Flush : A System Development Tool Based on Scade/Lustre

Jan Mikac, Paul Caspi

Flush : A System Development Tool Based on Scade/Lustre

Jan Mikáč
Jan.Mikac@imag.fr

Paul Caspi
Paul.Caspi@imag.fr

Laboratoire Verimag (CNRS, UJF, INPG)
2, avenue de Vignate
France – 38610 Gières

ABSTRACT

In safety-critical control systems, the Scade/Lustre development environment has proved its value, with notable achievements such as the Hong-Kong subway signalling system and Airbus A380 flight controls. The interest of the approach comes from the synchronous data-flow style of the Lustre language which makes is well-adapted to the culture of control engineers. At the same time Lustre is endowed with simple formal semantics which makes it amenable to formal development.

The currently running Flush project consists of building a formal system development tool on top of it, by taking advantage of the formal properties of the Lustre language. To this end, a refinement calculus is defined, encompassing both functional and temporal aspects. Refinement proof obligations are generated, and several proof approaches can be used to discharge them: model-checking, abstract interpretation, and theorem proving through repeated induction and, finally translation to PVS proof obligations. The resulting methodology is illustrated on the island example used by J.R. Abrial for presenting the B system method.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Tools*; D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms

Design, Verification

Keywords

synchronous language, refinement, temporal refinement

FMICS'05, September 5–6, 2005, Lisbon, Portugal.

1. INTRODUCTION

Computerised control is one of the computer domains that yields the most safety-critical applications and in which 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 (Vdm[15], Z[21], B[2]) 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 [4].

On the contrary, the Lustre/Scade approach was tailored, from the very beginning [11] to this application domain and has shown many successes in safety critical control, for instance Framatome nuclear plant emergency shut down [5], the Hong-Kong subway signalling system [16], the Airbus fly-by-wire systems [6], 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 [18] and to popular control architectures [7],
- 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, plug-in for Prover [20] and other model-checkers [12], tools for abstract interpretation [13] and an interface [10] to PVS [19], the well-known theorem-proving assistant.

Yet, at present, these tools were not integrated into a complete development framework like the one cited above. This is the object of the currently running Flush[1] project. In this project, a refinement calculus has been defined, inspired from both the one of B [2] and the one of TLA [17] which encompasses both functional and temporal refinement [14]. This paper describes the implementation we have recently performed of this calculus. In doing this, we shall follow the "island example" [3] proposed by J.R. Abrial for illustrating the development of systems in B. This example will allow us to informally describe the Lustre language, the way we use

[1] An acronym for Formal Lustre Helper.

it to model non deterministic programs, the way we refine these programs so as to obtain deterministic controllers and, finally, the way we handle temporal refinement.

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.

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 **x** and **y**:

x	x_0	x_1	x_2	...
y	y_0	y_1	y_2	...
x+y	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	...

The sum operator + is applied point-wise on the two flows: the synchrony resides in 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.

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.

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

x	x_0	x_1	x_2	...
pre x	@	x_0	x_1	...
x -> pre x	x_0	x_0	x_1	...
x fby x	x_0	x_0	x_1	...

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[9].

Finally, there are operators for sampling (`when`) and holding (`hold`).[2] Both of them apply to any flow **x** and another boolean flow **c** which is called clock. Sampling is the more intuitive operation: 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, as illustrated in table 1. More formal definitions can be found in [11, 10, 14].

Table 1: Sampling and holding in Lustre

x	x_0	x_1	x_2	x_3	x_4	x_5	...
c	f	f	t	f	t	t	...
y = x when c			x_2		x_4	x_5	...
hold(u, c, y)	u_0	u_0	x_2	x_2	x_4	x_5	...

2.3 Lustre Nodes

Lustre programs are called "nodes". A node defines a function over flows: it is made of declarations for inputs, outputs and possibly local typed flows, and of equations relating these flows to each other, as in the following example:

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

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	...

Finally, Lustre allows the definition of *assertions*: The statement

```
assert e;
```

where `e` is a boolean expression, means that the value of `e` is constrained, by assumption, to take always the value `true`. We take advantage of this feature in Lustre to define relations between flows in a functional manner: Given a node

```
node N(x...; y...) returns (prop :bool);
let ... tel
```

the assertion

```
assert N(x, y);
```

defines a relation between flows `x` and `y`.

3. THE "ISLAND" MODEL

Having shortly presented Lustre, we can move to the example.

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`.

3.1 A first specification

The formal translation in Lustre of this informal specification is shown in table 2.

Besides constant declarations, the specification is made of two components, the `island` model and the required property which is considered as a post-condition of the former. Both are modelled as Lustre nodes. For instance, the equation of the "island" node says that the value of the number

[2] Actually `hold` is not a Lustre primitive but we use it for the sake of simplicity. Its Lustre definition is:

```
node hold(u:type; cl:bool; x:type when cl)
returns (y:type)
let
  y = if cl then current x
      else (u -> pre y);
tel
```

where `type` is any Lustre type.

Table 2: The island initial specification

```
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
```

of cars `n` is *always, i.e., at any time* equal to either the initial or the preceding number of cars in the island plus the number of cars that enter minus the number of cars that exit at that time. Similarly, the post-condition computes the truth value of the property we want the island to fulfill. This means that we want the boolean flow `prop` to always yield the value `true`. This amounts to the synchronous observer technique promoted in [12].

As we can see, the Lustre style is akin to a temporal logic of the past, where, furthermore, every object, be it a proposition or a state is a flow, i.e., a sequence.

3.2 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 3) stating some useful assumptions, such as the fact that the initial value `ninit` is smaller than the required maximum value `nmax`. Please note that whereas post-conditions bear on both inputs and outputs of the `island` node, pre-conditions are allowed to bear only on inputs.

Table 3: Useful pre-conditions

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

3.3 Trying to prove it

In 3.2, we said that we tried to prove the property but we did not say how. Let us see now how it is done in the Flush tool: it consists of automatically building a proof Lustre node, whose only output is the truth value of the property we want to prove (table 4). Then this node can be dispatched to any convenient proof tool, such as the ones cited in introduction.

Table 4: The proof node

```
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
```

This proof node illustrates also some other features of Lustre:

- the ability to impose assumptions to the behaviour of a node thanks to the `assert` clause which says that its argument, here the flow corresponding to the expression `island_pre(nin, nout)`, is always true, and
- the functional node instantiation, like in the expression `island(nin, nout)`. Please note here that this is a very powerful mechanism because a node defines a function over flows and functions over flows can be dynamic systems encapsulating memories (states). This is the case of the `island` node because of the `pre n` construction which makes `n` a *state variable*. In this sense, Lustre nodes can be seen as as powerful constructs as, for instance, B machines.

Yet, clearly enough, the proof attempt fails, because the `island_pre` model does not prevent cars from arriving on the island at any rate. For instance `nin=nmax+1` and `nout=0` is a counter-example.

4. NON DETERMINISTIC CONTROLLER

This failure, yet, is interesting in that it shows us that our island 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 island model. 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 5. It just computes the truth value of the desired properties. Then, we can encapsulate both `island` and `controller` within a global model, the `controlled_island` (table 6) where we say, thanks to the assertion mechanism, that the truth value computed by the controller stays always true.

Table 5: A non deterministic controller

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

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 the same 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.

Table 6: The controlled island

```
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
```

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

5. TEMPORAL REFINEMENT

We thus have at this step a correct controller. Yet, it is not effective for at least two reasons: it is non deterministic and it needs sensors for measuring the flows of input and output cars. Let us address first 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.

Temporal refinement [14] in Lustre[3] is handled thanks to the Lustre sampling (`when`) and holding (`hold`) primitives. At this stage, we only need the sampler, whose behaviour is at table 1.

5.1 The sensor handlers

Given these primitives, writing sensor handlers is fairly easy. The result is displayed at table 7. At section 3.1, 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.

[3]See appendix A for a quick reference to the refinement.

Table 7: Car counting handler

```
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
```

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 in front of the sensor, 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.

5.2 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 8 and merely consists of encapsulating the sensor handlers with the clock generation.

Table 8: The variable refinement function

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

5.3 Refining the model

Applying this change of variable to the model of section 4 yields the model of table 9. However, 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 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 [14].

Table 9: The refined model

```
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
```

6. A DETERMINISTIC CONTROLLER

Yet, the controller 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 (`pre island(nin,nout)`) and on the worst traffic prediction consisting of maximising the entrance traffic (`+ nm`) and minimising the exit traffic (`- 0`), the total predicted number of cars exceeds the limit (`nmax`), I must set the red light.

Table 10: The deterministic controller

```
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 - 0 > 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 = hold(true, cl,
             lightcontrol(nin, nout));
  assert light(cin, red);
  prop = true when cl;
tel
```

The resulting refined controller is displayed at table 10. This table also illustrates some subtleties of Lustre clock calculus [11]: the `lightcontrol` node which computes the light works on the `cl` clock while the `light` node which uses the light is unsampled. We thus need to adapt the rates between them. This is the part played by the `hold` node, which, as seen in diagram 1 fills in the "holes" due to the sampling. This is done by using the last value provided by the sampled value. Now, it may be that, if the sampling clock is initially false, such a last value does not exist. This is the use of the `hold` first argument which serves as an initialisation. Note that, in the controller, `red` has to be conservatively initialised to `true`.

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 12. It can be remarked that the proof of this node is the only difficult part in the system development as it involves some linear arithmetics. 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 11 shows these added nodes.

Table 11: Local pre and post conditions

```
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 = hold(true, cl,
              (0 <= nin) and (nin <= nm) and
              (0 <= nout) and (nout <= nm));
tel
```

Table 12: The refinement proof obligation

```
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 = hold(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
         hold(true, cl,
              (nin2 = nin)   and
              (nout2 = nout) and
              (pr2 = pr));
tel
```

7. CONCLUSION

In this paper, we have shown how to build a development method on top of Scade/Lustre, a popular language in the field of safety-critical control systems. The main features of this framework are

- the equational and declarative nature of the language makes it simpler 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 control theory point of view we have adopted here which contrasts to 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 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 [8]. 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.

APPENDIX

A. REFINEMENT THEORY EXCERPT

In this section, we provide a summary of our (temporal) refinement theory, as a quick reference for the reader. For further details, please consult [14].

A.1 Systems

We model a program by a total[4] relation $S \subseteq T \times T'$, where T represents the inputs and T' represents the outputs and local variables[5]. Graphically, we will note it as

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

We add two predicates: $P \subseteq T$ called pre-condition and $Q \subseteq T \times T'$, the post-condition.

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

P restricts the domain of the inputs, so that (P, S) forms a partial relation. Q provides an abstraction of the actual computations of S: (P, Q) forms the abstract interface of the program.

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.

[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.

[5] Outputs and local variables have the same role with respect to computations, so that we will treat them in the same way. The visibility and scoping is not of interest here.

- 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.

A.2 General refinement

Given another system S'

$$P' : \quad U \overset{S'}{\longrightarrow} U' \quad : Q'$$

we want S' to refine S: we want to be able to use S' instead of S, while preserving the correctness and the reactivity of the latter. As the two systems are not necessarily typed in the same way, we suppose the existence of two (*Lustre*) mappings $\sigma : T \longrightarrow U$ and $\tau : U' \longrightarrow T'$ which ensure the corresponding changes of variables.

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

There are three proof obligations:

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

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

In this setting, we can prove that "S' refines S" means:

- S' preserves the correctness of S: especially, if Q holds for S (under P), then Q holds for S' too. Thus, the overall post-condition of S' is $Q \wedge Q'$ and it is correct to use S' instead of S.
- If S' is reactive, then S is reactive: non-reactive specifications (S) cannot be refined into reactive programs, while reactive specifications can. In fact, exhibiting an Lustre-implementable (an thus reactive) program refining some specification S proves the reactivity of the latter.
- Moreover, the refinement is transitive:

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

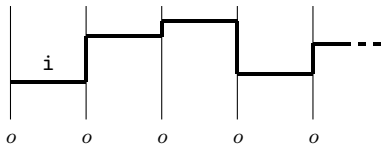
Hence, we can apply the refinement in a step-wise manner.

A.3 Temporal refinement

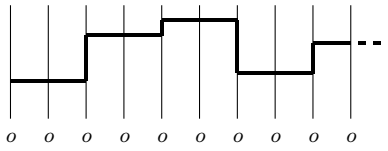
When adapting the previous general refinement setting to Lustre, we find the following additional constraints, due to the Lustre type system:

- If S' refines S, then the inputs of S' form a subset of the inputs of S. This constraint comes from the $[\mathrm{REA_{S\to S'}}]$ proof obligation, which implies a subtyping constraint between the inputs.
- Conversely, all outputs of S have to form a subset of the outputs of S'. Thus, refinement allows us to "forget" some inputs and "create" new outputs.
- There is no data refinement in Lustre, so that the types of the common inputs and outputs have to be the same in S and S'.
- The basic clock of S' may run faster than that of S, provided that every time when an output is due in S, S' provides a value and this value is a correct output of S. In the meantime, S' may output any values.

  To illustrate this point, let us suppose that the abstract system takes an input signal `i` and provides an output (o) at some rate:



  Then, a refined system may, for instance, run twice as fast:



We can summarize the temporal refinement as follows:

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. $[\mathrm{COR_{S\to S'}}]$ $\forall (x,y') \in T \times U'.$ $P(x) \wedge Ck\_P'(ck', \sigma(x)) \wedge S'((ck', \sigma(x)), y') \Rightarrow$ $S(x, \tau(y'))$
7. $[\mathrm{REA_{S\to S'}}]$ $\forall x \in T.\ P(x) \Rightarrow D\_P'(\sigma(x))$
8. $[\mathrm{COR_{S'}}]$ $\forall (x',y') \in U \times U'.\ P'(x') \wedge S'(x',y') \Rightarrow Q'(x',y')$

A.4 Practical application

In this paper, we create a closed world (the island), then we apply the above refinement to some parts of the world. We extract a controller, which we refine into an implementable controller. The rest of the world provides an environment to the controller.

B. 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.

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] J.-L. Colaço 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] J.Mikáč and P.Caspi. Temporal refinement in Lustre. In *Slap'05*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005. to appear.

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

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

[17] L.Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. and Sys.*, 16(3), 1994.

[18] 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.

[19] 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.

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

[21] J.M. Spivey. *Understanding Z: A specification language and its formal semantics*. Cambridge University Press, 1988.