Introduction to SCADE

Julien Delange <julien.delange@esa.int>
About this course

• Introduction, not a complete lecture
  • Cover most SCADE concepts
  • For interested students, resources available on the internet
  • See the links section

• Focused on main SCADE aspects and practical use
  • Flow-based approach, state machines definition
  • Relation with the LUSTRE language
  • Code generation & certification concerns
Overview

- Flow-based approach, introduction to Lustre
- SCADE for data-flow specification
- SCADE state machines
- Code generation
- Conclusion & links
Overview

- Flow-based approach, introduction to Lustre
- SCADE for data-flow specification
- SCADE state machines
- Code generation
- Conclusion & links
Lustre: rationale

• Safety-critical software must be deterministic
  • System execution behaves as expected (specs.)

• Reaction to events and associated values
  • Sensors, actuators

• Domain-specific and math approach
  • System specified engineers
  • Not programmers!
Transformation vs. reactive systems

- **Transformational systems**
  - Take input, produce outputs
  - Execution time not bound
  - Ex: calculator, C program, etc.

- **Reactive systems**
  - React to signals
  - System rhythm is predefined
  - Ex: speed control system
Lustre: overview

• Data-flow approach
  • New data are produced at each \( i \) instant
  • Data are computed at each reaction

• Many advantages
  • Closed to maths
  • Data-based dependencies
Data-flow example

\[ X(t) = 2 \times Y(t) + Z(t) \]
\[ W(t) = X(t) + 1 \]

- Must consider data dependencies
  - \( X \) depends on \( Y \) and \( Z \)
  - \( W \) depends on \( X \) (and so, on \( Y \) and \( Z \))
- New data produced at each instant
Lustre, flow-based synchronous language

• Software separated into NODES
  • Take one or several input values
  • Produces one or several output values at each reaction

• Use usual types
  • Integer, boolean, etc.

• Lustre operators
  • Usual: +, -, mod, /, *, etc …
  • pre(val): value at previous reaction
  • current(val): value at current reaction
Lustre, syntax

- Expressions separated with `;`
  - Parallel execution

- Assignment `=`
  - X = 1

- Initial value `→`
  - X = 0 → pre (X) + 1;

- Previous value `pre()`
  - X = pre (Y)

- Current value `current()`
  - X = current (Y)

- Verification system values: `assert()`
  - Assert (not X)
Lustre, syntax (2)

- **Node call**: `nodename (param1, param2, ...)`
  - `X = mynode (param1)`

- **Tuples** `(val1, val2) = (var3, var4)`
  - `(X, Y) = (2, 3)`
Lustre, syntax (3)

- **Traditional boolean operators**
  - and, or, not, xor

- **Condition** if
  - \( N = 0 \rightarrow \text{if (Y) then X} \)
  - \( N = 0 \rightarrow \text{if (Y) then X else Z} \)
  - \( N = 0 \rightarrow \text{if (Y) else Z} \)

- **Filtering values** when
  - \( Y = Z \text{ when X} \)
node COMPUTE (Y, Z: int) returns (X, W: int)
let
  X = 2 * Y + Z; W = X + 1;
tel
node EDGE (b : bool)
returns (edge : bool);
let
edge = false → b and not pre b;
tel
Lustre: cyclic data dependencies

- Cyclic dependency
  - Non-sense!
- Self-dependency
  - A data cannot depend on itself!

\[
\begin{align*}
X &= \text{if } W \text{ then } Y \text{ else } Z; \\
Y &= \text{if } W \text{ then } Z \text{ else } X;
\end{align*}
\]

\[
\begin{align*}
X &= X + 1; \\
X &= X + 1;
\end{align*}
\]
node WATCHDOG1 (set, reset, deadline : bool)
  returns (alarm : bool)
  var watchdog_is_on : bool;
let
  alarm = deadline and watchdog_is_on;
  watchdog_is_on = false → if set then true
                       else if reset then false
                       else pre(watchdog_is_on);
  assert not (set and reset);
tel
node WATCHDOG2 (set, reset: bool, delay : int) returns (alarm : bool)
var
remaining_delay : int;
deadline : bool;
let
  alarm = WATCHDOG1(set,reset,deadline);
deadline = EDGE (remaining_delay = 0);
remaining_delay = if set then delay
  else (0->pre(remaining_delay) - 1);
tel
node COMPUTE (Y, Z: int)
returns (X, W: int)
let
    X = 2 * Y + Z; W = X + 1;
tel
Overview

- Flow-based approach, introduction to Lustre
- SCADE for data-flow specification
- SCADE state machines
- Code generation
- Conclusion & links
SCADE

- Flow-based design
  - Similar as Lustre

- State machines integration
  - Interaction with flows

- Specific graphical notation
  - Drag & drop approach
SCADE: main features

• System design
  • Data flows & state machines
  • Predefined operators

• Simulation
  • Graphical simulation, automatic GUI integration

• Code generation
  • Certified code
  • Integration with RTOS
SCADE: overview
SCADE: state-flow design

- Define input and output signals
  - Specify types
  - Potential for ASN.1 types reuse

- Use predefined operators & SCADE blocks
  - Lustre operators + many additional
  - Drag & drop approach

- Potential use of extended library
  - Math functions, etc.
  - Ex: abs(), log(), ...
SCADE: state-flow example

Initial value
(← Lustre operator)

Additioner
(,+ Lustre operator)

Text expression

Previous value
(pre() Lustre operator)

valout
SCADE: corresponding Lustre node

node incrementer ()
  returns (valout : int)
let
  Valout = 0 → pre (valout) + 1;
tel
SCADE: data-flow simulation

Signal values

Input/output values

System cycles
Overview

• Flow-based approach, introduction to Lustre

• SCADE for data-flow specification

• SCADE state machines

• Code generation

• Conclusion & links
State machines characteristics

- **Input and output**
  - Use same types as data flow diagrams

- **States**
  - States and sub-states
  - Start in an initial state
  - Content = application behavior

- **Transitions**
  - From one state to another
  - Triggered by condition
SCADE: state machine example

When on, ison=true

When off, ison=false

Inputs & outputs

Transitions conditions

Julien Delange <julien dot delange at esa dot int>
SCADE: state machine simulation

Active state

State application content

Fired transition
Mixing state machines and data flows

- Execution semantics
  - Parallel execution

- Data flows in state machines
  - Output value state-dependent

- Data flows as input for state machines
  - Use output of DF as SM input
Mixing state machines and data flows (2)

Data-flow block

State-machine block
Overview

- Flow-based approach, introduction to Lustre
- SCADE for data-flow specification
- SCADE state machines
- Code generation
- Conclusion & links
Code generation overview

- Certifiable code
  - DO178 compliance

- Clean code, rigid structure!
  - Avoid Simulink/Matlab code generation trends!

- Interfacing potential with user-defined code
  - Clean code = easy integration
Code generation structure

- **Type** \texttt{outC<_operator__pkg>}
  - **Structure**
  - One member for each input/output/states
  - Other member for input/output/states computations

- **Reaction function**
  - Produce new data or compute new states
  - \texttt{void <operator__pkg> (outC<_operator__pkg>*)}

- **Init/reset function**
  - Put default struct values
  - \texttt{void <operator__reset__pkg> (outC<_operator__pkg>*)}
Code generation, files

- **Generated files**
  - `<operator__pkg>.h`: types & functions for code integration
  - `<operator__pkg>.c`: implementation of system behavior
  - `kcg_types.h`
    - mapping of SCADE types to C
    - application-specific types

- **SCADE-included files**
  - External libraries

- **Other files**
  - Generation/optimization reports
```c
#include "incrementer_mypkg.h"

int main()
{
    outC_incrementer_mypkg val;

    incrementer_reset_mypkg (&val);

    while (1)
    {
        incrementer_mypkg (&val);
        printf ("val=%d\n", val.valout);
        sleep (1);
    }
}
```
Overview

• Flow-based approach, introduction to Lustre

• SCADE for data-flow specification

• SCADE state machines

• Code generation

• Conclusion & links
Conclusion

- Application concerns abstraction
  - Close to engineering concerns
  - Simulation & validation facilities

- Automatic certified code generation
  - Automatic compliance with DO178
  - Reduce development time & costs

- Requires integration
  - Communication with system environment
  - Interaction with other languages
Perspectives

- Integration with system environment
  - Merge SCADE models with other application modeling approaches
  - Automatic integration of SCADE generated code

- But brings new problems
  - System consistency (types to be shared)
  - Management of heterogeneous environment

- Certification with other standards
  - ECSS, AUTOSAR, ...
About this course

• Introduction, not a complete lecture
• Cover most SCADe concepts
• For interested students, resources available on the internet
• See the links section

• Focused on main SCADe aspects and practical use
• Flow-based approach, state machine definition
• Relation with the Lustre language
• Code generation & certification concerns
Overview

• Flow-based approach, introduction to Lustré
• SCADe for data-flow specification
• SCADe state machines
• Code generation

Conclusions & links
Overview

• Flow-based approach, introduction to Lustre
• SCADe for data-flow specification
• SCADe state machines
• Code generation
• Conclusion & links
Lustre: rationale
• Safety-critical software must be deterministic
• System execution behaves as expected (specs.)
• Reaction to events and associated values
• Sensors, actuators
• Domain-specific and math approach
• System specified engineers
• Not programmers!
Transformation vs. reactive systems

- Transformational systems
  - Take input, produce outputs
  - Execution time not bounded
  - Ex: calculator, C program, etc.

- Reactive systems
  - React to signals
  - System rhythm is predefined
  - Ex: speed control system
Lustre: overview

• Data-flow approach
• New data are produced at each instant
• Data are computed at each reaction
• Many advantages
• Closed to maths
• Database dependence
Julien Delange <julien.delange@esa.int>

Data flow example

• Must consider data dependencies
• X depends on Y and Z
• W depends on X (and so, on Y and Z)
• New data produced at each instant

\[
X(t) = 2 \times Y(t) + Z(t)\\
W(t) = X(t) + 1 + Y + Z
\]
Lustre, flow-based synchronous language

- Software separated into NODES
- Take one or several input values
- Produces one or several output values at each reaction
- Uses usual types
  - Integer, boolean, etc.
- Lustre operators
  - Usual: +, -, mod, /, * etc ...
  - pre(val): value at previous reaction
  - current(val): value at current reaction
Lustré, syntax
• Expressions separated with ;
• Parallel execution
• Assignment =
  • X = 1
  • Initial value → X = 0 → prev(X) + 1;
  • Previous value prev()
  • X = prev(Y)
  • Current value curr()
  • X = curr(Y)
• Verification system values:
  • assert()
  • Assert (not X)
JULIEN DELANGE  < julien.delange@esa.int >

LUSTRE, SYNTAX (2)

• NODE CALL:
  node name (param1, param2, ...)

• X = my node (param1)

• Tuples
  (val1, val2) = (var3, var4)

• (X, Y) = (2, 3)
Lustre, syntax (3)

- Traditional boolean operators
  - and, or, not, xor
- Conditional if

\[ N = 0 \rightarrow \text{if } (Y) \text{ then } X \]

\[ N = 0 \rightarrow \text{if } (Y) \text{ then } X \text{ else } Z \]

- Filtering values

\[ Y = Z \text{ when } X \]
Lustre: first node

node COMPUTE (Y, Z : int)
return (X, W : int)
let
X = 2 * Y + Z;
W = X + 1;
end
Julien Delange <julien.delange@esa.int>

Lustre: EDGE node

edge (b: bool) returns (edge: bool)

let edge = false → b and not pre b;

<table>
<thead>
<tr>
<th>Inst</th>
<th>b</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>5</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>
Lustré: cyclic data dependencies

$X = \text{if } W \text{ then } Y \text{ else } Z;$

$Y = \text{if } W \text{ then } Z \text{ else } X;$

- Cyclic dependency
- Non-sense!
- Self-dependency
- A data cannot depend on itself!

$X = X + 1;$

$X = X + 1;$
Lustre: WATCHDOG1 node

WATCHDOG1 (set, reset, deadline: bool)
returns (alarm: bool)

var watchdog_is_on: bool;

let alarm = deadline and watchdog_is_on;

watchdog_is_on = false → if set then true
else if reset then false
else pre (watchdog_is_on);

assert not (set and reset);
tel
Julien Delange <julien.delange@esa.int>

Lustre: WATCHDOG2 node

node WATCHDOG2 (set, reset: bool, delay: int)
return (alarm: bool)

var
remaining_delay: int;
deadline: bool;
let
alarm = WATCHDOG1 (set, reset, deadline);
deadline = EDGE (remaining_delay = 0);
remaining_delay = if set then delay
else (0 -> pre (remaining_delay) - 1);
Julien Delange < julien.delange@esa.int>

Lustre: first node

node COMPUTE (Y, Z : int)
return (X, W : int)
let
X = 2 * Y + Z ; W = X + 1 ;
Overview

• Flow-based approach, introduction to Lustre
• SCADe for data-flow specification
• SCADe state machines
• Code generation
• Conclusion & links
Julien Delange <julien.delange@esa.int>

SCADE
• Flow-based design
• Similar as Lustre
• State machines integration
• Interaction with flows
• Specific graphical notation
• Drag & drop approach
SCADe: main features

• System design
• Data flows & state machines
• Predefined operators
• Simulation
  • Graphical simulation, automatic GUI integration
• Code generation
  • Certified code
• Integration with RTOS
SCADE: state-flow design

- Define input and output signals
- Specify types
- Potential for ASN.1 types reuse
- Use predefined operators & SCADE blocks
- Lustre operators + many additional
- Drag & drop approach
- Potential use of extended library
- Math functions, etc.
- Ex: abs(), log(), . . .
SCADE: state-flow example

Init value

\( \rightarrow \) Lustre operator

Addition

\( + \) Lustre operator

Text expression

Previous value

\( \text{prev}(\) Lustre operator \)
JULIEN DELANGE < julien.delange@esai.in >

SCADE: corresponding Lustre node

node increment ( ) returns ( valout : int )

let
Valout = 0 → pre ( valout ) + 1 ;
Overview

• Flow-based approach, introduction to Lustre
• SCADe for data-flow specification
• SCADe state machines
• Code generation
• Conclusion & links
State machine characteristics

- Input and output
- Use same types as data flow diagrams
- States
- States and sub-states
- Start in an initial state
- Content = application behavior
- Transitions
- From one state to another
- Triggered by condition
SCADE: state machine example

State Transitions

- When off, isOn = false
- When on, isOn = true

Inputs & Outputs
JULIEN DELANGE <julien.delange@esa.int>

SCADE: state machine simulation

Active state
Fired transition
State application content
Mixing state machines and data flows
• Execution semantics
• Parallel execution
• Data flows in state machines
• Output value state-dependent
• Data flows as input for state machines
• Use output of DF as SM input
Mixing state machines and data flows (2)

Data-flow block

State-machine block
OVERVIEW

• Flow-based approach, introduction to Lustre
• SCADe for data-flow specification
• SCADe state machines
• Code generation
• Conclusion & links
Code generation overview
• Certifiable code
• DO 178 compliance
• Clean code, rigid structure!
• Avoid Simulink/Matlab code generation trends!
• Interfacing potential with user-defined code
• Clean code = easy integration
Code generation structure

- Type operator _pkg
- Structure
  - One member for each input/output/states
  - Other member for input/output/computations
- Reaction function
  - Produce new data or compute new states
- void operator _pkg(out C _ operator _pkg *)
- Init/reset function
  - Put default struct values
- void operator _reset _pkg(out C _ operator _pkg *)
JULIEN DELANGE <julien.delange@esa.int>

CODE GENERATION, FILES

- Generated files
  - `operator__pkg.h`: types & functions for code integration
  - `operator__pkg.c`: implementation of system behavior
  - `kcg_types.h`: mapping of SCADe types to C
  - Application-specific types
  - SCADe included files
  - External libraries
  - Other files
  - Generation/optimization reports
# include "incrementer_mypkg.h"

int main() {
  incrementer_mypkg val;
  incrementer_reset_mypkg(&val);
  while (1) {
    incrementer_mypkg(&val);
    printf("val = %d \n", val.valout);
    sleep(1);
  }
}
Overview

• Flow-based approach, introduction to Luster
• SCAD e data-flow specification
• SCAD e state machines
• Code generation
• Conclusion & links
Conclusions

- Application concerns abstraction
- Close to engineering concerns
- Simulation & validation facilities
- Automatic certified code generation
- Automatic compliance with DO 178
- Reduce development time & costs
- Requires integration
- Communication with system environment
- Interaction with other languages
Perspectives

• Integration with system environment
• Merge SCADe models with other application modeling approaches
• Automatic integration of SCADe generated code
• But brings new problems
• System consistency (types to be shared)
• Management of heterogeneous environment
• Certification with other standards
• ECS, AUTOSAR, ...