Temporal Refinement for Lustre

Jan Mikac, Paul Caspi

Temporal Refinement for Lustre

Jan Mikáč Paul Caspi

{Jan.Mikac|Paul.Caspi}@imag.fr

Abstract

This paper proposes a refinement calculus 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. We illustrate its use on a small example and conclude by proposing some future research perspectives.

Key words: Lustre, refinement, temporal refinements

1 Introduction

In the domain of critical systems, program correctness proofs are important, and often mandatory. Generally, both the system and the 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[13] or SMV[18].) In other cases, a theorem-prover (such as PVS[21] or Coq[14]) 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, the correct-by-construction programming was proposed: the two main ideas are: - the program correctness is maintained from the specification to the final product, - the difficulty to prove 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

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

ultimate program) is guaranteed to meet the initial specifications. Some well-known refinement-based development methods are B[2], Z[5] and VDM[15]; a more theoretical framework for refinement is given for instance by TLA[16,1] or [3].

Lustre[12] is a functional synchronous dataflow language whose industrial variant Scade is used in many critical applications, such as avionic control systems. It would thus be interesting to define a refinement framework for Lustre.

Unfortunately, using B or Z for Lustre turns out to be cumbersome [1] : some Lustre features, such as the absence of recursive computations or bounded memory are possibly lost by plunging 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 Lustre source.

In order to preserve the specificity (and simplicity) of Lustre, we have therefore decided to propose a refinement calculus 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 (the notion of "synchronous observer"[11]), so that Lustre provides its own formal framework. Furthermore, there exist a model-checker for Lustre (Lesar) and a property prover (Gloups[9]). We base this paper on a previous proposal[19], which is here (largely) generalized.

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 "customizing" the previous calculus to the needs of Lustre and to our particular practical needs. This allows us to define a computer-aided both functional and temporal refinement calculus. Section 5 proposes an example of application and section 6 concludes and offers some perspectives.

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^ω 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.

[1] A translation of Lustre in B is proposed in [10]; for the inverse way see for instance [4].

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

$$p : (D_1^{\infty}, \ldots, D_n^{\infty}) \longrightarrow (D_1'^{\infty}, \ldots, D_m'^{\infty})$$

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 $\mathbf{x}$ and $\mathbf{y}$:

$\mathbf{x}$	x_0	x_1	x_2	$\ldots$
$\mathbf{y}$	y_0	y_1	y_2	$\ldots$
$\mathbf{x+y}$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$\ldots$

The sum operator $+$is applied point-wise on the two flows: the synchrony amounts to the fact that an element of the $\mathbf{x+y}$ flow exists if and only if the corresponding elements of the $\mathbf{x}$ and $\mathbf{y}$ 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 ε 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.

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

$$if\ (t.\mathbf{c})\ then\ (x.\mathbf{x})\ else\ (y.\mathbf{y}) = x.(if\ \mathbf{c}\ then\ \mathbf{x}\ else\ \mathbf{y})$$

$$if\ (f.\mathbf{c})\ then\ (x.\mathbf{x})\ else\ (y.\mathbf{y}) = y.(if\ \mathbf{c}\ then\ \mathbf{x}\ else\ \mathbf{y})$$

$$if\ \varepsilon\ then\ \varepsilon\ else\ \varepsilon = \varepsilon$$

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

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

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[8].

Finally, there are operators for sampling (`when`) and holding (`current`) flows. Sampling applies to any flow $\mathbf{x}$ and a boolean flow $\mathbf{c}$ 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 whenc			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**:

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

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

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] This condition is checked during the clock calculus.

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` memorizes 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^∞. 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	...

Formally, the semantics of a Lustre program (node) is the least fixpoint associated with the system of equations. The existence and unicity 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):

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

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 [17] and Lucid Synchrone [7]. However, section 4 will propose a means to overcome this limitation.

2.5 Lustre Mappings

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

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

It can be shown[6] 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 formalize the refinement problem and give some general results about it.

3.1 Systems

We generalize 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

[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 generalizations 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 generalizations of programs with n inputs.

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

$$T \xrightarrow{\quad S \quad} T'$$

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* – S with domain restricted to P. This is comparable to operations in B[2]: 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.

$$P : \quad T \xrightarrow{\quad S \quad} T' \quad : Q$$

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 fulfills 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 $\sigma : T \longrightarrow U$

[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.

and $\tau : U' \longrightarrow T'$ which ensure the corresponding changes of variables.

$$\begin{array}{lccc} P: & T & \xrightarrow{\quad S \quad} & T' \quad : Q \\ & \sigma \downarrow & & \uparrow \tau \\ P': & U & \xrightarrow{\quad S' \quad} & U' \quad : Q' \end{array}$$

Then, inspired by sufficient conditions given in [2] and by our previous work[19], we define the following three proof obligations:

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

$$[\mathrm{REA}_{S \to S'}] \ \forall x \in T.\ P(x) \Rightarrow P'(\sigma(x))$$

$$[\mathrm{COR}_{S'}] \quad \forall (x, y) \in U \times U'.\ P'(x) \wedge S'(x, y) \Rightarrow Q'(x, y)$$

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:

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

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

The correctness preservation property is clearly what we expected: if a system S' verifies $[\mathrm{COR}_{S \to S'}]$ 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.

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

[6] Think of feasibility in B[2]: 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.

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 $\phi : V' \longrightarrow U'$,

$$\begin{array}{lccc} P: & T & \xrightarrow{\quad S \quad} & T' \quad : Q \\ & \sigma\downarrow & & \uparrow\tau \\ P': & U & \xrightarrow{\quad S' \quad} & U' \quad : Q' \\ & \rho\downarrow & & \uparrow\phi \\ P'': & V & \xrightarrow{\quad S'' \quad} & V' \quad : Q'' \end{array}$$

the precise transitivity lemmas are

$$\frac{[\mathrm{COR_S}] \wedge [\mathrm{COR_{S \to S'}}] \wedge [\mathrm{REA_{S \to S'}}] \wedge [\mathrm{COR_{S' \to S''}}]}{[\mathrm{COR_{\tau \circ \phi \circ S'' \circ \rho \circ \sigma}}]}$$

$$\frac{[\mathrm{REA_{S''}}] \wedge [\mathrm{COR_{S \to S'}}] \wedge [\mathrm{REA_{S \to S'}}] \wedge [\mathrm{COR_{S' \to S''}}] \wedge [\mathrm{REA_{S' \to S''}}]}{[\mathrm{REA_S}]}$$

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

if $S_1 \sqsubseteq S_2$ and $S_2 \sqsubseteq S_3$ and $\ldots$ and $S_{n-1} \sqsubseteq S_n$

then $S_1 \sqsubseteq S_2 \sqsubseteq S_3 \sqsubseteq \ldots \sqsubseteq S_n$ 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 "customize" 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[16,1]: if the state of the abstract system does not change ("the system is stuttering"), then any behavior 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[17,20]. 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

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

The clocks in the above expression are

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

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 α; yet, as ck is on α ($ck :: \alpha$), so that the output of clk runs faster than its input.

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

$$\sigma \ : \ T \longrightarrow Bool \times T$$
$$\sigma(x) = clk(x), x$$

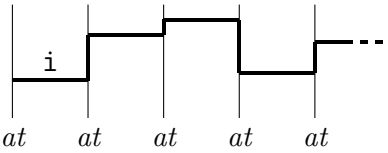
This change of variables leaves the input x unchanged, it provides only a boolean flow ck which runs faster than x, which amounts to making x stutter.

Example

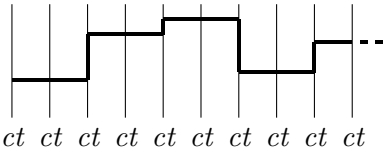
```
node Plus2(i:int)
returns (s:int)
let
  s = i + 2 ;
tel
```

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 Real-Time Temporal Refinement

In real-time applications such a general temporal refinement may not be well adapted because the *clk* function is data-dependent and we do not know in advance the amount of time acceleration it provides. A much safer situation occurs when the clock does not depend on the inputs, as in the `Plus2` example. This is what we call a *Real-Time Temporal Refinement*.

4.4 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

(i) $\sigma : T \longrightarrow T'$ is a Lustre mapping

(ii) $\tau : U' \longrightarrow U$ is a Lustre mapping

(iii) $f : Bool \times T' \longrightarrow Bool$ is a clock-preserving Lustre mapping

(iv) $Ck\_P'(ck', x') = (ck' = true \rightarrow pre(f(ck', current(@, ck', x'))))$

(v) $P'(ck', x') = Ck\_P'(ck', x') \wedge D\_P'(x')$

Proof Obligations

(vi) $[\mathrm{COR}_{S \rightarrow S'}]$
$\forall (x, y') \in T \times U'.$
$P(x) \wedge Ck\_P'(ck', \sigma(x)) \wedge S'((ck', \sigma(x)), y') \Rightarrow S(x, \tau(y'))$

(vii) $[\mathrm{REA}_{S \rightarrow S'}]$
$\forall x \in T.\ P(x) \Rightarrow D\_P'(\sigma(x))$

(viii) $[\mathrm{COR}_{S'}]$
$\forall (x', y') \in U \times U'.\ P'(x') \wedge S'(x', y') \Rightarrow Q'(x', y')$

We can notice that:

- In the proof obligation (vii) 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 (v). We see here that pre-conditions are split into two parts, a clock pre-condition Ck_P and a data pre-condition D_P'.

5 Example

In this section, we illustrate the practical application of the previous results on the example of a *modulo n counter* .

5.1 Specifications

Informally, we want to derive -by refinement(s)- a system which would receive two inputs – a boolean R (for "reset") and an integer n – and would output an integer c which grows from 0 to $n-1$ and then starts at 0 again. The R input is designed to force a reinitialization of the counter at any time. Finally, the n input is not supposed to evolve during the normal computations; n may change only when R occurs.

Formally, we can split this specification into three parts:

The pre-condition

```
node Counter1_pre(R: bool; n: int when R)
returns (prec: bool)
let
  prec = (current n) > 0 ;
tel
```

which says that the input n is always positive. We see here that safety predicates are coded in Lustre thanks to observer nodes [11] computing the truth value of the predicate.

The post-condition

```
node Counter1_post(R: bool; (n: int) when R; c: int)
returns (post: bool)
let
  post = (c = 0) ->
         ((R => (c = 0)) and
         ((not R) => ((c = (pre c) + 1) or
                      ((c = 0) and pre (c = current n - 1)))));
tel
```

which describes the desired behaviour of the system.

The system itself

```
node Counter1(R: bool; (n: int) when R; o: int)
returns (c :int)
let
  c = o;
  assert Counter1_post(R, n, o);
tel
```

where `assert` is the Lustre construct allowing the restriction of inputs. The clause `assert p;` means any input combination that makes the property `p` always true.

This allows us to define a relational node by adding an extra input that stands as an oracle for the output and by constraining this input.

Since we are in the early stages of the refinement process, we do not write actual Lustre programs and we can admit a lighter formulation which gets rid of this extra input:

```
node Counter1(R: bool; (n: int) when R)
returns (c :int)
let
  assert Counter1_post(R, n, c);
tel
```

We have specified the behavior of `Counter1` by the post-condition: in this way, the node is automatically correct with respect to its post-condition.

One may wonder what would happen if we added another formula to the node body – a formula which would be inconsistent with `Counter_post`. In such a case, `Counter1` would be *correct* with respect to `Counter_post`, but *not reactive*, so that we would not be able to refine it into an actual Lustre program.

5.2 First Refinement

There is obviously a very simple implementation of a counter modulo n:

```
node Counter2(R:bool; (n:int) when R)
returns (c:int);
refines Counter1;
let
  c = 0 -> if R then 0
           else if pre (c = current(n) - 1))
                then 0
                else (pre c) + 1 ;
tel
```

We do not specify the pre and post conditions of this node: this means that `Counter2` copies the corresponding properties from `Counter1`.

Let us remark that we do not refine time. σ and τ functions are identities and $Ck\_P'(ck', \sigma(x)) = true$. Therefore, the refinement obligations become simpler:

$[\mathrm{COR}_{1\to 2}]$ $\forall (x, y) \in T \times T'. P_1(x) \wedge S_2(x, y) \Rightarrow S_1(x, y)$

$[\mathrm{REA}_{1\to 2}]$ $\forall x \in T.\ P_1(x) \Rightarrow P_1(x)$ because $P_2 = P_1$!

$[\mathrm{COR}_2]$ $\forall x, y \in T \times T'.\ P_1(x) \wedge S_2(x, y) \Rightarrow true$ since $Q_2 = true$

The last two proof obligations are obvious, thus the only remaining proof obligation is the first one, which can be expressed as a special Lustre node `Refinement1` whose only output is true if and only if the proof obligation holds.

```
node Refinement1(R: bool; (n: int) when R; c: int)
returns (cor: bool)
var prec, s1, s2: bool;
let
  prec = (current n) > 0 ;
  s2 =  (c = (0 -> if R then 0
                  else if pre (c = (current(n) - 1))
                       then 0
                       else (pre c) + 1)) ;
  s1 = (c=0) -> ((R => (c = 0)) and
                ((not R) => ((c = (pre c) + 1) or
                             ((c = 0) and
                               pre (c = current n - 1)))));
  cor = (prec and s2 => s1) ;
tel
```

This node takes as inputs all flows of `Counter1`. The idea here is that the inputs are not constrained, so that a proof of the `cor` output will be universally quantified over all inputs, as stated in $[\mathrm{COR}_{1\to 2}]$.

The `Refinement1` node uses three local boolean variables (`prec`, `s1` and `s2`). They are defined by copying the corresponding definitions from `Counter1` and `Counter2`.

By using our Lustre-specific invariant prover Gloups[9], we can effectively prove that `cor` is always true. [7] Thus the node `Counter2` refines `Counter1`, and in particular meets the post-condition Q. As `Counter2` is an implementable Lustre node, we could stop here: we have an effective (and reactive) implementation.

5.3 Second Refinement

In order to illustrate other features of our calculus, we will carry out one more refinement step. In fact, as Lustre/Scade are also used for describing hardware circuits, let us imagine that we need to implement the counter as a circuit.

The `Counter2` node is not well adapted for this purpose: `Counter2` suggests to compute the value of `n-1` at each clock cycle, then to compare it with the previous value of `c` and possibly increment `c`. This makes three arithmetical operations (`n-1`,the comparison,`c+1`) at each cycle: since each operation

[7] Gloups basically transforms a proof of a property on flows into a proof of a scalar property by using induction. Then it discharges resulting proof obligations into PVS and lets the user prove them.

requires an ALU,[8] we need three ALUs to carry out the calculations in one cycle. This is too expensive: we could perhaps spare some ALUs by running faster and by performing the arithmetical operations successively on the same ALU. Therefore, we propose the following refinement, which is supposed to run twice as fast as `Counter2`:

The clock pre-condition

```
node Counter3_ck_pre(cl: bool; R: bool when cl; n: int when R)
returns (ck_prec: bool);
let
  ck_prec = (cl = (true-> pre not cl));
tel
```

The function

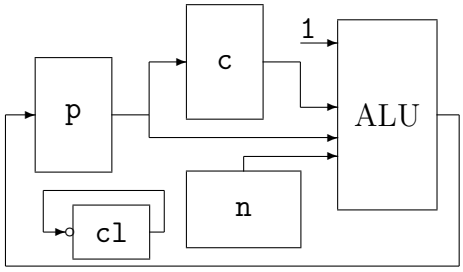
```
node Counter3(cl: bool; R: bool when cl; n: int when R)
returns (clp: bool; c: int when clp);
refines Counter2 by factor 2;
var p: int when cl;
let
  clp = not cl;
  p = 0 -> if R then 0
           else pre (((current c) when cl) + 1) ;
  c = 0 ->
      if ((current R) when clp) then 0
      else if (current(p) = current(n)) when clp
           then 0
           else (current p) when clp ;
tel;
```

`Counter3` introduces new variables: boolean `cl, clp` and an integer `p`. The boolean `cl` is in fact a clock that has a period twice as long as the basic clock of the node. It is intended to define two different phases:

- In the first phase (when `cl` is true), we add 1 to the previous value of `c` and the result is stored into `p`. During this phase, the value of `c` remains unchanged.
- In the second phase (`clp` is true), the value of `p` is compared to `n` and the value of `c` is updated accordingly to the result of the comparison.

In this way, we can use only one ALU: only one arithmetical operation takes place in each phase. As we have supposed that `Counter3` runs two times faster than `Counter2`, the result is present in `c` on time (at the end of the second phase). The following figure can be a hint to better understand what we intended (only the data paths are featured – the control part is left out):

[8] Arithmetic-logic unit



For the same reasons as in the case of `Counter2`, we do not prove the reactivity and local correctness of `Counter3`. The remaining proof obligation $[\mathrm{COR}_{2\to 3}]$ can be written as

$$\forall (x, y, x', y') \in T \times T' \times U \times U'.$$
$$P(x) \wedge x' = \sigma(x) \wedge Ck\_P'(ck', x') \wedge S'((ck', x'), y') \wedge y = \tau(y') \Rightarrow S(x, y)$$

σ and τ are identity mappings on the values, but the basic clocks of `Counter2` and `Counter3` are different, so that the proof node is more complicated:

```
node Refinement2(cl, clp: bool;
   R1:bool when cl; n1:int when R1; c1:int when cl;
   R2:bool when cl; n2:int when R2; p2:int when cl;
   c2:int when clp)
returns (cor: bool)
var prec, s2, tau: bool;
    s1: bool when cl ;
let
  prec = (current (current n)) > 0 ;
  sigma = (R2 = R1) and (current(n2) = current(n1)) ;
  tau = (current(c1) = current(c2)) ;
  s1 = (c1 = (0 -> if R1 then 0
                  else if pre (c1 = (current(n1) - 1))
                       then 0 else (pre c1) + 1)) ;
  s2 = current(p2 = 0 -> if R2 then 0
                         else pre (((current c2) when cl) + 1))
       and current(c2 = 0 ->
         if ((current R2) when clp) then 0
         else if (current(p2) = current(current(n2))) when clp
              then 0
              else (current p2) when clp) ;
  tau = clp => (current(c1) = current(c2)) ;
  assert (cl = (true-> pre not cl)) ;
  assert (clp = not cl) ;
  cor = (prec and s2 and tau and clp => current(s1)) ;
tel
```

- The inputs of `Refinement2` contain all inputs of `Counter2`, as well as those of `Counter3`. This is necessary, since the basic clocks of the two nodes are not equal.
- The correspondence between the abstract and concrete inputs is given in `sigma`, while `tau` describes the link between the outputs.
- As in `Refinement1`, `prec` gives the pre-condition, `s1` the behaviour of the abstract system and `s2` that of the concrete one.
- The assertions constrain the clocks (`cl` and `clp`) to the expected behaviour, as given in `Counter3_ck_pre`.

Once again, Gloups helps us to prove[9] that `cor` holds. Thus `Counter3` *is* a refinement of `Counter2` (and by transitivity of `Counter1`).

We showed in this example how we intend to apply our refinement calculus on actual Lustre programs – we illustrated both functional and temporal refinement and proved them correct. Although the proof obligations were written by hand, we think that the overall algorithm is clear. As a by-product, we also illustrated the power of Lustre to express properties over Lustre programs.

6 Conclusion

This article presented a temporal refinement calculus for the programming language Lustre. We derived it from a general-purpose refinement calculus by taking into account some specificities of Lustre and our actual proof possibilities in as series of simplifications and restrictions based on some sufficient condition.

The resulting calculus (summarized in section 4.4) is sound – in the sense that it preserves the correctness and the non-reactivity of the 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.

We illustrated the refinement on the example of a modulo n counter, for which we derived a hardware-implementable program from a formal specification. All the proof obligations could be proved.

A tool is currently under construction (implemented as a front-end for Gloups) which allows deriving the proof obligations in an automated way. Such a tool will allow us to further investigate the possibilities offered by our refinement calculus by handling more complex case studies.

[9] The actual proof node that is given to Gloups is somewhat different from `Refinement2`: by some rewritings, we get rid of the current-when operators, which introduce a lot of complexity into Gloups proofs. However, we preferred to present the original node here, as the rewritten one is not easily readable.

References

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

[2] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1995.

[3] 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.

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

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

[14] 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.

[15] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

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

[17] 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.

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

[19] Jan Mikáč. Raffinement pour Lustre. rapport de DEA, VERIMAG, 2002. unpublished.

[20] 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.

[21] Sam Owre Natarajan Shankar John Rushby D.W.J Stringer-Calvert. PVS language reference. Technical report, SRI International, December 2001.