Extending Structural Test Coverage Criteria for Lustre Programs with Multi-clock Operators

Virginia Papailiopoulou, Laya Madani, Lydie du Bousquet, Ioannis Parissis

University of Grenoble - Laboratoire d'Informatique de Grenoble
BP72, 38402 Saint Martin d'Hères Cedex - France
{Virginia.Papailiopoulou, Laya.Madani, Lydie.du-Bousquet, Ioannis.Parissis}@imag.fr

Abstract. Lustre is a formal synchronous declarative language widely used for modeling and specifying safety-critical applications in the fields of avionics, transportation or energy production. Testing this kind of applications is an important and demanding task during the development process. It mainly consists in generating test data and measuring the achieved coverage. A hierarchy of structural coverage criteria for Lustre programs has been recently defined to assess the thoroughness of a given test set. They are based on the operator network, which is the graphical representation of a Lustre program and depicts the way that input flows are transformed into output flows through their propagation along the program paths. The above criteria definition aimed at demonstrating the opportunity of such a coverage assessment approach but doesn't deal with all the language constructions. In particular, the use of multiple clocks has not been taken into account. In this paper, we extend the criteria to programs that use multiple clocks. Such an extension allows for the application of the existing coverage metrics to industrial software components, which usually operate on multiple clocks, without negatively affecting the complexity of the criteria.

1 Introduction

Synchronous software is normally part of safety-critical applications in such domains as avionics, transportation and energy. Formal specification is usually required to model the system behavior along the different levels of the development process. Such a specification not only describes the correct function of the system but also it defines the conditions under which that correct function is reached. That specification can be further used to automatically generate test data.

Several programming languages have been proposed to specify and implement synchronous applications, such as Esterel [2], Signal [8] or Lustre [5,1]. Lustre is a declarative, data-flow language, which is devoted to the specification of real-time applications. It provides formal specification and verification facilities and ensures efficient C code generation. It is based on the synchronous approach which demands that the software reacts to its inputs instantaneously. In practice, that means that the software reaction is sufficiently fast so that every change in the external environment is taken into account. As soon as the

order of all the events occurring both inside and outside the program is specified, time constraints describing the behavior of a synchronous program can be expressed [6]. These characteristics make it possible to efficiently design and model synchronous systems.

A graphical tool dedicated to the development of critical embedded systems and often used by industries and professionals is SCADE (Safety Critical Application Development Environment). SCADE is a graphical environment used in the development of safety-critical embedded software. It is based on the LUSTRE language and it allows the hierarchical definition of the system components and the automatic code generation. From the SCADE functional specifications, C code is automatically generated, though this transformation (SCADE to C) is not standardized. This graphical modeling environment is used mainly in the aerospace field (Airbus, DO-178B); however its capabilities serve also transportation, automotive and energy.

In major industrial applications, the testing process usually consists in producing test cases based on the functional requirements of the system under test. Test objectives and test data are constructed with regard to the system requirements and the coverage evaluation is applied on the generated C code. For programs written in sequential languages, several adequacy criteria have been presented in the past, such as path/branch coverage criteria, LCSAJ (Linear Code Sequence And Jump) [10] and MC/DC (Modified Decision Condition Coverage).

These criteria are not conformed with the synchronous paradigm and cannot be applied on LUSTRE programs to assess how thoroughly the produced test data have tested the corresponding specification. Furthermore, it is difficult to formally relate the coverage measurement results with the system specification and the test objective. To deal with this problem, especially designed structural coverage criteria for LUSTRE programs have been proposed [7]. Although these criteria are comparable to the existing data-flow based criteria [9,3], they are not the same. They aim at defining intermediate coverage objectives and estimating the required test effort towards the final one. These criteria are based on the notion of the *activation condition* of a path, which informally represents the propagation of the effect of the input edge through the output edge.

However, the above coverage criteria can be applied only on specifications that are defined under a unique global clock. The global clock is a boolean flow that always values *true* and defines the frequency of the program execution cycles. Other, slower clocks can be defined through boolean-valued flows. They are mainly used to prevent useless operations of the program and to save computational resources by forcing some program expressions to be evaluated strictly on specific execution cycles. Thus, nested clocks may be used to restrict the operation of certain flows when this is necessary, without affecting at the same time the rest of the program variables. In LUSTRE, using multiple clocks is possible through two specific operators, `when` and `current`. In this paper, we propose the extension of the existing coverage criteria taking into account the `when` and `current` operators. In fact, we define the activation conditions for the paths

containing these operators in order that the coverage criteria are applicable on such paths. The complexity of the criteria, in terms of the cost of computing the paths and their activation conditions, is not increased.

The paper is structured in three main sections. Section 2 provides a brief overview of the essential concepts on LUSTRE language. Section 3 presents the existing coverage criteria for LUSTRE programs while in section 4 we thoroughly demonstrate their extension to the use of multiple clocks. Section 5 concludes and shows some perspectives for future work.

2 Overview of LUSTRE

LUSTRE [5] is a data-flow language. Contrary to imperative languages which describe the control flow of a program, LUSTRE describes the way that the inputs are turned into the outputs. Any variable or expression is represented by an infinite sequence of values and take the n-th value at the n-th cycle of the program execution, as it is shown in Figure 1. At each tick of a global clock, all inputs are read and processed simultaneously and all outputs are emitted, according to the synchrony hypothesis.

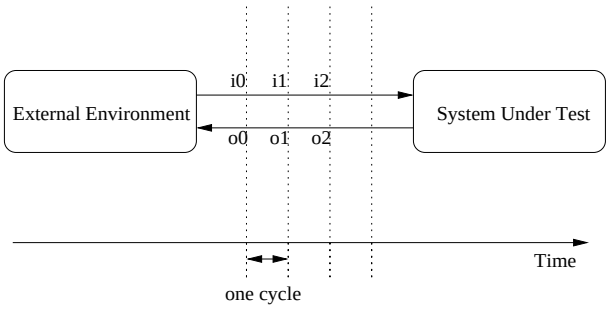

Fig. 1. Synchronous software operation

A LUSTRE program is structured into nodes. A node is a set of equations which define the node outputs as a function of its inputs. Each variable can be defined only once within a node and the order of equations is of no matter. Specifically, when an expression E is assigned to a variable X, X=E, that indicates that the respective sequences of values are identical throughout the program execution; at any cycle, X and E have the same value. Once a node is defined, it can be used inside other nodes like any other operator.

The operators supported by LUSTRE are the common arithmetic and logical operators (`+`, `-`, `*`, `/`, `and`, `or`, `not`) as well as two specific temporal operators: the *precedence* (`pre`) and the *initialization* (`->`). The `pre` operator introduces to the flow a delay of one time unit, while the `->` operator -also called *followed by* (`fby`)- allows the flow initialization. Let $X = (x_0, x_1, x_2, x_3, \ldots)$ and $E =$

$(e_0, e_1, e_2, e_3, \ldots)$ be two LUSTRE expressions. Then `pre(X)` denotes the sequence $(nil, x_0, x_1, x_2, x_3, \ldots)$, where *nil* is an undefined value, while `X->E` denotes the sequence $(x_0, e_1, e_2, e_3, \ldots)$.

LUSTRE does not support loops (operators such as `for` and `while`) nor recursive calls. Consequently, the execution time of a LUSTRE program can be statically computed and the satisfaction of the synchrony hypothesis can be checked.

A simple LUSTRE program is given in Figure 2, followed by an instance of its execution. This program has a single input boolean variable and a single boolean output. The output is *true* if and only if the input has never been *true* since the beginning of the program execution.

```
node Never(A: bool) returns (never_A: bool);
let
  never_A = not(A) -> not(A) and pre(never_A);
tel;
```

	c_1	c_2	c_3	c_4	...
A	*false*	*false*	*true*	*false*	...
never_A	*true*	*true*	*false*	*false*	...

Fig. 2. Example of a LUSTRE node.

2.1 Operator Network

The transformation of the inputs into the outputs in a LUSTRE program is done via a set of operators. Therefore, it can be represented by a directed graph, the so called *operator network*. An operator network is a graph with a set of N operators which are connected to each other by a set of $E \subseteq N \times N$ directed edges. Each operator represents a logical or a numerical computation. With regard to the corresponding LUSTRE program, an operator network has as many input edges (respectively, output edges) as the program input variables (respectively, output variables).

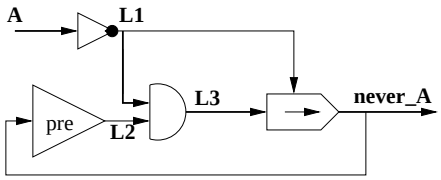

Fig. 3. The operator network for the node `Never`.

Figure 3 shows the corresponding operator network for the node of Figure 2.

An operator represents a data transfer from an input edge into an output edge. There are two kinds of operators:

a) the basic operators which correspond to a basic computation and

b) the compound operators which correspond to the case where in a program, a node calls another node[1].

A basic operator is denoted as $\langle e_i, s \rangle$, where e_i, $i = 1, 2, 3, \ldots$, stands for its inputs edges and s stands for the output edge.

2.2 Clocks in LUSTRE

In LUSTRE, any variable and expression denotes a flow, i.e. each infinite sequence of values is defined on a clock, which represents a sequence of time. Thus, a flow is the pair of a sequence of values and a clock.

The clock serves to indicate when a value is assigned to the flow. That means that a flow takes the n-th value of its sequence of values at the n-th time of its clock. Any program has a cyclic behavior and that cycle defines a sequence of times, i.e. a clock, which is the *basic clock* of a program. A flow on the basic clock takes its n-th value at the n-th execution cycle of the program. Slower clocks can be defined through flows of boolean values. The clock defined by a boolean flow is the sequence of times at which the flow takes the value *true*.

Two operators affect the clock of a flow: `when` and `current`.

`when` is used to **sample** an expression on a slower clock. Let E be an expression and B a boolean expression with the same clock. Then `X=E when B` is an expression whose clock is defined by B and its values are the same as those of E's only when B is *true*. That means that the resulting flow X has not the same clock with E or, alternatively, when B is *false*, X is not defined at all.

`current` operates on expressions with different clocks and is used to **project** an expression on the immediately faster clock. Let E be an expression with the clock defined by the boolean flow B which is not the basic clock. Then `Y=current(E)` has the same clock as B and its value is the value of E at the last time that B was *true*. Note that until B is *true* for the first time, the value of Y will be *nil*.

The sampling and the projection are two complementary operations: a projection changes the clock of a flow to the clock that the flow had before its last sampling operation. Trying to project a flow that was not sampled produces an error. Table 1 provides the use of the two temporal LUSTRE operators in more details.

---

[1] For the time being, we only consider basic operators.

E	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	...
B	*false*	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*	...
X=E when B			$x_0 = e_2$		$x_1 = e_4$			$x_2 = e_7$	$x_3 = e_8$	...
Y=current(E)	$y_0 = nil$	$y_1 = nil$	$y_2 = e_2$	$y_3 = e_2$	$y_4 = e_4$	$y_5 = e_4$	$y_6 = e_4$	$y_7 = e_7$	$y_8 = e_8$	...

Table 1. The use of the operators `when` and `current`.

An example [4] of the use of clocks in LUSTRE is given in Figure 4.

```
node mux(m:int) returns (c:bool; y:int);
var (x:int) when c;
let
  y = if c then current(x) else pre(y)-1;
  c = true -> (pre(y)=0);
  x = m when c;
tel;
```

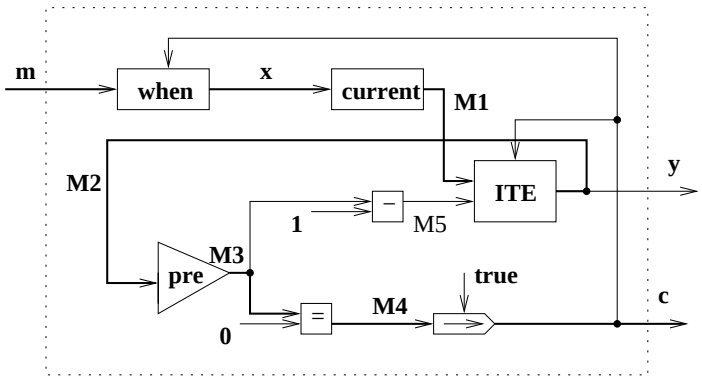

Fig. 4. The `mux` example and the corresponding operator network.

The LUSTRE node `mux` receives as input the signal m. Starting from this input value when the clock c is true, the program counts backwards until zero; from this moment, it restarts from the current input value and so on.

3 Coverage Criteria for LUSTRE programs

3.1 Activation Conditions

Given an operator network N, paths can be defined in the program. That is, the possible directions of flows from the input through the output. More formally, a

path is a finite sequence of edges $\langle e_0, e_1, \ldots, e_n \rangle$, such that for $\forall i \epsilon\, [0, n-1]$, e_{i+1} is a successor of e_i in N. A *unit path* is a path with two successive edges. For instance, in the operator network of Figure 3, there can be found the following paths.

$p_1 = \langle A, L_1, never\_A \rangle$
$p_2 = \langle A, L_1, L_3, never\_A \rangle$
$p_3 = \langle A, L_1, never\_A, L_2, L_3, never\_A \rangle$
$p_4 = \langle A, L_1, L_3, never\_A, L_2, L_3, never\_A \rangle$

Obviously, one could discover infinitely many paths in an operator network depending on the number of cycles repeated in the path (i.e. the number of `pre` operators in the path). However, we only consider paths of finite length by limiting the number of cycles. That is, a path of length n is obtained by concatenating a path of length *n-1* with a unit path (of length 2). Thus, beginning from unit paths, longer paths could be built; a path is finite if it contains no cycles or if the number of cycles is limited.

A boolean LUSTRE expression is associated with each pair $\langle e, s \rangle$, denoting the condition on which the data flows from the input edge e through the output s. This condition is called *activation condition*. The evaluation of the activation condition depends on what kind of operators the paths is composed of. Informally, the notion of the activation of a path is strongly related to the propagation of the effect of the input edge through the output edge. More precisely, a path activation condition shows the dependencies between the path inputs and outputs. Therefore, the selection of a test set satisfying the paths activation conditions in an operator network leads to a notion for the program coverage. Since covering all the paths in an operator network could be impossible, because of their potentially infinite number and length, in our approach, coverage is defined with regard to a given path length.

Table 2 summarizes the formal expressions of the activation conditions for all LUSTRE operators (except for `when` and `current` for the moment). In this table, each operator `op`, with the input e and the output s, is paired with the respective activation condition $AC\,(e, s)$ for the unit path $\langle e, s \rangle$. Noted that some operators may define several paths through their output, so the activation conditions are listed according to the path inputs.

Let us consider the path $p_2 = \langle A, L_1, L_3, never\_A \rangle$ in the corresponding operator network for the node `Never` (Figure 3). The condition under which that path is activated is represented by a boolean expression showing the propagation of the input A through the output $never\_A$. To calculate its activation condition, we progressively apply the rules for the activation conditions of the corresponding operators according to Table 2[2]. Starting from the end of the path, we reach

[2] In the general case (path of length n), the path p containing the `pre` operator is activated if its prefix p' is activated at the previous cycle of execution, that is $AC\,(p) = false$ `->` $pre\,(AC\,(p'))$. Similarly in the case of the initialization operator `fby`, the given activation conditions are respectively generalized in the forms: $AC\,(p) = AC\,(p')$ `->` $false$ (i.e. the path p is activated if its prefix p' is activated

Operator	Activation condition
$s = NOT\,(e)$	$AC\,(e, s) = true$
$s = AND\,(a, b)$	$AC\,(a, s) = not\,(a)\ or\ b$ $AC\,(b, s) = not\,(b)\ or\ a$
$s = OR\,(a, b)$	$AC\,(a, s) = a\ or\ not\,(b)$ $AC\,(b, s) = b\ or\ not\,(a)$
$s = ITE\,(c, a, b)$	$AC\,(c, s) = true$ $AC\,(a, s) = c$ $AC\,(b, s) = not\,(c)$
relational operator	$AC\,(e, s) = true$
$s = FBY\,(a, b)$	$AC\,(a, s) = true$ -> $false$ $AC\,(b, s) = false$ -> $true$
$s = PRE\,(e)$	$AC\,(e, s) = false$ -> $pre\,(true)$

Table 2. Activation conditions for all LUSTRE operators.

the beginning, moving one step at a time along the unit paths. Therefore, the necessary steps would be the following:

$AC\,(p_2) = false$ -> $AC\,(p')$, where $p' = \langle A, L_1, L_3\rangle$
$AC\,(p') = not\,(L_1)\ or\ L_2\ and\ AC\,(p'') = A\ or\ pre\,(never\_A)\ and\ AC\,(p'')$, where $p'' = \langle A, L_1\rangle$
$AC\,(p'') = true$

After backward substitutions, the boolean expression for the activation condition of the selected path is:

$$AC\,(p_4) = false \text{ -> } A\ or\ pre\,(never\_A).$$

In practice, in order for the path output to be dependent on the input, either the input has to be *true* at the current execution cycle or the output at the previous cycle has to be *true*; for the first cycle of the execution, the input needs to be *false*.

3.2 Coverage Criteria

A LUSTRE/SCADE program is compiled into an equivalent C program. Provided that the format of the generated C code depends on the compiler, it is hard to fix a formal relation between the original LUSTRE program and the final C one. In addition, major industrial standards, such as DO-178B in the avionics field, demand coverage to be measured on the generated C code. Therefore, three coverage criteria specifically defined for LUSTRE programs have been proposed [7]. They are specified on the operator network according to the length of the paths and the input variable values.

---

at the initial cycle of execution) and $AC\,(p) = false$ -> $AC\,(p')$ (i.e. the path p is activated if its prefix p' is always activated except for the initial cycle of execution).

Let $\mathcal{T}$ be the set of test sets (input vectors) and $P_n = \{p|length(p) \leq n\}$ the set of all paths in the operator network whose length is inferior or equal to n. Hence, the following families of criteria are defined for a given and finite order $n \geq 2$. The input of a path p is denoted as $in\,(p)$ whereas a path edge is denoted as e.

1. **Basic Coverage Criterion (BC).** This criterion is satisfied if there is a set of test input sequences, $\mathcal{T}$, that activates at least once the set P_n. Formally, $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $AC\,(p) = true$. The aim of this criterion is basically to ensure that all the dependencies between inputs and outputs have been exercised at least once. In case that a path is not activated, certain errors such as a missing or misplaced operator could not be detected.
2. **Elementary Conditions Criterion (ECC)**. In order that an input sequence satisfies this criterion, it is required that the path p is activated for both input values, *true* and *false* (taking into account that only boolean variables are considered). Formally, $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $in\,(p) \wedge AC\,(p) = true$ and $not\,(in\,(p)) \wedge AC\,(p) = true$. This criterion is stronger than the previous one in the sense that it also takes into account the impact that the input value variations have on the path output.
3. **Multiple Conditions Criterion (MCC).** In this criterion, the path output depends on all the combinations of the path edges, also including the internal ones. A test input sequence is satisfied if and only if the path activation condition is satisfied for each edge value along the path. Formally, $\forall p \in P_n$, $\forall e \in p$, $\exists t \in \mathcal{T}$: $e \wedge AC\,(p) = true$ and $not\,(e) \wedge AC\,(p) = true$.

The above criteria form a hierarchical relation: MCC satisfies all the conditions that ECC does, which also subsumes BC.

4 Extension of coverage criteria to `when` and `current` operators

The aim of this paper is to extend the above criteria in order to support the two temporal LUSTRE operators `when` and `current`, which handle the use of multiple clocks since this is the case for many industrial applications.

The use of multiple clocks implies the filtering of some program expressions. It consists in changing their execution cycle, activating it only at certain cycles of the basic clock. Consequently, the associated paths are activated only if the respective clock is true. As a result, the tester must adjust this rarefied path activation rate according to the global *timing*.

In this section, we present the definition for the path activation conditions for `when` and `current`, followed by their formal verification. Then, we demonstrate the application of the extended criteria as well as the coverage evaluation, using the simple example of the inverse counter of Section 2.2.

4.1 Activation Conditions for `when` and `current`

Informally, the activation conditions associated with the `when` and `current` operators are based on their intrinsic definition. Since the output values are defined according to a condition (i.e. the *true* value of the clock), these operators can be represented by means of the conditional operator `if-then-else`. For the expression E and the boolean expression B with the same clock,

- `X=E when B` could be seen as `X=if B then E else NON_DEFINED` and similarly,
- `Y=current(X)` could be seen as `Y=if B then X else pre(X)`.

Hence, the formal definitions of the activation conditions result as follows:

Definition 1. *Let e and s be the input and output edges respectively of a* `when` *operator and let b be its clock. The activation conditions for the paths $p_1 = \langle e, s\rangle$ and $p_2 = \langle b, s\rangle$ are:*
$AC(p_1) = b$
$AC(p_2) = true$

Definition 2. *Let e and s be the input and output edges respectively of a* `current` *operator and let b be the clock on which it operates. The activation condition for the path $p = \langle e, s\rangle$ is:*
$AC(p) = b$

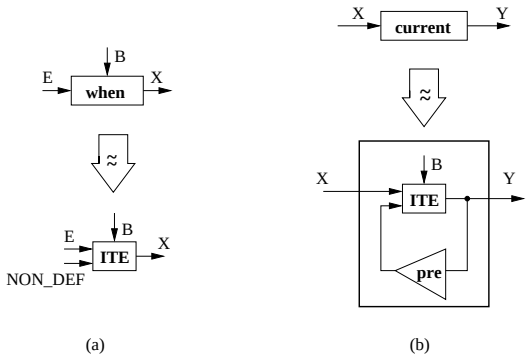

Fig. 5. Modeling the `when` and `current` operators using the `ITE`.

As a result, to compute the paths and the associated activation conditions of a LUSTRE node involving several clocks, one has just to replace the `when` and `current` operators by the corresponding conditional operator (see Figure 5). At this point, two basic issues need to be farther clarified. The first one concerns the

`when` case. Actually, there is no way of defining the value of the expression X when the clock B is not *true* (branch *NON_DEF* in Figure 5(a)). By default, at these instants, X does not occur and such paths (beginning with a non defined value) are infeasible[3]. In the `current` case, the operator implicitly refers to the clock parameter B, without using a separate input variable (see Figure 5(b)). This hints at the fact that `current` always operates on an already sampled expression, so the clock that determines its output activation should be the one on which the input is sampled.

Let us assume the path $p = \langle m, x, M_1, M_2, M_3, M_4, c\rangle$ in the example of Section 2.2, displayed in bold in Figure 4. Following the same procedure for the activation condition computation and starting from the last path edge, the activation conditions for the intermediate unit paths are:

$AC(p) = false$ -> $AC(p_1)$, where $p_1 = \langle m, x, M_1, M_2, M_3, M_4\rangle$
$AC(p_1) = true\ and\ AC(p_2)$, where $p_2 = \langle m, x, M_1, M_2, M_3\rangle$
$AC(p_2) = false$ -> $pre(AC(p_3))$, where $p_3 = \langle m, x, M_1, M_2\rangle$
$AC(p_3) = c\ and\ AC(p_4)$, where $p_4 = \langle m, x, M_1\rangle$
$AC(p_4) = c\ and\ AC(p_5)$, where $p_5 = \langle m, x\rangle$
$AC(p_5) = c$

After backward substitutions, the activation condition of the selected path is:

$$AC(p) = false \text{ -> } pre(c).$$

This condition corresponds to the expected result and is compliant with the above definitions, according to which the clock must be true to activate the paths with `when` and `current` operators.

In order to evaluate the impact of these temporal operators on the coverage assessment, we consider the operator network of Figure 4 and the paths:
$p_1 = \langle m, x, M_1, y\rangle$
$p_2 = \langle m, x, M_1, M_2, M_3, M_4, c\rangle$
$p_3 = \langle m, x, M_1, M_2, M_3, M_5, y\rangle$

Intuitively, if the clock c holds true, any change of the path input is propagated through the output, hence the above paths are activated. Formally, the associated activation conditions to be satisfied by a test set are:

$AC(p_1) = c$

$AC(p_2) = false$ -> $pre(c)$

$AC(p_3) = not(c)\ and\ false$ -> $pre(c)$.

[3] An infeasible path is a path which is never executed by any test cases, hence it can never be covered.

Eventually, the input test sequences satisfy the basic criterion. Indeed, as soon as the input m causes the clock c to take the suitable values, the activation conditions are satisfied, since the latter depend only on the clock. In particular, in case that the value of m at the first cycle is an integer different to zero (for sake of simplicity, let us consider $m = 2$), the BC is satisfied in two steps since the corresponding values for c are c=*true*, c=*false*. On the contrary, if at the first execution cycle m equals to zero, the basic criterion is satisfied after three steps with the corresponding values for c: c=*true*, c=*true*, c=*false*. These two samples of input test sequences and the corresponding outputs are shown in Table 3.

	c_1	c_2	c_3	c_4	...
m	$i_1 (\neq 0)$	i_2	i_3	i_4	...
c	*true*	*false*	*false*	*true*	...
y	i_1	$i_1 - 1$	0	i_4	...

	c_1	c_2	c_3	c_4
m	$i_1 (= 0)$	i_2	i_3	...
c	*true*	*true*	*false*	...
y	0	i_2	$i_2 - 1$	...

Table 3. Test cases samples for the input m.

4.2 An illustrative example

Let us consider a LUSTRE node that receives at the input a boolean signal *set* and returns at the output a boolean signal *level*. The latter must be true during *delay* cycles after each reception of *set*. Now, suppose that we want the *level* to be high during *delay* seconds, instead of *delay* cycles. Taking advantage of the use of the `when` and `current` operators, we could call the above node on a suitable clock by filtering its inputs. The second must be provided as a boolean input *second*, which would be *true* whenever a second elapses. The node must be activated only when either a *set* signal or a *second* signal occurs and in addition at the initial cycle, for initialization purposes. The LUSTRE code is quite simple and it is shown in Figure 6, followed by the associated operator network[4].

Similarly to the previous example, the paths to be covered are:
$p_1 = \langle set, T_2, T_3, T_9, level \rangle$
$p_2 = \langle delay, T_8, T_3, T_9, level \rangle$
$p_3 = \langle set, T_1, ck, T_2, T_3, T_9, level \rangle$
$p_4 = \langle second, T_1, ck, T_2, T_3, T_9, level \rangle$
$p_5 = \langle set, T_1, ck, T_8, T_3, T_9, level \rangle$
$p_6 = \langle second, T_1, ck, T_8, T_3, T_9, level \rangle$

[4] The nested node `STABLE` is used unfolded, since with this criteria definition, the dependencies between a called node inputs and outputs cannot be determined.

```
node TIME_STABLE(set, second: bool; delay: int) returns
(level: bool);
var ck: bool;
let
  level = current(STABLE((set, delay) when ck));
  ck = true -> set or second;
tel;

node STABLE(set: bool; delay: int) returns (level: bool);
var count: int;
let
  level = (count>0);
  count = if set then delay
        else if false->pre(level) then pre(count)-1
        else 0;
tel;
```

Fig. 6. The node `TIME_STABLE`: a simple example with the `when` and `current` operators.

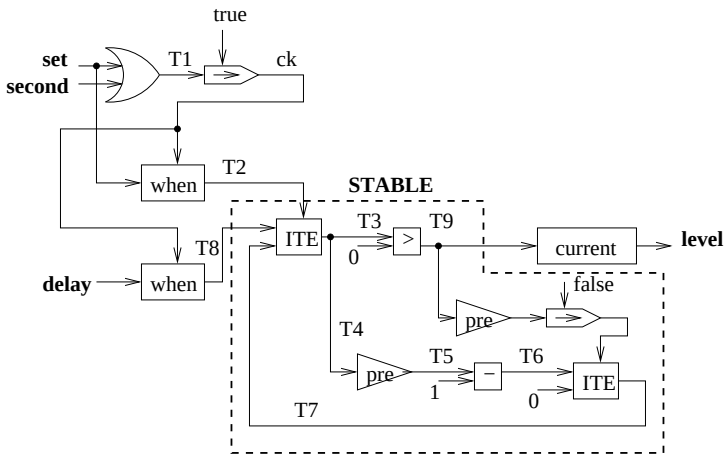

Fig. 7. The operator network for the node `TIME_STABLE`.

To cover all these paths, one has to select a test set satisfying the following activation conditions, calculated as it is described above:

$AC(p_1) = ck$, where $ck = true$ -> $set\ or\ second$
$AC(p_2) = ck\ and\ set$
$AC(p_3) = ck\ and\ second\ or\ not\ (set)$
$AC(p_4) = ck\ and\ false$ -> $second\ or\ not\ (set)$
$AC(p_5) = ck\ and\ set\ and\ false$ -> $set\ or\ not\ (second)$
$AC(p_6) = ck\ and\ set\ and\ false$ -> $second\ or\ not\ (set)$

Since the code ensures the correct initialization of the clock, hence its activation at the first cycle, the above paths are always activated at the first execution cycle. For the rest of the execution, the basic criterion is satisfied with the following test sequence for the inputs $(set, second)$: $(1,0)$,$(0,1)$, $(1,1)$. This test set, which contains almost every possible combination of the inputs, satisfies also the elementary conditions criterion (ECC), since the activation of the paths depends on both boolean inputs.

Admittedly, the difficulty to meet the criteria is strongly related to the complexity of the system under test as well as to the test case generation effort. Moreover, activation conditions covered with short input sequences are easy to be satisfied, as opposed to long test sets that correspond to complex instance executions of the system under test. Experimental evaluation on more complex case studies, including industrial software components, is necessary and part of our future work in order to address these problems. Nonetheless, the enhanced definitions of the structural criteria presented in this paper complete the coverage assessment issue for LUSTRE programs, as all the language operators are supported. In addition, the complexity of the criteria is not further affected, because, in substance, we use nothing but `if-then-else` operators.

5 Conclusion

We presented the extension of the LUSTRE structural coverage criteria to support the use of multiple clocks. We defined the activation conditions for the temporal operators `when` and `current`, which are used to affect the clock of a LUSTRE expression. We applied the results on suitable examples and we described how the criteria could be employed. Yet, the research work presented in this paper needs to be implemented and incorporated in LUSTRUCTU, a tool which measures the structural coverage of LUSTRE programs.

In SCADE, coverage is measured through the Model Test Coverage (MTC) module, in which the user can define his own criteria by defining the conditions to be activated during testing. Thus, our work could be easily integrated in SCADE in the sense that activation conditions corresponding to the defined criteria (BC, ECC, MCC) could be assessed once they are transformed into suitable MTC

expressions. These issues are currently investigated within the framework of a collaborative research project[5].

Future work includes the evaluation of the proposed criteria involving industrial case studies. Furthermore, it is necessary to analyze the test sets to determine their ability to satisfy the criteria and observe what happens with the paths that the tests cannot cover.

Integration testing issues are also under study. In case of long paths to be covered, the total path number highly increases causing the coverage measures to be non applicable. As a result, integration testing requires extending the definition of the activation conditions to internal nodes, that is to the operators that the user can define. Such an extension should make it possible to apply the code coverage criteria on LUSTRE nodes that call other nodes (compound operators) without having to unfold the latter ones and reducing the overall complexity.

References

1. Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
2. F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, 1991.
3. Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Eng.*, 15(11):1318–1332, 1989.
4. Alain Girault and Xavier Nicollin. Clock-driven automatic distribution of lustre programs. In *3rd International Conference on Embedded Software, EMSOFT'03*, volume 2855 of *LNCS*, pages 206–222, Philadelphia, USA, October 2003. Springer-Verlag.
5. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
6. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.
7. A. Lakehal and I. Parissis. Structural test coverage criteria for lustre programs. *In the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), a joint event of ESEC/FSE'05*, pages 35–43, Lisbon, Portugal, September 2005.
8. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming Real-Time Applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.
9. Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, 11(4):367–375, 1985.
10. M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng.*, 6(3):278–286, 1980.

[5] SIESTA project (www.siesta-project.com), funded by the French National Research Agency.