SCADE 6, a Formal Language for Embedded Software Development

Jean-Louis Colaço
ANSYS System Business Unit

Theoretical Aspects of Software Engineering
14 Sep. 2017
ANSYS: World’s Leading Engineering Software Provider

FOCUSED
This is all we do.
Leading product technologies in all physics areas
Largest development team focused on simulation
and embedded software

TRUSTED
96 of the top 100
FORTUNE 500 Industrials
ISO 9001 and NQA-1 certified

STRONG
45000+
CUSTOMERS in every major industry segment

CAPABLE
2500+ employees
75 locations
40 countries

PROVEN
Recognized as one of the world’s MOST INNOVATIVE
AND FASTEST-GROWING COMPANIES*

INDEPENDENT
Long-term financial stability
CAD agnostic
Based on standards

LARGEST
3x The size of our nearest competitor
ANSYS System Business Unit

Model-Based Systems Engineering

System Safety Analysis
medini™ analyze

System Architecture

System Simulation & Digital Twins

Simplorer

Model-Based Software Engineering

3D Physics Simulation

System/WH Architecture

SW Components (FMI)

ROM
SCADE

- Safety Critical Application Development Environment
- Scade 6 is the underlying language of SCADE Suite
- belongs to the family of synchronous languages
- is a dialect of Lustre (data-flow oriented)
- includes major extensions in its version 6 (SCADE 6)
- is a DSL dedicated to the development of critical systems

P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice.
Lustre: a declarative language for programming synchronous systems.
Scade and safety critical applications

- most of safety critical applications are digital controllers;
- block diagrams and state machines are widely used in control engineering;
- good matching between language and diagrams (semantics and intuition);
- good properties of the language: runs in finite memory, deterministic;
- Scade compiler (code generator) is qualified for several standards: DO-178C (DO-330 TQL 1), EN-50128, IEC-61508.
  i.e. it can be used without having to verify its output.
Formal Methods in avionics standard


"Establishing a formal model of the software artifact of interest is fundamental to all formal methods. In general a model is an abstract representation of a given set of aspects of the software that is used for analysis, simulation, and/or code generation. In the context of this document, to be formal, a model should have an unambiguous, mathematically defined syntax and semantics. This makes it possible to use automated means to obtain guarantees that the model has certain specified properties."
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
Reactive System

A system that interacts continuously with its environment (physics, user, ...)

Reactive System

Environment
A reactive system is a function of sequences to sequences.

$i_0, i_1, \ldots, i_n, \ldots$  \rightarrow  \text{Reactive System}  \rightarrow  o_0, o_1, \ldots, o_n, \ldots
A reactive system is a function of sequences to sequences.

Mathematical viewpoint

$\mathcal{S}$

Reactive System

$i_0, i_1, \ldots, i_n, \ldots$

$o_0, o_1, \ldots, o_n, \ldots$

$\textsc{SCADE}$ is a language to define mutually recursive sequences.
Operational viewpoint

$S_0$

Reactive System

$i_0, i_1, \ldots, i_n, \ldots$

$o_0, o_1, \ldots, o_n, \ldots$
Operational viewpoint

Reactive System

$i_0, i_1, \ldots, i_n, \ldots$  $S_1$

$\sigma_0, \sigma_1, \ldots, \sigma_n, \ldots$
Operational viewpoint

Synchronous Reactive System

\( i_0, i_1, \ldots, i_n, \ldots \) \rightarrow S_n \rightarrow \( o_0, o_1, \ldots, o_n, \ldots \)
Operational viewpoint

A step:
- read inputs
- compute outputs
- update internal state

let $f$ be the function that computes one reaction:
$$o_n, S_{n+1} = f(i_n, S_n)$$

the code generator produces the function $f$ and the initial state $S_0$.
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
The data-flow kernel

Point-wise extension of combinatorial operators:

\[
\begin{array}{c|c|c|c|c|c|c}
  x & x_0 & x_1 & \ldots & x_n & \ldots \\
  y & y_0 & y_1 & \ldots & y_n & \ldots \\
  x + y & x_0 + y_0 & x_1 + y_1 & \ldots & x_n + y_n & \ldots \\
\end{array}
\]

\(x + y\) represents the sequence \((x_n + y_n)_{n \in \mathbb{N}}\)
The data-flow kernel

Point-wise extension of combinatorial operators:

\[
\begin{array}{c|c|c|c|c|c}
  \times & \times_0 & \times_1 & \ldots & \times_n & \ldots \\
  \times + y & x_0 + y_0 & x_1 + y_1 & \ldots & x_n + y_n & \ldots \\
\end{array}
\]

\(x + y\) represents the sequence \((x_n + y_n)_{n \in \mathbb{N}}\)
likewise for: \textbf{not}, \textbf{and}, \textbf{or}, -, *, \ldots
The data-flow kernel

Point-wise extension of combinatorial operators:

\[ \begin{array}{c|c|c|c|c|c|c|c|}
   & x_0 & x_1 & \ldots & x_n & \ldots \\
---|---|---|---|---|---|
  y  & y_0 & y_1 & \ldots & y_n & \ldots \\
  ---|---|---|---|---|---|
  x + y & x_0 + y_0 & x_1 + y_1 & \ldots & x_n + y_n & \ldots \\
\end{array} \]

\(x + y\) represents the sequence \((x_n + y_n)_{n \in \mathbb{N}}\)

likewise for: \textbf{not}, \textbf{and}, \textbf{or}, +, *, \ldots

Constants and literals are lifted to sequences:

\[ \begin{array}{c|c|c|c|c|c|c|}
   & 2 & 2 & 2 & \ldots & 2 & \ldots \\
---|---|---|---|---|---|---|
  x  & x_0 & x_1 & \ldots & x_n & \ldots \\
  ---|---|---|---|---|---|---|
  2 \ast x & 2 \ast x_0 & 2 \ast x_1 & \ldots & 2 \ast x_n & \ldots \\
\end{array} \]
The data-flow kernel

Unit delay:

<table>
<thead>
<tr>
<th>x</th>
<th>x_0</th>
<th>x_1</th>
<th>...</th>
<th>x_n</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre x</td>
<td>nil</td>
<td>x_0</td>
<td>...</td>
<td>x_{n-1}</td>
<td>...</td>
</tr>
</tbody>
</table>

let x represent the sequence \((x_n)\) \(n \in \mathbb{N}\), pre \(x\) represents the sequence \((p_n)\) \(n \in \mathbb{N}\) defined by:

\[ p_0 = \text{nil} \quad \text{and} \quad \forall n \in \mathbb{N}, \quad p_{n+1} = x_n \]
The data-flow kernel

Unit delay:

\[
\begin{array}{c|c|c|c|c|c}
\text{x} & x_0 & x_1 & \ldots & x_n & \ldots \\
\text{pre x} & nil & x_0 & \ldots & x_{n-1} & \ldots \\
\end{array}
\]

let \( x \) represent the sequence \((x_n)_{n \in \mathbb{N}}\), \( \text{pre x} \) represents the sequence \((p_n)_{n \in \mathbb{N}}\) defined by:

\[ p_0 = nil \text{ and } \forall n \in \mathbb{N}, p_{n+1} = x_n \]

where \( nil \) is an undefined value of the right type.
The data-flow kernel

Initialization:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>x₀</th>
<th>x₁</th>
<th>…</th>
<th>xₙ</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y₀</td>
<td>y₁</td>
<td>…</td>
<td>yₙ</td>
<td>…</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>y₀</td>
<td>y₁</td>
<td>…</td>
<td>yₙ</td>
<td>…</td>
</tr>
</tbody>
</table>

\[ x \rightarrow y \]

…
The data-flow kernel

Initialization:

\[
\begin{array}{c|c|c|c|c|c|c}
\text{x} & \text{x}_0 & \text{x}_1 & \cdots & \text{x}_n & \cdots \\
\text{y} & \text{y}_0 & \text{y}_1 & \cdots & \text{y}_n & \cdots \\
\text{x} \rightarrow \text{y} & \text{x}_0 & \text{y}_1 & \cdots & \text{y}_n & \cdots
\end{array}
\]

combined with \text{pre} to build a delayed stream without \text{nil}:

\[
\begin{array}{c|c|c|c|c|c|c}
\text{x} & \text{x}_0 & \text{x}_1 & \cdots & \text{x}_n & \cdots \\
\text{pre} & \text{y} & \text{nil} & \text{y}_0 & \cdots & \text{y}_{n-1} & \cdots \\
\text{x} \rightarrow \text{pre} \text{ y} & \text{x}_0 & \text{y}_0 & \cdots & \text{y}_{n-1} & \cdots
\end{array}
\]
The data-flow kernel

filtering with a clock:

<table>
<thead>
<tr>
<th>h</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x₀</td>
<td>x₁</td>
<td>x₂</td>
<td>x₃</td>
<td>x₄</td>
<td>...</td>
</tr>
<tr>
<td>x  when h</td>
<td>x₀</td>
<td>_</td>
<td>x₂</td>
<td>x₃</td>
<td>_</td>
<td>...</td>
</tr>
</tbody>
</table>
The data-flow kernel

filtering with a clock:

<table>
<thead>
<tr>
<th>h</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x₀</td>
<td>x₁</td>
<td>x₂</td>
<td>x₃</td>
<td>x₄</td>
<td>...</td>
</tr>
</tbody>
</table>

x when h

<table>
<thead>
<tr>
<th>h</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a₀</td>
<td>_</td>
<td>_</td>
<td>a₁</td>
<td>_</td>
<td>...</td>
</tr>
</tbody>
</table>

Extension on the clock of the clock:

<table>
<thead>
<tr>
<th>h</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>current a</td>
<td>a₀</td>
<td>a₀</td>
<td>a₀</td>
<td>a₁</td>
<td>a₁</td>
<td>...</td>
</tr>
</tbody>
</table>
Determinism of Lustre

determinist... if used with care!
Determinism of Lustre

determinist... if used with care!
Synchronous principles give deterministic parallel composition.
Determinism of **Lustre**

determinist... if used with care!
Synchronous principles give deterministic parallel composition.
**BUT** this is not the only source of non determinism:
the initial state must be well managed
and **Lustre** does not guarantee that! (*nil* in memories)
Determinism of Lustre

determinist... if used with care!
Synchronous principles give deterministic parallel composition. **BUT** this is not the only source of non determinism:
the initial state must be well managed
and Lustre does not guarantee that! (nil in memories)

**note:** this is not an issue to verify properties because either they are independent of
the initial state or they are falsifiable.
Scade 5 example
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
Genesis of **Scade 6**

Needs:
- Control (activation) structures: conditionals, automaton
- Arrays and primitives to use them
- Ensure determism (handling of the *nil* issue)
Solving non determinism

The case of **current** is hard to solve in general:

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>a₀</td>
<td>_</td>
<td></td>
</tr>
<tr>
<td>current</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>a₀</td>
<td>a₀</td>
<td>...</td>
</tr>
</tbody>
</table>
Solving non determinism

The case of **current** is hard to solve in general:

<table>
<thead>
<tr>
<th></th>
<th>false</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>a</td>
<td>_</td>
<td>_</td>
<td>_</td>
<td>a₀</td>
<td>_</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>nil</th>
<th>nil</th>
<th>nil</th>
<th>a₀</th>
<th>a₀</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**motto**: do not depend on a *model-checker* to state the correctness, use classical tools of programming language design:

- constructions (syntax, semantic);
- typing disciplines.
Solving non determinism

The case of \textit{current} is hard to solve in general:

\begin{tabular}{c|c|c|c|c|c|c|}
\hline
h & false & false & false & true & false & \ldots \\
\hline
a & _ & _ & _ & a_0 & _ & \ldots \\
\hline
\textit{current} & a & \textit{nil} & \textit{nil} & \textit{nil} & a_0 & a_0 & \ldots \\
\hline
\end{tabular}

\textbf{motto}: do not depend on a \textit{model-checker} to state the correctness, use classical tools of programming language design:

\begin{itemize}
  \item constructions (syntax, semantic);
  \item typing disciplines.
\end{itemize}

\textbf{proposition}:

\begin{itemize}
  \item replace \textit{current} and
  \item define a dedicated type system that ensures determinism.
\end{itemize}
An alternative to "current"

To avoid initialization issue, a common Lustre pattern is to use it combined with a test of the clock:

```plaintext
if h then current x else e
```

where $h$ is the clock of $x$. 
An alternative to "current"

To avoid initialization issue, a common Lustre pattern is to use it combined with a test of the clock:

\[
\text{if } h \text{ then current } x \text{ else } e
\]

where \( h \) is the clock of \( x \).

**proposition:** introduce a primitive that merges streams on complementary clocks.

---

Paul Caspi and Marc Pouzet  
Synchronous Kahn Networks.  

Grégoire Hamon  
Calcul d’horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML  
Thèse Université Pierre et Marie Curie, 14 Nov. 2002

Marc Pouzet  
2006
<table>
<thead>
<tr>
<th>h</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$a_0$</td>
<td>_</td>
<td>$a_1$</td>
<td>$a_2$</td>
<td>_</td>
<td>...</td>
</tr>
<tr>
<td>b</td>
<td>_</td>
<td>$b_0$</td>
<td>_</td>
<td>_</td>
<td>$b_1$</td>
<td>...</td>
</tr>
<tr>
<td>merge ($h; a; b$)</td>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$a_1$</td>
<td>$a_2$</td>
<td>$b_1$</td>
<td>...</td>
</tr>
</tbody>
</table>

In Lustre:

```
if $h$ then current $a$ else current $b$.
```

But merge does not introduce memories.
merge

<table>
<thead>
<tr>
<th>h</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a₀</td>
<td>_</td>
<td>a₁</td>
<td>a₂</td>
<td>_</td>
<td>...</td>
</tr>
<tr>
<td>b</td>
<td>_</td>
<td>b₀</td>
<td>_</td>
<td>_</td>
<td>b₁</td>
<td>...</td>
</tr>
<tr>
<td>merge (h; a; b)</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>...</td>
</tr>
</tbody>
</table>

in Lustre: if h then current a else current b.

But merge does not introduce memories.
Initialization analysis

**principle:** add a very simple type system with two types:

- 1 for a stream that may start by \textit{nil};
- 0 for a stream that is always defined.

**subsumption:** an argument of type 0 can always be used in a position where 1 is required.

**Property:** the outputs of the root node never contain a \textit{nil}.

Jean-Louis Colaço and Marc Pouzet.
Type-based Initialization Analysis of a Synchronous Data-flow Language.
Initialization analysis: pre and $\rightarrow$

pre : $0 \rightarrow 1$

pre (pre $x$) : cannot be typed

| $\text{pre (pre } x\text{)}$ | $\text{nil}$ | $\text{nil}$ | $x_0$ | $\ldots$ | $x_{n-2}$ | $\ldots$
|-------------------------------|-------------|-------------|-------|---------|-----------|---------|
| $v \rightarrow \text{pre (pre } x\text{)}$ | $v_0$ | $\text{nil}$ | $x_0$ | $\ldots$ | $x_{n-2}$ | $\ldots$

the nil in second position cannot be eliminated.

$\rightarrow : \forall \delta, \delta \times 1 \rightarrow \delta$
Example: Rising edge detection

```scade
defining dead
node rising_edge(a : bool) returns (o : bool)
  o = a and not pre a;

type: 0 → 1

node root(a, b : bool) returns (o : bool)
  o = rising_edge(a) or rising_edge(b);

type: 0 × 0 → 1
```
Example: Rising edge detection

```plaintext
% cat root_bad.scade
node rising_edge (a : bool) returns (o : bool)
    o = a and not pre a;

node root (a , b : bool) returns (o : bool)
    o = rising_edge (a) or rising_edge (b);

% kcg66 -root root root_bad.scade
SCADE Suite (R) KCG Code Generator 64-bit dev builds/proto/KCG-6.6i13-2-gadd2c4f
Copyright (C) Esterel Technologies 2002-2015 All rights reserved

*** Initialization Error (ERR_300): Initialization error
    at path root/
    Root node output must be well-initialized
No warning was found
1 error was found
No failure occurred
% 
```
Example: Rising edge detection

```
node rising_edge(a : bool) returns (o : bool)
  o = a and not pre a;

type: 0 → 1
```

```
node root(a, b : bool) returns (o : bool)
  o = false ->
      (rising_edge(a) or rising_edge(b));

type : 0 × 0 → 0
```
Example: Rising edge detection

```plaintext
% cat root_good.scade
define_node rising_edge (a : bool) returns (o : bool)
    o = a and not not a;
end define_node

define_node root (a, b : bool) returns (o : bool)
    o = false ->
        (rising_edge (a) or rising_edge (b));
end define_node

% kcg66 -root root root_good.scade
SCADE Suite (R) KCG Code Generator 64-bit dev builds/proto/KCG-6.6i13-2-gadd2c4f
Copyright (C) Esterel Technologies 2002-2015 All rights reserved

No warning was found
No error was found
No failure occurred
```
Need of control structure

In Lustre, only clocks allow to control computation; but they are hard to use. Users prefer to use *conditional activation*:

- **Scade 5**: `condact (c; N; e; i)`
- **Scade 6**: `(activate N every c initial default i) (e)`

**drawback**: needs to introduce an operator $N$ and does not allow to easily share a stream between different activations.
Scopes, control and explicit memories

- Introduction of guarded scopes: allows to select different sets of equations that produce the same streams.
- last ’x: access to the last value of x in its declaration scope (new construct).
Scopes, control and explicit memories

- Introduction of guarded scopes: allows to select different sets of equations that produce the same streams.
- `last ’x`: access to the last value of x in its declaration scope (new construct).
- Allows for different styles:

```
node counter () returns (o : int32)
  o = 1 -> pre (o + 1);
```
Scopes, control and explicit memories

- Introduction of guarded scopes: allows to select different sets of equations that produce the same streams.
- last ’x: access to the last value of x in its declaration scope (new construct).
- Allows for different styles:

```c
node counter () returns (o : int32)
  o = 1 -> pre (o + 1);
```

can also be written:

```c
node counter () returns (o : int32 last = 0)
  o = last ’o + 1;
```

o is manipulated as an explicit named memory.

... a flavour of imperative style.
Example: second degree equation

```plaintext
function second_degree(a, b, c: float64) returns (xr, xi, yr, yi: float64)
var delta : float64;
let
delta = b*b - 4 * a*c;
activate
if delta > 0
then
var d : float64;
let
d = sqrt(delta);
xr, xi = ((-b + d) / (2 * a), 0);
yr, yi = ((-b - d) / (2 * a), 0);
tel
else if delta = 0
then
let
xr, xi = (-b / (2 * a), 0);
yr, yi = (xr, xi);
tel
else if delta < 0
let
xr, xi = (-b / (2 * a), sqrt(-delta) / (2 * a));
yr, yi = (xr, -xi);
tel
returns xr, yr, xi, yi;
tel
```

Example: 

node sillywalk (c : bool) returns (o : int32 last = 0)
let
  activate
  if c then
    var inc : int32;
    let
      o = last 'o + inc;
      inc = (−17) → pre (36 → pre inc);
    tel
  else
    var inc : int32;
    let
      o = last 'o + inc;
      inc = (−3) → pre ((−33) → pre (25 → pre inc));
    tel
  returns o;
 tel
Scade 6 other constructs

- arrays and iterators
  
  Lionel Morel and Florence Maraninchi
  Arrays and contracts for the specification and analysis of regular systems

- modular reset of node instances
  
  Grégoire Hamon and Marc Pouzet
  Modular Resetting of Synchronous Data-flow Programs
  In ACM International conference on Principles of Declarative Programming (PPDP’00)

- hierarchical state machines
  
  Jean-Louis Colaço and Bruno Pagano and Marc Pouzet.
  A Conservative Extension of Synchronous Data-flow with State Machines.
  In ACM International Conference on Embedded Software (EMSOFT’05)
SCADE 6 example: digital watch
Scade 6 and Lustre kernels
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
Type checking

The type of the boolean values true and false is bool.

These three rules give the type associated to the three value kinds: char, integer and float.

An integer value can be used as a polymorphic literal. Its type must be a numerical type.

An operator \( f \) can be instantiated with an expression \( e \) if the type of \( e \) matches the types of the arguments of \( f \); the type of the instance is the output type of \( f \). The expression \( f(e) \) must be typable in a context that has at least the memory of \( f \) \((k_1 \leq k)\).

Operators arguments are of the right type.
Array accesses are within array bounds.
Clock checking

A polymorphic operator signature can be specialized by substituting the quantified clock variable $\alpha$ by a clock type $cl'\alpha$ and the carrier variables $X_i$ by carrier names $m_i$.

An operator $f$ with a clock signature $cl_1 \rightarrow cl_2$ can be instantiated with parameters $e$ of clock type $cl_1$.

The program can execute synchronously.

**Corollary:** no need to bufferize streams, can run with a finite amount of memory.

Jean-Louis Colaço and Marc Pouzet.
Clocks as first class abstract types.
In *Third International Conference on Embedded Software (EMSOFT’03)*
Causality analysis

An operator signature with quantified type variables can be replaced by a signature without universal quantification by replacing the quantified variables by fresh variables that are free in the typing environment.

No "instantaneous" cycle ($x_n = f(x_n)$)

Corollary: equations can be statically scheduled.
Initialization analysis

\[
\frac{H; H_{Last} \vdash e_1 : df_1^1 \times \cdots \times df_{n_1}^1 \quad H; H_{Last} \vdash e_2 : df_1^2 \times \cdots \times df_{n_2}^2}{H; H_{Last} \vdash e_1 -> e_2 : df_1^1 \times \cdots \times df_{n_1}^1}
\]

An *init* expression \((e_1 -> e_2)\) is well initialized if \(e_1\) and \(e_2\) are; the initialization type of the expression is the one of its first parameter.

\[
\frac{H; H_{Last} \vdash e : 0 \times \cdots \times 0}{H; H_{Last} \vdash \text{pre } e : 1 \times \cdots \times 1}
\]

Outputs are always defined (no *nil*).

**Corollary:** determinism.

Jean-Louis Colaco and Marc Pouzet.
Type-based Initialization Analysis of a Synchronous Data-flow Language.
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
**Scade 6 Compiler organization**

1. see previous 4 slides

2. Jean-Louis Colaço and Bruno Pagano and Marc Pouzet. 
   A Conservative Extension of Synchronous Data-flow with State Machines. 
   In ACM International Conference on Embedded Software (EMSOFT'05)

3. D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, 
   Clock-directed Modular Code Generation of Synchronous Data-flow Languages. 
Implementation of the qualified compiler (KCG)

- OCaml (≈ 50Klocs);
- with specific developments: code coverage tool for OCaml, simplified runtime with a Stop&Copy GC;
- formalized static semantics used as a precise specification (≈ 100p);
- based on a standards process: plans, specification, design (≈ 1000p), dev., unit tests (≈ 500Klocs), tests and reviews.

Certified development tools implementation in Objective Caml.

In International Conference on Functional Programming Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
Formal Verification of SCADE models

A SCADE model involves a bounded amount of memory
⇒ it represents a finite state system
⇒ model checking techniques apply
Formal Verification of Scade models

A Scade model involves a bounded amount of memory
⇒ it represents a finite state system
⇒ model checking techniques apply

Safety: something bad (undesirable) cannot happen
e.g. "Train doors cannot open while the train is rolling."

Liveness: something good (hoped-for) will eventually happen
e.g. "The train will eventually leave the station."
Formal Verification of Scade models

A Scade model involves a bounded amount of memory
⇒ it represents a finite state system
⇒ model checking techniques apply

Safety: something bad (undesirable) cannot happen
e.g. "Train doors cannot open while the train is rolling."

Liveness: something good (hoped-for) will eventually happen
e.g. "The train will eventually leave the station."

- A safety property expresses in Scade as a Boolean stream;
- proving it consists in verifying that this stream is constant and equal to true.

N. Halbwachs, F. Lagnier and C. Ratel.
Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre.
SCADE design verifier

- Based on Prover Technology proof engine

- SAT based model-checker: BMC, $k$—induction.

- Supports:
  - bounded integers (bitblasting).
  - unbounded integers.
  - rationals, used to support floats but not a safe abstraction.

- The translation from SCADE 6 to the engine (TECLA/HLL) is based on KCG.

M. Sheeran, S. Singh and G. Stalmark.
Checking safety properties using induction and a SAT-solver.
FMCAD 2000
Formal Verification in Embedded Software Industry

- Is a *must have* for SCADE evaluations.
- Main limitations to deployment:
  - Skills and patience (fantasy of push button solution).
  - Limited capabilities of the tool on numerical aspects (floats and non-linearities).
  - Lack of clear positioning in existing processes and standards.
- Successes in railway transportation:
  - RATP recommends the usage of formal verification to their suppliers; once skilled some use it for other project.
  - Order of magnitude of SAT instances: $10^6$ variables and $10^7$ clauses.
Presentation of the magic trick in G.Huet paper:

Why is this a card trick? Our boolean words are card decks, with true for red and false for black. Take an even deck $x$, arranged alternatively red, black, red, black, etc. Ask a spectator to cut the deck, into sub-decks $u$ and $v$. Now shuffle $u$ and $v$ into a new deck $w$. When shuffling, note carefully whether $u$ and $v$ start with opposite colors or not. If they do, the resulting deck is composed of pairs red-black or black-red; otherwise, you get the property by first rotating the deck by one card. The trick is usually played by putting the deck behind your back after the shuffle, to perform “magic”. The magic is either rotating or doing nothing. When showing the pairing property, say loudly “red black red black...” in order to confuse in the spectator’s mind the weak paired property with the strong alternate one.

There is a variant. If the cut is favorable, that is if $u$ and $v$ are opposite, just go ahead showing the pairing, without the “magic part.” If the spectator says that he understands the trick, show him the counter-example in the non-favorable case. Of course now you have to leave him puzzled, and refuse to redo the trick.

G. Huet.
Example: The Gilbreath Trick

- take a card deck where card color alternate;
- split it in two;
- ensure the bottom cards of the two sub-decks have different colors;
- riffle shuffle them.

Property:
the resulting deck is a list of pairs red-black or black-red.
Example: The Gilbreath Trick

The property is implied by the following one on Boolean streams:

\textit{if} s1 \textit{and} s2 \textit{be two alternate streams starting with different values; let } o \textit{be a stream built by “riffle shuffling” s1 and s2; then } o \textit{is such that it is a succession of pairs of different values.}

G. Huet.
The Gilbreath Trick: A case study in axiomatisation and proof development in the Coq Proof Assistant.
In \textit{Proceedings, Second Workshop on Logical Frameworks, Edinburgh, May 1991.}
**Example: The Gilbreath Trick**

```plaintext
node Gilbreath_stream (clock c:bool) returns (o, property: bool)
var
  s1 : bool when c;
  s2 : bool when not c;
  half : bool;
let
  s1 = (false when c) -> not (pre s1);
  s2 = (true when not c) -> not (pre s2);
  o = merge (c; s1; s2);
  half = false -> (not pre half);
  property = true -> not (half and (o = pre o));
tel
```

G. Huet.
The Gilbreath Trick: A case study in axiomatisation and proof development in the Coq Proof Assistant.
Synchronous Reactive System

Lustre/Scade 5

Scade 6

Type systems

Qualified Compiler (KCG)

Formal Verification of Scade 6 models

Conclusion
### Timeline of SCADE and influences

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td>Lustre</td>
</tr>
<tr>
<td>1986</td>
<td>SAGA-CG (Schneider Electric)</td>
</tr>
<tr>
<td>1988</td>
<td>SAO (Airbus)</td>
</tr>
<tr>
<td>1995</td>
<td>SCADE</td>
</tr>
<tr>
<td>2000</td>
<td>Lucid Synchrone</td>
</tr>
<tr>
<td>2008</td>
<td>Esterel</td>
</tr>
<tr>
<td>2011</td>
<td>Zélus (<a href="http://zelus.di.ens.fr/">http://zelus.di.ens.fr/</a>)</td>
</tr>
<tr>
<td>2015</td>
<td>Scade 6</td>
</tr>
<tr>
<td>2017</td>
<td>Hybrid Proto</td>
</tr>
</tbody>
</table>

© ANSYS, Inc.
Conclusion

- Use of state of the art programming language principles for an industrial qualified tool (> 100 avionic systems certified);
- Implementation in OCaml;
- Further step: certification in Coq and DO-330 qualification.

A Formally Verified Compiler for Lustre.
In International Conference on Programming Language, Design and Implementation (PLDI)

X. Leroy,
How much is a mechanized proof worth, certification-wise?
In Principles in Practice, January 2014
Conclusion

- Use of state of the art programming language principles for an industrial qualified tool (> 100 avionic systems certified);
- Implementation in OCaml;
- Further step: certification in Coq and DO-330 qualification.

A Formally Verified Compiler for Lustre.
In *International Conference on Programming Language, Design and Implementation (PLDI)*

X. Leroy,
How much is a mechanized proof worth, certification-wise?
In *Principles in Practice, January 2014*

Long, fruitful and continuing collaboration with Marc Pouzet.
**Scade Academic Program**

http://www.esterel-technologies.com/scade-academic-program/

**Contact:** scade-academics@ansys.com

© ANSYS, Inc.