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 has been used for more than twenty years in various application domains like avionics, nuclear plants, transportation, and automotive. SCADE was 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 to the highest standards for safety-critical applications.**

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

While previous publications have described some of its language constructs and compiler algorithms, no reference publication on 'Scade 6' exists to date. In this paper, we recall the decisions made in its design, illustrate the main language features and static analyses, and describe the compiler organization developed to satisfy the qualification process.

I. Introduction

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

The three languages were all founded on the *synchronous approach* [5] where a system is modeled ideally, with communications and computations supposed to be instantaneous, with formal checks of important safety properties like determinism and deadlock freedom, with the ability to generate an implementation that runs in bounded time and space, and with *a posteriori* verification that this implementation, whether in software or hardware, runs quickly enough.

These foundations immediately excited interest in industries having to deal with safety-critical applications implemented in software or hardware, and, 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, and in collaboration with the research laboratory VERIMAG in Grenoble, and the software publisher 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 persisted until 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' called KCG was developed. The first version was released in 1999. It is used in software projects with the most demanding safety levels of many standards (DO-178C, IEC 61508, EN 50128, IEC 60880 and ISO 26262), where high confidence in automation is expected. KCG demonstrated the interest of a semantically well defined language in the context of qualification processes. It is 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 to select them carefully so as to preserve the qualities that made SCADE accepted for safety-critical development. One such feature was the ability to mix models, namely the purely data-flow one already well covered, and control-flow ones better covered by languages like ESTEREL and SyncCharts [9], and to enable interactions between the two. Another limitation of the original SCADE was the lack of arrays. LUSTRE V4 provides powerful recursive array definitions well suited for hardware but imposes static expansion that are unsuitable for software. Finally, there were also requests for other language extensions (such as modules), more expressive types (in particular around numerics), and compiler optimizations.

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

- ESTEREL and SyncCharts for control-dominated systems expressed as hierarchical state machines;
- functional arrays and iterators [10];

[1]SCADE stands for *Safety-Critical Development Environment*.

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

978-1-5386-1925-4/17/$31.00

- LUCID SYNCHRONE [11], [12], [13] for the integration of data-flow, control-flow and type-based analyses.

Several other works were instrumental. For example, *mode automata* [14] made a proposition for mixing a subset of LUSTRE and the hierarchical automata of ARGOS [15] in a single model, but left several questions unanswered, in particular, the integration of such features into a complete language. The language ESTEREL V7[3] did integrate data-flow and control-flow features but it was tuned to generate efficient hardware. How to adapt it for software and to integrate it into a qualified compiler was unknown at that time.

The main design decision was to base the language and compiler on the following ideas: (1) define a minimal kernel language together with a static and dynamic semantics and use it as some kind of 'typed assembly language' from which to produce sequential code; (2) express richer programming constructs in terms of the basic language by source-to-source translations, and (3) give a static and dynamic semantics for all language constructs and ensure their preservation by the translations. For the kernel language, we defined a *clocked data-flow language*, similar to LUSTRE but with some modifications that we motivate in this paper.

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

In this paper, we describe how the design decisions were followed. We illustrate the main language features, the compile-time static analyses and the compiler architecture. We focus on the language itself; the graphical support and modeling tool are described in [17].

Section II recalls the LUSTRE kernel that underlaid SCADE until 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 styles. Section VI explains the treatment of arrays. Section VII discusses the design and qualification of the code generator. Section VIII concludes the paper.

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.

[3]This was the latest version of ESTEREL, developed at ESTEREL-TECHNOLOGIES EDA. Unfortunately, neither the compiler nor the reference manual are now publicly available after the company ceased to exist in 2008.

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

A. The core LUSTRE language

Sequences are the basic elements of LUSTRE and operations are lifted to apply pointwise, just as in mathematics when writing the pointwise 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 meaning of related expressions can be represented in a table, e.g.,

`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`	$2 * x_0$	$2 * x_1$	...	$2 * x_n$	...

The unit delay, **pre** (for 'previous'), is an important primitive:

`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 corresponding 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	...

Or, 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 $n \in \mathbb{N}$,

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

The last important notion is that of a clock. The clock of a stream indicates when its value is present (or ready). Clocks are modified by two operators, **when** and **current**. The former filters a stream according to a boolean condition, e.g.,

`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 rather indicates the absence of a value. Thus, the stream `x` **when** `h` is the sub-sequence $x_0, x_2, x_3, \ldots$. 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 whose input and output are on the same clock, then `f(x` **when** `h)` has clock `h` and gives the application of `f` to the subsequence of `x` filtered by `h`. Note that `h` can be any boolean expression and thus, it can encode a periodic clock.

A stream can be completed by maintaining 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 a clock h, the clock of **current** `a` is the clock of the clock of `a`. Hence, `a` cannot be on the fastest clock, the so called 'base' clock, of the system. **current** provides a 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`.

A program can be *synchronously executed* when the execution can proceed as a global sequence of steps where any stream expected to be present is indeed present and otherwise 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 any program that actually uses a stream at a clock different to that which is expected. E.g, writing `x + (x` **when** `h)` is rejected because this operator expects both 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 previous 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. Equations can be introduced or removed without changing the semantics of the node. E.g., the following node computes the exact same sequence as the previous one, but uses 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, $\forall n \in \mathbb{N}$, $x_n = e_n$.

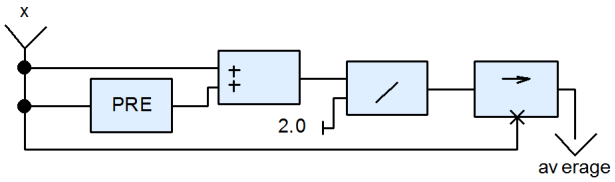

Fig. 1. The sliding average diagram

B. The question of determinism

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

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

Here, `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 — does 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 as a SAT problem. Yet, the effective complexity is hard to predict 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 uninitialized delay and refuses to compile a program where nil may happen anywhere but in the first position of a sequence.

III. SCADE 6: A NEW DATA-FLOW CORE

Instead of **current** we borrowed an alternative operator **merge** from LUCID SYNCHRONE. It 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);
```

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

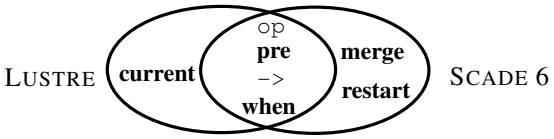

Fig. 2. Data-flow Cores

Unlike **current**, **merge** does not introduce a nil. Moreover, its implementation only uses local variables, rather than a memory, and is easier to compile efficiently. Finally, in LUSTRE one often uses equations of the form `o = if h then current a else current b;` with `a` on clock `h` and `b` on clock **not** `h`. This combination of elements is difficult to compile efficiently. It uses two memories (one for each **current**), that are difficult to eliminate, and three conditionals on `h`, one for every **current** plus the one of the condition, which require fusion. 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].

The second change to the data-flow core is the addition of a reset operator. Unlike ESTEREL, where it is a fundamental feature, LUSTRE does not provide a means to modularly reset a system on a Boolean condition, that is, to reinitialize all its state variables. This forces the user to explicitly add reinitialization conditions all over a program. For example, let `sum` be the following node:

```
node sum (x:int) returns (o:int);
  o = x + (0 -> pre o);
```

having a resetable version of this operator requires to replace `sum` by:

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

The SCADE 6 core, on the other hand, is extended with a built-in construct for resetting any node instance. For example, the expression `(restart sum every c)(e)` reinitializes the node instance `sum` when `c` is true. This reset primitive was first introduced in [20]. Here is the resetable version in SCADE 6:

```
node resetable_sum (x:int; c:bool) returns (o:int);
  o = (restart sum every c)(x);
```

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

SCADE 6 extends the dynamic semantics of LUSTRE to take the **merge** and modular reset constructs into account [22]. The static semantics encompasses 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 the type of a function contains all the information needed to check the correct use of that function.
- It allows good error diagnostics to be given, as far as the type language is simple enough.

The four dedicated type systems of the SCADE 6 compiler are summarized below. They are applied in sequence: if one fails, compilation stops. The type systems are presented in order of their application in KCG.

A. Types

The first, and relatively standard, static verification step is the *type checking*. Its main features are:

- all types must be declared; a type is 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 classes **numeric**, **float**, **integer**, **signed**, or **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 vectors 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 types 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-in and cannot be defined by users. Moreover, the language is first-order, that is, 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. A second step, after static expansion has been performed, checks that equality constraints are trivial.

Finally, types incorporate an attribute that is specific to SCADE 6. Combinatorial functions (whose current outputs only depend on current inputs) are given the kind **function** whereas stateful functions (whose outputs may also depend on past values) are given the kind **node**. Kinds are checked during typing in a simple way: if a function is declared with kind k, all the functions it calls must be at most of kind k i.e. nodes can call both nodes and functions while functions can only call functions.

The compiler imposes a strong typing discipline: programs which do not type check are rejected. Well typed programs satisfy the following property.

Property 1 (Well typed program execution): In a well typed program,

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

B. Clock checking

The clock analysis ensures that programs can be executed synchronously. Once done, every expression is clocked with respect to the global time scale, named the *base clock*. More precisely, the clock of a stream is an expression of the following language: $ck ::= ck \,\mathtt{on}\, e \mid \alpha$ 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 and only if e_2 is present with clock e_1 and equal to true. An expression with clock variable α is present if and only if α evaluates to true. Clock checking exists 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 uses a simpler formulation where equivalence between Boolean expressions is replaced by equivalence between clock names [27]. It further simplifies this approach by requiring that the clocks of variables be declared and that function types involve only a single clock variable (α). The original proposal, implemented in LUCID SYNCHRONE V3, does not impose this second restriction. Moreover, clocks were inferred.

Given a function definition, the compiler checks clocks and computes a clock signature. The signature for the function `hold` (Section III) is: $\forall\alpha.\alpha \times (h : \alpha) \times \alpha \,\mathtt{on}\, h \to \alpha$. 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 $\alpha \,\mathtt{on}\, h$. 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 all imported functions do so too.

C. Causality analysis

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

Following preliminary work [29], the causality analysis for SCADE 6 was specified as a type system. The intuition is to associate a time stamp to every variable and to check that the relation between time stamps is a partial order (and thus free of cycles). 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 $\forall\gamma_1,\gamma_2.\gamma_1 \times \gamma_2 \to \gamma_1$ which indicates that the output only depends instantaneously on the first input. From this signature, one can see that this operator breaks dependency cycles on its second input. `bwd_Euler` has type $\forall\gamma.\gamma \times \gamma \to \gamma$, which indicates the dependency of the output on both inputs. This information suffices to deduce that the second integration function cannot be used to break cycles.

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

D. Initialization analysis

The initialization analysis ensures 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 a type based on the following intuition.

- The type $\mathbf{1}$ signifies a stream that may have an uninitialized value nil at the very first instant;
- the type $\mathbf{0}$ signifies a stream that 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 ;
```

is assigned the initialization type signature: $\mathbf{0} \to \mathbf{1}$ and the function

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

is assigned the signature: $\forall\delta.\delta \times \delta \to \delta \times \delta$.

The initialization analysis does not oblige functions to return well initialized streams. For example, the following function, with signature $\mathbf{0} \times \mathbf{0} \to \mathbf{1}$, is accepted as a node declaration provided it is not the main node.

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

The following function, on the other hand, has signature $\mathbf{0} \times \mathbf{0} \to \mathbf{0}$ and can be used as the main node.

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

The main node defines what is finally executed on the target platform. Its outputs 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 to control 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 the lack of expressiveness of the clock language and of a lack of automation (namely, clock polymorphism and inference).

Clocks exist in SCADE 6 but the language proposes an alternative in the form 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 very useful for ensuring the consistency of language extensions with one other. We were also convinced that the data-flow core was expressive enough to support the 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 that is less expressive than SCADE 6 and not implemented via a source-to-source transformation.

The following sections present and illustrate the new constructs. Their formalization and compilation are presented 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` were 'clocked' by writing an equation `d = sqrt (delta` **when** `(delta > 0))`. Indeed, the translation of the function `second_degree` does precisely 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` together with an execution trace.

```
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 alternately incremented and decremented? One solution is to add the equation `last_o = 0 ->` **pre** `o` in parallel and to read from `last_o` rather than `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` and the second row of the execution trace. The variable `o` is initialized with `0`. The construct **last** 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 for expressing equations of the form `x =` **last** `'x + 1`, which have an imperative flavor, in a data-flow language.

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

C. Hierarchical Automata

State machines are a convenient way to specify sequential behaviour. There are two classical forms:

- Moore machines, where the current output is a function of the current state only;
- Mealy machines, where 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], SyncCharts [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. SyncCharts [9] was the graphical notation used in the industrial toolset based on ESTEREL.

SCADE 6 incorporates hierarchy à la SyncCharts. The states may themselves contain other state machines and data-flow equations. A difference with the approach of ESTEREL/SyncCharts 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. Note that SCCharts [32] also proposes a textual notation for a similar construct. The main features of SCADE 6 hierarchical state machines are borrowed from SyncCharts:

- an automaton must have one *initial* state;
- some states can be marked as *final*;
- transitions can be *weak* or *strong*;
- transitions may either reset or resume their target states;
- a *synchronization* mechanism allows for firing a transition when all the directly enclosed automata are in a final state.

1) Intuitive Semantics: The semantics have been formalized in [33] 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 determining the *active state* from the current *selected state*; executing the corresponding set of equations; and then determine the *selected* state for the next cycle. More precisely,

- At the first cycle, the state marked initial is the *selected state*.
- The guards of all strong transitions from the *selected state* are evaluated. The *active state* is the target of the first, in sequential order, fireable strong transition, and otherwise the *selected state*.
- The equations of the *active state* are executed.
- The guards of weak transitions from the *active state* are evaluated. The next *selected state* is the target of the first, in sequential order, fireable weak transition, and otherwise 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-state 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`. Replacing the weak transition (**until**) by a strong transition (**unless**) gives a causality error.

The second example is a node with two inputs: `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` is simply maintained in the initial state.

3) A complete example: The last example is an adaptation of the ESTEREL version [34] of Harel's digital watch [31], but 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 two 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 sub-scopes 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 permit more concise programs. The implicit equation `x` = **last** `'x;` contributes 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 are equipped with a collection of standard iterators (e.g., **map**, **fold**). The iterators are free of side effects and so preserve the functional style of SCADE 6. They complement, rather than replace, external functions over arrays. In particular, it is possible to map or fold a stateful function pointwise over all elements of an array. Array iterators enjoy several optimizations like the elimination of intermediate copies. The set of array operators and their compilation was 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 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

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

or is the prefix form of the **or** operation. 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 containing the scalar product of two vectors, the vector matrix product and the matrix product. These functions are all polymorphic and apply to any numeric 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 the permutation of the two dimensions of an array of arrays.

B. A stateful example

The following example is inspired by the interface present in a fighter plane, where, because of acceleration, the pilot may not always be able to precisely push the right button. To overcome any risk, command selection occurs in two steps: a preselection that works like radio buttons (selecting one button unselects the others) and a second step with a single button (no choice means no possible selection error) to confirm the preselection. The logic to manage this interface is quite regular and independent of 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`) of buttons are composed in parallel.

Buttons have a background and a foreground color depending on their state. When preselected, its background becomes yellow. When locked, its background becomes green. The node `Button` defines 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.

```
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);

  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, but, as before, the compiler generates a for loop.

VII. Code generation

A. Compiler Organization

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

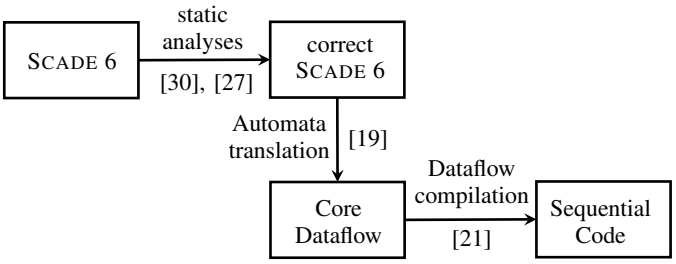

Fig. 4. SCADE 6 Compiler Organization.

Within the transformations, many optimisations are performed on the data-flow form (dead-code elimination, constant propagation, common sub-expression elimination, 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 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 chose OCAML [35] which in 2005 presented quite a challenge for a qualified tool. Indeed, certification standards often push companies to use well established technologies. We thus had to provide convincing evidence that OCAML was well adapted to write a compiler. The argumentation was built on the small distance between the formal specification and its implementation in OCAML. This industrial use of OCAML in a certified context is detailed in [36] and [37].

The current version of SCADE KCG comprises approximately fifty thousands lines of code (50 KLoC) and 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 over more than ten years to integrate new language features. The detailed design is more than one thousand pages long.

C. Towards Computer Aided Formal Qualification

The formalization of SCADE 6 was an important step in the development of 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 to the full language and maintaining a high level of confidence in the proof without the help of a computer appeared infeasible.

Proof assistants like COQ [38] allow for writing both programs, properties and computer checked proofs. The

COMPCERT C compiler [39], [40] is the first compiler developed in 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 involves more than just scientific questions.

The next step for SCADE 6 and KCG is to go further by using computer aided tools to obtain a proof of correctness of the compiler. Connecting this new object with the COMPCERT C compiler would yield 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 reset [41]. 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 presented the 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.

While the core of the language is about the same size as LUSTRE, the new features it proposes are a big improvement for SCADE users. The mix of fine grained data-flow equations and hierarchical automata is quite unique. The language is now as convenient for the development of the logic of a cockpit display as it is for the description of discrete control laws.

SCADE and KCG have been used in about one hundred DO-178B/C level A avionic systems worldwide; a quite significant result in this market.

And the story continues. The language already offers a good coverage of discrete-time systems programming, and adding continuous time modeling capabilities could be an axis of development for the near future. The work on ZÉLUS [42], [43] moves in this direction in the same spirit of collaboration between academia and industry,

ACKNOWLEDGMENT

This works owes a lot to Paul Caspi; and Gérard Berry for our lively discussions. We also want to thank all our colleagues in the Core Team for their hard work on SCADE 6 KCG; and our CTO, Bernard Dion, for his confidence and support. We thank the TASE 2017 organisers for inviting us to write this article. We warmly thank Timothy Bourke and Cédric Pasteur who provided numerous comments and helpful suggestions.

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 [44].

[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. 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.

[22] C. Auger, J.-L. Colaço, 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] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for safety-critical applications," in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Edinburgh, UK: ACM, June 2014.

[33] 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.

[34] G. Berry, "Programming a digital watch in Esterel V3," INRIA, Tech. Rep. 1032, 1989.

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

[36] 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.

[37] 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.

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

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.