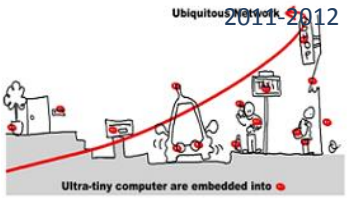
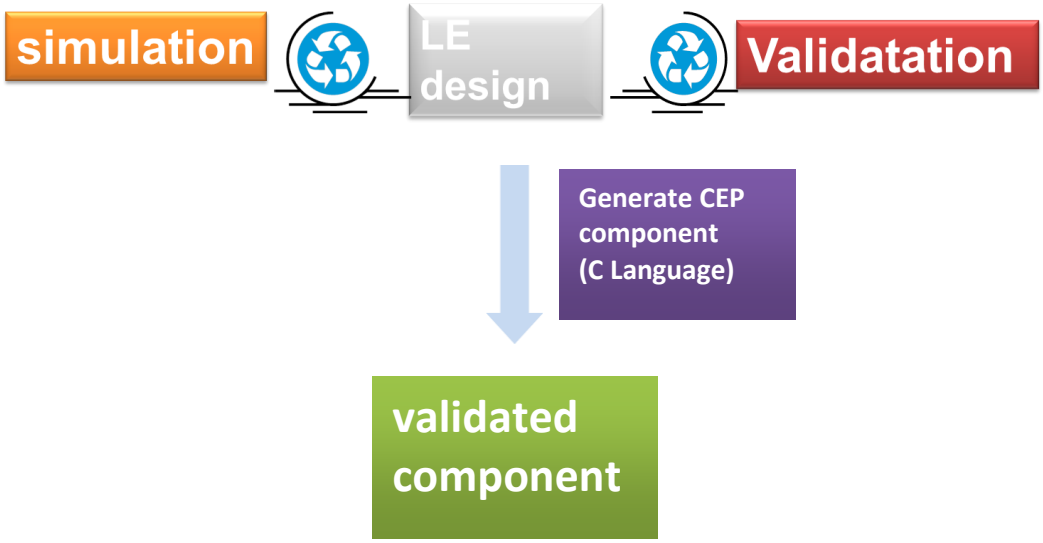
TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH



2011-2012

INTRODUCTION

The purpose of this tutorial is to illustrate the lecture of synchronous language for creating a validated CEP node. The main goal is to design a validated CEP (Complex Event Processing) component and try to communicate it with other components using the MQTT approach. Here is the main scheme to design our CEP:



In this tutorial, we will consider the design of a road_information application that informs the user about which road is better to take, according to his chosen destination and updated information about each road (accident, roadworks, traffic jam, etc...) .

According to our validation concern, we will follow the methodology detailed in the lecture. Thus, we will describe a **synchronous component** to ensure the correct behavior of our road_information manager and then introduce it as a validated Complex Event Processing (CEP).

To design this component and check its behaviors for correctness, we will use the CLEM toolkit. We will first study the CLEM toolkit which allows us to design and validate synchronous components.

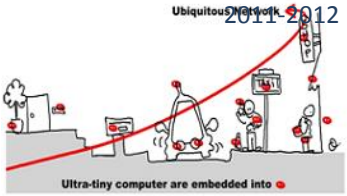
THE CLEM TOOLKIT

The CLEM toolkit is a set of tools around the LE synchronous language. It allows specifying constraint component and to generate automatically its code.

First of all, CLEM implements a modular compilation of LE programs, CLEM offers a simulation means and generates output code for hardware targets as well as software ones. In particular, CLEM generates **blif** code, **C** code, **C#** code, **SMV**

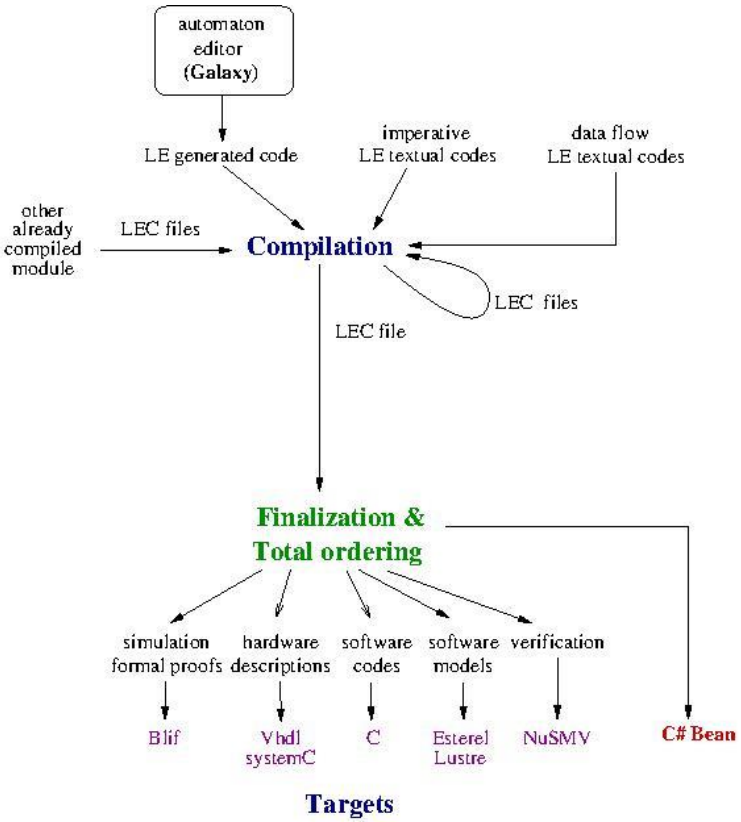
1. BLIF: this format is the entry format for simulation and verification tool (respectively blif_simul and blif_check or Xeve). The simulator for pure simulation involved in CLEM generates automatically this format before calling blif_simul. However, the verification tool is not integrated in CLEM.

annie.ressouche@inria.fr, ines.sarray@inria.fr



2. SMV: this format allows to enter NuSMV model checker in order to validate CTL properties.

3. C code defines to functions to run and reset the automaton model of a program and computed by the compiler.



THE CLEM TOOLKIT

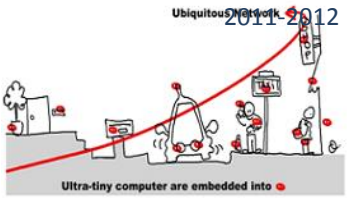
The following software helps us to handle a complete application:

- CLEM: main software of CLEM toolkit. It is LE program compiler. It takes as input LE modules (defined in a .le file) and compiles them and generates an internal format (LEC). Indeed this format is a concise representation of equation systems and Mealy machines, models of LE programs. CLEM also generates code for different targets, in particular the BLIF format suited to simulate LE module behavior, SMV format and C code to plug component in MQTT design.

- blif_simul: is the simulator. It is called with the simulation option of CLEM.

- Xeve: is the tool that performs model checking of LE module expressed in BLIF format.

- NuSMV is the model checker performing CTL formulas validation for LE module expressed in SMV format.



To use this software, put it in a "bin" folder and add to the **PATH** environment variable, the path to reach this bin folder:

```
export PATH = $PATH:/user/toto/bin
```

Assuming that the different software are in: /user/toto/bin

As LE language is useful to describe the synchronous components, we start by introducing LE.

THE LE LANGUAGE

LE offers 3 kinds of design:

1. An imperative language with particular synchronous operators
2. Explicit Mealy machine described as automaton
3. Implicit Mealy machine defined by Boolean equation systems

THE IMPERATIVE LANGUAGE

The language unit is the module, thus to define a program the syntax is the following:

```
module WIEO:

 end
```

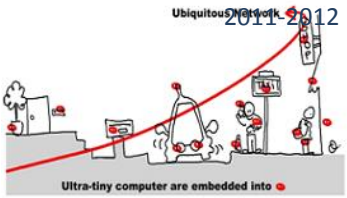
Each module falls into two parts: a declaration part and an instruction part. The declaration part defines the input and output signals, the module handles and also the declaration of its sub modules:

```
module WIEO:
Input: I;
Output:O1,O2;

end
```

The body of a module is a LE statement built with the help of the synchronous operators of LE. Each operator has semantics to explain formally their behavior. In particular, the semantics of an operator must tell us if it terminates in the instant or not, because its behavior depends of this information. We detail the main operators and give an intuitive description of their semantics:

- **nothing:** empty instruction which do nothing and takes no time (instantaneous). However it terminates in the current instant.
- **halt**: stops forever the evaluation
- **emit S** : set the signal **S** present in the environment. It also terminates in the instant.
- **present S {P1} else P2** : if **S** is present then **P1** is executed otherwise **P2** is.
- **P1 || P2**: synchronous parallel operator, it terminates when both of its arguments have terminated.

annie.ressouche@inria.fr, ines.sarray@inria.fr



- **P1 >> P2:** sequence operation, **P2** is executed when P1 is terminated. For instance , emit S1 >> emit S2 is instantaneous because emit is instantaneous and S1 and S2 are set present in the environment simultaneously.

- **local S1,S2, …. {P}**: signals **S1**, **S2**, …. are local in **P**. Local signals are set to undefined before the execution of **P** and their status is refined during this execution to present or always undefined. These signals are useful to allow the communication between the two arguments of a parallel.

- **loop {P}**: executes indefinitely **P**. This latter must last at least one instant. When **P** has been executed, it is started again instantaneously.

- **wait S**: stops until **S** is present. However, the presence of S is not tested in the first instant, then this instruction is not instantaneous but last at least one instant. For instance, if we consider the instruction wait I >> emit O; even if I is present in the first instant, O is not emitted. Otherwise, if I is present in the second instant (or in a next one), then O is emitted.

- **pause**: do nothing but takes one instant. Mainly use to force the duration of an instruction. For instance, if we consider the instruction pause>> emit O, the signal is emitted in the second instant of the execution of the instruction.

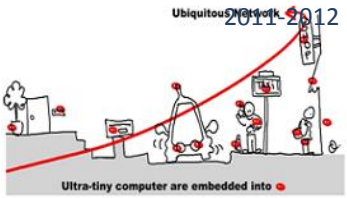
- **weak abort {P} when S**: executes P and stops when S is present and terminates the evaluation of the current instant. However, the preemption signal is not listen in the first instant. In the following example: **weak abort {pause >> emit** O1 **>> emit** O2 **>> pause >> emit** O3**} when** S, if S is present in the first instant, the evaluation continues (S is not listen), if S is present in the second instant O1 and O2 are emitted and the evaluation of the instruction is terminated. Otherwise, at the third instant, O3 is emitted and the normal evaluation is over.

- **strong abort {P} when S** : behaves similarly as weak abort except that the evaluation does not terminates the current instant when the preemption signal is present. For instance, in the previous example, if S is present in the second instant the evaluation of the instruction terminates without emitting O1 and O2.

- **run mod** : to call the module **mod**. This instruction put into practice the modular compilation of CLEM. You can compile the module **mod** and save the LEC code generated in a **file mod.lec** and then reload this code when compiling a main module containing this **run mod** instruction. If the module **mod** is not already compiled, CLEM compiler will do the job. However, a declaration:
  Run:
  "path": mod-file: mod;
  is required in the main module specification: path is the path to the file that contains the module **mod**, mod-file is the name of the file containing mod (here, we assume that its name is mod-file.le) and finally**, mod** is the name of the called module.

```
module WIEO:
Input: I;
Output:O1,O2;
```


annie.ressouche@inria.fr, ines.sarray@inria.fr

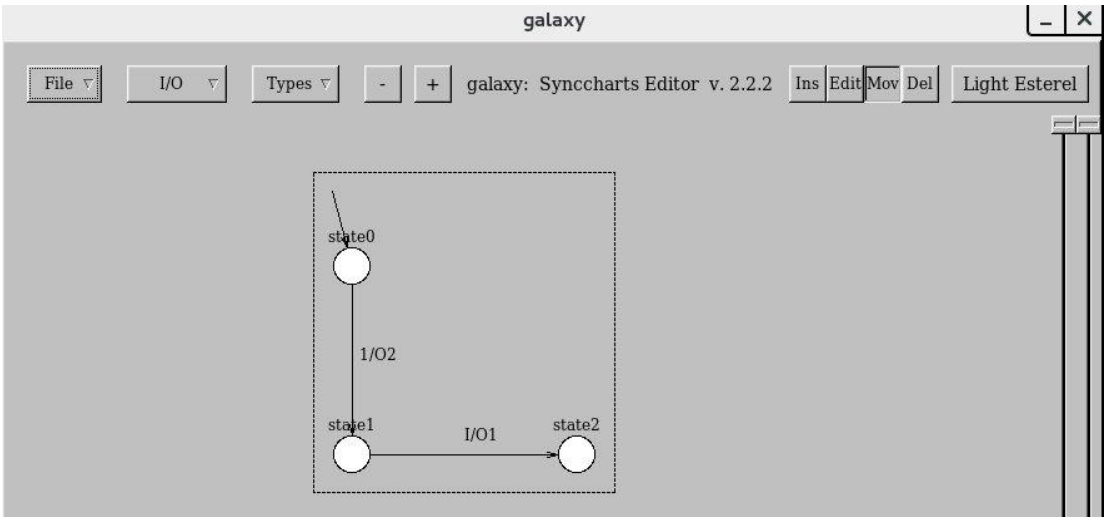



```
wait I >> emit O1
||
emit O2

end
```

In this example, O2 is first emitted in the first instant and then I is tested at each instant as soon as it is present, then O1 is emitted and the execution is over; after whatever are the signals in the environment, nothing will append.

EXPLICIT MEALY MACHINE

To design explicit Mealy machine, we rely on a gui (GALAXY) which allows to specify hierarchical and parallel Mealy automata. It is a general tool to design different kinds of automata. In particular, it offers a light esterel mode (light esterel stands for LE) devoted to design LE explicit Mealy machine. It has a unique view and is very simple to use. Here is the window you get:



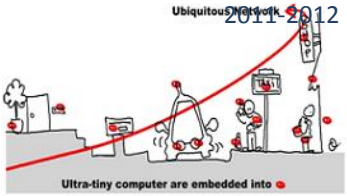
GALAXY DESIGN FOR THE PREVIOUS WIEO EXAMPLE

To define WIEO automaton, GALAXY offers the following menus:

File menu

In this menu, you have the following choices:

1. New: to create an automaton according to a selected model. You can choose between: basic automaton, parallel automata, light esterel, synccharts. For our purpose, we choose light_esterel.
2. Name: this field must be assigned, it is the name of the project and is mandatory to save, load, ect. Here the name is WIEO.
3. Save: save the automaton into two formats: WIEO.gal which is an internal format to load again the automaton in GALAXY, and WIEO.le an internal LE format for automaton, to be loaded in CLEM. This saving operation relies on the name given to the project.



4. Load: to load a ".gal" file
5. Export: to export in an "xfig" format allowing the integration of automata design in document.
6. Quit.

I/O menu

This menu allows defining the inputs, the outputs and the modules you can call in state. To define such modules a **Run** item asks you for the name of the module, the name of the file where is defined the module, and the path to reach this file (you can use the Browse facility). These declarations will be attached in the saved ".le" file.

Design menu

At the right upper part of the window, you find the "design" buttons:

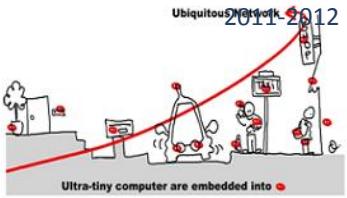
1. **Ins**: to define states and transitions. A mouse click creates a state and a mouse drag from a state to another draws a transition. A click in a state draws a circular transition from and to this state. To define the initial state , just drag a transition from the background to the state.
2. **Edit**: to complete the drawn states and transitions. In this mode, if you click on:
    a. a state, you can define a name for the state, the run module which be called in this state (previously defined with the I/O menu) and also some actions. These latter are output signals emitted all the instants you stay in this state during an evaluation.
    b. a transition, you define its triggering condition (Condition) and the emitted signals (Actions). The trigger part is a Boolean expression built from the inputs defined in the model, and actions are the outputs of the model.
3. **Move** and **Del** are drawing facilities to move and delete.

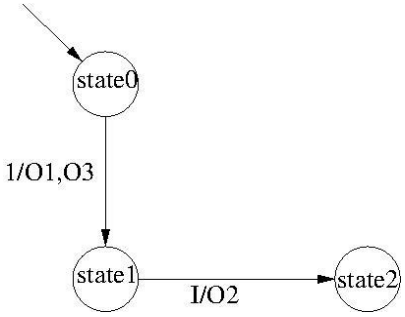
In light esterel modelling, all unrelated states are in parallel. A dashed line shows the part of the design which are in parallel.

IMPLICIT MEALY MACHINE

This last kind of model offered by LE allows defining Mealy machine by a set of registers and Boolean equation systems. Registers are particular entities which are represented by two Boolean variables to handle the current value and the next value. In such a model, states are encoded by the valuation of registers and thus the next value of registers is the computation of the next state in the equation system. Hence, equations compute the next values of registers and the values of output signals. For instance, considering the following explicit automaton:

annie.ressouche@inria.fr, ines.sarray@inria.fr





The corresponding implicit Mealy machine in LE, will be defined as :

```
module Parallel:
Input:I;
Output: O1, O2,O3;

Mealy Machine
Register:
X0: 0: X0next;
X1: 0 : X1next;

X0next = X0 and not X1;
X1next = X0 and X1 or not X1 and Ior not X0 and X1;

O1 = not X0 and not X1;
O2 = X0 and not X1 and I;
O3 = not X0 and not X1;

end
```

Notice that some implicit Mealy machines have no register. They just describe an equation system and are called "combinatorial".

SIMULATION

CLEM offers two ways to simulate a design:

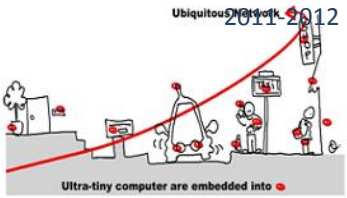
1. A pure simulator (button "start pure simulator")
2. A simulator taking into account valued signals (button "start valued simulator cles")

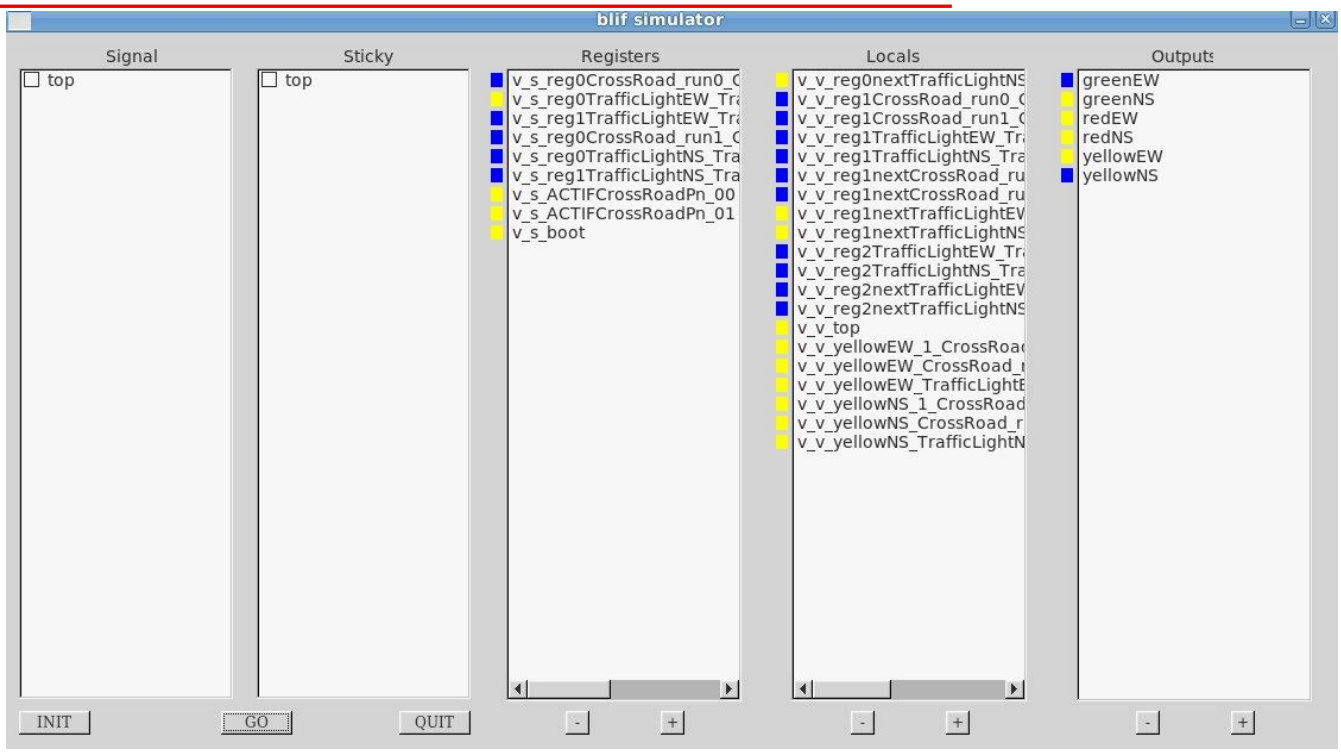
In this lab, we will use the pure simulator.

Here is the simulation of the CrossRoad example introduced in the lecture:

annie.ressouche@inria.fr, ines.sarray@inria.fr

TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH





You can set input events present in selecting them in the left column. You launch the computation of the current instant with the GO button. Resulting outputs are in the right column. Colors have the following I meaning: red means undefined, yellow means absent and blue means present.

VERIFICATION

As already said the simulation feature of CLEM automatically calls the blif_simul software. To achieve the verification phase of a design, CLEM offers two ways: either rely on the Xeve model checker or on the NuSMV one.

To run the **Xeve** model checker, you must first generate the BLIF code of your design with the BLIF button of CLEM.

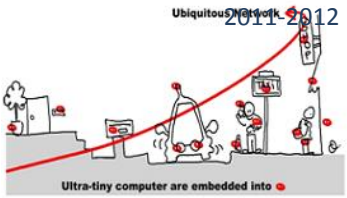
1. load the BLIF file of your model
2. Choose an output you want to check
3. Choose the verification you want: always true or false; exists with value true or false. If the property fails, a counter example is generated. This counter example shows a path leading to a state where the property fails.

This model checker checks for the presence or the absence of an output signal. Thus, it must be used with the observer technique, if you want to check a more complex safety property.

To run the **NuSMV** model checker, I you must first generate the SMV code with the SMV button and then call the NuSMV model checker. NuSMV can also be used with the observer technique. Pratically, check if the NuSMV software in the archive runs on your linux system. If not, you must install it from NuSMV.2.6.0.tgz code in the archive or from the website: http://nusmv.fbk.eu/.

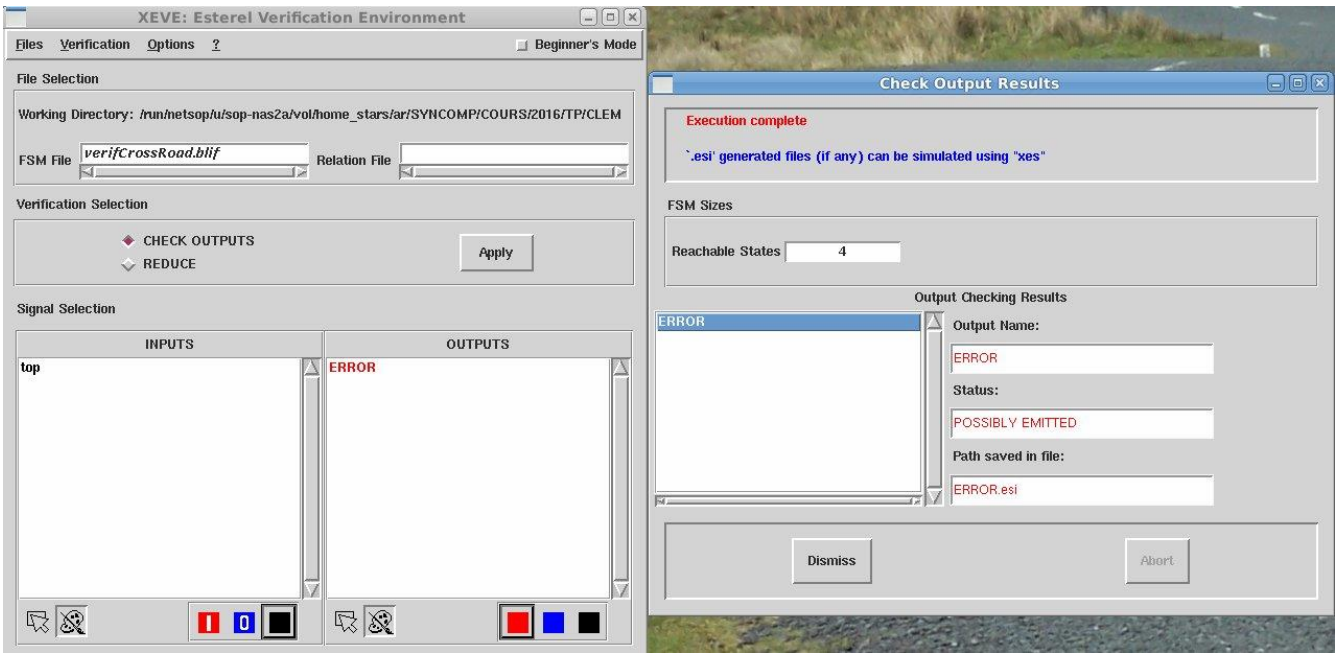
annie.ressouche@inria.fr, ines.sarray@inria.fr

TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH



Continuing with the CrossRoad example, we consider the observer defined in the lecture which emits a failure when the CRossRoad emits simultaneously greenNS and yellowEW or greenEW and yellowNS. According to the observer technique, we built a new module: verifCrossRoad which is the parallel composition of the CrossRoad module and the observer one (also see the Lecture). The goal of the verification phase is to prove that failure is never emitted.

First, we will use Xeve. With CLEM we generate the file verifCrossRoad.blif and we call Xeve:

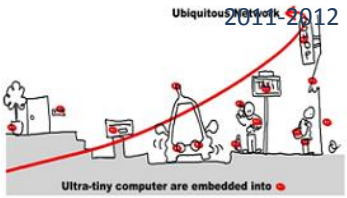


First, we load the module verifCrossRoad.blif. Then, we check that ERROR is never emitted. The result is on the right window. There is a failure because the signal ERROR can be emitted and we get an error and the counter example is generated in a file ERROR.esi

Second, we use NuSMV. With CLEM we first generate the file verifCrossRoad.smv and we call NuSMV in interactive mode

```
[ar@golazzina CLEM]$ NuSMV -int

*** This is NuSMV 2.6.0 (compiled on Thu Dec 22 13:54:55 2016)

*** Enabled addons are: compass

*** For more information on NuSMV see <http://nusmv.fbk.eu>

*** or email to <nusmv-users@list.fbk.eu>.

*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>
```


annie.ressouche@inria.fr, ines.sarray@inria.fr




```
*** Copyright (c) 2010-2014, Fondazione Bruno Kessler


*** This version of NuSMV is linked to the CUDD library version 2.4.1

*** Copyright (c) 1995-2004, Regents of the University of Colorado


*** This version of NuSMV is linked to the MiniSat SAT solver.

*** See http://minisat.se/MiniSat.html

*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson

*** Copyright (c) 2007-2010, Niklas Sorensson


NuSMV > read_model -i verifCrossRoad.smv

NuSMV > flatten_hierarchy

NuSMV > encode_variables

NuSMV > build_model

NuSMV > check_ctlspec  -p "AG !mverifCrossRoad.failure"

-- specification AG !mverifCrossRoad.failure  is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample
```

PRACTICAL ISSUES TO INSTALL NUSMV AND XEVE

First, to install **xeve**: download the archive xeve.tgz:

1. extract the files (tar xvzf archive.tgz).
2. In esterelv5_92.linux/bin folder edit the xeve file and change the ESTEREL variable definition to the exact path leading to esterelv5_92.linux folder in your own environment.
3. Copy xeve in your “bin” folder.

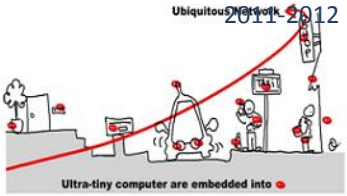
Second, to install **NuSMV**: download the NuSMV-2.6.0.tgz archive:

1. extract the files (tar xvzf NuSMV-2.6.0.tgz) .
2. follow the instructions from README.txt.
3. Copy NuSMV in your “bin” folder.

EXERCISE

annie.ressouche@inria.fr, ines.sarray@inria.fr

TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH



As a training exercise, define in CLEM a module tictac that emits tic the first instant and tac the second instant and indefinitely do it again. Put the design of the module in a "tictac.le" file and simulate it with the simul feature of clem. Then, implement the same module as an explicit Mealy machine and name it tictac1 (for instance), save it. You will get a tictac1.le file and simulate it in clem.

DESIGNING A ROAD_INFORMATION COMPONENT WITH CLEM

SPECIFICATION

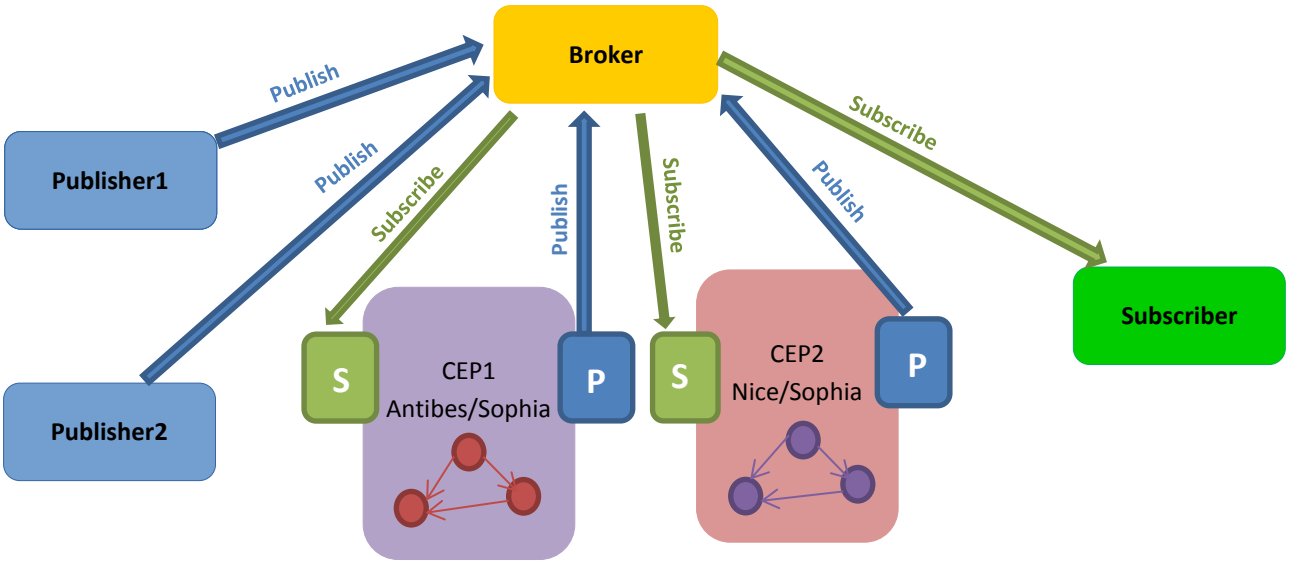
This road_information component contains information about the road map of Sophia, Antibes and Nice.

To simplify the work, we will present four propositions: Sophia_Antibes, Antibes_Sophia, Sophia_Nice and Nice_Sophia.

For each proposition we have two possible roads to take, according to the choice of the user (example : Sophia_Nice), the component will look for information about the two roads related to this choice and will suggest the best and fastest road to take, depending on the information that it receives from the broker.

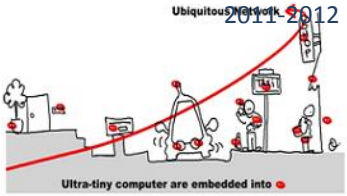
In this tutorial we will create two CEPs: One for Sophia/Nice region that will contain the information of the four roads related to its two propositions (Sophia_Nice, Nice_Sophia), and the other one is about Sophia/Antibes region.

Each CEP will subscribe to the broker to receive the needed information. If the user chooses one of its proposition (example Nice_Sophia), the CEP will treat the received information about the roads from the broker, make its processing and publish the suggestion to the broker again, so that the user can see which road is better to take.



Our component will make its choice depending on some specific input information:

annie.ressouche@inria.fr, ines.sarray@inria.fr



- Presence of an accident
- Roadworks
- Number of cars (>20)
- Road closed

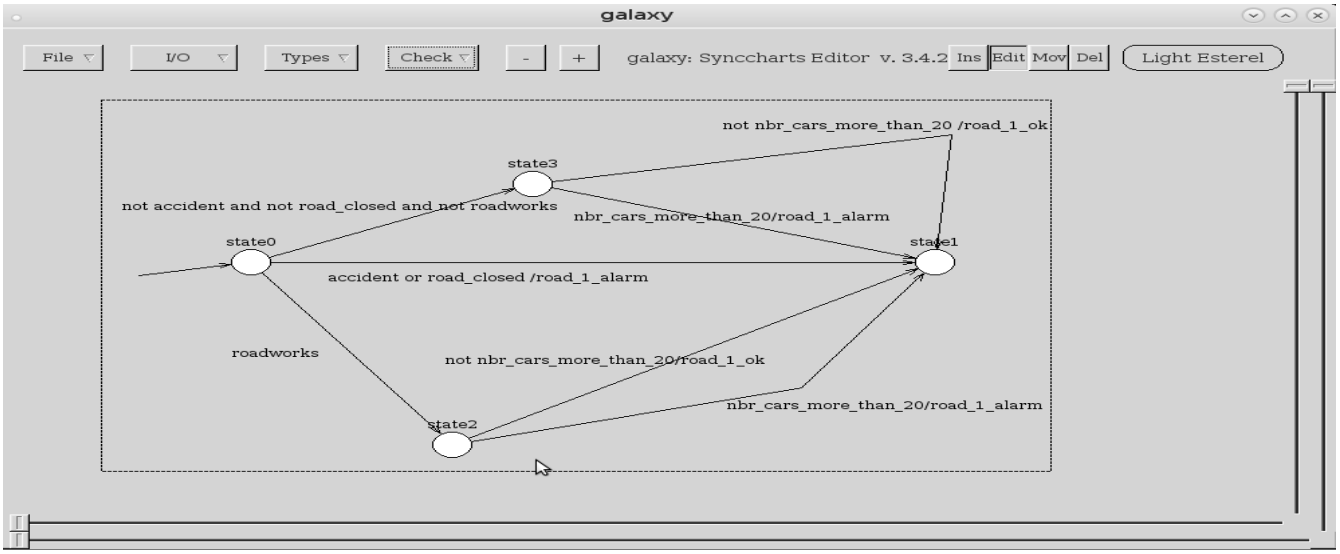
* If there's an accident or the road is closed, the CEP will send automatically a road_alarm information.
* If there are roadworks, the CEP will check the number of cars. If the number of cars is less than 20 then it will send an information saying that the user can take this road (ex: road_ok).
* If there is no accident, no roadworks, and no road closed, the CEP will verify the number of cars in the road.

This processing should be done for the two roads of each proposition and should work in parallel.
One important specification that we have to verify also, is that if there are roadworks in the two roads of the proposition, the CEP will suggest to the user to take the road_1, by default.

A ROAD_INFORMATION MODEL

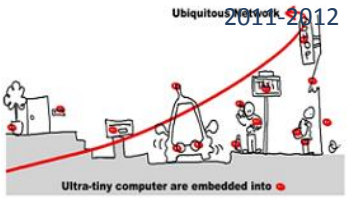
As it is mainly a controller, it is recommended to rely on explicit Mealy machines to carry out the design. The following diagram shows a possible implementation of a road information component behavior with GALAXY (In this example we design the behavior of the first road of the proposition Nice_Sophia). There are four inputs: accident, roadworks, road_closed and nbr_cars_more_than_20. We assume that road information component decide that the road can be taken or not according to these four signals. There are two outputs: road_1_ok and road_1_alarm.

Thus, we design a state machine with 4 states. Transitions are triggered with the input signals and the output variables are sustained, we can have transitions without output variables.



ROAD1_NICE_SOPHIA DESIGN IN GALAXY

annie.ressouche@inria.fr, ines.sarray@inria.fr



TO DO

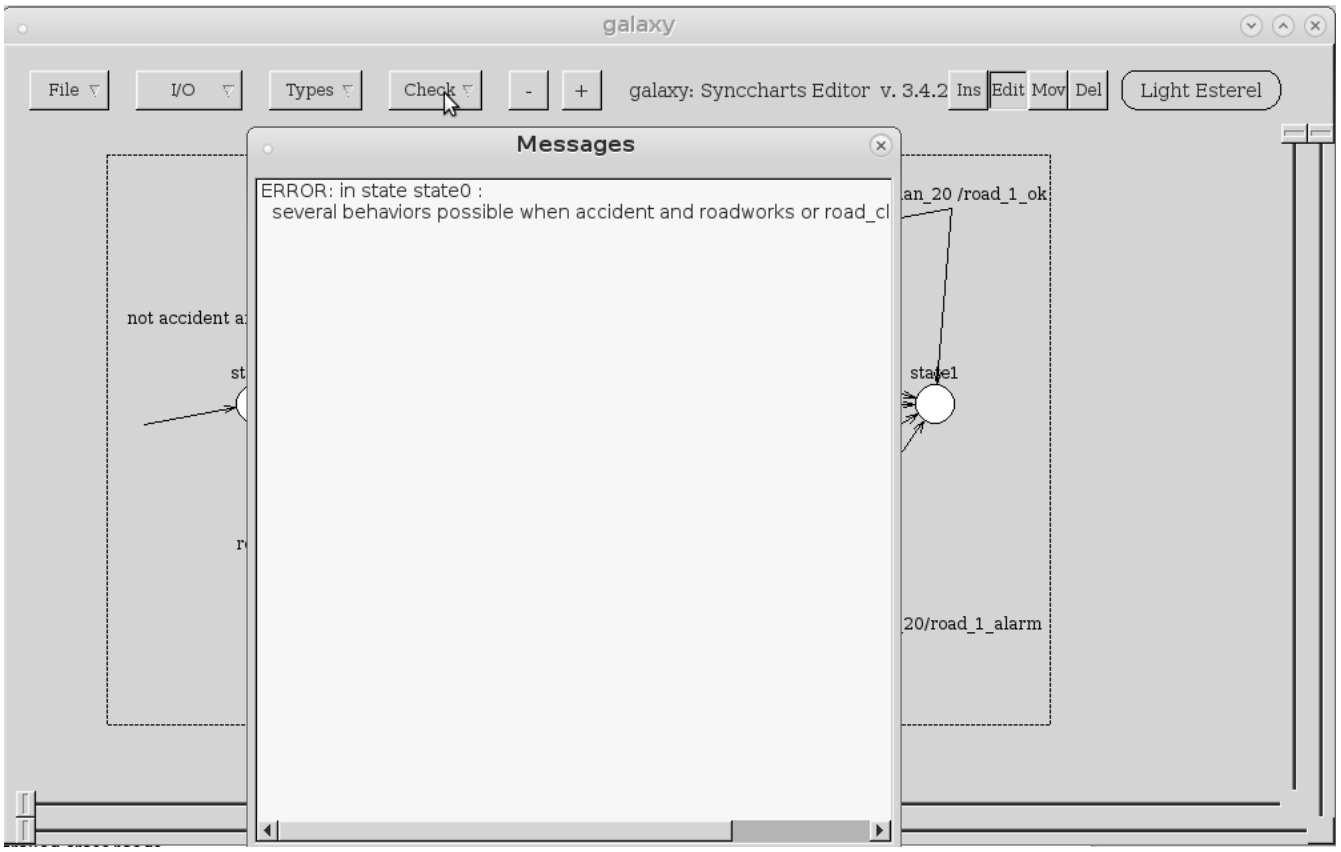
Design your own road model with Galaxy.

DETERMINISM VERIFICATION

As we said in the lecture, one of the most important feature of the synchronous language is determinism.

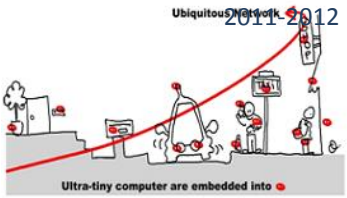
To check the determinism of our mealy machine using galaxy, click on the button check.

You will find that the determinism is not ensured with this mealy machine and some errors will appear saying that there are some behaviors that are not mentioned.

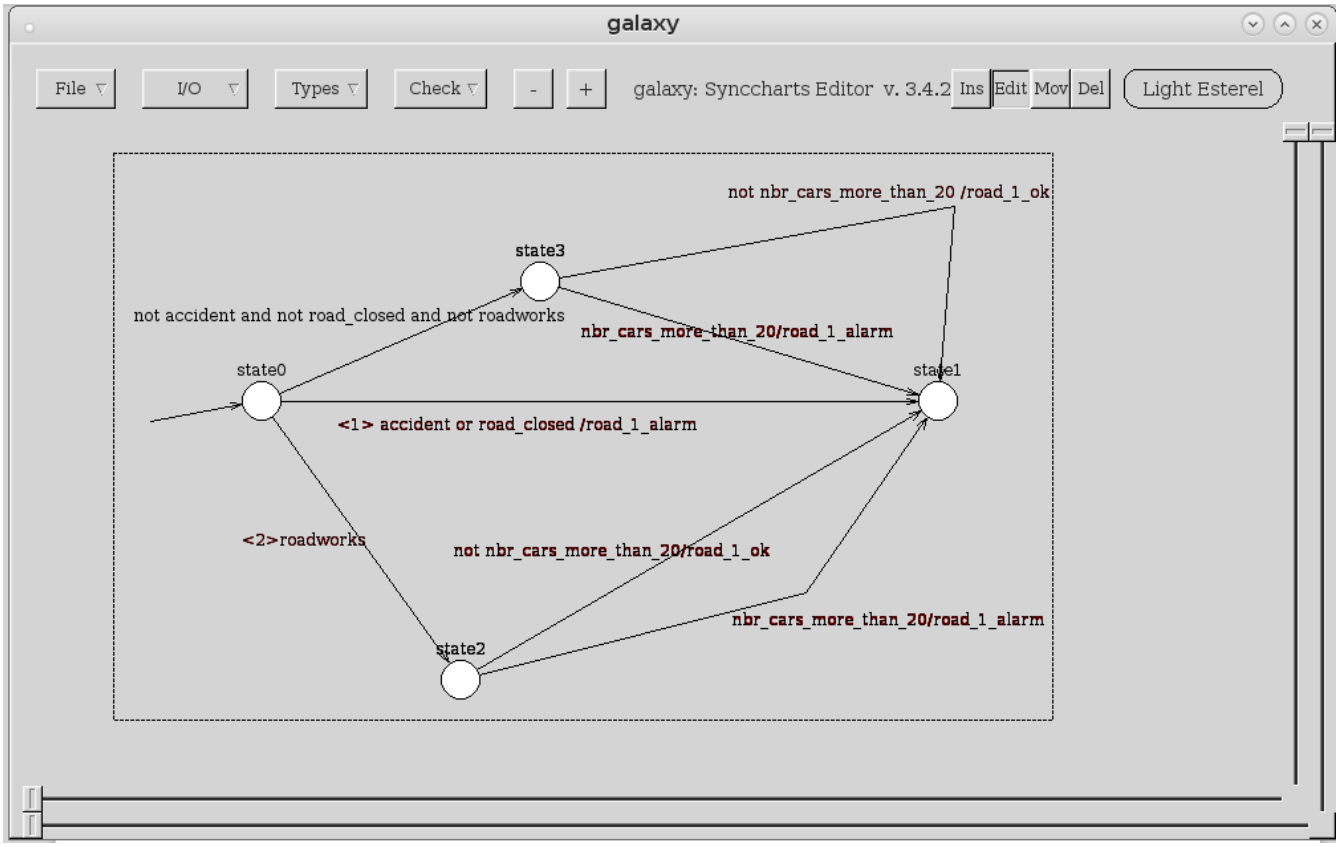


ERROR MESSAGE AFTER DETERMINISM CHECKING

annie.ressouche@inria.fr, ines.sarray@inria.fr



As we don't want to complicate more our mealy machine, we will not add these cases. On the other hand, we will use an other concept that galaxy tool offers to us which is priority, We will give the strongest priority (presented as “<1>”) to the transition: accident or road_closed. The transition: roadworks will have the second priority (presented as “ <2>”).This means that if the mealy machine receives two inputs (accident(or road_closed) and roadworks in the same time, it will use the transition that has the strongest priority which is “ accident or road_closed” transition in our case.



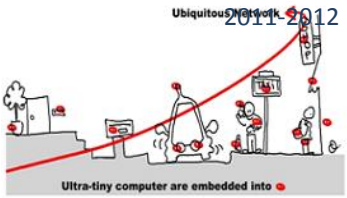
TRANSITION PRIORITIES IN GALAXY

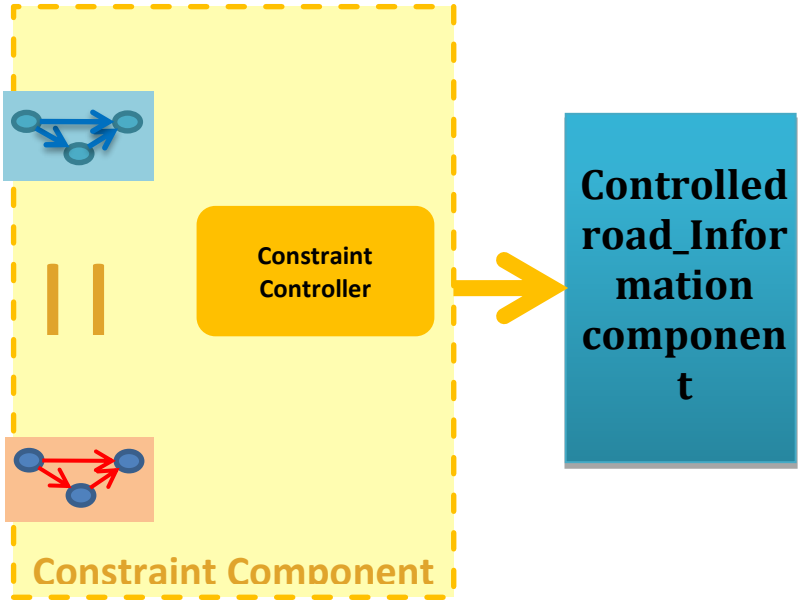
TO DO

Correct your mealy machine, check the determinism again and simulate it using the Clem tool.

NICE_SOPHIA ROAD_INFORMATION COMPONENT

Following the approach described in the lecture, we must design a constraint component made of the respective models of the two road_information of the proposition “Nice_Sophia” and a constraint controller specifying the constraints on the respective outputs of each road_information model to get a controlled road_information component.

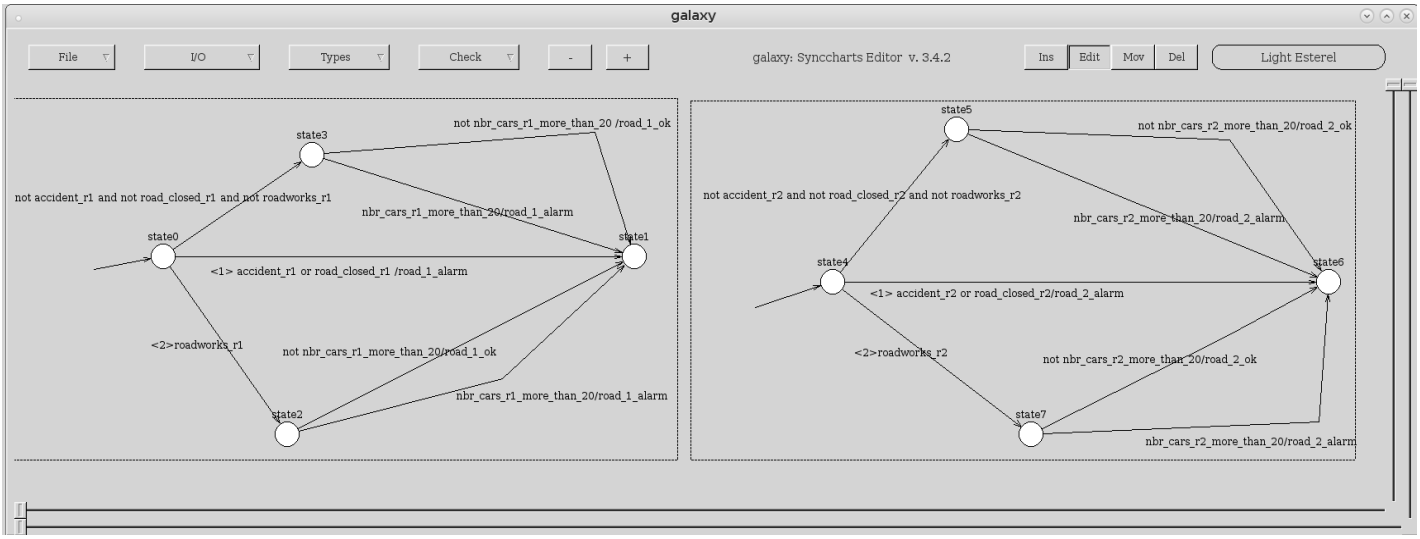




The road_information constraint controller

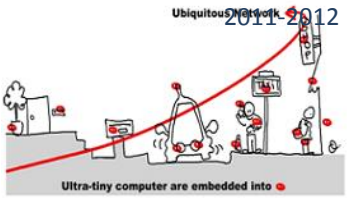
SPECIFICATION

The first idea we could have is to define the two road_information models of Nice_Sophia proposition. These two models should work in parallel. Then the constraint controller turns out to be useless and the constraint component is just the parallel of these two models. For instance we could design the following parallel Mealy machine:



PARALLEL MEALY MACHINE

annie.ressouche@inria.fr, ines.sarray@inria.fr



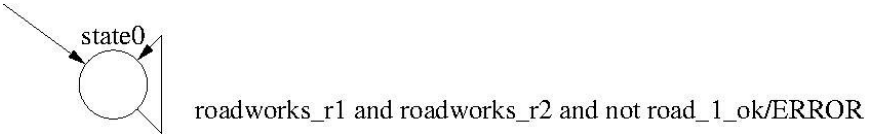
TO DO

Design and simulate your own road_information component.

VERIFICATION

Before using this road_information component, you must ensure that it is correct. For this aim, we will use model checking technique. The property we want to prove is: if roadworks_r1 and roadworks_r2 both hold, then by default road_1_ok is sent.

To achieve the proof we can rely on xeve or NuSMV model checkers and the observer technique is used. Thus, we define an observer of the property. According to observer approach, we describe the property in GALAXY, as a Mealy machine:

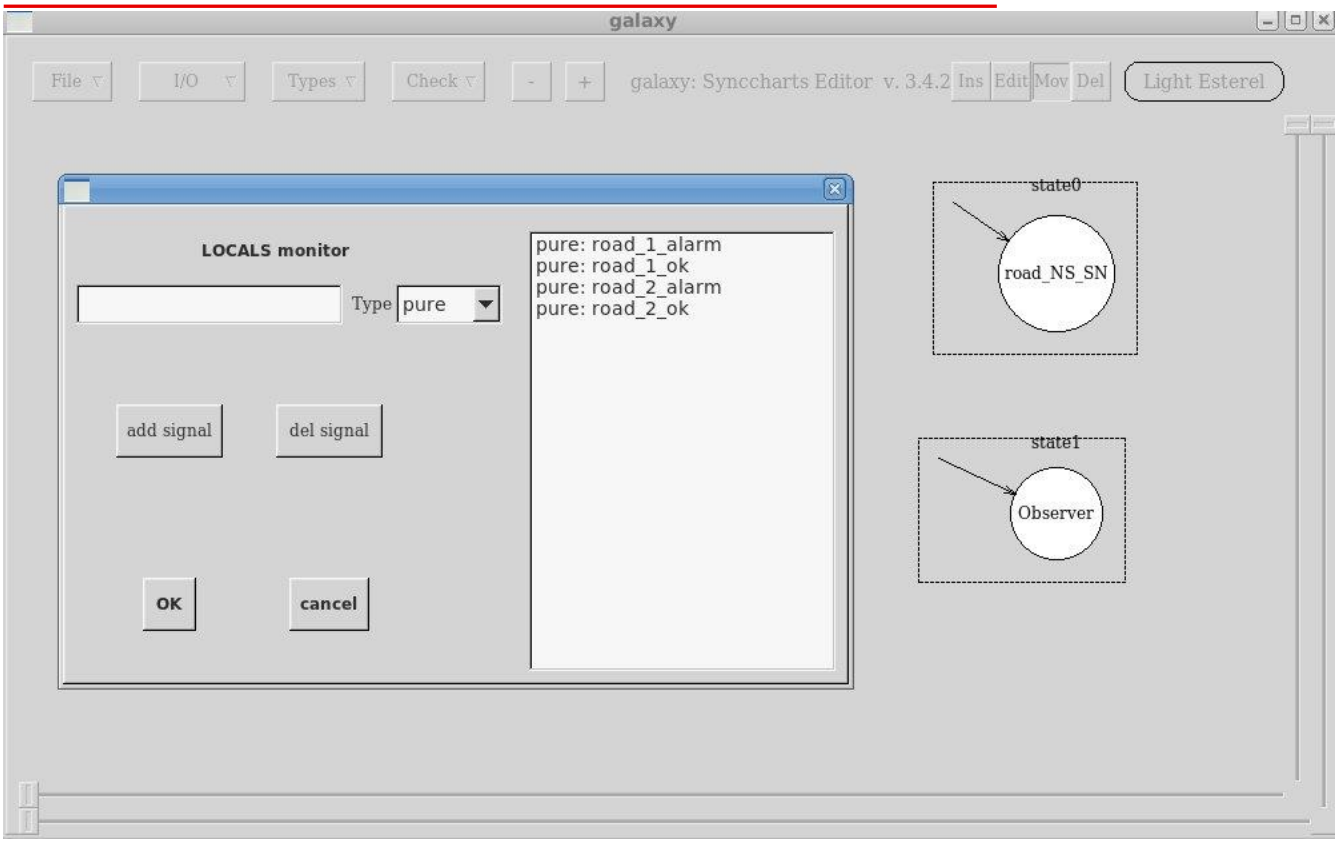


THE OBSERVER (NAMED OBSERVER.GAL) LISTEN 3 INPUTS AND OUTPUTS AN ERROR WHEN THE PROPERTY IS VIOLATED..

However we could also describe the property as an implicit Mealy machine as well.

Now to run the model-checker, we must define another module (let us call it verifRoad_NS_SN) which is the parallel composition of the road_NS_SN design and the observer:

annie.ressouche@inria.fr, ines.sarray@inria.fr

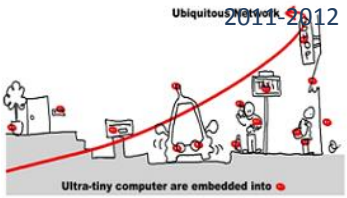


VERIFROAD_NS_SN IS THE PARALLEL OF ROAD_NS_SN AND THE OBSERVER. IT INPUTS SIGNALS ARE ALL THE INPUT SIGNALS OF ROAD_NS_SN AND IT OUTPUT SIGNAL IS ERROR AND THE OUTPUT SIGNALS OF ROAD_NS_SN MODULE BECOME LOCALS.

We also can specify this verification module in LE:

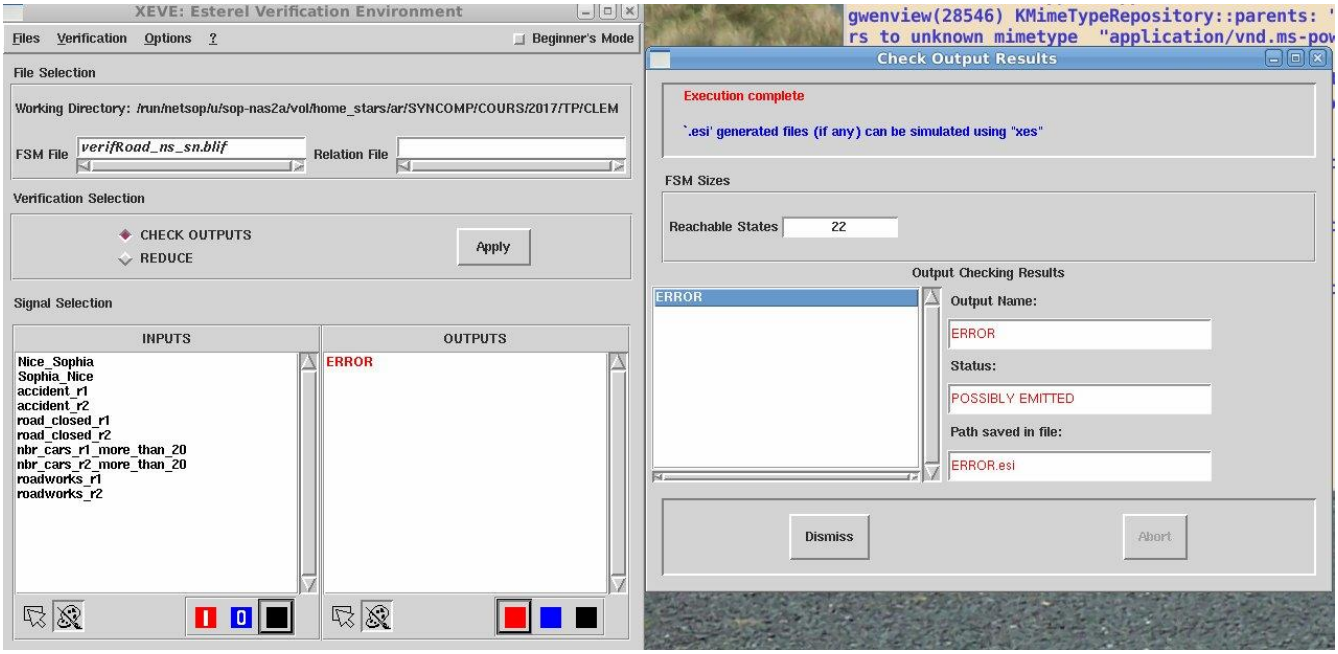
```
module verifRoad_NS_SN:

Input:
Nice_Sophia,Sophia_Nice,accident_r1,accident_r2,road_closed_r1,road_closed_r2,

nbr_cars_r1_more_than_20,nbr_cars_r2_more_than_20,roadworks_r1,roadworks_r2;

Output: ERROR;

local road_1_alarm,road_1_ok,road_2_alarm,road_2_ok

{

     run road_NS_SN

||

    run Observer

}
```


annie.ressouche@inria.fr, ines.sarray@inria.fr




```
end
```

Now to validate the property, we generate with CLEM the blif code and we call xeve:



The property is false and we must correct our design until the property becomes true, before using it as a CEP.

On another hand, we can also generate the SMV code for verifROAD_NS_SN module and enter NuSMV, following the description detailed previously. The validation of the property: “AG !mverifROAD_NS_SN” returns also a failure.

TO DO

Program this Road_Information component in CLEM, verify if the property holds. If not, correct the design until the property holds.

THE CEP COMPONENT

Now we will design the behavior of our CEP Components using galaxy tool also. As we said above, we will create two CEPs, they represent Successively Nice/Sophia and Antibes/Sophia regions. For the moment we will create the NICE/Sophia CEP, It represents a “big” hierarchical mealy machine that calls the two propositions mealy machines (the parallel mealy machine of Sophia_Nice et the other one of Nice_Sophia), we will add two inputs:

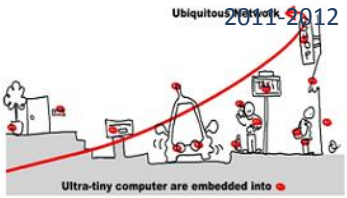
Nice_Sophia: That will allow the running of the the parallel mealy machine of Nice_Sophia.

Sophia_Nice: That will allow the running of the the parallel mealy machine of Sophia_Nice.

To call the two parallel mealy machines, we will use and define the run operations in the galaxy tools and redeclare all the inputs and outputs that we will use.

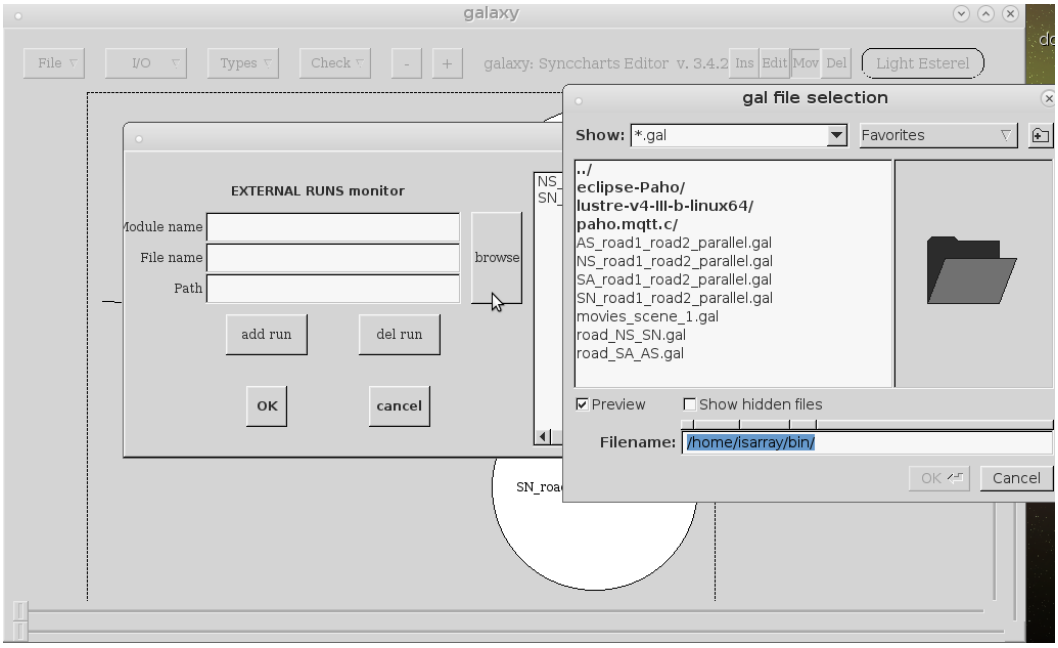
annie.ressouche@inria.fr, ines.sarray@inria.fr

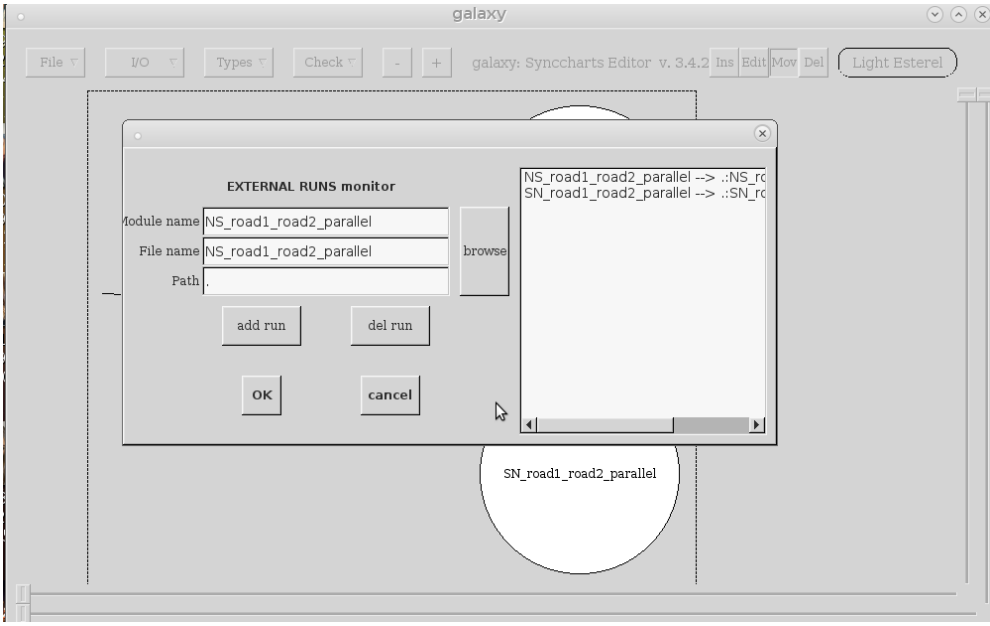
TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH



How to use runs ?

1- Create a new run (in the I/O), click on browse and choose your mealy machine that you want to call

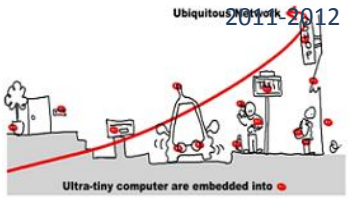


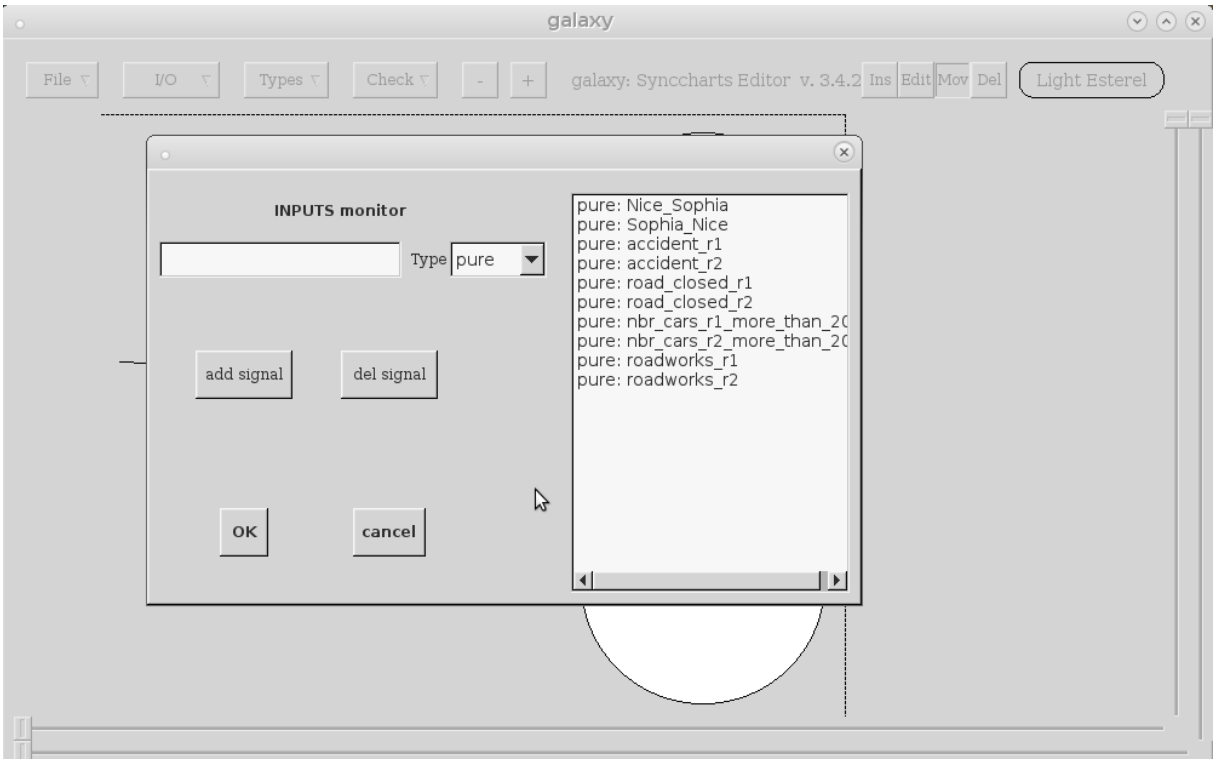


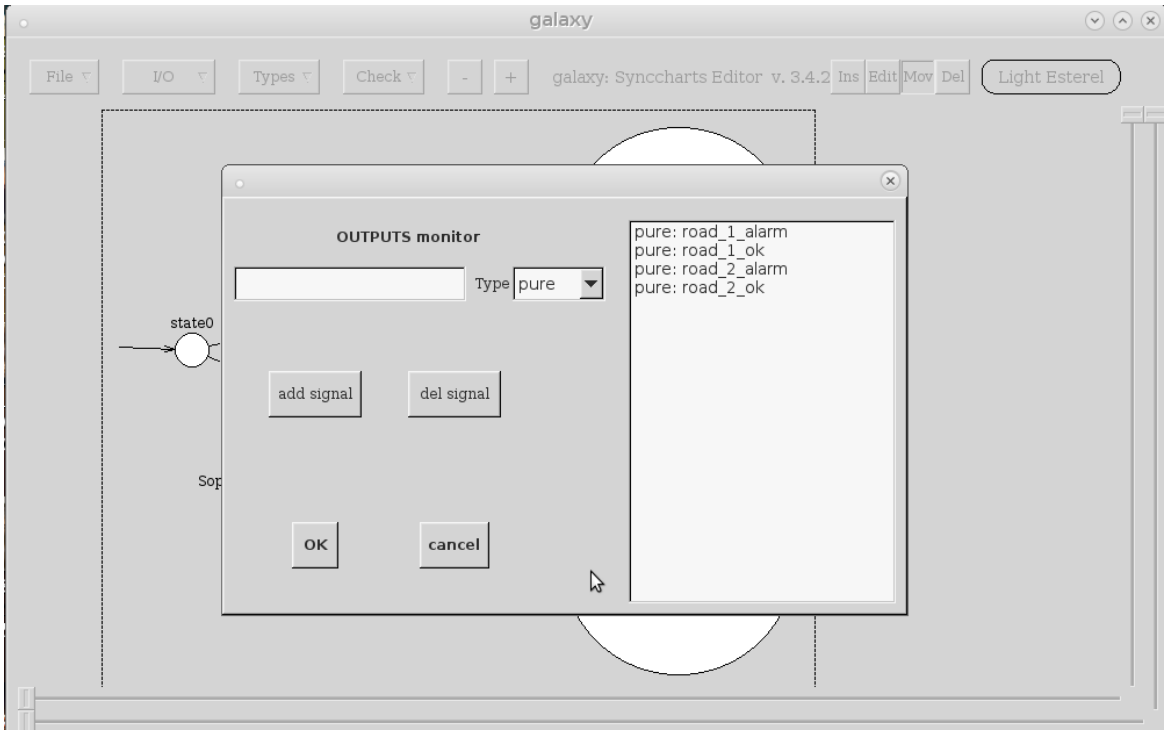
2- Define all the inputs and outputs that we will use in this CEP

annie.ressouche@inria.fr, ines.sarray@inria.fr

TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH

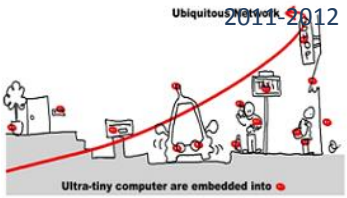




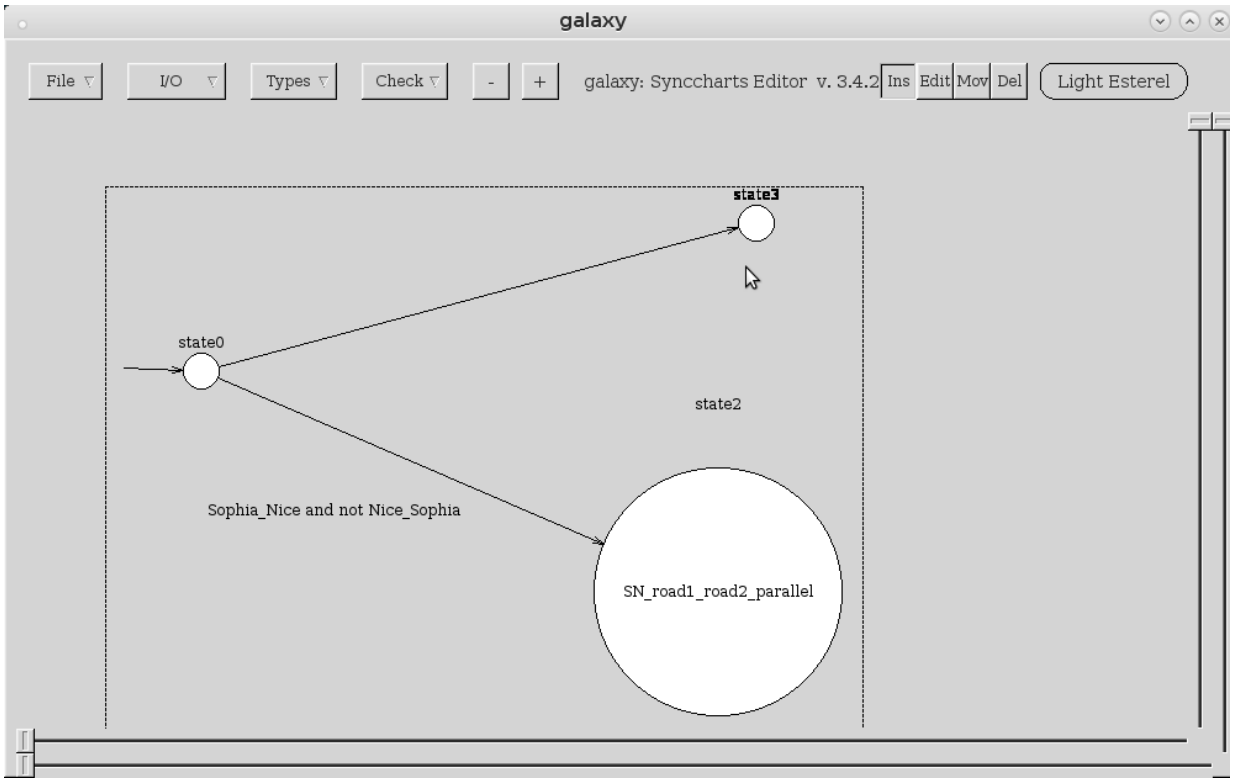


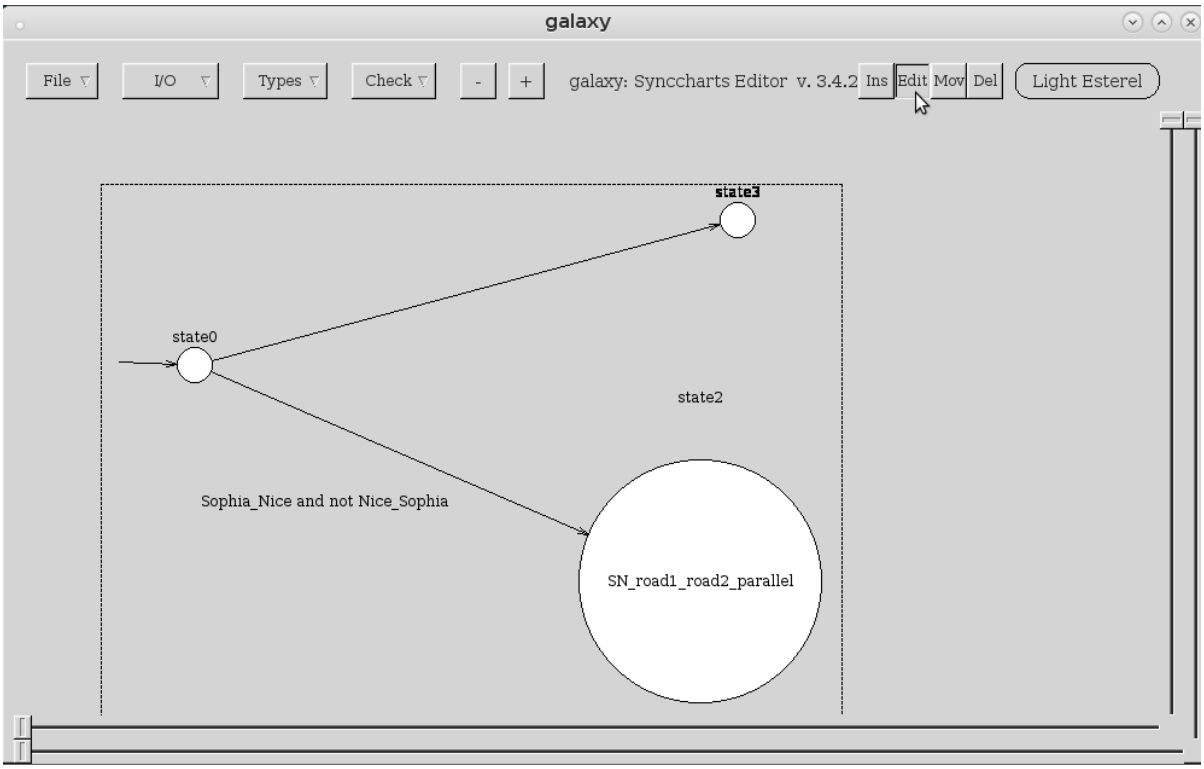
annie.ressouche@inria.fr, ines.sarray@inria.fr

TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH



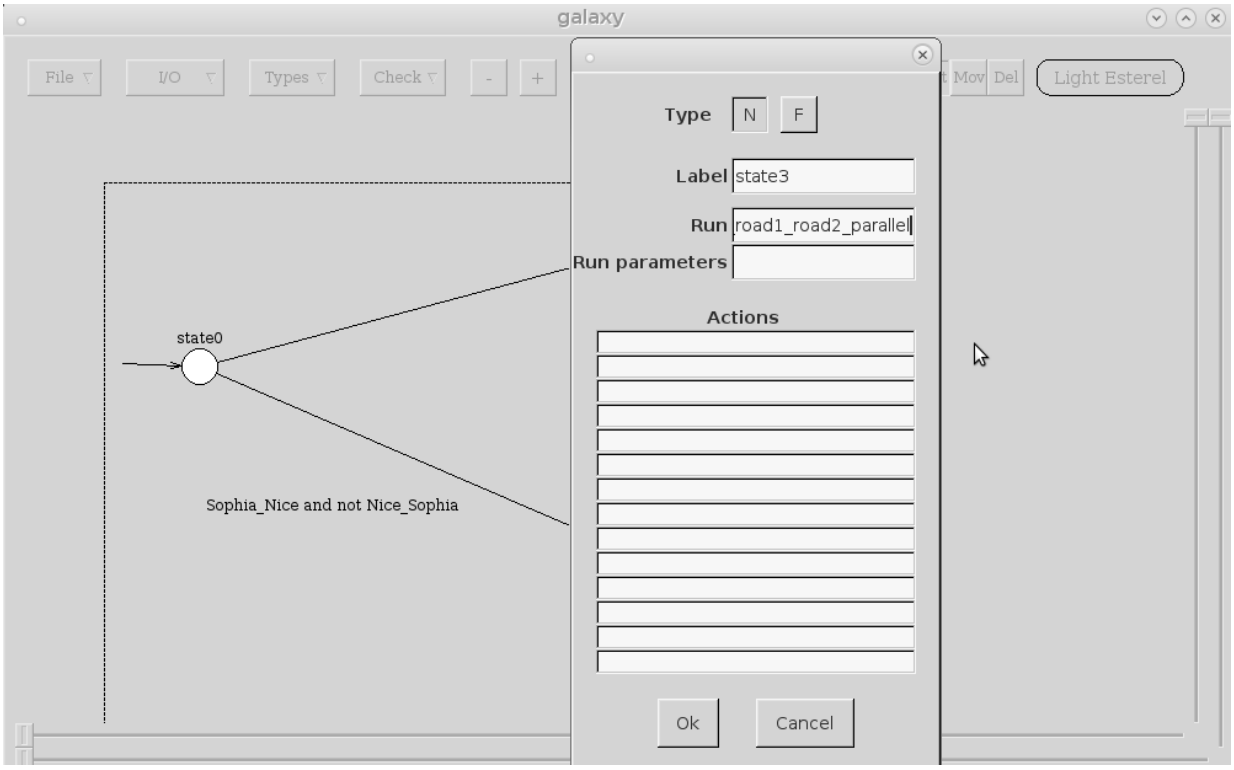
3- Create a new state, and modify its properties to add the name of the called module(mealy machine name)

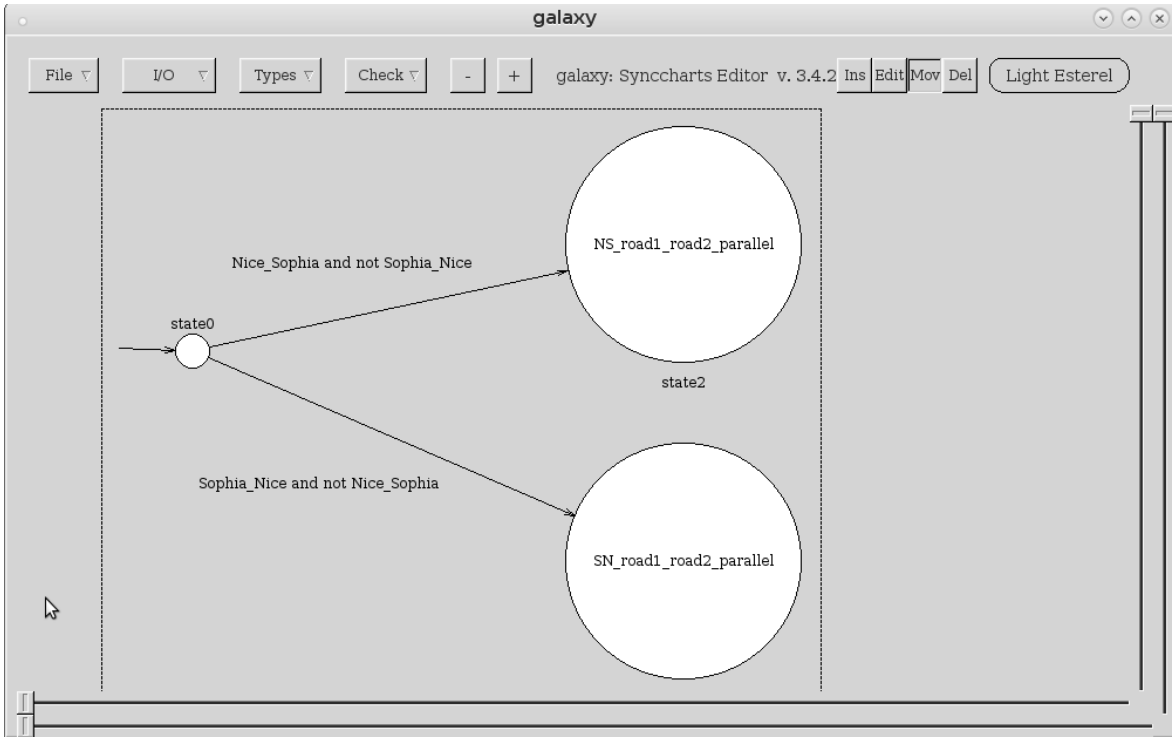




annie.ressouche@inria.fr, ines.sarray@inria.fr

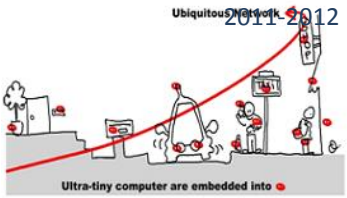
TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH





NICE/S

annie.ressouche@inria.fr, ines.sarray@inria.fr



TO DO

Create the second parallel Mealy machine of Sophia_Nice road_Information and the create the NICE/SOPHIA Component.

COMMUNICATION WITH MQTT APPROACH

As we need to establish a communication between the different CEPs and the environment data using the MQTT approach, we need to create an MQTT client that takes into consideration the behavior of our CEP.

In this tutorial we will use the C code to establish the communication between different MQTT client, to this aim we need to generate the C code of our CEP mealy machine.

C CODE GENERATION

To generate the C code for our First CEP component, just call clem and generate the C code, it will generate two files: (.h and .c files). The generated files will define mainly two methods: **road_NS_SN_reset_automaton(parameters..)** and **road_NS_SN_automaton(parameters...).** According to the synchronous approach, mentioned previously, the CEP component model is a Mealy machine and it is represented with a set of Boolean equations (see the lecture). Hence, the C code mainly implements a run of this Mealy machine. It is the main goal of **road_NS_SN _automaton** method.

TO DO

Generate the C code of the CEP component model.

COMMUNICATION WITH MQTT

As we will use C language, you should install eclipse-paho to create MQTT clients (publishers and subscribers).

MQTT CLIENT IN C PROGRAMMING LANGUAGE AND POSIX LIBRARIES

See for eclipse paho installation on Linux and Windows : https://www.eclipse.org/paho/clients/c/

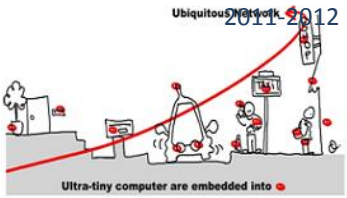
Here you can find sample POSIX/C code to test MQTT API in the Paho Eclipse project : https://projects.eclipse.org/projects/technology.paho

See for libraries: https://www.eclipse.org/downloads/download.php?file=/paho/1.0/m2mqtt/M2Mqtt_4.0.0.0_bins.zip

See for sample codes: https://www.eclipse.org/downloads/download.php?file=/paho/1.1/eclipse-paho-mqtt-c-windows-1.0.3.zip and explanations at https://eclipse.org/paho/clients/c/

annie.ressouche@inria.fr, ines.sarray@inria.fr

TUTORIAL: CREATING A VALIDATED CEP NODE IN A MQTT APPROACH



MQTT CLIENT

After generating the road_information CEP component, we need to create an MQTT client that subscribes to a broker (here we will use mosquitto broker) reads information, makes the processing using the generated methods of the designed CEP component, and publishes the results into the broker.

You will find attached three files :

-**1 / testPub.c** : This file contains an example of a publisher client code.

-**2/ MQTTClient_subscribe.c:** This file contains an example of a subscriber client code.

-3/ CEP1_Test.c : This file contains the implementation of our first CEP.

TO DO

With the help of the two previous files (**testPub.c** and **MQTTClient_sunscribe.c**), finish the implementation of messageTest, msgArrived and main functions (Everything is mentioned in the file **CEP1_Test.c**).

You can use this link also to have more information about the predefined functions:
https://www.eclipse.org/paho/files/mqttdoc/MQTTClient/html/index.html

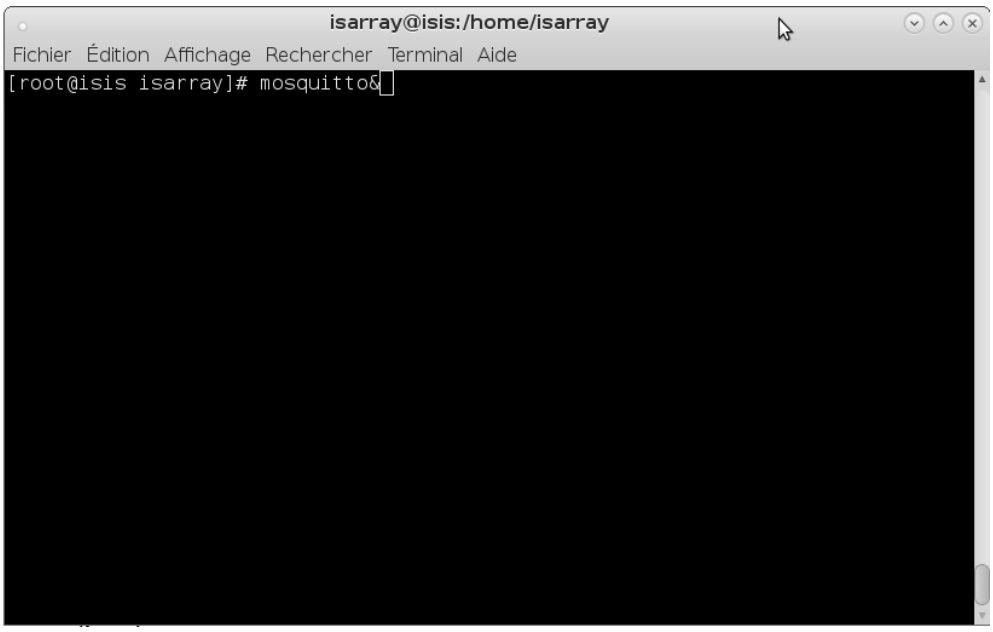
To compile the code you should use this command **: gcc road_NS_SN.c CEP1_Test.c -l paho-mqtt3c -o CEP_Test**

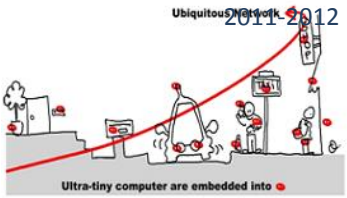
road_NS_SN.c: the name of the C code generated by the clem tool

CEP1_Test.c : the name of the client file (CEP), you can choose the name that you want.

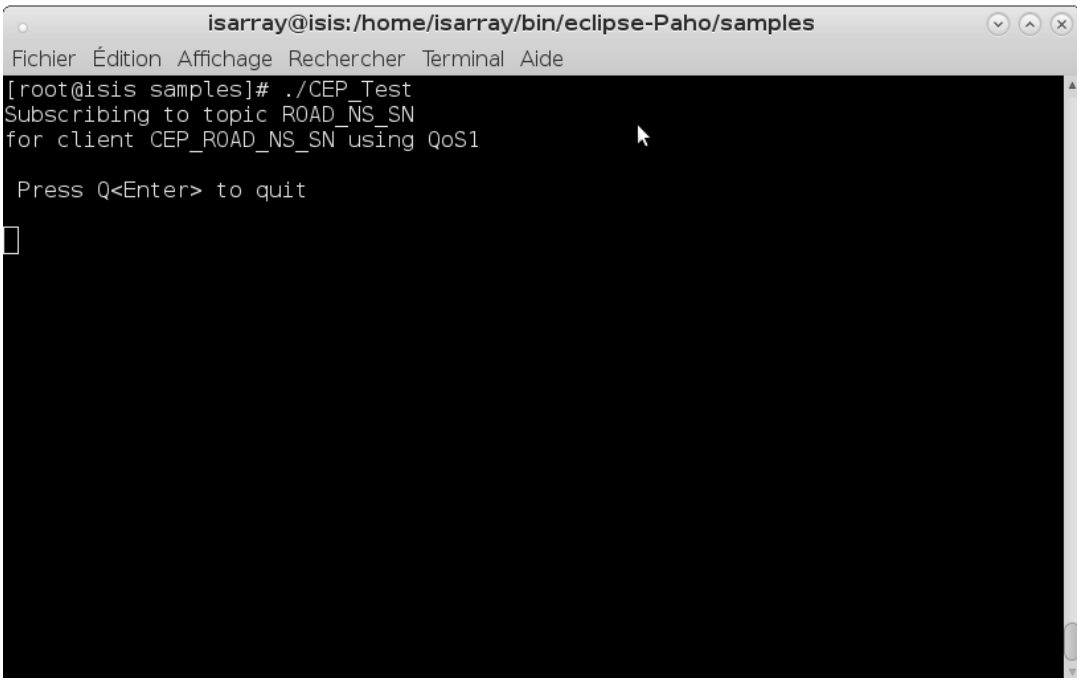
TEST

After the compilation, we will test our test: to this aim you should execute mosquitto in background using the command: **mosquitto&**



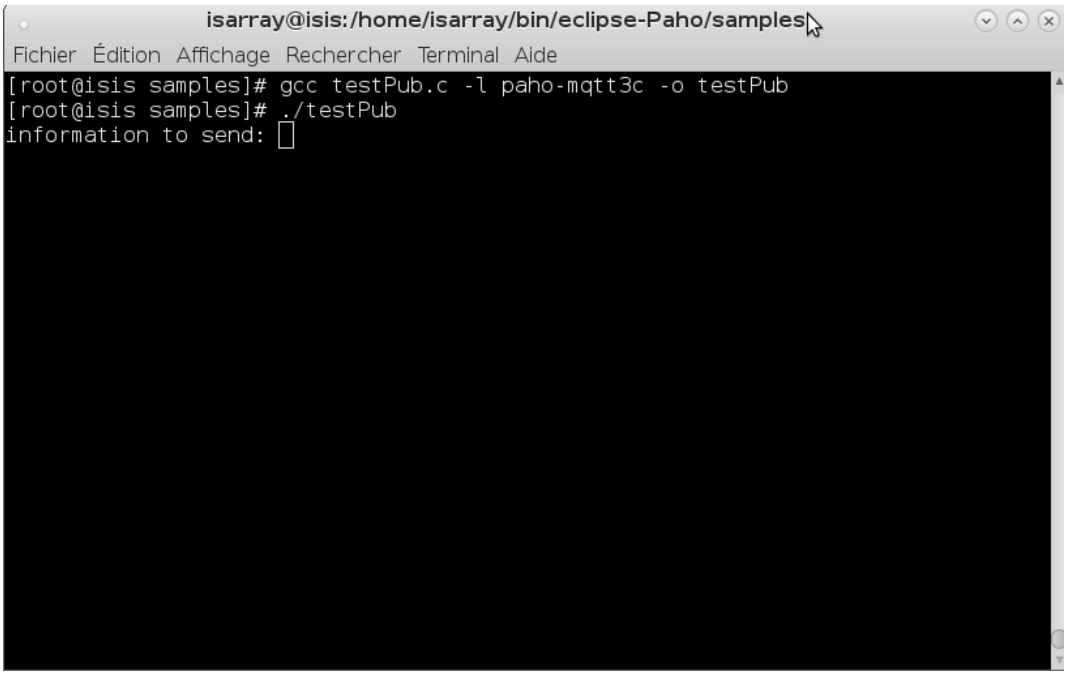


you can execute your client using the command**: ./CEP_Test (** in case you used the same name as the tutorial after the option -o)

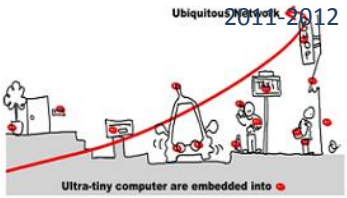


You will find attached a file of a publisher that allows you to write and send information. In another terminal, compile it using the command**:**

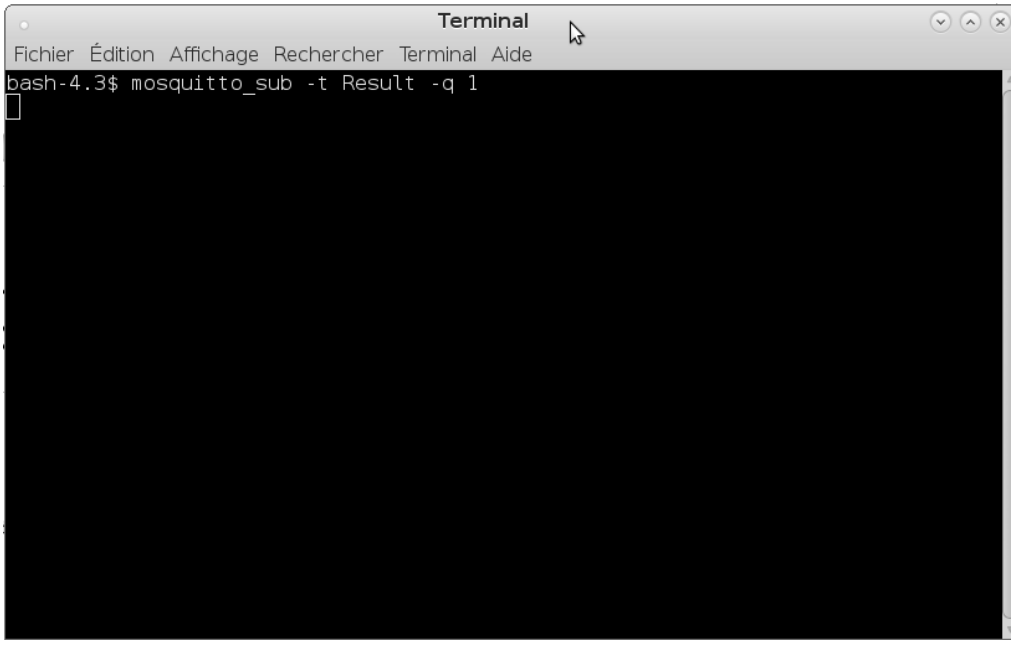
gcc testPub.c -l paho-mqtt3c -o testPub and then execute it with the command **./testPub:**



annie.ressouche@inria.fr, ines.sarray@inria.fr

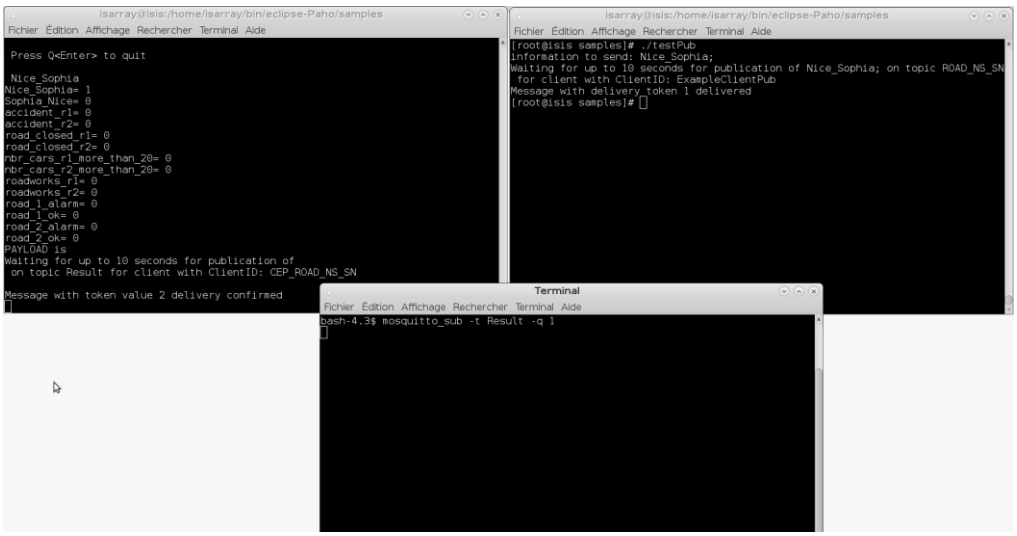


We need another subscriber in another terminal to see if the result of the CEP were publisher in the broker, to this aim we will create a simple subscriber that will subscribe to the topic **"Result"** using the command : **mosquitto_sub -t Result -q 1** and we will see if it will receive the result or not.

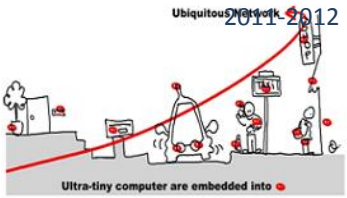


Now we will send some information to our client using the publisher "testPub", we will write first the choice of the user, (Nice_Sophia or Sophia_Nice)

Be careful!! : you should use the same names of your automata inputs and outputs and you should always put ";" at the end.

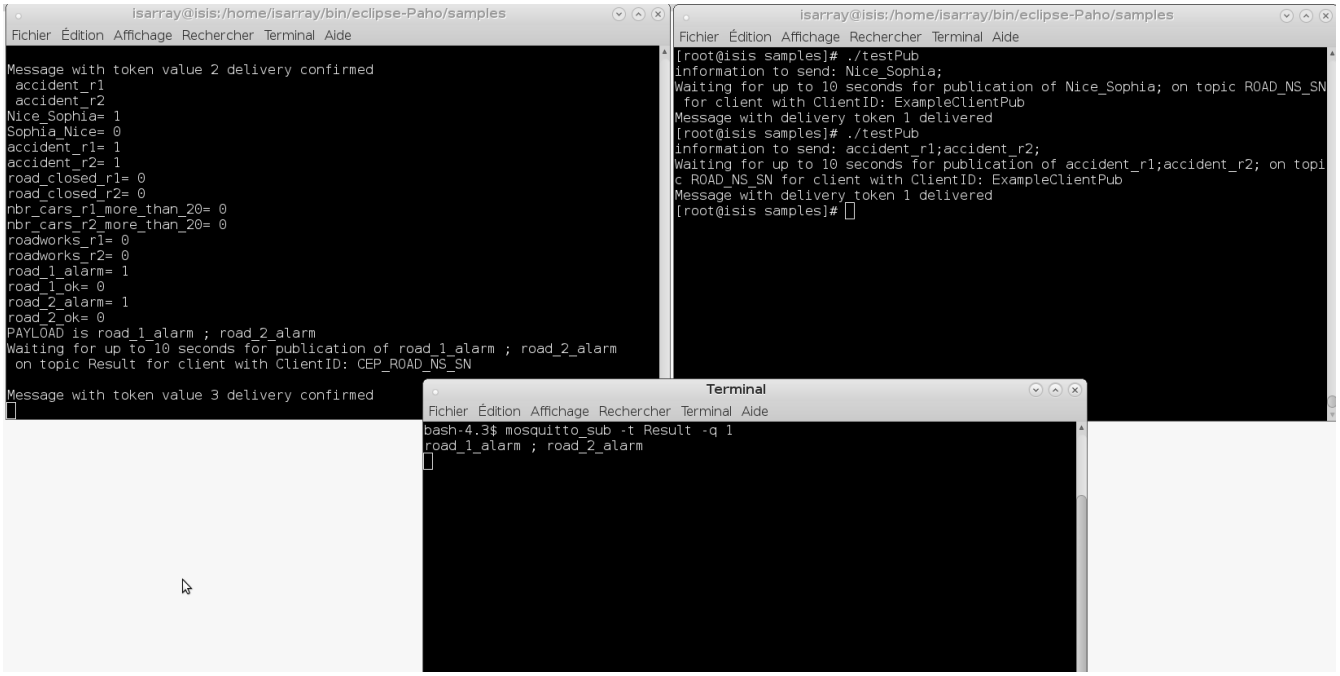


annie.ressouche@inria.fr, ines.sarray@inria.fr



In the first time we sent “Nice_Sophia;”, the CEP client didn't send any result because in the automata the first transition consider only one of the two inputs (Nice_Sophia or Sophia_Nice) and doesn't send any output result, the transition executed when the automata receive this input is to run the parallel mealy machine related to the information related to the choice of the user, in this case, the called automata is the one of the rod_information model of the proposition Nice_Sophia.

Now we will enter other information such us: **accident_r1;accident_r2;**



In this case, the automata has as result the two outputs **: road_1_alarm and road_2_alarm;** These results were received by the **mosquitto subscriber,** which proves that the execution and the communication of the automata with the broker is done.

TO DO

1/ Create the MQTT Client and make the communication Test.

2/ Do the same work for the region Antibes_Sophia: (Creation of automata, CEP component design, generation of the C code, creation of the MQTT client and the tests)