the LTS graphs. But, if in both cases the actions for sending and receiving a coin are restricted, concretely putting *coinSync* and *coinSync(5)* under the *hide* token, they will be weakly bisimilar, as the the coin send-receive synchronization will be an internal action (tau), as it is presented below.

The code and the graph for version 1:

Ex3V1

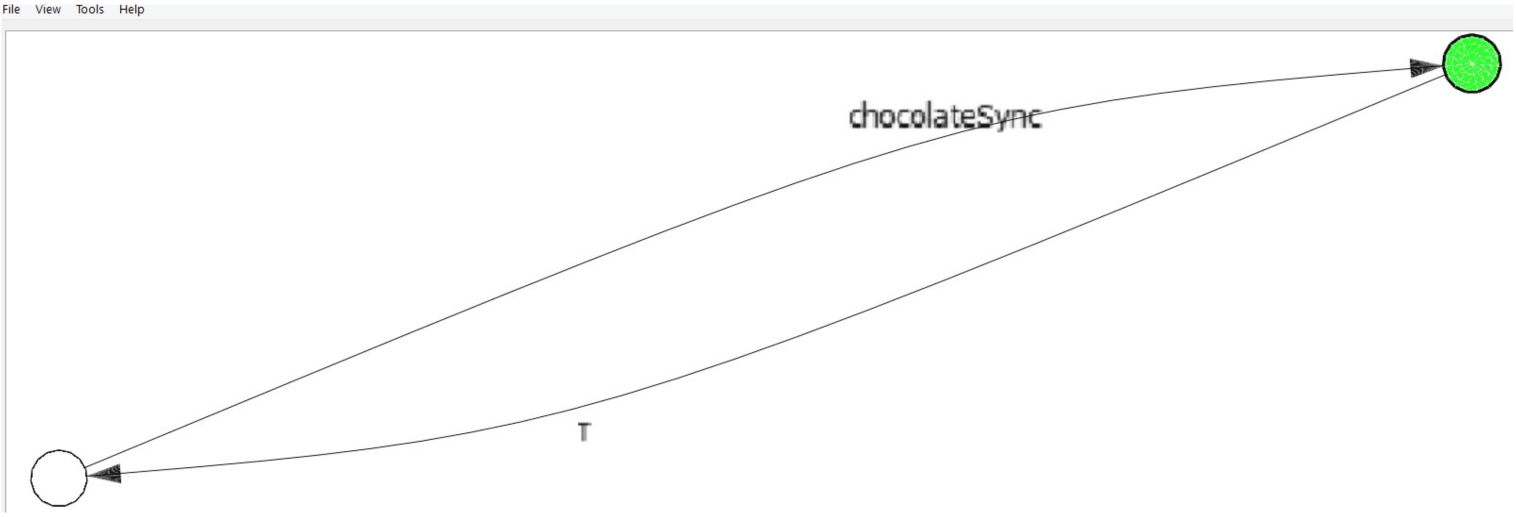
```
act
%VM can receive five, ten, fifteen and can send chocolate, energyBar and water
    takeFive, takeTen, takeFifteen, giveChocolate, giveEnergyBar, giveWater;
%User sends five and receives chocolate;
    giveFive, takeChocolate;
%Sync channels
    coinSync, chocolateSync;

proc
    User = giveFive.takeChocolate.User;
    VM = takeFive.giveChocolate.VM + takeTen.giveEnergyBar.VM + takeFifteen.giveWater.VM;

init
    hide( {coinSync},
            allow( {coinSync, chocolateSync},
                    comm( {takeFive|giveFive -> coinSync, giveChocolate|takeChocolate -> chocolateSync},
                            VM || User
                    )
            )
    );
```



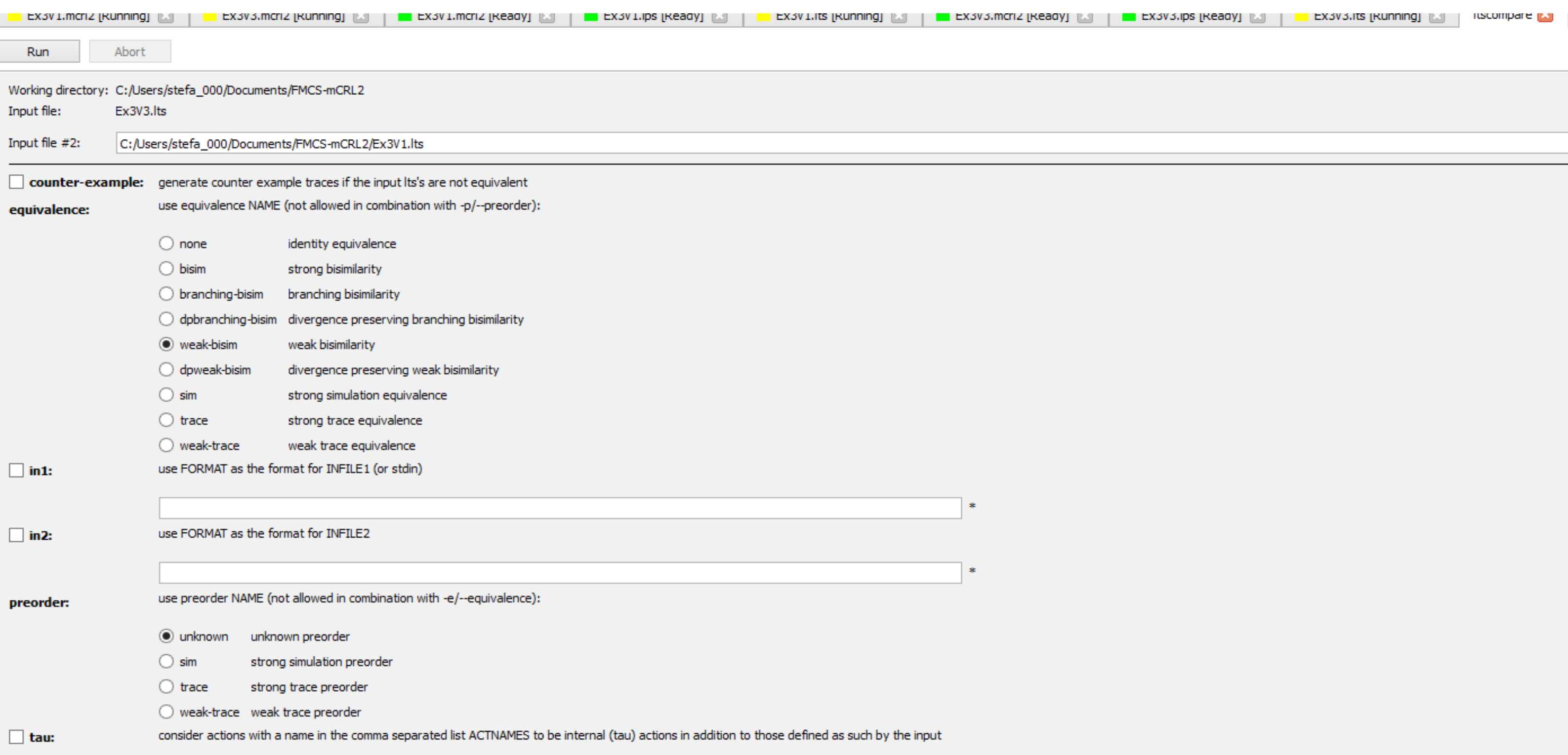
The code and the graph for version 2:

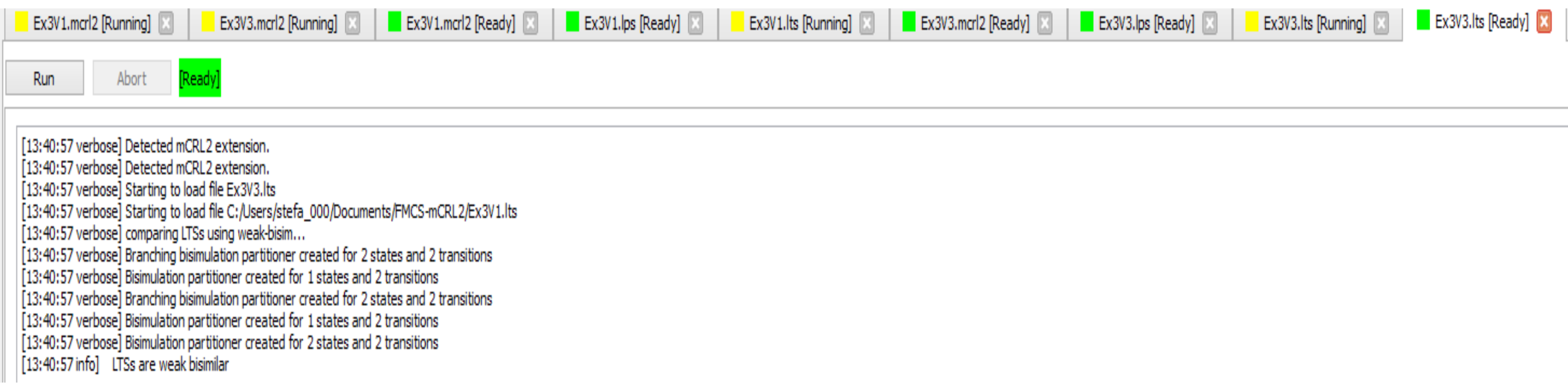
Ex3V3

```
act
%VM can accept five, ten, fifteen on 'acc' channel, then gives an item and acts like VM
%User can insert 5 or 10 or 15 on 'ins' channel and, depending on the inserted coin, he takes chocolate, energyBar or water
    ins, acc: Nat;
    takeChocolate, giveItem, giveChocolate, giveEnergyBar, giveWater;
%Sync channels
    coinSync: Nat;
    chocolateSync;

proc
     User(n:Nat) = (n == 5) -> ins(n).takeChocolate.User(n);

     VM = sum n:Nat.( (n == 5) -> acc(n).giveChocolate.VM
                    <> (n == 10) -> acc(n).giveEnergyBar.VM
                    <> (n == 15) -> acc(n).giveWater.VM
                  );

init
     hide( {coinSync},
           allow( {coinSync, chocolateSync},
                   comm( {ins|acc -> coinSync, takeChocolate|giveChocolate -> chocolateSync},
                          VM || User(5)
                       )
                )
          );
```



The LTS compare:





Exercise 4:

The code:

Ex4

```
%define actions
act
    send, send', sendSync:Nat;
    ack, ack', ackSync;
    error, error', errorSync;
    trans, trans', transSync:Nat;
    acc: Nat;
    del':Nat;

%define processes Send, Rec and Med, which will run in parallel
proc
%For a list received as a parameter for process Send, take the first element as long as list is not empty and try to send it.
%Thus, there will be send one at a time, in order.
    Send(phoneBook:List(Nat)) = (#phoneBook > 0 )-> acc(head(phoneBook)).Sending(phoneBook);
%Send the first element and wait for a signal. Each phone number is an element of the phoneBook and is considered a natural m
    Sending(phoneBook:List(Nat)) = send'(head(phoneBook)).Wait(phoneBook);
%If it is an error, retry to send it, else continue with the remaining list
    Wait(phoneBook:List(Nat)) = ack.Send(tail(phoneBook)) + error.Sending(phoneBook);

%Receiver gets the phone number, deletes request for that phone nr and sends feedback
    Rec(n:Nat) = trans(n).Del(n);
    Del(n:Nat) = del'(n).Ack(n);
    Ack(n:Nat) = ack'.Rec(n);

%Medium receives a natural number = phone number and tries to send it.
    Med = sum n:Nat.(( n <= 9999999) -> send(n).Med'(n));
%If there is an internal error try to handle it, else transmit the number to the receiver.
    Med'(n:Nat)= tau.Err + trans'(n).Med;
%Send an error signal to the sender.
    Err = error'.Med;

init
    hide( {sendSync, errorSync, transSync, ackSync},
          allow( {acc, del', sendSync, errorSync, transSync, ackSync},
                 comm( {send|send' -> sendSync, ack|ack' -> ackSync, error|error' -> errorSync, trans|trans' -> transSync},
                       Send([3453345, 2334123, 6564322, 4342312, 5432543, 5432555])|| Med ||
                            Rec(3453345)|| Rec(2334123) || Rec(6564322) || Rec(4342312) || Rec(5432543) || Rec(5432555)
                 )
          )
    );
```

```
[13:07:06 verbose] type checking process specification...
[13:07:06 info] Specification is a valid mCRL2 specification
```

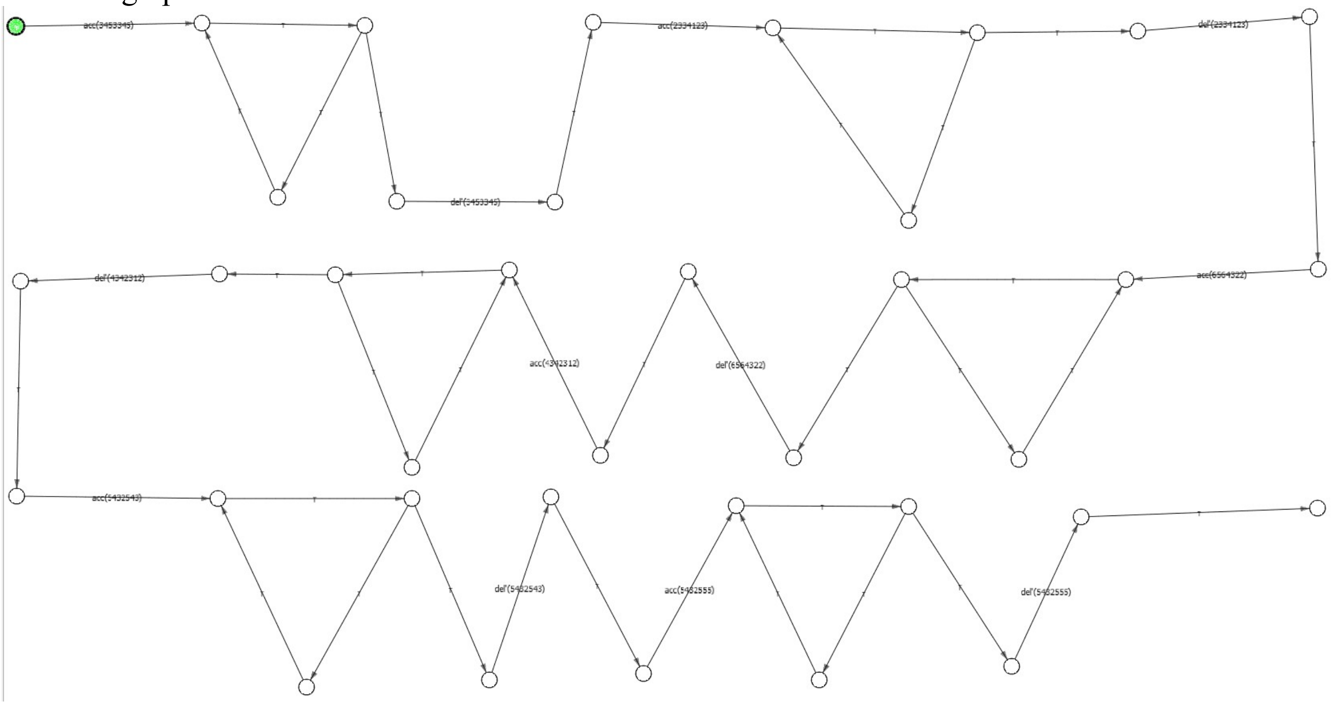
The *Send* process receives a list of phone numbers, it check if the list is empty and sends the first number, which is the head of the list, to the *Med* process. Then, it continues its behavior with the remaining elements in the list, namely the tail of the list. This process stops when the list is empty.

The *Med* process has the possibility of receiving natural numbers no bigger than 9 999 999, which is the largest number of 7 digits. The lower threshold is not verified as I want to let the possibility for transmit a phone number like this: 0 000 010 for example. Once received such a number, if an internal error occurs, it gets back to the state when waits for a number. Otherwise, it sends the number to the receiver, namely *Rec* process.

The *Rec* process receives a number from the *Med,* it deletes the request for that number and sends an acknowledge signal to the sender. It gets as a parameter a natural number which represents the phone number.

For each phone number, there is one *Rec* process running in parallel, as it implements the behavior of unpredictable data transmission. This means that, despite the sender transmits packets in order, they might not arrive in the same order at the receiver.

The LTS graph:



The LTS graph is pretty similar to the graph from exercise 2, just it repeats 6 times, one time for each number. Also, the last tau action doesn't get back to the initial state, it goes to another state where it is performed *acc* action for the next number in the list. Each sequence executes 5 tau's one *acc* and one *del'* for the respective number.