Scade 6: A Formal Language for Embedded Critical Software Development

Jean-Louis Colaço, Bruno Pagano, Marc Pouzet

Scade 6: A Formal Language for Embedded Critical Software Development

(Invited Paper)

Jean-Louis Colaço
ANSYS/Esterel-Technologies,
Jean-Louis.Colaco@ansys.com

Bruno Pagano
ANSYS/Esterel-Technologies,
Bruno.Pagano@ansys.com

Marc Pouzet
UPMC/ENS/INRIA Paris
Marc.Pouzet@ens.fr

***Abstract*—SCADE is a high-level language and environment for developing safety critical embedded control software. It is used for more than twenty years in various application domains like avionics, nuclear plants, transportation, automotive. SCADE has been founded on the synchronous data-flow language Lustre invented by Caspi and Halbwachs. In the early years, it was mainly seen as a graphical notation for Lustre but with the unique and key addition of a code generator qualified with the highest standards for safety critical applications.**

In 2008, a major revision based on the new language 'Scade 6' was released. This language originally combines the Lustre data-flow style with control structures borrowed from Esterel and SyncCharts, compilation and static analyses from Lucid Synchrone to ensure safety properties. This expressiveness increase for SCADE together with a qualified code generator have dramatically widened the scope of applications developed with.

While previous publications have described some of its language constructs and compiler algorithms, no reference publication on 'Scade 6' existed so far. In this paper, we come back to the decisions made for its design, illustrate the main language features, static analyses, and the compiler organization in the context of a qualification process.

I. Introduction

Synchronous languages [1] were introduced about thirty years ago by the concomitant work on three academic languages: SIGNAL [2], ESTEREL [3] and LUSTRE [4]. These *domain specific languages* were targeted for real-time control software, allowing to write a modular and mathematically precise system specification, to simulate, test and verify it, and to automatically translate it into embedded executable code.

They were founded on the *synchronous approach* [5] where a system is modeled ideally, with communications/computations supposed to be instantaneous, formally checking on the model important safety properties like determinism, deadlock freedom, the ability to generate an implementation that runs in bounded time and space, and verifying *a posteriori* that this implementation (software or hardware) is fast enough.

These foundations immediately raised the interest of industries having to deal with safety critical applications implemented in software or hardware, in particular those assessed by independent authorities and following certification standards [6]. This is the context in which SCADE [1] was initiated in the mid nineties, with the support of two companies, Airbus and Merlin Gerin, by a collaboration between the research laboratory VERIMAG in Grenoble, and the software editor VERILOG [7]. Since 2000, SCADE is developed by ANSYS/ESTEREL-TECHNOLOGIES. [2]

In the early years, the underlying language of SCADE was essentially LUSTRE V3 [8], augmented with a few specific features requested by users but minor in terms of expressiveness, to which was added a graphical editor. This situation held up to the version 5 of SCADE. To support the development of critical applications without having to verify the consistency between the SCADE model and the generated code, a 'qualified code generator' known as KCG, was developed, with the first version released in 1999. KCG has been used (and is still used) in software projects up to the most demanding safety levels complying with standards DO-178C, IEC 61508, EN 50128, IEC 60880 and ISO 26262 where a high confidence in automation is expected. This code generator demonstrated the interest of a semantically well defined language for the qualification process. It is very unique in the field of embedded software and contributed to the industrial success of SCADE.

The objective in designing SCADE 6 was to provide novel language features to widen the scope of applications developed with SCADE, but carefully selected to preserve the qualities that made SCADE accepted for safety critical development. One was the mix of models, from purely data-flow ones already well covered, to control-flow ones better covered by languages like ESTEREL and the SyncChart [9], and complex interactions between the two. An other limitation of SCADE was the absence of arrays. LUSTRE V4 provided powerful recursive arrays definitions very well suited for hardware but the static expansion they imposed was inadequate for software. Finally, there were also quests for other language extensions (such as modules), more expressive types (in particular around numerics), compiler optimizations.

To meet these objectives, we were guided by several works:

- ESTEREL and SyncChart for control-dominated system expressed by hierarchical state machines;
- functional arrays and iterators [10];
- LUCID SYNCHRONE [11], [12], [13] for the integration of data-flow and control-flow and type-based analyses.

[1] SCADE stands for: *Safety Critical Development Environment*

[2] http://www.ansys.com/products/embedded-software/ansys-scade-suite

Several other works were instrumental. For example, *mode automata* [14] gave a first answer for writing mixed models between a subset of LUSTRE and the hierarchical automata of ARGOS [15]. Yet, several questions remained, in particular the integration in a complete language. The language ESTEREL V7 [3] did integrate data-flow and control-flow but it was tuned for generating very efficient hardware. How to adapt it for software and integrate it into a qualified compiler was unknown at that time.

The main design decision was to built the language and compiler on the following idea: (1) define a minimal kernel language together with a static and dynamic semantics and used as some kind of 'typed assembly language' from which sequential code is produced; (2) express richer programming constructs in terms of the basic language by a source-to-source translation, and give a static and dynamic semantics for all language constructs that preserves this translation semantics. For the kernel language, we defined a *clocked data-flow language*, close to LUSTRE but with some modifications that we motivate.

This design decision was put into practice in RELUC [4], a prototype language and compiler written in OCAML that was used to experiment new programming constructs and compilation techniques. This prototype evolved continuously between 2000 and 2006. In 2006, SCADE 6 was launched from it, with a first release in 2008.

In this paper, we present the way this design decision has been followed. We illustrate the main language features, compile-time static analyses and the compiler architecture. The paper focuses on the language, information about graphical support and modeling tool were published in [17].

Section II reminds the LUSTRE kernel behind SCADE till version 5. Section III presents the new core language on which SCADE 6 is built. Section IV illustrates the static semantics of SCADE 6. Section V presents the mix of data-flow and control-flow. Section VI explains the treatment of arrays. Section VII discusses the code generator design and qualification. Section VIII gives a few concluding remarks.

In the paper, we use LUSTRE for the underlying language of SCADE until version 5 and SCADE 6 for the new versions.

II. FROM LUSTRE CORE TO SCADE 6 CORE

LUSTRE is a synchronous interpretation of the block diagrams used for decades by control engineers. In this interpretation, time is discrete and can be identified by an integer. Hence, a discrete-time signal is a *sequence* or *stream* of values and a system is a stream function.

A. The core LUSTRE language

Since sequences are the basic elements of LUSTRE, operations are lifted to apply pointwise. This is what is done in maths when writing the point-wise sum of two sequences:

$$(x_n)_{n\in\mathbb{N}} + (y_n)_{n\in\mathbb{N}} = (x_n + y_n)_{n\in\mathbb{N}}$$

Constants and literals are also lifted to streams by infinitely repeating them. The evolution can be represented by the table:

`2`	2	2	...	2	...
`x`	x_0	x_1	...	x_n	...
`y`	y_0	y_1	...	y_n	...
`x + y`	$x_0 + y_0$	$x_1 + y_1$	...	$x_n + y_n$	...
`2 * x`	$2 * x_0$	$2 * x_1$	...	$2 * x_n$	...

An important primitive is the unit delay **pre** (for 'previous'):

`x`	x_0	x_1	...	x_n	...
pre `x`	*nil*	x_0	...	x_{n-1}	...

If `x`=$(x_n)_{n\in\mathbb{N}}$, **pre** `x` is 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 first value of a stream can be specified with the initialization operator (`->`):

`x`	x_0	x_1	...	x_n	...
`y`	y_0	y_1	...	y_n	...
`x -> y`	x_0	y_1	...	y_n	...

More formally: $\begin{cases} (\texttt{x -> y})_0 = x_0 \\ \forall n \in \mathbb{N}, (\texttt{x -> y})_{n+1} = y_{n+1} \end{cases}$

Its combination with **pre** defines the initialized delay:

`x`	x_0	x_1	...	x_n	...
pre `y`	*nil*	y_0	...	y_{n-1}	...
`x ->` **pre** `y`	x_0	y_0	...	y_{n-1}	...

The following LUSTRE equation illustrates them:

```
nat = 0 -> 1 + pre nat;
```

which means that, forall $n \in \mathbb{N}$:

$$\begin{aligned} \texttt{nat}_n &= (\texttt{0 -> 1 +}\ \textbf{pre}\ \texttt{nat})_n \\ &= 0 \text{ if } n = 0 \\ &= (\texttt{1 +}\ \textbf{pre}\ \texttt{nat})_n \\ &= 1 + nat_{n-1} \text{ otherwise} \end{aligned}$$

The last important notion is that of a clock. The clock of a stream tells when its current value is present (or ready). Clocks are modified by two operators, **when** and **current**. A stream can be filtered according to a boolean condition:

`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	_	...

The _ is not a special value but indicates the absence of a value. Thus, the stream `x` **when** `h` is the sub-sequence $x_0, x_2, x_3, \ldots$. We say that its clock is `h`, that is, it is present when `h` is present and true. By filtering a stream, it is possible to model a slow process. E.g., if `f` is a stream function such that its input and output are on the same clock, then `f(x` **when** `h)` has clock `h` and contains the application of `f` to the sub-sequence of `x` filtered by `h`. Note that `h` can be any boolean expression and thus, it can encode a periodic clock.

[3]This was the latest version of ESTEREL, developed at ESTEREL-TECHNOLOGIES EDA. It is unfortunately no more publicly available — even its reference manual — after the company stopped in 2008.

[4]The first publication mentioning RELUC is [16]

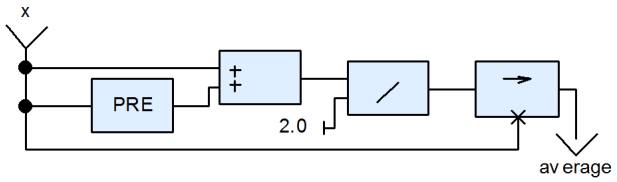

Fig. 1. The sliding average diagram

A stream can be completed by keeping its value between two samples. This corresponds to a zero-order hold.

	h	**true**	**false**	**false**	**true**	**false**	...
	a	a_0	_	_	a_1	_	...
current	a	a_0	a_0	a_0	a_1	a_1	...

If a has some clock h, the clock of **current** a is the clock of the clock of a. Hence, a cannot be on the fastest clock (termed the 'base' clock) of the system. **current** is the way to go from a slow process to a fast one. E.g., **current**(f(x **when** h)) returns a stream whose clock is that of x and h.

A program can be *synchronously executed* when the execution can proceed as a global sequence of steps where streams expected to be present are indeed present and those expected absent are indeed absent. In particular, a combinatorial function that expects its two arguments to be present or absent, e.g., the operation $+$, have its two arguments present or absent. All computations of the corresponding Kahn Process Network are clocked according to a global time scale, removing all necessary buffer synchronisations [18].

A dedicated static analysis, named the *clock calculus*, statically rejects a program that actually uses a stream at a clock different from what is expected. E.g, writing x + (x **when** h) is rejected because the sum operator expects its two arguments to be on the same clock.

In LUSTRE, a user defined operator (or stream function) is introduced by the keyword **node**. Below is the example of a smoothing function that computes the average of its input x with its previours value **pre** x. Figure 1 shows the corresponding block diagram.

```
node sliding_average (x : real) returns (average : real);
let
  average = x -> (x + pre x) / 2.0;
tel
```

The body is an unordered set of equations which allows introduce/remove auxiliary equations. E.g., the following node computes the very same sequence with a local variable s:

```
node sliding_average (x : real) returns (average : real);
val s: real;
let
  average = x -> s / 2.0;
  s = x + pre x;
tel;
```

An equation of the form $x = e$, where x is a variable and e an expression holds at every instant, that is, $\forall n \in \mathbb{N}$, $x_n = e_n$.

B. The question of determinism

The semantics of LUSTRE formally defines what is the current value of a stream. The compiler checks that this value exists, is unique, can be computed sequentially from current inputs, possibly past computed values and in bounded time and space. Parallelism is not the only source of non determinism. Operators may introduce non determinism too. An example is the operator **pre** whose initial value nil is undetermined. It is thus important that an observed output does not depend on it. **current** also introduces nil. E.g.:

	h	**false**	**false**	**false**	**true**	**false**	...
	a	_	_	_	a_0	_	...
current	a	nil	nil	nil	a_0	a_0	...

h is the clock of a. The prefix of nil values is arbitrarily long, unless h is initially **true**. **current** (**pre** x) is another example of a stream defined only after the second value of x.

The decision problem — is a given output depend on the actual value of nil? — is undecidable in the general case and at least combinatorial. It can be safely approximated by a SAT problem. Yet the time complexity and good diagnostics are difficult to give. Moreover, its conclusion — the system is safe — would have to be justified in the context of a qualified compiler. For SCADE 6, we took a more modest approach, designing a dedicated initialization analysis which deals with the particular case of the un-initialized delay and refuse to compile a program where nil may happen anywhere but in a first position of a sequence.

III. SCADE 6: A NEW DATA-FLOW CORE

Instead of **current** we chose an alternative operator **merge** borrowed from LUCID SYNCHRONE and which merges two complementary streams.

	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	...
hold(i, h, a)		a_0	a_0	a_1	a_2	a_2	...

A zero-holder hold(i, h, a) which holds the value of a itself on clock h is programmed: [5]

```
node hold (i: 't; clock h : bool ; a: 't when h)
  returns (o : 't)
  o = merge(h; a; (i -> pre o) when not h);
```

Contrary to **current**, **merge** does not introduce a nil. Moreover, its implementation does not use any memory but only local variables and is easier to compile efficiently. Finally, there are common situations in LUSTRE of an equation of the form: o = **if** h **then current** a **else current** b; with a on clock h and b on clock **not** h that is difficult to compile efficiently. It uses two memories (one for each **current**) that are difficult to remove and three conditionals on h (one for every **current** plus the one of the condition) which need to be fused. This equation is equivalent to: o = **merge**(h; a; b);

Finally, the **merge** is generalized to an n-ary form for merging several complementary sequences [19].

We now see the second change made on the data-flow core. LUSTRE does not provide a mean to modularly reset a system on a Boolean condition, that is, to re-initialize all its state

[5] The **let**/**tel** braces are optional when the body contains a single equation.

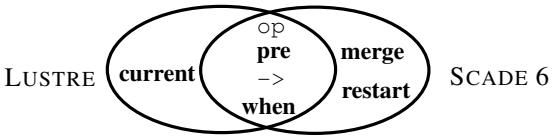

Fig. 2. Data-flow Cores

variables whereas it is a primary feature in ESTEREL. This forces the user to explicitly add re-initialization conditions all over the place in the code. E.g., replace `sum` by:

```
node resetable_sum (c:bool) returns (o:int);
  o = if c then 0 else (0 -> 1 + pre o);
```

The SCADE 6 core is extended with a built-in construct for resetting any node instance. For example, the expression (**restart** `sum` **every** `c`)`()` re-initializes the node instance `sum` when `c` is true. The reset primitive was first introduced in [20].

Figure 2 summarizes the differences between the two cores. This data-flow core is described in more detail in [21] and constitutes the basic language of SCADE 6.

IV. STATIC SEMANTICS

The dynamic semantics of SCADE 6 is that of LUSTRE with extensions to take the **merge** and modular reset into account [22]. The static semantics gathers all the invariants that a program must satisfy before considering its execution. For SCADE 6 we express them as typing problems so that, quoting Robin Milner, "well-typed programs cannot go wrong" [23]. This approach enjoys two properties:

- A type system is modular in the sense that a function type gathers all the information needed to check the correct use of this function.
- It allows for giving good error diagnostics, as far as the type language is simple enough.

Four dedicated type systems exist in the SCADE 6 compiler that are summarized below. They are applied in sequence: when one fail, the compilation stops. The type systems are presented following the order they are applied in KCG.

A. Types

The first (and pretty standard) static verification step is the *type checking*. Its main features are:

- All types must be declared; a type can be an enumerated set of values, a record, an array parameterized by a size, or an abstract type.
- type equivalence is based on structural equality;
- the language provides a number of built-in type classes, like **numeric** and **integer**. E.g., **int8**, **int16**, **int32**, etc. are elements of the class **integer**.
- types can be polymorphic and possibly constrained by the type class **numeric**, **float**, **integer**, **signed**, **unsigned**.
- functions may be parameterized by a size. Such a parameter can be used in an array type.

Figure 3 illustrates several of these features. It defines a few operators working on matrices and vector whose sizes are given as parameters and whose coefficients are of a numeric type. The function `root` makes use of the generic matrix product for particular type and sizes.

The type system is formalized in the KCG project documentation. It is a simplified form of the type classes used in Haskell [24]. In particular, type classes are built-it and cannot be defined by the user. Moreover, the language is first-order (it is not possible to write a function which takes or returns a function). A type expression may also contain a size expression (e.g., **float64**`^3^7` defines the type of matrices of size 7×3 of doubles). A size must be a compile-time static expression. To avoid having to incorporate a decision procedure — is size expression $x + y$ equal to $y + x$? — type checking is performed in two steps: the first step does regular type checking but generates a set of equality constraints between size expressions. In a second step, one static expansion has been performed, it checks that equality constraints are trivial.

Finally, Types must incorporate an information that is specific to SCADE 6. Combinatorial functions (whose current outputs only depend on current inputs) are given the kind **function** whereas a statefull function (whose outputs may also depend on the past) are given the kind **node**. Kinds are checked during typing simply: if a function is declared with kind k, all the function it calls must also be of kind k.

The compiler imposes a strong typing discipline, that is, programs which do not type check are rejected. This allows stating the following property.

Property 1 (Well typed program execution): A well typed program is such that:

- arguments of functions have the expected type;
- array accesses are within array bounds.

B. Clock checking

The purpose of this analysis is to ensure that programs can be executed synchronously. Once done, every expression is clocked with respect to the global time scale, named the *base clock*. Precisely, the clock of a stream is an expression of the following language: $ck \quad ::= \quad ck \,\mathtt{on}\, e \mid \alpha$
where e is a Boolean expression of the core language and α a clock variable. For example, an expression with clock $(\alpha \,\mathtt{on}\, e_1) \,\mathtt{on}\, e_2$ is present if an only if e_2 is present with clock e_1 and true. An expression with clock variable α is present iff α evaluates to true. The clock checking existed since the early days of LUSTRE [25]. In [18], it was shown to be a typing problem, precisely a typing problem with dependent types [26]. SCADE 6 adopts this point-of-view but takes a simpler formulation where equivalence between Boolean expressions is replaced by name equivalence [27]. It adds an extra simplification to this proposal by imposing that the clock of variables is declared and a function to use a single clock variable (α) whereas the original proposal, implemented in LUCID SYNCHRONE V3, did not impose those two restrictions.

Given a function definition, the compiler checks clocks and computes a clock signature. The signature for the function `hold` (Section III) is: $\forall\alpha.\alpha \times (h : \alpha) \times \alpha \,\mathtt{on}\, h \rightarrow \alpha$. It states

that, for any clock α, the first input of `hold` must have clock α, the second, named h, clock α, the third, clock α on h. Then, the output has clock α.

Property 2 (Synchronous execution): A well clocked SCADE 6 model can execute synchronously.

A corollary is that a SCADE 6 model can be implemented in bounded memory, provided that imported functions do.

C. Causality analysis

The purpose of this analysis is to ensure that a set of processes running synchronously produces one and at most one output at every reaction. LUSTRE follows a simple approach, reducing it to the analysis of instantaneous loops in the data-dependences relation between variables. A more expressive *constructive causality* was proposed for ESTEREL [28].

Following a preliminary work [29], the causality analysis for SCADE 6 has been specified as a type system. The intuition is to associate a time stamp to every variable and to check that the relation between those time stamps is a partial order (thus, with no cycle). We illustrate it on the following two integration functions:

```
node fwd_Euler <<K, T>> (IC : 't ; u : 't)
  returns (y : 't) where 't numeric
  y = IC -> pre (y + K * T * u);

node bwd_Euler <<K, T>> (IC : 't ; u : 't)
  returns (y : 't) where 't numeric
  y = IC -> pre y + K * T * u;
```

The causality type of `fwd_Euler` is $\forall\gamma_1,\gamma_2.\gamma_1 \times \gamma_2 \rightarrow \gamma_1$ which indicates that the output only depends instantaneously of its first input. From this signature, one can see that this operator is able to break a dependency cycle on its second input. `bwd_Euler` has type $\forall\gamma.\gamma \times \gamma \rightarrow \gamma$ that expresses the dependency of the output on both inputs. This is enough information to deduce that this integration function cannot be used to break a cycle.

Property 3 (Schedulability): A causal SCADE 6 model can compiled into a statically scheduled sequential code.

D. Initialization analysis

The purpose of this analysis is to ensure that the behaviour of a system does not depend on the unspecified value nil. A simple type-based analysis with sub-typing is described in [30]. For every expression, it computes its type with the following intuition:

- The type $\mathbf{1}$ is that of a stream which may have an uninitialized value nil at the very first instant;
- the type $\mathbf{0}$ is that of a stream which is always initialized.

It induces the natural sub-typing relation $\mathbf{0} \leq \mathbf{1}$, meaning that an expression which is always initialized can be given to an expression that is expected to have type $\mathbf{1}$. E.g., the uninitialized rising edge operator:

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

gets the initialization type signature: $\mathbf{0} \rightarrow \mathbf{1}$ and the following function:

```
node min_max(x, y : int32) returns (mi, ma : int32)
  mi, ma = if x <= y then (x, y) else (y, x);
```

gets the signature: $\forall\delta.\delta \times \delta \rightarrow \delta \times \delta$.

The initialization analysis does not force all functions to return well initialized streams. Hence, the following function (with signature $\mathbf{0} \times \mathbf{0} \rightarrow \mathbf{1}$) which is accepted as a node declaration is not accepted if this node is the main node.

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

whereas the following (with signature $\mathbf{0}\times\mathbf{0} \rightarrow \mathbf{0}$) is accepted.

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

The main node defined what is finally executed on the target platform. Its outputs, in particular, must always be of type $\mathbf{0}$.

Property 4 (Determinism): A well initialized SCADE 6 model is deterministic in the sense that it never produces an output that depends on an undefined value (nil).

This analysis is defined for a synchronous data-flow language in [30]. It is applied to the full SCADE 6 language.

V. CONTROL-STRUCTURES

In LUSTRE, clocks are the only way for controlling the execution of a computation: an expression is computed only when its clock is true. Unfortunately, their use in LUSTRE is not easy, partly because of a lack of expressiveness of the clock language and automation (particularly clock polymorphism and inference).

Clocks exists in SCADE 6 but the language proposes an alternative by mean of dedicated control-structures. These are essentially syntactic sugar in the sense that they are translated into well-clocked equations of the data-flow core. This approach appeared extremely useful to ensure that all language extensions were consistent with each other. We were also convinced that the data-flow core was expressive enough to support this translation. The PhD work of Hamon [13] was pioneering in this direction.

In [14], Maraninchi and Rémond introduce the language of *mode-automata* that mixes a subset of LUSTRE with ARGOS-like hierarchical automata. A compilation into guarded equations was proposed but with a source language which was less expressive than SCADE 6 and not done via a source-to-source transformation.

The following sections present and illustrate the new constructs. Their formalization and compilation are given in [19].

A. Activation blocks

The activation block is the simplest way of expressing that some equations are only active according to a boolean condition. The example below is a function that computes the complex solution of a second degree polynomial. This is a typical example where case analysis is needed. Depending on the sign of the discriminant, one of three solutions is selected.

```
function imported sqrt (x:float64) returns (y:float64);

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
```

The square root function is declared as imported (it is not a built-in primitive of SCADE 6). A local variable `d` is introduced to name the result. `d` only exists when `delta > 0`, as if `d` was 'clocked' by writing an equation `d = sqrt (delta` **when** `(delta > 0))`. Indeed, the translation of the function `second_degree` precisely does that: it introduces such a clocked equation for every defined variable.

B. Scope and shared variables

The previous example illustrates the situation where a shared variable is defined by different equations and a single one is active at a time. Only the active equations are executed. In particular, an expression **pre**(e) activated when a condition c is true denotes the previous 'observed' value of e, that is, the value that e had, the last time c was true. This is illustrated on the function `move1` with an execution trace given below:

```
node move1 (c : bool) returns (o : int32)
  activate if c then o = (0 -> pre o) + 1;
               else o = (0 -> pre o) - 1; returns o;

node move2 (c : bool) returns (o : int32 last = 0)
  activate if c then o = last 'o + 1;
               else o = last 'o - 1; returns o;
```

c	true	true	false	false	true	false	...
move1(c)	1	2	−1	−2	3	−3	...
move2(c)	1	2	−1	0	1	0	...

But how to communicate between two exclusive branches, e.g., to define a signal that is incremented and decremented? One solution is to add the equation `last_o = 0 ->` **pre** `o` in parallel and use `last_o` in the two branches. SCADE 6 provides a simpler and more intuitive way for communicating the value of a shared variable. It is illustrated in the function `move2` with the corresponding chronogram. The variable `o` is initialized with `0`. The construct **last** in **last** `'o` applies to a name, not an expression. **last** `'o` denotes the previous 'computed' value of `o`. This construct is not primitive in SCADE 6 in the sense that it is translated into the basic data-flow core. It is a convenient construct to express in a data-flow manner, equations of the form `x =` **last** `x + 1` which, by the way, gives an imperative flavor.

In the proposal for *mode automata* [14], the operator **pre** applied on a shared variable `x` behaves like **last** `'x`, which does not correspond to the **pre** of LUSTRE.

C. Hierarchical Automata

State machines are a convenient way to specify sequential behaviour with the two classical forms:

- Moore machines, when the current output is a function of the current state only;
- Mealy machines, when the current output is a function of both the current state and current input.

In [31], Harel introduced *Statecharts*, an extension of state machines to express complex systems in a modular and hierarchical way. ARGOS [15], SyncChart [9] and ESTEREL integrate this expressiveness within a synchronous framework with static conditions to ensure the existence and uniqueness of a reaction in every state. SyncChart [9] was the graphical notation used in the industrial tool-set based on ESTEREL.

SCADE 6 incorporates hierarchy a la SyncChart but where states may themselves contain other state machines and/or data-flow equations. A difference with the approach of ESTEREL/SyncChart is the existence of a textual support for automata. In general, a graphical representation of state machines is preferred, but proposing a textual support maintains the language and the graphical notation in a simple one to one correspondence and all the transformation work is concentrated at the compiler level. The main features of SCADE 6 hierarchical state machines are borrowed from the SyncChart:

- An automaton must have one *initial* state;
- some states can be marked to be *final*;
- a transition can be *weak* or *strong*;
- a transition may either reset or resume its target state;
- a *synchronization* mechanism allows for firing a transition when all the automata inside the state are in a final state.

1) Intuitive Semantics: The semantics has been formalized in [32] and through a translation into the data-flow core [19]. SCADE 6 imposes an extra constraint: *at most one transition is fired per cycle*.

A cycle consists in deciding, from the current *selected state* what is the *active state*; execute the corresponding set of equations; then determine what is the *selected* state for the next cycle. Precisely:

- At first cycle, the *selected state* is the state marked initial.
- Evaluate all the guards of the *selected state* strong transitions. The *active state* is the target of the first (taken sequentially) firable strong transition if any, otherwise it is the *selected state*.
- Execute the equations of the *active state*.
- Evaluate all the guards of the *active state* weak transitions. The next *selected state* is the target of the first (taken sequentially) firable weak transition if any, otherwise it is the current *active state*.

2) Two simple examples: The example below shows a node that returns an integer output `o` with last value initialized to `0`. It is defined by a two states automaton. `Up` is the initial state.

In this mode, o is incremented by 2 until o >= 12. Then, the next state is Down. In this state, o is decremented until it reaches value 0 and the next state is Up, etc.

```
node up_down() returns (o : int32 last = 0)
  automaton
    initial state Up
      o = last 'o + 2;
      until if o >= 12 resume Down;

    state Down
      o = last 'o - 1;
      until if o = 0 resume Up;
  returns o;
```

Because the transitions are weak, the guards can involve the current value of o. Hence, replacing the weak transition (**until**) by a strong transition (**unless**) would lead to a causality error.

The second example is a two inputs node: tic and toc.

```
node tictoctic (tic, toc : bool) returns (o : int32 last = 0)
  automaton
    initial state WaitTic
    unless if tic restart CountTocs;

    state CountTocs
    unless if tic resume WaitTic;
      o = 0 -> if toc then (last 'o + 1) else last 'o;
  returns o;
```

The initial state WaitTic waits for an occurrence of tic then immediately goes to the state CountTocs. This state is entered by **restart** which reinitializes all of its state variables (in particular the initialization ->) and thus o. Because WaitTic does not provide a definition for o, its last value must be declared. The value of o stay unchanged in the initial state.

3) A complete example: The last example is a simple version of the digital watch written in ESTEREL [33] limited to watch and stopwatch mode. It has four input buttons:

- stst : start/stop button
- rst : reset button
- set : set time button
- md : mode selection button

and it displays the following information:

- HH . MM . SS : time information
- L : lap time indicator
- S : setting time mode active indicator
- Sh : setting hour mode (minutes otherwise)

Basically, three automata run in parallel. Two are simple counters, one for the time (automaton Stopwatch) and the other for the stop watch (automaton Watch). There is also a process that manages the display and the Lap time (automaton Display). The watch has too modes, one where it counts time; the other where the current time is set. This program is supposed to be executed periodically with a base clock of 10ms. When a variable is declared, it can be given a last value and/or a default value (e.g., **var** isStart : **bool default** = **false**). The default value is the definition of the variable in the subscopes that omit its definition. If no default value is specified, the implicit definition for a variable x is x = **last** 'x;. The declared last value (e.g. d : **int8 last** = 0;) defines the initial value of its last (here **last** 'd for instance). These two features allow for writting shorter programs. The implicit equation x = **last** 'x; participates to the imperative flavor of these constructs.

```
node watch (stst, rst, set, md : bool)
  returns (HH, MM, SS : int8;
           L, S, Sh : bool default = false last = false)
var
  isStart : bool default = false; -- is chrono started?
  is_w : bool default = false;    -- is in clock mode?
  m, s, d : int8 last = 0;        -- chrono timers
  wh, wm, w, ws : int8;           -- clock timers
let
  w = 0 -> (pre w + 1) mod 100;
  ws = 0 -> (if w < pre w
             then pre ws + 1 else pre ws) mod 60;

  automaton Stopwatch
    initial state Stop
    unless if stst and not is_w resume Start;
           if rst and not (false -> pre L)
              and not is_w restart Stop;
    m, s, d = (0, 0, 0) -> (last 'm, last 's, last 'd);

    state Start
    unless if stst and not  is_w resume Stop;
    let
      d = (last 'd + 1) mod 100;
      s = (if d < last 'd
          then last 's + 1 else last 's) mod 60;
      m = if s < last 's then last 'm + 1 else last 'm;
      isStart = true;
    tel
  returns m, s, d, isStart;

  automaton Watch
    initial state Count
    let
      wm = 0 -> (if ws < last 'ws
                 then last 'wm + 1 else last 'wm) mod 60;
      wh = 0 -> (if wm < last 'wm
                 then last 'wh + 1 else last 'wh) mod 24;
    tel
    until if set and  is_w restart Set;

    state Set
    let
      S = true;
      automaton SetWatch
      initial state SetHours
      let
        Sh = true;
        wh = (if stst then last 'wh + 1
              else if rst then last 'wh + 23
                  else last 'wh) mod 24;
      tel
      until if set and is_w restart SetMinutes;

      state SetMinutes
      wm = (if stst then last 'wm + 1
            else if rst then last 'wm + 59
                else last 'wm) mod 60;
      until if set and is_w restart SetEnd;

      final state SetEnd
      returns Sh, wh, wm;
    tel
    until synchro resume Count;
  returns S, Sh, wh, wm;

  automaton Display
    initial state DisplayWatch
    unless if md and not S resume DisplayStopwatch;
    HH, MM, SS, is_w = (wh, wm, ws, true);

    state DisplayStopwatch
    unless if md and not S resume DisplayWatch;
    var lm,ls,ld : int8 last = 0; -- chrono display
    let
      HH, MM, SS = (lm, ls, ld);

      automaton LapManagement
        initial state Stopwatch
        lm, ls, ld = (m, s, d);
```

```
        until if rst and  isStart restart Lap;

        state Lap
        L = true;
        until if rst restart Stopwatch;
      returns lm, ls, ld, L;
    tel
  returns HH, MM, SS, is_w, L;
tel
```

VI. Extension with arrays

The arrays of SCADE 6 are functional and come wa collection of iterators (e.g., **map**, **fold**, functional update) as they exist in functional languages. Those operators are free of side effects and so, preserve the functional style of SCADE 6. They do not replace external functions applied to arrays but complement them. In particular, it is possible to map (or fold) a statefull function point-wise to all elements of an array. Array iterators enjoys several optimizations like the elimination of intermediate copies. The set of array operators and their compilation has been first established by Morel [10].

As a first example, consider the function `exists` which, given a static parameter `n` and a boolean array `b` of length `n`, returns true if and only if one element is true.

```
function exists <<n>>(b : boo^n) returns (o : bool)
  o = (fold $or$ <<n>>) (false, b);
```

or is the operation **or** used in prefix notation. The function `exists` is combinatorial; hence, it can be declared with the keyword **function**. The semantics is that of the full unfolding; yet, the compiler generates a for loop.

A. A combinatorial example

Figure 3 gives a more complete example, with the scalar product of two vectors, the vector matrix product and the matrix product. All these functions are polymorphic and apply to any numerical type with vectors and arrays whose sizes are specified as a static input. Function `root` shows an instance of the matrix product for specific sizes and with type **float64**. The function `MatVectProd` uses a special primitive **transpose** that allows to permute two dimensions of an array of arrays.

B. A statefull example

The following example is inspired by the interface present in a fighter plane, where, because of acceleration, the pilot may not be precise in selecting the right push button. To overcome this risk, command selection is done in two steps: a pre-selection that works as radio buttons (selecting one button un-selects the other) and a second step done with a single button (no choice, thus no possible selection error) to confirm the pre-selection. The logic to manage this interface is quite regular and independent on the number of buttons. We give an implementation in SCADE 6 where a state machine specifies the behaviour of one button and a parameterized number `n` that are composed in parallel.

Buttons have a background and a foreground color depending on their state. When a button is pre-selected, its background is yellow. When locked, the background of the pre-selected buttons becomes green. The node `Button` defines all these behaviours; its inputs are the position of the considered button, the lock command, the unlock command and a Boolean indicating if another button is pushed to implement the *radio button* behaviour.

```
function prod_sum (acc_in, ui, vi: 'T)
  returns (acc_out:'T) where 'T numeric
  acc_out = acc_in + ui * vi;

-- scalar product of two vectors: u . v
function ScalProd <<n>> (u, v: 'T^n)
  returns (w:'T) where 'T numeric
  w = (fold prod_sum <<n>>) (0, u, v);

-- product of a matrix by a vector: A(m,n) * u(n)
function MatVectProd <<m, n>> (A: 'T^m^n; u: 'T^n)
  returns (w: 'T^m) where 'T numeric
  w = (map (ScalProd <<n>>) <<m>>)(transpose (A; 1; 2), u^m);

-- matrix product: A(m,n) * B(n,p)
function MatProd <<m, n, p>> (A: 'T^m^n; B: 'T^n^p)
  returns (C:'T^m^p) where 'T numeric
  C = (map (MatVectProd <<m, n>>) <<p>>)(A^p, B);

function root (A: float64^3^7; B: float64^7^5)
  returns (C: float64^3^5)
  C = MatProd <<3, 7, 5>> (A, B);
```

Fig. 3. Example: matrix operations

```
type bk_color = enum {grey, yellow, green};
type fr_color = enum {black, white};

node Button (button, lock, unlock, other : bool)
  returns (background : bk_color; foreground : fr_color)
let
  automaton
    initial state Unselected
    unless if lock restart LockedUnselection ;
           if button restart Preselected;
    background, foreground = (grey, white);

    state Preselected
    unless if lock restart LockedSelection;
           if button or other restart Unselected;
    background, foreground = (yellow, white);

    state LockedSelection
    unless if unlock restart Preselected;
    background, foreground = (green, white);

    state LockedUnselection
    unless if unlock restart Unselected;
    background, foreground = (grey, black);
  returns background, foreground;
tel

const n : int16 = 8; -- number of buttons

node TwoStepsSelect(Lock : bool; buttons : bool^n)
  returns (bk_buttons : bk_color^n;
           fg_buttons : fr_color^n;
           LockLight : bool)
sig lockSig, unlockSig;
var buttonPressed : bool;
let
  automaton LockManagement
    initial state LockLow
      unless if Lock do {emit 'lockSig} restart LockHigh;
      LockLight = false;

    state LockHigh
      unless if Lock do {emit 'unlockSig} restart LockLow;
      LockLight = true;
  returns LockLight;

  bk_buttons, fg_buttons = -- instantiate buttons
    (map Button <<n>>)
      (buttons, 'lockSig^n, 'unlockSig^n, buttonPressed^n);
```

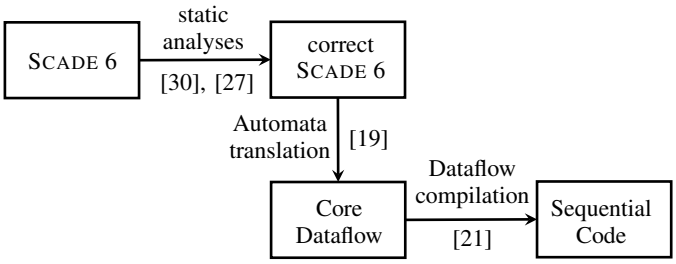

Fig. 4. SCADE 6 Compiler Organization.

```
  buttonPressed =          -- exists one pressed button?
    (fold or <<n>>) (false, buttons);
tel
```

The interest of this example lies in the iteration of an operator that encapsulates a state (the state of the corresponding button, through the state machine). The semantics is that of the unfolded version; yet, compiled as a for loop.

VII. CODE GENERATION

A. Compiler Organization

The organization of the compiler (KCG) is rather classical. Static analysis are applied in sequence right after parsing. If they all succeed, code generation starts with a sequence of source-to-source transformations that rewrite all the constructs into the data-flow core. This core is extended with array iterators. Then, the data-flow core is translated into an intermediate sequential language. At last, target imperative code (mainly C and ADA) is emitted. Figure 4 summaries these steps at a coarse grain; corresponding bibliographic references are given on the arrows.

Among the transformation, many optimisations are done on the data-flow form (dead-code elimination, constant propagation, common sub-expression, iterator compositions, etc). The scheduling in the data-flow compilation implements heuristics to limit memory size. Control structures are merged in the sequential representation.

B. Qualified Development

Qualification is based on traceability between a specification and the implementation. The specification details the principles presented in this paper. The source and intermediate languages have been formally specified together with the static semantics (defined by inference rules) and source-to-source transformations (defined by rewrite rules). Those specifications are used by the development team to implement the compiler and by an independent verification team to test it.

For the implementation, we choose OCAML [34] which was quite a challenge for a qualified tool, in 2005. Indeed, certification standards often push companies to use well established technologies. We thus had to convince that OCAML was well adapted to write a compiler. The argumentation was built on the small distance between the formal specification and their implementation in OCAML. This industrial use of OCAML in a certified context is detailed in [35] and [36].

The current version of SCADE KCG is approximately fifty thousands lines of code (50 KLOC) and it uses a simplified OCAML runtime to satisfy the objectives of the standards. The formalized static semantics for the whole input language is about one hundred pages long and has been updated for more than ten years to integrate new language features. The detailed design is more than one thousand pages long.

C. Towards a Computer Aided Formal Qualification

The formalization made for SCADE 6 was an important step to get a qualified code generator. Yet, this formalization was done by hand and some important parts were not considered. The draft [22] was a first proof of correctness for the data-flow core down to sequential code. Extending it for the full language and with high confidence in the proof correctness without the help of a computer appeared out of reach.

Proof assistants like COQ [37] allow for writing both programs, properties and computer checked proofs. The COMPCERT C compiler [38], [39] is the first compiler that is developed this way. Its industrial application and qualification is now considered seriously but making a formal process match industrial certification standards is a new challenge that does not reduce to a scientific question.

The next step for SCADE 6 and KCG is now to go further by using computer aided tools to get a proof of correctness of the compiler. Then connecting this new object with the COMPCERT C compiler would lead to a mathematically proven translation from a high-level synchronous language to assembly code. A first step have been achieved recently for the data-flow core without the reset [40]. The prototype compiler is called VELUS. When compilation succeeds, the generated assembly is proved to be semantically equivalent to the data-flow program.

VIII. CONCLUSION

This paper has shown principal language features of SCADE 6 together with the main design choices for its compilation. It relates a long and fruitful collaboration between industry and academia and is a concrete example of transfer of state-of-the-art research work on computer language design and implementation.

If the core of the language remained on the same size as LUSTRE, the new rich features it proposes are a big improvement for the SCADE designers. The proposed mix of fine grain data-flow and hierarchical automata mix is quite unique. The language is now as convenient to develop the logic of a cockpit display than it is to develop a discrete control law.

SCADE and KCG have been used in about hundred DO-178B/C level A avionic systems all over the world, a quite significant result in this market.

And the story continues. The language offers a good coverage of discrete-time system programming, adding continuous time modeling capabilities could be an axis of development for the near future. Following the same collaboration framework, the work on ZÉLUS [41], [42] is opening the way.

ACKNOWLEDGMENT

This works owes a lot to Paul Caspi; and Gérard Berry for our living discussions. We also want to thank all the colleagues of the Core team for their hard work on SCADE 6 KCG and our CTO, Bernard Dion, for his confidence and support.

REFERENCES

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, Jan. 2003.

[2] A. Benveniste, P. LeGuernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, pp. 103–149, 1991.

[3] G. Berry and G. Gonthier, "The Esterel synchronous programming language, design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[5] G. Berry, "Real time programming: Special purpose or general purpose languages," *Information Processing*, vol. 89, pp. 11–17, 1989.

[6] Berry, G., "Formally unifying modeling and design for embedded systems - a personal view," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016*, T. Margaria and B. Steffen, Eds. Corfu, Greece: Springer International Publishing, October 10-14 2016, pp. 134–149, proceedings, Part II.

[7] N. Halbwachs, "A synchronous language at work: the story of lustre," in *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, July 2005.

[8] N. Halbwachs and P. Raymond, "A tutorial of Lustre," 2002, http://www-verimag.imag.fr/Publications-Synchrones.html.

[9] C. André, "Representation and Analysis of Reactive Behaviors: A Synchronous Approach," in *CESA*. Lille: IEEE-SMC, july 1996.

[10] L. Morel, "Array iterators in lustre: From a language extension to its exploitation in validation," *EURASIP Journal on Embedded Systems*, 2007.

[11] P. Caspi, G. Hamon, and M. Pouzet, *Real-Time Systems: Models and verification — Theory and tools*. ISTE, 2007, ch. Synchronous Functional Programming with Lucid Synchrone, english translation of [43].

[12] M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI, April 2006.

[13] G. Hamon, "Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML," Ph.D. dissertation, Université Pierre et Marie Curie, Paris, France, 14 novembre 2002.

[14] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Science of Computer Programming*, no. 46, pp. 219–254, 2003.

[15] Florence Maraninchi and Yann Rémond, "Argos: an automaton-based synchronous language," *Computer Languages*, no. 27, pp. 61–92, 2001.

[16] J.-L. Colaço and M. Pouzet, "Type-based Initialization Analysis of a Synchronous Data-flow Language," in *Synchronous Languages, Applications, and Programming*, vol. 65. Electronic Notes in Theoretical Computer Science, 2002.

[17] F. X. Dormoy, "SCADE6 A model Based Solution For Safety Critical Software Development," in *Embedded Real Time Software and Systems (ERTS)*, Toulouse, January 2008.

[18] P. Caspi and M. Pouzet, "Synchronous Kahn Networks," in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[19] J.-L. Colaço, B. Pagano, and M. Pouzet, "A Conservative Extension of Synchronous Data-flow with State Machines," in *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[20] G. Hamon and M. Pouzet, "Modular Resetting of Synchronous Data-flow Programs," in *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[21] D. Biernacki, J.-L. Colaco, 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.

[22] C. Auger, J.-L. Colaco, G. Hamon, and M. Pouzet, "A formalization and proof of a modular Lustre compiler," 2010, accompaning paper of LCTES'08.

[23] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Science*, vol. 17, pp. 348–375, 1978.

[24] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad-hoc," in *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. ACM, Jan. 1989, pp. 60–76.

[25] 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*. ACM, 1987.

[26] S. Boulmé and G. Hamon, "Certifying Synchrony for Free," in *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, vol. 2250. La Havana, Cuba: Lecture Notes in Artificial Intelligence, Springer-Verlag, December 2001.

[27] J.-L. Colaço and M. Pouzet, "Clocks as First Class Abstract Types," in *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[28] G. Berry, "The constructive semantics of pure esterel," 2002, draft book. Available at: http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf.

[29] P. Cuoq and M. Pouzet, "Modular Causality in a Synchronous Stream Language," in *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

[30] J.-L. Colaço and M. Pouzet, "Type-based Initialization Analysis of a Synchronous Data-flow Language," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 3, pp. 245–255, August 2004.

[31] D. Harel, "StateCharts: a Visual Approach to Complex Systems," *Science of Computer Programming*, vol. 8-3, pp. 231–275, 1987.

[32] J.-L. Colaço, G. Hamon, and M. Pouzet, "Mixing Signals and Modes in Synchronous Data-flow Systems," in *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[33] G. Berry, "Programming a digital watch in esterel v3," INRIA, Tech. Rep. 1032, 1989.

[34] X. Leroy, "The Objective Caml system release 4.03. Documentation and user's manual," INRIA, Tech. Rep., 2017.

[35] 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)*, ser. Lecture Notes in Computer Science. Springer-Verlag, January 2008.

[36] 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 (ICFP)*. ACM, September 2009.

[37] "The coq proof assistant," 2017, http://coq.inria.fr.

[38] S. Blazy, Z. Dargaye, and X. Leroy, "Formal verification of a C compiler front-end," in *FM 2006: Int. Symp. on Formal Methods*, ser. Lecture Notes in Computer Science, vol. 4085. Springer-Verlag, 2006, pp. 460–475.

[39] X. Leroy, "A formally verified compiler back-end," *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[40] 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)*. Barcelona, Spain: ACM, June 19-21 2017.

[41] T. Bourke and M. Pouzet, "Zélus, a Synchronous Language with ODEs," in *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*. Philadelphia, USA: ACM, April 8–11 2013.

[42] T. Bourke, J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet, "A Synchronous-based Code Generator For Explicit Hybrid Systems Languages," in *International Conference on Compiler Construction (CC)*, ser. LNCS, London, UK, April 11-18 2015.

[43] P. Caspi, G. Hamon, and M. Pouzet, *Systèmes Temps-réel : Techniques de Description et de Vérification – Théorie et Outils*. Hermes, 2006, vol. 1, ch. Lucid Synchrone, un langage de programmation des systèmes réactifs, pp. 217–260.