Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 59130, 16 pages
doi:10.1155/2007/59130

Research Article

Array Iterators in Lustre: From a Language Extension to Its Exploitation in Validation

Lionel Morel

IRISA-INRIA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France

Received 29 June 2006; Revised 27 November 2006; Accepted 18 December 2006

Recommended by Jean-Pierre Talpin

The design of safety critical embedded systems has become a complex task, which requires both appropriate language features and efficient validation techniques. In this work, we propose the introduction of array iterators to the synchronous dataflow language Lustre as a mean to alleviate this complexity. We propose these new operators to provide Lustre programmers with a new mean for designing regular reactive systems. We study a compilation scheme that allows us to generate efficient loop imperative code from these iterators. This language aspect of our work has been fruitful since the iterators are being introduced in the industrial version of Lustre. Finally, we propose to take these regular structures into account during the validation process. This approach has already shown its applicability on different real-life case studies. The work we relate here is thus complete in the sense that our propositions at the language level are taken into account both at the compilation and the validation levels.

1. INTRODUCTION

1.1. Reactive systems and the synchronous approach

Reactive systems, as defined in [1], are characterized by the interaction with their environment being the prominent aspect of their behavior. Software embedded in aircraft, nuclear plants, and similar physical environments, is a typical example. Moreover, they interact with a noncollaborative environment, which may impose its own rhythm: it does not wait, nor reissue events. Synchronous languages [2] represent an important contribution to the programming of reactive systems. They are all based on the perfect synchrony hypothesis that establishes that communications between different components of a system are instantaneous and, more importantly, that computations performed by components are seen as instantaneous from their environment's point of view. Among these languages, the most significant ones are Esterel [3], Lustre [4], and Signal [5]. These languages offer a strong formal semantics and associated validation tools. They are now commonly used in highly critical industry for the design of control systems in avionics, nuclear power plants, and so forth.

1.2. Lustre: the language and associated verification tools

The language

In this work, we are more particularly interested in the language Lustre. It is dataflow in the sense that every variable X represents an infinite flow of values $(X_1, X_2, \ldots, X_i, \ldots)$, X_i being the value taken by X at the ith instant of the execution of the program. Classical operators (or, and, not, $+$, $-$, $*$, /, mod, $>$, $>=$, etc.) are applied pointwise on the flows. For example, the conditional expression if C then E1 else E2 (where C is a Boolean expression and E1 and E2 are 2 expressions of the same type) describes a flow X such that for all n. If C_n then $\mathsf{X}_n = \mathsf{E1}_n$ else $\mathsf{X}_n = \mathsf{E2}_n$. Here, n represents the successive instants of the execution of the system. Two operators are used to manipulate the flows directly. The pre, defines a local *memory* (for all $n > 0$, pre $(X)_n = X_{n-1}$) while the *arrow* allows to initialize a flow ($X \rightarrow Y = (X_1, Y_2, \ldots, Y_i)$). Lustre programs (*nodes*) possess input, output, and local variables (*flows*) and every output/local variable is defined by exactly one equation. The program of Figure 1 implements a simple accumulator. At the first instant, c takes the value of expression "if e then 1 else 0." Then at every instant n, c takes as value the sum of the same expression (if e then 1 else 0) to which we add the value of c at instant $n - 1$ (pre(c)).

```
node Accumulator (e : bool) returns (c : int);
let
  c= (0 -> pre c) + if e then 1 else 0;
tel
```

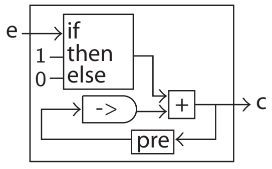


Figure 1: The Accumulator program.

```
<Initialize memories>
Always do{
  <Read inputs>
  <Compute outputs>
  <Update memories>}
```

Figure 2: Synchronous execution scheme.

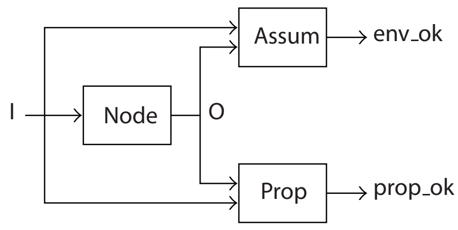


Figure 3: Validation with observers.

We have only talked about a single notion of time, induced by the sequence of values of variables. It defines a *global* clock, that can be noted as the constant true (the Boolean flow being true at each and every instant in time). For developing embedded applications, it is often necessary to describe subsystems evolving at different rhythms. In this respect, Lustre provides two operations: a sampler when and a projector current. Although we will not use these clock operations in the examples throughout this paper, all our propositions extend naturally to them. We will thus not present these operations in details here.

Compilation

A set of Lustre equations describes a network of operators, and is equivalent to the description of a combinational circuit. The same constraints apply: sets of equations with instantaneous loops are ruled out by the compiler. For example, $\{x = y + z, z = x + 1, \ldots\}$ is a set of fix-point equations that perhaps has solutions. It is however not accepted as a valid dataflow program. Lustre programs are compiled into imperative programs in C, which have the form of Figure 2, the infinite loop being classically called the *reactivity loop*.

Expressing safety properties

As Lustre is intended mainly to program safety critical systems, an important issue is the formal verification of safety properties expressed on programs. These properties are expressed through the use of synchronous observers [6]. These observers are standard Lustre nodes that take as inputs both the inputs and outputs of the program to be verified and give back one Boolean output representing the truth value of the property to prove.

Verification scheme

In general, we want to prove that a certain program P satisfies a certain safety property Prop knowing that certain hypothesis on its environment holds, described by an assertion Assume. Such an assertion can also be described by an observer, and introduced through an assert clause. The verification scheme in Figure 3 is used by Lustre verification tools, such as Lesar [6], a symbolic model checker, or nBac [7], an abstract interpreter, to show that, as long as the assertion observer outputs true, so does the property observer.

Technology transfer

The language Lustre is developed at Verimag,[1] since the mid eighties. During its history, it has always been very close to the need of embedded system designers (particularly in embedded control of critical systems). This has led to the creation of a tool called SCADE, developed now by Esterel Technologies,[2] that is actually a graphical version of Lustre. Although the evolution of both languages is independent in practice, they stay very close and, as is exemplified by the present work, Lustre often serves as an exploratory platform for SCADE.

1.3. A language extension: from language design to compilation to validation

The goal of this paper is to describe an *extension* of the Lustre language to include new operators and the consequence of such an extension on (1) the whole design process; (2) the compilation process; and (3) the verification of properties. This extension started first as a language issue, and more precisely a concern of making the language not more expressive but *easier to use* for some particular types of applications, as we will see below. Facing an increasing complexity, designers using the SCADE environment wanted somehow to have the possibility to express regular programs in a somehow natural way. Although operators for designing regular *hardware* systems existed in the language, they were not adapted to targeting software code generation. Trying to

[1] http://www-verimag.imag.fr/SYNCHRONE.
[2] http://www.esterel-technologies.com.

overcome these drawbacks led quite naturally to the introduction of new operators, called iterators, that were specifically designed to answer this particular demand from programmers. The definition of the operators themselves was also motivated by some compilation aspects: an important concern was to *introduce operators for which the generation of more efficient code is straightforward*. This whole process is reported in Section 2. It starts from motivations for the new operators and goes through the actual definition of the iterators down to compilation and optimization aspects.

The natural prolongation of this definition of a *language extension* was to be able to take these new constructs in the validation process. In Section 3, we propose a validation technique for iterative Lustre programs. More precisely, this technique is based on a slicing algorithm of the regular structures implied by the use of the iterators. From a property on a program expressed with iterations on arrays, we are able to generate smaller proof obligations expressed on elements of arrays.

An interesting aspect of this work is that the introduction of a language feature, with a first goal being to make the description of certain types of applications easier, has raised several interesting problems that concern both compilation and validation. Starting from a language request, we have tried to answer it and studied the implication of our solution on the compilation and validation processes. This makes the whole approach a good example of language design, showing how a theoretical work can be inspired by actual realistic applications and lead to a complete solution being actually applicable in practice.

1.4. Plan of the document

This paper is organized in two distinct parts. In Section 2, we study the language aspects.[3] Starting from the motivations for introducting of array iterators (see Section 2.1), we continue with the definition of their syntax and semantics (see Section 2.2) and with the study of the compilation of these operators into imperative code (see Section 2.3). Finally, we present a technique for optimizing cascades of iterations (see Section 2.4). Section 2.5 will briefly present works related to these language aspects. The second part of the paper, Section 3, studies a validation methodology that takes advantage of the regular structure introduced by the iterators. After a brief introduction, we spend few paragraphs (see Section 3.1) on the question of the form of the properties we are considering. Our proof methodology is presented in Section 3.2. Related works are then commented in Section 3.3. Finally, Section 4 concludes and gives some perspectives about this work.

2. ARRAYS AND ITERATORS: A LANGUAGE ISSUE

2.1. A long story

Arrays were first introduced in Lustre in the Ph.D. work of Rocheteau [9]. The Pollux code generator [10], resulting from this work, is devoted to the generation of synchronous circuits. The circuits produced by Pollux were to be implemented on the PAM [11], a machine developed by DEC-PRL for fast hardware prototyping, which is actually a matrix of Xilinx's programmable gate arrays. The operators proposed, that we recall now, do not increase the expressive power of the language, but they allow for more natural description of these synchronous circuits.

2.1.1. Arrays

Let τ be a type and n a constant. n is known at compile-time: for criticality reasons we do not allow array access by dynamic indexes. And n is different from 0, meaning that we do not allow empty arrays. $\tau\hat{}n$ is the type of arrays of size n and whose elements' type is τ.

The following constructors are available in the language. [0, 2, 3] represents the array with elements 0, 2, and 3. true^3 = [true, true, true]. *Slice extraction*:

$$A[i\cdots j]=\begin{cases}[A[i],A[i+1],\ldots,A[j]] & \text{if } i\leq j,\\ [A[i],A[i-1],\ldots,A[j]] & \text{if } j>i,\end{cases} \quad (1)$$
$$0\leq i,j\leq \text{“size of A.”}$$

Concatenation

If A is of size n and B of size m then A—B is of size n+m and is defined by A|B = [A[0], A[1], . . . , A[n · · · 1], B[0], B[1], . . . , B[m · · · 1]]. All the polymorphic operators of the language (if · · · then · · · else . . . , pre, ->) can be applied to arrays. The size of an array T can be a generic parameter of a node in which T is defined or used, with the condition that for every call to that node, this parameter be instantiated by a static constant. Finally, the with operator allows for a static recursion mechanism in the language. Here, *static* means that this recursion must be ensured to terminate at compilation. The with operation allows to describe the termination condition of the recursion, which must be statically verifiable to hold.

Example

To illustrate the use of these operations we define an n-bits adder ADD in Figure 4. It takes as input two arrays A and B and computes as output the array S as the binary sum of A and B.

2.1.2. Compilation

The Pollux compiler (officially Lustre-V4) was implemented for taking care of these array notations. It basically expands arrays into independent variables. Consider the n-bits adder introduced earlier (see Figure 4). The first pass generates the intermediate program of Figure 5. The whole structure of data in arrays has been completely lost. Instead of the array A of size n, we now have n independent variables (A_0, . . . , A_9). Of course the C code obtained from thisintermediate format will also have independent variables

[3] This work has been presented in a slightly shorter form in [8].

```
const n=10;

node FULL_ADD(ci, a, b : bool)
returns (co, s : bool);
let
   s = a xor (b xor ci);
   co = (a and b) xor (b and ci) xor (a and ci);
tel

node ADD(A,B : bool^n)
returns (S : bool^n; overflow : bool);
var CARRY : bool^n;
let
   (CARRY,S) = FULL_ADD([false]
                  | CARRY[0 · · · n-2], A, B);
   overflow = CARRY[n-1];
tel
```

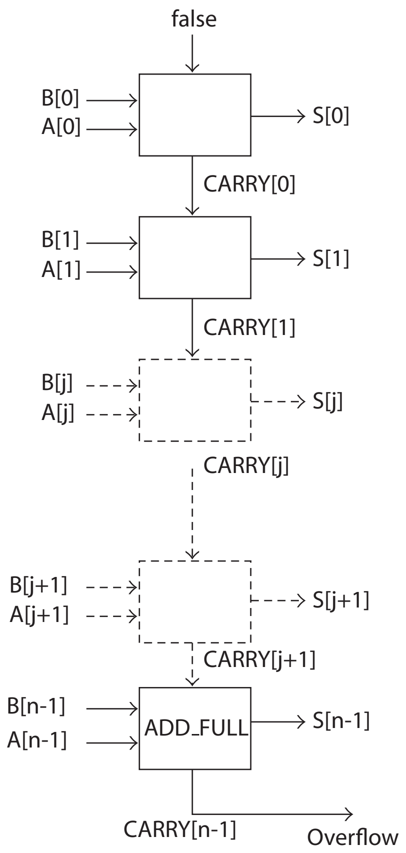


FIGURE 4: An *n*-bits adder in Lustre-v4.

instead of arrays. This method is well adapted to hardware targeting since, in the end each element of an array ought to be represented by *one wire* on the target hardware. Moreover, this approach allows for a straightforward use of standard validation tools associated to the "Lustre without arrays."

2.1.3. Towards array iterators: some motivations

For software generation, this array expansion technique is useless and can actually be harmful, leading to unnecessary code explosion. The code obtained is slow (1): we get as many memory access as there are elements in the original arrays, instead of one memory access per array in the case where we would preserve the arrays in the generated code; and (2) big: instead of generating as many assignments as there are elements in arrays, one could hope to be able to generate loops with one assignment only. For the ADD example, one would like to get the code of Figure 6. The next code generator, Lustre-V5, was an attempt to generate loop code from the operators presented above. But, using slice mechanisms, one can write a program like the one given in Figure 7. It is clearly tedious, even though possible, to write an equivalent imperative loop for this kind of program.

Conclusions

(1) Compilation techniques presently used for arrays are not adapted for obtaining efficient software code. (2) It is not always easy to generate efficient loop-like imperative code from the operators provided in the language (e.g., slice expressions). (3) These operators are not easy to use when programming classical array algorithms like sorting, maximum, and so forth. This is a particularly strong argument from final users of SCADE. (4) When expanding arrays into independent variables data arrangement is lost while it could be kept for verification. This extra argument is the basis for the work described in Section 3.

2.2. Array iterators

We now introduce iterators inspired from functional operators like map or foldl into Lustre. They only enable simple dependencies between array elements and thus make easier the generation of loop code. Generating loop presents the following advantages considering the code generated: (1) *size*: in all the cases where we apply *n* times a computation C, we reduce the number of copies of C from *n* to 1; (2) *execution time*: a C program containing *n* assignments written in sequence is generally a bit slower than an equivalent program with a loop containing 1 assignment executed *n* times; (3) *amount of memory needed* during the execution: if we use iterators it is possible to identify and suppress useless intermediate variables (see Section 2.4); (4) *readability*: the operators we propose are easy to manipulate. Their use is similar to intuitive functional operators.

Iterators have been widely used in functional programming for more than twenty years. Our contribution consists mainly in adapting these constructs to a data-flow synchronous language. In particular, safety constraints have an important influence on the constructs we introduce. These have to be deterministic fixed-size iterations. In the sequel, n is an integer whose value must be known statically. T and T′ are arrays of size n. The τs are types and τ^n is the type "array of size *n* of elements of type τ." The size of the arrays is necessary only for the fill operator, but for uniformity we give it

```
node ADD (A_0: bool; . . . ; A_9: bool;
          B_0: bool; . . . ; B_9: bool)
returns (S_0: bool; . . . ; S_9: bool; overflow: bool);
var V59_CARRY_0: bool; . . . ; V67_CARRY_8: bool;
let
    S_0 = A_0 xor B_0 xor false;
    . . .
    S_9 = A_9 xor B_9 xor V67_CARRY_8;
    overflow = (A_9 and B_9) xor (B_9 and V67_CARRY_8)
              xor (A_9 and V67_CARRY_8);
    V59_CARRY_0 = (A_0 and B_0) xor (B_0 and false)
                  xor (A_0 and false);
    . . .
    V67_CARRY_8 = (A_8 and B_8) xor (B_8 and V66_CARRY_7)
                  xor (A_8 and V66_CARRY_7);
tel
```

FIGURE 5: Intermediate code Lustre for the ADD program.

```
for(i=0; i<n; i++){
   S[i] = A[i] xor (B[i] xor C[i]);
   CARRY[i] = (A[i] && B[i])
               || (B[i] && CARRY[i-1])
               || (A[i] && CARRY[i-1]);
}
overflow = CARRY[n-1];
```

FIGURE 6: For loop we wish to get for a program manipulating arrays.

X [0] = Y [0];

X [1 · · · 2] = Y [4 · · · 5];

X [3 · · · 5] = Y [1 · · · 3];

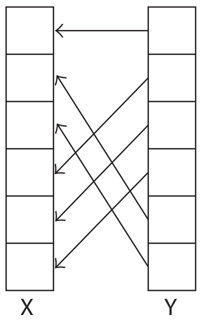


FIGURE 7: A slice expression and the corresponding dependencies between X and Y.

even for the others. For simplicity, definitions are only given for iterations of purely functional nodes, but the extension to state-full nodes is straightforward. Nodes are expressed as λ-terms. A graphical presentation of these iterators is available in Figure 8.

2.2.1. Definition

Map

If $g = \lambda t \cdot t'$, where t represents an array element and t' an expression depending on t, an abstract syntax of the map operator is $T_1 = \text{map}(g, T_2)$. It is semantically equivalent to $\{T_1[i] = g(T_2[i])\}_{i\in\text{range}(T_1)}$. If N (resp., O) is a node (resp., an operator) of signature $\tau_1 \times \tau_2 \times \cdots \times \tau_l \to \tau_1' \times \tau_2' \times \cdots \times \tau_k'$, then map≪ N,n≫ (resp., map≪O,n≫) is a node (resp., an operator)[4] of signature $\tau_1\hat{\ }n \times \tau_2\hat{\ }n \times \cdots \times \tau_l\hat{\ }n \to \tau_1'\hat{\ }n \times \tau_2'\hat{\ }n \times \cdots \times \tau_k'\hat{\ }n$.

Red

If $g = \lambda(t, \text{accu}) \cdot \text{accu}'$, the reduction r of an array T using g is $r = \text{red}(\text{init}, T, g)$, where init is the initialization expression of the reduction. It is semantically equivalent to $\{r_0 = \text{init}; \{r_{i+1} = g(r_i, T[i])\}_{i\in\text{range}(T)}; r = r_{\text{size}(T)}\}$. The operator red has this syntax: if N is a node of signature $\tau \times \tau_1 \times \tau_2 \times \cdots \times \tau_l \to \tau'$ then red≪N,n≫ is a node of signature $\tau \times \tau_1\hat{\ }n \times \tau_2\hat{\ }n \times \cdots \times \tau_l\hat{\ }n \to \tau'$.

Fill

If $g = \lambda\,\text{accu} \cdot (\text{accu}, \text{elt})$, we can have $r, T = \text{fill}(\text{init}, g)$, where init is the initialization of the filling process. It is semantically equivalent to $\{r_0 = \text{init}; \{r_{i+1}, T[i] = g(r_i)\}_{i\in\text{range}(T)}; r = r_{\text{size}(T)}\}$. In Lustre, fill has this syntax: if N is a node of signature $\tau \to \tau' \times \tau_1' \times \tau_2' \times \cdots \times \tau_k'$ then fill≪N,n≫ is a node of signature $\tau \to \tau' \times \tau_1'\hat{\ }n \times \tau_2'\hat{\ }n \times \cdots \times \tau_k'\hat{\ }n$.

Map_red

If $g = \lambda(\text{accu}, t) \cdot (\text{accu}, t)$, we have $(T_1, r) = \text{map\_red}(\text{init}, T_2, g)$. It is semantically equivalent to $\{r_0 = \text{init}; \{r_{i+1}, T_1[i] = g(r_i, T_2[i])\}_{i\in\text{range}(T_1)}; r = r_{\text{size}(T_1)}\}$. In Lustre, if N is a node of signature $\tau \times \tau_1\tau \times \tau_2 \times \cdots \times \tau_l \to \tau' \times \tau_1' \times \tau_2' \times \cdots \times \tau_k'$, then map_red≪N,n≫ is a node of signature $\tau \times \tau_1\hat{\ }n\tau \times \tau_2\hat{\ }n \times \cdots \times \tau_l\hat{\ }n \to \tau' \times \tau_1'\hat{\ }n \times \tau_2'\hat{\ }n \times \cdots \times \tau_k'\hat{\ }n$.

[4] From now on, we will not make the distinction between operators and nodes.

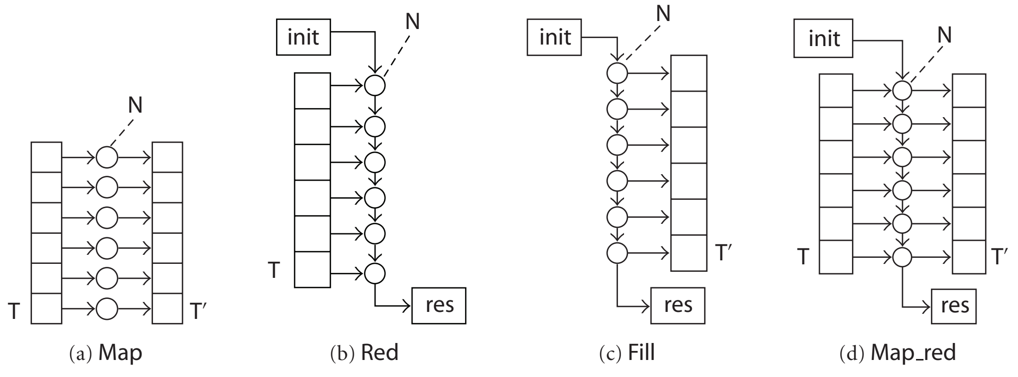


Figure 8: The four iterators introduced in Lustre.

```
node ADD(A,B:bool^n)
returns (S:bool^n;overflow:bool);
let
   overflow, S = map_red ≪ FULL_ADD; n ≫ (false, A, B);
tel
```

Figure 9: The adder, written with iterators.

2.2.2. Examples

N-bit adder

The adder example that we presented in Section 2.1 can be easily rewritten using a map_red iterator. The corresponding new version of the ADD node is given in Figure 9.

Selection of the ith element of an array

In Lustre it is not possible to select an element from an array directly from its index if the latter is given as dynamic expression (e.g., depending on the values of inputs of the program).

The iterators give us the possibility to build such functionality in a safe manner. Let us describe this program. It selects the *i*th element of an array of integers, *i* being an input. When the value of *i* is not valid (outside the bounds of the array), it returns a *default* value (here encoded as a constant default). The accumulator output of the iteration is a variable of type given in Figure 10(a).

At each stage of the iteration, these information represent: (1) the current element rank (initialized to default, and incremented of 1 at each stage); (2) the rank of the element to select simply initialized to rankToSelect. This field is propagated "as is;" (3) the value of the selected element, initialized to default. The corresponding iterated node is given in Figure 10(b).

To describe the selection of the *i*th element, we iterate selectOneStage on array. We thus define a variable of type iteratedStruct. The value of the selected element is then very simply given by iterationResult.elementSelected as shown in Figure 10(c).

```
type iteratedStruct={currentRank:int;
                     rankToSelect:int;
                     elementSelected:int};
```

(a)

```
node selectOneStage(acc_in: iteratedStruct; currentElt: int)
returns (acc_out : iteratedStruct)
let
   acc_out = {currentRank = acc_in.currentRank+1;
        rankToSelect = acc_in.rankToSelect;
        elementSele cted = if(acc_in.currentRank
                                 =acc_in.rankToSelect)
                             then currentElt
                             else acc_in.elementSelected};
tel
```

(b)

```
node selectElementOfRank_inArray_(i : int; array : int^size)
returns (elementSelected : elementType)
var iterationResult : iteratedStruct;
let
   iterationResult = red≪ selectOneStage;size≫({
      currentRank = 0,
      rankToSelect = rankToSelect,
      elementSelected = default},
      array);
   elementSelected = iterationResult.elementSelected;
tel
```

(c)

Figure 10

2.3. Compilation

The objective of this part is to describe the compilation scheme to translate iterative Lustre programs into imperative code with loops and arrays. We adopt a very simplistic approach. In particular, we are not interested in static verifications that should be performed. We suppose the following:

(i) the Lustre program is syntactically correct and it has been type-checked correctly;

```
node memo(accu_in:int)
returns (accu_out,t:int);       //Variables
let                             (1) int V;
  accu_out=accu_in->            (2) int T[10];
           pre(accu_in);        (3) int accu_out;
   t=accu_in;                   (4) int accu_in[10];
tel                             (5) int PREaccu_in[10];
node Tenlast(V:int)             //Initializing
returns (T:int^10);             // the iteration
var:foo:int;let                 (6) accu_out=V;
 foo,T=fill≪memo,10≫(V);
tel
                //Computing outputs
                (7) for(i=0;i<10;i++){
                (8)     accu_in[i]=accu_out
                (9)     T[i]=accu_in[i];
                //Initializing the memories
                (10)    if(init){ accu_out=accu_in[i];}
                (11)    else{
                (12)        accu_out=
                (13)            PREaccu_in[i];}}
                //Memorizing values
                (14) for(i=0;i<10;i++){
                (15)    PREaccu_in[i]=
                (16)        accu_in[i];}
```

Figure 11: An iterative Lustre program along with corresponding imperative code.

(ii) node calls have been inlined. During code generation, we thus only go through one node. The only other nodes we need to manipulate are those iterated in the main node.

We also do not take into account the generation of the infinite *reactivity loop* and concentrate only on *computation of outputs* and *update of memories*.

We do not have the room to give the complete algorithm here. We first illustrate it through the following example. Then, we give the outline of the algorithm. Details can be found in [12].

2.3.1. Example

We want to build a Lustre program that takes as input an integer flow V and builds an array T that contains the values of V in the last 10 instants (this number of instants has been fixed arbitrarily for the example). At each instant t, the ith element of T ($\mathsf{T}_t[i]$) contains the value of V at instant $t - i$ (V_{t-i}). The corresponding Lustre program, named Tenlast is given in Figure 11(left). It is made of a fill that iterates a node memo. At each level of the iteration, memo stores the accumulated value it receives in the corresponding element of T (represented by the output "t"). Through accu_out, it propagates the memory of the accu_in it receives. Note that during the first 10 instants, not all the elements of V have been properly set.

```
generateCode(mainN){
    generateVariableDeclarations(mainN);
    generateStep(mainN);
    generateUpdate(mainN);
}
```

Figure 12: The main function of the code generation algorithm.

On the right-hand side of Figure 11, we give the imperative code generated for the Tenlast node. Let us now look through this code. It contains variable declarations corresponding to the main inputs/outputs (V and T). Then (lines (4), (5), and (6)), we find declarations corresponding to variables that are local to the iterated node. The example raises two possibilities. First, some variables do not need to be memorized at each level of the iteration (accu_out in the example). For these, we can generate one scalar variable that can be reused at each level of the iteration. Now, some variables may also need to be memorized at each level between successive instants. This is the case for accu_in that is used both as an instantaneous value and as a memorized one (see node memo). For that, we generate two arrays, one for the value of all instances of accu_in during the current instant (declared at line (4)), and one for storing the previous values corresponding to pre(accu_in) (line (5)).

Line (6) initializes the output accumulator. Lines (7) to (13) compute the output. In that part, we generate exactly one for loop for each iteration present in the original program. Line (8) corresponds to the propagation of the accumulated value through the iteration. Then, line (9) corresponds to computing the array element T[i]. Lines (10) to (13) compute the output accumulator and distinguish two cases for that: the first instant (line (10)) and the rest of the execution.

A second loop is generated for each iteration, updating the memories that are local to the iterated node. In the example, we update (lines (14) to (16)) the memory array corresponding to pre(accu_in).

2.3.2. Intuition of the code generation algorithm

The code generation algorithm can be roughly decomposed into the steps shown in Figure 12. In this small presentation, we concentrate on aspects that are particularly relevant for the case of iterative programs. Most of usual problems arising in compiling synchronous programs (e.g., causality), code optimizations or efficiency have been put aside and can be added orthogonally.

Suppose that we start from a main node Minny where all node calls have been inlined. Particular attention needs to be given to the generation of variables. generateVariableDeclarations needs to generate the input, output, and local variables of the main node. But, it also needs to generate appropriate variables for memories that are used locally in the iterated nodes, as raised by the previous example. This generation is performed by a first complete traversal of the program that detects these memories.

```
node main(T:int^10)returns(T'':int^10);
var T':int^10;
let
    T'=map≪f,10≫(T);
    T''=map≪g,10≫(T');
tel
```

(a) Original cascade of iteration

```
node main(T:int^10)returns(T'':int);
let
    T''=map≪h,n≫(T );
tel
node h(in_f:int)returns(out_g:int);
var
    in_g:int;
    out_f:int;
let
    out_f=f(in_f);
    in_g=out_f;
    out_g=g(in_g);
tel
```

(b) Corresponding optimized program

Figure 13: An example of optimization of cascades of iterations.

A second traversal (implemented in the generateStep function) is needed to compute the actual computation of the output variables. Basically, for each Lustre equation it generates the corresponding imperative code. In the case of an iterative equation, the code generated is made of a for-loop that computes the accumulated variable as well as the output array variables (depending on the type of iteration). This function also takes care of the distinction between the initial instant and the rest of the execution.

Finally, the generateUpdate function will generate code for updating the memories that are either at the level of the main node, or at the level of iterated nodes. This is achieved by a third and last traversal of the program structure. For efficiency reasons, it could be coupled to generateStep.

2.4. Optimization

Example

A possible optimization appears when writing cascades of iterations. Consider the program of Figure 13(a). T′ is defined by a map of node f applied to T and T″ by a map of a node g applied to T′. The exact definition of f and g is of no importance here. From the definition of the map operator, we get that T′ and T″ are defined as

$$\forall i \in [0 \cdots n] \cdot T'[i] = f(T[i]),$$
$$\forall i \in [0 \cdots n] \cdot T''[i] = g(T'[i]). \quad (2)$$

From these definitions, it is obvious to see that each element of T″ can actually be defined by a composition of f and g applied to T:

$$\forall i \in [0 \cdots n] \cdot T''[i] = g(f(T[i])). \quad (3)$$

↗	Map	Fill	Red	Map-red
Map	⋆	—	⋆	⋆
Fill	⋆	—	⋆	⋆
Red	—	—	—	—
Map-red	⋆	—	⋆	⋆

Figure 14: Optimization possibilities.

While doing this, we have also used the fact that the only use we make of T′ in this program is as intermediate variable to compute T″ from T. Applying this kind of transformation directly on the Lustre program results in the program of Figure 13(b), semantically equivalent to the original one, where h has been built as a composition of f and g. We will comment on the relations with existing works in that domain in Section 2.5 but let us relate this kind of optimization to listlessness [13, 14] or deforestation [15] as they have been proposed in functional languages. Here, instead of generating the whole array T′, its elements are consumed as soon as they are produced. From a design point of view, this optimization is very useful in a context where programmers manipulate libraries of nodes performing classical array algorithms (e.g., in SCADE), not necessarily knowing that cascades appear.

Axiomatization

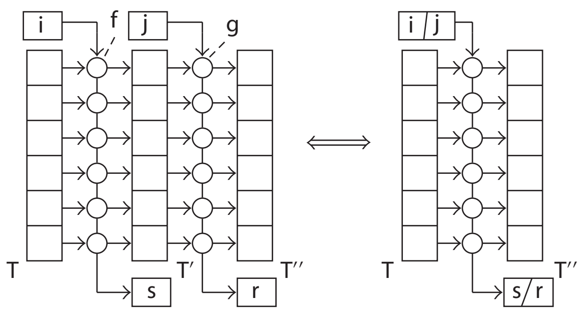
We have identified in total *nine* cascades where a similar technique can be applied. The table of Figure 14 identifies all possible optimizations. As an example, the first column of the second line reads: *the cascade "fill followed by map" can be optimized*. In order to apply these optimization axioms, we must have that (1) the result(s) of the first iteration are the input(s) of the second one; (2) these variables are not used in the rest of the node; (3) the cascade formed by the two iterations is optimizable (according to Figure 14). For keeping the presentation short, we only give the formalization for one of these optimizations.

Consider the cascade of Figure 15(a), where we suppose to have $f = a, t \cdot (a', t')$ and $g = a, t \cdot (a'', t'')$ (where a', t', a'', and t'' depend on a and t). If i_2 does not depend on r_1, we can apply the equivalence rule given in Figure 15(b) for rewriting the cascade as one iteration.

2.5. Related works

2.5.1. About iterators

The exact notion of iterators is closely related to higher-order functions and more generally to the functional programming style. Among the first propositions on this, we should recall the work of Backus [16] that basically introduces list-manipulation operators (such as *Insert* or *Apply to All*) to functional programing and wonders the first about possible simplifications of compositions of such functions. This work has been pursued during the years (leading to very nice formalisms such as BMF [17]).



(a) A graphical representation

$$\boxed{\begin{gathered}\text{map\_red}(j, \text{map\_red}(i, T, f), g)\\ \equiv\\ \text{map\_red}(\{i, j\}, T, \lambda\{a_1, a_2\}, t\cdot \text{ let}\\ \langle x, y\rangle = f(a_2, t) \text{ in let } \langle x', y'\rangle\\ = f(a_2, y) \text{ in } \langle\{x, x'\}, y'\rangle).\end{gathered}}$$

(b) The optimization rule

FIGURE 15: Optimizing the cascade map_red(map_red).

Right from the start, these works were meant to deal with infinite list (actually, more generally with infinite tree-like structures). The operations that we propose are very limited compared to the one included in many functional languages. This is mainly because of the *high criticality* of the application domain we aim at. Introducing iterators in the Lustre should typically not lead to unbounded computations and dynamic creation of data structures. The operations we propose are quite simple (actually already too complicated from the final user's point of view) and lead to unambiguously "*safe*" code. Such operations (map, reduces, etc.) have also been introduced into languages that are more closely related to Lustre such as $8_{1/2}$ [18], ALPHA [19] that are both dataflow languages. The difference here is that these operations have been introduced to help hardware architecture-related problems (parallelism of computation), which is quite the opposite goal from the one we have here.

2.5.2. About optimizations of cascades

In [20], Waters underlines the advantages of programming with *serial expressions* and of the optimization techniques that can be used in that framework. The basic type considered here is *list* and the advantages of using higher-order functions are presented. The author also underlines two reasons why the techniques are not widely used: (1) constructs proposed in functional languages are not easy to use; (2) the compilation techniques used are rarely efficient, mainly because intermediate structures are not taken care of properly in the case of cascades of serial functions.

This joins the work by Wadler on listlessness [13, 14] and later on deforestation [15]. Listlessness consists exactly in what we aim at in our optimization process of Section 2.4: intermediate lists should not be built completely before one can start to consume their elements. Deforestation is simply a generalization of listlessness to tree-like structures. An implementation of these deforestation techniques is presented in [21]. A technique derived from this, called *warm fusion* is presented in [22].

2.6. A word about impact and technology transfer

The ideas we have presented in this section have been the fruit of a thorough collaboration with Esterel Technologies. Jean-Louis Colaço, chief investigator regarding the Lustre language at Esterel Technologies has incorporated the iterations as well as the optimization algorithms presented earlier in the experimental compiler of the company. Convincing experimental results have been obtained, particularly on an Airbus A380-related case study. This application manages the electrical load in an aircraft. Redundancy of data and parallelism are central to this type of applications because they represent the best way to ensure fault tolerance. There, the introduction of iterators has lead to a target code-size reduction of a factor 300. This reduction was due both to the restructuring of the source code implied by the iterators and the generation of loops instead of inlined elementary computations.

Our iterations are well adapted to this kind of applications, as shown by this particular case study, but also by two other ones (both were taken from the avionics domain). The important practical result of this collaboration is that industrial partners have been convinced by the usefulness of the whole approach and that, as of 2008, the iterators will be part of the new SCADE 6.0 tool. During this collaboration [23], iterators have also been ported to Lucid Synchrone [24], an synchronous extension to ML. Last but certainly not least, the iterators are now included in the Lustre-v6 language version. The compiler, still under development at the time of writing this paper, implements the compilation and optimization phases that we have proposed.

3. EXPLOITING SYSTEM'S REGULARITIES: THE VALIDATION ASPECT

In the preceding section, we have introduced operators that are well adapted for the description of regular systems. We have focused our attention on the advantages of this language extension regarding language usability and code generation. The next step to be considered consists in taking this into account in the validation process.

Concerning verification, the approach that has been applied traditionally consists in expanding the arrays into independent variables and use standard validation techniques on the expanded code. This approach presents the following inconvenients:

(i) the regular programs we deal with are generally big and most verification tools will suffer from a state-explosion problem;

(ii) this expansion forbids tools to take this regular structure into account, while it might be of importance for validation.

The goal of the work presented in the subsequent sections is to propose a methodology for taking this regular structure into account during the validation process. In Section 3.1, we discuss the type of properties we want to be able to treat. Section 3.2 presents the methodology itself (based on a slicing algorithm) that, given a property on an iterative program that deals with arrays, produces a set of smaller properties on elements of arrays that are *sufficient* to prove the initial property. Finally, Section 3.3 sums up related works.

3.1. Expressing properties on arrays

As mentioned earlier, a Lustre property is expressed with an observer, that is, a node that has as inputs the inputs/outputs of the program being considered and as sole output a Boolean variable representing the truth value of the property. Such a property can be expressed on array variables using all the expressive power of the language. In general, we consider properties expressed as reductions with Boolean accumulator output or properties on results of reductions of different types.

As an extension, we introduce a new operator forall to the language. This allows for expressing *perfectly regular* properties that present the advantage of leading to a more conservative proof result. The introduction of this operator is motivated by the following: (1) it is not straightforward for the programmer to express a regular property using the standard red operation because the symmetry needs to be hidden in the reduction; (2) that symmetry being embedded in the reduction makes it actually hard to identify by automatic validation tools; (3) lots of practical examples we have encountered use arrays to express redundancy of data, typical in Fault-tolerant controllers. Classical properties on these redundant data are symmetric.

Forall

If $g = \lambda t \cdot b$ is an observer (t is a scalar parameter representing an array element and the expression b is Boolean), an abstract syntax for the forall operator is ok = forall(g, t). It is semantically equivalent to ok $= \bigwedge_{i=0}^{i=\text{size}-1} g(T[i])$. The operator forall has the following syntax: if P is an observer of signature $\tau_1 \times \tau_2 \times \cdots \times \tau_l \rightarrow$ bool then forall≪P,n≫ is a node of signature $\tau_1\hat{}n \times \tau_2\hat{}n \times \cdots \times \tau_l\hat{}n \rightarrow$ Boolean. Every property expressed with a forall can be translated in the form of a Boolean red iteration.

3.2. A proof methodology

We consider a validation scheme such as that of Figure 3. Now, consider a program P and a property φ. Both use iterations. Our goal is to prove φ on P. From a more practical point of view, we will consider that φ is integrated in P (see Figure 16) which leads us back to considering a reactive "*box*" from which a Boolean value is outputed. This observation greatly simplifies the presentation without reducing its generality.

We exploit the regular structure of both φ and P in order to extract proof objectives *simpler* than φ itself. In practice, these proof objectives are generated as Lustre observers. The advantage of this technical choice is that these proof objectives can be then fed to standard validation tools, like model-checkers and theorem provers.

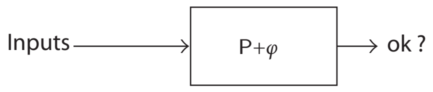


FIGURE 16: φ is integrated to P.

```
node obs(T1 : int^10) returns (ok : bool);
var T2 : int^10;
let
    T2 = map≪N;10≫(T1);
    ok = forall≪onePositive;10≫(T2);
    assert forall≪onePositive;10≫(T1);
tel
```

(a) Purely symmetric property

```
node obs_bis(elt_T1 : int) returns (ok : bool);
var elt_T2 : int;
let
    elt_T2 = N(elt_T1);
    ok = onePositive(elt_T2);
    assert onePositive(elt_T1);
tel
```

(b) Proof obligation for the property of Figure 17(a)

FIGURE 17

Our presentation will follow a gradually complicating path through different cases: in Section 3.2.1 we look at how to slice symmetric properties, that is, φ is expressed using a forall. We extend this simple approach to the case where the property is expressed by a single reduction red (in Section 3.2.2). In Section 3.2.3 we explain how to propagate the slice method to cascades of iterations such as those presented in Section 2.4. Finally, Section 3.2.4 considers the generalization of the approach to complex networks of operations and pinpoint limitations of our method.

3.2.1. *Simple* forall *properties*

Consider the observer of Figure 17(a). It expresses the following property: "*if all the elements of* T1 *are positive and if* T2 *is defined by a map of node* N, *then is it the case that the elements of* T2 *are also positive?*" Our slicing technique applies the following argument: to prove this property, it is sufficient to prove the property given at Figure 17(b) that expresses that "*if a variable* elt_T1 *is positive then a variable* elt_T2, *computed as the result of applying* N *to* elt_T1 *is positive*."

```
node Obs(T:int^s) returns (ok:bool);
var n:int;
let
    n = red≪plus;s≫(0,T);
    ok = phi(n);
tel
node phi(v:int) returns (Pv:bool);
let
    Pv = v>0;
tel
```

(a) A simple iterative property

```
Acc = 0;                          β
for(int i = 0;i<s;i++){
                                  γ
Accu = plus(Accu,T[i]);
                                  γ'
}
n = Accu;                         α
```

(b) Computing the value n

Figure 18

To formalize this, we introduce a form of transformation rule. The rule for the forall case is

$$\frac{\lambda \mathsf{t} \cdot \mathsf{P}(\mathsf{N}(\mathsf{t}))}{\lambda \mathsf{T} \cdot \forall (\mathsf{P}, \mathsf{map}(\mathsf{N}, \mathsf{T}))} \tag{4}$$

that reads *if the property is described by a cascade of a* map *followed by a* forall*, then to prove the property, it is sufficient to prove the iterated property* P *is valid after a call to node* N. In practice, the application of such a rule is done by generating the corresponding proof obligation as a Lustre node (see Figure 17(b)).

3.2.2. *Principle of the treatment of a* red *property*

We are now interested in a property expressed as a red. Consider the example of Figure 18(a). Node phi encodes the property *the value of* v *is positive* (here by positive we implicitly mean "strictly positive"). The property encoded by the node Obs is that *the sum of the elements of* T *is positive*.

Let us have a look at the imperative code (see Figure 18(b)) that represents the code executed for that program at every execution cycle to compute the value n. It is made of a simple for loop obtained from the red iteration (see Section 2.1.2). On the right-hand side of this program, we have indicated 4 mark points: α, β, γ, and γ'. The property ϕ must be verified in α in the imperative code. We know that in order to prove ϕ in α, it is sufficient to prove that ϕ is an invariant of the loop. To achieve that, we apply the standard Hoare-logic approach (see Section 3.3) and try to prove (1) a base case expressing that ϕ is true in β and (2) an invariance case expressing that ϕ is preserved by one passage in the loop body (i.e., if ϕ is true in γ it is also true in γ').

Base case

As a matter of fact, we will not consider the base case to hold in β but rather in γ' (after exactly one iteration has been performed). This comes from the fact that the initialization of the iteration, traditionally performed before the loop and thus fixing the invariant of the iteration before it actually starts, is made during the first round of our iteration. This is not restricting the approach since we do not consider empty arrays anyway. As an example, consider again the iteration of Figure 18(a). Under the hypothesis that *all the elements of* T *are positive*, ϕ is satisfied. The proof rule that we propose leads naturally to the following base case: *in β, the accumulator* Acc *is positive*. The corresponding *induction case* is *if* Acc *is positive in γ, then* Acc *is positive in γ'*. This latter is trivially true, but the base case is on the contrary trivially false because Acc is initialized to 0. By considering the base case in γ, it is true.

Manipulation rule

The properties we consider in the example along with points α and β are all represented in Figure 19. This also indicates the places in the iterations where we consider the base and invariance case. In order to prove this property, we will generate a conjunction of properties, this first one for proving the base case (see Figure 20(a)) and the other for proving the invariance of the property. This transformation is represented by the following rule: consider the property ϕ on the result of a reduction of a node N, that is, the expression (see Figure 19) $\lambda\,\text{init}, T\cdot\phi(\text{red(init,T,N)})$ where init is the initialization input of the reduction and T is the array input. From this, we generate two observers. The first one (see Figure 20(a)) expresses that ϕ is true after one passage in the iteration $\lambda\text{init}, t\cdot\phi(\mathsf{N}(\text{init}, \mathsf{t}))$. The second (see Figure 20(b)) expresses that if the property is satisfied at a any level of the iteration ($\phi(\text{acc})$), then it is also satisfied at the next level (after a call to the iterated node $\phi(N(\text{acc}, t))$) $\lambda\text{acc}, t\cdot\phi(\text{acc}) \Rightarrow \phi(\mathsf{N}(\text{acc}, \mathsf{t}))$. This rule is formalized by

$$\frac{\lambda\mathsf{init}, \mathsf{t} \cdot \phi(\mathsf{N}(\mathsf{init}, \mathsf{t})) \qquad \lambda\mathsf{acc}, \mathsf{t} \cdot \phi(\mathsf{acc}) \Longrightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{t}))}{\lambda\mathsf{init}, \mathsf{T} \cdot \phi(\mathsf{redd}(\mathsf{N}, \mathsf{T}, \mathsf{init}))}. \tag{5}$$

Taking an iterative assertion into account

Generally, properties are given along with *assertions* that encode the assumption that the user has on the input of the program, and that should hold so that the property can hold. To take these assertions into account, we simply apply our slicing algorithm to the property $\mathcal{H} \Rightarrow \mathcal{P}$.

3.2.3. *Propagation to cascades of iterations*

The preceding technique applies to single (red or forall) iterations. We are now interested in cascades of iterations. We will use the particular case of a map followed by a red as an illustration. Consider the cascade program of Figure 21. T is

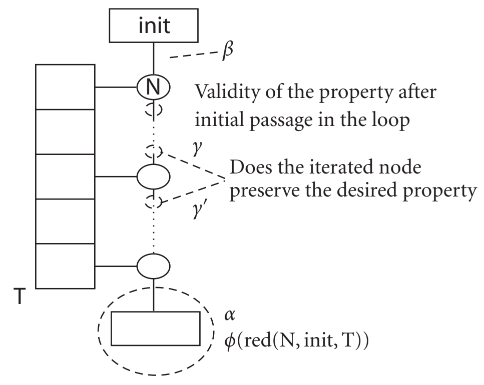
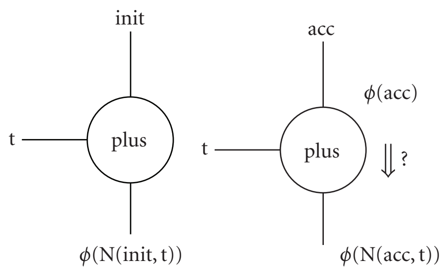

init
β
N
Validity of the property after initial passage in the loop
γ
Does the iterated node preserve the desired property
γ′
T
α
ϕ(red(N, init, T))

Figure 19: Iterative property.



(a) Base case (b) Invariance

Figure 20

computed by an iteration of a node plusOne so that every element of T is equal to the corresponding element of T′ plus 1: for all $i \cdot T[i] = T'[i] + 1$. We want to prove the same property as before and thus use the observer Obs of Figure 21. We follow the same line as before, and give in Figure 22(a) the imperative code corresponding to that program.

One way to treat cascades of iterations would be to first apply the optimization of Section 2.4 and then apply the rule given above. The optimizations are well adapted to code generation because the programmer is generally not interested in reading the generated code. For verification, this argument is not valid anymore. Readability is much more important, and one should avoid as much as possible to derive from the form of the original program.

We thus propose a manipulation rule similar to that proposed for a simple red iteration that allows us to keep the cascade's structure in the proof obligation we generate:

$$\frac{\begin{array}{c}\lambda\mathsf{init}, \mathsf{t} \cdot \phi(\mathsf{N}(\mathsf{init}, \mathsf{M}(\mathsf{t})))\\ \lambda\mathsf{acc}, \mathsf{t} \cdot \phi(\mathsf{acc}) \Longrightarrow \phi(\mathsf{N}(\mathsf{acc}, \mathsf{M}(\mathsf{t})))\end{array}}{\lambda\mathsf{init}, \mathsf{T} \cdot \phi(\mathsf{redd}(\mathsf{N}, \mathsf{map}(\mathsf{M}, \mathsf{T}), \mathsf{init}))}. \qquad (6)$$

3.2.4. Propagation to operation networks

The generalization of the approach presented in previous sections consists in traversing the cascades of iterations backwards (see Figure 23), starting from the final reduction and going back through the program structure until we have analyzed all its equations. For each iteration encountered during this traversal, we generate a node call for the base case of the induction and a node call for the invariance case. We are only interested in slicing the array structure. This traversal is limited to iterations. It is not applied to expressions like P_1 computing the initial value of iterations nor to general networks of operators (like P_2). We will see in Section 3.2.6 that we can apply this transformation to node calls C, under certain circumstances.

```
node Obs(T′:int^s)
returns (ok:bool);
var n:int;T:int^s;
let
  T = map≪plusOne;s≫(T′);
  n = red≪plus;s≫(0,T);
  ok = phi(n);tel
```

Figure 21: An observer with a simple cascade of iterations.

```
for(int i = 0;i<s;i++){
    T[i] = plusOne(T′[i]);
}

Acc = 0;
for(int i = 0;i<s;i++){
    Accu = plus(Accu,T[i]);
}
n = Accu;
```

(a) Imperative loop version

```
Acc = 0;                          | β
for(int i = 0;i<s;i++){           |
                                  | γ
    T[i] = plusOne(T′[i])         |
    Acc = plus(Accu,T[i]);        |
                                  | γ′
}                                 |
n = Accu;                         | α
```

(b) The same program after a loop fusion

Figure 22

3.2.5. Example

We will now illustrate our technique on the cascade of iterations given at Figure 24. In order to concentrate on the algorithm, we have deliberately built up an example. We are not particularly interested in the meaning of this program. Suffice it to say that it illustrates pretty well the possibilities of our approach. In Section 3.2.7, we will comment the interest of our method for real-life case studies.

The Boolean output ok of this observer is computed by a reduction of a node P applied on an array tab_out. This in turn is computed by a map of N on an array T4. T4 is an output of a map_red of Q. This iteration also computes the

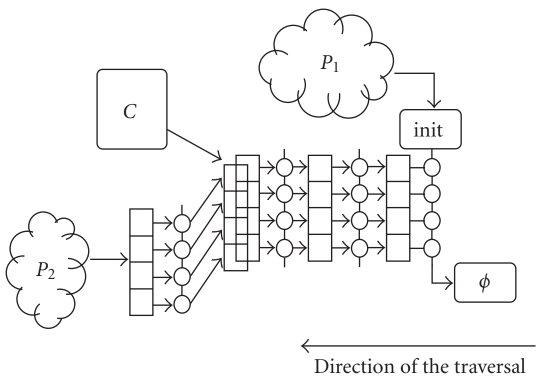


FIGURE 23: Limitations of our slicing algorithm.

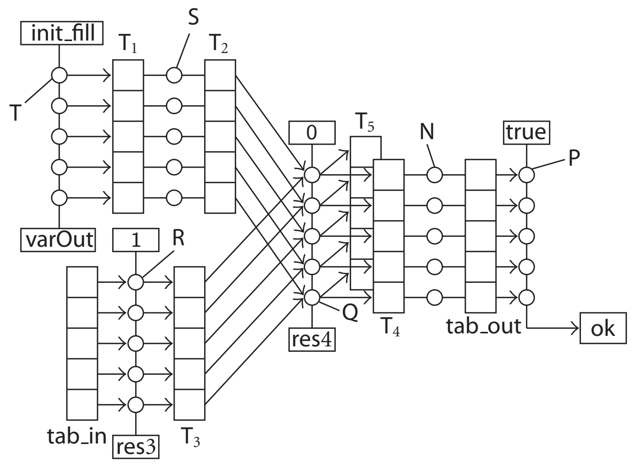


FIGURE 24: Iteration network of observer prop.

variable res4 and the array T5 (but we are not interested in these for our property). As parameters, it takes the constant 1 that initializes the iteration as well as two arrays T2 and T3. T2 is computed by a map of S on T1, itself filled by a fill initialized with an input of the program init_fill. Finally, T3 is computed along with res3, with a map_red iteration of R on tab_in (also an input of the observer).

The transformation of this observer is applied backward, starting from the reduction that computes the output ok. We start by generating equations defining two local variables: (1) okInit expresses that the iterated property is true after an initial call to the iterated node; (2) okInv expresses that the property is preserved by a call to the iterated node. To compute NV, we define two variables propRanki and propRanki_plus1. propRanki is an input of the new observer and represents the hypothesis that *the property is true at a rank* i. propRanki_plus1 represents the fact that *the property is true at rank* i+1. After we have generated these *basic* equations, we need to propagate this generation of the init and inv cases to the cascade of iterations until we reach the inputs of the program. For iteration

$$\text{res3}, \text{T3} = \text{map\_redd} \ll \text{R}; \text{size} \gg (1, \text{tab\_in}); \qquad (7)$$

we generate the following two equations:

$$\begin{aligned} &\text{res3\_init}, \text{elt\_T3\_init} = \text{R}(1, \text{elt\_tab\_in\_init}); \\ &\text{res3\_inv}, \text{elt\_T3\_inv} = \text{R}(\text{accIn\_R}, \text{elt\_tab\_in\_inv}). \end{aligned} \qquad (8)$$

In Figure 25, we give a graphical representation of the observer thus generated.

3.2.6. Taking node calls into account

As mentioned earlier, the algorithm we propose is propagating this slicing of arrays until it reaches some boundaries, defined by the form of the network of operators that constitutes the program. One possibility of extension of this algorithm is to take node calls into account. When encountering node calls during the traversal, there are actually two situations that we can consider. First, the node called is just used to encapsulate regular computations. Then we can inline this node *on the fly* and apply the transformation to the iterations that it contains. Second, the node actually encapsulates complicated computations, that the user particularly does not want to be exposed to the rest of the program, as it would make the analysis particularly difficult. In that case, it would make a lot of sense for the user to give a *contract* to that node. An *assume-guarantee contract* [25] is a form of local specification. It is made of an assertion Boolean clause, that specifies what the component expects from its environment, and a guarantee clause that specifies what the component guarantees to its environment, assuming that the latter satisfies its assumption. If such a contract is given for the node encountered, it is possible to abstract the node away (its outputs can become inputs of the proof obligation) and actually replace it by an assertion on these newly created inputs. This assertion is defined as the logical implication of the assume and guarantee clauses assume ⇒ guarantee, and can be built syntactically. The same technique as before can be used now to slice this new proof obligation.

3.2.7. Implementation and impact

This whole approach is implemented in a prototype tool named GOuPIL.[5] This should be seen as a tool provided to Lustre programmers in order to facilitate the validation of their programs. The manipulations are automated, but the user is still intensively implicated since he needs to command the tool. Some of the manipulations we proposed here (e.g., the replacement of a node call by an assertion) are not automated yet, this is still a work in progress. At any moment in the process, the user can get the generated Lustre proof obligations. The automatic connection to model checkers such as

[5] Available from http://www.irisa.fr/espresso/Equipe/Morel/html/distrib-goupil/goupil-1.0.html.

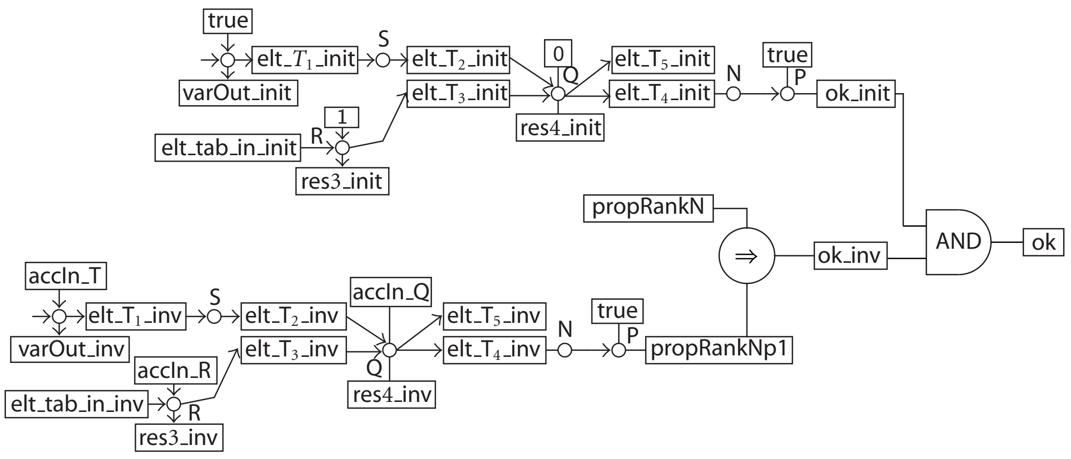


Figure 25: The observer prop_dup.

nBac or theorem provers is not implemented yet, so once he gets proof obligations as Lustre observers from GOuPIL, the user still needs to run manually the adequate tool. However, it has served as an experimental platform and has helped us to validate the approach on several large-scale case studies.

The whole technique has been applied to a significant case study, which we will not describe completely here, for obvious space reasons. This application is again taken from the avionics field. It is an implementation in Lustre of an EDF (earliest deadline first) protocol to manage a list of processes that have to run on a single processor. Arrays are used to describe lists of priority and deadline information about processes. Iterators are used to manage these lists and decide, at every instant, which process should be executed next. We want to prove a global property on the system, that states that the process that is being executed is always the one with the higher priority.

To evaluate our propositions, we have first tried to use standard verification to prove the property. We tried to prove it using the Lustre tool box, that includes the model-checker Lesar, and the abstract interpreter nBac. This rapidly led to a state explosion, which was due to the regularity and the size of the system combined with the numerical aspect of the property. Then, starting from the original property, we applied our slicing algorithm with GOuPIL. This gave us a proof objective comparable to the one of Figure 25. The numerical part of this proof objective was still complex, but we were able to finally prove it with nBac.

3.3. Related works

Symmetry

Techniques taking symmetry into account have been proposed in several works and are generally applied on the state machine representation of programs. Their main goal is to reduce the state explosion associated with model checking. Structural symmetries imply an equivalence relation between the states on an automaton. It is then sufficient to explore only one state per equivalence class when traversing the automaton. The first use of a notion of symmetry was proposed for Petri-Nets [26]. The most significant propositions in that domain are those of Emerson et al. [27–29] and of Dill and Ip as implemented in the Murφ system [30–32]. Recognizing symmetries is performed on a graph model of the system while our approach is provided at *language level*. This presents two advantages: it gives better feedback to the user and we do not constrain ourselves to using this method for one particular validation technique (model checking) but keep availability of the whole validation framework.

Proof of iterative algorithms

A particular rule of Hoare logic, first proposed in [33] and largely studied in [34], concerns the proof of *while* iterations **while** B **do** S. To prove P{**while** B **do** S}R, we need to (1) find an invariant Inv, that is, a property Inv such that Inv and B{S}Inv (i.e., that Inv is preserved by the statement S); (2) prove that P ⇒ Inv and that Inv and not B ⇒ R. The major difficulty of this approach is of course to *find* the invariant Inv. In our particular case, we will try to prove that R *is* an invariant, so that R and B{ S} R. Of course, the two conditions are not *necessary* to prove R since R is maybe not an invariant of the iteration. So whenever our method is not able to prove R it does not mean that R is false, but simply that we cannot answer. Finally, this proof of a while loop is only partial. But this is sufficient in our case since iterations have statically-fixed sizes (hence termination is guaranteed).

Verification of regular architectures

In a domain closer to ours, [35] and more recently [36] have proposed a verification method for regular architectures described in Alpha [19]. Part of this method consists in simplifying the system's implementation by applying inductive rules on unidimensional structures. This induction rule applies to flow variables and allows the authors to prove invariance of a property over time in a way similar to that proposed in [37]. This rule, which in many ways resembles the one we propose in Section 3.2.2 is not used to treat other (spatial) dimensions of the program.

4. CONCLUSIONS

In this paper, we have proposed an extension of the synchronous language Lustre with array iterators. From a language point of view, these operators do not increase the expressive power provided to the final user. Rather, they make more natural the expression of iterative algorithms. Another important aspect of these operators is that it is easy to generate efficient imperative loop-like code from them. The compilation method outlined in Section 2.3 has been put in practice in the experimental compiler of Esterel Technologies, and will be included in the next official release of SCADE. The efficiency of the code generation also accounts to the optimization of cascades of iterations. In SCADE, iterators will be provided as libraries of standard iterative algorithms. These optimizations are then very useful because cascades can appear without the programmer being aware of them.

Trying to make our approach as complete as possible, we have then studied the possibility of taking the regular structure implied by iterations in the validation process. To that end, we have proposed a slicing algorithm that, given a Lustre property on an iterative program generates smaller proof obligations (expressed also in Lustre) on parts of programs concerning elements of arrays. These proof obligations are only sufficient: if we are able to prove that they are true, then the original property is true as well. If we are not able to prove these obligations, then we cannot conclude. In practice, these proof obligations are generated as Lustre observers. This allows for the full power of validation tools available around the language instead of making the slicing algorithm incorporated in a specific tool.

This work is particularly interesting because it is somewhat complete: starting from a requirement from actual users of the Lustre language, we have studied a language extension, a compilation scheme taking full advantage of this extension as well as an adapted validation technique. Moreover, the technology transfer partnership with Esterel Technologies has already led to the integration of the iterators in the SCADE tool set (to be effective in the next version, available in 2008). As far as validation is concerned, the results we had on applying the slicing mechanics to different case studies are a good promise. An interesting perspective is certainly to build a more convincing interface for applying it and better connections to validation tools to make the treatment of proof obligations more transparent.

ACKNOWLEDGMENTS

This work, representing the main contribution of the author's Ph.D. thesis [38], has been supported by CNRS and INPG. The author is deeply indebted to Florence Maraninchi, without whom most of the ideas appearing in this paper would never have made their way, and to MM. Marc Pouzet and Patrice Quinton for their thorough reviews of the thesis.

REFERENCES

[1] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and Models of Concurrent Systems*, pp. 477–498, Springer, New York, NY, USA, 1985.

[2] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

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

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

[5] P. Le Guernic and A. Benveniste, "The synchronous language SIGNAL," in *Proceedings of the 2nd Workshop on Large-Grained Parallelism*, M. R. Barbacci, Ed., pp. 56–57, Pittsburgh, Pa, USA, November 1987, Carnegie-Mellon University Software Engineering Institute.

[6] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST '93)*, M. Nivat, C. Rattray, T. Rus, and G. Scollo, Eds., Workshops in Computing, pp. 83–96, Twente, The Netherlands, June 1993.

[7] B. Jeannet, *Partitionnement Dynamique Dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones*, Ph.D. thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 2000.

[8] L. Morel, "Efficient compilation of array iterators for Lustre," in *Proceedings of the 1st Workshop on Synchronous Languages, Applications, and Programming (SLAP '02)*, F. Maraninchi, A. Girault, and É. Rutten, Eds., vol. 65 of *Electronic Notes in Theoretical Computer Science*, Grenoble, France, April 2002.

[9] F. Rocheteau, *Extension du langage Lustre et application la conception de circuits: le langage Lustre-V4 et le système Pollux*, Ph.D. thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 1992.

[10] F. Rocheteau and N. Halbwachs, "Pollux: a Lustre-based hardware design environment," in *Proceedings of the International Workshop on Algorithms and Parallel VLSI Architectures II*, P. Quinton and Y. Robert, Eds., pp. 335–346, Chateau de Bonas, France, June 1991.

[11] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to programmable active memories," in *Systolic Array Processors*, J. McCanny, J. McWhirter, and E. Swartzlander, Eds., pp. 301–309, Prentice-Hall, Englewood Cliffs, NJ, USA, 1989.

[12] L. Morel, "Generating imperative code from Lustre iterators," http://www.irisa.fr/espresso/Equipe/Morel/Publications/algoCodeGeneration/algo.pdf.

[13] P. L. Wadler, "Listlessness is better than laziness: lazy evaluation and garbage collection at compile-time," in *Proceedings of the ACM Symposium on LISP and Functional Programming*, pp. 45–52, Austin, Tex, USA, August 1984.

[14] P. L. Wadler, "Listlessness is better than laziness II: composing listless functions," in *Proceedings of a Workshop on Programs as Data Objects*, vol. 217 of *Lecture Notes in Computer Science*, pp. 282–305, Copenhagen, Denmark, October 1985.

[15] P. L. Wadler, "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science*, vol. 73, no. 2, pp. 231–248, 1990.

[16] J. Backus, "Can programming be liberated from the von neumann style? A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[17] R. S. Bird, "Lectures on constructive functional programming," in *Constructive Methods in Computer Science*, M. Broy, Ed., vol. F55 of *NATO ASI Series*, pp. 151–216, Springer, New York, NY, USA, 1988.

[18] J.-P. Sansonnet, O. Michel, and D. De Vito, "8-1/2: data-parallelism and data-flow," Tech. Rep. LRI-CNRS, Université Paris-Sud, Orsay Campus, France, 1992.

[19] C. Mauras, *Alpha, un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*, Ph.D. thesis, Université de Rennes I, Rennes, France, December 1989.

[20] R. C. Waters, "Automatic transformation of series expressions into loops," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 52–98, 1991.

[21] A. Gill, J. Launchbury, and S. L. Peyton Jones, "A short cut to deforestation," Tech. Rep., University of Glasgow, Glasgow, UK, October 1993.

[22] J. Launchbury and T. Sheard, "Warm fusion: deriving build-catas from recursive definitions," in *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pp. 314–323, La Jolla, Calif, USA, June 1995.

[23] J.-L. Colaco and M. Pouzet, "Prototypages," Rapport final du projet GENIE II, Verilog SA, Paris, France, Janvier 2000.

[24] P. Caspi and M. Pouzet, "Lucid Synchrone, a functional extension of Lustre," Tech. Rep., Laboratoire LIP6, Université Pierre et Marie Curie, Paris, France, 2000.

[25] F. Maraninchi and L. Morel, "Logical-time contracts for reactive embedded components," in *Proceedings of the 30th EUROMICRO Conference on Component-Based Software Engineering Track (ECBSE '04)*, vol. 30, pp. 48–55, Rennes, France, August 2004.

[26] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen, "Towards reachability trees for high-level petri nets," in *Advances in Petri Nets*, vol. 188 of *Lecture Notes in Computer Science 1984*, pp. 215–233, Springer, New York, NY, USA, 1985.

[27] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," in *Proceedings of the 5th International Conference on Computer Aided Verification*, pp. 463–478, Austin, Minn, USA, November 1993.

[28] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry reductions in model checking," in *Proceedings of the 10th International Computer Aided Verification Conference*, pp. 145–458, Vancouver, BC, Canada, June-July 1998.

[29] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha, "Exploiting symmetry in temporal logic model checking," *Formal Methods in System Design*, vol. 9, no. 1-2, pp. 77–104, 1996.

[30] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *Proceedings of the 11th International Conference on Computer Hardware Description Languages and Their Applications (CHDL '93)*, D. Agnew, L. Claesen, and R. Camposano, Eds., vol. 32 of *IFIP Transactions A: Computer Science and Technology*, pp. 97–112, Amsterdam, The Netherlands, April 1993.

[31] C. N. Ip and D. L. Dill, "Efficient verification of symmetric concurrent systems," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '93)*, E. Straub, Ed., pp. 230–234, Cambridge, Mass, USA, October 1993.

[32] C. N. Ip and D. L. Dill, "Verifying systems with replicated components in murφ," in *Proceedings of the 8th International Conference on Computer Aided Verification (CAV '96)*, vol. 1102 of *Lecture Notes in Computer Science*, pp. 147–158, New Brunswick, NJ, USA, July-August 1996.

[33] C. A. R. Hoare, "An axiomatic basis of computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[34] S. K. Basu and J. Misra, "Proving loop programs," *IEEE Transactions on Software Engineering*, vol. 1, no. 1, pp. 76–86, 1975.

[35] C. Dezan and P. Quinton, "Verification of regular architectures using ALPHA: a case study," Tech. Rep., INRIA, Paris, France, June 1994.

[36] K. Morin-Allory, *Vérification Formelle dans le Modèle Poly edrique*, Ph.D. thesis, Université de Rennes 1, Rennes, France, 2004.

[37] C. Dumas and P. Caspi, "A PVS proof obligation generator for Lustre programs," in *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, vol. 1955 of *Lecture Notes in Artificial Intelligence*, pp. 179–188, Saint Denis, France, November 2000.

[38] L. Morel, *Exploitation des Structures Régulières et des Specifications Locales pour le Developpement Correct de Systèmes Réactifs de Grande Taille*, Ph.D. thesis, Institut National Polytechnique de Grenoble, Grenoble, France, 2005.