Journal of Software Engineering and Applications, **2013, 6, 27-35**
http://dx.doi.org/10.4236/jsea.2013.610A004 Published Online October 2013 (http://www.scirp.org/journal/jsea)

Automatic Generation of Test Cases in Regression Testing for Lustre/SCADE Programs

Trinh Cong Duy[1], Nguyen Thanh Binh[1], Ioannis Parissis[2]

[1]The University of Danang, University of Science and Technology, Da Nang, Vietnam; [2]Grenoble INP LCIS, University of Grenoble, Valence, France.
Email: Tcduy@dut.udn.vn, ntbinh@dut.udn.vn, ioannis.parissis@grenoble-inp.fr

Received July 24th, 2013; revised August 23rd, 2013; accepted August 31st, 2013

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. During the development process, Lustre programs (or SCADE) are often upgraded, so regression test should be performed to detect bugs. However, regression test is generally costly, because the number of test cases is usually very large. In this paper, we present the solution to automatically generate test cases in regression testing of Lustre/SCADE programs. We apply this solution to regression testing for case study U-turn System.

Keywords: Regession Testing; Test Case; Lustre; SCADE; Model Checking

1. Introduction

Lustre [1] is a declarative, data-flow language, which is devoted to the specification of synchronous and realtime applications. It ensures efficient code generation and provides formal specification and verification facilities. It is based on the synchronous approach which demands that the software reacts to its inputs instantaneously. In practice, it means that the software reaction is sufficiently fast so that every change in the external environment is taken into account. 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) [2]. It is an environment based on the Lustre language, which is used for 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 is not standardized. This graphical modeling environment is used mainly in the aerospace field (Airbus, DO-178B), however it is also used in the domains of transportation, automotive and energy.

Software maintenance is an activity which includes enhancements, error corrections, optimization and deletion of existing features. These modifications may cause the system to work incorrectly. Therefore, regression testing becomes necessary. Regression testing is a type of software testing that seeks to uncover new software bugs, or regressions, existing functional and non-functional areas of a system after changes, such as enhancements, patches or configuration changes, which have been made to them. The intent of regression testing is to ensure that a change among those mentioned above has not introduced new faults [3]. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software [4].

Common methods of regression testing include rerunning previously-completed tests and checking whether program behavior has changed and whether previously-fixed faults have reemerged. Regression testing can be used to test a system efficiently by systematically selecting the appropriate minimum set of tests needed to adequately cover a particular change.

In this paper, we present the solutions of automatic generation of test cases in regression testing for Lustre/SCADE programs. Our idea is to determine the minimum number of test cases, which can detect all possible errors. We propose the use of model checking techniques for

automatically generating test cases. We also define the correlation between requirement specification and test cases. When there is a change in the requirements specification, we identify which test cases are removed, reused or created. Therefore, the minimum set of test cases is designed to reduce operating costs for regression testing of the new version.

In this paper, Section 2 describes an overview of the Lustre/SCADE environment and testing Lustre/SCADE programs using model checking technique. In Section 3, we propose the solution of automatic generation of test cases in regression testing for Lustre/SCADE programs. Section 4, we will apply this approach to the U-turn section management system. Finally, the conclusion is presented.

2. Using Model Checker for Testing Lustre/Scade Programs

2.1. The Lustre Language and SCADE Environment

A Lustre program is structured into nodes; a node is a set of equations which define its 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, $X = E$, it 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 operators.

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 and it is used to refer to the previous execution cycle. The → operator—also called *followed by* (**fby**)—allows the flow initialization and it is used to refer to the first execution cycle. Let $X = (x0, x1, x2, x3,\ldots)$ and $(e0, e1, e2, e3,\ldots)$ be two Lustre expressions. Then **pre(X)** denotes the sequence $(nil, x0, x1, x2, x3\ldots)$, where **nil** is an undefined value, while $X \rightarrow E$ denotes the sequence $(x0, e1, e2, e3,\ldots)$ [5].

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 hypothesis of synchrony can be checked.

- ***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 fiow X has not the same clock as 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 fiow 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*.

A simple Lustre program is given in **Figure 1**. This program receives a signal set and returns a boolean level that must be true during delay seconds after each reception of set. The seconds are provided by a boolean input second, that is true whenever a second elapses. The program reuses the node STABLE, calling it at a suitable clock, by filtering its input parameters.

SCADE (Safety Critical Application Development Environment) is a graphical environment commercialized by Esterel Technologies [6]. It is based on the synchronous language Lustre. So, we are often referred to as Lustre/SCADE. **Figure 2** shows the relationship between language Lustre and the environment SCADE. Lustre/SCADE usually used to build the applications of reactive system [7].

```
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
```

Figure 1. Example of a Lustre program.

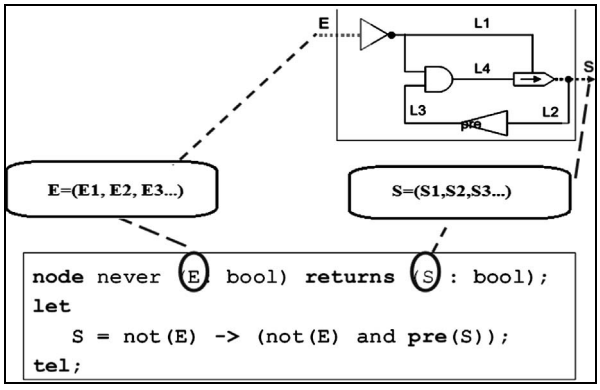

Figure 2. Ex: performance Lustre programs with SCADE.

2.2. Specification of a Software in Lustre Program

The main application area of synchronous programming is safety-critical software. For such software, three kinds of specification are needed:

- The *functional specification* of the software is a Lustre node computing the software outputs from the software inputs. This node is deterministic: a given input sequence will always cause the software to issue the same output sequence.
- The *software environment* specification is a set of invariant properties providing a nondeterministic description of the valid software inputs.
- The *safety properties* are invariant temporal logic formulae stating that some dangerous behaviors will never occur.

We have some useful nontrivial temporal operators which can be expressed as Lustre nodes. Consider the following property:

"*Any* occurrence *of a critical situation must be followed by an alarm within five second delay*."

Such a property relates to three events: the critical situation occurrence, the alarm, and the deadline. The latter can be provided externally, and it can also be easily expressed in Lustre. A general pattern for this property is the following:

"*Any* occurrence *of event A is followed by an occurrence of event B before the next occurrence of event C.*"

However, this formulation is not directly translatable into Lustre, since it refers to what happens in the future following an A occurrence, while Lustre only allows references to the past with respect to the current instant. That is why it is first translated into the equivalent past expression:

"*Anytime C occurs, either A has never occurred previously, or B has occurred since the last occurrence of A.*"

We define a node, taking three Boolean input parameters A, B, C, and returning a Boolean output X such that X is always true if and only if the property holds:

node *Always_from_to_*(*A*, *B*, *C* **:** ***bool***)
returns*(ok* **:** ***bool***);
let
ok = *Once_since(C*, *B*) ***or*** *Always_since(A*, *B*);
tel

The node *Once_since*(*A*, *B*) (resp. *Always_since*(*A*, *B*)) returns a true value if and only if its first input has been once (resp. continuously) true since the last time its second input was true:

node *Once_since_*(*A*, *B* **:** ***bool***)
returns(*ok* **:** ***bool***);
let
ok = ***if*** *B* ***then*** *A* ***else***(***true*** $\rightarrow$ (*A* ***or*** *pre* (*ok*)));
tel

node *Always_since_*(*A*, *B* **:** ***bool***)
returns(*ok* **:** ***bool***);
let
ok = ***if*** *Never*(*B*) ***then true***
else if *B* ***then*** *A* ***else*** *A* ***and*** *pre* (*ok*);
tel

The node *Never* (*A*) returns a true value if and only if *A* has never been true in the past:

node *Never*(*A* **:** ***bool***)
returns(*ok* **:** ***bool***);
let
ok = *not A* $\rightarrow$ (***not*** *A* ***and*** *pre*(*ok*));
tel

2.3. Testing Lustre/SCADE Programs with Model Checker

Model checkers are formal verification tools that evaluate a model to determine if it satisfies a given set of properties [8]. A model checker will consider every possible combination of inputs and state, making the verification equivalent to exhaustive testing of the model. If a property is not true, the model checker produces a counterexample showing how the property can be falsified.

Model checking techniques require two inputs: model and property. Properties are specified with temporal logic. Lustre language can be considered as a subset of a temporal logic, we can use Lustre for define both model and properties. The use of a programming language to express both programs and their properties is interesting, since all the structuring facilities of the language become available for readability and expressiveness.

The idea of testing with model checkers is to interpret counter examples as test cases. A suitable test case execution framework can extract from this the test data, and also the expected results (*i.e.*, test oracle). A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case, testing this transition the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B [9].

A test purpose describes the desired characteristics of a test case that should be created. For example, it could describe the final state of the test case, or a sequence of states that should be traversed. The test purpose is specified in temporal logic and then converted to a never-claim by negation; this asserts that the test purpose never becomes true. The counterexample illustrates how the never-claim is violated, and thus shows how the original test purpose is fulfilled. These never-claims are called trap properties, and for each item that should be covered one trap property is generated, in **Figure 3** we create test cases with model checker uses trap property.

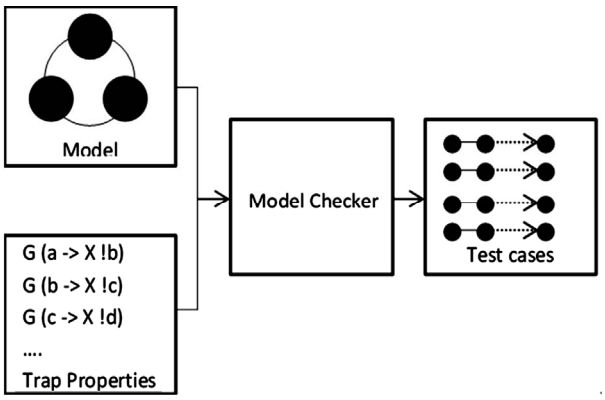

Figure 3. Creating test cases with model checker.

In many cases, testing with model checkers is applied to reactive systems, which read input values from sensors and set output values accordingly. The system reacts to inputs by setting output values, such that a logical step in a counterexample can be mapped to an execution cycle of the system under test. In the reactive system scenario, counterexamples can directly be interpreted as test cases [10]. Because test cases are always finite, it is necessary to distinguish between traces with or without loopback when mapping a trace to a test case. In this paper, we use this technique to test for reactive application Lustre/ SCADE.

One of its main advantages is that it can be used as a temporal logic to express software invariant properties. The satisfaction of the latter can be proven by model checking. In this work, we used LESAR tool.

Given a program P and a safety property S, LESAR checks that S is true in all states of program P under some assumptions (expressed by an assertion A). The problem thus reduces to proving that the only boolean output of P is always true during any execution of the program which permanently satisfies the assertion A. The verification is performed on a finite state abstraction of the program. Any numerical computation is deliberately ignored, and boolean expressions depending on numerical variables (e.g., comparisons) are considered nondeterministic. LESAR takes a verification program as input. It is written in Lustre. It has the same inputs than the original program and only one boolean output. This output is equal to the safety property to be proved. The assert operator allows to restrict the verification to the inputs satisfying the environment constraints [11].

3. Generation of Test Cases for Lustre/ SCADE Programs

3.1. Problem Statement

The Lustre/SCADE applications usually require very high quality and rigorous testing activities before they are deployed to use. In this section of the paper, we present the solutions for generating test cases in regression testing of Lustre/SCADE programs. In particular, the model checking technique will be used for automatically generating test cases. We also define the correlation between requirement specification and test cases. When there is a change in the requirements specification, we can identify which test cases are removed, re-used and new test cases. Since then, the minimum set of test cases is designed to reduce operating costs for regression testing of the new version.

During the development process, the system is often updated, regression test should be performed to detect bugs. However, in the regression testing process, if we perform all the test cases in the old version and generate the new test cases, the operation is really very expensive. So, we need to optimize the number of test cases for implementing a regression testing. The following figure shows the process of developing the test suite for regression testing in the new version.

In **Figure 4**, we can see: in the first version, we do not use the regression testing but the model checking for verifying application model to generate test cases. In this version, the selection of test cases has not been applied, all test cases are created to be done. From the second version, the regression test will be applied. We need apply the technique of selection and automatic generation of test cases in regression testing, for eliminating unnecessary test cases to reduce costs, we call this technique is TSTG (Technique of selection and automatic generation of test cases). The details of this technique will be presented in Sections 3.2 and 3.3.

Example: We illustrate the use of Lustre as a language for expressing functional and safety features and environment invariant properties on a subway device. This device is an automatic "U-turn" section management system (called UMS), allowing trains, at each terminus of the subway line, to switch from one track to the other and go back to the opposite direction [12].

The U-turn section (**Figure 5**) is composed of three tracks A, B, C and a switch S. Assuming the entering track is A and the exiting track is C, trains switching from A to C must first wait for S to connect A with B,

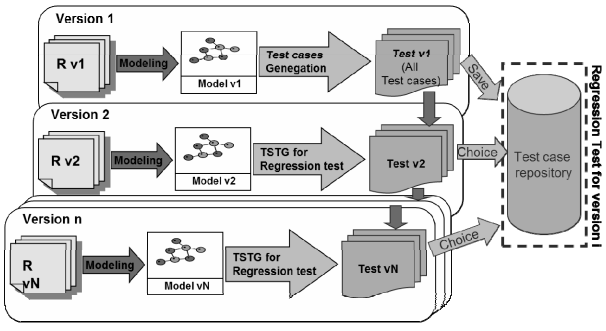

Figure 4. Evolution of the system and regression test.

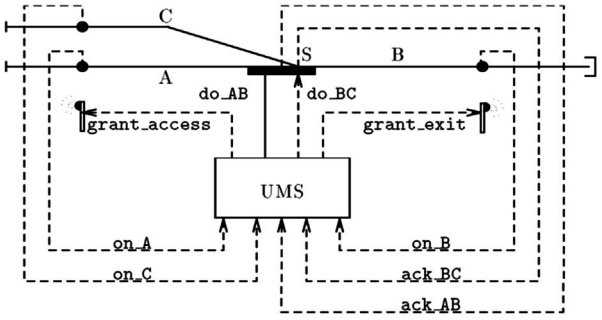

Figure 5. The UMS system and its environment.

then transit on B and wait again for S to connect B with C before going back on C. Since UMS must both drive the switch and manage train movement in order to avoid accidents into the section, its behavior is typically reactive: upon receipt of information about the U-section configuration (*i.e.* switch status and train position inside the section), it should deliver positioning requests to the switch and access grants to trains. These four kinds of events can be modeled by the following signals.

- **ack_AB** and **ack_BC** are emitted by the switch to indicate whether it actually connects A with B or B with C. When none of these signals is active (*i.e.* when the actual connection is changing), trains must not take the switch.
- **on_A**, **on_B** and **on_C** are emitted by three sensors, one on each track of the section. They are active as long as there is a train on their respective track.
- **do_AB** and **do_BC**, emitted by UMS, are the requests for the switch to connect A with B or B with C.
- **grant_access** and **grant_exit**, also emitted by UMS, are grants for trains to move along the section (*i.e.* traffic lights). The first will allow trains to access the section only if it is empty and the switch connects A and B.

The second will allow trains to exit B only if the switch connects B with C.

3.2. Requirements of the Process Control in Version 1

We formally express here some safety properties of the UMS system (variables empty section and only_on_B are assumed to be defined as in the above node, UMS):
The access is granted to a train only if the section is Empty **(*R*1)**:

Implies (*grant_access*, *empty_section*

The switch positioning requests, **do_AB** and **do_BC** should never be simultaneously active **(*R*2)**:

not (*do_AB and do_BC*)

The switch must always connect A to B (resp. B to C) from the instant when a train is allowed to enter (resp. to exit) the section until it has arrived on track B (resp. it has actually left it):

Always_from_to (*ack_AB*, *grant_access*, *only_on_B*) **(*R*3)**

And

Always_from_to (*ack_BC*, *grant_exit*, *empty_section*) **(*R*4)**

Assumptions: With the requirement of this version, we have the test suite with 4 test cases {*T*1, *T*2, *T*3, *T*4} for requirements {*R*1, *R*2, *R*3, *R*4}.

3.3. Requirements of the Process Control in Version 2

In version 2 of U-turn system, there is one update in requirement (R1): The access is granted to a train when the section is empty or the train is only on C (R1') and other requirements do not change, in comparison with previous versions.

Implies(*grant_access*, *empty_section or only_on_C*)

We can see that the requirements R2, R3, R4 do not change in version 2, so will be have some test cases not affected when the updated version. Normally, when testing version 2, we often have to run all the test cases created for versions 1 and the new test cases in version 2. Hence, requires a lot of costs. The problem here is that we need to define a set of test cases which will be implemented in regression testing for version 2, and it should be a least. We will propose technical solution to solve this problem in the next sections and present empirical results in Part 4.

3.4. Test Case Life Cycle in Regression Testing

In regression testing, identifying test case needed for executing is very important, since, we don't need to re-execute the old test cases for old requirements in the previous version (the requirements have not been impacted by the evolution). In this work, we introduce the concept of the test cases life cycle in regression testing. Each test case will be created until deleted (removed) in regression testing must meet the requirements: when there is a change in software versions, if a test case has been used in the older versions, but not affected by the upgrade process, then it will be removed. The test case is affected by the upgrade will be updated to match the new version. The test cases for new requirements in the new version will be created. This content is described in **Figure 6**.

3.5. Generation of Test Cases in Regression Test

In regression testing, the main issue is to create a set of test cases for the new version. Here, we have two versions, a new version *M'* and the previous version *M* (**Figure 7**). The basic idea in the construction of the set of test cases is as follows: We compare the current ver-

sion (*M'*) with the previous version (*M*) of the system to determine: The new requirements, the requirements need to be changed, the requirements are not affected and the requirements are removed in the new version (in *M'*). We present a method for automatic identification of test cases when the requirements are affected by the process of changing the system version. The test cases included in the old version, but not affected by the evolutionary process, will be eliminated in the new version. Only the test cases of the requirements are affected by change, in the new version, they will be kept and the new test cases are generated.

After determining the requirements affected by the evolution of the system, in **Figure 8**, we use model checker for automatic generation of test cases, and use this technique for determined the relationship between requirements and test cases, when there is an update in requirements, some test cases will be removed or updated, or newly created. We will build a collection of regression test cases for new version.

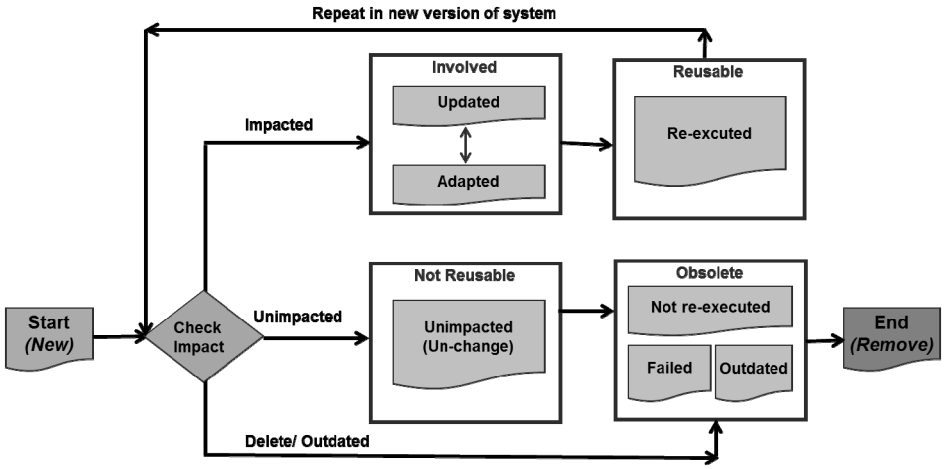

Figure 6. Test case life cycle in regression testing.

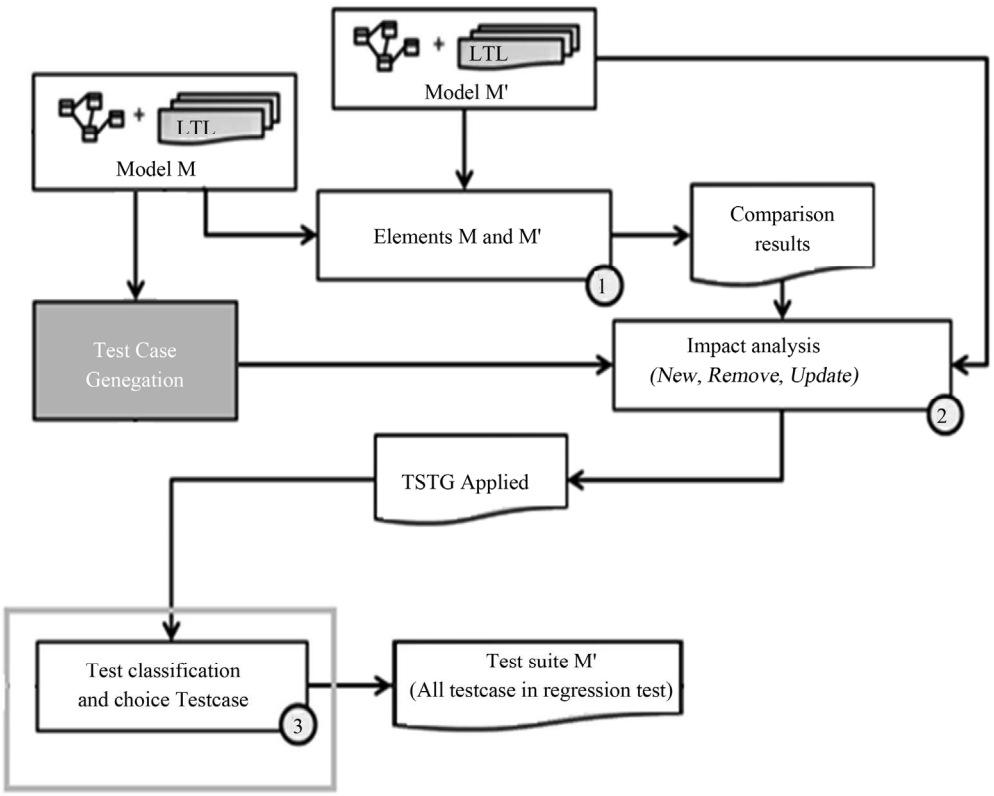

Figure 7. Process to determine the status of the test cases.

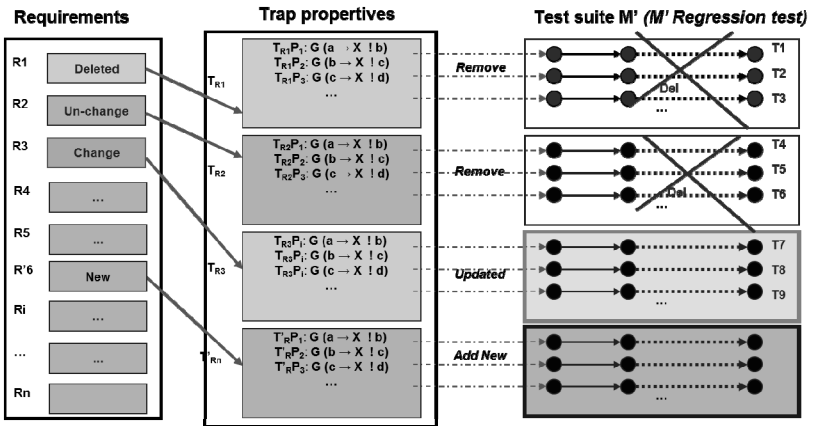

Figure 8. Generation of trap propertives (test cases) for regression testing.

4. Generation of Test Cases for the UMS Program

In this section, we have applied this approach to the example U-turn section management system. The UMS systems are described in detail in the example in Section 3.1. The UMS program controls the U-turn section of a sub-way and written by language Lustre. A U-turn section allows trains to switch from one track to another, and to go back in the opposite direction [13]. The above specifications are formally expressed by the following Lustre node in **Figure 9**.

Here, we have two versions of requirement for the UMS system. In version 1, all the test cases need to be executed. When there is a change from version 1 to version 2, in the version 2, we will include regression testing techniques. The most important thing is to build a set of test cases will be executed.

4.1. Generation of Test Cases for Version 1

We will use model checking techniques to generate test cases for version 1. We will use the Lesar tool for verifying UMS program based on the safety properties.

4.2. Safety Properties of the Process-Control Software

With requirements in Section 2.1, we have the set of safety properties required from our U-turn management system:

no_collision = implies (*grant_access*, *empty_section*);
exclusive_req = not (*do_AB and do_BC*);
no_derail_AB =
always_from_t o(*ack_AB*, *grant_access*,*only_on_B*);
no_derail_BC =
always_from_to(*ack_BC*, *grant_exit*,*empty_section*);

As mentioned before, this system must always avoid the occurrence of the two dangerous situations. The global property to prove is expressed by the following

```
node UMS(on_A,on_B,on_C,ack_AB,ack_BC: bool)
     returns (grant_access,grant_exit,
          do_AB,do_BC: bool);
var empty_section, only_on_B: bool;
let
   grant_access = empty_section
        and ack_AB;
   grant_exit = only_on_B and ack_BC;
   do_AB = not ack_AB and empty_section;
   do_BC = not ack_BC and only_on_B;
   empty_section = not(on_A or on_B
        or on_C);
   only_on_B = on_B and not(on_A or on_C);
tel
```

Figure 9. The Lustre program for the UMS system.

Lustre equation:
property =no_collision
and *exclusive_req*
and *no_derail_AB*
and *no_derail_BC*.

4.3. Environment Constraints

The following Lustre expressions are invariant properties of the U-turn section:

The switch cannot simultaneously connect the track B to both tracks A and C:

not (*ack_AB* ***and*** *ack_BC*)

Once in a given position, the switch remains stable unless it is requested to move to the opposite position:

Always_from_to(*ack_AB*, *ack_AB*, *do_BC*)

And

Always_from_to(*ack_BC*, *ack_BC*, *do_AB*)

With those environment constraints, we can model the environment for the UMS system in Lustre source code in **Figure 10** (ASSERTIONS section) [14].

To generate test cases for this version, we need to take two steps: applying model checking and generating test cases

***Step* 1.** Using Lesar tool for model checking UMS prgram

***Step* 2.** Generating test cases from counter-examples.


```
node UMS_verif(on_A,on_B,on_C,ack_AB,
     ack_BC: bool)
      returns(property: bool);
var
   grant_access,grant_exit: bool;
   do_AB,do_BC: bool;
   no_collision,exclusive_req: bool;
   no_derail_AB,no_derail_BC: bool;
   empty_section, only_on_B: bool;
let
   empty_section = not(on_A or on_B or on_C);
   only_on_B = on_B and not(on_A or on_C);
      -- ASSERTIONS
   assert not(ack_AB and ack_BC);
   assert true ->
         always_from_to(ack_AB,ack_AB,do_BC);
   assert true ->
         always_from_to(ack_BC,ack_BC,do_AB);
   assert empty_section -> true;
   assert true ->
         implies(edge(not empty_section),
           pre grant_access);
   assert true -> implies(edge(on_C),
           pre grant_exit);
   assert true -> implies(edge(not on_A),on_B);
   assert true -> implies(edge(not on_B),
                on_A or on_C);
   -- UMS CALL
   (grant_access,grant_exit,do_AB,do_BC) =
    UMS(on_A,on_B,on_C,ack_AB,ack_BC);

   -- PROPERTIES
   no_collision =
    implies(grant_access,empty_section);
   exclusive_req =
         not(do_AB and do_BC);
   no_derail_AB =
         always_from_to(ack_AB, grant_access,
         only_on_B);
   no_derail_BC =always_from_to(ack_BC,
         grant_exit, empty_section);
   property =no_collision and exclusive_req and
         no_derail_AB and no_derail_BC;
tel
```

Figure 10. The verification program with Lesar.

The main idea is: generation test cases using counter-examples in model checking. We need to define some trap properties in model checking.

In requirements R1: the access is granted to a train only if the section is *Empty.*

empty_section = not (*on_A or on_B or on_C*);

And

Implies (*grant_access*, *empty_section*).

To test this property, we need to check:

If the access is granted, then the section is empty (*i.e.*, we need to set the system to a state where the access is granted and then observe the value of on_A, on_B and on_C). The trap property here is: *grant_access*.

If one signal among on_A, on_B or on_C is true, then grant access is false. So trap properties could be (they correspond to all the possible positions of the train in the section):

TRAP1: trap_*property = on_A* ***and*** *not_on_B* ***and*** *not on_C*;

or TRAP2: *trap_property =not on_A* ***and*** *on_B* ***and*** *not on_C*;

or TRAP3: *trap_property =not on_A* ***and*** *not_on_B* ***and*** *on_C*;

or TRAP4: *trap_property =on_A* ***and*** *on_B* ***and*** *not on_C*;

or TRAP5: *trap_property = on_A and on_B* ***and*** *on_C*;

With the trap properties, we can create test cases with model checker using Lesar tool and **Figure 11** is an example:

In here, we have 05 trap properties {*TRAP*1, *TRAP*2, *TRAP*3, *TRAP*4, *TRAP*5} for requirement R1. So, with R1, we have 05 test cases {*T*1_1, *T*1_2, *T*1_3, *T*1_4, *T*1_5}. With the requirements in version 1, we have the test suite in **Table 1**.

4.4. Test Case Generation in Regression Testing for the UMS

In version 2 of the UMS system, we have only one update in requirement (*R*1): The access is granted to a train when the section empty or the train only on C (*R*1'):

Implies (*grant_access*, *empty_section or only_on_C*)

When changing from version 1 to version 2, we need to define a set of test cases will be implemented in regression testing for version 2.

We can see, requirements R2, R3, R4 do not change in version 2, so the test case:

{*T*2_1, *T*2_2, *T*2_3, *T*2_4,*T*2_...},
{*T*3_1, *T*3_2, *T*3_3, *T*3_4,*T*3_...},
{*T*4_1, *T*4_2, *T*4_3, *T*4_4,*T*4_...}.

are not affected when have updated, only requirement R1 has updated. Applying TSTG technique (in Section 2), we will remove all test cases for requirements R2, R3, R4 and update test cases for requirement R1.

We have status of test suite for version 2 in **Table 2**.

Here, we need to update test cases for R1' in version 2. To test this property, we need to check: If the access is granted, then the section is empty or the train in the section only_on_C. The same with version 1, the trap property here is: *grant_access*. If one signal among on_A,

```
 - - - TRANSITION 1 - - -
ack_AB  and not ack_BC and not on_B and not on_A and ont
on_C
- - - TRANSITION 2 - - -
not ack_AB  or
on_B
FALSE PROPERTY
```

Figure 11. Test counterexample result with model checker.

Table 1. The test suite for version 1 of the UMS.

ID	Req	Test Cases
1	R1	{T1_1, T1_2, T1_3, T1_4, T1_5}
2	R2	{T2_1, T2_2, T2_3, T2_4, T2_...}
3	R3	{T3_1, T3_2, T3_3, T3_4, T3_...}
4	R4	{T4_1, T4_2, T4_3, T4_4, T4_...}

Table 2. Status of test suite in version 2 of the UMS.

ID	Test cases	Status
1	{T1_1, T1_2, T1_3, T1_4, T1_...}	Update
2	{T2_1, T2_2, T2_3, T2_4, T2_...}	Remove
3	{T3_1, T3_2, T3_3, T3_4, T3_...}	Remove
4	{T4_1, T4_2, T4_3, T4_4, T4_...}	Remove

on_B is true (we don't need to check signal among on_C) then grant access is false. Based on the trap properties in version 1, we have: The trap property TRAP3 of R1: *not on_A* and *not_on_B and on_C* is not useful for R1' and regardless to the value of on_C. Therefore, we will remove TRAP3 in R1' and update other trap properties. So trap properties for R1' could be:

TRAP1':

trap_property = on_A ***and*** *not_on_B*;

or TRAP2':

trap_property = not on_A ***and*** *on_B*;

or TRAP'4 = TRAP'5:

trap_property = on_A ***and*** *on_B*;

So, in version 2, the set of test cases will be implemented for regression testing is:

{T'1_1, T'1_2, T'1_4}.

The general idea of using model checking in software testing is not new; the ability to construct counter examples by a model checker has been proposed as a way of deriving test cases. However, in this work, we have applied model checking to generate test cases for regression testing and applied to Lustre/SCADE programs. We also define the correlation between requirement specification and test cases. When there is a change in the requirements specification, we identify which test cases are removed, reused or created. Therefore, the minimum set of test cases is designed to reduce operating costs for regression testing of the new version.

5. Conclusion and Future Work

In this paper, we present the solution to automatically generating test cases in regression testing of Lustre/ SCADE programs. In this solution, we propose the use of model checking techniques for automatically generating test cases. We also define the correlation between requirement specification and test cases. Since, the minimum set of test cases is designed to reduce operating costs for regression testing of the new version. Finally, we apply this solution in regression testing for the case study U-turn System.

The case study used in this paper is a fairly simple application. To demonstrate scalability of the method and applicability to more realistic, complicated software systems, we plan to use larger specifications taken from real life examples. We intend to implement tool for automatic create trap properties from specification and from model of Lustre/SCADE programs.

REFERENCES

[1] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, "The Synchronous Data Flow Programming Language Lustre," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305-1320. http://dx.doi.org/10.1109/5.97300

[2] Esterel Technologies Web Site, Editor of the SCADE Suite, Based on the Lustre Language, 2013. http://www.esterel-technologies.com/products/scade-system

[3] G. Myers, "The Art of Software Testing," Wiley, Hoboken, 2004.

[4] R. Savenkov, "How to Become a Software Tester. Roman Savenkov Consulting," 2008, p. 386.

[5] B. Seljimi and I. Parissis, "Automatic Generation of Test Data Generators for Synchronous Programs: Lutess v2," *DOSTA* '07: *Workshop on Domain Specific Approaches to Software Test Automation*, ACM, New York, 2007, pp. 8-12.

[6] Esterel Technologies Web Site, 2013. http://www.esterel-technologies.com/products/scade-system/scade-system-design

[7] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Realtime Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1270-1282.

[8] E. Clarke, O. Grumberg and D. Peled, "Model Checking," The MIT Press, Cambridge, Massachusetts, 2001.

[9] J.-J. Wang, B. Zhang and Y. Chen, "Test Case Set Generation Method on MC/DC Based on Binary Tree," 5*th International Conference on Machine Vision* (*ICMV* 2012): *Computer Vision*, *Image Analysis and Processing*, 13 March 2013.

[10] G. Fraser, F. Wotawa and P. E. Ammann, "Testing with Model Checkers: A Survey," *Journal Software Testing, Verification & Reliability*, Vol. 19, No. 3, 2009, pp. 215-261.

[11] N. Halbwachs, "Lustre Program Verification: The Tool Lesar, Synchronous Programming of Reactive Systems," *The Springer International Series in Engineering and Computer Science*, 1993, pp. 139-147.

[12] N. Halbwachs, D. Pilaud, F. Ouabdesselam and A.-C. Glory, "Specifying, Programming and Verifying Real-Time Systems, Using a Synchronous Declarative Language," *Workshop on Automatic Verification Methods for Finite State Systems*, *LNCS* 407, Grenoble, 12-14 June 1989.

[13] F. Ouabdesselam and I. Parissis, "Testing Techniques for Data-Flow Synchronous Programs," *Proceedings of the Second International Workshop on Automated and Algorithmic Debugging*, Saint-Malo, 22-24 May 1995.

[14] N. Halbwachs, F. Lagnier and C. Ratel, "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language Lustre," *IEEE Transactions on Software Engineering*, Vol. 18, No. 9, 1992, pp. 785-793. http://dx.doi.org/10.1109/32.159839

