Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 203 (2008) 35–48
www.elsevier.com/locate/entcs

Towards Mutation Analysis for Lustre Programs

Lydie du Bousquet[1] and Michel Delaunay[1]

Universités de Grenoble
LIG (UJF-CNRS)
BP72, 38402 St Martin d'Hères Cedex, France

Abstract

Mutation analysis is usually used to provide indication of the fault detection ability of a test set. It is mainly used for unit testing evaluation. This paper describes mutation analysis principles and their adaptation to the Lustre programming language. Alien-V, a mutation tool for Lustre is presented. Lesar model-checker is used for eliminating equivalent mutant. A first experimentation to evaluate Lutess testing tool is summarized.

Keywords: Mutation analysis, data-flow programming language, Lustre, test, Lesar, Lutess.

1 Introduction

In recent years, software quality assurance has received growing attention since it is recognized that a high level of quality is necessary to make both the client and the supplier confident in the product, and to reduce the maintenance costs. Testing is the most used technique for checking whether the required quality has been achieved. The testing purpose is to uncover the largest possible number of faults which have crept into the product during the construction stages. Due to the increasing complexity of software, the testing efficiency/quality has to be controlled.

During the last decade, the growing interest in synchronous languages from large companies has initiated significant contributions to the practical validation problem of synchronous software. Contrary to many other areas, and thanks to the rigorous mathematical semantics of this approach, much of current synchronous software testing theory and practice is not built on wishful thinking: several specification-based testing methods have been designed, implemented and have shown to be

[1] Email: {lydie.du-bousquet,michel.delaunay}@imag.fr

doi:10.1016/j.entcs.2008.05.009

effective at revealing errors [13,33,21,4,24]. Furthermore, all these methods allow to automate the test data generation process.

In this context, our previous works on testing concerned validation of Lustre programs. For this purpose, we have elaborated and we are currently improving Lutess a testing tool dedicated to synchronous programs [13,32,35]. Lutess produces randomly and dynamically test sequences. A natural question is then to evaluate the "quality" of the test data produced.

Mutation analysis has been introduced by De Millo in 1978 [11]. 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. The original work concerns Fortran programs. Since 1978, mutation analysis has been widespread, improved and evaluated [10,28,15,23,34,36]. Briand *et al.* have demonstrated that mutants can provide a good indication of the fault detection ability of a test suite [3].

In the following, section 2 details mutation analysis. Sect. 3 describes the adaptation of mutation operator for Lustre and introduces our mutation tool. Sect. 4 deals with equivalent mutant detection using the Lesar model-checker. Sect. 5 describes a first evaluation of Lutess testing tool. Sect. 6 concludes and draws some perspectives.

2 Mutation analysis

2.1 Principles

Mutation analysis consists in introducing a small syntactic change in the source code of a program in order to produce a *mutant* [11] (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*.

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 behavior of all 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 killed. For a program P, let M_T be the total number of mutant produced with respect to a particular fault model F. Let M_E and M_K be the number of equivalent and killed mutants. The mutation score of the test set T with respect to the fault model F is defined as:

$$MS(P,T,F) = \frac{M_K}{M_T - M_E}$$

A test set is *mutation-adequate* if the mutation score is equals to 1 [2]. Briand *et*

[2] *Mutation testing* aims at producing tests until the maximal mutation score is obtained.

Mutation operator	Description	Levels
AAR	array reference for array reference replacement	CCA
ABS	absolute value insertion	PDA
ACR	array reference for constant replacement	CCA
AOR	arithmetic operator replacement	PDA
ASR	array reference for scalar variable replacement	CCA
CAR	constant for array reference replacement	CCA
CNR	comparable array name replacement	CCA
CRP	constant replacement	PDA
CSR	constant for scalar variable replacement	CCA
DER	DO statement end replacement	SAL
DSA	DATA statement alteration	PDA
GLR	GOTO label replacement	SAL
LCR	logical connector replacement	PDA
ROR	relational connector replacement	PDA
RSR	RETURN statement replacement	SAL
SAN	statement analysis (replacement by TRAP)	SAL
SAR	scalar variable for array reference replacement	CCA
SCR	scalar for constant replacement	CCA
SDL	statement deletion	SAL
SRC	source constant replacement	CCA
SVR	scalar variable replacement	CCA
UOI	unary operator insertion	PDA

Table 1
Mutation operator types for Fortran 77

al. have demonstrated that mutation analysis can provide a good indication of the fault detection ability of a test suite [3].

Mutation analysis relies on two assumptions. The first one is called *the programmer competent hypothesis*. It assumes that the programs are "nearly correct", that is to say, mostly correct with possibly simple faults. The second assumption is *the coupling assumption*. It assumes that a test set covering simple faults is able to detect more complex ones.

2.2 Tools for mutation

Mutation analysis is usually used to evaluate the adequacy of test data set produced during unit testing. It has been adapted for several programming languages, and lots of tools have been proposed [36]. For instance, Mothra tool supports Fortran 77 and ADA [10,31]. MuJava [23] and JMutator [3] are tools for Java. C-Patrol system is for C [1], NMutator for C#[3], Alien for VHDL [26]. Mutation analysis was also applied to Lustre [25], Petri-Nets, Final State Machine (FSM), Statecharts, and Estelle [17,16,15,12].

2.3 Mutation operators

The key of mutation analysis is the fault model. Fault model is expressed as a set of *mutation operators*. The original mutation operator set was proposed for Fortran

[3] http://www.inria.fr/rapportsactivite/RA2002/triskell/module7.html

77. It was derived from studies of programmer errors. This mutation operator set has been refined during more than 15 years [10,28]. It is given Table 1.

Twenty-two operators were defined. Those operators were classified into eight classes and three levels (SAL, PDA and CCA) [10]. Statement AnaLysis (SAL) replace each statement by a TRAP [4], by a CONTINUE or by a RETURN (for subprogram). It also replaces the (target) label in each GOTO and each DO statement.

The Predicate and Domain Analysis (PDA) takes the absolute value or the absolute negative value of an expression. It replaces one arithmetic/relational/logical operator by another, inserts a unary operator preceding an expression, alters a value of a constant or alter a DATA statement.

Coincidental Correctness Analysis (CCA) replaces a scalar variable, an array reference or a constant by another scalar variable, array reference or constant. It also replaces a reference to an array name by a reference of another array name.

2.4 Weaknesses of mutation analysis

Beyond the relevance of the mutation operators, the two main weaknesses of mutation analysis are (1) the cost and (2) equivalence decision.

Mutation cost

Mutation analysis is generally very expensive: lots of mutants are produced. Time is required to execute them in order to kill them. The number of mutant produced depends on the fault model and the program. Budd found that the number of mutants is roughly proportional to the number of reference times the number of data objects [8]. Acree *et al* estimate the number of mutants to be on the order of the square of the number of source lines [2].

Two main strategies have been proposed to reduce this cost: weak and selective mutation. *Weak mutation* consists in comparing *internal states* (instead of outputs) of both program and mutant in order to detect differences. The observation can be done after the execution of the faulty expression, instruction, basic block and program. Weak mutation testing requires to produce less test that classical (strong) mutation.

N-selective mutation is mutation omitting the N most productive mutation operators. In [30], 2-selective was defined by omitting SVR (scalar variable replacement) and ASR (array reference for scalar variable replacement) operators. 4-selective mutation also omits CSR (scalar for constant replacement) and SCR (scalar for constant replacement).

Equivalence decision

A mutant which has exactly the same behavior as the original program is considered to be equivalent to the orignal program. Deciding if a mutant is equivalent to the program is an important step before computing the mutation score. Otherwise, it would not be possible to reach a mutation score of 1.

[4] executing the TRAP will kill the mutant.

At the beginning of mutation analyis, equivalence decision was a complete manual process. It has been proved that "in general there cannot be a complete algorithmic solution to the equivalence problem" [27]. However, some works have been done to detect equivalent mutants automatically as much as possible. For instant, compiling technics or domain-constraints analysis were used [27,29], but they detect only a part of equivalent mutant set.

3 Applying mutation analysis to Lustre

3.1 Brief presentation of Lustre *language*

Lustre [18] 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 behavior of the program.

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

Fig. 1. A simple Lustre program

Lustre offers usual arithmetic, boolean and conditional operators and two specific operators: **pre**, the "previous" operator, and -> the "followed-by" operator [5]. Fig. 1 gives a Lustre program implementing a simple stopwatch (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. `Current` and `When` are two other temporal specific operators of Lustre used for sampling signals.

[5] 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 $(e_0, f_1, ..., f_n...)$.

3.2 Fault model

Mutation operators proposed for Fortran, Java or C are not completly re-usable for Lustre, since it is a data-flow language. A first step of our work was to select a subset mutation operators that was compatible with Lustre language specificities. As we mentioned Sect. 2.3, mutation operators are classified into three groups: statement analysis (SAL), predicate analysis (PDA), coincidental correctness (CCA).

All operators from the class CCA were selected, except those dealing with array reference. In dataflow languages, and especially in Lustre, arrays are much more than a data structure. They are a powerfull way of constructing programs and define regular networks [6]. Simple syntactic change in the array reference usually produce an incorrect Lustre program. This is due the fact that Lustre is a strongly type language. For instance, let us consider the example given Fig. 2. In this node, the input and the output are two arrays of integers. The equation states that for (`i=0 to 5, b[i]=a[i]+1`). For this equation, replacing an array reference by a constant (in (`b` or `a`)) or modifing the size of one tabular will lead to an error.

```
node example(a : int^6)   -- array of integers
returns (b : int^6 );
let
  b[0..5] = a[0..5] + 1^6 ;
tel;
```

Fig. 2. An other Lustre program (with arrays)

All mutation operators of the PDA type except DSA were selected. Indeed Data Statement Alteration (DSA) can not directly be applied for Lustre, since there is no data statement. In node given Fig 1, Arithmetic Operator Replacement (AOR) would replace `+` by `-`. The specific Lustre operator `pre` is considered for Unary Operator Insertion (UOI).

No SAL operators were selected. A Lustre node is a set of equations which can be written in any order. Since there should be exactly one equation for each output and local variable, it is not possible to delete a statement (SDL mutation operator). Moreover, Lustre has no DO, GOTO and RETURN statement (or similar ones). So related mutation operators (DER, DSA, GLR and RSR) have no sense here.

Lustre language has four specific temporal operators (`pre`, `followed-by`, `current` and `when`). As said previously, the operator `pre` is considered for UOI mutation operator. For `followed-by`, `current` and `when` opertaors, we are currently searching adequate mutation operator. However, thanks to "classical" mutation operators (changes in variables, constants and non-temporal operators), it is possible to alterate the behavior of sequential programs. For instance, for node `chrono`, it is possible to replace the variable `raz` or the constants 0 or 1, which will modify the behavior of `chrono`.

Program	# lexemes	# node	# operands	# operators	# mutants
Conditionner	61	1	30	20	32
UMS	66	1	32	18	16
Sorting 1	184	3	127	15	5
Sorting 2	369	1	221	104	100
Sorting 3	489	1	289	140	140
Sorting 4	932	1	562	343	924
Monitoring	268	6	86	111	3
Supplying	555	8	296	126	68
Lift	905	5	421	259	220

Table 2
Quantitative elements about mutated programs

3.3 Mutation tool for Lustre

Alien-V [6] is a tool we built for mutating Lustre nodes. This tool was produced within a collaboration between LSR (team VASCO) and LCIS (team VALSYS). The multi-language mutant generator for VHDL and C developped by LCIS (Alien) [26] was extended to Lustre.

The mutation analysis is mainly a lexical process, since it is a multi-language tool. A mutation operator table is an input of the tool. It is possible to adapt this table to define specific mutation operators. For the moment, only AOR, LOR, and ROR are defined by default (Arithmetic/Logical/Relational Operator Replacement). CSR (Constant for Scalar variable Replacement) and SCR (Scalar for Constant Replacement) have to be manually parameterized for each program.

Some work is currently undertaken to improve Alien-V. We have used our mutation tool on several examples:

- an *Air-conditionner controller system* (Conditionner) specified in [5], and described in Lustre in [22],
- a *subway U-turn section* (UMS) [19],
- a *simplified monitoring of accelerometer sensors* (Monitoring) [7],
- a *water supplying system* (Supplying) [14]
- four examples of *8-integer sorting applications* (Sorting)
- a *lift* system [32].

We have selected those 9 programs since they present different properties. Sorting programs are combinatorial examples. Conditionner and UMS are simple sequential boolean one-node programs. Monitoring is a simple sequential boolean program calling library nodes. Supplying is a more complex sequential boolean program, and Lift is a more complex boolean program which uses arrays. Most of these programs were provided with environment descriptions and safety properties.

Table 2 presents some quantitative elements about those examples and the results of the mutation analysis. As it can be noticed, mutation analysis for Lustre programs produces proportionnally less mutants than mutation analysis for imper-

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

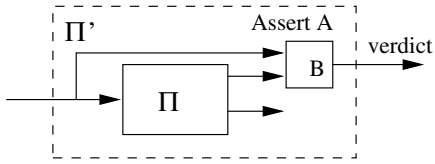

Fig. 3. Verification program structure

ative programming languages (see Sect. 2.4).

4 Detecting equivalent mutants

4.1 Lesar: a model-checker for Lustre

Lesar [19,20] is a model-checker for Lustre. It can be used to prove the correctness of a Lustre program with respect to some safety properties or to compare two programs.

As input, Lesar need a *verification program* [19]. A *verification program* is a specific Lustre program Π' built out of three elements (fig. 3):

- a program Π to be verified,
- a property P expressed by a boolean expression B which should be invariably *true*,
- some assumptions on the environment (environment constraints); those assumptions are boolean expressions (A) which can be assumed to be always true.

The verification is performed on a finite state abstraction Π'' of the program Π'. The verification principle is the following: proving that Π'' holds is equivalent to enumerating its finite set of states, checking that in each state (belonging to a path starting from initial state and on which the assertions are always true) and for each input vector, Π'' output evaluates to true. Lesar was originally a boolean tool. A special algorithm has been added into Lesar in order to treat contraints on numerical values.

4.2 Applying Lesar *for detecting equivalent mutant*

As previously said, mutation analysis can generate mutants equivalent to the initial program. It is the case when both mutant and original program have always the same observable behavior. Eliminating equivalent mutants is required, otherwise a maximal mutation score can not be reached.

Lesar can be used to detect equivalent mutants produced for a Lustre program. To do that, one has to construct a verification program that is the comparison of the mutant and original programs, as it is done Fig. 4. When some environment description is 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 (unconditionnal mutant-equivalency).

We have applied Lesar to detect equivalent mutants for our 9 examples (see

```
node VerifPgm(in_0,..,in_n )   --inputs
returns (ok: bool);
var outo_0,..,outo_m           -- output for the original node
    outm_0,..,outm_m           -- output for the mutant
let
  (outo_0,..,outo_m) = original_node(in_0,..,in_n );
  (outm_0,..,outm_m) = mutant(in_0,..,in_n );

     -- property to be checked
   ok = (outo_0=outm_0) and ... and (outo_m=outm_m);
tel;
```

Fig. 4. A verification program to detect mutant equivalence

Program	# mutants	# eq. mutants
Conditionner	32	2
UMS	16	0
Sorting 1	5	-
Sorting 2	100	-
Sorting 3	140	-
Sorting 4	924	-
Supplying	68	15
Lift	220	0

Table 3
Mutant and equivalency

sect. §3.3). Although we generally have the environment descriptions for these examples, we wanted to demonstrate unconditionnal mutant-equivalency.

LESAR was very quick to detect equivalent mutants for Conditionner, UMS, Monitoring and Supplying. However, LESAR does not provide any result for the four 8-integer Sorting examples. We initially thought it was due to integer values. However, it was not possible to detect equivalent mutants with 8-boolean Sorting programs (same programs, with boolean inputs and outputs and same mutants produced). Lift program is composed of 5 nodes (one main node calling once time each of the four other nodes). Mutation analysis was done on each node. It was possible to detect equivalent mutant considering each node separatly.

5 Evaluating test data

As said in the introduction, mutation analysis was proposed to evaluate the quality/adequacy of a test set. So, to evaluate Alien-V, we wanted to evaluate test data produced by our testing tool LUTESS. In the first part of this section, we briefly present the testing tool, and then we describe a first experiment using Alien-V to determine test data mutation-adequacy.

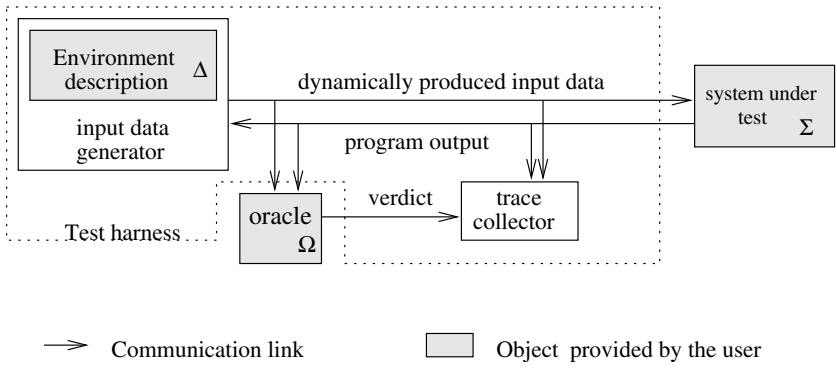

Fig. 5. Lutess

5.1 Lutess *testing tool: an overview*

Lutess [13,32] is a testing tool which we developed to validate reactive synchronous software. It requires three elements: an environment description written in Lustre (Δ), a program under test (Σ) and an oracle (Ω) providing the program requirements (fig. 5). Lutess builds a random generator from the environment description and constructs automatically a test harness which links the generator, the program under test and the oracle. The program under test and the oracle are both synchronous executable programs, with boolean inputs and outputs. They can be supplied as Lustre programs.

The test is operated on a single action-reaction cycle, driven by the generator. The generator randomly selects an input vector for the program under test and sends it to this latter. The program under test reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated. The oracle observes the program inputs and outputs, and determines whether the software specification is violated. The testing process is stopped when the user-defined length of the test sequence is reached.

Basically, the Lutess generator selection algorithm chooses a valid [7] input vector in an equally probable way. In each environment state, any valid input vector has the same probability to be selected. Lutess offers also various facilities to guide the generation (with property or statistical descriptions) and replay some test sequences (re-do) [13,22].

5.2 *Test generation evaluation*

To evaluate the mutation-adequacy of a test set, our general process is the following. On one hand, we produce one or several testing sequences with a tool (here Lutess). On the other hand, we produce "oracle" programs for Lutess (one for each non-equivalent mutant). Such an oracle program takes as inputs the original program inputs and outputs; it returns one boolean, which value is false each step the considered mutant produces different outputs than the original program (see

[7] An input is valid if and only if it is complying with the environment description.

Mu-	set A			set B			set C			set D		
tant	min	avg	max	min	avg	max	min	avg	max	min	avg	max
15	663	719.97	776	667	727.90	834	2	17.73	63	0	8.87	121
31	678	735.07	813	3	10.83	25	93	399.70	936	6	256.97	727
32	1416	1477.33	1580	9	19.83	34	101	407.13	937	7	256.87	727
47	3568	3790.83	3958	2074	5122.97	7101	9618	9812.03	9930	9973	9990.43	9999
53	0	3.03	11	0	4.17	13	11	333.47	899	0	254.83	724

min/max are the minimum/maximum numbers of differences observed for one sequence among the 30 of a set.

Avg is the sum of differences observed for the 30 sequences divided by 30.

Table 4
Some results for supplying example

Fig. 6). We then use the re-do function of Lutess, which feeds an oracle program with inputs/outputs previously obtained with the original program.

To carry out a first evaluation of Lutess, we focus on the Supplying example. Four sets of data were produced with environment constraints: without any guiding (A), with property guiding (B), with statistical guiding (C and D). Each time, 30 sequences of 10000 steps were generated. For each set, all mutants were killed. But not all test sequences killed every mutant. For some mutants, there were some test-sequences for which we could not observed any differences between the original program and the mutants.

For each set of data, we count how many steps a difference between the mutant and the original program could be observed. It was then possible to "compare" the sets of data (see Table 4). We call "mutant difficult to kill with a method" mutant for which differences could be observed in less than 1% of steps in average on the 30 sequences.

Mutants that are difficult to kill are usually those concerning "initial state" or "limit situation". Mutants difficult to kill are not the same in the different test data sets. This suggests that the generation methods of Lutess produce different types of data.

```
node OracleMutant(in_0,..,in_nn, outo_0,..,outo_m)
                --inputs and outputs for the original node
returns (ok: bool);
var
   outm_0,..,outm_m           -- output for the mutant
let
   (outm_0,..,outm_m) = mutant(in_0,..,in_n );

   -- property to be checked
   ok = (outo_0=outm_0) and ... and (outo_m=outm_m);
tel;
```

Fig. 6. A oracle program for Lutess to kill mutants

6 Conclusion and perspectives

Summary of the work

Mutation analysis aims at the evaluation of the adequacy of a test set with respect to a fault model. To do that, one has to produce all mutants corresponding to this 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 behavior of all faulty programs from the original program. Mutation analysis has been introduced by DeMillo in 1978 for Fortran 77 programs [11]. It has been widespread for different types of languages. [10,28,15,23,34,36]; and Briand *et al.* have demonstrated that mutants can provide a good indication of the fault detection ability of a test suite [3].

In this paper, we have adapted mutation analysis to Lustre programs. Lustre is a synchronous data-flow language. The data-flow nature of this language requires to adapt mutation operators that were originally proposed. Alien-V, the tool we built for Lustre program is presented. It has been experimented on 9 programs, from simple to more complex ones.

The main difficulty with mutation analysis is the detection of equivalent mutants. Equivalent mutants have exactly the same observable behavior than the original program. Eliminating equivalent mutant is required to obtain a maximum *mutation-score*. Since Lustre is based on a solid mathematical foundation, it is possible to construct proofs about the programs. For instance, Lesar is a model-checker for Lustre. To detect equivalent mutants, we have used Lesar. Unfortunately, it was not possible to kill mutants for some programs dealing with integers.

Finally, we have used mutant produced by Alien-V to evaluate *mutation-adequacy* of test data produced by Lutess our testing tool with different guides. First results show that the fact that a mutant is "difficult" to kill depends on the guides used for the generation. This means that Lutess generation method is really influenced by the guides.

Perspectives for Alien-V

For the moment, the fault model we used is mainly a selection of mutation-operator previously defined for imperative languages. We are currently defining adequate mutation operators for temporal Lustre operators (`followed-by`, `current` and `when`).

Mutation analysis was initially defined to evaluate adequacy of test data produced during unit-testing. Recently, some works have been proposed to extend mutation analysis to evaluate data set produced during integration testing [9]. A fault model specific to integration test evaluation was proposed. In Lustre, programs are oftenly structured with several nodes. Integration testing is therefore required. The next step for Alien-V is to define/adapt mutation operators to integration testing as it is done in [9].

Using mutation for test data adequacy evaluation

A first experimentation has been done to evaluate the mutation-adequacy of test data produced by LUTESS. We would like to use mutation as a help to decide the end of testing. Moreover, we want to evaluate the influence (1) of the environment constraints and (2) of LUTESS guiding methods on the mutation-adequacy of test data.

References

[1] Agrawal, H., 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] Agree, A. T., T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward, *Mutation analysis*, Technical report git-ics-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA (1979).

[3] Andrews, J. H., L. C. Briand and Y. Labiche, *Is mutation an appropriate tool for testing experiments?*, in: *27th International Conference on Software Engineering (ICSE'05)* (2005), pp. 402–411.

[4] Arditi, L., A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, L. Leblanc and R. de Simone, *Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP*, in: *ERCIM workshop on Formal Methods for Industrial Critical Systems*, Trento, Italy, 1999.

[5] Atlee, J. M. and J. D. Gannon, *State-based model checking of event-driven system requirements.*, IEEE Trans. Software Eng. **19** (1993), pp. 24–40.

[6] Benveniste, A., 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** (2003), pp. 64–83.

[7] Bourret, P., A. Fernandez and C. Seguin, *Statistical criteria to rationalize the choice of run-time observation points in embedded software*, in: *1st International Workshop on Testability Assessment (IWoTA'04)* (2004), pp. 41–49.

[8] Budd, T. A., *Mutation analysis of program test data*, Phd thesis, Yale University, New Haven, CT, USA (1980).

[9] Delamaro, M. E., J. C. Maldonado and A. P. Mathur, *Interface mutation: An approach for integration testing.*, IEEE TSE **27** (2001), pp. 228–247.

[10] DeMillo, R., 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*, Banff, Canada, 1988, pp. 142–151.

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

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

[13] du Bousquet, L., F. Ouabdesselam, J.-L. Richier and N. Zuanon, *Lutess: a specification-driven testing environment for synchronous software*, in: *21st International Conference on Software Engineering* (1999), pp. 267–276.

[14] Duc, B. M., "Conception et modélisation objet des systèmes temps réel," Eyrolles, 1998, 341 pp.

[15] Fabbri, S. C. P. F., 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.

[16] Fabbri, S. C. P. F., J. C. Maldonado, P. C. Masiero and M. E. Delamaro, *Proteum/fsm: A tool to support finite state machine validation based on mutation testing.*, in: *19th International Conference of the Chilean Computer Science Society (SCCC '99)* (1999), pp. 96–104.

[17] Fabbri, S. C. P. F., J. C. Maldonado, P. C. Masiero, M. E. Delamaro and W. E. Wong, *Mutation testing applied to validate specifications based on petri nets.*, in: *Formal Description Techniques VIII, Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques (FORTE)*, IFIP Conference Proceedings **43** (1995), pp. 329–337.

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

[19] Halbwachs, N., 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 (1992), pp. 785–793.

[20] Halbwachs, N., D. Pilaud, F. Ouabdesselam and A.-C. Glory, *Specifying, Programming and Verifying Real-Time Systems, using a synchronous declarative language*, in: *Workshop on automatic verification methods for finite state systems, LNCS 407* (1989).

[21] Jagadeesan, L., A. Porter, C. Puchol, J. Ramming and L. Votta, *Specification-based Testing of Reactive Software: Tools and Experiments*, in: *19th International Conference on Software Engineering*, 1997.

[22] Lakehal, A., F. Ouabdesselam, I. Parissis and J. Vassy, *Models for synchronous software testing*, in: *First International Workshop on Model, Design and Validation, 2004* (2004), pp. 41–50.

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

[24] Marre, B. and A. Arnould, *Test Sequences Generation from Lustre Descriptions: GATeL*, in: *15th IEEE International Conference on Automated Software Engineering (ASE)* (2000).

[25] Mazuet, C., *Stratégies de Test pour des Programmes Synchrones - Application au Langage Lustre*, Technical report, Institut National Polytechnique de Toulouse, Toulouse, France (1994).

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

[27] Offutt, A. J. and W. M. Craft, *Using compiler optimization techniques to detect equivalent mutants*, Softw. Test., Verif. Reliab. **4** (1994), pp. 131–154.

[28] Offutt, A. J. and H. J. H., *A semantic model of program faults*, in: S. J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, Californy, USA, 1996, pp. 195–200.

[29] Offutt, A. J. and J. Pan, *Automatically detecting equivalent mutants and infeasible paths*, Softw. Test., Verif. Reliab. **7** (1997), pp. 165–192.

[30] Offutt, A. J., G. Rothermel and C. Zapf, *An experimental evaluation of selective mutation*, in: *ICSE*, 1993, pp. 100–107.

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

[32] Parissis, I. and F. Ouabdesselam, *Specification-based Testing of Synchronous Software*, in: *4th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, San Francisco, USA, 1996.

[33] Raymond, P., D. Weber, X. Nicollin and N. Halbwachs, *Automatic testing of reactive systems*, in: *19th IEEE Real-Time Systems Symposium (RTSS'98)* (1998).

[34] Scholivé, M. and C. Robach, "Simulation-based fault injection and testing using the mutation technique," Kluwer Academic Publishers, 2003 .

[35] Seljimi, B. and I. Parissis, *Using CLP to Automatically Generate Test Sequences for Synchronous Programs with Numeric Inputs and Outputs*, in: *17th Int. Symposium on Software Reliability Engineering (ISSRE'06)* (2006), pp. 105–116.

[36] Wong, W. E., "Mutation Testing for the New Century," Kluwer Academic Publishers, 2001.