

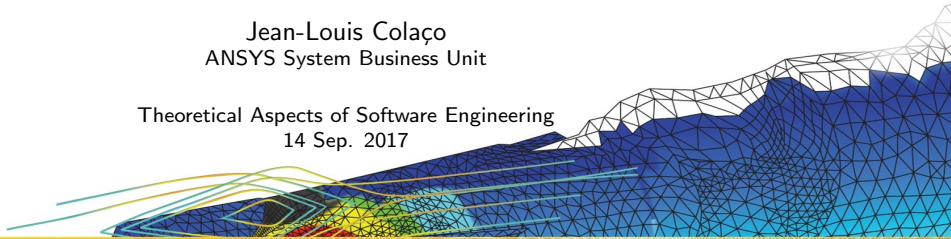
The ANSYS logo is displayed in a black rectangular box. The word "ANSYS" is written in a bold, sans-serif font. The letters "AN" are white, and "SYS" is yellow. A small registered trademark symbol (®) is located to the upper right of the "S".

ANSYS®

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 a every
 major industry segment

CAPABLE



PROVEN

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



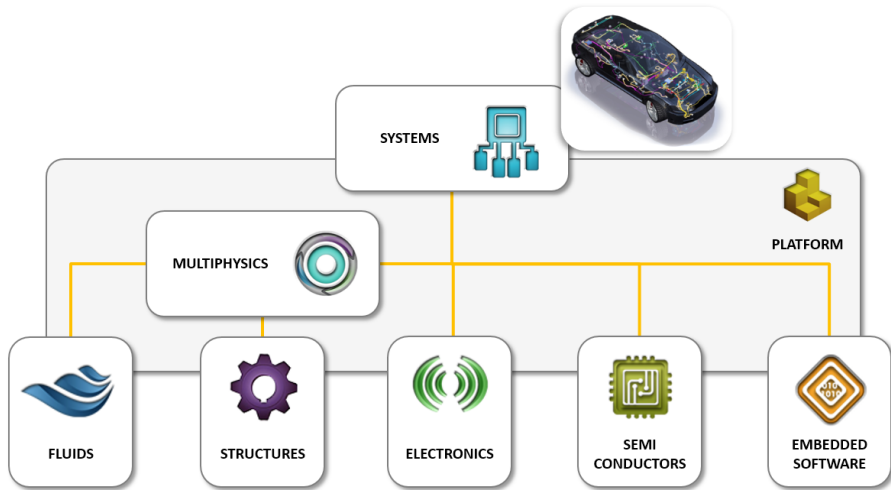
LARGEST

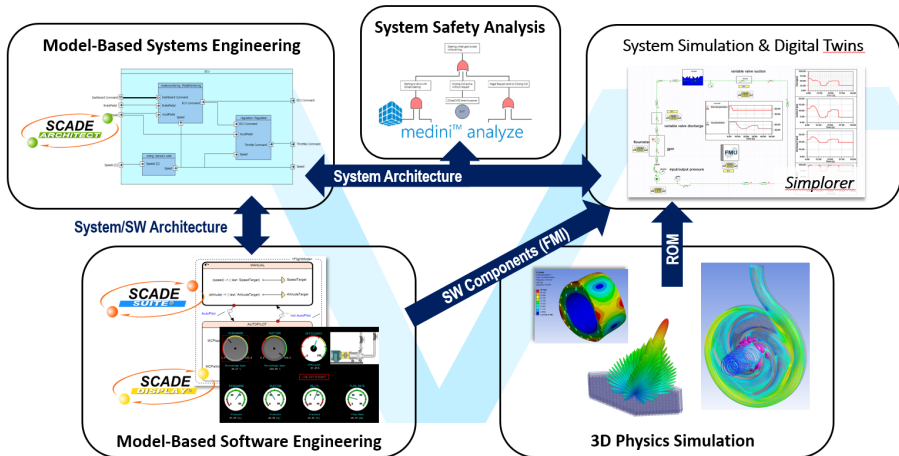
3x The size of our nearest
 competitor

INDEPENDENT

Long-term financial stability
 CAD agnostic
 Based on standards

ANSYS Simulation Platform Overview





- ▶ Safety **C**ritical **A**pplication **D**evelopment **E**nvironment
- ▶ 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.

In *14th ACM Symposium on Principles of Programming Languages*. 1987.

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.

From *Formal Methods Supplement to DO-178C and DO-278A* (DO-333, Dec. 2011):

"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

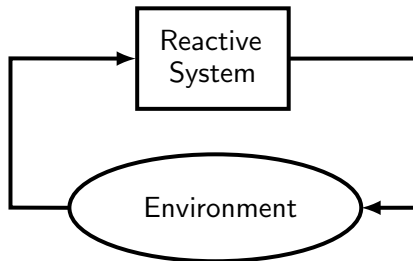
Type systems

Qualified Compiler (KCG)

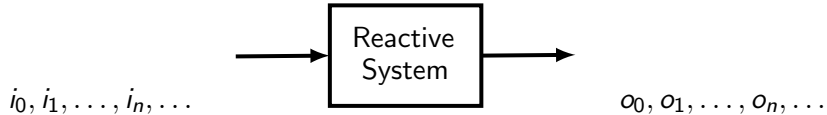
Formal Verification of SCADE 6 models

Conclusion

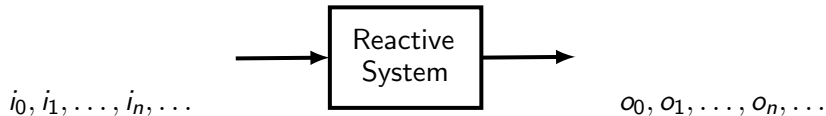
A system that interacts continuously with its environment (physics, user, ...)



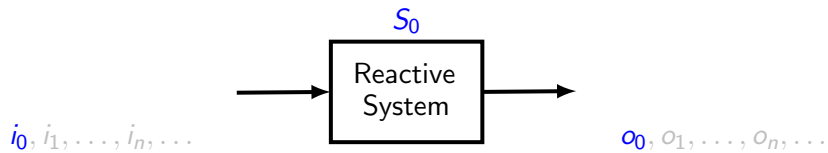
A reactive system is a function of sequences to sequences.

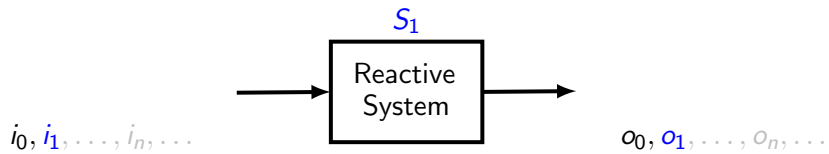


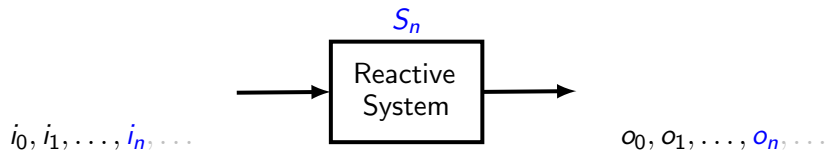
A reactive system is a function of sequences to sequences.

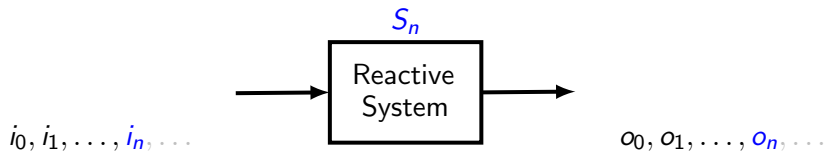


SCADE is a language to define mutually recursive sequences.









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

Point-wise extension of combinatorial operators:

x	x_0	x_1	\dots	x_n	\dots
y	y_0	y_1	\dots	y_n	\dots
$x + y$	$x_0 + y_0$	$x_1 + y_1$	\dots	$x_n + y_n$	\dots

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

Point-wise extension of combinatorial operators:

x	x_0	x_1	...	x_n	...
y	y_0	y_1	...	y_n	...
x + y	$x_0 + y_0$	$x_1 + y_1$...	$x_n + y_n$...

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

likewise for: **not**, **and**, **or**, $-$, $*$, ...

The data-flow kernel

Point-wise extension of combinatorial operators:

x	x_0	x_1	\dots	x_n	\dots
y	y_0	y_1	\dots	y_n	\dots
$x + y$	$x_0 + y_0$	$x_1 + y_1$	\dots	$x_n + y_n$	\dots

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

likewise for: **not**, **and**, **or**, $-$, $*$, \dots

Constants and literals are lifted to sequences:

2	2	2	\dots	2	\dots
x	x_0	x_1	\dots	x_n	\dots
$2 * x$	$2 * x_0$	$2 * x_1$	\dots	$2 * x_n$	\dots

Unit delay:

	x	x ₀	x ₁	...	x _n	...
pre	x	<i>nil</i>	x ₀	...	x _{n-1}	...

Unit delay:

$$\frac{\mathbf{x} \parallel \begin{array}{|c|c|c|c|c|} \hline x_0 & x_1 & \dots & x_n & \dots \\ \hline \end{array}}{\mathbf{pre} \ \mathbf{x} \parallel \begin{array}{|c|c|c|c|c|} \hline nil & x_0 & \dots & x_{n-1} & \dots \\ \hline \end{array}}$$

let \mathbf{x} represent the sequence $(x_n)_{n \in \mathbb{N}}$, $\mathbf{pre} \ \mathbf{x}$ 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.

Initialization:

	x	x ₀	x ₁	...	x _n	...
	y	y ₀	y ₁	...	y _n	...
x →	y	x ₀	y ₁	...	y _n	...

Initialization:

x	x_0	x_1	...	x_n	...
y	y_0	y_1	...	y_n	...
x \rightarrow y	x_0	y_1	...	y_n	...

combined with **pre** to build a delayed stream without *nil*:

x	x_0	x_1	...	x_n	...
pre y	<i>nil</i>	y_0	...	y_{n-1}	...
x \rightarrow pre y	x_0	y_0	...	y_{n-1}	...

filtering with a clock:

	h	true	false	true	true	false	...
	x	x_0	x_1	x_2	x_3	x_4	...
x when	h	x_0	-	x_2	x_3	-	...

filtering with a clock:

	h	true	false	true	true	false	...
	x	x_0	x_1	x_2	x_3	x_4	...
x when	h	x_0	-	x_2	x_3	-	...

Extension on the clock of the clock:

	h	true	false	false	true	false	...
	a	a_0	-	-	a_1	-	...
current	a	a_0	a_0	a_0	a_1	a_1	...

determinist. . . if used with care!

determinist. . . if used with care!

Synchronous principles give deterministic parallel composition.

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)

determinist. . . if used with care!

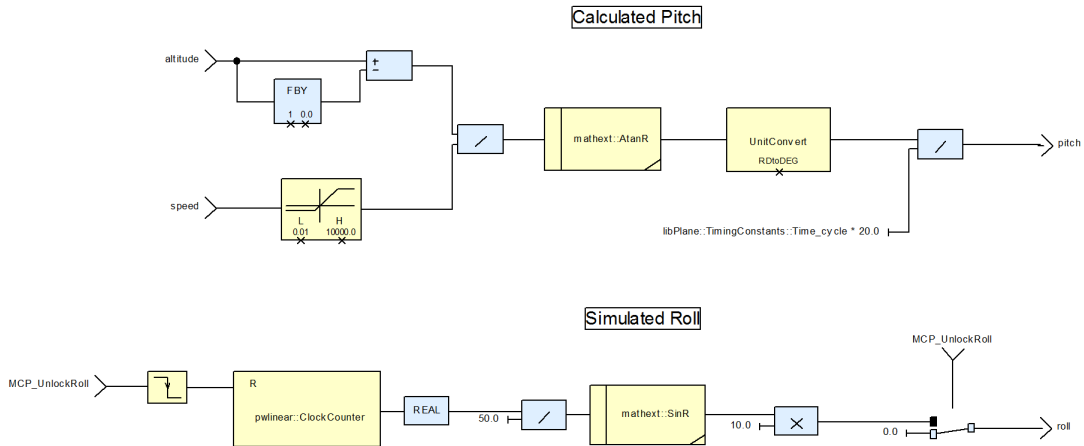
Synchronous principles give deterministic parallel composition.

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.



Synchronous Reactive System

LUSTRE/SCADE 5

SCADE 6

Type systems

Qualified Compiler (KCG)

Formal Verification of SCADE 6 models

Conclusion

Needs:

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

Solving non determinism

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

	h		false		false		false		true		false		...
	a		-		-		-		a_0		-		...
current	a		<i>nil</i>		<i>nil</i>		<i>nil</i>		a_0		a_0		...

Solving non determinism

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

	h		false		false		false		true		false		...
	a		-		-		-		a_0		-		...
current	a		<i>nil</i>		<i>nil</i>		<i>nil</i>		a_0		a_0		...

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 **current** is hard to solve in general:

	h	false	false	false	true	false	...
	a	-	-	-	a_0	-	...
current	a	<i>nil</i>	<i>nil</i>	<i>nil</i>	a_0	a_0	...

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

- ▶ constructions (syntax, semantic);
- ▶ typing disciplines.

proposition:

- ▶ replace **current** and
- ▶ define a dedicated type system that ensures determinism.

An alternative to "current"

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

```
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:

```
if h then current x 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.

In ACM SIGPLAN International Conference on Functional Programming (ICFP), Philadelphia, Pennsylvania, May 1996.

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

Lucid Synchrone version 3.0, Tutorial and Reference Manual.

2006

h	true	false	true	true	false	...
a	a_0	-	a_1	a_2	-	...
b	-	b_0	-	-	b_1	...
merge (h; a; b)	a_0	b_0	a_1	a_2	b_1	...

h	true	false	true	true	false	...
a	a_0	-	a_1	a_2	-	...
b	-	b_0	-	-	b_1	...
merge (h; a; b)	a_0	b_0	a_1	a_2	b_1	...

in LUSTRE: **if** h **then current** a **else current** b.

But **merge** does not introduce memories.

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

- ▶ **1** for a stream that may start by *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 *nil*.

Jean-Louis Colaço and Marc Pouzet.

Type-based Initialization Analysis of a Synchronous Data-flow Language.

International Journal on Software Tools for Technology Transfer (STTT), Vol.6, Num.3, November 2004.

Initialization analysis: pre and ->

pre : $0 \rightarrow 1$

pre (pre x) : cannot be typed

$$\frac{\text{pre (pre x)} \parallel \begin{array}{c|c|c|c|c} \text{nil} & \text{nil} & x_0 & \dots & x_{n-2} & \dots \end{array}}{v \rightarrow \text{pre (pre x)} \parallel \begin{array}{c|c|c|c|c} v_0 & \text{nil} & x_0 & \dots & x_{n-2} & \dots \end{array}}$$

the *nil* in second position cannot be eliminated.

-> : $\forall \delta, \delta \times 1 \rightarrow \delta$

Example: Rising edge detection

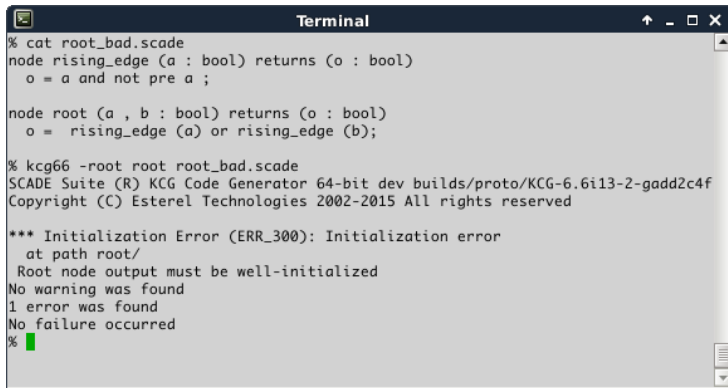
```
node rising_edge(a : bool) returns (o : bool)  
  o = a and not pre a;
```

type: **0** → **1**

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

type : **0** × **0** → **1**

Example: Rising edge detection



```
Terminal
% 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

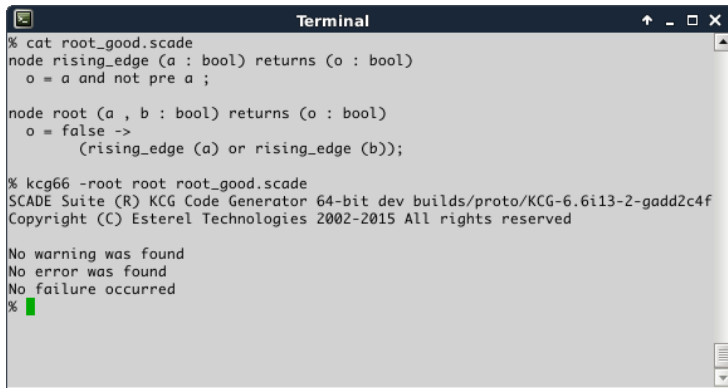
```
node rising_edge(a : bool) returns (o : bool)
  o = a and not pre a;
```

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



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

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

% 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
% █
```

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

- ▶ SCADE 5: **conduct** (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:

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

can also be written:

```
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

```
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 — 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

```

c	true	true	true	false	false	false	false	true	true	false	false	false	true	false	false	false	false	false	true	true	false	true	true	true	...		
_then/inc	-17	36	-17	-	-	-	-	36	-17	-	-	-	36	-	-	-	-	-	-17	36	-	-17	36	-17	36	...	
_else/inc	-	-	-	-3	-33	25	-3	-	-	-33	25	-3	-	-33	25	-3	-33	25	-3	-	-	-33	-	-	-	...	
o	-17	19	2	-1	-34	-9	-12	24	7	-26	-1	-4	32	-1	24	21	-12	13	10	-7	29	-4	-21	15	-2	34	...

▶ arrays and iterators

Lionel Morel and Florence Maraninchi

Arrays and contracts for the specification and analysis of regular systems

In *Proceedings. Fourth International Conference on Application of Concurrency to System Design, 2004. ACSD 2004.*

▶ 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)*

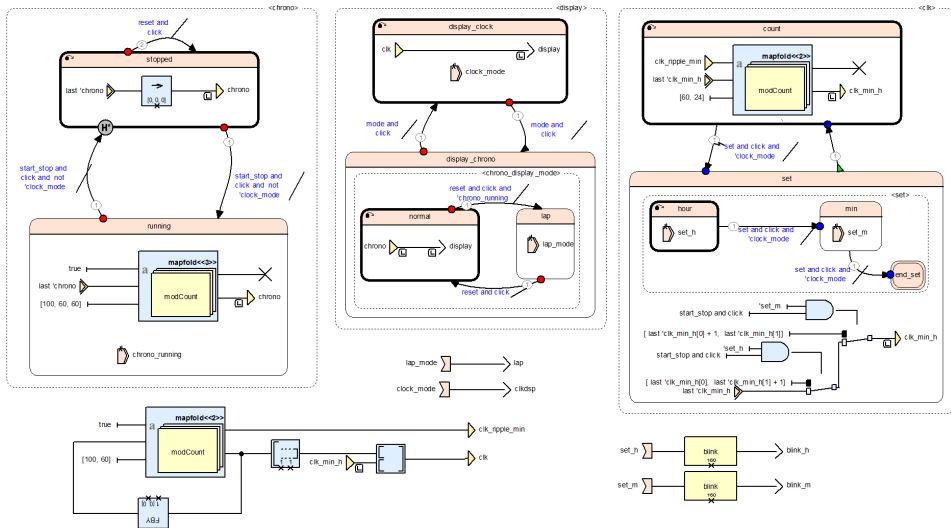
▶ hierarchical state machines

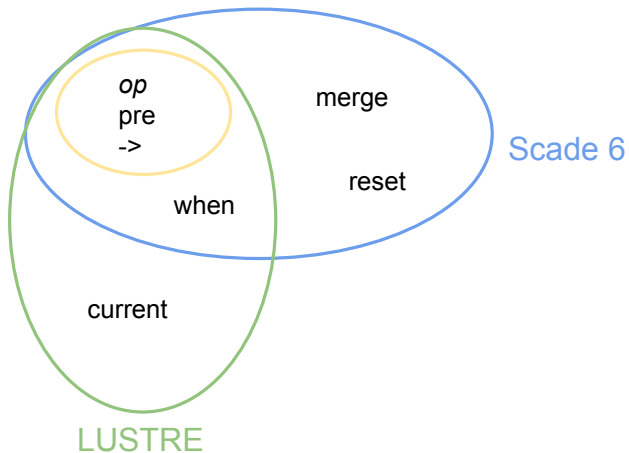
Jean-Louis Colaço and Bruno Pagano and Marc Pouzet.

A Conservative Extension of Synchronous Data-flow with State Machines.

In *ACM International Conference on Embedded Software (EMSOFT'05)*

SCADE 6 example: digital watch





Synchronous Reactive System

LUSTRE/SCADE 5

SCADE 6

Type systems

Qualified Compiler (KCG)

Formal Verification of SCADE 6 models

Conclusion

Type checking

$$\frac{[\text{LRM-149}]}{(\text{TRUE})} \frac{}{H; S_z; C \vdash^0 \text{true} : \text{bool}} \quad \frac{[\text{LRM-150}]}{(\text{FALSE})} \frac{}{H; S_z; C \vdash^0 \text{false} : \text{bool}}$$

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

$$\frac{[\text{LRM-151}]}{(\text{CHAR})} \frac{}{H; S_z; C \vdash^0 \text{CHAR} : \text{char}} \quad \frac{[\text{LRM-152}]}{(\text{INTEGER})} \frac{}{H; S_z; C \vdash^0 \text{INTEGER} : \text{int}} \quad \frac{[\text{LRM-153}]}{(\text{REAL})} \frac{}{H; S_z; C \vdash^0 \text{FLOAT} : \text{real}}$$

These three rules give the type associated to the three value kinds: CHAR, INTEGER and FLOAT.

$$\frac{[\text{LRM-643}]}{(\text{POLYMORPHIC LITERAL})} \frac{}{H; S_z; C \vdash^0 (\text{INTEGER} : 't') : 't \text{ where } 't \text{ is num}}$$

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

$$\frac{[\text{LRM-154}]}{(\text{INSTANCE})} \frac{H; S_z; C \vdash^k f : \tau_1 \xrightarrow{k_1} \tau_2, k_1 \leq k \quad H; S_z; C \vdash^k e : \tau'_1 \quad C \vdash \tau'_1 \sqsubseteq \tau_1}{H; S_z; C \vdash^k f(e) : \tau_2}$$

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.

$$\text{(CLK OPERATOR SPECIALISATION)} \frac{\text{[LRM-252]} \quad H \vdash^{clf} f : \forall \alpha. \forall X_1, \dots, X_n. cl_1 \longrightarrow cl_2}{H \vdash^{clf} f : cl_1 \longrightarrow cl_2 [clf' / \alpha] [m_i / X_i]_{i \in [1..n]}}$$

A polymorphic operator signature can be specialized by substituting the quantified clock variable α by a clock type clf' and the carrier variables X_i by carrier names m_i .

$$\text{(CLK INSTANCE)} \frac{\text{[LRM-253]} \quad H \vdash^{clf} f : cl_1 \longrightarrow cl_2 \quad H \vdash^{clf} e : cl_1}{H \vdash^{clf} f(e) : cl_2}$$

An operator f with a clock signature $cl_1 \longrightarrow cl_2$ 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)*

$$\text{(DEP OPERATOR SPECIALISATION)} \frac{\text{[LRM-330]} \quad H; H_{last}; W; C \vdash f : \forall \gamma_1, \dots, \gamma_n. \gamma_1 \times \dots \times \gamma_n \longrightarrow ct \quad \forall i \in [1..n], \gamma'_i \notin FCV(H)}{H; H_{last}; W; C \vdash f : \gamma'_1 \times \dots \times \gamma'_n \longrightarrow ct[\gamma'_i/\gamma_i]_{i \in [1..n]}}$$

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.

$$\text{(DEP INSTANCE)} \frac{\text{[LRM-331]} \quad H; H_{last}; W; C \vdash f : ctf_1 \times \dots \times ctf_n \longrightarrow ct \quad H; H_{last}; W; C \vdash e : ctf'_1 \times \dots \times ctf'_n}{H; H_{last}; W; (C \cup \{ctf_i \supseteq ctf'_i / i \in [1..n]\}) \vdash f(e) : ct}$$

The causality type of an operator instantiation is the causality type of the outputs if the inputs satisfy the constraint of being bigger than the type of the argument i.e. if the inputs of this instance depend on the flows represented by e .

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

Corollary: equations can be statically scheduled.

Inspired by:

Pascal Cuoq and Marc Pouzet

Modular Causality in a Synchronous Stream Language.

In *European Symposium on Programming (ESOP'01)*

Initialization analysis

$$\frac{\text{[LRM-431]} \quad H; H_{Last} \vdash e_1 : df_1^1 \times \dots \times df_n^1 \quad H; H_{Last} \vdash e_2 : df_1^2 \times \dots \times df_n^2}{\text{(INIT ARROW)} \quad H; H_{Last} \vdash e_1 - > e_2 : df_1^1 \times \dots \times df_n^1}$$

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.

$$\frac{\text{[LRM-432]} \quad H; H_{Last} \vdash e : \underbrace{0 \times \dots \times 0}_n}{\text{(INIT PRE)} \quad H; H_{Last} \vdash \mathbf{pre} \ e : \underbrace{1 \times \dots \times 1}_n}$$

Outputs are always defined (no *nil*).

Corollary: determinism.

Jean-Louis Colaço and Marc Pouzet.

Type-based Initialization Analysis of a Synchronous Data-flow Language.

International Journal on Software Tools for Technology Transfer (STTT), Vol.6, Num.3, November 2004.

Synchronous Reactive System

LUSTRE/SCADE 5

SCADE 6

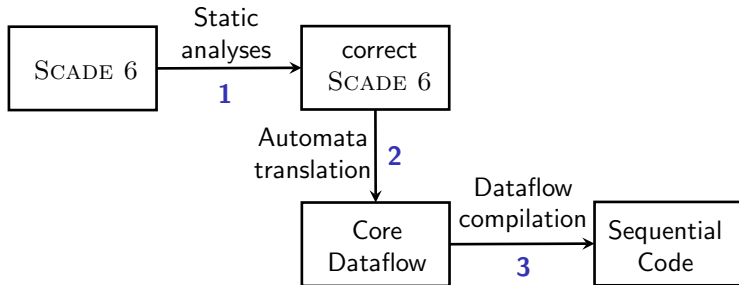
Type systems

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.
In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), Tucson, Arizona, June 2008*

Implementation of the qualified compiler (KCG)

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

B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J-L. Colaço, T. Moniot and P. Wang.

Certified development tools implementation in Objective Caml.

In *International Symposium on Practical Aspects of Declarative Languages PADL 08*. LNCS. Springer-Verlag, January 2008.

B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J-L. Colaço, T. Moniot, P. Wang and P. Manoury.

Experience Report: Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework

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.

In IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems. September 1992.

- ▶ 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

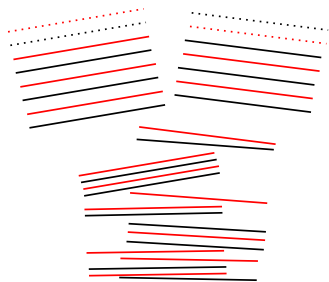
Formal Verification in Embedded Software Industry

- ▶ 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: http://projects.laas.fr/IFSE/FMF/J4/slides/P07_Evguenia_Dmitrieva.pdf
 - ▶ 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.

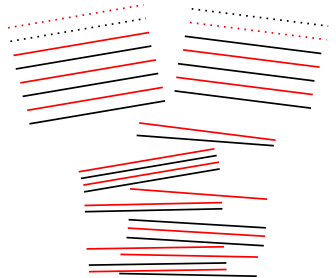
The Gilbreath Trick: A case study in axiomatisation and proof development in the Coq Proof Assistant.
In Proceedings, Second Workshop on Logical Frameworks, Edinburgh, May 1991.

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**.*



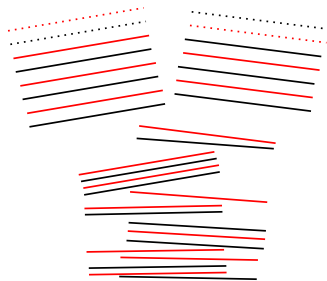
G. Huet.

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

Example: The Gilbreath Trick

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

if s_1 and s_2 be two alternate streams starting with different values; let o be a stream built by “riffle shuffling” s_1 and s_2 ; then o is such that it is a succession of pairs of different values.



G. Huet.

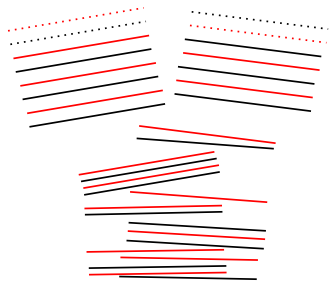
The Gilbreath Trick: A case study in axiomatisation and proof development in the Coq Proof Assistant.
In *Proceedings, Second Workshop on Logical Frameworks, Edinburgh, May 1991*.

Example: The Gilbreath Trick

```
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.
In Proceedings, Second Workshop on Logical Frameworks, Edinburgh, May 1991.

Synchronous Reactive System

LUSTRE/SCADE 5

SCADE 6

Type systems

Qualified Compiler (KCG)

Formal Verification of SCADE 6 models

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.

T. Bourke, P.-E. Dagand, M. Pouzet, and L. Rieg. T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet and L. Rieg
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

- ▶ 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.

T. Bourke, P.-E. Dagand, M. Pouzet, and L. Rieg. T. Bourke, L. Brun, P.-E. Dagand, X. Leroy, M. Pouzet and L. Rieg
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.



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

Contact: scade-academics@ansys.com