Structural Test Coverage Criteria for Integration Testing of LUSTRE/SCADE Programs*

Virginia Papailiopoulou[1], Ajitha Rajan[2], and Ioannis Parissis[3]

[1] INRIA Rocquencourt
[2] Université Joseph Fourier - Laboratoire d'Informatique de Grenoble
[3] Grenoble INP - Laboratoire de Conception et d'Intègration des Systèmes
virginia.papailiopoulou@inria.fr, ajitha.rajan@imag.fr,
ioannis.parissis@grenoble-inp.fr

Abstract. LUSTRE is a formal synchronous declarative language widely used for modeling and specifying safety-critical applications in the fields of avionics, transportation, and energy production. In such applications, the testing activity to ensure correctness of the system plays a crucial role in the development process. To enable adequacy measurement of test cases over applications specified in LUSTRE (or SCADE), a hierarchy of structural coverage criteria for LUSTRE programs has been recently defined. A drawback with the current definition of the criteria is that they can only be applied for unit testing, i.e., to single modules without calls to other modules. The criteria experiences scalability issues when used over large systems with several modules and calls between modules. We propose an extension to the criteria definition to address this scalability issue. We formally define the extension by introducing an operator to abstract calls to other modules. This extension allows coverage metrics to be applied to industrial-sized software without an exponential blowup in the number of activation conditions. We conduct a preliminary evaluation of the extended criteria using an Alarm Management System.

1 Introduction

LUSTRE [4,1] is a synchronous dataflow language widely used to model reactive, safety-critical applications. It is based on the synchronous approach where the software reacts to its inputs instantaneously. LUSTRE also forms the kernel language for the commercial toolset, SCADE (Safety Critical Application Development Environment[1]), used in the development of safety-critical embedded software. SCADE has been used in several important European avionic projects such as AIRBUS A340-600, A380, EUROCOPTER.

The verification and validation activity ensuring the correctness of the system is very important for applications in the safety-critical systems domain. For applications modeled in the LUSTRE language, this concern has been addressed

* This research work is supported by the SIESTA project (www.siesta-project.com), funded by ANR, the French National Research Foundation.

[1] www.esterel-technologies.com

G. Salaün and B. Schätz (Eds.): FMICS 2011, LNCS 6959, pp. 85–101, 2011.

either by means of formal verification methods [5] or using automated testing approaches [7,9]. State space explosion still poses a challenge for applying formal verification techniques to industrial applications. Hence, developers use intense testing approaches to gain confidence in the system correctness.

To determine whether the testing process can terminate, several test adequacy criteria based on the program control flow have been used in the past, such as branch coverage, LCSAJ (Linear Code Sequence And Jump) [10] and MCDC (Modified Condition Decision Coverage) [2]. These criteria do not conform to the synchronous data-flow paradigm and when applied do not provide meaningful information on the Lustre specification. To tackle this problem, Lakehal et al.[6] and Papailiopoulou et al.[8] defined structural coverage criteria specifically for LUSTRE specifications. The structural coverage criteria for LUSTRE are based on *activation conditions* of paths. When the activation condition of a path is *true*, any change in the path entry value causes modification of the path exit value within a finite number of time steps.

The LUSTRE coverage criteria defined in [6] and [8] can only be used for unit testing purposes, i.e. adequacy can only be measured over a single LUSTRE node (program units). To measure adequacy over Lustre nodes with calls to other nodes, the calls would have to be inline expanded for coverage measurement. Doing this results in an exponential increase in the number of paths and activation conditions to be covered. As a result, the current definition of the criteria does not scale to large and complex systems.

To tackle this scalability issue, we extend the definition of Lustre coverage criteria to support node integration. In our definition, we use an abstraction technique that replaces calls to nodes with a new operator, called the *NODE* operator. This abstraction avoids the entire set of paths of the called node from being taken into account, and instead replaces it with a unit path through the *NODE* operator. We evaluated the extended criteria over an Alarm Management System used in the field of avionics.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the essential concepts of the LUSTRE language and the existing coverage criteria defined for LUSTRE programs. In Section 3, we define the extended coverage criteria for integration testing, and Section 4 presents a preliminary evaluation.

2 Background

2.1 Overview of LUSTRE

A LUSTRE program is structured into nodes. Nodes are self-contained modules with inputs, outputs, and internally-declared variables. A node contains an unordered set of equations that define the transformation of node inputs into outputs through a set of operators. Once a node is defined, it can be called (or instantiated) within other nodes like any other expression in the specification.

In addition to common arithmetic and logical operators (`+`, `-`, `*`, `/`, `and`, `or`, `not`), LUSTRE supports two temporal operators: *precedence* (`pre`) and *initialization*

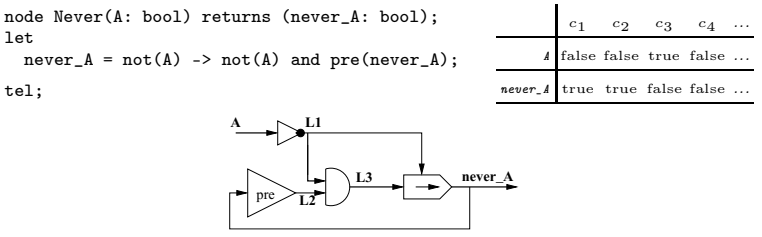

Fig. 1. A LUSTRE node and its operator network

(->). The `pre` operator introduces a delay of one time unit, while the -> operator, also called *followed by* (`fby`), allows the initialization of a sequence. 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 $(x_0, e_1, e_2, e_3, \ldots)$.

Figure 1 illustrates a simple LUSTRE program and an instance of its execution. This program has a single input boolean variable (`A`) and a single output boolean variable (`never_A`). The output is *true* if and only if the input has never been *true*. Lustre nodes are usually represented by an *operator network*, a labeled graph connecting operators with directed edges. An operator in the network specifies data-flow transfers from inputs to outputs. An edge specifies the data-flow between two operators. There are three kinds of edges: input, output and internal edges. Input (or output) edges are occurrences of input (or output) variables of the LUSTRE node. Internal edges correspond to occurrences of local variables. The operators *not, and, pre,* and *followed by* in the operator network of Figure 1 are connected so that they correspond exactly to the sequence of operations in the Lustre node *Never* in Figure 1. At the first execution cycle, the output `never_A` from the *followed by* operator is the input edge L1 (negation of input `A`). For the remaining execution cycles, the output is the input edge L3 (conjunction of previous `never_A` and the negation of `A`).

2.2 LUSTRE Coverage Criteria

Given an operator network N, paths in the program can be viewed as possible directions of flows from the input to the output. 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 $\langle e_i, e_{i+1} \rangle$. For instance, in the operator network of Figure 1, we can find the following paths.

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

All these paths are *complete paths*, because they connect a program input with a program output; the paths p_1 and p_2 are *elementary paths*, because they contain no cycles while the paths p_3 and p_4 contain one cycle[2]. Lakehal et al.[6]

[2] Cyclic paths contain one or more pre operators.

Table 1. Activation conditions for Boolean/Relational/Conditional operators

Operator	Activation condition
$s = NOT\,(e)$	$AC\,(e, s) = true$
$s = AND\,(a, b)$	$AC\,(a, s) = not\,(a)\ or\ b$; $AC\,(b, s) = not\,(b)\ or\ a$
$s = OR\,(a, b)$	$AC\,(a, s) = a\ or\ not\,(b)$; $AC\,(b, s) = b\ or\ not\,(a)$
$s = ITE\,(c, a, b)$	$AC\,(c, s) = true$; $AC\,(a, s) = c$; $AC\,(b, s) = not\,(c)$
$s = op\,(e)$, where $op\epsilon\,\{<, >, \leq, \geq, =\}$	$AC\,(e, s) = true$
$s = op\,(a, b)$, where $op\epsilon\,\{+, -, \star, /\}$	$AC\,(a, s) = true$ $AC\,(b, s) = true$

Table 2. Activation condition definitions using path prefix p' and last operator

Last Operator	Activation condition
$Boolean/Relational/Conditional$	$AC\,(p) = AC(p')$ and $OC(e_{n-1}, e_n)$
$FBY\,(init, nonInit)$	$AC\,(p) = AC(p')$ -> $false$ for initial cycle $AC\,(p) = false$ -> $AC(p')$ for all but the initial cycle
$PRE\,(e)$	$AC\,(p) = false$ -> $pre\,(AC(p'))$

only considered paths of finite length. A path is considered finite if it contains no cycles or if the number of cycles is limited.

Lakehal et al. associate a notion of an *activation condition* with each path. When the activation condition of a path is true, any change in the path entry value causes eventually the modification of the path exit value. A path is activated if its activation condition has been true at least once during an execution. Table 1 summarizes the formal expressions of the activation conditions for boolean, relational, and conditional LUSTRE operators. In this table, each expression $s = op\,(e)$ in the first column, that uses the operator `op` with the input e and output s, is paired with the respective activation condition $AC\,(e, s)$ for the unit path $\langle e, s\rangle$ formed with operator `op`. Activation condition for a unit path with the `NOT` operator is always *true*, since the output is always affected by changes in the input. For operators with more than one input, such as $AND(a, b)$, there will be more than one unit path –$\langle a, s\rangle$and$\langle b, s\rangle$– from each input to the output. The activation conditions for such operators are listed from each operator input to each output. To exemplify, consider the activation conditions for $AND(a, b)$ –$AC(a, s)$, and $AC(b, s)$. The activation condition for the unit path from input a to output s, $AC(a, s)$, is true when the effect of input a propagates to the output s. This happens when input a is $false$, in which case the output is $false$ regardless of the value of input b. It also occurs when input b is *true*, the output value solely depends on the value of input a. As a result, $AC(a, s) = not(a)\ or\ b$. The activation condition, $AC(b, s)$, from input b is computed in a similar fashion.

For a given path p of length n, $\langle e_1, ..., e_{n-1}, e_n\rangle$, the activation condition $AC(p)$ is recursively defined as a function of the last operator op in it ($e_{n-1} \in$

$in(op)$ and $e_n \in out(op)$) and its path prefix p' of length $n-1$,$\langle e_1, ..., e_{n-1}\rangle$. If p is a single edge ($n = 1$), $AC(p)$ is *true*. Table 2 presents definitions of $AC(p)$ based on the last operator *op* and the path prefix p'. Let $OC(e_{n-1}, e_n)$ be the activation condition associated with operator *op* on the unit path (e_{n-1}, e_n).

To illustrate the activation condition computation, consider the path $p_2 = \langle A, L_1, L_3, never\_A\rangle$ in the operator network for the node `Never` in Figure 2. Path p_2 and its sub paths are illustrated in Figure 2. The activation condition for p_2 will be a boolean expression that gives the condition under which effect of input A progresses to output $never\_A$. To calculate this condition, we progressively apply the rules for the activation conditions of the corresponding operators in Table 1 and Table 2. Equations (1), (2), (3), and (4) shown below along with Figure2 illustrate this computation. We start from the end of the path and progress towards the beginning, moving one step at a time along the unit paths.

$$AC\,(p_2) = false \rightarrow AC\,(p')\ \ where\ p' = \langle A, L_1, L_3\rangle\,. \tag{1}$$

$$AC(p') = not(L_1)\ or\ pre(never\_A)\ and\ AC(p'') \tag{2}$$

$$AC(p'') = true \tag{3}$$

Upon backward substitution of values for AC(p') and AC(p'') from Equations 2 and 3, respectively, into Equation 1 we get

$$AC\,(p_2) = false \rightarrow A\,or\,pre\,(never\_A) \tag{4}$$

The activation condition for path p_2 in Equation 4 states that at the first execution cycle, the path p_2 is not activated. For the remaining cycles, to activate path p_2 either the input A needs to be *true* at the current execution cycle or the output at the previous cycle needs to be *true*.

Criteria Definition. Lakehal et al.[6] defined three coverage criteria for Lustre/SCADE programs and implemented a coverage measurement tool called Lustructu based on the definitions. The coverage definitions are for a given finite path length. To understand our contributions in extending the criteria later in the paper, we present the formal definitions of the criteria from [6].

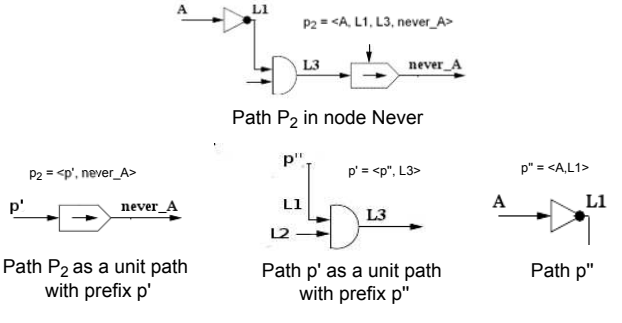

Fig. 2. Path p_2 and its sub paths in node `Never`

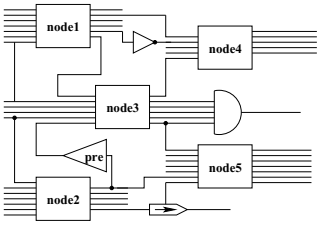

Fig. 3. Example of a complex LUSTRE program

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 less than or equal to n. The input of a path p is denoted as $in\,(p)$ and a path edge is denoted as e. A family of criteria were defined by Lakehal et al. for a finite path length n:

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 to ensure that all dependencies between inputs and outputs have been exercised at least once.

Elementary Conditions Criterion (ECC). In order to satisfy this criterion for a path p, it is required that the path p be activated for both values, *true* and *false,* of the input (taking only boolean input variables into consideration). Formally, $\forall p \in P_n$, $\exists t_1 \in \mathcal{T}$: $in\,(p) \wedge AC\,(p) = true$ and $\exists t_2 \in \mathcal{T}$: $not\,(in\,(p)) \wedge AC\,(p) = true$. This criterion is stronger than the basic criterion since it also takes into account the impact of input value variations on the output.

Multiple Conditions Criterion (MCC). This criterion checks whether the path output depends on all combinations of the path edges, including the internal ones. To satisfy this criterion, test input sequences need to ensure that the activation condition for each edge value along the path is satisfied. Formally, $\forall p \in P_n, \forall e \in p, \exists t_1 \in \mathcal{T}$: $e \wedge AC\,(p) = true$ and $\exists t_2 \in \mathcal{T}$: $not\,(e) \wedge AC\,(p) = true$.

For a given path length n, [6] shows that MCC_{n} subsumes ECC_{n} which in turn subsumes BC_{n}.

3 Integration Testing Approach

3.1 Motivation

For simple LUSTRE programs, coverage computation can be performed in a short amount of time. If the path length and number of paths is low, the number of activation conditions to be satisfied is also low, and unsatisfied conditions can be easily identified and analyzed by test designers. However, for complex Lustre programs, the number of activation conditions is usually high. This is particularly true for the MCC criterion, where the number of activation conditions to be satisfied increases dramatically with the length and the number of paths. As a result, coverage assessment and analysis become prohibitively expensive making

the criteria inapplicable. One of the primary reasons for the extremely high number of paths and activation conditions in complex Lustre programs is due to the presence of node calls. For instance, Figure 3 partially illustrates a complex node with calls to node1, node2, node3, node4, and node5. In order to measure coverage of the complex node in Figure 3 with the existing definition of Lustre criteria in [6], the node calls would have to be expanded into their definitions. The paths and the corresponding activation conditions are locally computed within each expanded node call, and these are then combined with expressions in the global node to compute the final activation conditions. This often results in a huge number of paths and activation conditions in the global node.

3.2 Path Activation Conditions

To help tackle this issue of intractable number of activation conditions, we define an approximation of the operator network by abstracting the called nodes. We replace calls to nodes with a new operator, `the NODE` operator, in the operator network. The inputs and the outputs of the `NODE` operator are the inputs and the outputs of the called node. This abstraction avoids the entire set of paths of the called node from being taken into account for coverage assessment. However, the abstraction is such that it is still possible to determine whether the output of a called node depends on a given input of the called node. More precisely, as illustrated in Figure 4, at the global level the set of paths beginning from an input e_i and ending at an output s_i, in the operator network of the called node, is represented by a single unit path $p = \langle e_i, s_i \rangle$ using the `NODE` operator. To compute the activation condition of p, we first compute the activation conditions of all paths from the edge e_i to the edge s_i in the called node. We then combine these activation conditions using the disjunction operator into a single activation condition for p.

To exemplify, consider the unit path $p2 = \langle e2, s2 \rangle$ at the global level in Figure 4. To compute the activation condition of $p2$ with the $NODE$ operator $NODE0$, we first compute the activation conditions of paths from the edge $e2$ to the edge $s2$ in the called node. Depending on the number of cycles we consider in the called node, the set of paths from $e2$ (input b) to $s2$ (output d) in the called node is $\{\langle b, d\rangle, \langle b, d, t1, d\rangle, \langle b, d, t1, d, t1, d\rangle, \langle b, t1, d, t1, d, t1, d\rangle ...\}$. We compute activation conditions of each of these paths; $\mathrm{AC}(\langle b, d\rangle)$, $\mathrm{AC}(\langle b, d, t1, d\rangle)$, and so forth. Finally, we compute the activation condition of the unit path $p2$ at the global level as the disjunction of each of these individual activation conditions (depending on the number of cycles considered), $AC(p2) = AC(\langle b, d\rangle) \vee AC(\langle b, d, t1, d\rangle) \vee AC(\langle b, d, t1, d, t1, d\rangle) \vee ...$

We would like readers to note that, even though the proposed abstraction requires the computation of activation conditions for paths in the called node, the computation effort is only increased by a linear amount (by the number of paths from the edge e_i to the edge s_i in the called node). The number of paths and activation conditions in the callee node is *not* affected by the number of paths from the edge e_i to the edge s_i in the called node since the abstraction represents them by a unit path. This, however, is not the case in the original

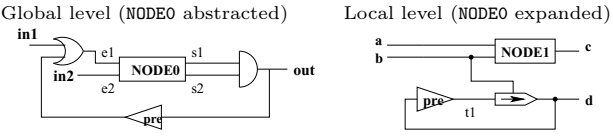

Fig. 4. A LUSTRE node using a compound operator

criteria definition. The original definition substitutes the *entire* operator network of the called node into the callee node, resulting in an exponential increase in the number of paths and activation conditions in the callee node. As a result, our approach still results in significant savings in effort required for coverage assessment. In Section 4, we illustrate the difference in number of activation conditions between the original criteria definitions and our proposed extension using a case study.

A called node may also contain calls to other nodes which in turn may call other nodes and so on. An example of such an occurrence is shown in Figure 4, where at the global level there is a call to NODE0 which in turn calls NODE1. As a result, we get a tree structure of called nodes and every level of this tree corresponds to a different *depth* of node integration. In Figure 4(a), let $AC^0(p)$ be the activation condition of $p = \langle e_1, s_1 \rangle$ at level zero. At level one, since `NODE0` calls `NODE1` (Figure 4(b)), the following expression describes the correlation of activation conditions between the different levels of integration:

$$AC^0(\langle e_1, s_1 \rangle) = AC^1(\langle a, c \rangle) \tag{5}$$

In general, at level (or depth) m, a path activation condition depends on the activation conditions at level $m+1$, as seen in the following definition.

Definition 1. *Let E and S be the sets of inputs and outputs respectively of a* `NODE` *operator at level m such that there is an input $e_i \in E$ and an output $s_i \in S$. Let $n>0$ be any positive integer and $p_1, p_2, \ldots, p_k$ be the paths from the input e_i to the output s_i the length of which is less than or equal to n. The* ***abstract activation condition*** *of the unit path $p = \langle e_i, s_i \rangle$ at depth m and path length n is defined as follows:*

$$AC_n^m(p) = AC_n^{m+1}(p_1) \vee AC_n^{m+1}(p_2) \vee \cdots \vee AC_n^{m+1}(p_k) \tag{6}$$

Similar to the activation condition definition for the basic operators, the abstract activation condition signifies the propagation of the effect of the input value to the output. Disjunction of the activation conditions of the involved paths, ensures that at least one of the dependencies of s_i on e_i is taken into account. In other words, if at least one path of length lower or equal to n in the called node is activated, then the unit path p for the `NODE` operator is activated, $AC_n^m(p) = true$, implying that the value of the output s_i is affected by the value of the input e_i of the `NODE` operator.

3.3 Extended Criteria Definition

We extend the original Lustre coverage criteria for unit testing[6] to support node integration using the abstraction technique discussed in the previous section. We replace calls to nodes with the `NODE` *operator*. We re-define the three classes of path-based criteria—BC, ECC, and MCC—to account for node integration. We use three input parameters in the definition of each criterion,

1. *Depth of integration:* This parameter determines the level at which abstraction of node calls using the NODE operator will begin (i.e where the abstract activation condition will be used). This parameter is not present in the definition of the original criteria.
2. *Maximum path length* from an input to an output: This parameter is used in the original criteria definition. However, the paths to be covered and their length vary depending on the depth of integration parameter.
3. *Number of cycles* to consider in paths: This parameter is used in the original criteria definition to indicate the number of cycles to take into account in paths whose length is at most equal to the maximum path length parameter.

Depth of Integration. Coverage analysis of a node using the parameter $depth = i$ indicates that called nodes up until depth i are inline expanded, while called nodes at depths i and deeper than i are abstracted using the `NODE` operator. The node being covered corresponds to depth 0. For instance, in Figure 5 that represents the call graph of an application with several embedded nodes (used in [6]), covering node #1 using the parameter $depth = 0$ implies that calls to all nodes starting from nodes #2, #3, and #4 (since they are called by node#1) will be abstracted using the `NODE` operator. If we set the depth of integration as $depth = 1$, it implies that calls to nodes #2, #3, and #4 will be inline expanded and all other nodes (that are in turn called by nodes #2, #3, and #4) will be replaced by the `NODE` operator. Developers can choose a value for the integration depth parameter according to the complexity of the system being covered. If no value is specified for this parameter, we use a default value of 0 corresponding to the common definition of integration testing (focusing on the current node and using abstractions for all called nodes).

Maximum Path Length. The maximum path length parameter used in the original criteria definition in [6] takes on a slightly new meaning in our criteria definition with the node operators and the integration depth parameter. For instance, in the node of Figure 4, for the paths with length 4, we can identify path $p = \langle in_2, e_2, s_2, out \rangle$ if we consider an integration depth of 0 (i.e. `NODE0` is abstracted using the node operator). However, if we use an integration depth of 1, `NODE0` will be unfolded. The previously considered path $p = \langle in_2, e_2, s_2, out \rangle$ with length 4 will now correspond to $p' = \langle in_2, b, d, out \rangle$ with a path length of 4 or to $p'' = \langle in_2, b, d, t_1, d, out \rangle$ with a path length of 6 if we consider one cycle. Additionally, the computation of the abstract activation conditions associated with a `NODE` operator for called node N depends on the maximum path length and number of cycles we choose in N.

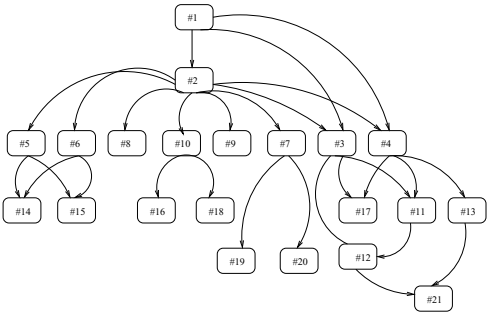

Fig. 5. Structure of a large application

Criteria Definition. We extend the family of criteria defined for unit testing in Section 2.2 (BC, ECC, MCC), for integration testing with the `NODE` operator. We name the extended criteria *iBC*, *iECC,* and *iMCC* (*i* stands for "integration-oriented").

Let m be the integration depth, n the maximum path length, P_n the set of all paths of length lower or equal to n at this depth, l the maximum path length to be considered for the abstract activation condition computation in NODE operators, and $\mathcal{T}$ the set of input sequences. Let $in\,(p)$ denote the input of path p and e denote an internal edge.

The integration-oriented Basic Coverage criterion (*iBC*) requires activating at least once all the paths in the set P_n for the given depth of integration.

Definition 2. *The operator network is covered according to the integration-oriented basic coverage criterion* $iBC^m_{n,l}$ *if and only if:* $\forall p \in P_n$, $\exists t \in \mathcal{T}$: $AC^m_l\,(p) = true$.

The integration-oriented Elementary Conditions Coverage criterion (*iECC*) requires activating each path in P_n for both possible values of its boolean inputs, *true* and *false*.

Definition 3. *The operator network is covered according to the elementary conditions criterion* $iECC^m_{n,l}$ *if and only if:* $\forall p \in P_n$, $\exists t_1 \in \mathcal{T}$: $in\,(p) \wedge AC^m_l\,(p) = true$ *and* $\exists t_2 \in \mathcal{T}$: $not\,(in\,(p)) \wedge AC^m_l\,(p) = true$.

The integration-oriented Multiple Conditions Coverage criterion (*iMCC*) requires paths in P_n to be activated for every value of all its boolean edges, including internal ones.

Definition 4. *The operator network is covered according to the integration-oriented multiple conditions criterion* $iMCC^m_{n,l}$ *if and only if:* $\forall p \in P_n$, $\forall e \in p$, $\exists t_1 \in \mathcal{T}$: $e \wedge AC^m_l\,(p) = true$ *and* $\exists t_2 \in \mathcal{T}$: $not\,(e) \wedge AC^m_l\,(p) = true$.

Subsumption Relation. From the above definitions, it clearly follows that the satisfaction of a criterion $iC \in \{iBC, iECC, iMCC\}$ for a maximum path length n in an operator network implies the satisfaction of the criterion for all path lengths less than n in the operator network. This subsumption relation

is assuming a given integration depth m and maximum path length l for the abstract activation conditions. In other words, $iC^m_{s,l} \subseteq iC^m_{n,l}$ for any $s \leq n$.

With regard to the integration depth parameter, there is no subsumption relation. Consider, for instance, $iBC^m_{n,l}$ and $iBC^{m-1}_{n,l}$. The satisfaction of the former requires to unfold some node calls that are abstracted in the latter at depth *m-1*. As a result, the paths of length less than or equal to n may be different between the two criteria and, therefore, the subsumption relation is not obvious. For the same reason, the original criteria definitions do *not subsume* the extended criteria. Abstraction using the node operator results in a different operator network for the extended criteria from the original criteria. As a result, for the same maximum path length n in the operator network of the global node, the set of paths considered for satisfaction of criterion $C \in \{BC, ECC, MCC\}$ does not necessarily subsume the set of paths considered for satisfaction of criterion $iC \in \{iBC, iECC, iMCC\}$.

Lustructu Tool[3]**.** For coverage measurement using the extended criteria definitions presented in this Section, we enhanced the existing Lustructu tool presented in [6] to support abstraction of node calls and computation of abstract activation conditions. We currently only support an integration depth of 0, i.e., all node calls are abstracted using the *NODE* operator. In the future, we plan to support other values of the integration depth parameter. The Lustructu tool takes as input the Lustre program, name of the node to be integrated, coverage criteria to be measured, and the parameters for the coverage criteria (path length, number of cycles). The tool computes and returns the activation conditions for the integrated node for the selected criteria.

4 Empirical Evaluation

In this section, we evaluate the relative effectiveness of the extended coverage criteria against the original Lustre coverage criteria using an Alarm Management system (AMS) that was developed for an embedded software in the field of avionics. The main functionality of the system is to set off an alarm when the difference between a flight parameter provided by the pilot, and the value calculated by the system using sensors exceeds a threshold value. We believe the system, even though relatively small, is representative of systems in this application area. The AMS is implemented using twenty one Lustre nodes and has several calls between nodes.

Table 3 provides size information on the biggest seven of the twenty one nodes in the system[4]. The main node in the system (node #1) calls node #2 and node #3 that together implement the core functionality of the system. For our evaluation, we use node #2 in the AMS, since it contains several nested temporal loops and two levels of integration, rendering it complex and interesting for us. The two levels of integration in node#2, as depicted by the call graph in Figure 6,

[3] `http://membres-liglab.imag.fr/parissis/TOOLS/LUSTRUCTU/`

[4] The remaining fourteen nodes are relatively small and not complex.

Table 3. Size of the alarm management system nodes

	Code			Op. Net.	
node	*LOC*	*inputs*	*outputs*	*edges*	*operators*
#1	830	29	13	275	190
#2	148	10	3	52	32
#3	157	10	1	59	37
#4	148	10	3	52	32
#5	132	6	5	36	24
#6	98	9	2	33	22
#7	96	7	2	33	22

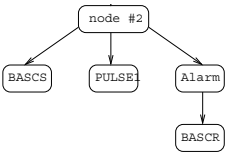

Fig. 6. Call graph for node #1

is as a result of calls to three nodes, two of which in turn call another node. In our evaluation of node #2, we attempt to answer the following two questions,

1. Does the proposed integration oriented criteria reduce the *testing effort* when compared to the original criteria for node integration?
2. Is the proposed integration oriented criteria effective in *fault finding*?

To answer the first question we observe (1) whether the number of activation conditions needed to satisfy the extended criteria is lower than what is needed for the original criteria, and (2) whether test sequences needed for achieving coverage using the extended criteria are shorter than test sequences needed for the original criteria. To address the second question with regard to fault finding effectiveness, we create several mutations of node#2 and determine whether tests are effective in revealing the mutants.

4.1 Testing Effort

In this section we attempt to answer the first question in our evaluation with regard to testing effort. We assess the testing effort in two ways (1) using number of activation conditions, and (2) using test sequence length.

Number of Activation Conditions. Comparing the number of activation conditions between the two criteria helps assess the relative difficulty in satisfying the criteria. It also serves as a good indicator of the relative effort that needs to be spent in coverage measurement and analysis, since more activation conditions usually means a greater number of test cases and longer test case execution time. Analysis of why satisfactory coverage is not achieved becomes difficult with a large number of activation conditions. In our evaluation, we compare the number of activation conditions required for iBC vs BC, iECC vs ECC, and iMCC vs MCC to assess the relative testing effort involved.

Table 4. Number of activation conditions for node #2

	# ACs					
# cycles	iBC	iECC	iMCC	BC	ECC	MCC
1	29	58	342	50	100	1330
3	29	58	342	131	262	4974

In Table 4, we present the number of activation conditions for the extended versus the original criteria for node #2. The data in the table corresponds to complete paths with at most 1 and 3 cycles. As seen in Table 4, the extended criteria, iBC, iECC and iMCC, require significantly fewer activation conditions than their counterpart with no integration, BC, ECC, and MCC, respectively[5]. The difference in number of activation conditions is particularly large between the iMCC and MCC criteria, 342 vs 1330 for 1 cycle and 342 vs 4974 for 3 cycles. As the number of cycles increase, the difference in activation conditions becomes more considerable. From these results, we conclude that the number of activation conditions for the extended criteria is significantly smaller than the original Lustre coverage criteria for node#2.

Test Sequence Length. We now compare the length of test sequences needed to satisfy the extended criteria versus the original criteria. We use the length of test sequences as an indicator of testing effort based on the assumption that developers would need to spend more time and effort constructing longer test sequences. In order to assess the testing effort in an unbiased fashion, we use randomly generated test sequences and measure both criteria. We generated test sequences varying from length 1 up to 1000. We generated five such sets. We do this to reduce the probability of skewing the results by accidentally picking an excellent (or terrible) set of test sequences. We measured the coverage criteria (BC, ECC, and MCC – with and without node integration) over varying test sequence lengths using the following steps:

1. Randomly generate test input sequences of length 1 upto 1000. Generate five such sets with different seeds for random generation.
2. Using each of the test sequences in step 1, execute the coverage node for the criteria and compute the coverage achieved.
3. Average the coverage achieved over the five sets of test sequences.

Figure 7 shows the relation between coverage achieved and test sequence length for iBC vs BC, iECC vs ECC, iMCC vs MCC. Note that these results consider paths with cycles repeating at most 3 times in the node. Figure 7 illustrates that

[5] Node #2 does not contain any temporal loops at the top level (level 1 in Figure 6). As a result, the number of paths for the extended version with the NODE operator is constant regardless of the number of cycles considered. In contrast, in the original version, the number of paths dramatically increases with the number of cycles since the called nodes contain temporal loops. If Node #2 contained temporal loops at the top level, the difference in activation conditions would be more considerable.

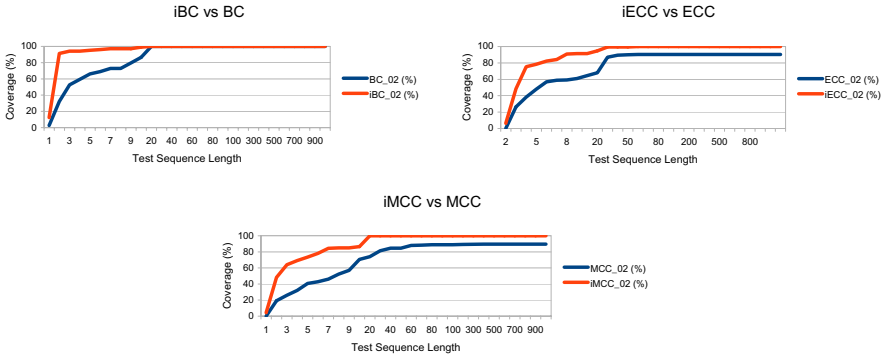

Fig. 7. Comparison of coverage criteria for test sequence lengths 1 to 1000

to achieve the same level of coverage, the extended criteria require shorter test sequences in all three cases - BC, ECC, and MCC. For instance, to achieve 80% iMCC we need test sequences of length 6, whereas to achieve 80% MCC we need test sequences of length 30. Additionally, for all three extended criteria it was possible to achieve 100% coverage with the randomly generated test sequences. On the other hand, for the original criteria, we could only achieve 100% coverage for BC. The maximum coverage achieved for ECC and MCC was only 90% and 89%, respectively, even with test sequences of length 1000. This may be either because the criteria have activation conditions that are impossible to satisfy, or because some of the activation conditions require input combinations that are rare (absent in the generated test sequences).

For iBC vs BC in Figure 7, 100% coverage is achieved for both criteria using test sequences of length at least 20. However, the gradient of the curves for the two criteria are very different. Test sequences of length less than 20 achieve higher iBC coverage than BC. We make a similar observation for iECC vs ECC. Test sequences of length 50 were sufficient to achieve maximum seen coverage for both iECC and ECC (100% for iECC and 90% for ECC). Any given test sequence achieves higher iECC coverage than ECC. For iMCC vs MCC, we find test sequences of length 50 were sufficient to achieve 100% coverage of the iMCC criterion. On the other hand, we need test sequences of length at least 500 to achieve the maximum seen coverage of 89.67% for MCC. In addition, like for BC and ECC, any given test sequence achieves higher iMCC coverage than MCC.

To summarize, the observations made with Figure 7 indicate that the extended criteria require shorter test sequences, and therefore lesser testing effort, than the original criteria for the same level of coverage. The difference is especially significant in the case of iMCC vs MCC. Additionally, it is possible to achieve 100% coverage of the extended criteria but not of the original criteria (except for BC) with the randomly generated test sequences.

Based on these observations and the conclusions about the number of activation conditions made previously, we believe that the extended criteria are more practical and feasible for test adequacy measurement over large systems.

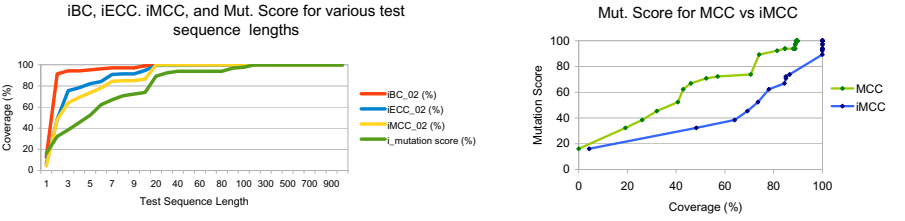

Fig. 8. Mutation Score for Extended Criteria

4.2 Fault Finding Effectiveness

Mutation testing [3] is widely used as a means for assessing the capability of a test set in revealing faults. In our evaluation, we created mutants by seeding a single fault at a time in the original Lustre specification. To seed a fault, we used a tool that randomly selects a LUSTRE operator in the original program and replaces it with a mutant operator in such a way that the mutant program is syntactically correct. Table 5 illustrates the LUSTRE operators we selected and the corresponding mutations for it. We created 26 mutants of node #2. Our procedure for evaluating fault finding effectiveness involved the following steps:

1. Run each of the randomly generated test sequences in Section 4.1 (5 sets of test sequences with length 1 to 1000) on the original Lustre specification of node #2 and record the output.
2. For each of the 26 mutants, run each of the randomly generated test sequences used in step 1 and record the output.
3. For each test sequence, compare the mutant output in step 2 with the corresponding oracle output in step 1. If there is a difference, then the test sequence killed (or revealed) the mutation. Otherwise, the test sequence failed to reveal the seeded fault.
4. The mutation score of a test sequence is the ratio of the number of killed mutants to the total number of mutants (26 in our evaluation).
5. We average the mutation score for each test sequence length over the 5 generated sets.

Figure 8 plots the achieved coverage for iBC, iECC and iMCC and the mutation score. There is a correlation between the criteria satisfaction ratio and the number of killed mutants. This correlation is low for iBC (correlation coefficient of 0.64) since it is a rather weak measure of coverage. Correlation for iECC (0.87) is higher and it is more effective in killing the mutants. iMCC is the most effective in killing the mutants with the highest correlation (0.94). Figure 8 compares

Table 5. Mutations

OPERATOR	NOT	AND	OR	PRE	$<, >, =, \leq, \geq$	$+, -, \star, /$
MUTANT	PRE, $[delete]$	OR, FBY	AND, FBY	NOT, $[delete]$	$<, >, =, \leq, \geq$	$+, -, \star, /$

the mutation score achieved by the extended criterion, iMCC, versus MCC. It is evident that the original criterion is definitely more rigorous than the extended criterion. For instance, achieving 80% MCC reveals 90% of the seeded faults. On the other hand, achieving 80% iMCC only reveals 65% of the seeded faults. Nevertheless, as seen in Section 4.1, achieving the original criterion requires test sequences significantly longer than what is needed for the extended criterion. The shortest test sequence that achieves maximum MCC (89%) is of length 500 and reveals 100% of the faults. The shortest test sequence that achieves maximum iMCC (100%) is of length 50 and reveals 94% of the faults. On an average, this shortest test sequence failed to kill 1 of the 26 mutants. In all 5 sets of randomly generated tests, the shortest test sequence achieving 100% iMCC failed to kill one particular mutant where the `or` LUSTRE operator was replaced by the `fby` operator. Test sequences of length 200 or greater were able to kill this particular mutant. Admittedly, abstraction of the activation conditions over the called nodes leads to a trade off between the test sequence length needed for maximum coverage and the fault finding effectiveness. Nevertheless, as seen in our preliminary evaluation, the iMCC criteria is 94% effective in fault finding, only missing one of the 26 faults, with a reasonably short test sequence.

4.3 Threats to Validity

We face two threats to the validity of our evaluation. First, the AMS is relatively small when compared to other industrial systems. However, it is representative of other systems in the embedded systems domain. Second, the mutants were created by changing the LUSTRE operators. These mutants may not be representative of faults encountered in practice. However, since information on faults that commonly occur is not easy to come by, the fault finding assessment using the mutations is a helpful indicator of the effectiveness of the technique.

5 Conclusion

Previous work [6] proposed structural coverage criteria over LUSTRE programs for measuring unit testing adequacy. This criteria when applied to integration testing experiences severe scalability issues because of the exponential increase in the number of activation conditions to be covered for nodes with calls to other nodes. In this paper, we have presented an extended definition of the LUSTRE structural coverage criteria, called iBC, iECC, iMCC, that helps to address this scalability problem. We use an abstraction that replaces the calls to nodes with a *NODE* operator. This abstraction avoids the entire set of paths of the called node from being taken into account for coverage assessment. However, the abstraction ensures that we can still determine whether the output depends on a given input of the called node. To provide flexibility in dealing with specific needs and complexity of systems, we provide parameters in the criteria definition such as integration depth and path length inside integrated nodes that can be tuned by developers. We hypothesize that the extended criteria will result in considerable

savings in testing effort while still being effective at fault finding. We conducted a preliminary evaluation of this hypothesis using an Alarm Management System. The extended criteria reduced the number of activation conditions by as much as 93% (for MCC) and was effective at revealing 94% of the seeded faults.

References

1. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1), 64–83 (2003)
2. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing 9(5), 193–200 (1994)
3. DeMillo, R.A.: Test Adequacy and Program Mutation. In: International Conference on Software Engineering, pp. 355–356. Pittsburg, PA (1989)
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991)
5. Halbwachs, N., Lagnier, F., Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language lustre. IEEE Trans. Software Eng. 18(9), 785–793 (1992)
6. Lakehal, A., Parissis, I.: Structural coverage criteria for lustre/scade programs. Softw. Test., Verif. Reliab. 19(2), 133–154 (2009)
7. Marre, B., Arnould, A.: Test sequences generation from lustre descriptions: Gatel. In: IEEE International Conference on Automated Software Engineering, Grenoble, France, pp. 229–237 (October 2000)
8. Papailiopoulou, V., Madani, L., du Bousquet, L., Parissis, I.: Extending structural test coverage criteria for lustre programs with multi-clock operators. In: The 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), L'Aquila, Italy (September 2008)
9. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: IEEE Real-Time Systems Symposium, pp. 200–209 (1998)
10. Woodward, M.R., Hedley, D., Hennell, M.A.: Experience with path analysis and testing of programs. IEEE Trans. Softw. Eng. 6(3), 278–286 (1980)