The COMON Case Study

Chaouki MAIZA, Simplice DJOKO-DJOKO, Erwan JAHIER, Pascal Raymond, Nicolas Halbwachs

SYNCHRON 2011

Outline

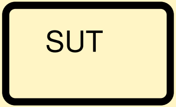
Outline

Lurette - Automatic Functionnal Tests of Reactive Sytems

Lurette - Automatic Functionnal Tests of Reactive Sytems

- Black box functionnal test
  - Comparing an implementation and a specification
- Automatical test
  - Generation of stimuli (SUT inputs)
  - Tests results checking (oracle)
- Based on a formal description of
  - *System Under Test* properties
  - Different hypothesis done on the environment
- Reactive Systems
  - The SUT reacts to the environment that it tries to control (feedback)
  - A *realistical* environment must acts the same way

Data Flow - Global View

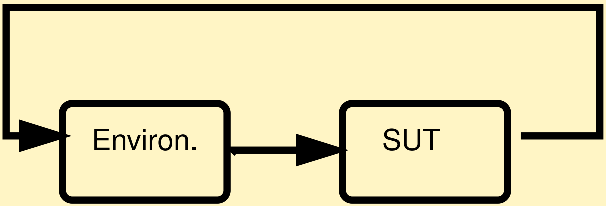


Data Flow - Global View

The environment can be seen as a

- Non-deterministic reactive system
- Constrainted test vector generator

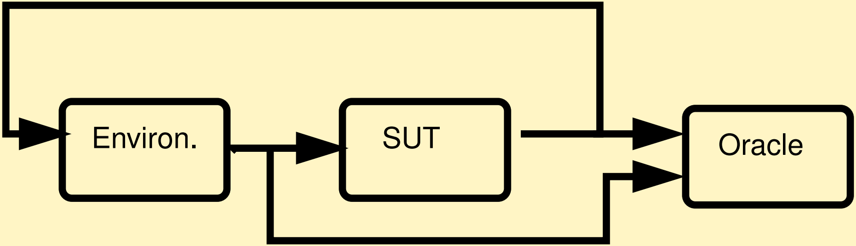
Data Flow - Global View



The environment can be seen as a

- Non-deterministic reactive system
- Constrainted test vector generator

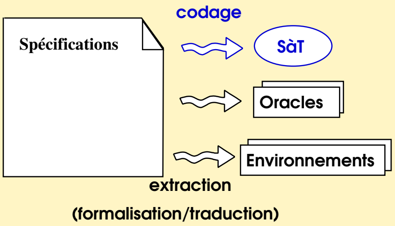
Data Flow - Global View



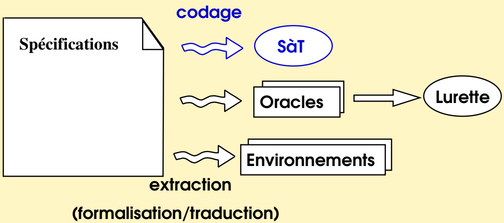
The environment can be seen as a

- Non-deterministic reactive system
- Constrainted test vector generator

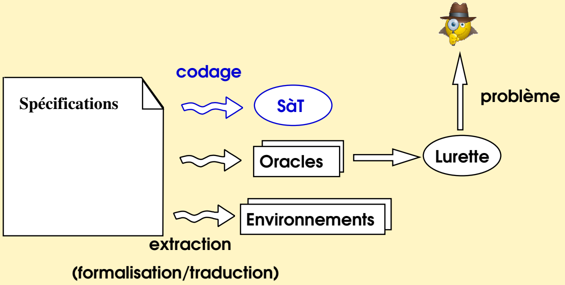
Lurette Test Process



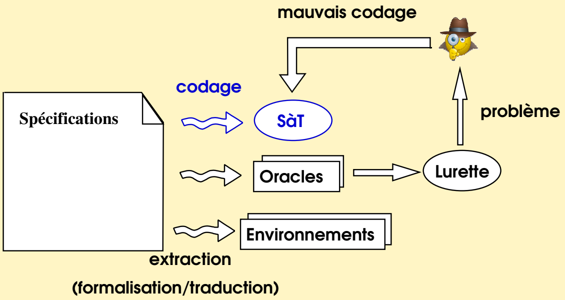
Lurette Test Process



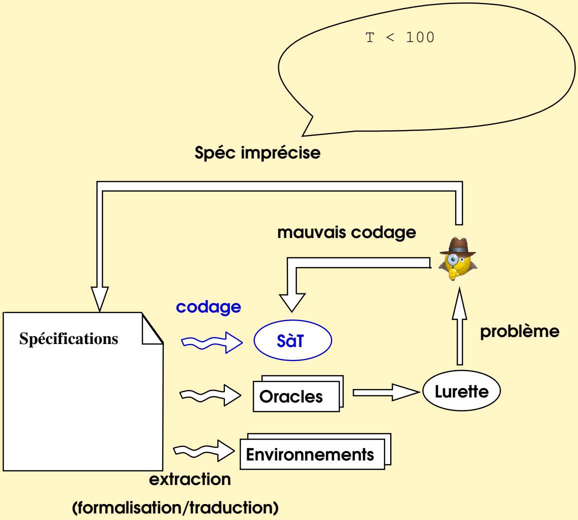
Lurette Test Process



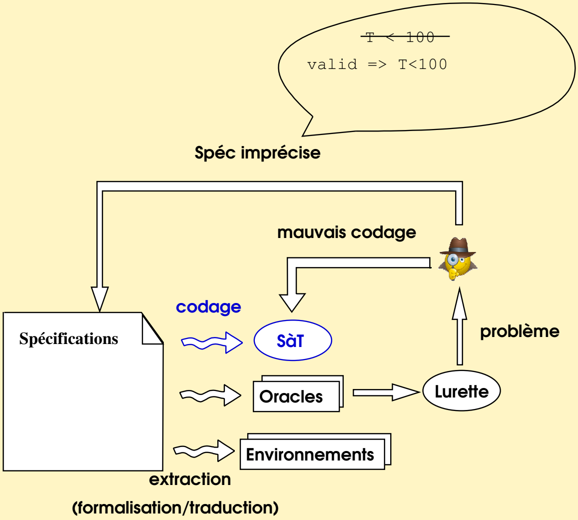
Lurette Test Process



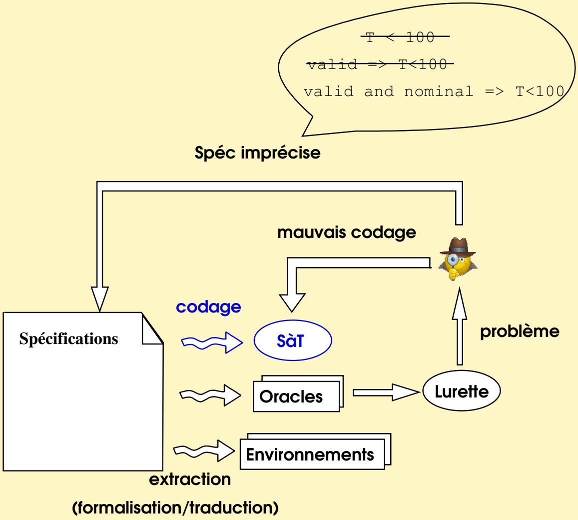
Lurette Test Process



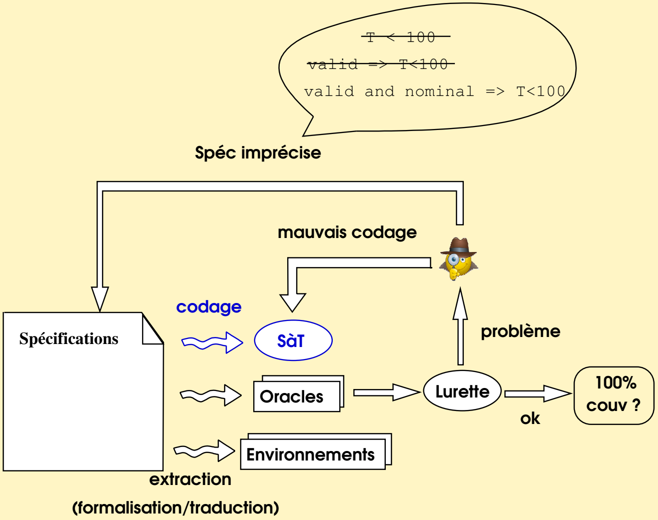
Lurette Test Process



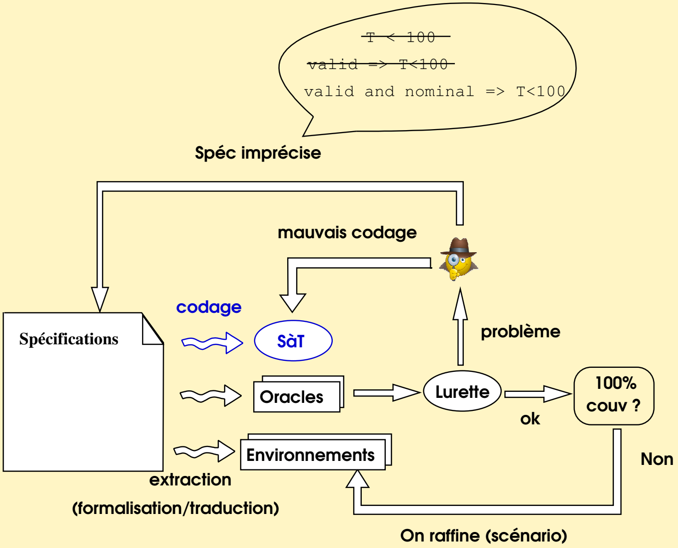
Lurette Test Process



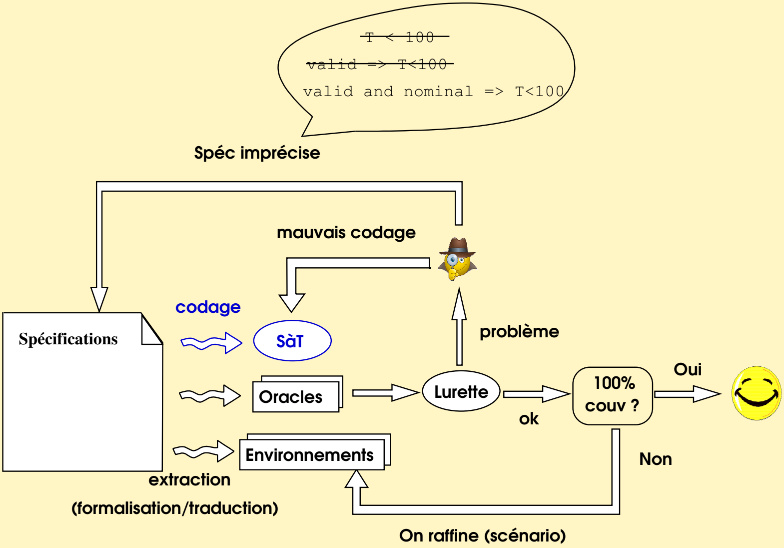
Lurette Test Process



Lurette Test Process



Lurette Test Process



Outline

COMON - Continuous Test Chain

Control-Command for civil nuclear systems

- ATOS WorldGrid
- Rolls-Royce Civil Nuclear
- Corys
- VERIMAG

COMON - Continuous Test Chain

Control-Command for civil nuclear systems

- ATOS WorldGrid
- Rolls-Royce Civil Nuclear
- Corys
- VERIMAG

		N2
		Interface
N1CL	Interface	N1
	Interface	Interface
		N0

COMON - Continuous Test Chain

Control-Command for civil nuclear systems

- ATOS WorldGrid ADACS : visualization
- Rolls-Royce Civil Nuclear SCADE : simulation
- Corys ALICES : simulation
- VERIMAG Lurettte : test manager

		N2 <- ATOS
		Interface
RRCN -> N1CL	Interface	N1 <- ALICES
	Interface	Interface
		N0 <- ALICES

COMON - Continuous Test Chain

1. Develop an executable model (in ALICES) of all the sub-systems

COMON - Continuous Test Chain

1. Develop an executable model (in ALICES) of all the sub-systems
2. Test those models by confrontation to a formal description of the system properties (written in natural language)

COMON - Continuous Test Chain

1. Develop an executable model (in ALICES) of all the sub-systems
2. Test those models by confrontation to a formal description of the system properties (written in natural language)
3. Integration of real systems (emulation/stimulation)
   - the oracles of point 2 can be reused
   - the models of point 1 can serve as references

COMON - Continuous Test Chain

1. Develop an executable model (in ALICES) of all the sub-systems
2. Test those models by confrontation to a formal description of the system properties (written in natural language)
3. Integration of real systems (emulation/stimulation)
    - the oracles of point 2 can be reused
    - the models of point 1 can serve as references

- Testing an open reactive system is made by simulating its environment and confronting it to an executable model

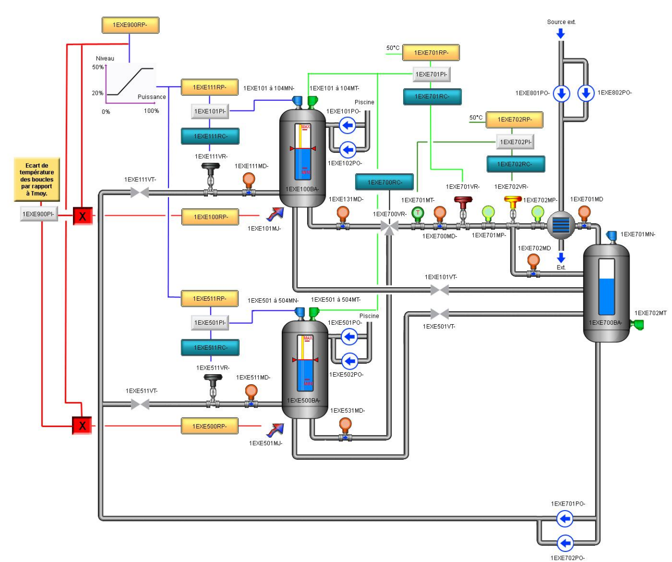
COMON - Continuous Test Chain

1. Develop an executable model (in ALICES) of all the sub-systems
2. Test those models by confrontation to a formal description of the system properties (written in natural language)
3. Integration of real systems (emulation/stimulation)
   - the oracles of point 2 can be reused
   - the models of point 1 can serve as references

- Testing an open reactive system is made by simulating its environment and confronting it to an executable model

- The central point in this approach is obtaining, testing and validating an executable model earlier.

Experiments done during this case study



Experiments done during this case study

- Formalisation from natural language into
  - Lustre for the oracles
  - Lutin for the environments/stimulators

Experiments done during this case study

- Formalisation from natural language into
  - Lustre for the oracles
  - Lutin for the environments/stimulators
- Macrolibrary of generic nodes to help writing oracles and stimulators
  - Checking that a numerical value remains in a specific range
  - Calculating different system states (situation) :
    - nominal, low power, 2/3-1/3, loop 1, emergency
  - Detecting the stability of a variable
  - Doing an action and waiting for the stabilitry
  - etc.

Outline

An oracle supervising threshold crossings

When a high threshold is crossed, the N2 must show an alarm after 5 seconds and reciprocally.

An oracle supervising threshold crossings

When a high threshold is crossed, the N2 must show an alarm after 5 seconds and reciprocally.

```
alarm_raised = timer(alarm, 5.0);
high_threshold_crossed = timer((v > high_threshold), 5.0);
```

```
ok1 = (falling_edge(high_threshold_crossed) => alarm_raised);
ok2 = (alarm => high_threshold_crossed);
```

An oracle supervising threshold crossings

When a high threshold is crossed, the N2 must show an alarm after 5 seconds and reciprocally.

```
alarm_raised = timer(alarm, 5.0);
high_threshold_crossed = timer((v > high_threshold), 5.0);
```

```
ok1 = (falling_edge(high_threshold_crossed) => alarm_raised);
ok2 = (alarm => high_threshold_crossed);
```

NB : it is the same for low, very high and very low thresholds

An oracle supervising the system situations

- We extracted different system situations from the natural language description : `nominal`, `low power`, `emergency`
- Then we expressed relations between these properties :
  `nominal <-> low power <-> emergency`

An oracle supervising the system situations

- We extracted different system situations from the natural language description : `nominal`, `low power`, `emergency`
- Then we expressed relations between these properties :
  `nominal <-> low power <-> emergency`

```
node oracle_situation(nominal, low_power, emergency : bool)
returns(ok:bool);
 ok = true ->
      (pre nominal => nominal or low_power) and
      (pre low_power => low_power or nominal or emergency) and
      (pre emergency => emergency or low_power);
```

An oracle supervising the system stability

In nominal mode, after each operator order, all the sensors values must be stables after 5 minutes

An oracle supervising the system stability

In nominal mode, after each operator order, all the sensors values must be stables after 5 minutes

```
C = true_since(no_new_order, 300.0) and nominal;
ok = (C => is_stable)
```

An oracle supervising the system stability

In nominal mode, after each operator order, all the sensors values must be stables after 5 minutes

```
C = true_since(no_new_order, 300.0) and nominal;
ok = (C => is_stable)
```

- *Covering* this test means generating a sequence where `C` is true

An oracle supervising the system stability

In nominal mode, after each operator order, all the sensors values must be stables after 5 minutes

```
C = true_since(no_new_order, 300.0) and nominal;
ok = (C => is_stable)
```

- *Covering* this test means generating a sequence where `C` is true
- Yet, the need of a scenario where the operator orders dont change too quickly

Outline

A (pseudo-)random failure generator

Goal : generating `n` failures randomly without triggering classified actions

A (pseudo-)random failure generator

Goal : generating `n` failures randomly without triggering classified actions

- `Emergency` : a constraint specifying the failures that will trigger a classified action
  - 2 redundant components failing at the same time
  - 3 ou 4 quadri-redundant sensors failing toghether

A (pseudo-)random failure generator

Goal : generating `n` failures randomly without triggering classified actions

- `Emergency` : a constraint specifying the failures that will trigger a classified action
  - 2 redundant components failing at the same time
  - 3 ou 4 quadri-redundant sensors failing toghether
- We randomly choose between the solutions of the constraints

A (pseudo-)random failure generator

Goal : generating `n` failures randomly without triggering classified actions

- `Emergency` : a constraint specifying the failures that will trigger a classified action
  - 2 redundant components failing at the same time
  - 3 ou 4 quadri-redundant sensors failing toghether
- We randomly choose between the solutions of the constraints
- A minimal scenario to
  - avoid switching between failures at each cycle
  - switch anyway after 200 cycles (or after a `reset`)

A (pseudo-)random failure generator

Goal : generating `n` failures randomly without triggering classified actions

- `Emergency` : a constraint specifying the failures that will trigger a classified action
  - 2 redundant components failing at the same time
  - 3 ou 4 quadri-redundant sensors failing toghether
- We randomly choose between the solutions of the constraints
- A minimal scenario to
  - avoid switching between failures at each cycle
  - switch anyway after 200 cycles (or after a `reset`)

```
node(n:int; reset:bool) returns(P1,P2,P3,P4,P5,P6) =
 let Emergency = two_false(P1,P2) or three_false(P3,P4,P5,P6) in
 let nb_failures = (if P1 then 1 else 0)+...+(if P6 then 1 else 0) in
  loop {
    { (not Emergency and nb_failures = n) |> nb_failures = n }
    fby loop 200 {maintain(P1) and ... maintain(P6) and not reset}
  }
```

Failures-i.lut ; Failure Demo

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])
2. Use a node that change the opening order step by step until it reaches the target

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])
2. Use a node that change the opening order step by step until it reaches the target
3. When the target is reached, restarts in 1.

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])
2. Use a node that change the opening order step by step until it reaches the target
3. When the target is reached, restarts in 1.

In order to bring the opening order step by step to the target :

1. Keep the same order until obtaining system stability

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])
2. Use a node that change the opening order step by step until it reaches the target
3. When the target is reached, restarts in 1.

In order to bring the opening order step by step to the target :

1. Keep the same order until obtaining system stability
2. Getting closer to the target order by at most `step` amount
3. If the target is not reached, restarts in 1.

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])
2. Use a node that change the opening order step by step until it reaches the target
3. When the target is reached, restarts in 1.

In order to bring the opening order step by step to the target :

1. Keep the same order until obtaining system stability
2. Getting closer to the target order by at most `step` amount
3. If the target is not reached, restarts in 1.

This illustrate Lutin ability to express complex scenarii.

A (pseudo-)random operator scenario

Change the opening order of a gate while supervising the values of sensors (level)

1. Choose a target opening order ([0 ; 100])
2. Use a node that change the opening order step by step until it reaches the target
3. When the target is reached, restarts in 1.

In order to bring the opening order step by step to the target :

1. Keep the same order until obtaining system stability
2. Getting closer to the target order by at most `step` amount
3. If the target is not reached, restarts in 1.

This illustrate Lutin ability to express complex scenarii.
-> operator.lut ; Operator demo

Outline

Demo !

- The 4 systems communicate
- Lurette pilots the tests
- Use the 2 scenarios presented before
- Use (most of) the oracles presented before