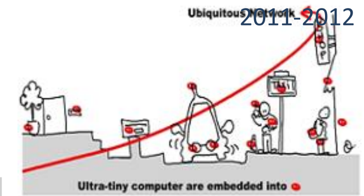
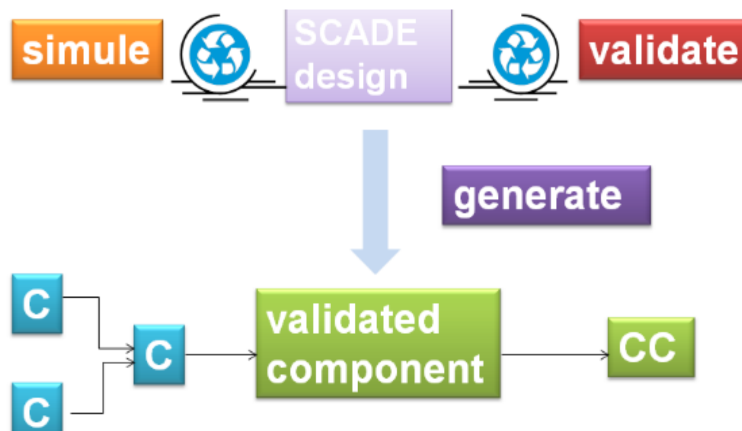
TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP



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 first study the SCADE tool which allows us to design and validate synchronous monitors.

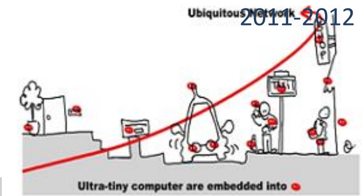
THE SCADE SUITE

SCADE SUITE INSTALLATION

1. Download the archive ***ET_SCADE64.zip.***
2. Launch the setup: ……\SCADE64\EsterelSCADE642\WINDOWS\SCADE\ScadeSetup; SCADE64 being the place where the archive has been extracted. **Select the Design Verifier** in order to install it.
3. Define the user environment variable ***ESTERELD_LICENSE_FILE*** = path where the license file (***ESTERELD.lic***) is. For instance, ESTERELD_LICENSE_FILE = C\Users\ar\Documents\Scade; the file ESTERELD.lic being at this place.

CREATING A PROJECT

First, you must create a project: ***File >> New*** opens a browser. You must supply a name for your project and the path where all the project files will be created. For instance, let us call the project: **TimerClock.** A file **TimerClock.vsw** is generated. It is the main file to reload your project in Scade. Now you have a working space falling into four sub windows:



1. A top window containing several toolbars to draw the application. In particular you will find buttons to declare inputs and outputs, data flow operators: mainly pre, init, if then else and buttons to define automata: synchronous state machine (ssm for short), state and transition labels. You will find as well buttons to check the coherency of your model, to simulate and to generate C code.

2. A left window with different views of the application:

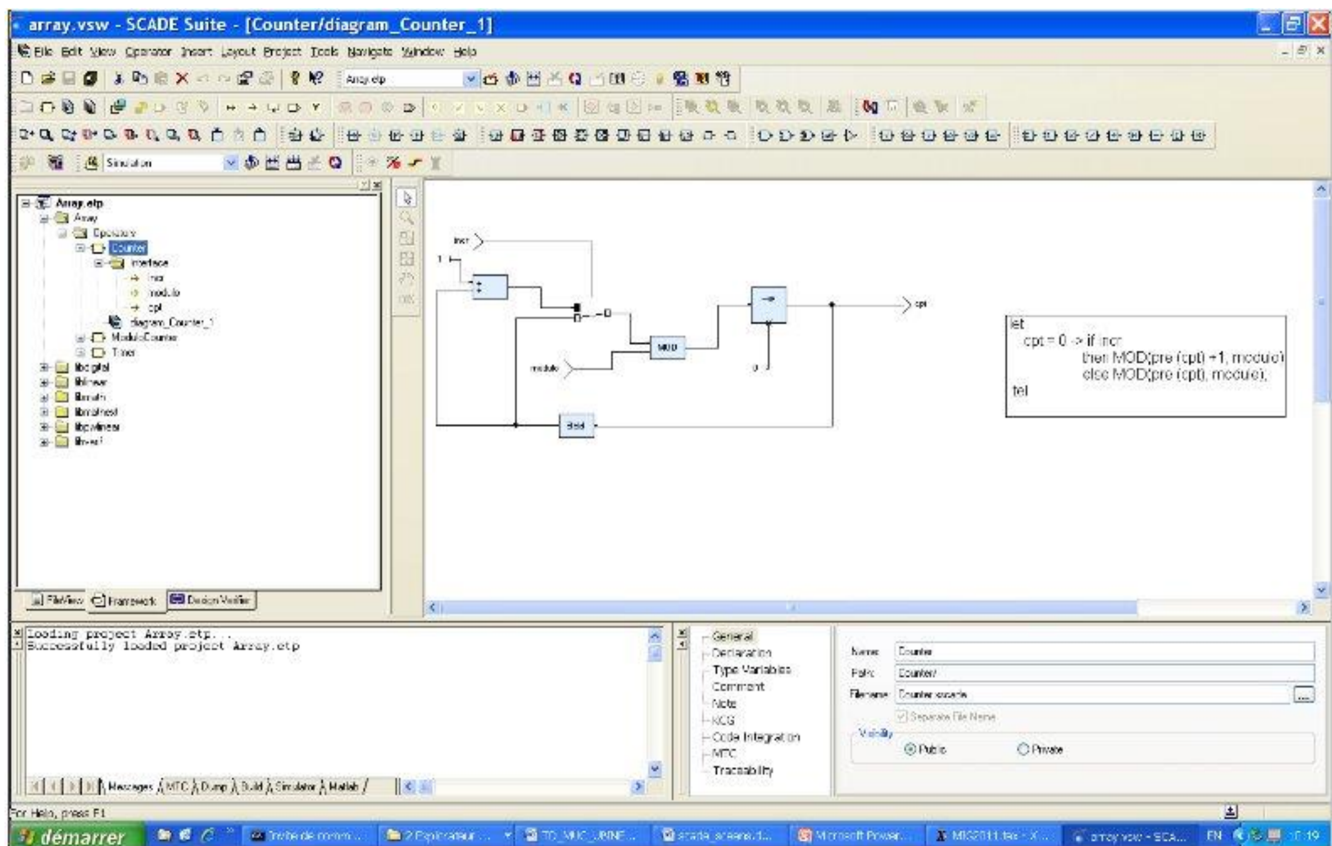
   a. ***File View*** is used to see the different files composing the project: the specification (Timer.ept) the (possibly hierarchical) organization of the project (Model Files), and the libraries containing predefined operators. In particular, the verification library (libverif) useful to define observers

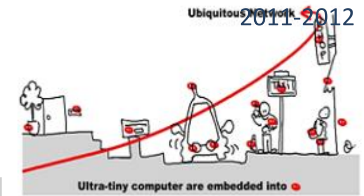
   b. ***Framework View*** is the main view in which the application is designed. The different operators the application is composed of are defined in the Operator thumbnail. But, if there is some hierarchy in the design of yours operators, the hierarchy is only shown in the ***File View***.

   c. ***Design Verifier*** to see the observers defined in the application and to launch the model-checking facility to verify it.

3. A main window (in the center) to draw the application

4. A bottom window to put up both results of operations (at left) and properties of selected objects in the other windows (at right), where you can modify them.



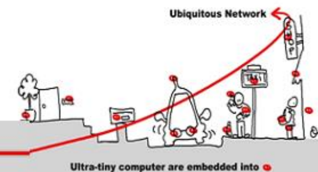


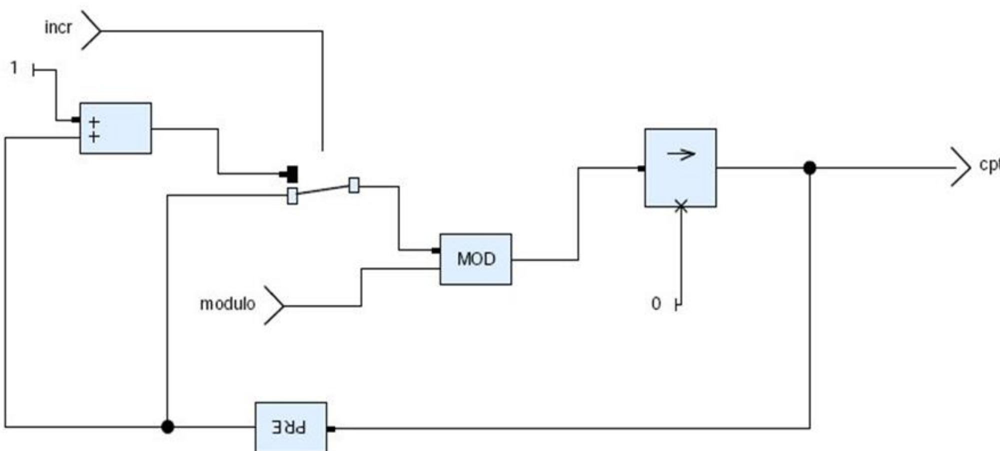
An example of SCADE windows

PROJECT SPECIFICATION

In practice, to draw an operator **Timer** of the **TimerClock** project, we select the ***Framework vi***ew. Click right in the ***Operator*** thumbnail and then select ***Insert >> Operator***. Choose a name for the operator (**Timer** for instance). Now the **Timer** operator is created with both and interface and a body. First you must define its interface (***Interface*** thumbnail): insert the inputs and the outputs of the **Timer** operator. If you click right on an item, you can change its property in the bottom right window. Input and output events can be valued with usual types (int, Boolean, etc..) or with user defined types. You can insert constants and types by clicking right in the **TimerClock** thumbnail. Then, you define the body of the operator. Click on diagram_Timer_1 thumbnail and a drawing space is open in the center window.

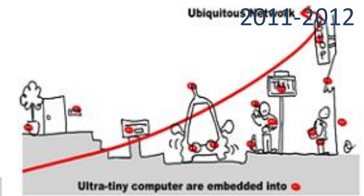
You can choose to define either a data flow diagram or an synchronous state machine (or both). To this end select the constructors in the upper toolbars window. In blue there are the data flow constructors, in pink the state machine ones.





An example of the data flow design of the modulo counter operator

Then the coherency (syntax and semantics) of the design can be check with the Check button in the upper toolbar and we can simulate it.



TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP

INTERACTIVE SIMULATION

1. To launch the simulator click on the ***Run*** button in simulation mode (small window in the top toolbars). Simulation is performed step by step, you must click the step button in the toolbar to execute a step.
   This notion of step corresponds to a reaction or an instant of the logical time. If the simulator is not active: Project>>Code Generator>>Set Active Configuration>>simulation and click on the button set active.
2. The main window of the simulator displays the model with current values along the wires. To change the value of inputs, choose ***Instance View*** in the left window and change the values with a (long) click on the input you want to assign. Three others windows are supplied:
   - A log window to show the history of the simulation
   - A window to observe the values of variables (***watch***)
   - A window to see the evolution of variables as a chronogram (*graph*)

   To insert a variable in one of these display windows, click right on it and select ***watch*** or ***graph*** (or both)
3. You can also save a scenario (**Timer.sss**) and reload it after.

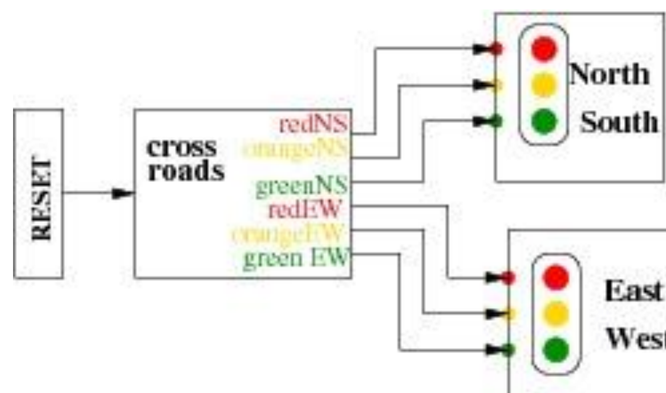
EXERCISE

As a training exercise, design the Timer application defined in the lecture. You must define the three operators: Counter, ModuloCounter and Timer. Then simulate them.

Scade also offers a model-checking tool and C code generation. We will study these features during the Crossroads use case we will design now.

DESIGNING A CROSSROADS SYNCHRONOUS MONITOR WITH SCADE

SPECIFICATION

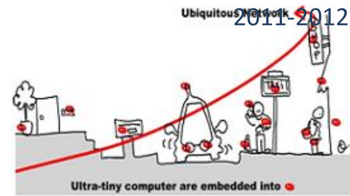


A crossroads with two orthogonal roads (east-west and north-south) is controlled by two traffic lights. Each traffic light works as follows: at each instant, the traffic light manages three Boolean outputs: red, orange, green.

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

Annie Ressouche INRIA Sophia Antipolis Méditerranée

annie.ressouche@inria.fr



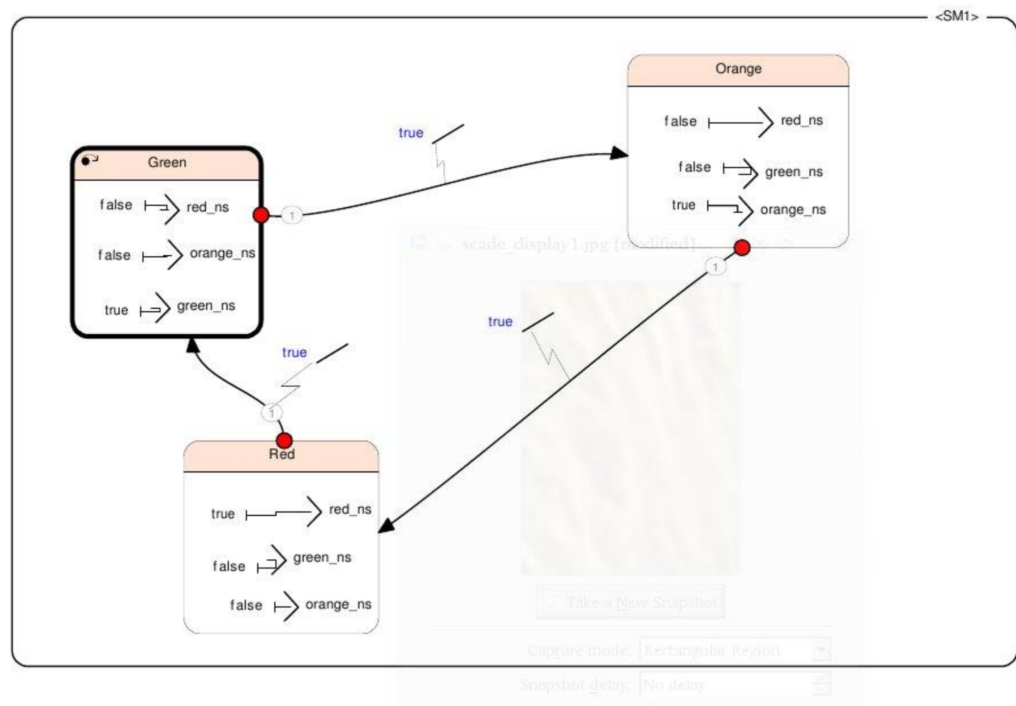
red → orange → green

We consider that the duration of each light is the duration of the clock of the traffic light.

TRAFFIC LIGHT MONITOR

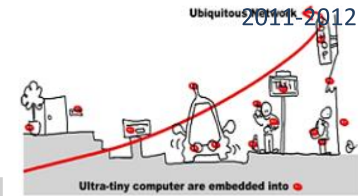
As it is mainly a controller, it is recommended to rely on synchronous state machine to carry out the design. The following diagram shows a possible implementation of a traffic light behavior. There are no inputs, we assume that the traffic light switch from a light to another according to the logical time. There are three outputs: green_ns, red_ns and orange_ns.

Thus, we design a state machine with 3 states: Green, Orange and Red. Transitions are triggered with true (meaning with a tick of the logical time) and in each state the output variables are set. This traffic light starts in Green state (we call it TrafficLightNS).



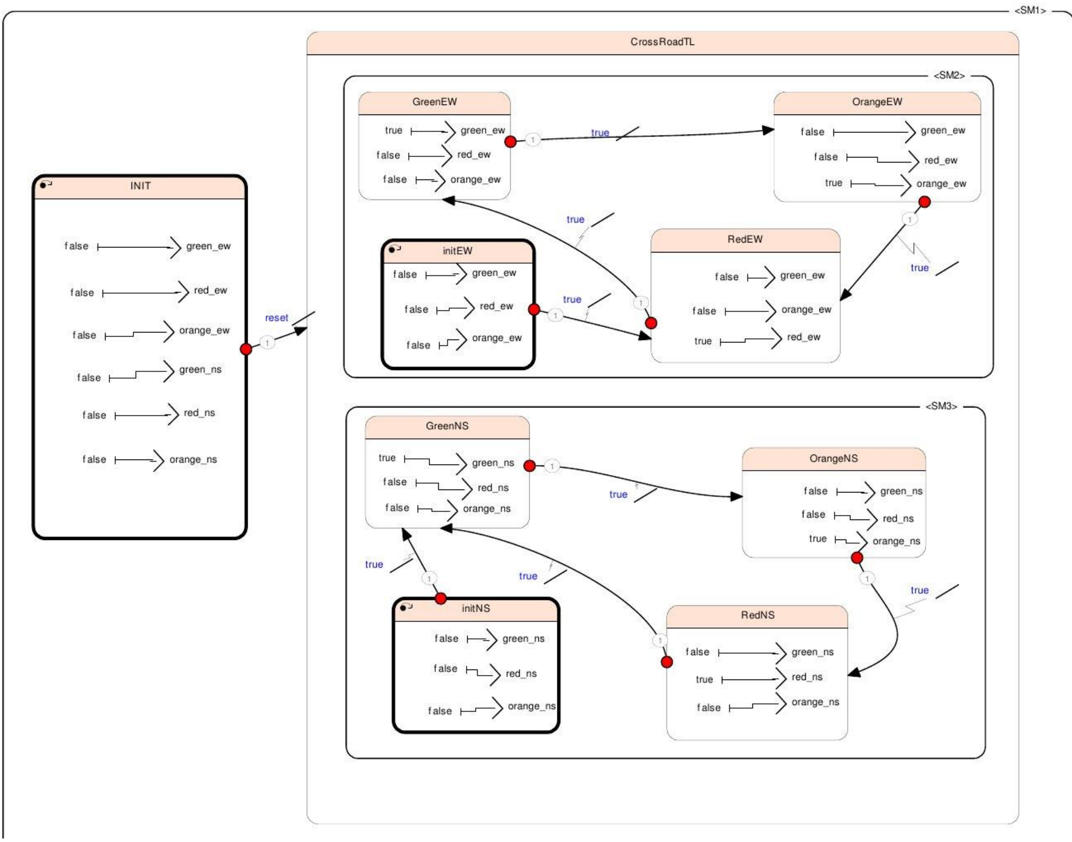
TrafficLightNS design in Scade Synchronous State Machine

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 state machine:



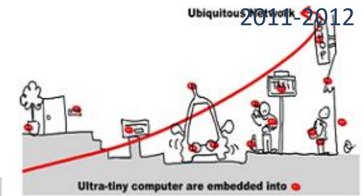
The crossroads specification is a state machine composed of two states an INIT state where all the variables are initialized and a CrossRoads state which contains itself two state machines in parallel. The first one manages the NS traffic light which starts in green and the second one manages the EW traffic lights which start in red.

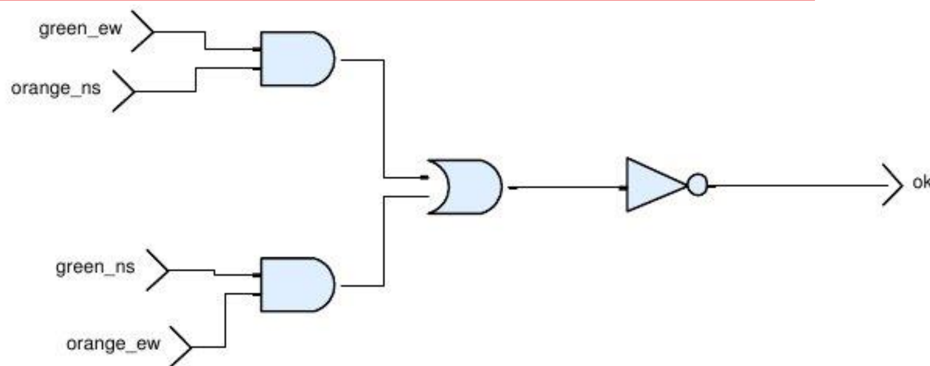
Design and simulate your own crossroads controller.

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 green_ns and orange_ew or green_ew and orange_ns.

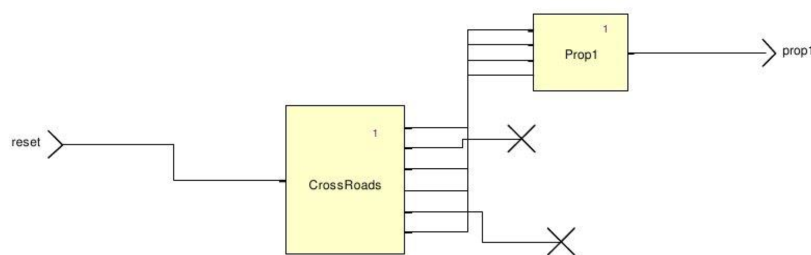
To achieve the proof we rely on the model-checking facility offers by Scade. Indeed, verification is performed by a sat-solver and the observer technique is used. Thus, we define an observer of the property. It is often more convenient to describe observers as data flow diagrams. We introduce in the project a specific operator called Prop1:





The observer listen the outputs of the Crossroads design green_ew, green_ns, orange_ew and orange_ns. It output a Boolean variable ok and the model checker will prove that prop is always true.

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



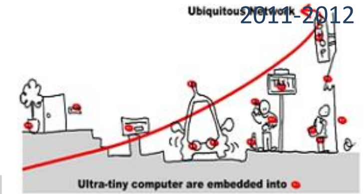
CrossRoadsVerif has an input variable reset and a Boolean output prop1. We connect respectively the green_ns output of the CrossRoads on the green_ns input of the Prop1 observer, and so on for the three others variables listened by the observer. The output ok of Prop1 is connected on the prop1 output of CrossRoadsVerif.

Now to validate the property we define a ***proof objective***. Select (right click) the prop1 output of the CrossRoadsVerif operator and choose ***Insert>>Proof Objective***. Then in the Design Verifier window, under the ***Proof Objectives*** thumbnail, we get an entry: CrossRoadsVerif.prop1. Right click again on this entry and choose ***analyze***. The model checker verifies that prop1 is always true. If not, you can get a counter example to understand why the property fails.

Perform the validation and correct your design until the property becomes true.

C CODE GENERATION

When the CrossRoads Scade design is correct, we generate the C code in order to plug it into a WComp application managing two traffic lights. First, in the upper toolbars, choose ***KCG*** in the small window instead of ***Simulation***. Then, just at the right there is a ***Generate*** button. Choose the place where the C code will be set. To illustrate the generation mechanism and also the plugging in a WComp application, we take into consideration the TrafficLightNS operator described previously. Select the operator in your Scade design and generate the C code.This generation phase mainly



provide you with two files: TrafficLightNS.h and TrafficlIghtNS.c. The header file contains the declaration of two C structures:

1. inC_TrafficLightNS: contains an entry for each input variable of the program.
2. outC_TrafficLightNS: contains an entry for each output variable of the program, an entry for recording each state value (true or false) and an initialization entry.

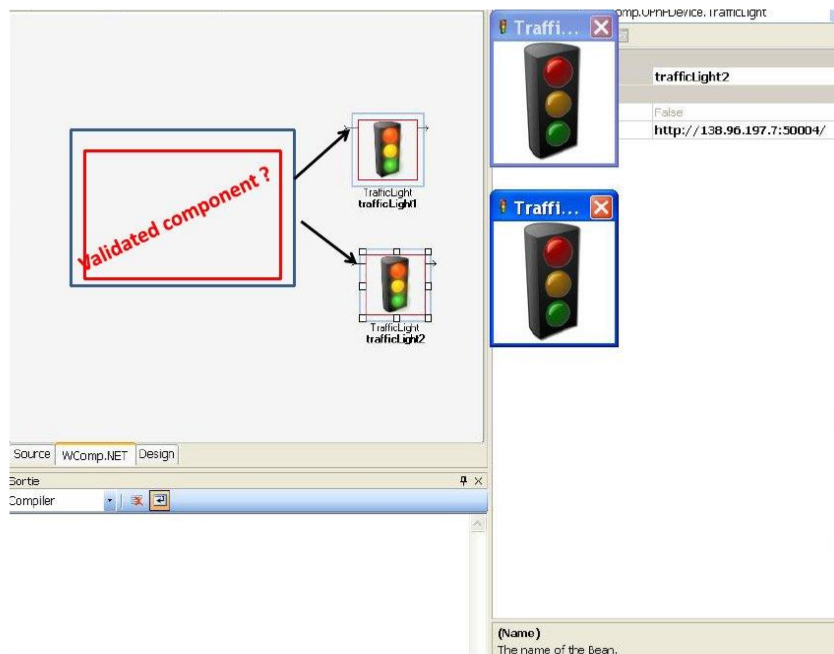
TrafficLightNS.c contains the definition of the structures and the definition of two functions you must use when you embed the generated code (it is the goal). TrafiicLightNS_reset() resets the outC_TrafficLightNS structure in its initial state and set all the input variables of inC_TrafficLIghtNS structure to their initial values (default values are provided for basic types). The main function is the TrafficLightNS function with has two argument respectively of type inC_TRafficLightNS* and outC_TrafficLIghtNS*. In the TrafficLightNS example, as there is no input variable in the interface, no inC_TrafficLightNS structure is created and the TrafficLightNS function has only a single argument to memorize output values. This function performs a step in the automata execution of the program.

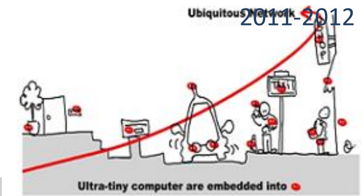
In addition there are a set of header files (.h) and code files (.c) to supply the KCG generator files defining KCG types, KCG predefined types, constants, etc….. Finally, in order to compile these generated files in your own application, you need the library where all these predefined features are defined. This, it is more comfortable to copy in your directory the library directory of Scade (~SCADE63\SCADE Suite\libraries).

Generate the C code for the CrossRoads operator.

SYNCHRONOUS MONITOR INTEGRATION IN WCOMP

The goal of the tutorial is to build in WComp an application managing the crossroads previously specified. We use two instances of the UpNP devices already defined in a previous tutorial and we want to build a validated application :



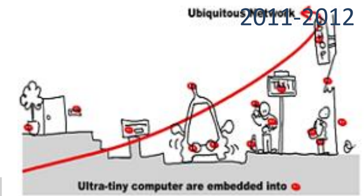


To this aim, we will plug the validated CrossRoads synchronous monitor in the application. To do this, we first build a native DLL of the generated code by Scade and second we will define a C# Bean calling functions define in this DLL.

CREATING CROSSROADS NATIVE DLL

To create a native DLL, we first define in C a file to export the structures and the functions defined in the generated C code. Here is the code mandatory to export TrafficLightNS synchronous monitor.

```
// This is a file for the construction
// of a win32 dll.
// TrafficLightNS_function


#define EXPORT  __declspec(dllexport)
#include <stdlib.h>
#include <stdio.h>
#include "TrafficLightNS.h"


//Function to create the output structure
EXPORT outC_TrafficLightNS* newOutC_TrafficLightNS()
{
 outC_TrafficLightNS* outC =
                     (outC_TrafficLightNS*)calloc(1,sizeof(outC_TrafficLightNS));
 TrafficLightNS_reset(outC);
 return outC;
}


//Functions to obtein outputs after one step
EXPORT int get_green_ns (outC_TrafficLightNS* outC)
{
  return outC->green_ns;
}
```




```
EXPORT int get_red_ns(outC_TrafficLightNS* outC)

{

  return outC->red_ns;

}

EXPORT int get_orange_ns(outC_TrafficLightNS* outC)

{

  return outC->orange_ns;

}



//Function to do one step

EXPORT void step(outC_TrafficLightNS *outC)

{

  TrafficLightNS(outC);

}
```

TrafficLightNS_functions.C

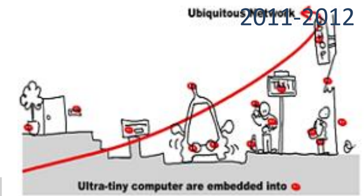
Then, we must compile TrafficLightNS.c and TrafficLightNS_functions.c files into a native DLL. We rely on the CL compiler of Visual C++. In a ***MS-DOS command*** file (***Invite de Commandes*** in French):

1. verify that CL exists, if not launch : **C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat program**
2. then create the dll : **cl –LD TrafficLight_functions.c Trafffic_light.c –FeTrafficLightNS_Scade.dll**

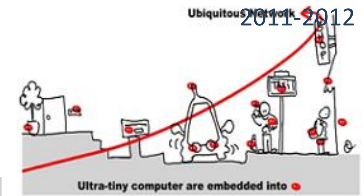
BUILD SYNCHRONOUS MONITOR BEAN UNDER SHARPDEVELOP

The last phase of the process is to build a C# component in Sharpdevelop. You must create a new solution for designing the targeted component. Then add a file to your new solution: right click on the name of your solution and then choose Add>New File. Here is a the C# code for the TrafficLIghtNS bean:

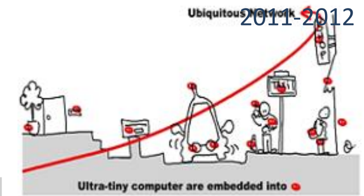
```
using System;

using System.Runtime.InteropServices;

using WComp.Beans;
```



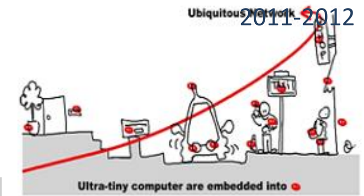

```
namespace WComp.Beans
{
  [Bean (Category="MyCategory")]
  public class TrafficLightNSBean
  {
        //Imports for DLL

  [DllImport("D:/SynComp/TP/TrafficLights/TrafficLights/WCompCode/traffic_light/TrafficLightNS_Scade.dll", SetLastError = true)]
        unsafe private extern static void* newOutC_TrafficLightNS();

  [DllImport("D:/SynComp/TP/TrafficLights/TrafficLights/WCompCode/traffic_light/TrafficLightNS_Scade.dll", SetLastError = true)]
        unsafe private extern static int get_green_ns(void* outC);

  [DllImport("D:/SynComp/TP/TrafficLights/TrafficLights/WCompCode/traffic_light/TrafficLightNS_Scade.dll", SetLastError = true)]
        unsafe private extern static int get_red_ns(void* outC);

  [DllImport("D:/SynComp/TP/TrafficLights/TrafficLights/WCompCode/traffic_light/TrafficLightNS_Scade.dll", SetLastError = true)]
        unsafe private extern static int get_orange_ns(void* outC);

  [DllImport("D:/SynComp/TP/TrafficLights/TrafficLights/WCompCode/traffic_light/TrafficLightNS_Scade.dll", SetLastError = true)]
        unsafe private extern static void step(void* outC);
        unsafe void* TrafficLightNS_outC;


        public unsafe TrafficLightNSBean()
        {
              TrafficLightNS_outC = newOutC_TrafficLightNS();
        }
```




```
public unsafe void doStep()
{
        step(TrafficLightNS_outC);
        Green = (get_green_ns(TrafficLightNS_outC)== 1);
        Red = (get_red_ns(TrafficLightNS_outC)==1);
        Orange =( get_orange_ns(TrafficLightNS_outC)==1);
}
// Bean methods
private bool green;
private bool orange;
private bool red;
// Bean Properties get and set
public bool Green {
        get { return green; }
        set {
                if (value != green) {
                        green = value;
                        if (green) {
                                FireGreenEvent();
                        }
                }
        }
}
public bool Orange {
        get { return orange; }
        set {
                if (value != orange) {
                        orange = value;
                        if (orange) {
```




```
                        FireOrangeEvent();
                    }
                }
            }
        }
        public bool Red {
            get { return red; }
            set {
                if (value != red) {
                    red = value;
                    if (red) {
                        FireRedEvent();
                    }
                }
            }
        }

        public delegate void BoolEventHandler(bool b);
        public event BoolEventHandler RedChanged;
        public event BoolEventHandler OrangeOffChanged;
        private void FireRedEvent() {
            if (RedChanged != null){
                RedChanged(Red);
                OrangeOffChanged(Orange);
            }
        }
       // bean events
        public event BoolEventHandler OrangeChanged;
        public event BoolEventHandler GreenOffChanged;
```




```
        private void FireOrangeEvent() {

                if (OrangeChanged != null) {

                        OrangeChanged(Orange);

                        GreenOffChanged (Green);

                }

        }

        public event BoolEventHandler GreenChanged;

        public event BoolEventHandler RedOffChanged;

        private void FireGreenEvent() {

                if (GreenChanged != null){

                        GreenChanged(Green);

                        RedOffChanged (Red);

                }

        }

 }

}
```

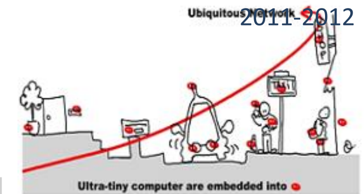
This code is composed of:

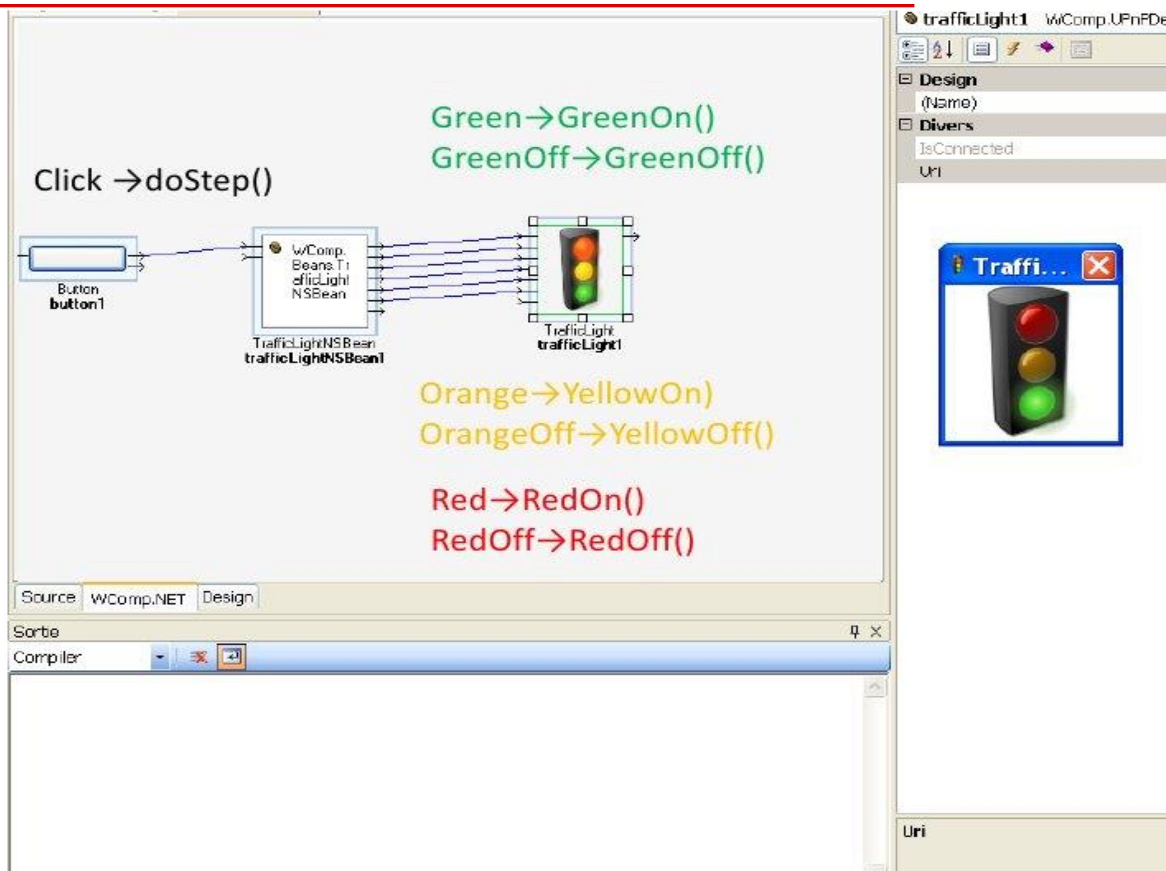
1. The declaration of the functions defined in TrafficLightNS.c and TrafficLight_functions.c we will use in the bean code preceded by an import Dll declaration.
2. The starting method of the bean (public unsafe TrafficLightNSBean()) to initialize the output structure.
3. The main method to perform a step in the automata. This latter (called here doStep) launch the step function (TrafficLightNS) of the C code and collect the value of green_ns, red_ns and orange_ns variables.
4. The properties Green, Orange and Red associated to the bean methods.
5. Finally, the definition of the behaviors of events connected to methods of the TrafficLight UpNP component (Green_On, Green_off, etc…)

The next step is to compile the TrafficLightNS bean and to copy the resulting DLL (let us call it TrafficLightWCompBean.dll) into the Beans directory of WComp (in the file: “Documents”/WComp.NET/Beans). To compile the bean code, you must allow to use unsafe code: switch on the correct flag in Projet>Options du projet>Compiler. TrafficLightWCompBean.dll is created in ...\bin\Debug.

Thus, performing the right connections in WCom, we design the application:

TUTORIAL: CREATING A VALIDATED CROSSROADS COMPONENT IN WCOMP





Here, we introduce our TrafficLightNSBean in a WComp application. We connect it to the TrafficLight UpNP component and we rely on the click of a button to simulate the tick of the synchronous logical time

CROSSROADS IN WCOMP

Define the appropriate CrossRoads_functions.c file and then design the CrossRoads bean and integrate link it to two TrafficLights in a WComp application.

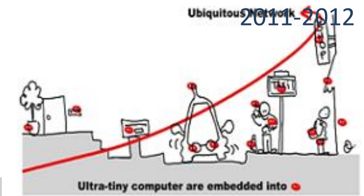
COMPOSITION IN WCOMP

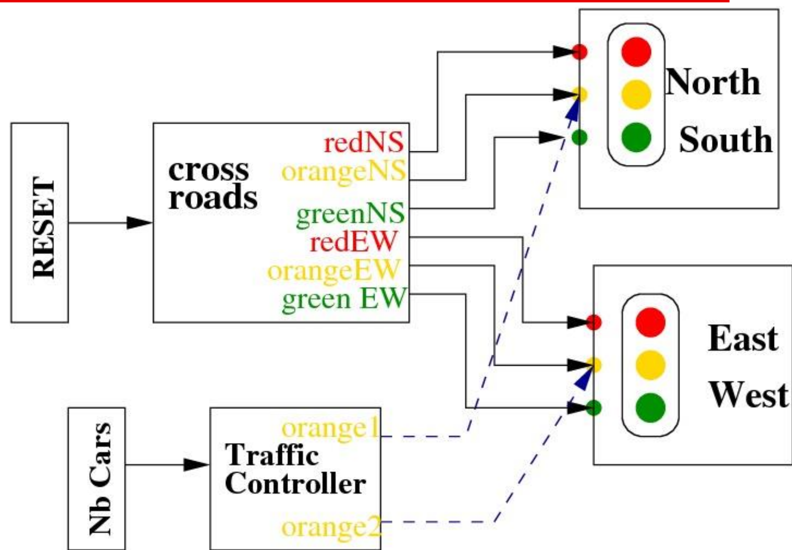
The last point we want to illustrate during this tutorial is the composition of synchronous monitors.

Continuing with the CrossRoads use case, we add a Traffic Controller component to the design to take into account the traffic density. This component listen the number of cars on the two roads and has two outputs orange1 and orange2. When the maximal number of cars is less than a given constant, its two outputs are true. After specifying the Traffic Controller in Scade, you must compose it with the CrossRoads component component to get an only new component in the WComp design. This composition relies on the definition of a constraint function (see the lecture) to define correctly what happens when $orange_1$ and $orange_2$ are true. Indeed, in such a case, only the respective orange lights of North South traffic light and East West traffic light are highlighted and the others are not.

Annie Ressouche INRIA Sophia Antipolis Méditerranée

annie.ressouche@inria.fr





The composition operation relies on the synchronous product. As we have already seen when putting the two traffic lights in parallel, in Scade synchronous product results in putting the two state machines of the two monitors side by side in a state. Then, a constraint function can be designed in data flow style to says how $orange_1$ ($orange_2$) are combined with the output events of the CrossRoads monitor.