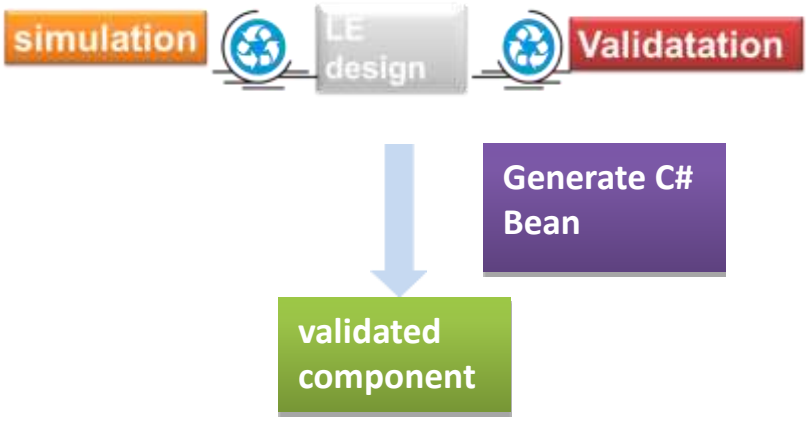
# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

2011-2012

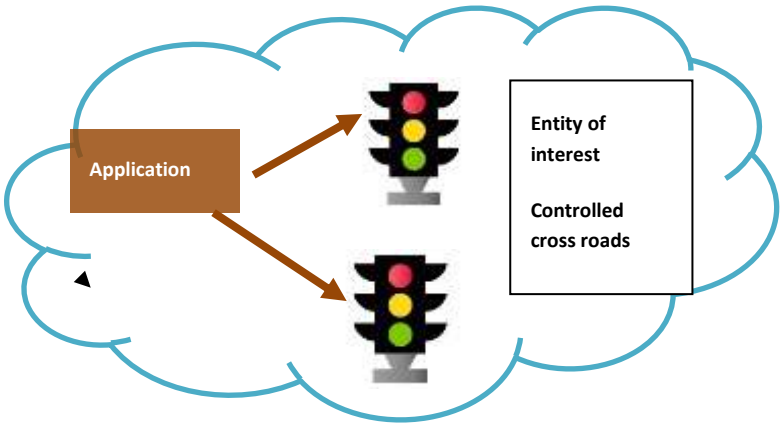
## INTRODUCTION

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




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

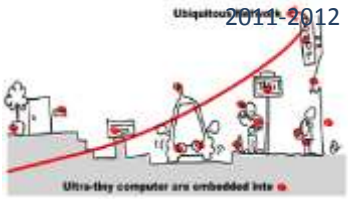
According to our validation concern, we will follow the methodology detailed in the lecture. Thus, we will describe a **constraint component** to verify the behavior of our traffic lights manager and then introduce it as a validated component in WComp.




Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP



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 monitors.

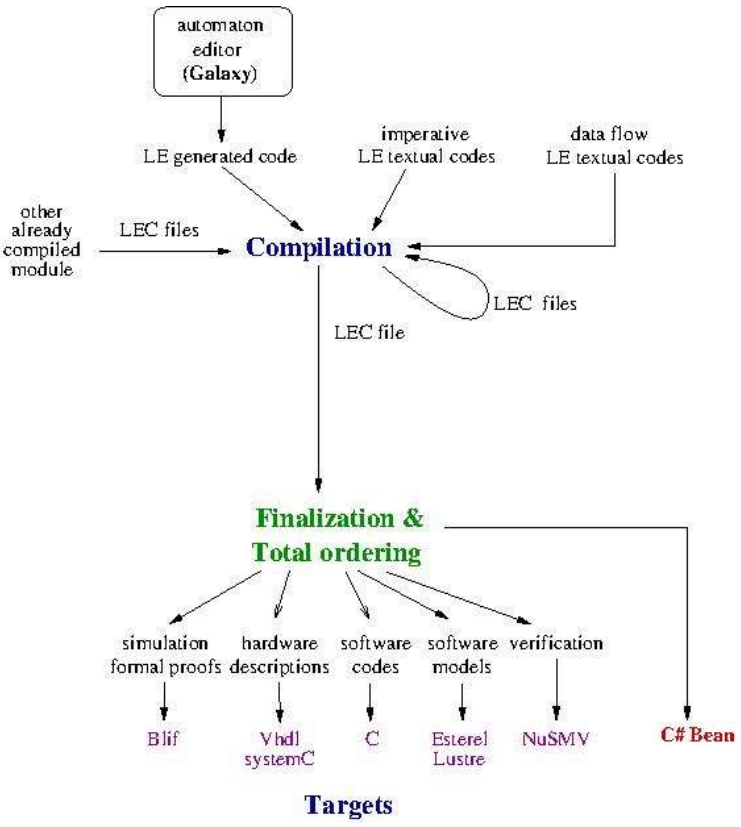
## THE CLEM TOOLKIT

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

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 and C# code for WComp:

1. BLIF: this format is the entry format for simulation and verification tool (respectively blif_simul and blif_check). 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.

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




THE CLEM TOOLKIT

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

2011-2012

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 and C# code to plug component in WComp.
- blif_simul: is the simulator. It is called with the simulation option 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 both constraint controllers and constraint 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;
```

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

```
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.

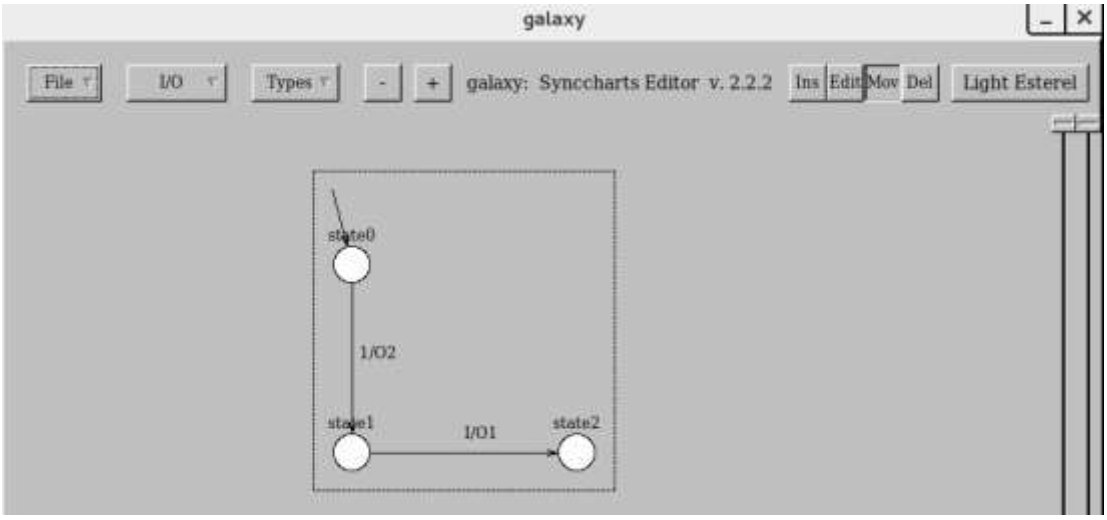
- **run mod** : to call the module **mod**. This instruction put into practice the modular compilation of CLEM. You can compile the module **mod** and save the LEC code generated in a **file mod.lec** and then reload this code when compiling a main module containing this **run mod** instruction. If the 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.

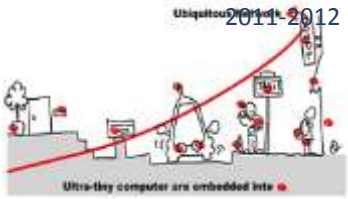
## EXPLICIT MEALY MACHINE

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, 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:

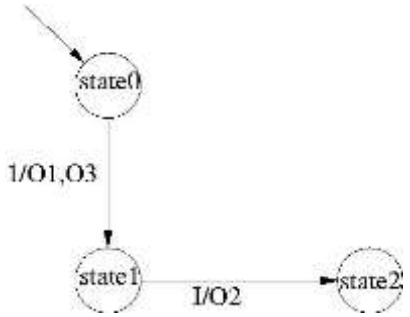
1. **Ins**: to define states and transitions. A mouse click creates a state and a mouse drag from a state to another draws a transition. A click in a state draws a circular transition from and to this state. To define the initial state , just drag a transition from the background to the state.
2. **Edit**: 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.

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

## 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

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”)

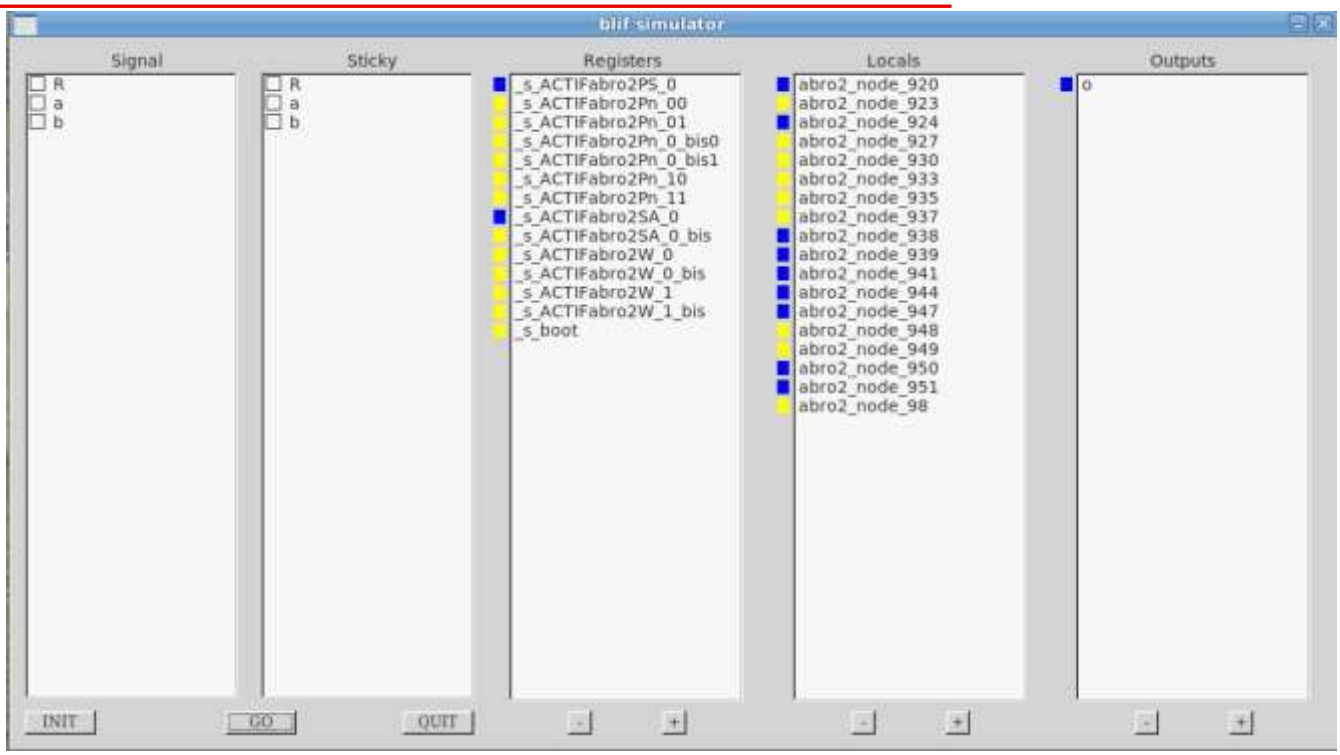
In this lab, we will use the pure simulator:

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

2011-2012




**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. To run the blif_check 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 in an ".rst" file. This counter example shows a path leading to a state where the property fails.

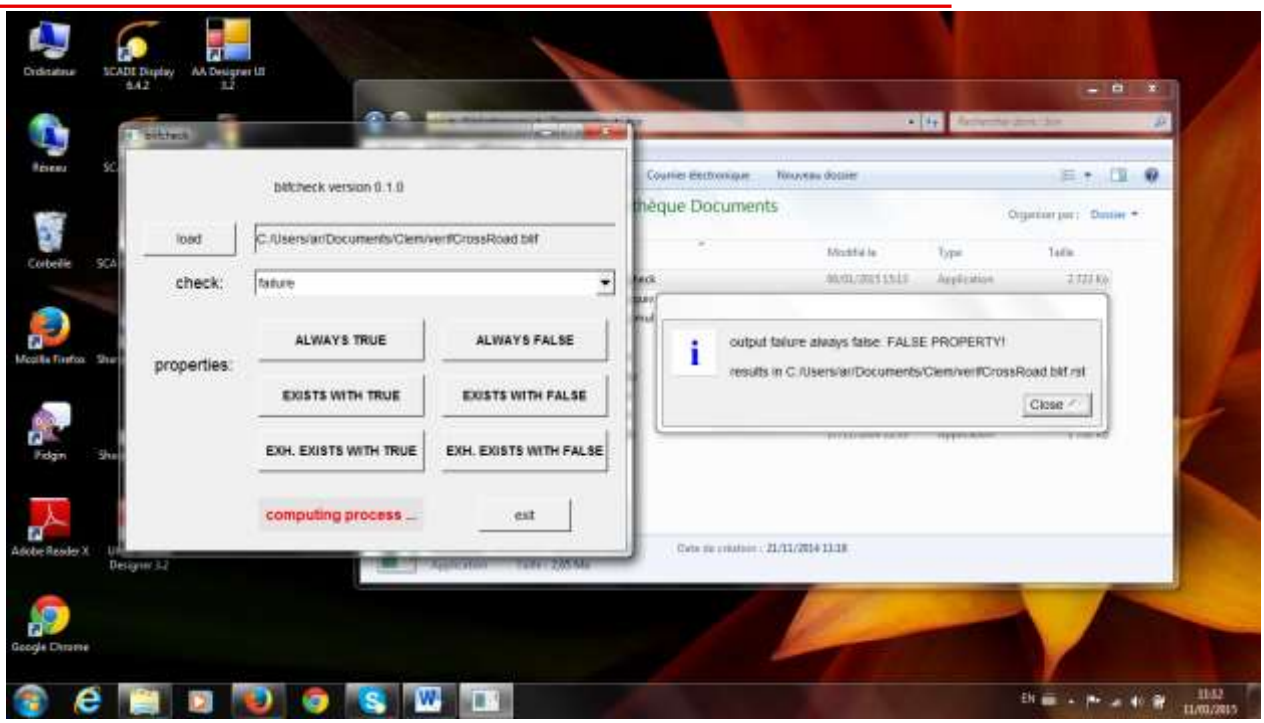
This model checker checks for the presence or the absence of an output signal. Thus, it must be used with the observer technique, if you want to check a more complex safety property.

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP






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 CONSTRAINT COMPONENT WITH CLEM

## SPECIFICATION

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP


A cross roads 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:

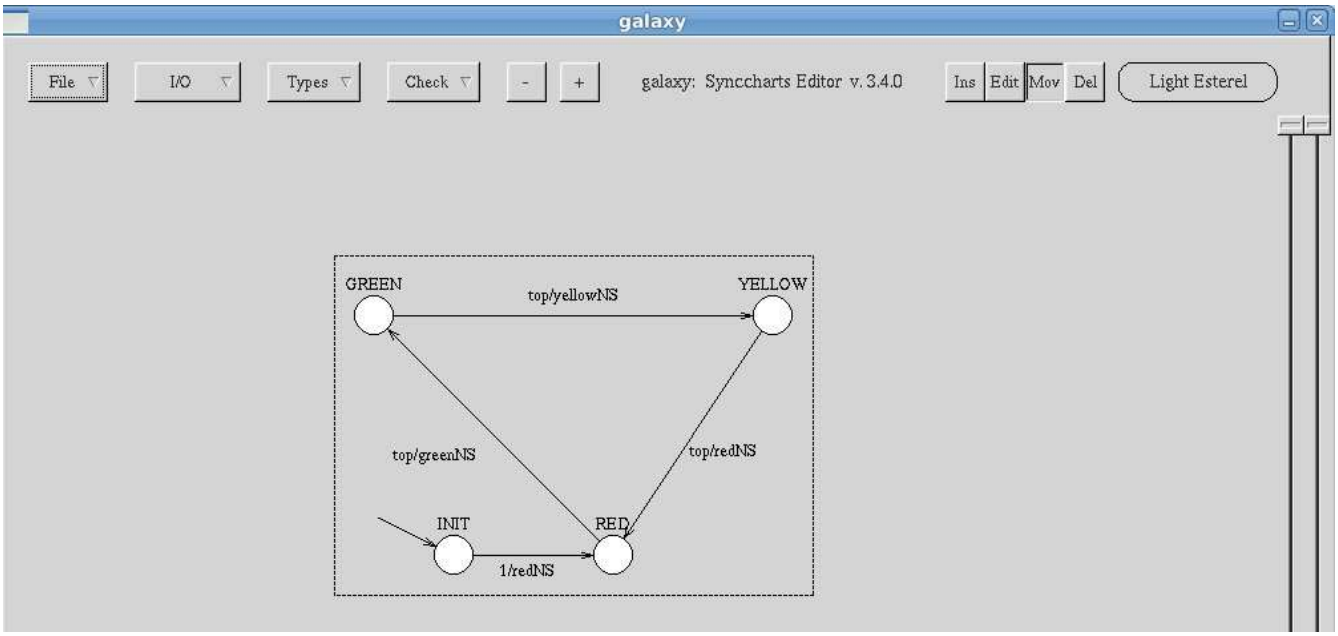
red → yellow → green

The cross roads must respect two constraints: (1) it works only if both traffic lights correctly work as mentioned before and (2) it highlights the respective traffic lights in a consistent way (no green lights in same time for instance).

## TRAFFIC LIGHT MODEL

As it is mainly a controller, it is recommended to rely on explicit Mealy machine to carry out the design. For a first attempt to implement a traffic light, we will 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. 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 switches 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 4 states: INIT, GREEN, YELLOW and RED. Transitions are triggered with top in GREEN, RED and YELLOW states and the corresponding color is emitted. To make this traffic light starting in red, we add an initial state and an initial transition triggered when the Mealy machine starts an emitting a redNS (1/redNS).




TrafficLightNS design in GALAXY

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

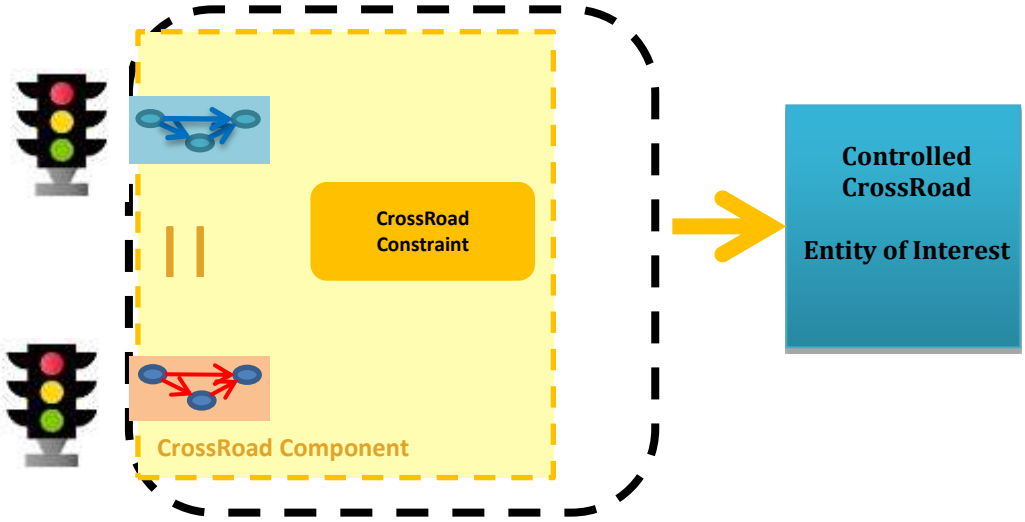
# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP


## TO DO

Design your own trafficLightNS model and simulate it.

## CROSSROADS CONSTRAINT COMPONENT

Following the approach described in the lecture, we must design a constraint component made of the respective models of the two traffic lights and a constraint controller specifying the constraints on the respective outputs of each traffic light to get controlled cross roads.




The Crossroad constraint controller

## CROSS ROADS MODELING

The first idea we could have is to define two traffic light models (TrafficLightNS and TrafficLightEW), the first one starting by emitting a RedNS and the second one a GreenEW. With this approach, we think that we will respect the constraint of consistency between the respective highlighted lights of TrafficLightNS and TrafficLightEW.

According to the figure above, we also need to design a constraint controller. In this application, the constraint controller is simple. We just want that it verifies that the cross road works only if the two traffic lights do. As this constraint controller has in charge to tell us how the respective outputs of each traffic light are combined, it is more natural to use a CLEM Mealy machine to express it. Nevertheless, a Galaxy design is also possible. The following CLEM code implements a possible constraint controller for the cross road:

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

```
module CrossRoadConstraint:

Input: greenNS, redNS, yellowNS, greenEW, redEW, yellowEW;

Output: greenNSC, redNSC, yellowNSC, greenEWC, redEWC, yellowEWC;


local isNS, isEW

{

   Mealy Machine

    isNS = greenNS or redNS or yellowNS;

    isEW = greenEW or redEW or yellowEW;

    greenNSC = greenNS and isEW;

    redNSC = redNS and isEW;

    yellowNSC = yellowNS and isEW;

    greenEWC = greenEW and isNS;

    redEWC = redEW and isNS;

   yellowEWC = yellowEW and isNS;

}

end
```

**This code represents a Boolean equation system that computes the values of CrossRoad outputs from the outputs of TrafficLightNS and TrafficLightEW. Moreover, two local variables are introduced: IsNS is true is TrafficLightNS works (meaning that one of its lights is on) and isEW acts similarly for TrafficLightEW.**

Now, the design of the overall constraint component is just the parallel composition of the three already defined synchronous components (see figure above):

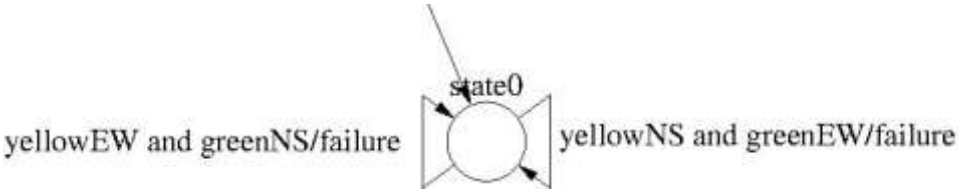
```
module CrossRoad:

Input: top;

Output: greenNSC, redNSC, yellowNSC, greenEWC, redEWC, yellowEWC;


Run:

"C:\Users\ar\Documents\TP\2016\CLEM" : TrafficLightNS : TrafficLightNS;

"C:\Users\ar\Documents\TP\2016\CLEM" : TrafficLightEW : TrafficLightEW;
```

```
"C:\Users\ar\Documents\TP\2016\CLEM" : CrossRoadConstraint : CrossRoadConstraint;


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

{

   run TrafficLightNS

 ||

   run TrafficLightEW

 ||

   run CrossRoadConstraint

}

end
```

## VERIFICATION

Before introducing this constraint component in a WComp application, we want to verify that it has no incorrect behaviors and if the consistency between lights is ensured. Thus we want to formally prove that: we don't have greenNS and yellowEW or greenEW and yellowNS, for instance.

To achieve the proof we rely on blif_check and the observer technique is used. Thus, we define an observer of the property. The observer corresponding to the property can be designed in GALAXY as follows:




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 a CLEM 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 CrossRoad design and the observer:

```
module verifCrossRoad:

Input: top;

Output: failure;

Run:

"C:\Users\ar\Document\TP\2016\CLEM": CrossRoad : CrossRoad;

"C:\Users\ar\Document\TP\2016\CLEM": observer : observer;


local greenNSC, redNSC, yellowNSC, greenEWC, redEWC, yellowEWC

{

   run CrossRoad

 ||

   run observer

}

end
```

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 generate the BLIF format of this module with CLEM. Then, relying on blif_check we can exhaustively verify if the property holds in testing if failure 'exists with true". If the result is true, it means that there is some path where failure is true (and blif_check shows you such a path). It also means that In this case, you should improve your design until the property will hold.

## TO DO

1. Implement your own CrossRoad , simulate and verify the property with blif_check.
2. if the property is not true (which is the case with the previously mentioned design), you must correct your design in improving TrafficLightNS and TrafficLigthEW and also in changing the constraint controller.

## ADDING INHIBITORS

To complete the design, we must add inhibitors for TrafficLIghtNS and TrafficLIghtEW that will take into account the disappearance of one of them. These inhibitors allow changing the behavior of the constraint component without changing the design. For instance, to complete the CrossRoad specified here, we introduce in the design two signals:

inhibNS and inhibEW, the role of which is to inhib the outputs of the missing traffic light when they are present. First, these inhibitors are introduced in the constraint controller:

```
module CrossRoadConstraint:

Input: greenNS, redNS, yellowNS, greenEW, redEW, yellowEW, inhibNS, inhibEW;

Output: greenNSC, redNSC, yellowNSC, greenEWC, redEWC, yellowEWC;

local isNS, isEW, g_NS, r_NS, y_NS, g_EW, r_EW, y_EW

{

   Mealy Machine

    g_NS = greenNS and not inhibNS;

    r_NS = redNS and not inhibNS;

    y_NS = yellowNS and not inhibNS;

    g_EW = greenEW and not inhibEW;

    r_EW = redEW and not inhibEW;

    y_EW = yellowEW and not inhibEW;

    isNS = g_NS or r_NS or y_NS;

     isEW = g_EW or r_EW or y_EW;

     greenNSC = g_NS and isEW;

     redNSC = r_NS and isEW;

     yellowNSC = y_NS and isEW;

     greenEWC = g_EW and isNS;

     redEWC = r_EW and isNS;

    yellowEWC = y_EW and isNS;

}

end
```

Then, we also complete the overall design:

```
module CrossRoad:

Input: top, inhibNS, inhibEW;

Output: greenNSC, redNSC, yellowNSC, greenEWC, redEWC, yellowEWC;
```

```
Run:
"C:\Users\ar\Documents\TP\2016\CLEM" : TrafficLightNS : TrafficLightNS;
"C:\Users\ar\Documents\TP\2016\CLEM" : TrafficLightEW : TrafficLightEW;
"C:\Users\ar\Documents\TP\2016\CLEM" : CrossRoadConstraint : CrossRoadConstraint;

local greenNS, redNS, yellowNS, greenEW, redEW, yellowEW
{
   run TrafficLightNS
 ||
   run TrafficLightEW
 ||
   run CrossRoadConstraint
}
end
```

## TO DO

Complete your correct crossroad to take into account inhibitors and simulate it to verify that inhibitors works as intended.

## C# BEAN CODE GENERATION

To generate the C# code for CrossRoad component, just call CLEM and generate 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.

## TO DO

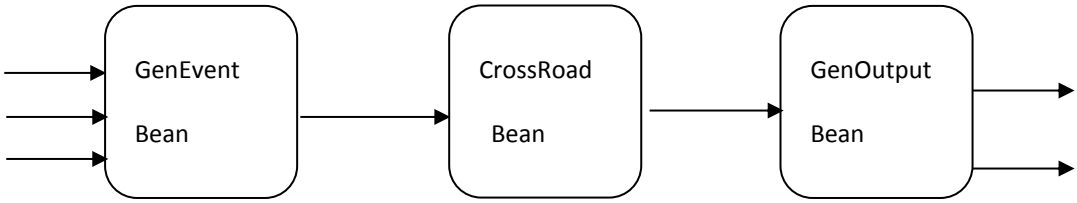
# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP



Generate CrossRaod C# Bean

## CROSSROADS 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 the behavior of our CrossRoad component with its constraints (represented previously using automata). It treats synchronous data, but, the communication of data in WCOMP is done in an asynchronous way. To resolve this issue, we need an asynchronous/synchronous transformer, composed of a synchronizer and a desynchronizer. The synchronizer (called “GenEvent Bean”) is placed before the CrossRoad component; it will group asynchronous events into synchronous instants, serialize them according to a grammar (in our work we will use “grammaireA”) to transform them into one string, and will send them to the CrossRoad component, according to grouping predefined policies. The crossroad component will receive these data, make its processing and then send the results (which are also synchronous data) as one string also to the desynchronizer. The desynchronizer (called “GenOutput Bean”) is placed after the CrossRoad, it will receive the synchronous data from the constraint component and immerse them into asynchronous environment.

### TO DO

Introduce your validated CrossRoad.cs component in a designed and connect it to two instances of traffic lights following the process described in the lecture. Below, we detail this process using the C# Bean of WComp libraries dedicated to implement a synchronizer and a desynchronizer. Of course, it is just an example of use and you can choose another implementation. In this description, the CSharp cross road file we have generated is named: CrossRoadv1.cs.

### PROCESS TO DESIGN YOUR CROSSROADS COMPONENT IN WCOMP

To design your CrossRoad application, you need to rely on WComp components which will allow listening to asynchronous events and sending a serialization of events representing logical instant (see the lecture).

In this part we will use the files below:

- Input_out_generator.dll
- GrammaireA.cs
- MyEvent.cs
- CrossRoad.cs (which you will generate using Clem)
- TrafficLight.exe

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

**Input_out_generator.dll:**

This file should be placed under: Documents/Wcomp.NET/Beans. It will help you to obtain the beans:

- Events
- GenInput
- GenOutput
- CrossRoadDataTransformer

**GrammaireA.cs / MyEvent.cs / CrossRoad.cs:**

**1/ Create a new WcompBeanSolution and copy these files into it.**

The file **MyEvent.cs** contains the class in which we define the structure of our events.

The file **GrammaireA.cs** contains the class which defines how the serialization and deserialization of the events is done.

The file **CrossRoadv1.cs** represents the behavior of our crossRoad and the constraint that should be respected to ensure its correct processing.

**2/ you should also add these references below:**

- Beans
- Util
- System
- System.Data

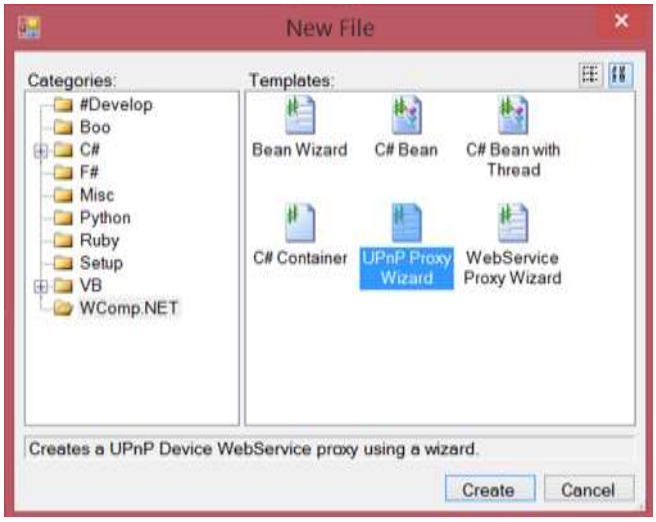
**3/ Compile the project**:

If there is not any error a new dll file will be created in **C:\Users\isarray\Documents\SharpDevelop Projects\your_Solution_Name\bin\Debug**

You should copy this file under **Documents/Wcomp.NET/Beans** (don't forget to close SharpDevelop before that)**.**

**4/Activate the TrafficLight.exe and open SharpDevelop again.**

**5/ Choose File - > new - > File - > WComp.NET - > UPnP Proxy Wizard**




**6/ Select TrafficLight and then Next and Finish**


**7/ Choose File - > new - > File - > WComp.NET - > C# Container**

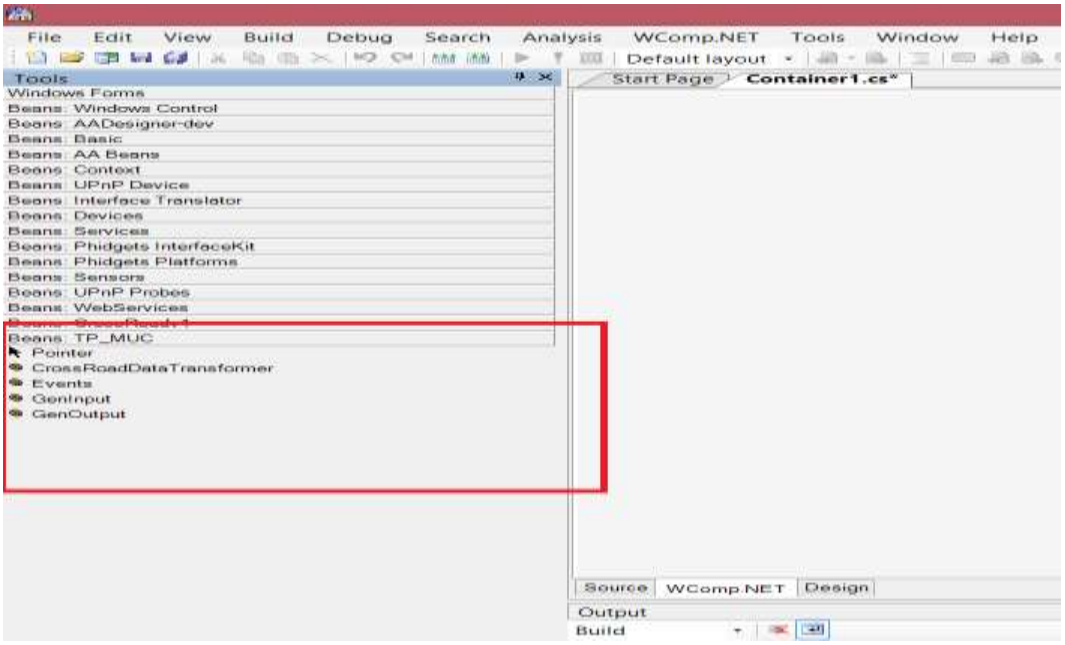
**8/ We will use the beans below:**

- **In the category : TP_MUC** :
  - **Events**
  - **GenInput**
  - **GenOutput**

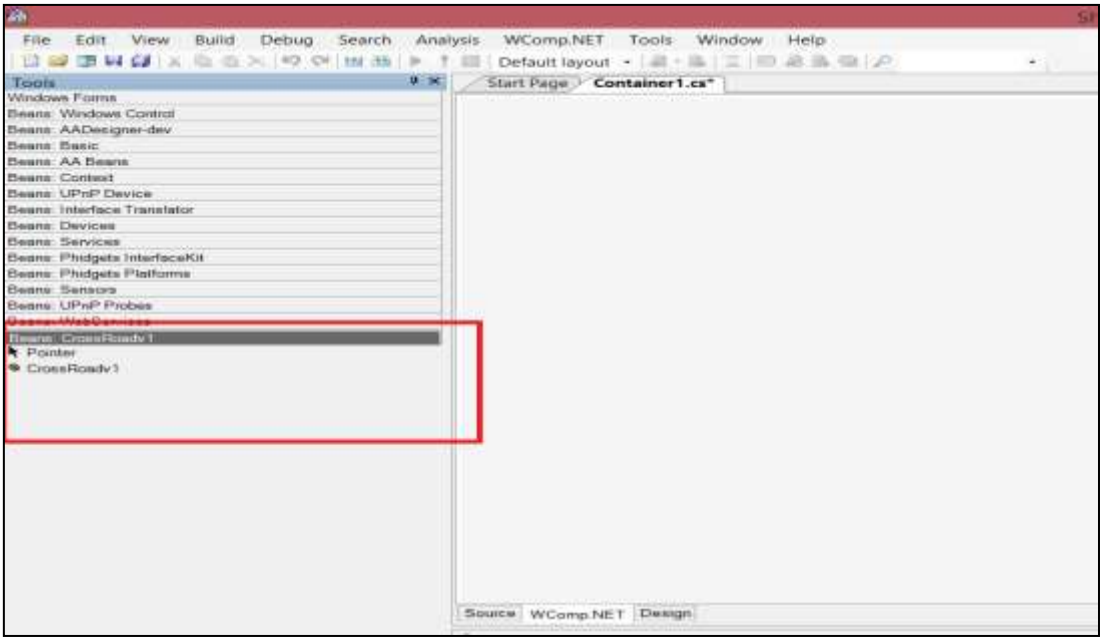
Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

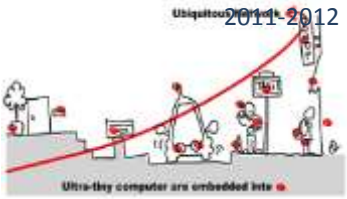
- **CrossRoadData Transformer**



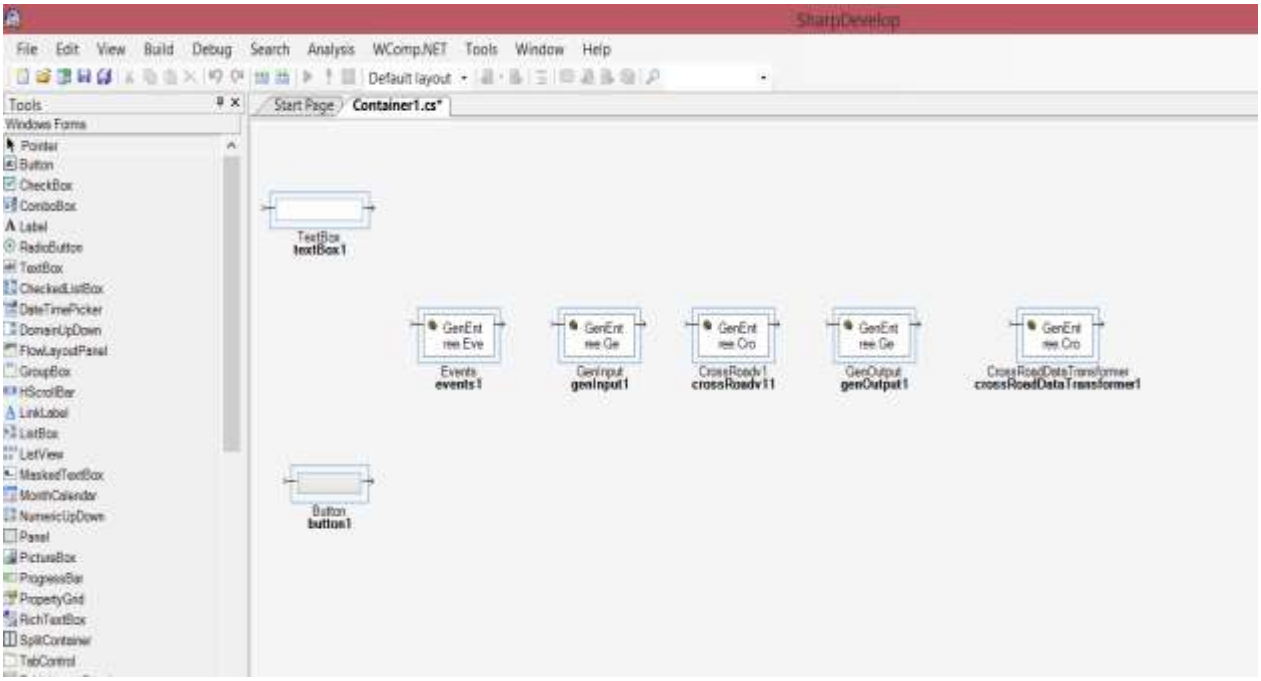

- **In the category CrossRoadv1**
  - **CrossRoadv1**




- **In the category UPnP Device**
  - **TrafficLight**

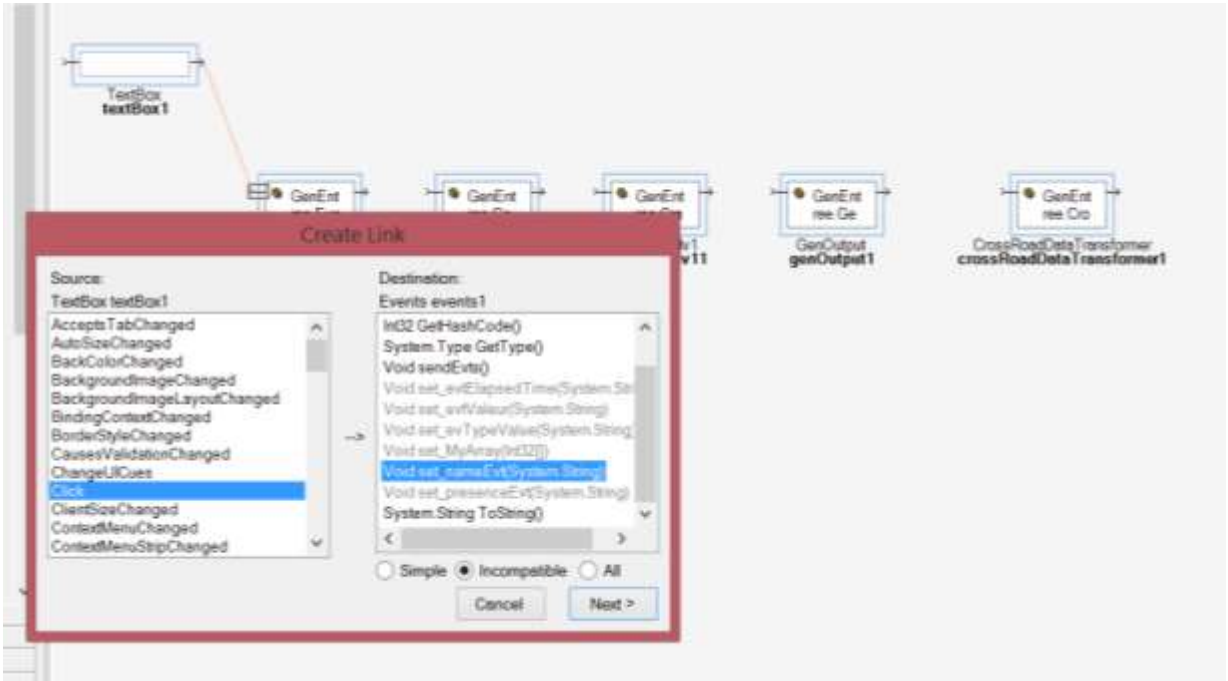


Our CrossRoad needs the event named “top” to execute its processing, so we added a TextBox to mention the name of events sent to the crossroad and a button to create these events.




### 9/ create links between beans

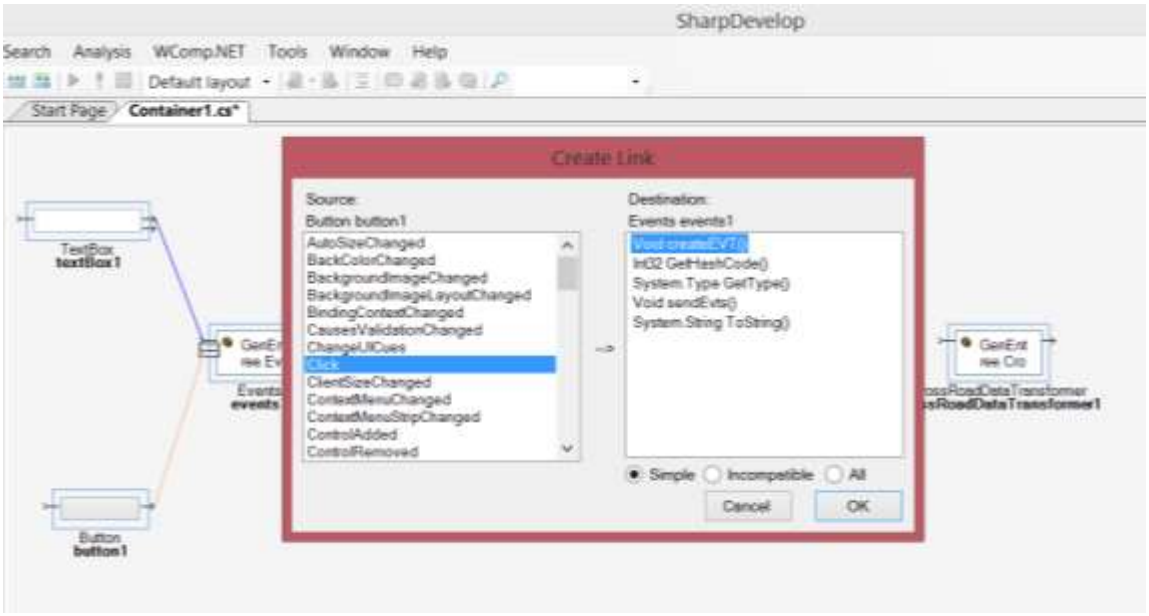
1- Link 1 : TextBox -> Events



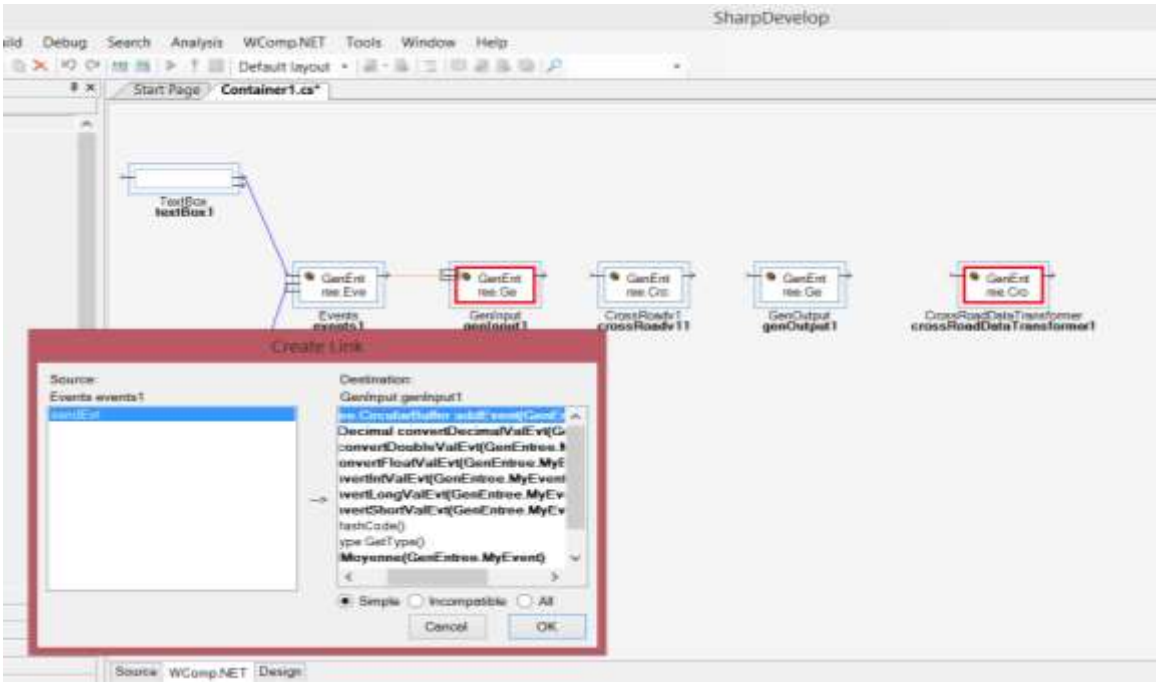

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

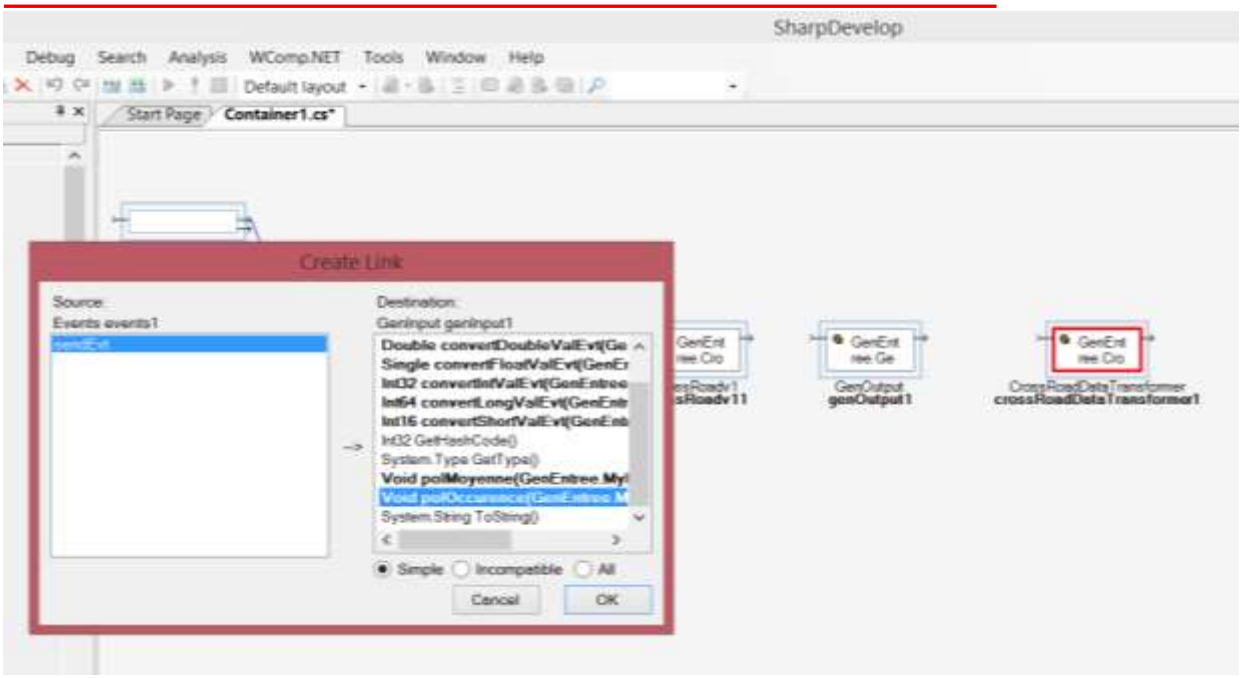
2- Link2 : Button -> Events




3- Link3 : Events -> GenInput

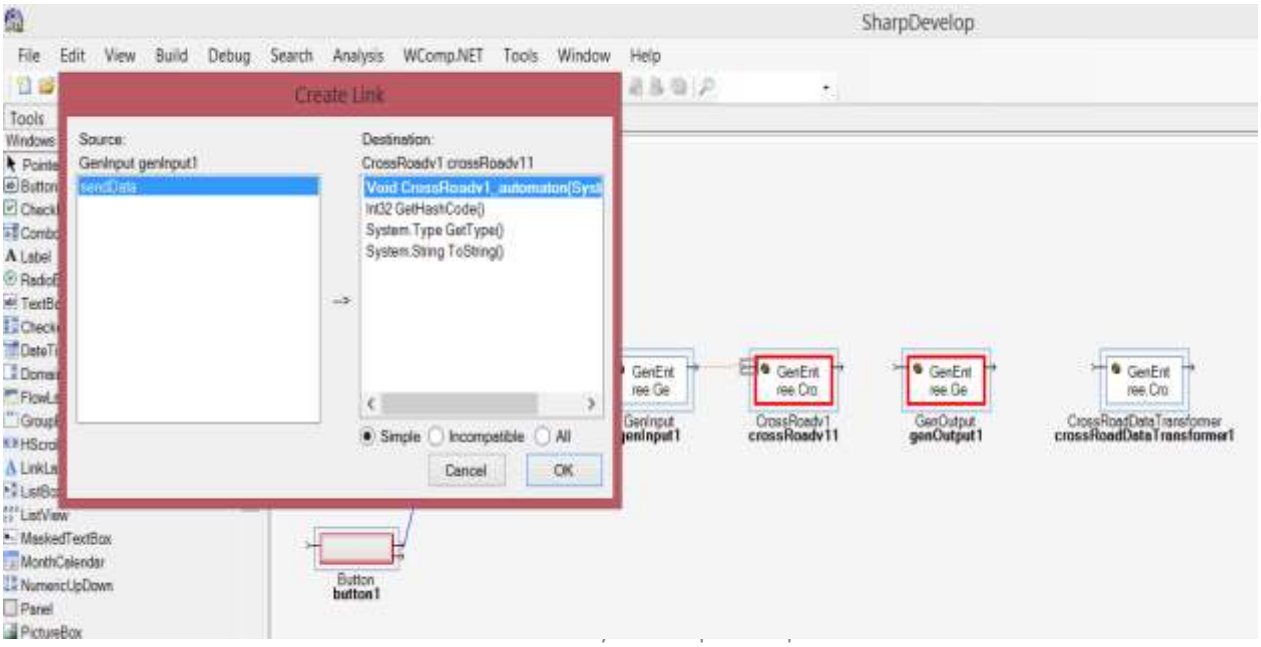



The method “addEvent” will add the event to a circular buffer.



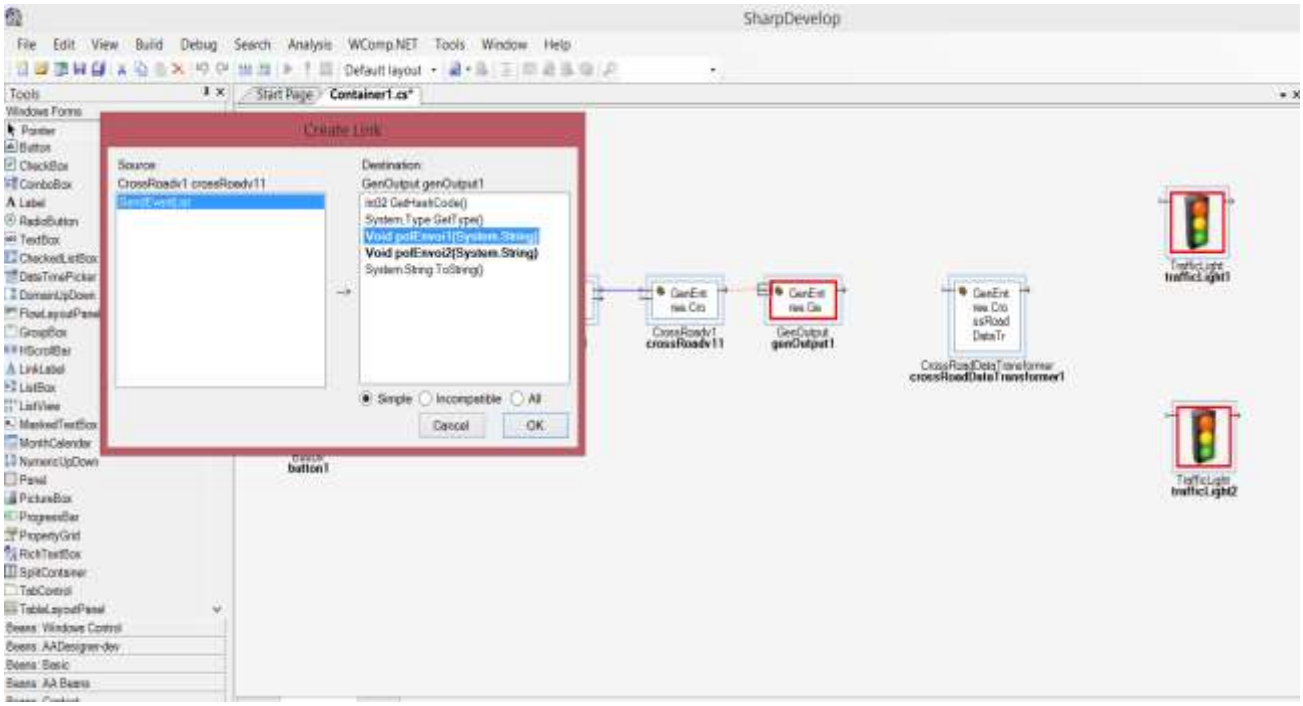

Here we choose the policy to send the events, we choose the occurrence policy, which means that when we will have an occurrence of the event “top” all the events stored in the circular buffer (before this occurrence) will be serialized and sent to the cross road.

4- Link4: GentInput -> CrossRoadv1



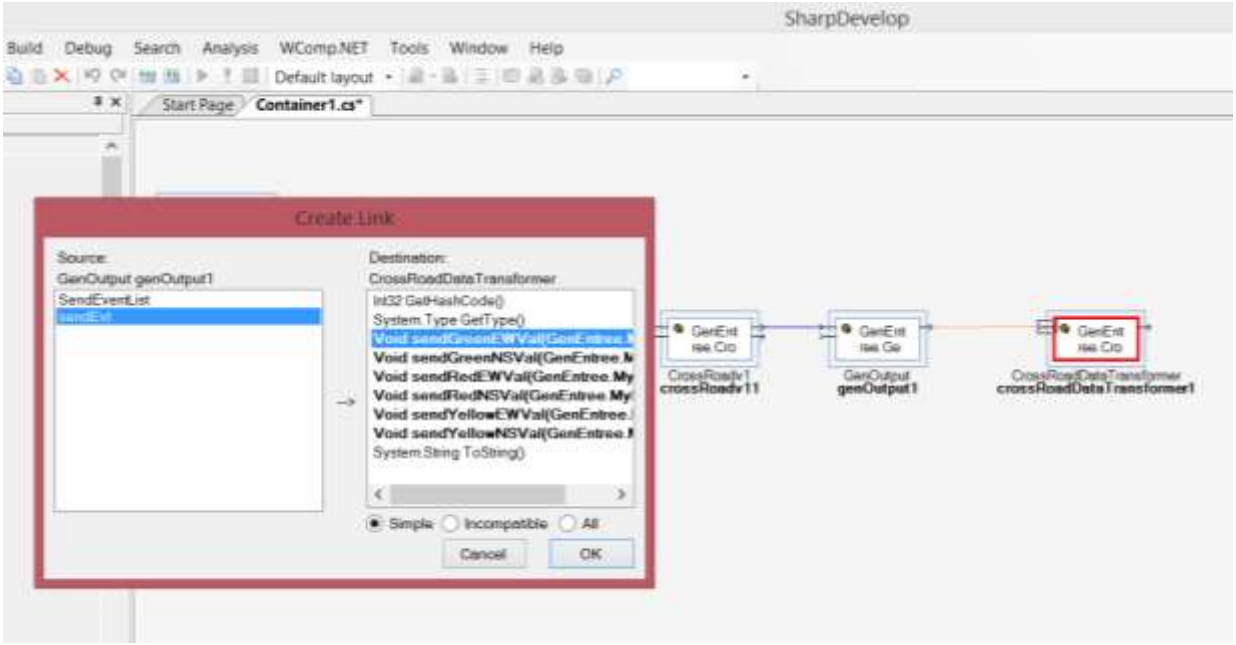

annie.ressouche@inria.fr, ines.sarray@inria.fr

5- Link5 : CrossRoadv1 -> GenOutput



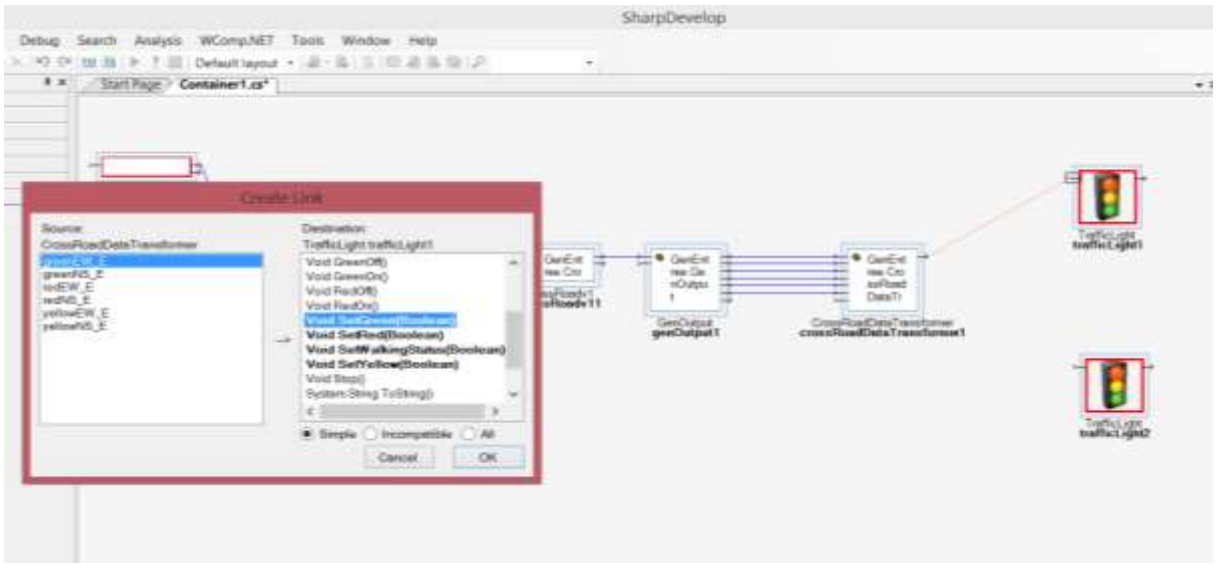

The GenOutput bean is the component which will generate asynchronous events coming as output of the CrossRoad. This component implements two sending policies: polSend1 which will send all the events included in the string outputted by CrossRoad Bean (each event will be sent separately); polSend2 which will send the list of events. Here we choose polSend1

6- Link 6 : GenOutput -> CrossRoad Data Tansformer



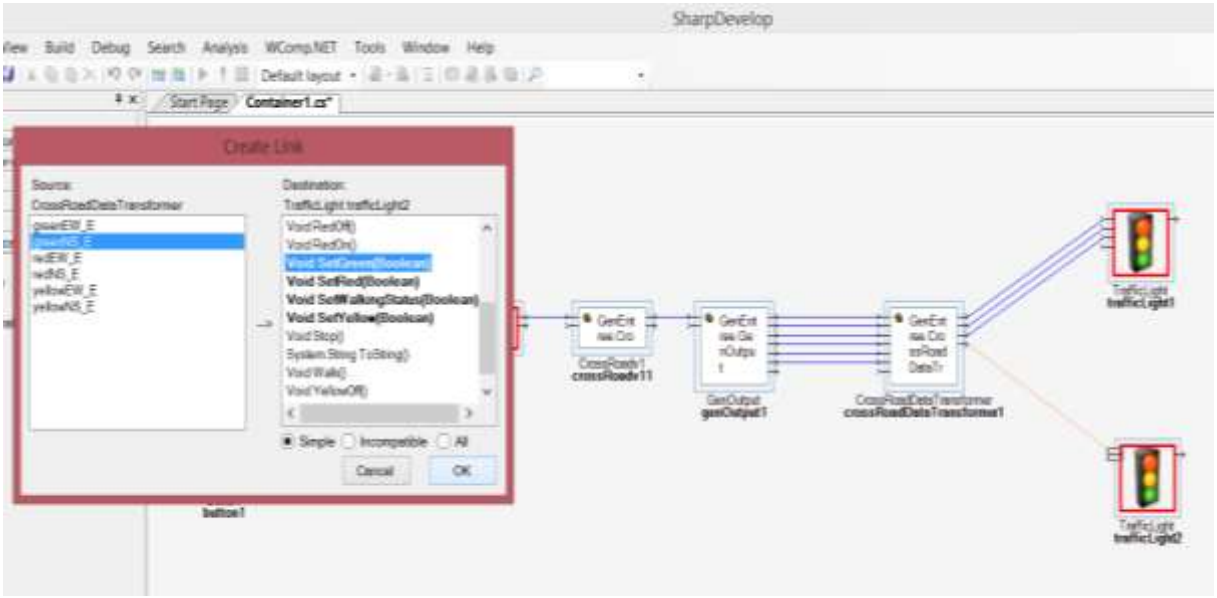
You should do the same work with all the methods (**sendGreenEWVal(), sendGreenNSVal(), sendRedEWVal(), sendRedNSVal(),sendYellowEWVal(), and sendYellowNSVal()** ); these methods will receive the events, make a test to check if it is the needed event or not and then send the right data to the traffic lights.

7- Link7 : CrossRoad Data Tansformer -> TrafficLight1




You should do the same work for redEW_E and yellowEW_E

8- Link8 : CrossRoad Data Tansformer -> TrafficLight2

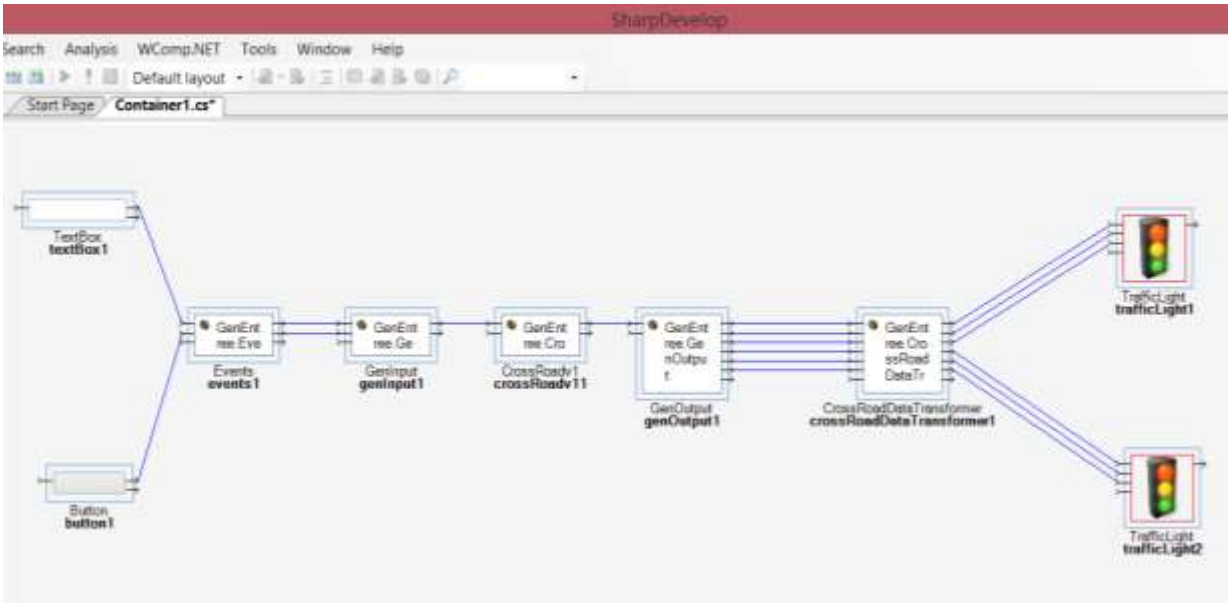



Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

You should do the same work for redNS_E and yellowNS_E.

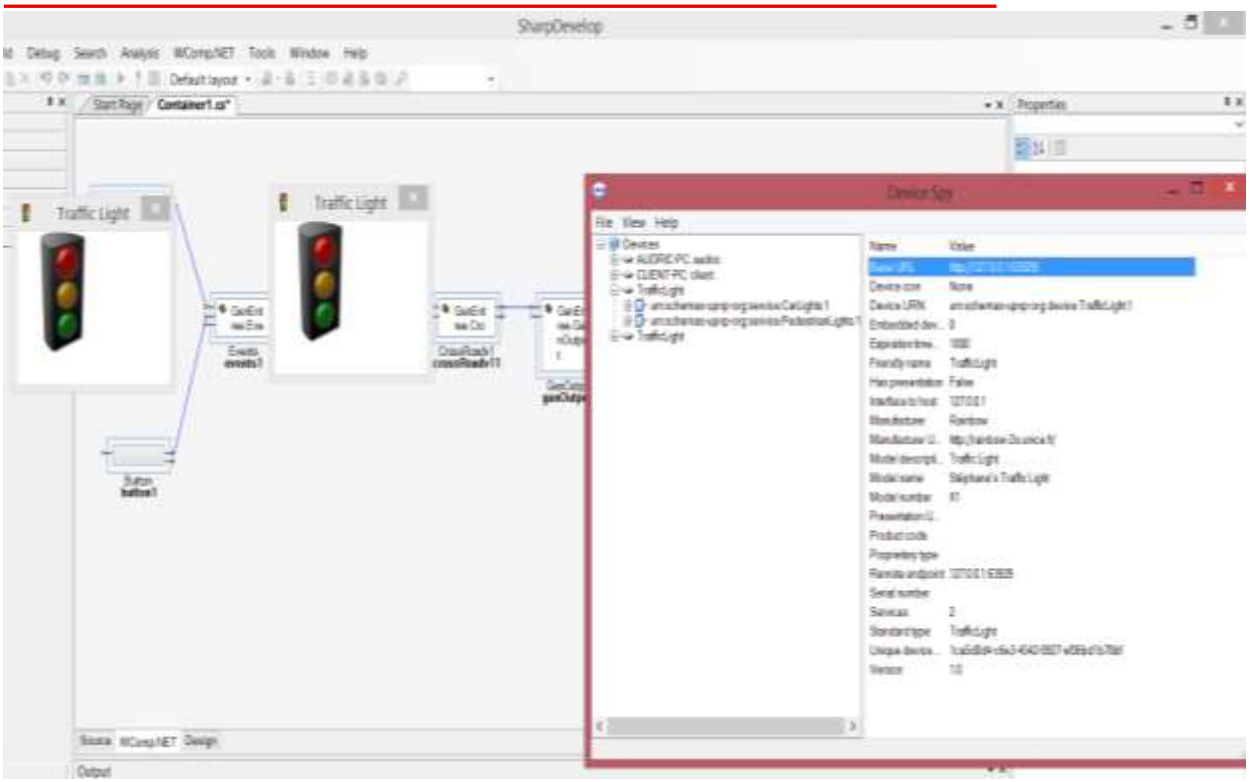
10/ Make the test.

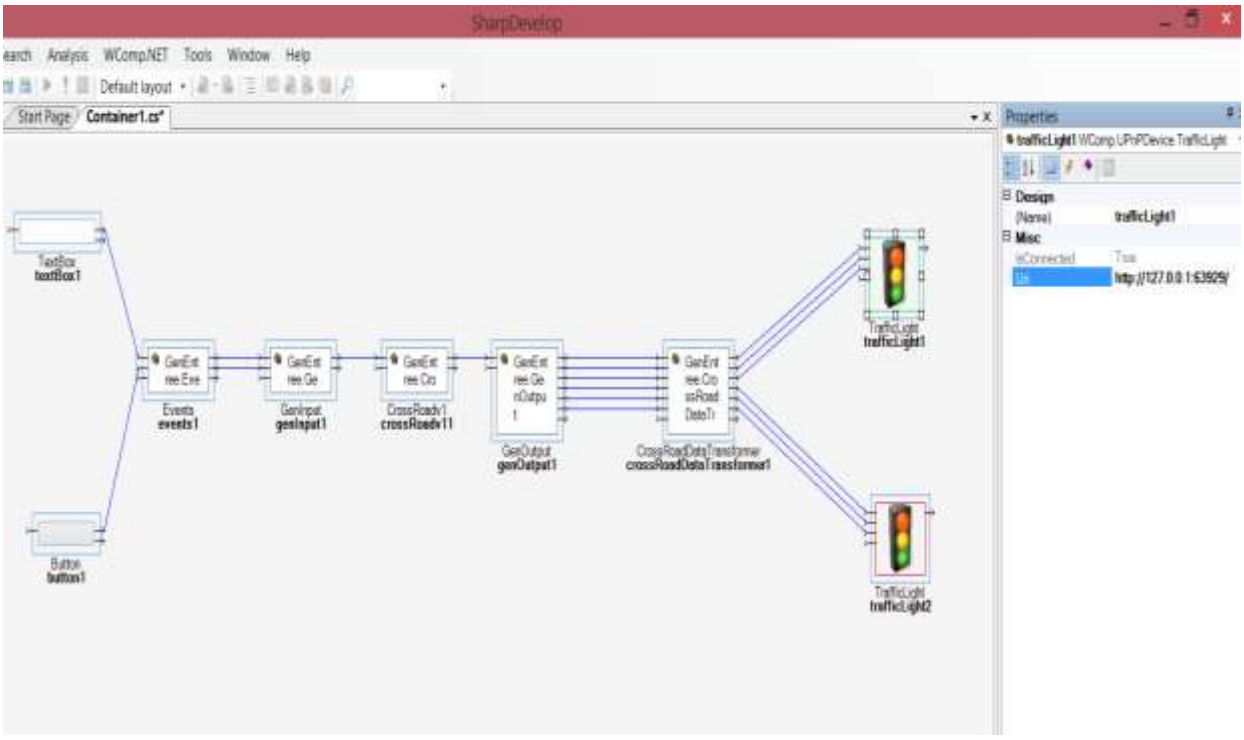



To make your test, you need first to activate a second traffic light and execute the **DeviceSpy**, you will see that the two trafficLights are detected. You have to change the address of your TrafficLights in your Wcomp Container with these trafficLights' addresses.

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

2011-2012

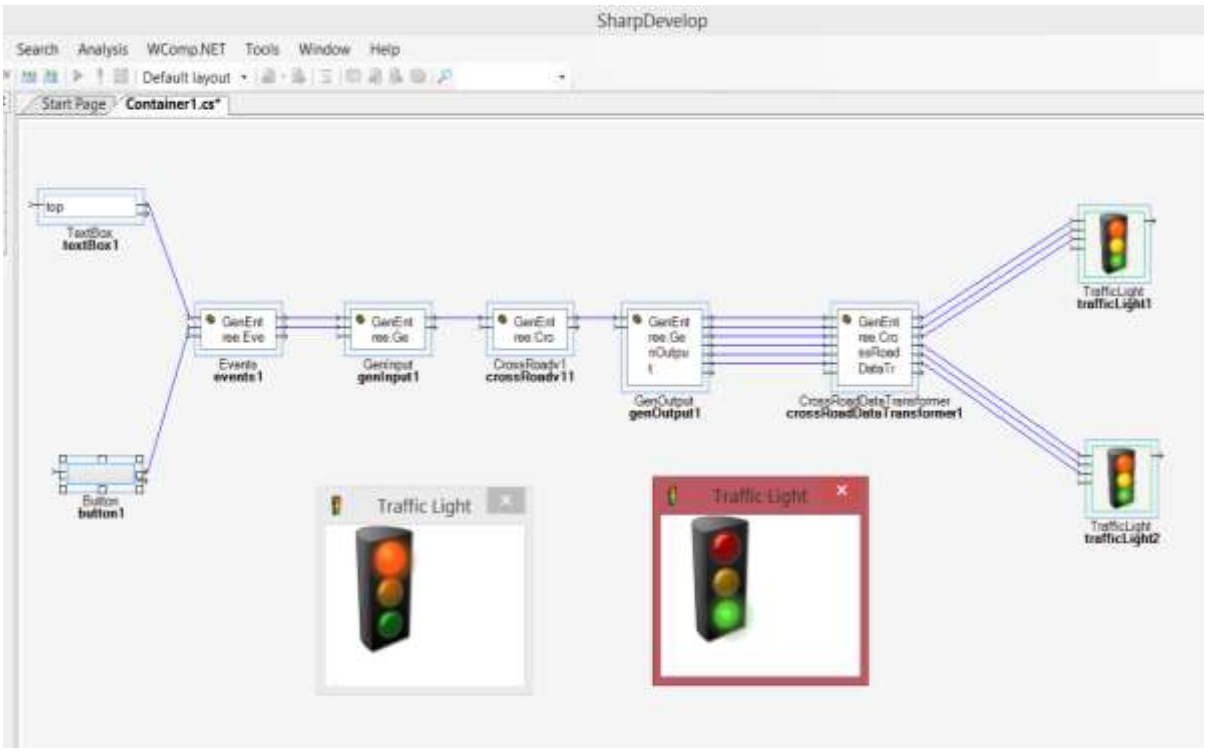



Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

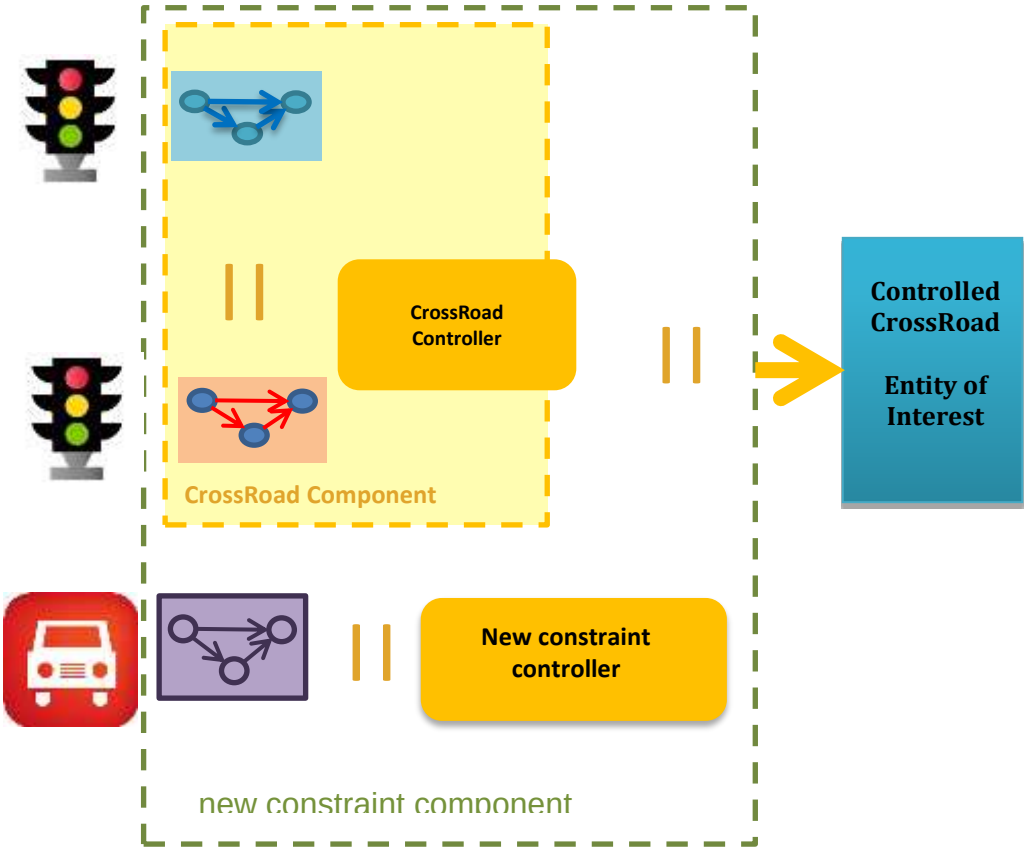
annie.ressouche@inria.fr, ines.sarray@inria.fr

Finally, write the name of the event “top”, click on it and then click on the button. The first time you will click on the button, it will create the event button and add it on the circular buffer, but it won’t be sent to the crossRoad because there is no occurrence of this event. The second time you click on the button, a new event will be sent and then the first event will be sent to the crossroad and the processing will be executed (In the beginning lights will be off because in its initial state all the lights should be off , so don’t worry ;)



# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

## APPEARANCE OF A DEVICE




Continuing with the CrossRoad use case, a CarFilter device appears and a new application can be created which takes into account this feature. The goal is to regulate the car traffic and if the number of cars is less than a fixed constant the cross road must maintain the yellow light in both directions. Hence, the model of the car filter is a small Mealy automata which listens a signals nbCars present when the number of cars is less than the bound and emit a signal yellow in reaction. When yellow is present, the normal behavior of the cross roads is preempted and yellowNS and yellowEW are maintained.

Thus, always according to the approach describe in the lecture, we will build a new constraint component, composed of: (1) the old constraint component, the carFilter model and a new constraint controller which must integrate new constraints taking account carFilter presence.

```
module crossRoadwithCarFilter:
Input: top, nbCars, inhibNS, inhibEW;
Output: greenNSF, yellowNSF, redNSF, greenEWF, yellowEWF, redEWF;
```

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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

```
Run:
".": CrossRoad: CrossRoad;
"." : CarFilter: CarFilter;
"." : crossRoadwithCarFilterConstraint : CrossRoadwithCarFilterConstraint;

local greenNSC, yellowNSC, redNSC, greenEWC, yellowEWC, redEWC, yellow

{

   run crossRoad

  ||

    run carFilter

 ||

    run crossRoadwithCarFilterConstraint
}

end
```

The new constraint component (called crossRoadwithCarFilter)

The new constraint controller (crossRoadwithcarFilterConstraint) can be implemented with a clem automata or an implicit Mealy machine as well. Here is an example of clem Mealy machine implementation:

```
module crossRoadwithCarFilterConstraint:

Input:

   yellow, yellowNS, greenNS, redNS, yellowEW, greenEW, redEW, inhibNS, inhibEW;

Output: greenNSF, yellowNSF, redNSF, greenEWF, yellowEWF, redEWF;

Mealy Machine

   greenNSF = greenNSC and not yellow;

   redNSF = redNSC and not yellow;

   yellowNSF = yellowNSC or yellow;

   greenEWF = greenEWC and not yellow;

   redEWF = redEWC and not yellow;

   yellowEWF = yellowEWC or yellow;


end
```

# TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

2011-2012

## TO DO

Implementation of this crossRoadwithCarFilter in CLEM, then in your previous design in WComp.

Annie Ressouche & Ines Sarray INRIA Sophia Antipolis Méditerranée

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