International Journal of Electrical and Computer Engineering (IJECE)
Vol. 7, No. 5, October 2017, pp. 2635~2644
ISSN: 2088-8708, DOI: 10.11591/ijece.v7i5.pp2635-2644

LusRegTes: A Regression Testing Tool for Lustre Programs

Nguyen Thanh Binh[1], Trinh Cong Duy[2], Ioannis Parissis[3]
[1,2]The University of Danang - University of Science and Technology, Vietnam
[3]University Grenoble Alpes, Grenoble INP-LCIS, France

Article Info

Article history:

Received Mar 16, 2017
Revised May 16, 2017
Accepted Jul 11, 2017

Keywords:

Activation condition
Lustre programs
Operator network
Regression testing tool
Test data generation

ABSTRACT

Lustre is a synchronous data-flow declarative language widely used for safety-critical applications (avionics, energy, transport...). In such applications, the testing activity for detecting errors of the system plays a crucial role. During the development and maintenance processes, Lustre programs are often evolving, so regression testing should be performed to detect bugs. In this paper, we present a tool for automatic regression testing of Lustre programs. We have defined an approach to generate test cases in regression testing of Lustre programs. In this approach, a Lustre program is represented by an operator network, then the set of paths is identified and the path activation conditions are symbolically computed for each version. Regression test cases are generated by comparing paths between versions. The approach was implemented in a tool, called LusRegTes, in order to automate the test process for Lustre programs.

Corresponding Author:

Nguyen Thanh Binh,
Information Technology Faculty,
The University of Danang - University of Science and Technology,
54 Nguyen Luong Bang, Danang, 550000, Vietnam.
Email: ntbinh@dut.udn.vn

1. INTRODUCTION

Lustre is a declarative, data-flow language [1], devoted to the specification of synchronous and real-time applications. It ensures efficient code generation and provides formal specification and verification facilities. It is based on the synchronous approach, which demands the software to react to its inputs instantaneously. In practice, this 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), which has been commercialized by Esterel Technologies. It is an environment based on the Lustre language and it allows the hierarchical definition of the system components and the automatic code generation.

The Lustre/SCADE environment is widely used for safety-critical applications (avionics, energy, transport...). In such applications, software testing [2], [3] for ensuring the quality plays a vital role. The objective of testing activity is to reveal errors in applications as soon as possible in development and maintenance phases.

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, in existing functional and non-functional areas of a system after changes, such as enhancements, patches or configuration changes, have been made to them. The intent of regression testing is to ensure that a change such as those mentioned above has not introduced new faults [4]. In addition, one of the main objectives for regression testing is to determine whether a change in one part of the

software affects other parts of the software. 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.

Extensive studies have been carried out focusing on the automatic test data generation. Several tools have been proposed, mainly concerning the formal specification processing to produce test input sequences. Some tools for programs specified in Lustre are GATeL [5], Lurette [6] and Lutess [7], [8]. However, these tools don't support regression test generation. Hence, our research work aims to develop the LusRegTes tool, which allows an automatic generation of regression test data for Lustre programs. This approach was recently introduced in [9]. Lustre programs are unordered sequences of equations, defining how their input flows are transformed into output flows through a set of operators. Therefore, the most suitable representation for Lustre programs is a directed graph, called operator network. We define then the correlation between path activation conditions of operator networks and test cases. When there is a change in path activation conditions, we identify the test cases to be removed, reused and created. Since, the minimum set of test cases is designed to reduce operating costs for regression testing of the new version.

The paper is organized as follows. Section 2 presents an overview of the Lustre language, the SCADE environment, and testing in Lustre/SCADE. The approach of test case generation in regression testing for Lustre/SCADE programs is presented in Section 3. Section 4 introduces the implementation of the LusRegTes tool and Section 5 presents the experimentation. Finally, the paper finishes by the conclusion and future work.

2. BACKGROUND

2.1. Regression Testing

Testing can be used to build the confidence in the correctness of the software and to increase the software reliability. The major difference between the regression testing and the development testing is that during regression testing an established set of tests may be available for reuse. Regression testing is essential when software product has been changed and it is often applied during maintenance phase. Regression testing ensures that the modified program meets its specification and new errors are uncovered [4].

Regression testing, which is an expensive maintenance process, is responsible for revalidating the modified software. Maintenance cost of the software is high as compared to its development cost. The maintenance cost of the software may exceed to around 70% relevant to the development cost [4]. Maintenance has three types: perfective, adaptive and corrective. Corrective maintenance is performed to correct error that has been uncovered in some part of the software. Adaptive maintenance is performed when software is modified to ensure its compatibility with the new environment in which it will operate. Perfective maintenance is performed to add new features to the software or to improve performance of the software. Regression testing is done in all these types [4]. There are two types of regression testing: progressive regression testing and corrective regression testing. Corrective regression testing is applied when specification is not changed; probably some other changes have been made, *i.e.* correcting an error. In this case, test data can be reused. Progressive regression testing is applied when specification has been changed and new test data must be designed at least for the modified parts of the specification.

Regression testing uses two approaches to test the modified software: retesting all and selective tests. Retesting all approach chooses all test cases from the unchanged software to test the changed software, but the approach is time consuming as well as resource consuming. Selective retesting approach chooses a subset of the tests from the old test suit to test the modified software. In regression testing, selecting a suitable subset with an efficient algorithm is a major area of research. Traditionally, the process of regression testing is conceptualized as follows. Given a program L_1, a modified version L_2, and a test suite T_1, which is used to test L_1 together with the expected results. In order to reduce costs and to reuse T_1 as effectively as possible, selective retesting techniques first select a subset of T_1, T' based on the modifications in L_2 compared with L_1. Second, if necessary, a set of new tests T'' is created, to test the new, modified and untested portions of L_2. Thus, the test suite T_2 for L_2 is defined as:

$T_2 = T' \cup T''$

In regression testing, identifying test cases for executing is very important, since, we don't need to re-run the old test cases for old requirements in the previous version (the requirements have not been impacted by the evolution) [10]. In this work, we use the concept of lifecycle test cases in regression testing. Each test case will be created until deleted (removed) in regression testing must meet the following requirements: When there is a change in software versions, if a test case has been used in older versions, but not affected by the upgrade process, then it will be removed; The test cases, which are affected by the

upgrade, will be updated to match the new version; The test cases for the new requirements in the new version will be created.

2.2. Lustre/SCADE

2.2.1. The Lustre language

Lustre is a synchronous data-flow language dedicated to programming reactive systems such as automatic control and monitoring systems in various fields like nuclear plants, civil aircraft and automotive systems. Lustre is well suited for programming the critical parts of real-time systems especially thanks to its well-formalized semantics and to the associated verification tools.

A Lustre program is structured into nodes [1]. 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, $X = E$, 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*** ($\rightarrow$).

- The ***precedence*** operator (*pre*) has been introduced to allow breaking data-flow loops and then defining a causally correct specification [11]. If E is an expression denoting the sequence (e_0, e_1, ... e_{n-1}, ...), the expression *pre*(E) denotes the sequence (*nil*, e_0, e_1, ... e_{n-1}, ...) where *nil* is an undefined value. In other words, *pre*(E) returns the value of the expression at the moment $t - 1$ (if $t > 0$, *nil* otherwise).
- The ***initialization*** operator ($\rightarrow$ or *fby*), called ***followed by***, makes it possible to define the initial value for an expression (*i.e.* at $t = 0$). If E and F are expressions denoting, respectively, the sequences (e_0, e_1, ... e_n, ...) and (f_0, f_1, ... f_n, ...), then $E \rightarrow F$ denotes the sequence (e_0, f_0, f_1, ... f_n, ...).

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.

2.2.2. The SCADE environment

SCADE is a graphical environment commercialized by Esterel Technologies. It is based on the synchronous language Lustre. So, we are often referred to as Lustre/SCADE. Lustre/SCADE is usually used to build the applications of reactive systems. 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 it is also used in the domains of transportation, automotive and energy [1].

2.3. Model of Lustre programs

Lustre is a data-flow language: the input flows of a program are transformed into the output flows through a set of dependent or independent operators. As in SCADE descriptions, the most usual representation for Lustre programs is a directed graph, called operator network.

2.3.1. Operator network of Lustre program

Lustre programs are usually represented as operator networks. An operator network is a labeled graph connecting operators by means of directed edges. An *operator* (logical or numerical) specifies data-flow transfers from inputs to outputs. An *edge* specifies the data flow between two operators. An operator network contains usual logical (*and*, *or*, *not*), arithmetical (+, −, /, *) and relational operators (*LT* (<), *LTE* (<=), *EQ* (==), *NEQ* (< >), *GT* (>), *GTE* (>=)), the conditional operator (*ITE*) and the temporal operators (*pre*, *fby*).

It is assumed that two functions are associated with any operator *op: in*(*op*) returns the set of the operator input edges and *out*(*op*) returns the set of the operator output edges. There are three kinds of edges: *input*, *output* and *internal* edges. Input (resp. output) edges are occurrences of input (resp. output) variables of the Lustre program. Internal edges correspond to occurrences of local variables. Every edge has a single source operator and a single destination operator. An edge e_2 is a successor of an edge e_1 if and only if there is an operator of which e_1 is an input and e_2 is an output.

For sake of convenience, in addition to the previous operators, entry operators and exit operators are introduced. Entry (resp. exit) operators have no inward (resp. outward) edges and are connected to the network through input (resp. output) edges [11]. All operators are single output.

2.3.2. Paths in the model

The operator network defines paths within program [11]. A path $p = (e_1, e_2, \ldots e_n)$ is a finite sequence of successive edges. The length of p is the number n (where $n > 1$) of edges in p (a path of length n is called *n-path*). Particular cases of paths are:

- Unit paths (paths of length equal to 2);
- Cyclic paths, containing one or more *pre* operators.

In addition, the path $(e_1, e_2, \ldots e_{n-1})$ is called a prefix of the path $(e_1, e_2, \ldots e_{n-1}, e_n)$ where $n > 2$.

2.3.3. Operator Predicate

Let (e, s) be a unit path and *op* an operator such that $e \in in(op)$ and $s \in out(op)$.

The operator predicate $OC(e, s)$ associated with (e, s) is a boolean expression such that:

- $OC(e, s) = true$ if *op* is the *not* operator or a relational operator.
- $OC(e, s) = not(e)$ *or* e' if *op* is an *and* operator and $in(op) = \{e, e'\}$.
- $OC(e, s) = e$ *or* $not(e')$ if *op* is an *or* operator and $in(op)$ $\{e, e'\}$.
- $OC(c, s) = true$, $OC(e, s) = c$ *and* $OC(e', s) = not(c)$ if *op* is an *ITE* operator such that $in(op) = \{c, e, e'\}$.

2.3.4. Activation Conditions

The condition upon which a data flow is transferred from the input edge to the output edge of an operator is called activation condition [11]**.** An activation condition is associated 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.

Let $p = (e_1, e_2, \ldots e_{n-1}, e_n)$ be a *n*-path, let $p_0 = (e_1, e_2, \ldots e_{n-1})$ be the prefix of p and let *op* be the last operator of p, (*i.e.* $e_{n-1} \in in(op)$ *and* $e_n \in out(op)$).

The activation condition of p is a temporal boolean expression, $AC(p)$, defined as follows [12]: If $n = 1$ then $AC(p) = true$, then the activation condition of a single edge is always true.

For a given *n*-path p, the activation condition is recursively defined as a function of the operators in it. According to the type of the operators, the following three cases may be distinguished:

- $AC(p) = AC(p')$ and $OC(e_{n-1}, e_n)$ where *op* is a boolean, relational or conditional operator and $OC(e_{n-1}, e_n)$ is the predicate associated with operator *op*.
- $AC(p) = false \rightarrow pre(AC(p'))$ where *op* is a *pre* operator. This means that the path is activated if its prefix has been activated at the previous cycle. The *fby* ($\rightarrow$) operator states that such a path cannot be activated at $t = 0$.
- If *op* is the operator *fby*(*init*; *nonInit*), and if p_{init}, $p_{nonInit}$ are respectively the prefixes of *init* and *nonInit* then the corresponding activation conditions are defined by the following two equations. The first equation (1) states that the path is activated if its prefix is activated at the initial cycle; while in the second (2), the path is always activated but for the initial cycle.

$$AC(p) = AC(p_0) \rightarrow false \quad (1)$$
$$AC(p) = false \rightarrow AC(p_0) \quad (2)$$

3. REGRESSION TESTING APPROACH FOR LUSTRE/SCADE PROGRAMS

3.1. Problem statement

Lustre/SCADE applications usually require very high quality and rigorous testing activities before deploying. During the development process, the Lustre program is often updated, so regression test should be performed to detect bugs. As for any regression testing activity, optimizing the number of executed test cases is an important issue.

Suppose that we have a Lustre program L_1 and L_2 is a new version of program L_1. Comparing to version L_1, version L_2 may:

- Remove some functions;
- Add some new functions;
- Change some functions;

– Leave functions unchanged.

The objective is to perform regression testing of version L_2 with a minimum number of test cases. The approach will be presented in next section.

3.2. The approach

The idea is to compare two versions L_1 and L_2 in order to determine the differences between them:

– The new parts that are added into L_2;

– The parts of L_1 that are changed in L_2;

– The parts of L_1 that are not affected in L_2;

– The parts of L_1 that are removed from L_2.

Instead of considering directly the two versions, we compare their two sets of paths and identify the differences between them. Then, we generate test cases that cover such differences.

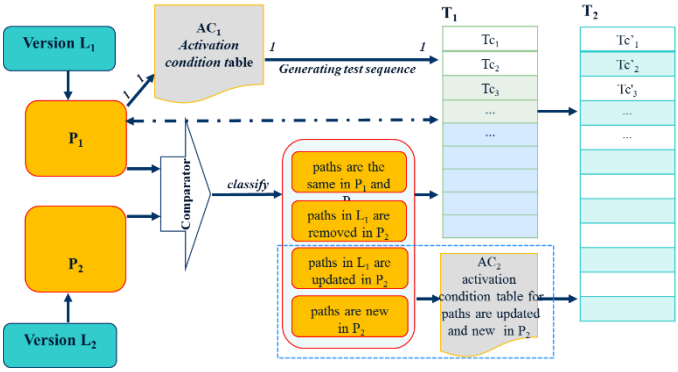

Figure 1. The overall approach of regression testing for Lustre programs

Figure 1 illustrates the approach, which consists of two following phases.

Phase 1: Generating test data for version L_1

We first determine the operator network, path table and activation condition table for version L_1. Then, we generate test cases (test sequences) covering the paths based on the activation condition table. The result of this phase is a set of test sequences T_1 and the relationship between path table P_1 and T_1.

Phase 2: Selecting and generating test data for version L_2

In this phase, we reuse the subset of test cases generated in phase 1 for testing version L_2 and we also create some new test cases covering the new paths in L_2. This phase consists of three steps as follows.

2.1) Determining path table P_2 from the operator network of version L_2.

2.2) Comparing two tables P_1 and P_2, then classifying paths. As a result, we obtain three subsets P_A, P_B and P_C:

– P_A: The paths removed in P_2 (this subset exists in P_1 but does not in P_2);

– $P_B = P_1 \cap P_2$: The subset of paths that are the same in P_1 and P_2;

– P_C: New paths in P_2.

2.3) Building the set of test cases for version L_2 based on P_A, P_B and P_C as follows:

– P_A: the subset of test cases covering P_A will not be reused in T_2;

– P_B: the subset of test cases, called T', covering P_B will be reused in T_2;

– P_C: a new subset of test cases, called T'', will be created to cover P_C.

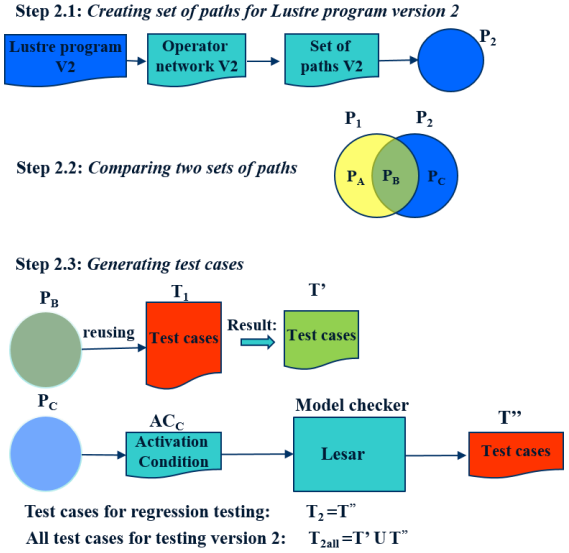

Figure 2. Generating and selecting test cases for version L_2

Figure 2 shows the steps of phase 2. As a result, $T_2 = T''$ is the set of test cases for regression testing, and $T_{2all}=T'$ U T'' is the set of test cases to test version L_2.

4. THE LUSREGTES TOOL

The LusRegTes tool is a test data generator, which consists of two main modules as follows (Figure 3).

– Module 1: generating test data for the first version;

– Module 2: selecting and generating test data for regression testing for the second version.

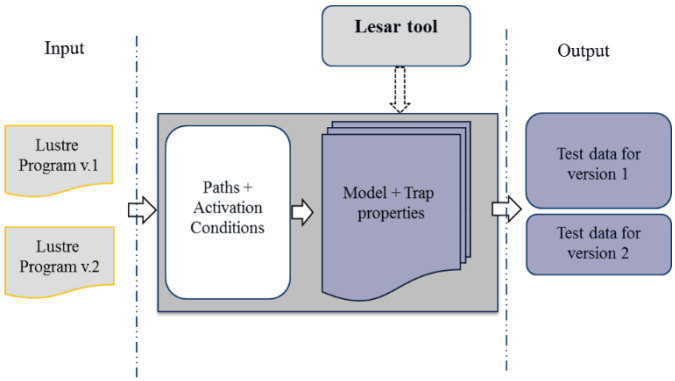

Figure 3. The LusRegTes tool

Module 1 of LusRegTes requires the following inputs: the first version of Lustre program to be analyzed, the length of path and the maximum number of loops in a path (that is, the number of times that a *pre* operator will be repeated in a path). This module automates running the Lesar tool with activation conditions.

As a result, it provides a set of test data for testing the first version. Module 1 is detailed in Figure 4.

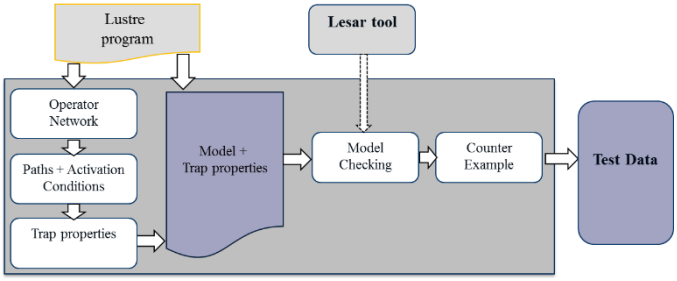

Figure 4. Module 1

Module 2 uses the results of module 1 and the new version of Lustre program as inputs. It selects and generates test data for regression testing. The module consists of three steps:

– A set of paths is determined based on the Lustre note of the new version. This set of paths will be compared with the set of paths of the previous version. The results of the comparison help identifying the trigger conditions that will be used in the next step.

– Model checker Lesar is used to generate the counter-examples.

– The counter-examples are analyzed and then test data are generated for the regression testing activity.

The entire process of Module 2 is shown in Figure 5.

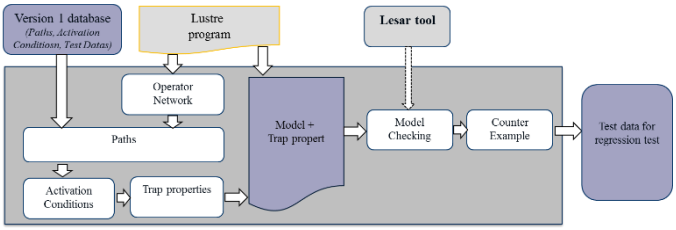

Figure 5. Module 2

The LusRegTes tool is developed in Java programming language, database MySQL and runs on the Linux operating system. Figure 6 shows a screenshot of the LusRegTes tool.

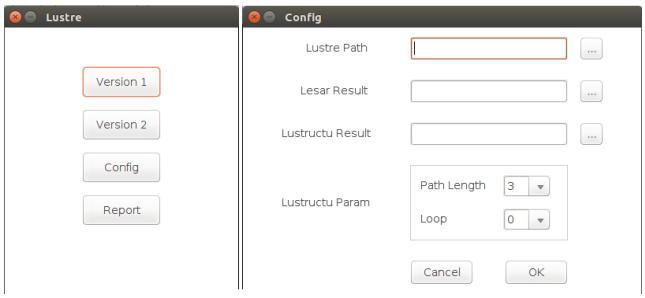

Figure 6. Screenshot of the LusRegTes tool

In the next part of this section, we will present the functions of this tool with a case study.

5. CASE STUDY

We have applied the LusRegTes tool to the Heater Controller System, which is presented in Figure 7. The Heater Controller System is a typical example of the reactive system was built by using

Lustre/SCADE. The controller's environment consists of three elements: a switch, a temperature sensor and a blower. The sensor emits three Boolean signals: *Tlow*, *Tok* and *Thigh* meaning that the temperature is: respectively, below, on or above the user-defined temperature. The controller uses the switch position (*On*) and the sensor signals to emit four control signals (*STOP*, *Deactivated*, *Hot* and *Cold*) to the blower. *STOP* means that the blower is stopped, *Deactivated* means that the blower is stand-by (no air is issued) while *Hot* and *Cold* mean respectively that hot air or cold air is issued. Two following versions of the Lustre program were developed for this system.

Version 1: Figure 7 presents the architecture of Heater Controller System.

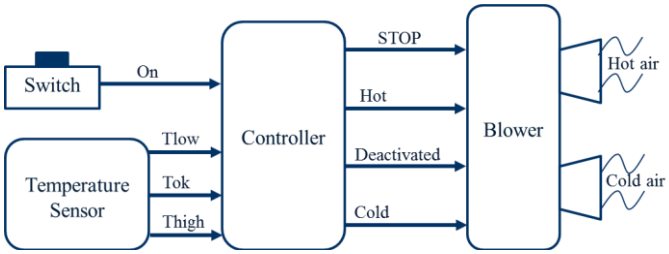

Figure 7. The Heater Controller System architecture

Version 2: Version 1 was evolved by adding two output signals *veryCold* and *veryHot*. If curent sensor is *Cold*, if *Thight* = *true*, signal is *veryCold***.** If curent sensor is *veryCold*, if *Thight* = *true*, signal is *veryCold* else if *Tlow* = *true* then signal is *Cold* (similar with *veryHot*).

When using the LusRegTes tool, the process is realized through two phases. First, module 1 will be run to generate test data for version 1 of the program (Figure 8).

Figure 8. Using module 1 to generate test data for version 1

After the first phase, we obtain the set of test cases T_1 for the Heater Controller System and the relationship between activation conditions and test cases is illustrated in Figure 9.

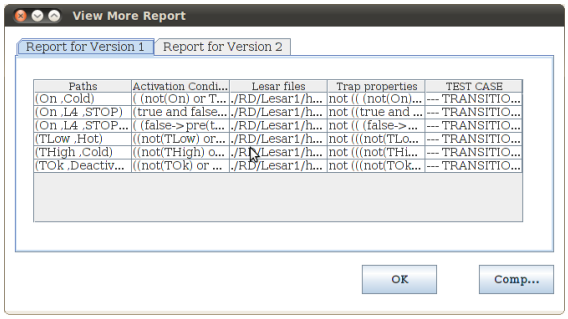

Figure 9. Activation conditions and test cases for version 1

Next, the LusRegTes tool is applied to version 2 of the Lustre program. It analyzes the paths and compares to version 1. The result of the comparison is shown in Figure 10.

- *Path A*: The paths are removed in version 2 (this subset exists in version 1 but does not in version 2).
- *Path B*: The subset of paths is the same in version 1 and version 2.
- *Path C*: The paths are new in version 2.

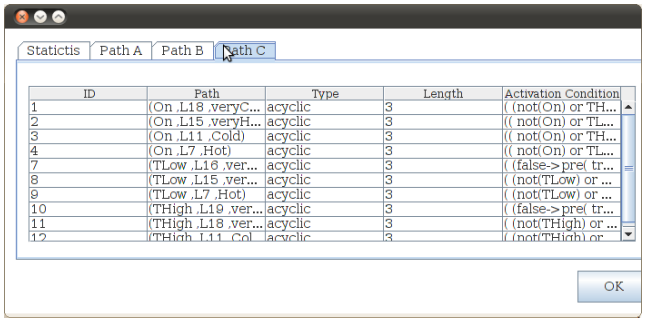

Figure 10. The result of the comparison

The activation conditions are computed for the new paths P_C. Next, the trap properties are created by negating the activation conditions. Finally, the LusRegTes tool runs the Lesar tool to generate the counter examples, which are transformed into the test cases for regression testing (Table 1).

Table 1. Set of test cases T'' for regression testing

Counter examples	Test cases	
Transition 1: not On	On=false	{(**0**, 1, 1, 1)}
Transition 1: not On	On=false	{(**0**, 1, 1, 1)}
Transition 1: On or not Tlow	On=true or Tlow=false	{(**1**, 1, 1, **0**)}
Transition 1: On or not Tlow	On=true or Tlow=false	{(**1**, 1, 1, **0**)}
Transition 1: true	Any	Any
Transition 1: Tlow	Tlow=true	{(0**,** 0, 0, **1**),
Transition 2: On or not Tlow	On=true or Tlow=false	(**1**, 1, 1, **0**)}
Transition 1: On or not Thigh	On=true or THigh=false	{(**1**, **0**, 1, 1)}
Transition 1: On or not Tlow	On=true or Tlow=false	{(**1**, 0, 1, **0**)}
Transition 1: true	Any	Any
Transition 1: Thigh	Thigh=true	{(0, **1**, 0, 0),
Transition 2: On or not Thigh	On =true or Thigh=false	(**1**, **0**, 1, 1)}

The test cases in Table 1 are used for regression testing of version 2. This allows testing only the changed parts and new parts in version 2.

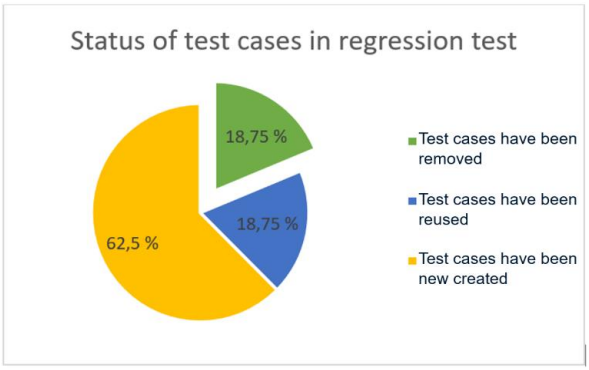

Figure 11. Chart of test cases status in regression test

Figure 11 shows the comparison of test cases in regression testing of the Heater Controller System. We can state that:

- 18,75% of test cases have been removed;
- 18,75% of test cases have been reused;
- 62,5% of test cases have been created.

6. CONCLUSION AND FUTURE WORK

In this paper, we present the LusRegTes tool for automatically generating test cases in regression testing for Lustre/SCADE programs. Test cases are generated by analyzing paths in operator network of Lustre programs. The tool helps identify only test cases needed for regression testing, so it can save costs and time effort during developing and maintaining programs, especially reactive systems developed in the Lustre/SCADE environment. The Heater Controller System that is a reactive system is used to illustrate the applicability of the tool. We are realizing more experiments with some complex systems developed by Lustre/SCADE.

REFERENCES

[1] N. Halbwachs, *et al.*, "The synchronous data flow programming language Lustre," *Proceedings of the IEEE*, 1991.
[2] S. K. Mohapatra and S. Prasad, "Test Case Reduction Using Ant Colony Optimization for Object Oriented Program," *International Journal of Electrical and Computer Engineering*, Vol. 5, No. 6, 2015.
[3] T. M. H. Le, *et al.*, "Survey on Mutation-based Test Data Generation," *International Journal of Electrical and Computer Engineering*, Vol. 5, No. 5, pp. 1164-1173, 2015.
[4] S. Yoo and M. Harman, "Regression Testing Minimisation, Selection and Prioritisation - A Survey," *Software Testing, Verificatio, and Reliability*, 2007.
[5] B. Marre and A. Arnould, "Test Sequences Generation From Lustre Descriptions: GATeL," *Fifteen IEEE International Conference on Automated Software Engineering*, IEEE Computer Society Press, pp. 229–237, 2000.
[6] E. Jahier, *et al.*, "Case Studies with Lurette V2," *International Journal on Software Tools for Technology Transfer*, Vol. 8, No. 6, 2006.
[7] L. du Bousquet, *et al.*, "Lutess: A Specification-Driven Testing Environment for Synchronous Software," *Proceedings of 21st International Conference on Software Engineering*, ACM, 1999.
[8] B. Seljimi and I. Parissis, "Automatic generation of test data generators for synchronous programs: Lutess v2," *Proceeding of Workshop on Domain specific approaches to software test automation*, 2007.
[9] C. D. Trinh, *et al.*, "A regression testing approach for Lustre/SCADE programs," *The Sixth International Symposium on Information and Communication Technology* (*SOICT 2015*), 2015.
[10] Y. L. Nancy and J. Wahi, "An overview of regression testing," *ACM SIGSOFT Software Engineering Notes*, 1999.
[11] A. Lakehal and I. Parissis, "Lustructu: A tool for the automatic coverage assessment of Lustre programs," *Proceedings of 16th IEEE International Symposium on Software Reliability Engineering*, 2005.
[12] A. Lakehal and I. Parissis, "Structural Coverage Criteria for Lustre/SCADE Programs," *Software Testing, Verification and Reliability*, John Wiley and Sons Ltd, Vol. 19, pp. 133–154, 2009.