HAL
open science

Mutation analysis to evaluate Lustre program specifications in the context of model-checking

Lydie Du Bousquet

To cite this version:

Lydie Du Bousquet. Mutation analysis to evaluate Lustre program specifications in the context of model-checking. [Research Report] RR-LIG-007, 2010. hal-00953404

HAL Id: hal-00953404

https://inria.hal.science/hal-00953404v1

Submitted on 7 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Les rapports de recherche du LIG

Mutation analysis to evaluate Lustre program specifications in the context of model-checking

LYDIE DU BOUSQUET, Associated Professor, LIG, Grenoble University (UJF),

RR-LIG-007
juillet 2010

http://rr.liglab.fr

ISSN 2105-0422

L I G

Mutation analysis to evaluate Lustre program specifications in the context of model-checking

Lydie du Bousquet
Laboratoire d'Informatique de Grenoble (LIG)
Universtités de Grenoble (UJF)
BP72, 38402 St Martin d'Hères Cedex, France
lydie.du-bousquet@imag.fr

Abstract

Mutation analysis is a technique used to evaluate the adequacy of a test set with respect to a fault model. This approach is mainly used for unit testing evaluation. It is also used to produce new test data.

In this paper, we apply mutation analysis in order to evaluate the accuracy of a specification in the context of reactive program verification by model-checking.

1 Introduction

A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. It implies running the software item in predetermined conditions (input selection), analyzing the obtained results, and identifying errors [20]. Testing can never completely establish the correctness of a program. For this reason, several methods have been proposed to increase confidence with respect to the test set provided, among which code coverage analysis or mutation analysis. Code coverage describes the degree to which the source code of a program has been tested. There are a number of coverage criteria, among which statement coverage, condition coverage, path coverage, def-use chains coverage [15, 24, 22].

Mutation analysis has been introduced by DeMillo in 1978 [6]. Its main purpose is to evaluate the quality/adequacy of a test set with respect to a fault model. It is mainly used for unit testing evaluation. Since 1978, mutation analysis has been widespread, improved and evaluated. Is has been adapted for several programming languages [5, 1, 19, 23]. Andrews *et al.* have demonstrated that mutants can provide a good indication of the fault detection ability of a test suite [2].

Mutation analysis has also been applied to evaluate model-checking. To prove that a system satisfies a property, a model-checker explores the state space of the system, and check that a property is true in all the accessible states. If it is not the case, a counter-example is provided. The verification by model-checking is exhaustive in the sense that all accessible states are explored. However, the verification is limited by the properties to be checked: if the property set does not cover all behaviors of the system, some faults may remain. In order to evaluate how a set of properties fully verifies a model, T.-C. Lee and P.-A. Hsiung have proposed to use mutation analysis [17]. The work have been carried out on timed automata. Six fault models were used and six associated mutation scores (called *mutation coverage*) were defined.

In the following, we focus on the validation of LUSTRE programs by model-checking. LUSTRE is one of the synchronous languages [13, 12]. During the last two decades, the growing interest in synchronous languages from large companies has initiated significant contributions to the practical validation problem of synchronous software [4].

Section 2 presents mutation analysis for test and illustrate how mutation analysis can be used for model-checking. Section 3 is dedicated to the presentation of LUSTRE language and illustrates the use of mutation-analysis in the context of model-checking. In section 4, we analyze some examples and discuss of the approach. Section 5 concludes.

2 Mutation analysis

2.1 General principles for test set evaluation

Mutation analysis consists in introducing a small syntactic change in the source code of a program in order to produce a *mutant* [6] (for instance, replacing one operator

by another or altering the value of a constant). Then the mutant behavior is compared to the original program. If a difference can be *observed*, then the mutant is marked as *killed*. If the mutant has exactly the same observable behavior as the original program, it is *equivalent* with respect to the fault model.

The original aim of the mutation analysis is the evaluation of a test set. To do that, one has to produce all mutants corresponding to a predefined fault model. If the test set can kill all non-equivalent mutants, the test set is declared *mutation-adequate*. This means that the tests are able to discriminate the behaviors of the faulty programs from the original program.

Adequacy of the test set is evaluated thanks to the *mutation score* (also called *adequacy score*). The mutation score is the percentage of non-equivalent mutants that are killed. A test set is *mutation-adequate* if the mutation score is equals to one. Andrews *et al.* have demonstrated that mutation analysis can provide a good indication of the fault detection ability of a test suite [2].

Mutation analysis is usually used to evaluate the adequacy of test set produced during unit testing. It has been adapted for several programming languages such as ADA, Java, C, VHDL, Petri-Nets, Final State Machine (FSM), Statecharts, and Estelle [25, 5, 23, 18, 1, 21, 11, 7].

2.2 Using mutation for specification analysis

Classical mutation analysis compares the mutant outputs to the original program outputs to decide if a mutant is killed or not by a test set. This allows evaluating how the test-set is good to detect small syntactical changes. When a specification is available, it is also possible to compare the mutant output to this specification in order to check that the specification is acute enough to detect the same types of faults.

In the case of model-checking, the specification is composed of a set of properties. A model-checker verifies if the properties hold in all accessible states of the program. But, even if it is the case, some errors may remain: the properties may not cover all the program. For instant, demonstrating that the program satisfies the *true* property does not bring useful feedback on the program correctness.

Using mutation analysis to evaluate how much a set of properties *covers* the program was originally proposed by T.-C. Lee and P.-A. Hsiung [17]. It was evaluated on timed automata. Hereafter is a short presentation of this work.

When a mutant does not satisfy the specification, it is said to be *covered*. The fraction of mutations detected by a specification determines its completeness. Some *coverage metrics* can be defined for the differents fault models. Its value is computed by the ratio of the number of covered mutations to the total number of non-equivalent mutations obtained with the chosen fault model.

In [17], six fault models were used and six mutation coverage metrics were deduced. Each fault model characterizes a different aspect of a system state graph and identifies different characteristic of the properties (expressed in TCTL). The mutation fault models are (1) *mutated initial* (concerns the initial state), (2) *delayed transition* (to delay a mode transition), (3) *shuttering mode* (to add self-loop transition to a mode), (4) *skipped mode* (to make a non-initial mode unreachable), (5) *removed transition* (deletes a transition from the state graphe), (6) *mutated invariant* (invariant of mode is changed).

In the following, we are going to use the same vocabulary than traditional mutation analysis. We will say that a mutant is *killed by the specification* if the mutant does not satisfy the specification (instead of covered). By extension, the *mutation-score of a specification* is the number of non-equivalent mutants that does not satisfy the specification (instead of coverage metric). Thus, if all mutants are killed by the specification, we can say that the *specification* is *mutation-adequate*.

3 LUSTRE program: validation and verification

3.1 Brief presentation of LUSTRE language

LUSTRE [13, 12] is a synchronous declarative data flow language. The synchronous hypothesis considers the program reaction time to be negligible with respect to the reaction time of its environment.

The synchronous data flow approach consists in presenting a temporal dimension into the data flow model. A flow or stream (basic entity) includes two parts: a sequence of values of a given type, and a clock representing a sequence of instants (on the discrete temporal scale).

A LUSTRE description, structured in a network of nodes, represents the relations between the inputs and the outputs of a system. These relations are expressed by means of operators (nodes or basic operators), of intermediate variables and of constants.

A node is defined by a set of equations. Any local variable or output must be defined by one and only one equation. The equations can be written in any order without changing the behaviors of the program.

LUSTRE offers usual arithmetic, boolean and conditional operators and two specific operators: `pre`, the "previous" operator, and $->$ the `followed-by` operator[1]. Fig. 1 gives a LUSTRE program implementing a simple stopwatch

[1] Let E and F be two expressions of the same type denoting the sequences $(e_0, e_1, ..., e_n ...)$ and $(f_0, f_1, ..., f_n, ...)$; **pre**$(E)$ denotes the sequence $(nil, e_0, e_1, ..., e_{n-1} ...)$ where nil is an undefined value. $E -> F$ denotes the sequence $(e_0, f_1, ..., f_n ...)$.

```
node chrono (raz : bool)
returns (n : int);
let
  n = 0 -> if raz then 0 else (pre(n)+1);
tel;
```

Figure 1. A simple LUSTRE program

(chronometer). The output `n` is set to 0 at the first step or when the `raz` input is true. It is incremented by one otherwise: the value of `n` at the current top is equal to the value of `n` at the previous top (pre `n`) plus one. The `current` and `when` operators are two other temporal specific operators of LUSTRE used for sampling signals.

An execution trace of a LUSTRE node is a sequence of input and output values. An execution trace is given Fig 2 for `chrono` node. When output values depend on current and previous inputs, the node is *sequential*. It deals with temporal operators. Otherwise, it is *combinatorial*. `chrono` is a sequential node.

	t_0	t_1	t_2	t_3	t_4	t_5
raz	0	0	0	1	1	0
n	0	1	2	0	0	1

Figure 2. An execution trace for `chrono`

3.2 Running Example: a temperature control system

Let us consider the following example of a temperature control system given in [3]. This system is composed of one heater, one air conditioner, some temperature sensors and one ON/OFF switch. The program has four inputs:

- *Running* is a signal active when the ON/OFF switch of the device is to the ON position.
- *BelowDesiredTemp*, *TempOk* and *AboveDesiredTemp* are signals issued by the temperature sensors. They are active when the current temperature is respectively lower, equal or higher than the desired temperature.

The program has four outputs, which are mutually exclusive signals. They set the current mode of the system:

- *OFF*, meaning that the system is turned off.
- *INACTIVE*, meaning that the system is on but neither the heater nor the air conditioner is on.
- *HEAT* and *AC*, meaning that either the heater or the air conditioner, respectively, is on and controlling the temperature.

The assumption made here on the environment is that exactly one of the three inputs *BelowDesiredTemp*, *TempOK* and *AboveDesiredTemp* is true at the same time. Another one is that the temperature cannot raise (resp. fall) from a value below (resp. above) the desired temperature to a value above (resp. below) it without reaching in the meantime the desired temperature.

It is possible to express the environment assertions and the expected safety properties of a system in LUSTRE, since it is also considered a temporal logic of the past. For the environment properties, the two assertions can be formalized as

1. (AboveDesiredTemp or BelowDesiredTemp or TempOK) and #(AboveDesiredTemp, BelowDesiredTemp, TempOK)
2. `once_from_to`(TempOK, BelowDesiredTemp, Above- DesiredTemp) and `once_from_to`(TempOK, AboveDesiredTemp, BelowDesiredTemp)

where # is a built-in logical operator ensuring that no more than one of the boolean expressions in its argument list is true at the same time. `once_from_to`(A,B,C) is a user-defined temporal operator returning a true value when the event A has occurred at least once between two subsequent occurrences of B and C. Similarly `always_from_to`(A,B,C) returns a true value when the event A continuously occurs between two subsequent occurrences of B and C. The definition of such temporal operators in LUSTRE can be found in [14]. A LUSTRE program for controlling that system is presented in [16] and illustrated Fig 3.

3.3 Lesar, a model-checker for Lustre

LESAR is a model-checker for LUSTRE. Given a program P and a safety property S, LESAR checks that S is true in all states of the 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 (see Fig. 4). 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.

For instance, for the temperature control system, one can express the following safety properties:

```
node Clim(Running, BelowDesiredTemp, TempOk, AboveDesiredTemp: bool)
returns (OFF, INACTIVE, HEAT, AC : bool);
let
  OFF = not Running;
  INACTIVE = Running and TempOk;
  HEAT = (Running and BelowDesiredTemp) ->
         (pre (OFF or INACTIVE or HEAT) and Running and BelowDesiredTemp);
  AC = (Running and AboveDesiredTemp) ->
       (pre (OFF or INACTIVE or AC) and Running and AboveDesiredTemp);
tel
```

Figure 3. A temperature control system in LUSTRE

```
node VP(Running, BelowDesiredTemp, TempOk, AboveDesiredTemp: bool)
returns (res: bool);
var  OFF, INACTIVE, HEAT, AC : bool;
let
-- environment constraints
  assert ((AboveDesiredTemp or AboveDesiredTemp or TempOk) and
           #(AboveDesiredTemp,AboveDesiredTemp,TempOk) );
  assert (once_from_to(TempOK,BelowDesiredTemp,AboveDesiredTemp) and
          once_from_to(TempOK,AboveDesiredTemp,BelowDesiredTemp));

-- program to be verified
  (OFF, INACTIVE, HEAT, AC) =
      Clim(Running, BelowDesiredTemp, TempOk, AboveDesiredTemp);

-- safety properties
  res = implies(OFF, not Runnning) and
        implies(INACTIVE, Runnning and TempOk) and
        implies(OFF, Runnning and BelowDesiredTemp) and
        implies(OFF, Runnning and AboveDesiredTemp);
tel
```

Figure 4. A verification program for the temperature control system

- The system is off only when the appliance is off.
  `implies(OFF, not Runnning)`
- The system is inactive only if the appliance is on and if the current temperature is equal to the desired temperature.
  `implies(INACTIVE, Runnning and TempOk)`
- The system is heating only if the appliance is on and if the current temperature is below than the desired temperature.
  `implies(OFF, Runnning and BelowDesiredTemp)`
- The system is cooling only if the appliance is on and if the current temperature is lower than the desired temperature.
  `implies(OFF, Runnning and AboveDesiredTemp)`

LESAR can prove that the program `Clim` satisfies the given properties, in less than a second. But it is difficult to determine if the properties describes all the program behaviors or only a part of them.

3.4 Mutation analysis for LUSTRE

Alien-V[2] is a tool we built for mutating LUSTRE nodes [8]. This tool was produced within collaboration between LSR (team VASCO) and LCIS (team VALSYS). The multi-language mutant generator for VHDL and C developed by LCIS (Alien) [21] was extended to LUSTRE.

Alien-V has a mutation operator table as input. It is possible to adapt it to define specific mutation operators. Arithmetic/Logical/Relational Operator Replacement are defined by default. CSR (Constant for Scalar variable Replacement) and SCR (Scalar for Constant Replacement) have to

[2]Lustre also means “5 years long” in French.

be manually parameterized for each program.

LESAR can be used to detect equivalent mutants produced for a LUSTRE program. The verification program should carry out the comparison of the mutant and original program. When some environment assertions are provided with the original program, it is possible to consider the mutant-equivalency with respect to the environment description (using the `assert` operator) or without considering environment (unconditional equivalence).

For the program `Clim`, Alien-V produces 193 mutants. `Clim` uses 8 boolean variables (input or output). There are 17 "instances" of these variables inside the program where a changed can be done[3]. Each of them can be replaced by the 7 other variables, the `true` value, the `false` value and the negation of the variable (170 mutants). There are also 11 instances of a logical operator, which can be replaced by two other operators (22 mutants). Among those 193 mutants, 12 of them are syntactically incorrect. 4 mutants are equivalent without considering environment (unconditional equivalence) and 18 of them are equivalent with respect to the environment description.

As presented previously, the `Clim` specification is composed of 4 safety properties. `Clim` satisfies those properties under the environment constraints. We use the LESAR model-checker in order to check if the 163 non-equivalent mutants satisfy or not those properties. The verification program proposed Fig. 4 was used with the mutated implementations of `Clim`. 86 non-equivalent mutants satisfy the 4 safety properties under the environment constraints. This means that those mutants were not killed by the specification. The mutation score of the specification is thus 86/163=0.52.

In fact, the specification provided for the temperature control system is composed two other properties: if the appliance is on, as soon as the current temperature goes below (resp. above) the expected temperature, the system will start heating (resp. cooling). These properties are expressed in LUSTRE with respect to the previous step.

- `implies(pre (Runnning and BelowDesiredTemp),HEAT)`
- `implies(pre (Runnning and AboveDesiredTemp), AC or pre AC)`

If the mutation analysis is carried out against the complete specification, we obtain that all mutants are killed (they do not satisfy the properties). The mutation score of the complete specification is thus 1.

[3]A mutation of the variables on the left side of the equal signe is not done since it produce syntaxtical incorrect programs.

4 Case studies and discussion

4.1 Case studies

We have applied this approach on three examples: a *subway U-turn section* (UMS) [14], a *water supplying system* (Supplying) [10], a *mouse controller* (Mouse) [9]. The results are detailed hereafter and a summary is presented tables 1 and 2. For this 3 simple programs, the specification mutation score is low (below 0.5).

The Mouse program is an interface between a 3-button device and an application program. It transforms "physical clicks" on a mouse into "logical clicks" usable by the application program. A physical click corresponds to the action of clicking one button. A logical click is defined by a couple of values. A logical click is simple, double or triple. A triple click can occur only during a specified time lap. A triple click results from three clicks on the same button in a row in less than T ticks. A double (resp. simple) click is obtained when the same button is pressed twice (resp. once) in a row in less than T ticks. A triple click has a priority over simple and double clicks. A double click has a priority over a simple click.

The inputs of the conversion program are the button numbers. The outputs of the conversion program are couples of values the click type and the button number. Three safety properties were expressed, stating that (1) a logical click can only be simple, double or triple. (2) a logical click correspond to only one button number and (3) a type of logical clic is always associated to a button number (and respectively). The environment assertions state only that it is not possible to press two (physical) buttons simultaneously.

The property set is incomplete since the time lap is not specified. Moreover, the output values are not specified with respect to the input ones. The reason why is that Lesar carry out an abstraction an integer value is used to compute the time elapsed between a physical and the associated logical click. It then provides "false" counter-examples due to this abstraction for any properties dealing with the computation of the time lap. This has been confirmed by the mutation analysis. 492 mutants were produced with ALien-V, among which 478 are not equivalent. Only 214 are killed by the properties, which represents a mutation score of 0.44.

The UMS program controls the U-turn section of a subway. A U-turn section allows trains to switch from one track to the other, and to go back in the opposite direction. A U-turn section 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, then transit on B and wait again for S to connect B with C before going back on C.

```
node oracle_ps( on_A, on_B, on_C,
                ack_AB, ack_BC: bool)
returns (res : bool);
var
-- UMS outputs
  grant_access, grant_exit, do_AB, do_BC: bool;
-- for safety property
  no_collision, exclusive_req : bool;
  no_derail_AB, no_derail_BC : bool;
-- for intermediate computations
  empty_section, only_on_B : bool ;
let

-- assertions
  assert not(ack_AB and ack_BC);
  assert always_from_to (ack_AB, ack_AB,do_BC);
  assert 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 equations
   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);

-- Safety 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);

res = no_collision and
      exclusive_req and
      no_derail_AB and
      no_derail_BC ;
tel
```

Figure 5. The U-turn system: assertions, code and safety properties together

The UMS Lustre program has 5 inputs and 4 outputs. It is a combinatorial node, composed of 6 equations. The specification is composed of 4 safety properties and 8 environment assertions. The code of the verification program is preproduced Fig. 5. Alien-V produces 174 syntactically correct mutants. 171 of them are not-equivalent (under the environment assertions). Among these, 80 are killed by the properties under the environment assertions. This means that the mutation score is 0.46.

This result may appear surprising: the program is very simple, the environment assertions are quite restrictive and all outputs are expressed in the properties. However, properties mostly state what should not be done (specification of some incorrect outputs) but do not specify what should be done (specification of the required behavior). For instance, a system that does “nothing” is safe. That is the case for 60 mutants that have at least one output always false and satisfy the properties. In this case, the property have been chosen to allow implementation freedom.

The water supplying system The water supplying system is composed of a artesian well, a tank, and two pumps. The second pump is used to distribute water in the building. It is started when at least one tap is turned on. It is stopped when all taps are turned off. The first pump is used to fill the tank when the water level in the tank is too low. Both pumps are equipped with temperature sensor. If a rise in temperature is observed, pump should be stopped. An information board, inside the building, is composed of 4 warming lamps. A green lamp indicates that the system is OK. Two red lamps indicate if the first or the second pump are off. A last red lamp indicates if the tank low level is reached.

The system has to control the pumps and the information broad. Its inputs consist of 9 booleans: the 3 levels of the tanks, the temperature sensors of the two pumps the manual commands for re-activation the sensor to detect dry pumping and an input indicates if one tap is open. The outputs consist of 6 booleans: the commands for the 2 pumps and the 4 lights. There are 4 environment assertions indicating the initial state of the pump and the evolution of the water level. 4 safety properties were defined.

Alien-V produces 285 mutants syntactically correct. 226 mutants were not equivalent. Among these, only 52 kill the specification (the mutation score is 0.23). Two situations can be observed. First, some parts of the system are not specified. For instance, 60 non-equivalent mutants deal with the values of the 4 lights. None of them are killed since the safety properties are not concerned by the values of the light. Second, some parts of the system (the control of the pumps) are specified as it is done for the U-Turn: incorrect behaviors are specified but not the correct ones. Here again, this choice was done to allow implementation freedom.

Program	# node	# input	# output	# env.	# safety
Conditioner	1	4	4	4	6
UMS	1	5	4	8	4
Supplying	8	9	6	4	4
Mouse	1	3	6	1	3

Table 1. Quantitative elements about the programs

Program	Conditioner	UMS	Mouse	Supplying
# mutants	181	174	492	285
# eqv (under env. assertions)	18	3	14	59
# killed	163	80	214	52
Score	1	0.46	0.44	0.23

Table 2. Quantitative elements about the mutants

4.2 Discussions

The application of the approach proposed by [17] on these 4 examples help us to identify some elements missing in the different specifications. Several types of situations were observed. Some properties were not described, because

1. they were simply forgotten (possibly for the demonstration purposes),
2. they concerned outputs that were judged to be non-critical,
3. they are not provable by the model-checker (here, they concerned liveness and abstraction on the integers),
4. some implementation freedom was left.

In order to detect situations such as the first one, it is interesting to apply mutation analysis. Admittedly, it is expensive but it provides a real feedback.

About the second point, not specifying non-critical outputs is often done on purpose. If it is the case, one may want to drop mutation analysis for those parts. In order to do that, we would like to couple mutation analysis with a ”slicing” approach. The idea would be to extract only critical parts (here equations) of the application out of the total code and restrict the mutation analysis on these parts. Typically, for the *Supplying* example, the critical parts deal with the control of the pumps. Being able to provide mutations on the associated equations would result to study less mutants (3/4 in this case).

For the third point, if properties that can not be proved are required, test can be applied. The mutation analysis can thus be applied at two steps. It will first help to identify the lacks of the specification. It will then help to evaluate the relevance of the test data. The advantage here is that the same mutants are used.

Finally, for the fourth point, when the specification allows some implementation freedom, the mutation analysis will provide possibly a low mutation score. The user may try to increase the mutation score by completing the specification in a wrong way, for instance, by duplicating the code. This could have been done for the U-Turn example, since the code is very “simple”.

5 Conclusion and perspectives

Mutation analysis has been proposed to evaluate the adequacy of a test set with respect to a fault model. It has been proposed also to evaluate how much a specification “covers” a program in the context of verification by model-checking. The original work concerned timed automata specified in TCTL. Here, we have applied the same idea for Lustre programs. We have studied 4 programs and their specification published in the litterature. The specifications are safety properties also expressed in LUSTRE (which can be considered as a temporal logic of the past) and proved with the LESAR model-checker.

This work allows us to drive the following conclusion. *Yes*, mutation analysis help to discover some lack in the specification: for three of the studied programs the mutation score of the specification is very low, due to missing properties. *But*, safety properties are sometime introduced with the idea to let some implementation freedom (several implementations can be accepted). In this situation, mutation score should be used carefully. It could be easy to add properties that are a copy of the code, just to increase the mutation score. Such a practice would improve superficially the confidence within the software, without improving its quality.

Our perspective of this work is to couple the mutation analysis with a slicing approach, in order to restrict the analysis only on parts that are concerned with the properties. This would decrease the cost of the mutation analysis (by reducing the time spent in the analysis) and provide a more precise verdict about the specification relevance. Moreover, it will reduce the risk to artificially over-specified non-critical parts.

References

[1] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Soft. Eng. Research Center, Dep. of Computer Science, Purdue Univ., Indiana, 1989.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th International Conference on Software Engineering (ICSE'05)*, pages 402–411, St. Louis, Missouri, USA, May 2005. ACM.

[3] J. Atlee and J. Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, pages 24–40, january 1993.

[4] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[5] R. DeMillo, D. Guindi, K. King, M. M. McCracken, and J. Offutt. An extended overview of the mothra software testing environment. In *2nd Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Canada, July 1988.

[6] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[7] S. do Rocio Senger de Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. Lopes de Souza. Mutation testing applied to estelle specifications. In *HICSS*, 2000.

[8] L. du Bousquet and M. Delaunay. Mutation analysis for lustre programs: Fault model description and validation. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 176–184, Sept. 2007.

[9] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[10] B. M. Duc. *Conception et modélisation objet des systèmes temps réel*. Eyrolles, 1998.

[11] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Mutation Testing applied to Validate Specifications Based on Statecharts. In *10th International Symposium on Software Reliability Engineering*, Boca Radon, FL, USA, 1999.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et Vérification des Systèmes Réactifs : le langage LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[14] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE. *IEEE Transactions on Software Engineering*, pages 785–793, September 1992.

[15] W. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, SE-2(3):37–44, 1976.

[16] A. Lakehal, F. Ouabdesselam, I. Parissis, and J. Vassy. Models for synchronous software testing. In *First International Workshop on Model, Design and Validation, 2004*, pages 41–50. IEEE, November 2004.

[17] T.-C. Lee and P.-A. Hsiung. Mutation coverage estimation for model checking. In *Second International Conference on Automated Technology for Verification and Analysis (ATVA)*, volume 3299 of *Lecture Notes in Computer Science*, pages 354–368, Taipei, Taiwan, ROC, November 2004. Springer.

[18] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.

[19] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: a mutation system for java. In *28th International Conference on Software Engineering (ICSE'06)*, pages 827–830, Shanghai, China, May 2006. ACM.

[20] G. Myers. *The Art Of Software Testing*. Wiley-Interscience, 1979.

[21] T. B. Nguyen and C. Robach. Mutation Testing Applied to Hardware: the Mutants Generation. In *Proceedings of the 11th IFIP International Conference on Very Large Scale Integration*, pages 118–123, Montpellier, France, Dec. 2001.

[22] S. Ntafos. A Comparison of Some Structural Testing Strategies. *IEEE Transactions on Software Engineering*, pages 868–874, june 1988.

[23] A. J. Offutt, J. Voas, and J. Payne. Mutation Operators for Ada. Technical Report ISSE-TR-96-06, George Mason University, 1996.

[24] S. Rapps and E. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, pages 367–375, april 1985.

[25] W. E. Wong. *Mutation Testing for the New Century*. Kluwer Academic Publishers, June 2001.