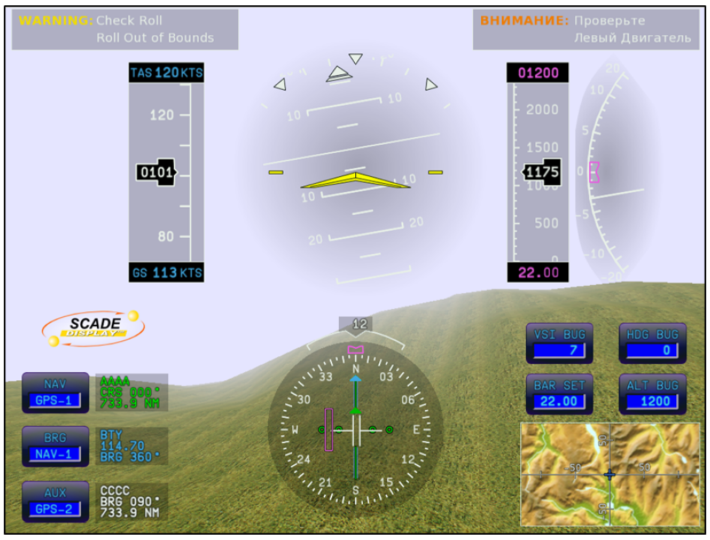
SCADE – BE1
Hands-on exercises



Frédéric Camps - LAAS/CNRS
fcamps@laas.fr

1 Objective of the BE

The general objective of this first series of exercises if to manipulate the concepts of the Lustre synchronous language via the SCADE tool. 3 steps will be studied:

- **Design**: switching from a high-level specification (textual) to a detailed specification (SCADE formalism data flow and state machines).
- **Validation** using simulation tools.
- **Verification** of the validation using model test coverage

Videos will also be provided all throughout the exercises in order to guide you step-by-step.

2 Exercises

2.1 CheckThreshold

Write a **CheckThreshold** operator that takes as input 2 real streams **SensorValue** and **Threshold**, which return a boolean output stream **alarm** of which the specification is as follows: the **alarm** output switches from false to true if **SensorValue** is greater than **Threshold**;

This node is in the course.
Carry out a semantic verification and simulate the behavior of this node.

2.2 RisingEdge

Write the **RisingEdge()** operator, which takes as input a boolean stream **c** and a boolean output stream **edge** which return true each time that the input stream **c** switches from false to true. In the 1st cycle, **edge** is false.

This node is in the course.
Carry out a semantic verification and simulate the behavior of this node.

2.3 FallingEdge

Write the **FallingEdge()** operator, which takes as input a boolean stream **c** and a boolean output stream **edge** which return true each time that the input stream **c** switches from true to false. In the 1st cycle, **edge** is false.

This node is in the course.
Carry out a semantic verification and simulate the behavior of this node.

2.4 Counter

Write a **Counter()** operator, which takes as input a Boolean stream **reset** and an integer stream **init**, which returns a integer output stream **count** and which increments by 1 its output at each instant. The counter is reset when **reset** is true. In the 1st cycle and at each reset, **count** is **init**.
This node is in the course.

Carry out a semantic verification and simulate the behavior of this node.

2.5 Clock Operator

We want to make a counter that sends information in the form of a pulse every 60 time ticks:
- The counter is reset every 60 time ticks and starts the count against from 0 to 60,
- The resetting can be set to a given value, here 60

1) Propose a solution with the counter operator and the logic and arithmetic operators. Note: be careful not to create causality problems.
2) Plot the States of the operator (inputs, outputs, intermediate points) in the form of a graph.

Carry out a semantic verification and simulate the behavior of this node.

2.6 SwitchOnOff

Write a **SwitchOnOff** operator that manages the behavior of a component and which takes as input 2 bool streams **Start** and Stop, which return a boolean output stream **OnOff** of which the specification is as follows:

- the **OnOff** output is false when the component is off (therefore in the Stopped state);
- the **OnOff** output switches to true when the component is on (therefore in the Started state);
- the component is off in the initial state and when it receives the **Stop** input;
- the component is on when it receives the **Start** input;

This node must be made using the state machines.

This node is in the course.
Carry out a semantic verification and simulate the behavior of this node.

3 System case study

The goal now is to implement a pressure and temperature controller in full. For this, you will be guided.

3.1 Specifications

A controller is connected to two sensor: pressure and temperature. An alarm is triggered each time the threshold is exceeded (temperature or pressure). The alarm lasts 30 time units. The controller triggers a cooling via a fan. The latter remains on for 120 time units then stops. The controller has temperature and pressure setpoints that are not to be exceeded, the latter can be modified.

No breakdown (controller, sensors, alarm, fan) is addressed in the study.

Maintenance is not represented. The system has been correctly installed and initialized. Turning off the system is not addressed.

3.2 Analysis of the system

- Extract the functional and non-functional registers from the specifications.
- Define the functions of the system: create a use case. Associate the requirements with each use case.
- Define the sequence diagram for:
  - exceeding temperature
  - exceeding pressure

3.3 Implementation in SCADE

Follow the various steps to implement the controller in SCADE.

3.4 Interface of the controler operator

Write the **Controler** operator which takes as input 2 real streams **Temperature** and **Pressure**, which returns 2 Boolean output streams **AlarmOnOff and FanOnOff**. This operator will also use local variables:

P_Alert: boolean
T_Alert: boolean
StartAlarm: boolean
StopAlarm: boolean
StartFan: boolean
StopFan: boolean

It will also use 2 constants to define the temperature and pressure thresholds:

thresholdTemp: 30.0°
thresholdPressure: 10.0

Create the inputs, outputs, the two constants as well as the local variables.

3.5 Re-use of CheckThreshold

The **Controler** operator must in a first step emit:

- A local variable of the Boolean type **t_Alert** when the temperature is higher than the constant **thresholdTemp**.
- A local variable of the Boolean type **p_Alert** when the pressure is higher than the constant **thresholdPressure**.

To do this, we will re-use the **CheckThreshold** operator by calling it twice in the **Controler** operator, once to manage the temperature and emit **t_Alert** and once to manage the pressure and emit **p_Alert.**

The **CheckThreshold** operator must now be called twice. Connect finally with the expected inputs of **Controler** and the 2 local variables.

3.6 Addition of 2 state machines

For the alarm:

- When **p_Alert** or **t_Alert** switch to true, this triggers:
  - Switching the local variable **StartAlarm** to true so as to start the alarm
  - Switching from the Active state to W_ALARM

- When 30 cycles have been executed, this triggers:
  - Switching to the CTRL state: starting from this state if the temperature or the pressure exceeds the threshold then we return to the W_ALARM state else we return to the ACTIVE state and the local variable **StopAlarm** switches to true so as to stop the alarm

For the fan:

- When **t_Alert** switches to true, this triggers:
  - Switching the local variable **StartFan** to true so as to start the fan
  - Switching from the Active state to W_FAN

- When 120 cycles have been executed, this triggers:
  - Switching the local variable **StopFan** to true so as to stop the fan
  - Returning to the Active state

So, in the **Controler** operator, 2 state machines must be created in parallel that will respectively be named **alarm_controler** and **fan_controler** in such a way as to implement the 2 behaviors hereinabove: timers for the alarm and for the fan.

3.7 Re-use of the SwitchOnOff operator

The **Controler** operator must now manage the alarm and the fan themselves.
The alarm starts when the local variable **StartAlarm** is true and stops when the local variable **StopAlarm** is true.
Likewise, the fan starts when the local variable **StartFan** is true and stops when the local variable **Stop**Fan is true.

These local variables are produced by the 2 previously-created state machines.

As the alarm and the fan can only be in 2 states (On or Off) according to the Start and Stop signals, the **SwitchOnOff** that we will instance twice is re-used: once for the alarm and another time for the fan.
Connect finally with the expected local variables of **Controler** and the 2 outputs to be produced.

3.8 Validation of the model

First of all, perform a semantic verification of the model.
Then, in order to validate, run the following scenario:

- The temperature is 3.0 and the pressure is 8.0 for 5 cycles
- The temperature is 9.0 and the pressure is 12.0 for 2 cycles
- The temperature is 9.0 and the pressure is 7.0 for 35 cycles
- The temperature is 11.0 and the pressure is 7.0 for 3 cycles
- The temperature is 4.0 and the pressure is 7.0 for 120 cycles

What do you observe?
Save this scenario.

3.9 Verification of the validation using model test coverage

Reminders:
In order to know whether if thanks to our scenarios during the simulation, we have tested all of the parts of our model, we are going to again simulate by running or scenario but by using in addition the Model Test Coverage (MTC) tool.

This tool will allow us, for each element in our Scade model, to know if during the simulation all of the criteria were covered.

For example, for a "greater than" operator, in order to be 100% covered with the "Decision Coverage" option of the Model Test Coverage instrumenter, we must during the simulation run a case where the "Greater than" output is false and one where it is true.

3.10 Instrumentation / Model Test Coverage

Using the explanatory document, run the instrumentation of your model with the "Operator DC" coverage criterion.

3.11 Traceability matrix

Use of RM Gateway. Perform the traceability between your solution and the DOORS requirements. You will be guided step-by-step.

3.12 Acquisition of coverage

Once the model is instrumented, open this instrumented model, refer to the explanatory document to:

- Add a result project
- Run the simulation and acquisition of the coverage
- Execute your simulation scenario saved earlier.
- Analyze the coverage obtained
- Execute an additional scenario in order to reach 100% of coverage if necessary

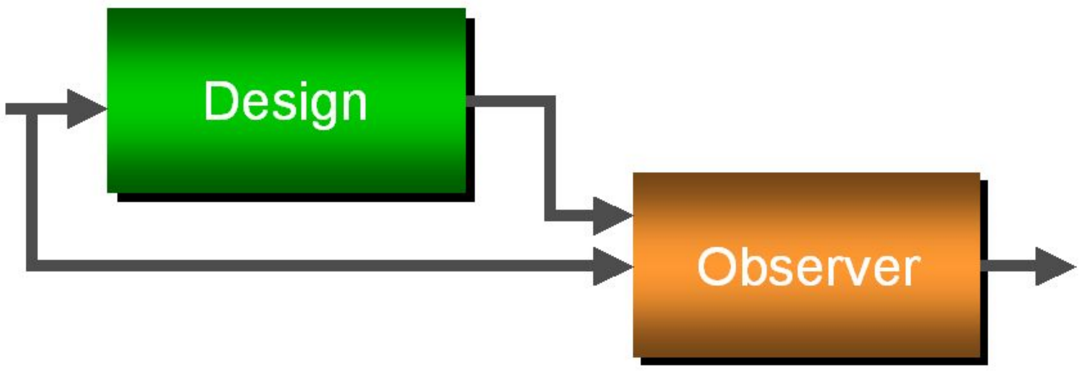
- Generate the report on the model coverage obtained

3.13 Verification via proof

Reminders:
Observers are special programs used to express properties or hypotheses. Their main characteristic is to have a single Boolean output.

This output must remain true as long as the inputs from the observer satisfy the property. We have come to say that the observer implements the property.

These observers can be used to check an "Observer" property on a "Design" program. If we want to prove this in the **SCADE** graphics tool, we must construct an Obs_Design node and call the **SCADE** formal verification tool on this node. The figure hereinbelow shows the way in which the Obs_Design verification program is constructed.



Property for the pressure and temperature controller:

1) Write an observer node "Prop1" implementing the following property: when the temperature exceeds a certain threshold then the alarm is systematically triggered.

Write the node "Obs_Prop1" connecting the node to be verified (Prop1) and the Controler node. Use the proof engine to show that this property is always true.

2) Propose a property linked to the temperature control system. Create the property and observe it.

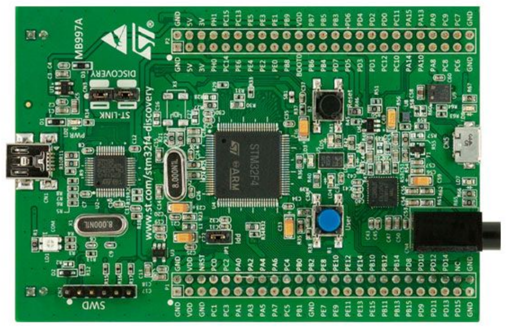
"Implies" operator

A	B	A=>B
0	0	1
0	1	1
1	**0**	**0**
1	1	1

This makes it possible to write: if a condition A is true, then the condition B must always be true and to appropriately test the case where A would be true but there would not be B and because of this the operator returns a false as output that the formal proof engine will detect.

3.14 Integration on card

Discovery of the STM32 card:
https://www.st.com/en/evaluation-tools/stm32f4discovery.html



In the integration phase the following elements will be taken into consideration:

- use of a temperature sensor,
- exceeding from a fixed threshold in the SCADE model,
- triggering of an alarm,
- triggering of a fan.

1) Then generate the code and your operator in the code of the project STM32.
2) Connection of the Dallas DS18B20 digital temperature sensor:

- yellow wire → PB4
- black wire → GND
- red wire → +5v

https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf

3) Load the Keil program and integrate your code in the control loop (follow the instructions in the course).

-o0o-