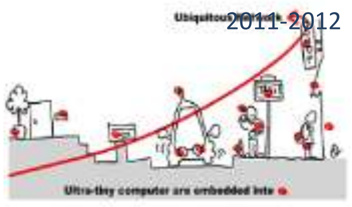
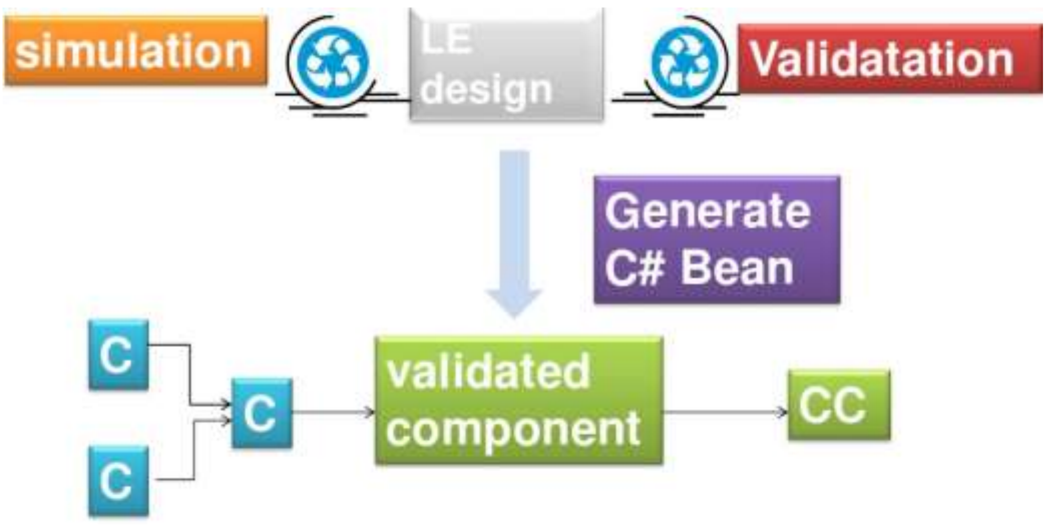


TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

INTRODUCTION

The purpose of this tutorial is to illustrate the Lecture on Verification of Component based adaptive middleware applications. The main goal is to design a validated component in WComp adaptive middleware. Here is the main scheme to design a component:



In this tutorial, we will consider the design in WComp of traffic lights managing a cross roads.

According to our validation concern, we will follow the methodology detailed in the lecture. Thus, we will describe a synchronous monitor to verify the behavior of our traffic lights manager and then introduce it as a validated component in WComp. To design this synchronous monitor 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 monitors.

THE CLEM TOOLKIT

The CLEM toolkit is a set of tools around the LE synchronous language. It allows specifying synchronous monitor and to generate automatically C# code to introduce in a WComp design, this synchronous monitor as a component Bean.

First of all, CLEM implements a modular compilation of LE programs and a specific format LEC has been introduced to record already compiled LE modules. Moreover, CLEM offers a simulation means and generates output code for hardware targets as well as software ones. In particular, CLEM generates C# code for WComp.

Different target formats are provided; we detail three of them which will be useful to design our application:

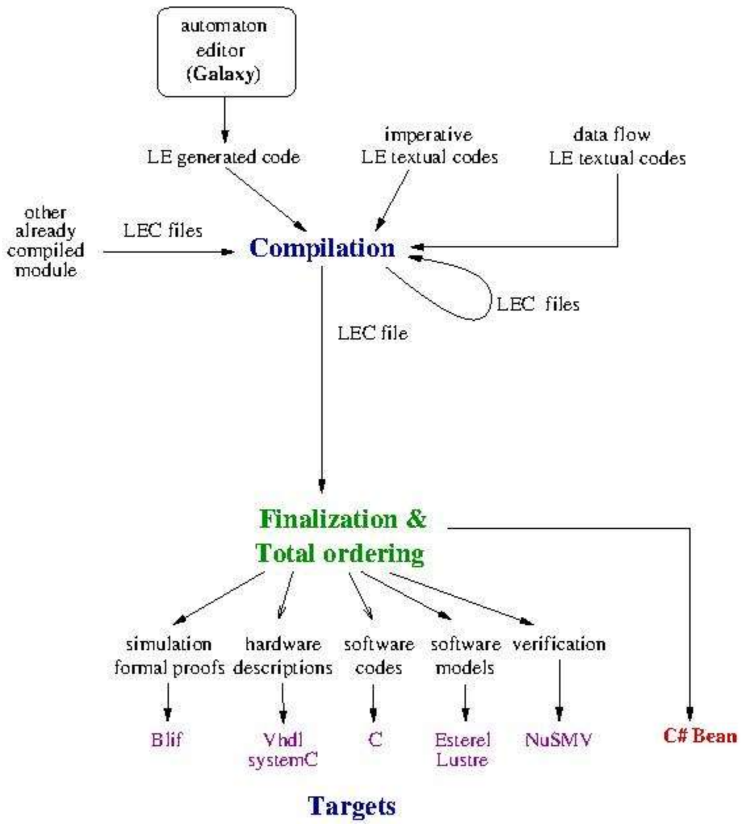
Annie Ressouche INRIA Sophia Antipolis Méditerranée

annie.ressouche@inria.fr

1. LEC: this internal format allows to load already compiled program

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

3. C# for WComp: this format allows implementing Bean in WComp.



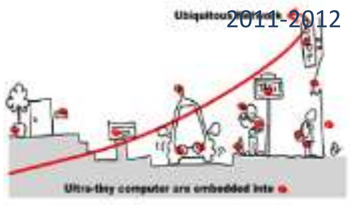
The following software helps us to handle a complete application:

- **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 LEC format. It also generates BLIF format and allows to simulate LE module behavior

- **clef**: i is a finalization software which considers LE module already compiled into LEC format. Indeed, this format is a concise representation of equation systems. Thus, clef simplifies the equations and generates code for different targets, in particular for C#.

- **blif_simul**: is the simulator. It is called with the simulation feature of **clem**.

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



To use this software, put it in a “bin” folder and add to the **Path** environment variable (Control Panel > System > Advanced parameters > Environment variables), the path to reach this bin folder. Here is an example of Path variable:

```
C:\Program Files\Common Files\Microsoft Shared\Windows Live;C:\Program Files
(x86)\Common Files\Microsoft Shared\Windows\Live;…………..;C:\Program
Files\Microsoft SQL Server\110\Tools\Binn\;C:\Users\ar\Documents\bin
```

Assuming that the different software are in: C:\Users\ar\Documents\bin

As LE language is useful to describe synchronous monitor behaviors, 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:

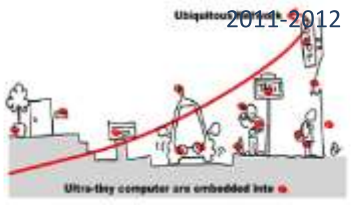
```
module WIEO:

 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



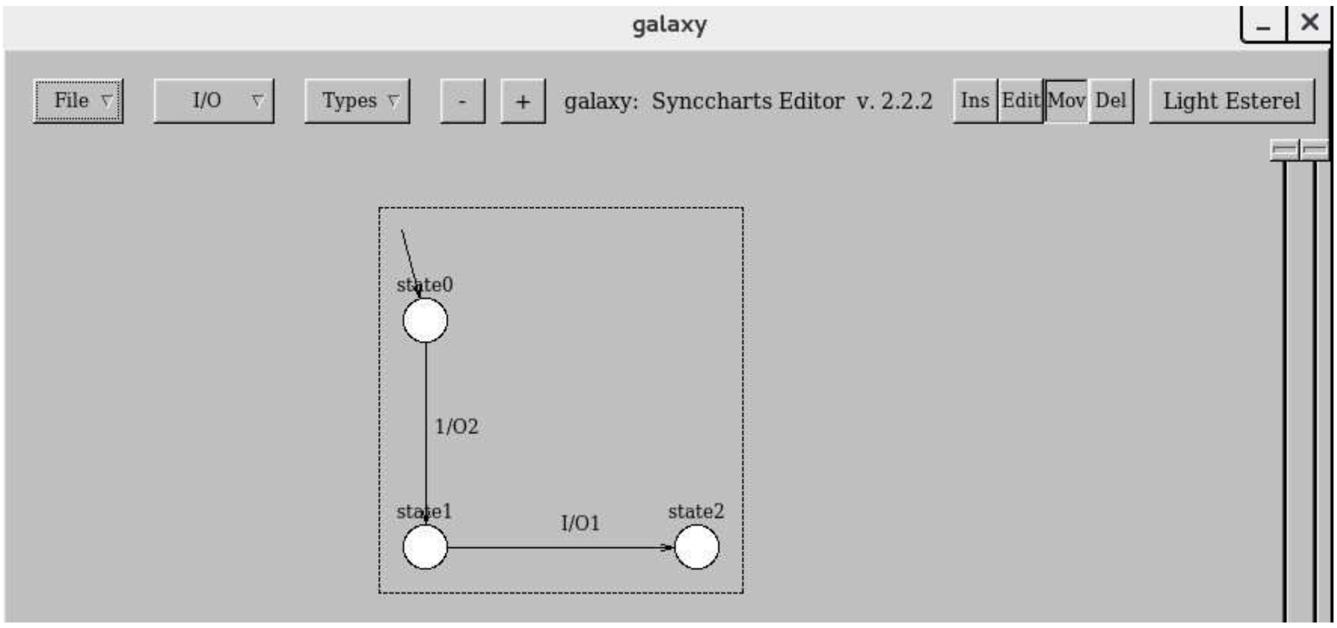
- **emit S** : set the signal **S** present in the environment. It also terminates in the instant.
- **present S {P1} else P2** : if **S** is present then **P1** is executed otherwise **P2** is.
- **P1 || P2**: synchronous parallel operator, it terminates when both of its arguments have terminated.
- **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;

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, syncharts. 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.

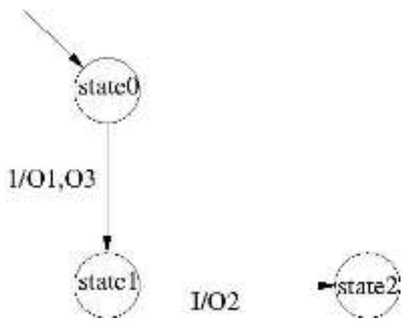
   b. a transition, you define its triggering condition (Condition) and the emitted signals (Actions). The trigger part is a Boolean expression built from the inputs defined in the model, and actions are the outputs of the model.

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:



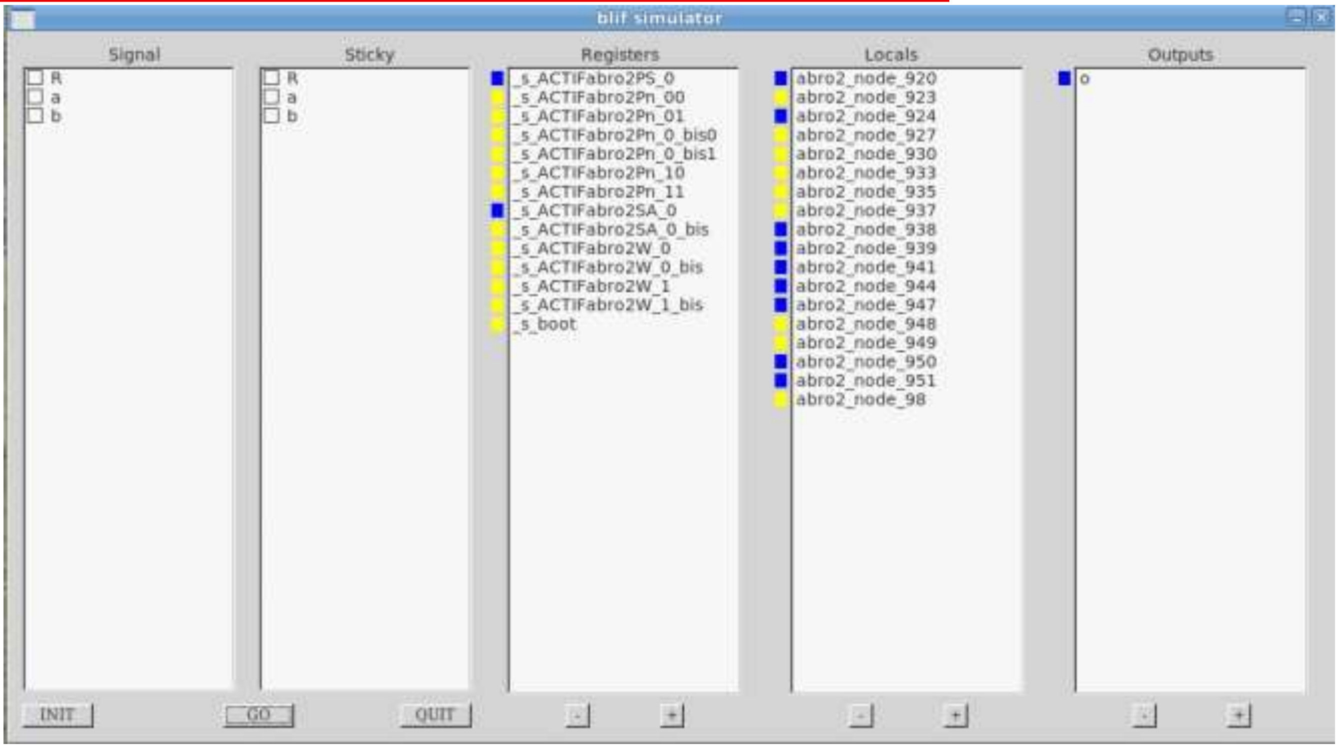
The corresponding implicit Mealy machine in LE, will be defined as :

```
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 AND VERIFICATION

To simulate, you just have to call *clem* and to simulate (*simul* button) your module:



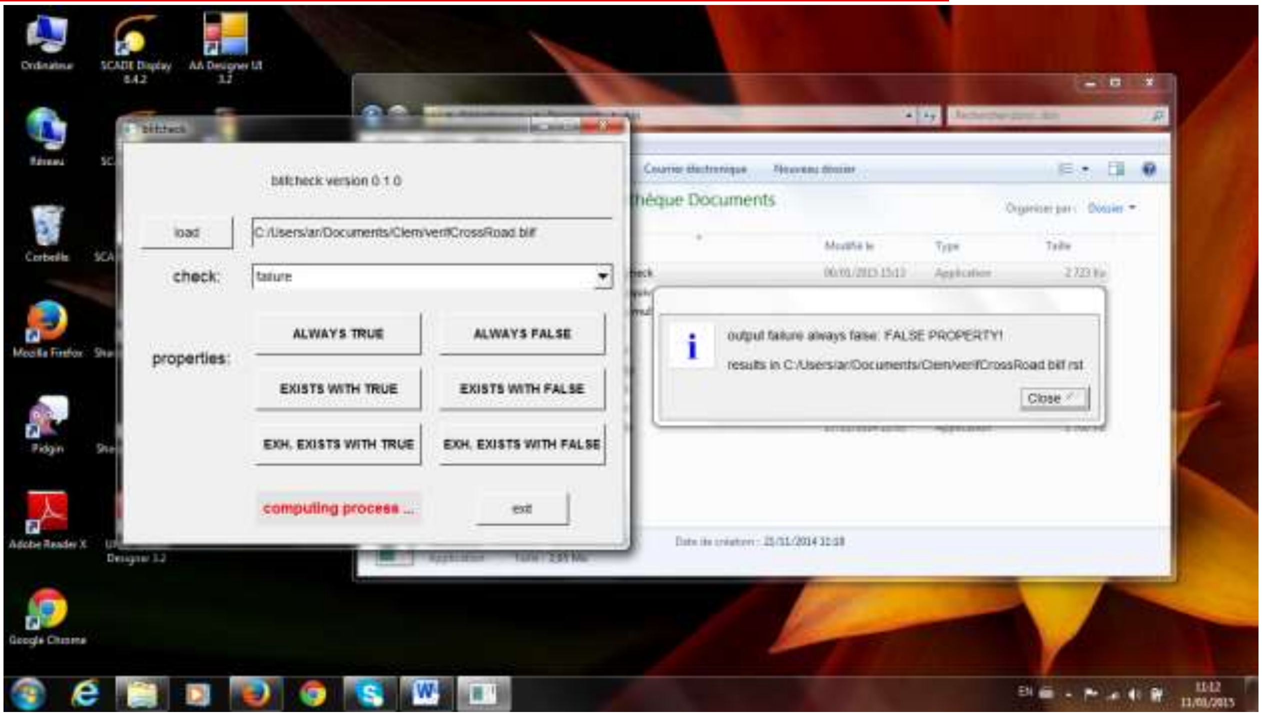
```
Signal        Sticky        Registers                        Locals                  Outputs
☐ R           ☐ R           🟦 _s_ACTIFabro2PS_0             🟦 abro2_node_920       🟦 o
☐ a           ☐ a           🟨 _s_ACTIFabro2Pn_00            🟨 abro2_node_923
☐ b           ☐ b           🟨 _s_ACTIFabro2Pn_01            🟦 abro2_node_924
                            🟨 _s_ACTIFabro2Pn_0_bis0        🟨 abro2_node_927
                            🟨 _s_ACTIFabro2Pn_0_bis1        🟨 abro2_node_930
                            🟨 _s_ACTIFabro2Pn_10            🟨 abro2_node_933
                            🟨 _s_ACTIFabro2Pn_11            🟨 abro2_node_935
                            🟦 _s_ACTIFabro2SA_0             🟨 abro2_node_937
                            🟨 _s_ACTIFabro2SA_0_bis         🟦 abro2_node_938
                            🟨 _s_ACTIFabro2W_0              🟦 abro2_node_939
                            🟨 _s_ACTIFabro2W_0_bis          🟦 abro2_node_941
                            🟨 _s_ACTIFabro2W_1              🟦 abro2_node_944
                            🟨 _s_ACTIFabro2W_1_bis          🟦 abro2_node_947
                            🟨 _s_boot                       🟨 abro2_node_948
                                                             🟨 abro2_node_949
                                                             🟦 abro2_node_950
                                                             🟦 abro2_node_951
                                                             🟨 abro2_node_98
```

This is the simulation window for abro LE module. 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 interpretation: red means undefined, yellow means absent and blue means present.

As already said the simulation feature of CLEM automatically calls the *blif_simul* software and generates the BLIF code associated with the module to allow *blif_simul* running. We can benefit from this generation, to call *blif_check* and perform formal verification. The following diagram shows you a run of blif_check. In this latter, you must:

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 in an ".rst" file. 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.


blifcheck version 0.1.0

load: C:/Users/ar/Documents/Clem/verifCrossRoad.blif

check: failure

output failure always false: FALSE PROPERTY!

results in C:/Users/ar/Documents/Clem/verifCrossRoad.blif.rst

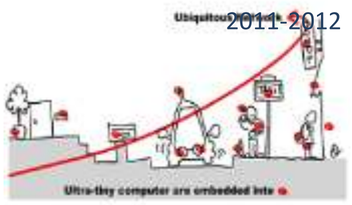
This is an example of blif_check usage. First, we load the module verifCrossRoad.blif. Indeed, this module is composed of the parallel of a module describing a crossRoad behavior and and observer checking a safety property and emitting failure when the property fails. Then, we check that failute is always false and we get an error and the counter example is generated in a file verifCrossRoad.blif.rst

EXERCISE

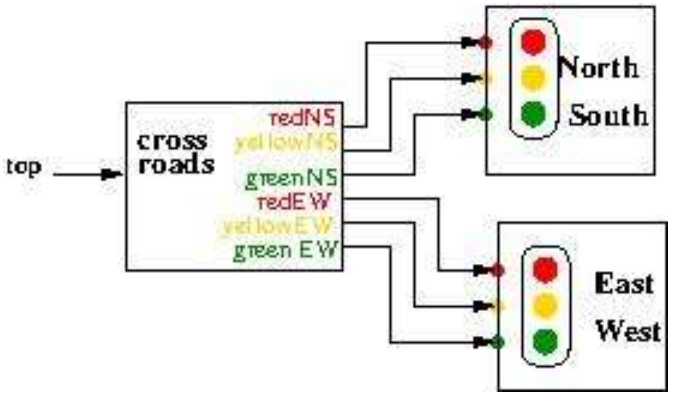
As a training exercise, define in CLEM a module tictac that emits tic the first instant and tac the second instant and indefinitely do it again. Put the design of the module in a “tictac.le” file and simulate it with the simul feature of clem. Then, implement the same module as an explicit Mealy machine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

DESIGNING A CROSSROADS SYNCHRONOUS MONITOR WITH SCADE

SPECIFICATION



TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP



A crossroads with two orthogonal roads (east-west and north-south) is controlled by two traffic lights. Each traffic light works as follows: at each instant, the traffic light manages three Boolean outputs: red, yellow, green.

These three outputs are exclusive and they are true only following the sequence:

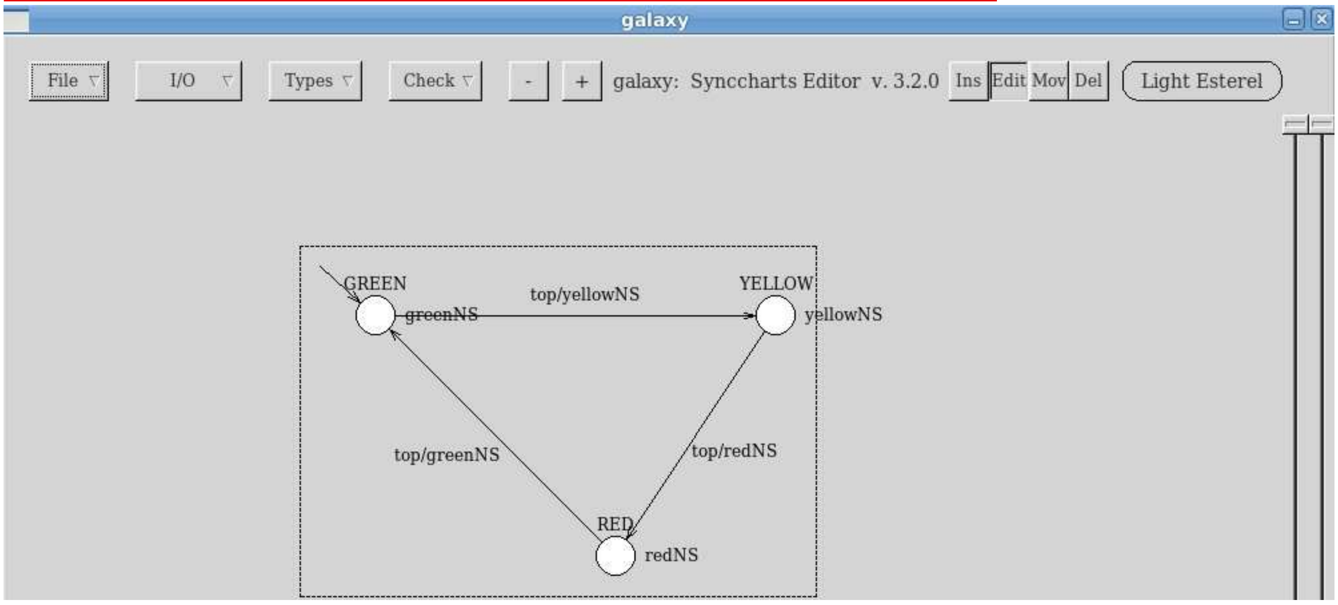
red → yellow → green

We consider that the duration of each light is the duration of the traffic light clock, and we will consider that this clock send a *top* signal .

TRAFFIC LIGHT MONITOR

As it is mainly a controller, it is recommended to rely on explicit Mealy machine to carry out the design. The following diagram shows a possible implementation of a traffic light behavior with GALAXY. There is an input top, we assume that the traffic light switch from a light to another according to this top signal. There are three outputs: greenNS, redNS and yellowNS.

Thus, we design a state machine with 3 states: Green, Orange and Red. Transitions are triggered with top and in each state the output variables are sustained. This traffic light starts in Green state (we call it TrafficLightNS).

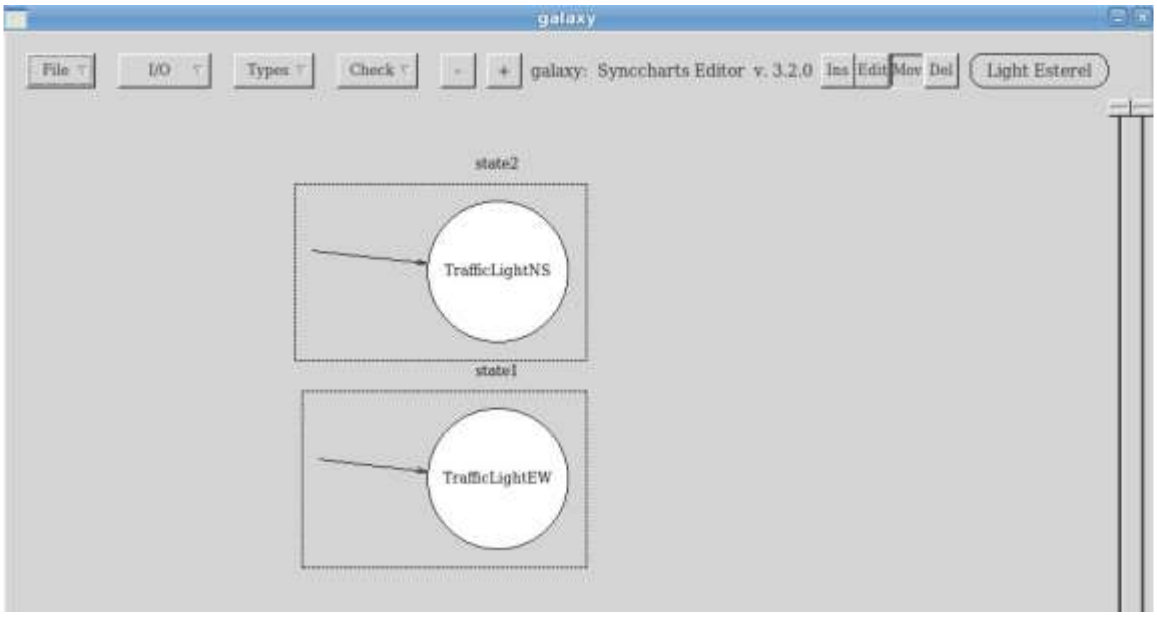


TrafficLightNS design in GALAXY

CROSSROADS SYNCHRONOUS MONITOR

SPECIFICATION

The idea we could have is to put two traffic lights in parallel, one starting in Green state and the other starting in Red state. For instance we could design the following hierarchical Mealy machine:



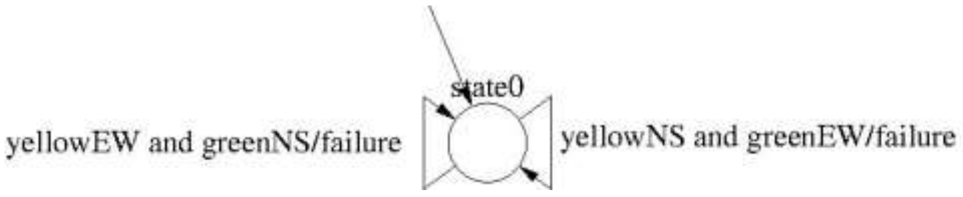
The CrossRoad specification in GALAXY consists in two automata in parallel. Each automaton is hierarchical and has only one state calling (with a Run operation) respectively TrafficLightNS and TrafficLightEW.

Design and simulate your own crossroads controller.

VERIFICATION

Before introducing this monitor as a synchronous monitor in a WComp application, we want to verify that it has no incorrect behaviors. The property we want to prove is: we don't have greenNS and yellowEW or greenEW and yellowNS.

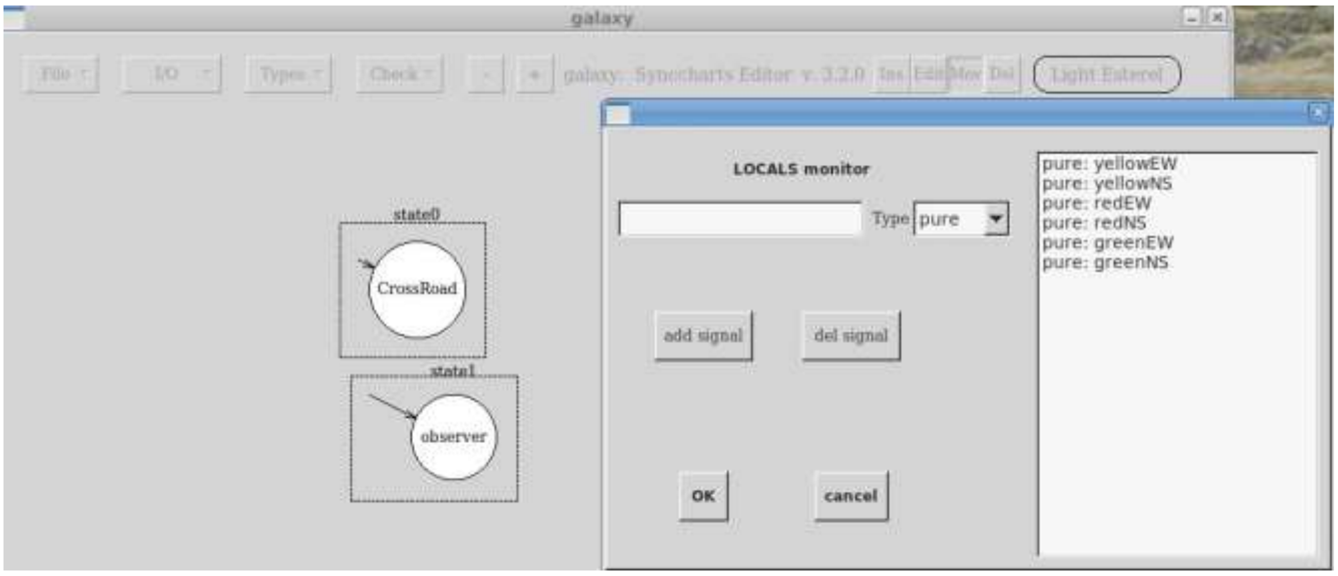
To achieve the proof we rely on blif_check and the observer technique is used. Thus, we define an observer of the property. According to observer approach, we describe the property in GALAXY, as a Mealy machine:



The observer (named observer.gal) listen four inputs: yellowEW, greenEW, yellowNS and greenNS and emits a failure signal when the property fails.

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 verifCrossRoad) which is the parallel composition of the CrossRoads design and the observer:



pure: yellowEW
pure: yellowNS
pure: redEW
pure: redNS
pure: greenEW
pure: greenNS

verifCrossRoad is the parallel of CrossRoad and the observer. It input signal is top, it output signal is failure and the output signals of CrossRoad module become locals.

Now to validate the property, we first simulate verifCrossRoad in order to get a BLIF file. Then, relying on blif_check we can verify if the property is true or not.

If the property is false, you can study the generated counter example to correct your design until it holds. Then, the CrossRoad monitor can be considered as validated and we can define the Bean for it.

C# BEAN CODE GENERATION

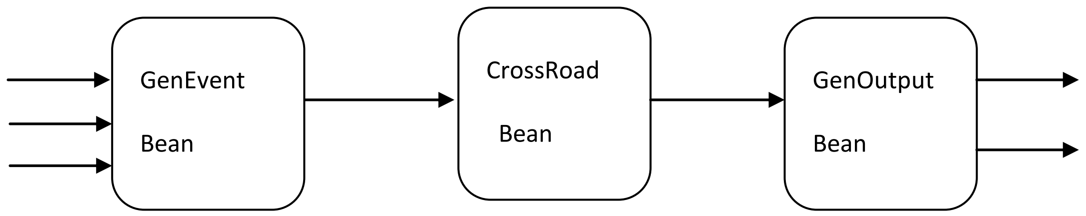
To generate the C# code for CrossRoad monitor, just call clem and save the file in LEC format. Then a call to clef allows generating the C# code (CrossRoad.cs).

The generated file will define a class CrossRoad in order to generate a CrossRoad Bean. This class defines mainly two methods: **CrossRoad_reset_automaton()** and **CrossRoad_automaton(string).** According to the synchronous approach, mentioned previously, CrossRoad 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 goal of CrossRoad_automaton method. First, this method takes as input a string of serialized input events, which are un-serialized according to a grammar (called grammaireA). The grammar and serialization and un-serialization methods are defined in **GrammaireA.cs** file. Then the sorted equation systems are evaluated with input events set either to true or false according to the present input events detected from the input string. Then, starting from this input setting the computation of registers next value and outputs is done by propagation. Finally, the serialization (always according to GrammaireA) is done and the output event is fired as a string encoding the output events.

Considering the method CrossRoad_reset_automaton, its goal is to set the register values to their initial values and to set the output absent. It is a reset, to put back the automaton in its initial state.

CROSSROAD IN WCOMP

To design a complete application in WComp, we must compile CrossRoad.cs and generate the dll to get a new Bean CrossRoad:



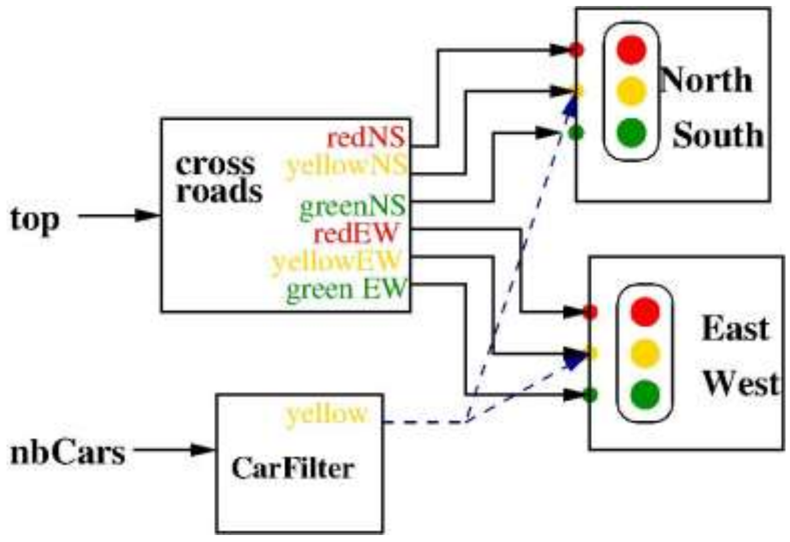
The CrossRoad Bean implements a synchronous monitor which listens to data arriving in an asynchronous way. It is why the entry and result of the CrossRoad_automation method is a string, i.e. a serialization of a set of events according to grammar grammaireA. Now, to design an application, you need to rely on a WComp component allowing to listen to asynchronous events and to send a serialization of events representing logical instant (see the lecture). To help you to design such a component, we provide an example of such a component (GenEvent Bean defined in GenEvent.cs file). This Bean listens to asynchronous events of type MyEvent (defined in MyEvent.cs) and a method **addEvent** put them in a circular buffer. It also implements two sending policies to build the synchronous instant (polAverage and polOccurrence). The first one makes the average of event respective data values while the second sends the events in the buffer when it receives a second occurrence of an event already in the buffer. Then a serialization is done before sending the event to the synchronous monitor for both policies.

In a reverse way, a component to generate asynchronous events coming as output of the CrossRoad Bean is needed. To help you, we also provide an example of such a component in GenOutput.cs. This component implements two sending policies: polSend1 which send all the events included in the string outputted by CrossRoad Bean; polSend2 which send the list of events.

You can be inspired by these examples of entry and output generators to implement the CrossRoad in WComp. For instance, you could define in WComp a solution which builds three Beans: a Bean for the entry generator, one for the output generator and a bean for the synchronous monitors. Then you can make an application in WComp.

You will find in the archive “GenInOut.zip” the following files: GenEvent.cs, GenOutput.cs, CircularBuffer.cs, MyEvent.cs, GrammaireA.cs which can help you to design your own generators.

COMPOSITION IN WCOMP



Continuing with the CrossRoads use case, we add a CarFilter component to the design to take into account the traffic density. This component listen a signal nbCars present when the number of cars is less than a given bound. Then a signal yellow is emitted and this latter preempts the usual running of the CrossRoads component and the two respective traffic light (NS and EW) are maintained to “yellow” (yellowNS and yellowEW are sustained and the others output signals are not). Thus, CarFilter is a new synchronous monitor composed with the CrossRoads one. As a result, the yellow lamp of each traffic light has multiple accesses. Then, you must define a constraint monitor to define correctly what happens when the signal yellow of CarFilter is present. Here is an example of the main composition module.

```
module CrossRoadComp:
Input: top, nbCars;
Output: greenNSC, yellowNSC, redNSC, greenEWC, yellowEWC, redEWC;
```

```
Run:
".": CrossRoad_v1: CrossRoad_v1;
"." : CarFilter: CarFilter;
"." : CrossRoadConstraint : CrossRoadConstraint;

local greenNS, yellowNS, redNS, greenEW, yellowEW, redEW, yellow

{
run CrossRoad_v1
||
run CarFilter
||
run CrossRoadConstraint
}

end
```

The CrossRoadComp monitor listen top and nbCars and its outputs are the six lamps of the cross road. It is the parallel composition of:

1. CrossRoad_v1 monitor which has been previously validated
2. CarFilter monitor which sends a yellow output when nbCars is true
3. CrossRoadConstraint which implements the constraints function:

    greenNSC = greens and not yellow

    redNSC = redNS and not yellow

    yellowNSC = yellowNS or yellow

    greenEWC = greenEW and not yellow

    redEWC = redEW and nor yellow

    yellowEWC = yellowEW or yellow

   This monitor can be implemented as an explicit Mealy machine with GALAXY or an implicit one as well.