Verimag

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

# Formal System Development with Lustre: Framework and Example

*Jan Mikáč, Paul Caspi*

**Report n$^o$ TR-2005-11**

V0.1 - June 2005

Reports are downloadable at the following address
http://www-verimag.imag.fr

UNIVERSITE JOSEPH FOURIER

CNRS

INPG

# Formal System Development with Lustre: Framework and Example

*Jan Mikáč, Paul Caspi*

V0.1 - June 2005

**Abstract**

This paper proposes a refinement framework for Lustre. First a very general calculus is provided, which ensures correctness and reactivity for a large class of systems. Then, this calculus is adapted to provide oversampling and temporal refinement. We obtain thus an effective calculus for Lustre, which allows us to refine both computations and time. The calculus and its use in the development of reactive systems are illustrated on the island example used by J.R. Abrial for presenting the B system method.

**Keywords**: Lustre, refinement, temporal refinement

**Reviewers**:

**How to cite this report**:

@techreport { ,
title = { Formal System Development with Lustre: Framework and Example},
authors = { Jan Mikáč, Paul Caspi},
institution = { Verimag Technical Report },
number = {TR-2005-11},
year = { 2005},
note = { available at http://www-verimag.imag.fr/index.php?page=techrep-list&lang=en}
}

# 1 Introduction

## 1.1 Critical Systems

In the domain of critical systems, proving program correctness is important, and often mandatory. Generally, both the system and correctness properties are described in some formalism and then proved. In some cases, when the system is a finite-state one, model-checking is the technique which allows proofs to be performed in a fully automatic manner. (Examples of popular model-checkers are Spin[20] or SMV[27].) In other cases, a theorem-prover (such as PVS[31] or Coq[21]) is used, which requires some user interaction.

However, attempting to prove a property on a final program may turn out to be complex due to the gap between the actual program and its specification, not to mention the possible complexity of the program itself. Moreover, discovering an error at this stage of the development may lead to important efforts for correcting it.

To overcome these difficulties, correct-by-construction programming was proposed with two main ideas : – the program correctness is maintained from the specification to the final product, – the problem of proving a property over a system is decomposed into some (hopefully easier) intermediate steps.

*Refinement* is one such technique: system specifications are expressed within some formal framework and the required properties are proved on them. Then, by repeatedly rewriting the specifications in a more precise form *within the same formal framework*, a machine-implementable form is finally reached. At each step of this refinement process, soundness proofs (known as “refinement proof obligations”) are performed, so that each stage (and therefore the ultimate program) is guaranteed to meet the initial specifications. Some well-known refinement-based development methods are B[3], Z[9] and VDM[23]; a more theoretical framework for refinement is given for instance by TLA[24, 2] or [5].

## 1.2 Critical Control Systems

Computerised control is one of the computer domains which contains the most safety-critical applications and where safe development methods are mostly needed. Just think of fly-by-wire, drive-by-wire, signalling, nuclear plant control systems. Yet, most formal development methods have not been initially designed for that purpose and thus, may not be fully adapted to it, though some truly impressive real-world projects have been achieved in this setting [6].

On the contrary, the Lustre/Scade approach was tailored, from the very beginning [18] to this application domain and has shown many successes in safety critical control, for instance Framatome nuclear plant emergency stop [7], the Hong-Kong subway signalling system [25], the Airbus fly-by-wire systems [10], Audi steer-by-wire systems [1]. The reasons why the approach has been successful seem to rely on two features of the Lustre language, its synchronous dataflow style, which makes it very close to the culture of control engineers, and its simple formal semantic, which has allowed it to be equipped with several useful tools, among which we can cite:

- interfaces from popular control modelling tools such as Simulink/Stateflow [29] and to popular control architectures [12],
- a Do178B level A compiler, which makes it well adapted to meet the needs for certification in the most demanding applications,
- several tools geared toward formal verification, a Prover plug-in [32] and other model-checkers [19], tools for abstract interpretation [22] and an interface [15] to PVS [31], the well-known theorem-proving assistant.

Yet, at present, these tools were not integrated into a complete development framework like the one cited above. Unfortunately, using B or Z for Lustre turns out to be cumbersome[1]: some Lustre features, such as the absence of recursive computations or memory boundedness are possibly lost by embedding Lustre into those far more general systems. Moreover, most of them are rather imperative-language oriented and a translation from Lustre into any of them, followed by a refinement, results in fact in a compilation of the

[1] A translation of Lustre in B is proposed in [17]; for the converse, see for instance [8].

Lustre source. Yet, users of Lustre/Scade are not interested in an imperative semantics of their programs – they use Lustre/Scade because of the dataflow approach and are not likely to switch to an other paradigm during a refinement cycle.

## 1.3 Report Content

Hence, in order to preserve the specificity (and simplicity) of Lustre, we have decided to propose a refinement framework more adapted to the language, a calculus which is inspired of (but not dependent on) the cited ones. Our approach relies on the fact that Lustre can be used to express both programs and invariant properties over them (thanks to the notion of "synchronous observer"[19]), so that Lustre provides its own formal framework. The theoretical part of this report is based on a previous proposal[28], which is here (largely) generalised.

The practical aspects of the proposed refinement calculus are handled by a series of tools. Our (still) experimental Flush tool creates refinement proof obligations, which are then proved thanks to Lesar (a model-checker), nbac[22] (abstract-interpretation tool) or Gloups[16], an interface to the PVS[31] theorem-proving assistant.

In the sequel, we briefly introduce the Lustre language (section 2), then in section 3 we propose a general purpose refinement framework. Section 4 is dedicated to "customising" the previous calculus to the needs of Lustre and to our particular practical needs. This allows us to define a computer-aided refinement calculus, which is both functional and temporal. Section 5 proposes an example of application and section 6 concludes and offers some perspectives.

# Contents

**1 Introduction** **1**
1.1 Critical Systems . . . 1
1.2 Critical Control Systems . . . 1
1.3 Report Content . . . 2

**2 Lustre** **5**
2.1 A Synchronous Dataflow Functional Language . . . 5
2.2 Lustre Operators . . . 5
2.3 Lustre Nodes . . . 6
2.4 Lustre and the Oversampling Question . . . 7
2.5 Lustre Mappings . . . 7

**3 Refinement** **9**
3.1 Systems . . . 9
3.2 Refinement . . . 9
3.3 Transitivity of the Refinement . . . 10

**4 Temporal Refinement** **13**
4.1 Temporal refinement in TLA and in Signal . . . 13
4.2 Oversampling in Lustre . . . 13
4.3 Summary of the Proposal . . . 14
4.4 Fairness and Real-Time . . . 15

**5 Example** **17**
5.1 The Island . . . 17
5.2 A Non Deterministic Controller . . . 18
5.3 Temporal Refinement . . . 19
5.4 A Deterministic Controller . . . 21

**6 Conclusion** **25**

# 2 Lustre

As a starting point, we present the programming language Lustre.

## 2.1 A Synchronous Dataflow Functional Language

Lustre is a **dataflow** language: its inputs and outputs (and local variables as well) are finite or infinite sequences of values of some scalar type. Formally, if $D$ is a domain of values, a flow over $D$ is an element of $D^\infty = D^* \cup D^\omega$ where $D^*$ is the set of finite sequences over $D$ and $D^\omega$ is the set of infinite ones. In this setting, flows are *complete partial orders* with respect to the prefix order over sequences: $x \leq y$ if there exists $z$ such that $y = x \oplus z$ where "$\oplus$" is the concatenation of sequences.

Lustre is a **functional** language: a Lustre program $p$ is a mapping over flows

$$p : (D_1^\infty, \dots, D_n^\infty) \longrightarrow (D_1'^\infty, \dots, D_m'^\infty)$$

Lustre is a **synchronous** language: the flows are consumed or created at "the same speed". To illustrate this, let us take the example of the sum of two integer flows **x** and **y**:

| | | | | |
|---|---|---|---|---|
| **x** | $x_0$ | $x_1$ | $x_2$ | $\dots$ |
| **y** | $y_0$ | $y_1$ | $y_2$ | $\dots$ |
| **x+y** | $x_0 + y_0$ | $x_1 + y_1$ | $x_2 + y_2$ | $\dots$ |

The sum operator + is applied point-wisely on the two flows: the synchrony amounts to the fact that an element of the **x+y** flow exists if and only if the corresponding elements of the **x** and **y** flows exist. In a sequence-like manner, we would define the operator as

$$(x.\mathbf{x}) + (y.\mathbf{y}) = (x + y).(\mathbf{x} + \mathbf{y}) \qquad \text{and} \qquad \varepsilon + \varepsilon = \varepsilon$$

where $\varepsilon$ is the empty sequence and . stands for the sequence constructor ($value.sequence$). One can notice that in the expression $(x + y).(\mathbf{x} + \mathbf{y})$, the first $+$ sums two integers, while the second one sums two sequences.

## 2.2 Lustre Operators

The same point-wise definition applies to other arithmetical and logical operators such as multiplication, conjunction, negation, if-then-else etc.

$$(a.\mathbf{a}) \textit{ and } (b.\mathbf{b}) = (a \wedge b).(\mathbf{a} \textit{ and } \mathbf{b}) \qquad \text{with} \qquad \varepsilon \textit{ and } \varepsilon = \varepsilon$$

$$\begin{aligned} \textit{if } (t.\mathbf{c}) \textit{ then } (x.\mathbf{x}) \textit{ else } (y.\mathbf{y}) &= x.(\textit{if } \mathbf{c} \textit{ then } \mathbf{x} \textit{ else } \mathbf{y}) \\ \textit{if } (f.\mathbf{c}) \textit{ then } (x.\mathbf{x}) \textit{ else } (y.\mathbf{y}) &= y.(\textit{if } \mathbf{c} \textit{ then } \mathbf{x} \textit{ else } \mathbf{y}) \\ \textit{if } \varepsilon \textit{ then } \varepsilon \textit{ else } \varepsilon &= \varepsilon \end{aligned}$$

Lustre contains also operators for memorising the previous value (`pre`), initialising a flow ($\rightarrow$) and their combination (`fby`):

$$\textit{pre } \mathbf{x} = @.\mathbf{x} \qquad (x.\mathbf{x}) \rightarrow (y.\mathbf{y}) = x.\mathbf{y} \qquad (x.\mathbf{x}) \textit{ fby } \mathbf{y} = x.\mathbf{y}$$

Here @ denotes an "undefined" value: it can be any value of the expected type. The @ value exists only in the semantics of the language, for definition purposes; it has no syntactic counter-part and the compiler is able to ensure that no actual computation will involve this value[14].

Finally, there are operators for sampling (`when`) and holding (`current`) flows. Sampling applies to any flow **x** and a boolean flow **c** which is called the clock. When the clock is true, a value is taken from the sampled flow, otherwise there is no output. Holding re-builds a flow from a "slower" one by filling the missing values with the last known one. Example:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **x** | $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | ... |
| **c** | $f$ | $f$ | $t$ | $f$ | $t$ | $t$ | ... |
| **x** when **c** | | | $x_2$ | | $x_4$ | $x_5$ | ... |
| current(**x** when **c**) | @ | @ | $x_2$ | $x_2$ | $x_4$ | $x_5$ | ... |

The gaps between the values of the "**x** when **c**" flow are depicted only for easier reading. Actually, the sequence of values of this flow is $(x_2, x_4, x_5, \ldots)$ with no "missing" or "undefined" values in between: the flow "**x** when **c**" is present only at positions where **c** evaluates to true: we will say that "**x** when **c**" is on clock **c**.

The clock of a flow $f$ is an important information for current: given that a $true$ value of the clock corresponds to an actual value in $f$, and that a $false$ value corresponds to a "gap", it is possible to rebuild a "faster" flow, as in the example above.

The clock of every flow is a statically known information, which is gathered during a checking phase of a Lustre program, known as *clock calculus*. The flow clock is therefore an implicit information attached to any flow and this is why Lustre syntax defines the one-parameter current operator (as above). Yet for the semantic definition of current, it is more convenient to make visible the clock **c** and the default value **v**:

$$\begin{array}{lllll} \varepsilon & when & \varepsilon & = & \varepsilon \\ x.\mathbf{x} & when & f.\mathbf{c} & = & \mathbf{x}\ when\ \mathbf{c} \\ x.\mathbf{x} & when & t.\mathbf{c} & = & x.(\mathbf{x}\ when\ \mathbf{c}) \end{array} \qquad \begin{array}{lll} current(v, \varepsilon, \varepsilon) & = & \varepsilon \\ current(v, f.\mathbf{c}, \mathbf{x}) & = & v.current(v, \mathbf{c}, \mathbf{x}) \\ current(v, t.\mathbf{c}, x.\mathbf{x}) & = & x.current(x, \mathbf{c}, \mathbf{x}) \end{array}$$

Then, we simply define $\texttt{current}(\mathbf{x}) = current(@, clock_of(\mathbf{x}), \mathbf{x})$.

In practice, any boolean flow can act as a clock. This implies the existence of a distinguished clock, the true constant, which is called *basic clock*. Unless stated otherwise, flows are considered to be on basic clock, meaning that they are always present. The basic clock is indeed the basis of the clock hierarchy: in our above example, "**x** when **c**" is on clock **c**, the flow **c** is on true, but true has no clock. As a consequence, the current operator cannot be applied on flows that are on basic clock[2]: no flow can "run faster" than the basic clock.

Other existing Lustre constructs (such as arrays) are only "syntactic sugar" defined on the top of the presented features. In particular, Lustre contains no (dynamic) memory allocation mechanism, nor allows recursive function calls.

## 2.3 Lustre Nodes

Actual Lustre programs are defined by a system of equations such as in the following example

```
node Sum(i:int)
returns (s:int)
var mem:int ;
let
  s = i → (mem + i);
  mem = pre s ;
tel
```

The Sum node has one integer input flow i, one integer output flow s and one local variable flow mem. The system of equations indicates that mem memories the previous value of s and thus s accumulates the sum of all previous values of i.

**Remark:** In Lustre syntax, any type is lifted to the equivalent type of flows. Thus int means here $int^\infty$. In the sequel, we adopt this convention.

An example of execution of the Sum node could be

| | | | | | | |
|---|---|---|---|---|---|---|
| **i** | 2 | 3 | -1 | 0 | 4 | ... |
| **mem** | @ | 2 | 5 | 4 | 4 | ... |
| **s** | 2 | 5 | 4 | 4 | 8 | ... |

[2]This condition is checked during the clock calculus.

Formally, the semantics of a Lustre program (node) is the least fixpoint associated with the system of equations. The existence and uniqueness of the least fixpoint are ensured by the Kleene theorem applied to the complete partial order of flows. In order to apply this theorem, we need to restrict the Lustre primitive constructs to be *continuous* mappings over flows, *i.e.,* such that $\sup_i f(x_i) = f(\sup_i x_i)$ for any increasing sequence $x_i$ of flows. It is noticeable that continuity implies monotony: a mapping over flows $f$ is monotonic if, for any $x, y$,

$$x \leq y \Rightarrow f(x) \leq f(y)$$

which, in the context of the prefix order over sequences can be interpreted as causality: the future of inputs cannot influence the past of outputs. This is why we can sequentially construct the outputs as a function of the inputs.

Thus, in the `Sum` example, the node describes the function (we have eliminated the `mem` variable):

$$\lambda i : \mathtt{int}^\infty.\ \ \mu s : \mathtt{int}^\infty.\ \ s = hd(i).(s + tl(i))$$

where $hd$ and $tl$ stand for the destructors of sequences.

## 2.4 Lustre and the Oversampling Question

In section 2.2, we have introduced clocks and mentioned the possibility of dealing with flows on different clocks. This is done by the means of the (over-loaded) keyword `when`. Thus in the `Sum` example, we could have declared `i` as explicitly being on the basic clock: "`i:int when true`".

Lustre imposes that a boolean flow be declared *before* it is actually used as a clock. Thus, the following node header is correct

```
node foo(...c:  bool; ...  x:  int when c; ...)
```

while it would be rejected if `x` were declared before `c`.

This simple rule, together with the clock calculus, which checks whether operators other than `when` and `current` combine only flows on the same clock, ensures that Lustre programs are synchronous. In particular, no Lustre node may have an output which would run faster than any of the inputs.

This amounts to saying that Lustre does not allow "oversampling", which contrasts with Signal [26] and Lucid Synchrone [13]. However, section 4 will propose a means to overcome this limitation.

## 2.5 Lustre Mappings

We can now see which mappings over flows can be *Lustre mappings*: they are

- built with Lustre operators
- which fulfil the type checking and the clock checking conditions

It can be shown[11] that these mappings are:

- continuous, hence causal,
- finite memory (because of the clock calculus and because a program can use only a finite number of memory operators),
- and without oversampling.

# 3 Refinement

After introducing Lustre, let us formalise the refinement problem and give some general results about it.

## 3.1 Systems

We generalise the notion of program in that we consider it to be a relation (rather than a function) between inputs and outputs: thus we shall be able to express even non-deterministic systems, which is impossible in standard Lustre[3]. For the sake of simplicity, we do not distinguish local variables and outputs: in fact both of them behave in the same way, except for the sake of scoping.

We consider that the inputs have some type $T$, the outputs some type $T'$. A program $S$ is thus a *total relation*[4] $S \subseteq T \times T'$. Graphically, we will note it as

$$T \overset{S}{\text{———}} T'$$

Moreover, we consider a property $P$ over the inputs ($P \subseteq T$) – called pre-condition – and a property $Q \subseteq T \times T'$ – called post-condition. By considering the couple $(P, S)$, we obtain a *partial relation* which is $S$ with domain restricted to $P$. This is comparable to operations in B[3]: an operation specification in B consists of a precondition (which defines when it is legal to use it) and a body which describes the actual behaviour.

$P$ and $Q$ form the interface of the system $S$: $Q$ is intended to provide an abstraction of the actual computations carried out in $S$. We can compare it to the invariant in B.

$$P: \quad T \overset{S}{\text{———}} T' \quad : Q$$

We define two meta-properties over $S$:

- $S$ is **correct** with respect to its interface ($P$,$Q$) iff

$$[\mathrm{COR_S}]\ \forall (x,y) \in T \times T'.\ P(x) \wedge S(x,y) \Rightarrow Q(x,y)$$

  *i.e.,* the pre-condition and the relation ensure the post-condition.

- $S$ is **reactive** with respect to $P$ iff

$$[\mathrm{REA_S}]\ \forall x \in T.\ P(x) \Rightarrow \exists y \in T'.\ S(x,y)$$

  *i.e.,* any input which fulfils the pre-condition yields at least one output.

## 3.2 Refinement

Informally, we want the refinement to behave as follows: if a system $S$ is refined by $S'$, we should be able to use $S'$ instead of $S$, while preserving its correctness and reactivity.

For the sake of generality, we suppose that the two systems $S$ and $S'$ are not necessarily typed in the same way: $S \subseteq T \times T'$ and $S' \subseteq U \times U'$. Therefore, we suppose the existence of two (*Lustre*) mappings $\sigma : T \longrightarrow U$ and $\tau : U' \longrightarrow T'$ which ensure the corresponding changes of variables.

$$\begin{array}{ccccccc} P: & & T & \overset{S}{\text{———}} & T' & & :Q \\ & \sigma & \downarrow & & \uparrow & \tau & \\ P': & & U & \overset{S'}{\text{———}} & U' & & :Q' \end{array}$$

[3] There is an other way to introduce non-determinism, which is using an additional input flow featuring the "random choice". However, as we address program development, we consider that *relations* are natural generalisations of functions, so that it is easy to understand that a relational specification can be refined to a functional program, whereas programs with $n+1$ inputs are not natural generalisations of programs with $n$ inputs.

[4] A relation $S \subseteq T \times T'$ is total if its domain is $T$, *i.e.* if $\forall t \in T.\exists t' \in T'.S(t,t')$. We choose that $S$ be total because it will be described in Lustre, which would make it total anyway.

Then, inspired by sufficient conditions given in [3] and by our previous work[28], we define the following three proof obligations:

$$\begin{array}{ll} [\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}] & \forall(x,y') \in T \times U'. P(x) \wedge S'(\sigma(x),y') \Rightarrow S(x,\tau(y')) \\ [\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}] & \forall x \in T.\ P(x) \Rightarrow P'(\sigma(x)) \\ [\mathrm{COR}_{\mathrm{S}'}] & \forall(x,y) \in U \times U'.\ P'(x) \wedge S'(x,y) \Rightarrow Q'(x,y) \end{array}$$

and we say that $S'$ refines $S$ (noted $S \sqsubseteq S'$) if the preceding conditions hold.

It is possible to prove that under these conditions, $S'$ preserves the correctness of $S$ and that the reactivity of $S'$ gives the one of $S$. More precisely, the following sequents are true:

$$\frac{[\mathrm{COR}_{\mathrm{S}}] \wedge [\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}]}{[\mathrm{COR}_{\tau\circ\mathrm{S}'\circ\sigma}]}$$

$$\frac{[\mathrm{REA}_{\mathrm{S}'}] \wedge [\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}] \wedge [\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}]}{[\mathrm{REA}_{\mathrm{S}}]}$$

The correctness preservation property is clearly what we expected: if a system $S'$ verifies $[\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}]$ then any property $Q$ that holds for $S$ under the precondition $P$, holds also[5] for $S'$ under $P$ and thus $\tau \circ S' \circ \sigma$ may be used instead of $S$.

The reactivity property may seem more surprising: one could expect it to go as "if $S$ is reactive, then $S'$ is reactive", but it turns out to be exactly the opposite. This is a well-known problem[6] which in our case amounts to saying that a non-reactive specification cannot be refined to a reactive program; on the other hand, a reactive specification can possibly be refined to a reactive implementation.

In fact, Lustre programs are always reactive, so that exhibiting a Lustre program that refines a given specification proves the reactivity of the latter.

## 3.3 Transitivity of the Refinement

At the end on the previous section, we have somehow anticipated the following result: the refinement is transitive

$$S \sqsubseteq S' \wedge S' \sqsubseteq S'' \Rightarrow S \sqsubseteq S''$$

Thus, for instance, the correctness of $S$ is preserved through iterated refinements until $S''$. In the case of a system $S'' \subseteq V \times V'$ with its change-of-variables mappings $\rho : U \longrightarrow V$ and $\phi : V' \longrightarrow U'$,

$$\begin{array}{ccccc} P: & T & \overset{S}{\text{———}} & T' & :Q \\ & \sigma \downarrow & & \uparrow \tau & \\ P': & U & \overset{S'}{\text{———}} & U' & :Q' \\ & \rho \downarrow & & \uparrow \phi & \\ P'': & V & \overset{S''}{\text{———}} & V' & :Q'' \end{array}$$

the precise transitivity lemmas are

$$\frac{[\mathrm{COR}_{\mathrm{S}}] \wedge [\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}] \wedge [\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}] \wedge [\mathrm{COR}_{\mathrm{S}'\to\mathrm{S}''}]}{[\mathrm{COR}_{\tau\circ\phi\circ\mathrm{S}''\circ\rho\circ\sigma}]}$$

$$\frac{[\mathrm{REA}_{\mathrm{S}''}] \wedge [\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}] \wedge [\mathrm{REA}_{\mathrm{S}\to\mathrm{S}'}] \wedge [\mathrm{COR}_{\mathrm{S}'\to\mathrm{S}''}] \wedge [\mathrm{REA}_{\mathrm{S}'\to\mathrm{S}''}]}{[\mathrm{REA}_{\mathrm{S}}]}$$

[5] As a consequence, under the pre-condition $P$, the system $S'$ verifies $Q$ (thanks to $[\mathrm{COR}_{\mathrm{S}\to\mathrm{S}'}]$) and $Q'$ (thanks to $[\mathrm{COR}_{\mathrm{S}'}]$). Thus, under $P$, the full post-condition of $S'$ is $Q \wedge Q'$.

[6] Think of feasibility in B[3]: in B, we prove the implementability of an abstract machine by refining it to an actual implementation – here, we propose to prove the reactivity of a specification by refining it to a reactive program.

Hence the refinement process can be applied in a step-wise manner:

$$\text{if } S_1 \sqsubseteq S_2 \text{ and } S_2 \sqsubseteq S_3 \text{ and } \dots \text{ and } S_{n-1} \sqsubseteq S_n$$
$$\text{then } S_1 \sqsubseteq S_2 \sqsubseteq S_3 \sqsubseteq \dots \sqsubseteq S_n \text{ and thus } S_1 \sqsubseteq S_n$$

---

In this section, we stated the formal framework of our research and defined a first refinement calculus. This refinement is very general: the purpose of the next section is to "customise" it to our precise needs.

# 4 Temporal Refinement

The two previous sections presented the Lustre language and some general refinement mechanism. Here, we adapt the refinement to our needs (Lustre) and our effective proving possibilities. This adaptation will result in a functional and temporal refinement calculus for Lustre.

## 4.1 Temporal refinement in TLA and in Signal

Temporal refinement is proposed in TLA[24, 2]: if the state of the abstract system does not change ("the system is stuttering"), then any behaviour of the concrete system during that time is refining the abstract one. In this setting, passing from an abstract system to the concrete one adds stuttering (analogous to $current$); the reverse operation erases stuttering (analogous to $when$).

Another comparable approach is the one adopted in the programming language Signal[26, 30]. Where TLA makes the abstract system stutter (keeps the current state), Signal uses special "absent" values to denote the fact that "nothing new happens".

When we try to adapt these approaches to our language, we find the apparent difficulty that in Lustre, we do not have implicit stuttering nor implicit clocks. The only way of getting an equivalent effect is to introduce an explicit clock which runs faster than any inputs of the system we want to refine. However, as we already explained in section 2.4, Lustre does not allow such an "oversampling". This is why it has always been thought that such a temporal refinement was impossible in Lustre.

## 4.2 Oversampling in Lustre

In this section, we present a way of overcoming the difficulty and allowing oversampling in Lustre. The key point is the use of pre-conditions which allow us to move from total to partial relations.

Consider any Lustre mapping $f : Bool \times T \rightarrow Bool$, which is clock-preserving, *i.e.,* such that the clock of the output is the same as the clock of its first input. For easier reading, we will indicate the clock of an expression by using a :: separator. Thus,

$$f : Bool \times T \rightarrow Bool :: \alpha \times \alpha \rightarrow \alpha$$

Given $f$, we can design the following mapping $clk$

$$\begin{array}{rcl} clk & : & T \longrightarrow Bool \\ clk(x) & = & ck \\ \text{where } ck & = & true \rightarrow pre(f(ck, current(@, ck, x))) \end{array}$$

The clocks in the above expression are

$$\left.\begin{array}{rcl} x & :: & ck \\ ck & :: & \alpha \end{array}\right\} \Rightarrow \quad current(@, ck, x) :: \alpha \quad \Rightarrow \quad \underline{ck :: \alpha}$$

and thus $clk :: ck \rightarrow \alpha$. As we can see, the $clk$ mapping is built on a Lustre expression. The use of one $pre$ makes it causal and finite memory, and the only thing that forbids it to be a Lustre mapping is the global oversampling: the input of $clk$ is on clock $ck$ and its output is on $\alpha$; yet, as $ck$ is on $\alpha$ ($ck :: \alpha$), so that the output of $clk$ runs faster than its input.

Having $clk$, we can design the (non Lustre) change of variables:

$$\begin{array}{rcl} \sigma & : & T \longrightarrow Bool \times T \\ \sigma(x) & = & clk(x), x \end{array}$$

This change of variables leaves the input $x$ unchanged, it provides only a boolean flow $ck$ which runs faster than $x$, which amounts to making $x$ stutter.

**Example**

```
node Plus2(i:int)
returns (s:int)
let
  s = i + 2 ;
tel
```

The `Plus2` node works as follows: at every instant, it reads the input value and outputs it after adding 2.

For efficiency reasons, we might want to implement the "+2" operation as two consecutive "+1"s. To allow this, we need to "make time run faster" for a node that would refine `Plus2`. Here, we could use the mapping $f_1$ defined by $f_1 = true \rightarrow not\ pre\ f_1$ which yields a sufficient speed-up: $f_1$ is $true$ once every two "ticks". The following diagram presents the idea:

Here we depict an example of evolution of the value of `i` (the thick line), as seen from within `Plus2`: every change of value of `i` corresponds to a "tick" of the basic clock of `Plus2`. These ticks are marked as *at* for *abstract tick*.

The same evolution of `i`, as seen from a node which benefits from the change of variables given by $f_1$. The input is unchanged, but there are twice as many base clock ticks *ct* (*concrete tick*). Thus the refined node has enough time to apply two $+1$ increments before the output is due.

□

Now we can see that refining some system $S(x, y)$ to $S'((ck', x'), y') = S'(\sigma(x), y)$ can be described, **from within the system** $S'$, by:

- the (trivially Lustre) identity change of variable on $x$
- and an extra pre-condition

$$Clk_P'(ck', x') = (ck' = true \rightarrow pre(f(ck', current(@, ck', x'))))$$

which makes it a *partial relation*.

## 4.3 Summary of the Proposal

To conclude the theoretical part, we give here a summary of the proposed refinement, obtained by combining the general refinement with the temporal one. The following formulas use the notations established in section 3.1. We suppose that the system $S$ has been proved correct: the aim of the following is to prove that $S'$ is correct and refines $S$.

**Syntactic and Static Check Obligations**

1. $\sigma : T \longrightarrow T'$ is a Lustre mapping
2. $\tau : U' \longrightarrow U$ is a Lustre mapping
3. $f : Bool \times T' \longrightarrow Bool$ is a clock-preserving Lustre mapping
4. $Ck_P'(ck', x') = (ck' = true \rightarrow pre(f(ck', current(@, ck', x'))))$
5. $P'(ck', x') = Ck_P'(ck', x') \wedge D_P'(x')$

**Proof Obligations**

6. $[\mathrm{COR}_{\mathrm{S}\rightarrow\mathrm{S}'}]$
$\forall (x, y') \in T \times U'$.
$P(x) \wedge Ck_P'(ck', \sigma(x)) \wedge S'((ck', \sigma(x)), y') \Rightarrow S(x, \tau(y'))$
7. $[\mathrm{REA}_{\mathrm{S}\rightarrow\mathrm{S}'}]$
$\forall x \in T.\ P(x) \Rightarrow D_P'(\sigma(x))$
8. $[\mathrm{COR}_{\mathrm{S}'}]$
$\forall (x', y') \in U \times U'.\ P'(x') \wedge S'(x', y') \Rightarrow Q'(x', y')$

We can notice that:

- In the proof obligation (7) we do not need to establish the clock pre-condition $Ck_P'$, which is here only for coding the oversampling aspect of the change of variable.
- On the contrary, once the refinement has been performed, this clock pre-condition becomes an integral part of the refined system and is treated as a component of the overall precondition of the new system for implementation as well as further refinements. This is stated at condition (5). We see here that pre-conditions are split into two parts, a clock pre-condition $Ck_P$ and a data pre-condition $D_P'$.

## 4.4 Fairness and Real-Time

The meaning of the $ck'$ clock (defined at point 4 of the summary) is: "*when $ck'$ is true, both the abstract system $S$ and the concrete system $S'$ evolve; when false, only the concrete one runs*". As we have imposed no particular condition on the values returned by the function $f$ which defines $ck'$, one can wonder what happens *if $ck'$ becomes false at some point in time and remains false forever*?

Such a setting corresponds to a situation where the abstract system advances up to a certain point (if $ck'$ has $n$ true values, the abstract system runs through its first $n$ reactions), then the $n+1$th abstract instant is infinitely refined in the concrete system ($ck'$ is false forever after). Several interesting points can be discussed here:

**Theoretical Correctness** The correctness proof obligation (6) states that any output $y'$ of the concrete system $S'$ has an admissible counter-part in $S$, which is $\tau(y')$. Given that $\tau$ is a Lustre mapping, it is either clock-preserving or sampling. Thus, the $y'$ flow is at least as long as $\tau(y')$.

This means that even if $ck'$ becomes false forever, the proof obligation (6) ensures that the output of the concrete system will be correct and in particular will be of the length required by the abstract specification.

Within this point of view, the "$ck'$ becomes false forever" case corresponds to a setting in which the output flow depends only on a finite number of the input flow values, so that stopping to read the input after the last relevant value *is* a correct refinement of the system.

**Correctness Proof** Our translation of the proof obligation (6) in Lustre cannot however encompass the whole meaning of (6). In particular, we cannot compare the whole outputs of the abstract and the concrete systems, because the outputs are *not synchronous*: at the point when $ck'$ becomes false forever, the concrete system outputs an infinite number of values before the abstract system outputs a single value.

Thus, the proof obligation created by Flush contains only the safety part of (6), which is the comparison of the outputs up to the point when $ck'$ freezes.

**Fairness Condition** To avoid the before mentioned drawbacks, we can add a simple sufficient condition that prevents $ck'$ from creating an infinite refinement:

> Any *false* value in the $ck'$ flow may be immediately[7] followed by only a finite number of *false* values.

This condition, together with the length preservation discussed in the Theoretical Correctness paragraph, ensures that each abstract instant is refined by a finite number of concrete instants. However, we cannot verify in practice whether this condition holds, since Lustre observers cannot handle liveness properties. Therefore, we define a more restrictive condition below.

**Real-Time Temporal Refinement** We can derive a safety property from the above condition:

> Given a positive integer $N$, any *false* value in the $ck'$ flow is immediately followed by at most $N - 1$ *false* values.

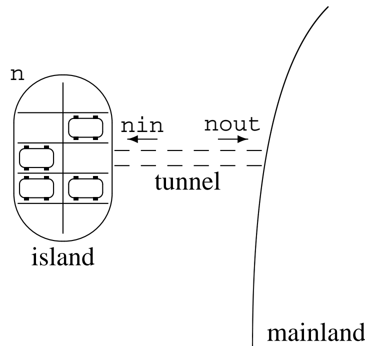
Knowing $N$, a Lustre observer of the above property can be written, so that we can verify whether the condition holds. We say that this condition establishes a *Real-Time Temporal Refinement*, because in such a setting, each abstract instant is refined by at most $N$ concrete ones, so that the worst-case execution time can be calculated. The case of periodical refinement, such as in the `Plus2` example, is a special case of the real-time temporal refinement.

[7] "immediately followed" means "without any *true* value in between"

# 5 Example

To illustrate our refinement calculus, we will follow the "island example"[4] proposed by J.R. Abrial for presenting the development of systems in B.

## 5.1 The Island




The setting is: there is an island which can be reached from the mainland by a tunnel. Cars can be parked on the island, but the capacity of the parking lot is limited. Formally, the island contains initially `ninit` cars. At each time unit, `nin` cars enter and `nout` cars exit the island. The required property is that the total number of cars `n` in the island never exceeds a given value `nmax`.

**A first specification** The formal translation in Lustre of this informal specification is shown at table 1. Besides constant declarations, the specification is made of two components, the `island` model and the

```
const ninit : int;

node island(nin, nout : int)
returns(n : int);
let
  n = (ninit -> pre n) + nin - nout;
tel

const nmax : int;

node island_post(nin, nout, n : int)
returns(prop : bool);
let
  prop = (n <= nmax);
tel
```

Table 1: The island initial specification

required property which is considered as a post-condition of the former, which means that we want the boolean flow `prop` to always yield the value `true`. This amounts to the synchronous observer technique promoted in [19].

**Improving the specification** When we try to check or prove the property, we obviously fail. At this point, we may think that this is because we have not captured all the necessary properties of the specification. Therefore we add another *pre-condition* node (table 2) stating some useful assumptions, such as the fact that the initial value `ninit` is smaller than the required maximum value `nmax`.

**Trying to prove it** We said that we had tried to prove the property but we did not say how. In fact, in terms of section 4.3, we attempted a proof of correctness of the `island` system ($[\text{COR}_{\texttt{island}}]$). We

```
node island_pre(nin, nout : int)
returns(prop : bool);
let
  prop = (0 <= ninit) and (ninit <= nmax) and
         (0 <= nin)   and (0 <= nout);
tel
```

Table 2: Useful pre-conditions

used the Flush tool to automatically build a proof Lustre node, whose only output is the truth value of the property we want to prove (see table 3). Then this node can be dispatched to any convenient proof tool, like the ones cited in introduction.

```
node island_proof(nin, nout, n : int)
returns(prop : bool);
let
  assert island_pre(nin, nout);
  prop = island_post(nin, nout,
                     island(nin, nout));
tel
```

Table 3: The proof node

Yet, clearly enough, the proof attempt fails.

### 5.2 A Non Deterministic Controller

This failure, yet, is interesting in that it shows us that our system model can not operate properly without a controller. In this sense, we can hope that our modelling is unbiased and we can now concentrate on designing a good controller. In doing this, we shall take the greatest care of not modifying our model of the system to be controlled. This will be obtained thanks to the good properties of Lustre: a Lustre node is a function and, as such, is free from side effects. If, in the course of the project, we neither modify `island` nor `island_pre`, we are guaranteed not to modify our model.

**Remark:** This methodology, inspired by control principles, is quite different from the one followed by Abrial, which is more of computer science inspiration. In Abrial's approach, the island and the controller are jointly modelled and the property is ensured from the very beginning. It is only at the end of the refinement process that the island and the controller get separated. Then, the absence of modelling bias is harder to assert.

The first controller we design is a non deterministic one, which "magically" ensures the desired property. Its design is shown at table 4. It just computes the truth value of the desired properties. Then, we can encapsulate both `island` and `controller` within a global model, called `controlled_island` (table 5) where we ensure, thanks to the assertion mechanism, that the truth value computed by the controller remains always true.

We can notice here that the `controller` uses the `island` model. Yet, this does not mean that it knows by magics the number of cars in the island. The key issue here is that, since functions over flows are dynamical systems, two instances of the same function behave in the same manner if and only if they have exactly the same inputs. In this sense, the `island` node instantiated in `controller` may have behaved differently than the actual island.

```
node controller(nin, nout: int)
returns(prop: bool);
let
  prop = island_pre(nin, nout) and
         island_post(nin, nout,
                     island(nin, nout));
tel
```

Table 4: A non deterministic controller

```
node controlled_island()
returns(n : int);
var nin,  nout: int;
let
  assert controller(nin,  nout);
  n = island(nin, nout);
tel

node controlled_island_post(n: int)
returns(prop : bool));
let
  prop = (n <= nmax);
tel
```

Table 5: The controlled island

Finally, we generate as before the proof node `controlled_island_proof` and then, by expanding it, the property holds by the sake of simple rewriting.

### 5.3 Temporal Refinement

We thus have at this step a correct controller. Yet, it is not effective for at least two reasons: the fact that it is non deterministic and it needs sensors for measuring the flows of input and output cars. Let us first address the second issue. The problem here is that we only have boolean sensors, which send a boolean `true`, each time a car passes in front of them and this raises a question of temporal refinement.

**The sensor handlers** Given the `current` and `when` primitives, writing sensor handlers is fairly easy. The result is displayed at table 6.

At section 5.1, we said that there was some number of cars per time unit without saying what the time unit was. We can now make this statement more precise by relating the speed of cars and the time unit thanks to the constant `nm` which gives the maximum number of cars that can pass in front of a sensor per time unit.

We can note moreover here that the clock generated by the `countmod` node is trivially periodic of period `nm`. Thus, the fairness issues raised at section 4.4 are fulfilled.

Then, the `countmod` node provides the "time-unit" clock, by counting modulo `nm`. Finally, the `carcount` node provides, at each time unit, the number of cars that passed the sensor at that time, as a function of the sensor boolean output and of the "time-unit" clock. Note the use of the `when` sampler. As a consequence, the output flow of the node is "slower" than its inputs.

```
const nm: int;

node countmod() returns (clock: bool);
var n:  int;
let
  n = ((0 -> pre n) mod nm) + 1;
  clock = (n = nm);
tel

node carcount(car, cl: bool)
returns (ncar: int)
var nc: int;
let
  nc = (if true -> pre cl then 0
         else (0 -> pre nc))
         + if car then 1 else 0;
  ncar = nc when cl;
tel
```

Table 6: Car counting handler

**Refining state variables** This allows us to design a change of variables that moves from the concrete sensor measurements to the previous abstract variables. It is shown at table 7 and merely consists of encapsulating the sensor handlers with the clock generation.

```
node refvar(cin, cout: bool)
returns(cl: bool; nin, nout: int)
let
  cl   = countmod();
  nin  = carcount(cin, cl);
  nout = carcount(cout, cl);
tel
```

Table 7: The variable refinement function

**Refining the model** Applying this change of variable to the model of section 5.2 yields the model of table 8.

Several interesting remarks can be proposed here:

- We applied the same change of variable independently to both the controller and the whole model. The reason is that we want to carefully distinguish between what serves to model the real world and the computations that take place in the controller.

- As a consequence, in the model, we come along with two versions of the clock, the one which is computed in the model, and which features the physical time, and the one which is computed in the controller and which features the computer clock. In our formal model, both are computed exactly the same, and thus are equal. In this sense, the `assert` clause on their being equal is useless. However, it is here to recall us that models and real-world are not the same and that the correctness

```
node controller1(cin, cout: bool)
returns(cl, prop: bool);
var nin, nout: int;
let
  cl,  nin,  nout = refvar(cin, cout);
  prop = controller(nin, nout);
tel

node controlled_island1()
returns(n : int);
refines controlled_island;
var cin, cout: bool;
    nin, nout: int;
    cl, clc  : bool;
    prop     : bool;
let
  cl,  nin,  nout = refvar(cin, cout);
  clc,  prop = controller1(cin,  cout);
  assert (cl = clc);
  assert prop;
  n = island(nin, nout);
tel
```

Table 8: The refined model

of our solution heavily relies on the ability of the computer clock to measure physical time. Though this is another subject, it shows that some provisions are to be taken for checking that this property holds in operational situations, for instance by providing the computer with a fault-tolerant clock.

Finally the question of this refinement correctness is obvious because a change of variable trivially preserves the system behaviour.

### 5.4 A Deterministic Controller

Yet, the controller at table 8 is not deterministic and we need to refine it. This is done by:

- Introducing a traffic light at the island entrance and forbidding by law cars to cross the red light. The `light` node is used to model this law.
- Introducing a light controller which is in charge of ensuring the island property. This is the creative part of the control design and it is based on the principles of Model Predictive Control:

  The point here is that the light controller cannot know the number of cars that will enter the island if it sets the green light, nor can it know the number of cars that will exit the island meanwhile. Thus, it takes a conservative policy which amounts to saying: if, based on my current knowledge and on the worst traffic prediction consisting of maximising the entrance traffic and minimising the exit traffic, the total predicted number of cars exceeds the limit, I must set the red light.

The resulting refined controller is displayed at table 9. This table also illustrates some subtleties of Lustre clock calculus [18]: the `lightcontrol` node which computes the light works on the `cl` clock while the `light` node which uses the light is not sampled. We thus need to adapt the rates between them, which is done by a `current` operator (used in its three-parameter version for better readability). Note that, in the controller, `red` has to be conservatively initialised to `true`.

```
node light(cin, red: bool)
returns(prop: bool);
let
  prop = (red => not cin);
tel

node lightcontrol(nin, nout: int)
returns(red: bool);
let
  red = (ninit -> pre island(nin, nout))
        + nm > nmax;
tel

node controller2(cin, cout: bool)
returns(cl, prop: bool);
refines controller1;
var nin, nout: int;
    red: bool;
let
  cl,  nin,  nout = refvar(cin, cout);
  red = current(true, cl,
                lightcontrol(nin, nout));
  assert light(cin, red);
  prop = true when cl;
tel
```

Table 9: The deterministic controller

In the same way, the output `prop` of `controller1` being on the `cl` clock, we need to provide `controller2` with a (fake) output on the same clock.

It remains now to show that this is a correct refinement. Flush constructs the proof node displayed at table 11. It can be remarked that the proof of this node is the only difficult part in the system development as it involves some linear arithmetic. In order to make it easier, we define useful properties of the `nm` constant we have introduced. This is done by a pre-condition node. Also, we can show here useful properties of our change of variable, namely that counting `nm` booleans cannot yield a sum larger than `nm`. This can be done by adding a post-condition node. Then Flush will automatically generate a proof node for this property (which is omitted here). Table 10 shows these added nodes.

**Remark:** We have chosen to use this particular example for several reasons: it allowed us to show all interesting points of our calculus on a reasonably small system and yet, the system remained realistic. On the other hand, we did not aim at showing that "we could do what B does": as we explained in the introduction, the purpose of our Lustre-oriented calculus is to provide a refinement *adapted to* and *expressed in* Lustre/Scade. We do not claim being as powerful as B (which would be untrue), we claim that our calculus is expressive enough for problems usually treated in Lustre, so that Lustre/Scade users can use the calculus without (extensive) training.

```
node controller1_pre(cin, cout: bool)
returns(prop: bool)
let
  prop = (0 < nm) and (nm <= nmax);
tel

node controller2_post(cin, cout: bool)
returns(prop: bool)
var cl: bool;
    nin, nout: int;
let
  cl,  nin,  nout = refvar(cin, cout);
  prop = current(true, cl,
                 (0 <= nin) and (nin <= nm) and
                 (0 <= nout) and (nout <= nm));
tel
```

Table 10: Local pre and post conditions

```
node controller1_2_proof(cin, cout: bool)
returns(prop: bool);
var nin, nout: int;
    red      : bool;
    cl, pr   : bool;
    nin2, nout2: int;
    cl2,  pr2  : bool;
let
  assert controller1_pre(cin, cout);
  cl,  nin,  nout = refvar(cin, cout);
  red = current(true, cl,
                lightcontrol(nin, nout));
  assert light(cin, red);
  pr  = true when cl;
  cl2,  nin2,  nout2 = refvar(cin, cout);
  pr2 = controller(nin2, nout2);
  prop = (cl2 = cl)     and
         current(true, cl,
                 (nin2 = nin)   and
                 (nout2 = nout) and
                 (pr2 = pr));
tel
```

Table 11: The refinement proof obligation

# 6 Conclusion

In this report, we presented a temporal refinement calculus for the programming language Lustre. The calculus was derived from a general-purpose refinement calculus by taking into account the specificities of Lustre, which resulted into a series of restrictions and simplifications. A summary is available in section 4.3. After that theoretical part, we illustrated the calculus on an extensive example, which shows how to develop a system by successive refinements on top of Lustre/Scade.

To sum up the approach we propose, let us characterise it by the following points:

- The equational declarative nature of the Lustre language makes it simple to use and alleviates the burden of managing names and scopes. In our framework, the only communication mechanism between components is the function call and it contrasts heavily with, for instance, what happens in B where several communication mechanisms are needed (e.g., uses, imports, sees,..) whose necessity seem to derive from the imperative, side-effect prone nature of the B language.
- The refinement calculus is sound – in the sense that it preserves the correctness and the non-reactivity of programs – and transitive. Thus, we can use it in a step-wise manner. The restrictions we imposed on the calculus ensure that we can effectively carry out the refinement proofs and that the refined system is still a Lustre one: in particular, we preserve the synchrony of the system.
- The control theory point of view we have adopted here contrasts with the more computer science orientation usually taken in this matter. In our opinion, this point of view provides a clearer and more convincing modelling in what concerns possible model bias. It should be noted, however, that this point is less tool- and language-dependent: the same point of view could have been also adopted in B.

However, it is also worth noticing that our approach is, at present, less powerful than usual ones with respect to logic capabilities. In particular, Lustre data-types are very restricted and there is no notion of recursive functions, though there has been attempts to overcome these limitations [13]. On the one hand, it can be thought that owing to the particular nature of control systems, these limitations may not be too restrictive. On the other hand, it still could be interesting to enhance the capabilities and scope of our approach. This can be a topic for future work.

## References


[1] M. Buhlmann. A. Krüger, D. Kant. Software development process and software-components for x-by-wire systems. In *SAE WorldCongress*, Detroit, MI, USA, March 2004. 1.2

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. 1.1, 4.1

[3] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1995. 1.1, 3.1, 3.2, 6

[4] J.R. Abrial. B: A formalism for complete correct system development. Conference given at Inria Rhône-Alpes, October 1999. 5

[5] R. J. R. Back and J. von Wright. Refinement calculus: Part I and II. In *Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, pages 42–63 and 67–93. Springer-Verlag New York, Inc., 1990. 1.1

[6] P. Behm, P. Desforges, and J.M. Meynadier. Météor : An industrial success in formal development. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*. Springer, 1998. 1.2

[7] J.L. Bergerand and E. Pilaud. SAGA; a software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988. 1.2

[8] D. Bert. Building Lustre synchronous control systems from B abstract machines: A case study. Technical report, LSR-IMAG, 1997. 1

[9] J. P. Bowen, A. Fett, and M. G. Hinchey, editors. *ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, 24–26 September 1998*, volume 1493 of *lncs*. Springer-Verlag, 1998. 1.1

[10] D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December 1994. ERA Technology. 1.2

[11] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992. 2.5

[12] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to Scade/lustre to TTA: A layered approach for distributed embedded applications. In *Languages, Compilers and Tools for Embedded Systems, LCTES 2003*, San Diego, June 2003. ACM-SIGPLAN. 1.2

[13] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Int. Conf. on Functional Programming*. ACM SIGPLAN, Philadelphia May 1996. 2.4, 6

[14] J.-L. Colaco and M. Pouzet. Type-based initialisation analysis of a synchronous data-flow language. In F.Maraninchi, editor, *SLAP02*, volume 65.5 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science B. V., April 2002. 2.2

[15] C. Dumas and P. Caspi. A PVS proof obligation generator for Lustre programs. In *7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, 2000. 1.2

[16] C. Dumas and P. Caspi. A PVS proof obligation generator for Lustre programs. In *7th International Conference on Logic for Programming and Automated Reasoning*, volume 1955 of *Lecture Notes in Artificial Intelligence*, 2000. 1.3

[17] Cécile Dumas. Méthodes déductives pour la preuve de programmes lustre. Thèse de doctorat de l'université Joseph Fourier, octobre 2000. 1

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 1.2, 5.4

[19] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag. 1.2, 1.3, 5.1

[20] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. 1.1

[21] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. rapport technique 204, INRIA, Aot 1997. Version révisée distribuée avec Coq. 1.1

[22] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, Venezia (Italy), September 1999. 1.2, 1.3

[23] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990. 1.1

[24] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. 1.1, 4.1

[25] G. LeGoff. Using synchronous languages for interlocking. In *First International Conference on Computer Application in Transportation Systems*, 1996. 1.2

[26] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL : a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986. 2.4, 4.1

[27] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 1.1

[28] Jan Mikáč. Raffinement pour Lustre. rapport de DEA, VERIMAG, 2002. unpublished. 1.3, 3.2

[29] P. Caspi S. Tripakis N. Scaife, C. Sofronis and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In G. Buttazzo, editor, *4th International Conference on Embedded Software, EMSOFT04*. ACM, 2004. 1.2

[30] D. Nowak, J.R. Beauvais, and J.P. Talpin. Co-inductive axiomatization of a synchronous language. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 387–399. Springer Verlag, October 1998. http://www.irisa.fr/prive/nowak/publis/tphols98.ps.gz. 4.1

[31] S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In *11th Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Verlag, 1992. 1.1, 1.2, 1.3

[32] M. Sheeran and G. Stlmarck. A tutorial on Stlmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000. 1.2