

Implementation design:

1. The Basic Translation of the Transition:

An example that will show how the transition will be translated to LOTOS, we will extend transition's translation to handle the synchronization.

```
MODULE COUNTER() {  
  
    VAR  
  
    input: boolean;  
    output: boolean;  
    current_state: integer;  
  
    TRANS GO_TO_NEXT_STATE:  
    enable:      ((input = true) /\ (current_state != 7));  
    assign:      current_state' := current_state + 1;  
    relation:    output' := false;  
}
```

```
process COUNTER [globals] (input:Bool,output:Bool,current_state:Integer) :noexit :=
```

```
// The operator [] (guard) used as pre-condition.
```

```
[ ( (input=true) and (current_state!=7) ) ]->
```

```
// The operator choice used to choose a random value from the specific range.
```

```
choice current_state_:Integer,output_:Bool []
```

```
// The operator [] (guard) used as post-condition condition.
```

```
[ ( (current_state_=current_state+1) and (output_=false) ) ]->
```

```
COUNTER [globals] (input,output_,current_state_)
```

```
endproc
```

The first coulomb describes a "component" and how this is translated to LOTOS is described in the second coulomb.

	Core_transition	Lotos_transition
1.	Enable ((input = true) /\ (current_state != 7))	Guard on the precondition [((input= true) and (current_state!=7))] Note: if the enable will not satisfy the execution of the process will not be continued
2.	Assign and Relation assign:current_state' := current_state+1; relation: output' := false;	Are combined to get a Guard on the post-condition [((current_state=current_state+1) and (output_ =false)]
3.	The variables with tag (e.g. output') in the post-condition.	The operator choice will choose a random value for these variables. choice current_state_:Integer,output_:Bool [] Note 1: The guard on the post condition will ensure that the tagged variable get the correct value. Note 2: If HOLD_PREVIOUS doesn't appear in the core program then we will choose a random value for all the parameters and variable in the module.
4.	Simulate the endless execution of the module In CORE.	Recursive call(e.g. COUNTER[global] (input...)

2. The Basic Translation of the Module:

Every module in the Core will be translated into a Process in LOTOS.

```

MODULE COUNTER() {
  TRANS SELF_LOOP:
    enable: input = false;
    assign: output' := false;

  TRANS GO_TO_NEXT_STATE:
    enable: ((input = true) /\ (current_state != 7));
    assign: current_state' := current_state + 1;
           output' := false;

  TRANS OUTPUT_IS_1:
    enable: ((input = true) /\ (current_state = 7));
    assign: current_state' := 0;
           output' := true;
}

```

The translation to the lotos will be:

```
process COUNTER [globals] :noexit :=

globals !"receive" ?output:Bool ?input:Bool ?current_state:Integer;
// Transition SELF_LOOP
( ( [input= false ]-> // guard operation: the pre-condition in CORE
choice output_:Bool []
[ output_= false]-> // guard operation: check that the post-condition of
// in CORE is statisfied.
globals !"send" !output_ !input !current_state; // output the variable's
// value on the shared
// gate.

COUNTER [globals] // simulate the continua's execution of CORE module.

[] // choose one of the LOTOS's transition.
// Transition GO_TO_NEXT_STATE
[(input= true) and (current_state!=7)]->
choice current_state_:Integer,output_:Bool []
[(current_state_= current_state+1) and (output_= false)]->
globals !"send" !output_ !input !current_state;
COUNTER [globals] )

[]
// Transition OUTPUT_IS_1
[(input= true) and (current_state= 7)]->
choice current_state_:Integer,output_:Bool []
[(current_state_= 0) and (output_= true)]->
globals !"send" !output_ !input !current_state;
COUNTER [globals] )

endproc
```

Notes:

1. The [] operator is used to choose one of the lotos_transition, which will be chooses if it's guard satisfies.
2. When the transition end, it recursively call the process to simulate a continues execution of a module in Core

3. The shared data in LOTOS(*):

Because there is no shared data in LOTOS where in CORE there are GLOBAL variables, we add a process "Global".

This process will synchronize on "globals" gate with the process "System"(which is equivalent to the Module System in the Core), and before any transition (in the LOTOS program) is executed it will receive the values from the process "Global" and will send the new values to it in the end of the transition, these actions will occurred on the gate "globals".

Example for "Global" process's code:

```
process Global [globals] (output:Bool,input:Bool,current_state:Integer)
:noexit :=

  // send the globals values
  globals !"send" !output !input !current_state;
  Global [globals] (output,input,current_state)
  []

  // recieve the globals values
  globals !"receive" ?output_:Bool ?input_:Bool ?current_state_:Integer;
  Global [globals] (output_,input_,current_state_)

endproc
```

The code that needed to be adding in the Transition is (highlighted in yellow):

// before the transition is executed it receive the global variables values from "Global" process.

```
globals !"send" ?output:Bool ?input:Bool ?current_state:Integer;
( ( [ (input= false) ]->
choice output_:Bool []
[ (output_= false) ]->
```

// now send to the "Global" process the new global values.

```
globals !"receive" !output_ !input !current_state_;
```

Notes:

- The **specification _SYSTEM** will instantiate this process, and will pass the initial values of the global variables to it (if there is no initial value, it will choose a random number from the range of the global variable).
- In order to identify that the "Global" process will send values to the transition or should it receive values, we use "Value matching" of "send" and "receive".
- Receiving the globals values from the "Global process" performed in the beginning of the process while sending the globals values are performed in the end of each transition.
- `globals !"receive" !output_ !input !current_state_;` will also be used to handle Full Synchronization (look at 8.1).

(*) Based on the article:

"Even, S., Fasse, F. : MERGE OPTIONS FOR LOTOS AND TM, October 7, 1994."
wwwhome.cs.utwente.nl/~transcoo/Del41.ps

4. The local variables in a module:

If there is a local variable in the module then they will be translated to parameters in the process. If these variables have an initial value then the process that instantiate it has to pass these values to it else the process will choose random values for these variables before it instantiate the other process.

5. Constants variable declaration:

If the core program defines a constant variable, then this constant value will be passed as a parameter to every process in the LOTOS program.

The constants variable will be initialized by the specification `_SYSTEM` which will pass the constants values to the process `SYSTEM` (according to their values in the `CORE` program) .

6. Handling CoJoin:

When a `COJOIN` condition is define to a process then we translate it to a Guard on the process, in this way the processor will not execute this process until the guard satisfies.

7. Handling Some types:

Range: In the program we are casting this type to integer.

Enum: Every enumerated value were passed as parameter to all the process as integer.

8. Translation problems:

Now we describe some of the problems and our solutions, and how this will affect the translation of the process/transition.

1. Full synchronization:

If we want to synchronize two processes in the LOTOS, then they should meet at a gate.

So in order to make two modules in Core synchronize on two transitions, then the translation of the transition should include an action on a gate.

In our implementation these `lotos_transitions` synchronize by "value matching" (on all the global variable in the program) on a gate, this should be in order that the two process when they choose a random value for a global variable then they choose the same value.

The "global" gate that was used to support shared data, is also used for this purpose.

Core	LOTOS
<code>M1 M2</code>	<code>M1[globals] globals M2[globals]</code>

The code that used for that in the transition is (highlighted in yellow):

```
[input= false ]->
choice output_:Bool []
[ output_= false]->

// it used to synchronized with another process in order to ensure that
// both processes choose the same values. Notice that it also used to pass
// the global values to the "Global process"
globals !"send" !output_ !input !current_state;
```

2. Interleaving:

If there are two interleaving modules, the question is how one of the processes passes the current values of the global variables to the other process, this problem appears also in partial synchronization.

The solution is presented in the topic "**3.The shared data in LOTOS**".

3. The partial synchronization:

This example will describe how we handle partial synchronization in the translations.

Let's assume that

$M1 \mid (a1,b1),(a2,b2),(a3,b2) \mid M2$

Then in order to translate this program in LOTOS we use a new process $M1_M2$

And every transition will use another gate for partial synchronization (in addition to "global" gate), this gate will have the same name of the transition.

- The code that will be added in the lotos_transition:
The transition's name in Core is ENTER_CRITIC2.
[can_change_shared = true]->
choice can_change_shared_:Bool,shared_:Integer []
[((shared_ = shared-1) and (can_change_shared_ = false)) = true]->
ENTER_CRITIC2 !x !shared_ !z !y;
globals !"send" !x !shared_ !z !y;
B [ENTER_CRITIC2,EXIT_CRITIC2,NON_CRITIC2,globals] (can_change_shared_)
- The $M1_M2$ will synchronize with $M1$ on gates {a1,a2,a3,globals}
And synchronize with $M2$ on gates {b1,b2}.
The definition of $M1_M2$ will be:
 - a1!var_glab1!var_global2....;b1! !var_glab1!var_global2....;globals!var_global1..
[]
 - a2!var_glab1!var_global2....;b2! !var_glab1!var_global2....;globals!var_global1..
[]
 - a3!var_glab1!var_global2....;b2! !var_glab1!var_global2....;globals!var_global1..
[]
 - globals!var_global1..

Explanation:

There are three possibilities:

1. When $M2$ executes transition $b2$ and because it synchronizes with $M1_M2$ on this gate (gate $b1$), then before $M2$ reaches the action on gate $b1$, in $M1_M2$ should occur an action on gate $a2$ or gate $a3$, let's assume that occurred an action on gate $a3$, but because $M1_M2$ is synchronized with $M1$ $a3$ then they reach the action gate $a3$ together.

2. When M1 execute transition "a1" and when it reach to the action on gate "a1" then M1_M2 also should reach to the gate "a1", now if M1_M2 continue to execute then really the transitions a1 and b1 is synchronizing, but if M1 continue then it will do an action on "global" gate but it also synchronized on this gate with M1_M2 → In this case also the two processes will synchronized in gate a1 and b1.
3. When M1_M2 execute a1, this case is similar to case 2.

Note: The last action on "global" gate in M1_M2 is to handle the situation when a transition in M1 is executed and this transition is not synchronized with any transition in M2.

In general the when we have:

$M1 \mid \text{pair1, pair2, \dots} \mid M2$

Then we define a new process M1_M2 that will synchronized with M1 on all the transitions names that appear in the left side of the pairs(and also with **globals**), and will synchronized with M2 on all the transition names that appear on the right side.

For example: if one of the pairs was $((a1, b1), c1), (d1, e1)$, then M1_M2 will synchronize with M1 on gates $\{a1, b1, c1\}$ and with M2 on $\{d1, e1\}$.

Also action order on the gate is the same as their order in the pairs, for example in the previous example the action would be: $a1!..; b1!.., c1!.., e1!..$

9. What we did not implement:

1. The “renaming” in the core.
2. We assume that there is conflict in the transition’s name in the processes.
Example:
Core: $M \mid (t1, t1) \mid M$
LOTOS: In this case we doesn’t handled. This is related to the implementation of the new help Process that we generate for the partial.
3. We assume that the types that we used “Integer”, “Boolean” are defined in the system and all the operation that used in the Core also defined in the LOTOS.
4. We don’t handle “Define” in the core.

10. Improvement:

1. If there is no global variable in the system then there is no need to the Process "Global".
Because the only use for the “Global” process is to solve the Shared Data in Lotos.
The Needed Change:
 - There will not be a process “Global”
 - On every process there was an action on “globals” gate (in order to receive the values of the globals), in this case we needn’t this action.
 - We need to handle the partial synchronization and the global synchronization, because in this case we don’t need to output the global values on the “globals” gate, and we needn’t to output the global gates on the gate that used for partial synchronization.

```

[can_change_shared = true]->
choice can_change_shared_:Bool,shared_:Integer []
[ ( (shared_ = shared-1) and (can_change_shared_ = false) ) = true]->
ENTER_CRITIC2 !x !shared_ !z !y;
globals !"send" !x !shared_ !z !y;
B [ENTER_CRITIC2,EXIT_CRITIC2,NON_CRITIC2,globals] (can_change_shared_) )

```

one solution is to output on the gates {"ENTER_CRITIC2","globals"} a known value (ex. 0), notice in the FULL LOTOS we couldn't use a gate without an action on it (input or output).

2. In our current implementation of the assignment we change it to a relation, but this behavior is not necessary, because we could pass the expr of the assignment to the process it self or to the global process.
 In this case we don't have to choose a random value for this variable and then check that this variable satisfy the relation(which was really assignment in core)