Mutation Analysis for Reactive System Environment Properties

Huy Vu Do, Chantal Robach
LCIS-ESISAR
BP 54, 50 rue B. de Laffemas
26902 Valence Cedex 09, France
{Huy-Vu.Do, Chantal.Robach}@esisar.inpg.fr

Michel Delaunay
LSR-IMAG
BP 72, 681 rue de la Passerelle
38402 St Martin d'Hères, France
Michel.Delaunay@imag.fr

Abstract

Reactive systems used in safety-critical domains demand high level of confidence. The development of these systems, which are submitted to several normative recommendations, is complex and expensive. Reactive systems can be developed by using the data-flow approach: many languages support this approach such as MATLAB/SIMULINK, LUSTRE/SCADE. *This paper concentrates on the* LUSTRE/SCADE *language, especially the description of reactive system environment properties in this language. The description of environment properties, which is important for the validation (the proof and the test) of reactive systems, is not easy. Hence, we would like to use the mutation technique to consolidate this difficult task: we use the* LESAR *model-checking tool to detect equivalent mutants and some test case generators such as* GATEL *or* LUTESS *tools to kill non-equivalent mutants.*

1. Introduction

Nowadays, reactive real-time systems are widely used in many safety-critical domains: automotive, aerospace, nuclear... These systems maintain a permanent interaction with a physical environment. They require a very high level of confidence, since a failure in any one of them could be a disaster.

As safety-critical systems, they are constructed using several normative recommendations such as DO-178B for avionics software [6]. According to [23], the software development process (Figure 1) is composed of:

- The high-level software requirements (HLR) produced directly from the system requirements and system architecture. They include specifications of functional and operational requirements, timing and memory constraints. They are written in natural language.
- The low-level software requirements (LLR) produced through the software design process.

In the development of these deterministic reactive real-time systems, the designer can use the data-flow approach to design systems. In this approach, a system can be viewed as a diagram of operators or subsystems. A subsystem is also a composition of operators. Many development environments support this approach such as SIMULINK [14], SCADE [24].

The SIMULINK language, which consists of mathematical formalisms (differential equations, finite difference equations, boolean equations, etc), can be used to program the high-level requirements of reactive software.

The language commonly used to program the low-level requirements for flight control systems and other avionics systems is the functional synchronous language LUSTRE [9], implemented in the SCADE environment.

In the development of these systems, the validation plays

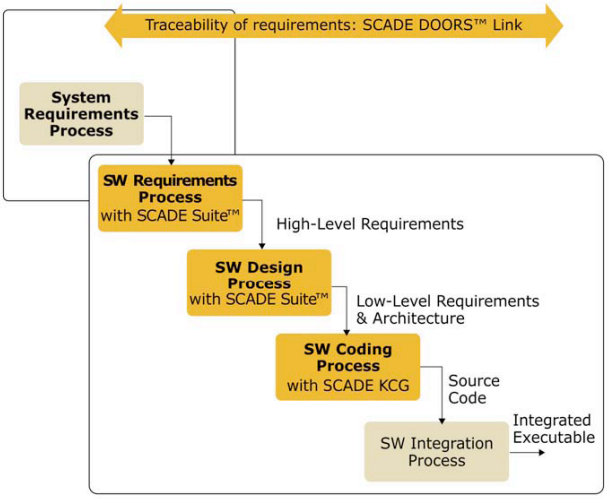

Figure 1. Software development processes with SCADE Suite

an important role since it ensures the confidence in these systems. The validation is divided into two activities: the proof of a part of the system, and the testing phase that reveals faults in the system. According to [10, 24], these activities are complementary methods.

The proof of a part of the software can be made with the SCADE design verifier [24]. GATEL [13] and LUTESS [7, 8] are tools for the interactive generation of test sequences from LUSTRE descriptions. Both tools use formal specifications of the software environment to generate test sequences. These specifications only use input variables of the software. Opposed to environment properties, safety constraints use the output variables of the software. Safety contraints are only used in the verification activity of the software. Both environment properties and safety constraints can be expressed in a temporal logic. Environment properties might be written as soon as the high level requirements (HLR) are developed. Early in the development cycle of the software, they can be used to prepare functional test sequences.

The formal specifications are expressed in a temporal logic formula. The environment properties must be true for any input sequence of the software. In our context, we use the LUSTRE langage in order to express these environment properties.

Although mutation technique is based on the competent analyst hypothesis [2], our experience tends to show that it is difficult to conceive correct and complete environment properties, especially for junior engineers. So, we think that a mutation technique could help designers to validate their temporal specifications.

In this paper, we present the LUSTRE language, the SCADE environment and the LUTESS tool (Sect. 2). We then explain our usage of mutation technique (Sect. 3). In section 4, we present our mutation technique process of LUSTRE descriptions. We illustrate these ideas with the air-conditioner example [3] and we present some results in Sect. 5.

2. LUSTRE language, SCADE environment, LUTESS and LESAR tools

2.1. LUSTRE language

LUSTRE [9] is a synchronous data flow specification language. It is a declarative 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. Inputs, outputs, variables and constants are represented by sequences of data (i.e. data flows) of the form $(e_0, e_1, e_2, ...)$.

A node is defined by a set of equations and assertions. Each node describes the relations between its input parameters, output parameters, global variables, local variables and constants. Any local variable must be defined by one and only one equation. Equation $X = E$ defines the variable X as being identical to the expression E and having the same sequence of values and the same clock: it means that, at every moment t related to their clock, $x_t = e_t$. It is the principle of substitution: X can be replaced by E anywhere in the program, and reciprocally. The equations can be written in any order without changing the significance of the program. An assertion `assert BOOL_EXPR` means that the boolean expression `BOOL_EXPR` is assumed to be always true during the execution of the program. Assertions are used to describe the environment properties of systems.

Expressions are made of variable identifiers, constants and operators. There are three categories of operators:

- primitive operators, such as: relational operators (`=`, `<>`, `<`, `<=`, `>`, `>=`), logical operators (`and`, `or`, `not`, `xor`, #), arithmetic operators (`+`, `-`, `*`, `/`, `div`, `mod`, `**`, `int`, `real`), conditional operators (`if`, `then`, `else`, `case`), temporal operators (`pre`, `fby`, `->`, `when`, `current`), assertion operator (`assert`).
- more complex operators, called *nodes*, created by the users;
- operators imported from another language (C,...).

The significance of two important temporal operators `pre` (*previous*) and `->` (*followed-by*) is detailed below.

- if E is an expression denoting the sequence (e_0, e_1, e_2,...), then $pre(E)$ denotes the sequence (nil, e_0, e_1,...) where nil is an undefined value;
- if E and F are two expressions of the same type respectively denoting the sequences (e_0, e_1, e_2,...), and (f_0, f_1, f_2,...), then $E -> F$ is an expression denoting the sequence (e_0, f_1, f_2,...).

2.2. LUSTRE example

In this paper, we use a LUSTRE example presented in [3]: this example allows the monitoring of an air-conditioner which diffuses hot or cold air. The system is monitored using a power switch and an adjustable thermostat. According

to the ambient temperature given by a temperature sensor, the system must diffuse hot or cold air until the `TempOK` thermostat state is reached.

The system is represented by a machine with four states:

- `STOP` corresponds to the state "out of order" of the air-conditioner;
- `HOT` means that the air-conditioner is running and diffusing hot air;
- `COLD` means that the air-conditioner is running and diffusing cold air;
- `IDLE` means that the air-conditioner is running but does not diffuse air (this situation occurs when the ambient temperature is identical to that indicated by the thermostat).

The changes of states are operated under certain conditions:

- `Go` is true when the power switch is on, otherwise it is false;
- `TooCold`, `TempOK` and `TooHot` are true when the ambient temperature is respectively lower, equal or higher than the temperature indicated by the thermostat of the air-conditioner. These three conditions constitute a simplified view of information coming from the temperature sensor and the thermostat.

There are two assumptions on the temperature variation:

1. one and only one of the conditions `TooCold`, `TempOK` and `TooHot` can be true at the same time;
2. between the passing to true of the `TooCold` condition and that of the `TooHot` condition, the `TempOK` condition will be true at least once (and conversely).

The block diagram of this example (Figure 2) has been created with the SCADE editor.

The LUSTRE code of this example is presented in Figure 3.

2.3. SCADE environment

The SCADE [24] environment was developed by the Telelogic company (now Esterel Technologies). This environment, derived from the LUSTRE language, supports the synchronous data flow approach for the development of reactive systems. This environment supplies a set of tools: graphical editor, simulator, model checker and code generators that automatically translate graphical specifications into C or Ada code. Some C code generators are certified DO-178B. Specific symbol libraries such as "libdigital", "libmath" and "libverification" are also included in the SCADE environment. Users can create their own libraries by using the graphical editor or by manually coding complex operators.

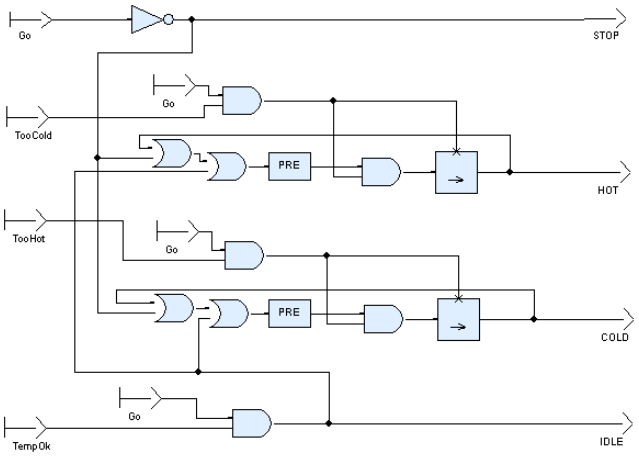

Figure 2. Air-conditioner block diagram

```
1  node Clim2 (Go, TooCold, TempOK, TooHot : bool)
2  returns    (STOP, IDLE, HOT, COLD : bool);
3
4  let
5      STOP = not Go;
6      IDLE = Go and TempOK;
7      -- HOT = Go and TooCold;
8      HOT =
9          (Go and TooCold) ->
10             (pre (STOP or IDLE or HOT) and
11              Go and TooCold);
12     -- COLD = Go and TooHot;
13     COLD =
14         (Go and TooHot) ->
15             (pre (STOP or IDLE or COLD) and
16              Go and TooHot);
17 tel;
```

Figure 3. Air-conditioner LUSTRE code

From the "libverification" library, standard operators such as `Implies`, `AlwaysAfterFirstCond`, `HasNeverBeenTrue`, `After1stTick`, `AtLeastNTicks`, `ImpliesWithin1Tick`, etc. are available for the system specifications.

We added the nodes `after`, `always_from_to`, `always_since`, `atlast`, `edge`, `jafter`, `once`, `once_from_to`, `once_since`, `two_states`, `xedge` to a library in order to extend the SCADE environment. See [19, 21] for informal specifications of these operators.

2.4. LUTESS: Architectural overview

LUTESS is a testing environment for synchronous reactive software. It produces automatically and dynamically test data with respect to some environment constraints of the program under test.

The architectural overview of LUTESS is presented in Figure 4.

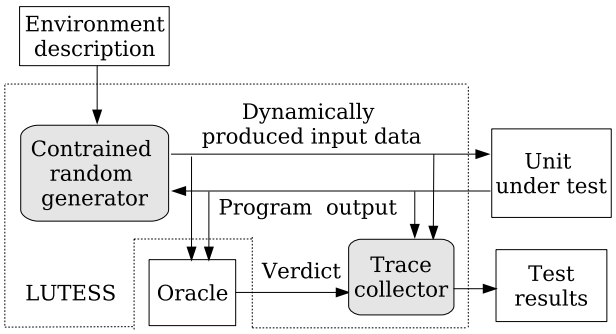

Figure 4. LUTESS **architectural overview**

The operation of LUTESS requires three elements: a random generator, a unit under test (UUT) and an oracle (as shown in Figure 4). LUTESS constructs automatically the test harness which links these three components, coordinates their execution and records the sequences of input-output relations and the associated oracle verdicts. These three components are just connected together and not linked into a single executable code. More details can be found in [7, 19].

The constrained random generator is automatically built by LUTESS from specifications written in LUSTRE and from operational profiles [15] stated partially in LUSTRE.

2.5. LESAR tool

LESAR [18, 21] is a model-checking tool for the Lustre language. It takes a compound verification program in Lustre (as depicted in Figure 5). It applies standard symbolic model-checking techniques, with abstraction of numerical variables, to perform the verification. If the `ok` output is false while the `realistic` output is always true, LESAR [18, 21] will find a counter-example (a sequence of states which leads to a bug).

3. Mutation Analysis

Mutation technique [1, 11, 16] induces the injection of faults into the software by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty descriptions with the goal of distinguishing the faulty descriptions from the original description [17]. Faulty descriptions are *mutants* of the original, and a mutant is *killed* when the output of the mutant is distinguished from that of the orginal description.

Some mutants may be functionally *equivalent* to the original description. Equivalent mutants always produce the same output as the original description.

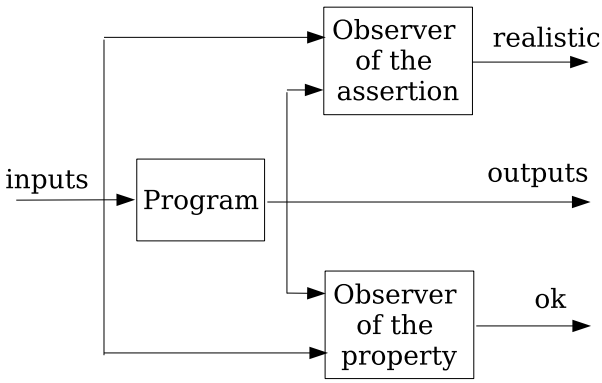

Figure 5. LESAR **compound verification program**

In the classic mutation, all the test cases submited to the original program produce correct results. If for a test case, the mutant produces a result different from the awaited result, then this mutant is marked as *dead*. The mutation score is the ratio between the number of dead mutants and the number of non-equivalent mutants.

Specification mutation operators for SMV [5] have been studied in [4]. A mutation operator has a corresponding fault class. Fault classes have been studied by Kuhn [12]. Many mutation operators can be transposed to the environment properties written in LUSTRE. Some new operators have been added in order to treat verification primitives such as `once_from_to`, `always_from_to`, `once_since`, etc.

4. Mutation analysis process

At the LCIS laboratory, we have developed a multi-language mutant generator for the VHDL, C languages [1, 16, 22] using the principles of [11]. Later, we extended it for the LUSTRE language. For each mutation analysis, we used a mutation operator table as an entry to our mutation generator and we have defined several operator tables for different mutations analysis. For the LUSTRE language, we need new mutation operators adapted to the temporal logic.

In restricting mutation technique on to environment properties, we can limit the number of generated mutants and large amounts of computations aren't necessary. Using a selective mutation — a "do fewer" approach — with the logical operator replacement (LOR), relational operator replacement (ROR) and the unary operator insertion (UOI, in fact the `not` or `pre` operator) and special temporal logic

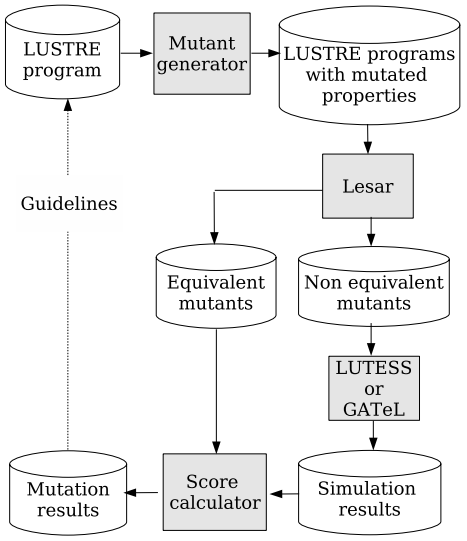

Figure 6. Schematic data flow diagram of the tool

operators, we limit the computional expense of generating and running vast numbers of mutants. We can also use a randomized selection of mutants [25].

The functions of the SCADE editor can be extended by using APIs in the TCL language. So, we can extract the environment properties from the SCADE model of the software and construct four elements:

1. a *environment description node* to calculate the environment properties replacing the original assertion such as the `PropEnvClim` node (Figure **??**) used in our case study,

2. an *oracle node* to compare the mutant result and the original result obtained by a call to the previous node such as the `Oracle_PropEnvClim` node for the LUTESS oracle (Figure **??**); note that we choose to use the original description of environment properties in the oracle,

3. a *unit under test* for the LUTESS tool which is a generated mutant. These mutants are generated by using our sample mutant generator as soon as mutant operators have been defined,

4. a *constraints description node* to generate test sequences such as the `PropEnvClim` node (Figure **??**).

With these last three elements, the LUTESS tool generates input test sequences.

It will be noted that we may potentially use our method on any LUSTRE node viewed as a *toplevel* node, for example, in a system testing activity.

Note that we use the SCADE SUITE (version 5.0.1) under Windows XP and the other tools run under UNIX/LINUX systems.

5. Case Study

We applied this method to the air-conditioner problem described in [3]. In the description of the LUSTRE/LUTESS code, we omit the description of a standard verification library used for the LUSTRE language.

The *original description* of the environment properties, is presented as:

```
1  node PropEnvClim (
2      Go,             -- position of the switch
3      TooCold,        -- ambient temp. < thermostat
4      TempOK,         -- ambient temp. = thermostat
5      TooHot : bool  -- ambient temp. > thermostat
6  )
7  returns (
8      ok : bool -- ''true'' if the environment
9                -- properties are respected
10 );
11
12 var
13     assert_1 : bool;
14     assert_2 : bool;
15
16 let
17     ok = assert_1 and assert_2;
18
19     assert_1 =
20         ((TooHot or TempOK or TooCold) and
21          # (TooHot, TempOK, TooCold));
22
23     assert_2 =
24         (once_from_to (TempOK, TooCold, TooHot) and
25          once_from_to (TempOK, TooHot, TooCold));
26 tel;
```

In this specification, according to the synchronism hypothesis of the LUSTRE descriptions, we translated the two assumptions on the temperature variation (see 2.2) into the `asset_1` and `asset_2` equations.

The LUTESS oracle, using the *original description* of the environment properties, is:

```
1  node Oracle_PropEnvClim (
2    -- ''mutant result''
3      ok : bool;      -- ''true'' if the environment
4                      -- properties are respected
5                      -- in the mutant.
6    -- Input variables
7      Go,         -- position of the switch
8      TooCold,        -- ambient temp. < thermostat
9      TempOK,         -- ambient temp. = thermostat
10     TooHot : bool -- ambient temp. > thermostat
11 )
12 returns (
13     verdict : bool -- ''true'' if the environment
14                    -- properties are respected
15 );
16
17 var
18     pp_ok    : bool;
19
20 let
21     verdict = (ok = pp_ok);
22
23     pp_ok   = PropEnvClim (
```

```
24          Go, TooCold, TempOK, TooHot
25      );
26  tel;
```

In our extended SCADE environment, `once_from_to` is a standard verification predicate: `once_from_to (A, B, C)` is false if `A` has been false once time between the `B` last occurrence and the first occurrence of `C` following the `B` occurrence.

The LUTESS test node of this example is:

```
1   testnode test_PropEnvClim (
2       ok : bool       -- ''true'' if the environment
3                       -- properties are respected
4   )
5   returns (
6       Go,             -- position of the switch
7       TempLow,        -- ambient temp. < thermostat
8       TempOK,         -- ambient temp. = thermostat
9       TooHot : bool   -- ambient temp. > thermostat
10  );
11
12  let
13      -- The thermostat is in at least one state.
14      environment ((TempLow or TempOK or TooHot));
15  tel;
```

It expresses that, if the thermostat is functionally correct, at least one of the three temperature levels is true. Note that the value of the input variable `Go` is independent from the thermostat state.

This LUTESS `test_PropEnvClim` test node can also be viewed as a *special* observer node [10]. You may add some conditional probalities in the test node description.

With the `lesar_PropEnvClim`:

```
1   node lesar_PropEnvClim (
2       Go,             -- position of the switch
3       TooCold,        -- ambient temp. < thermostat
4       TempOK,         -- ambient temp. = thermostat
5       TooHot : bool   -- ambient temp. > thermostat
6   )
7   returns (
8       verdict : bool -- truth if the environment
9                       -- properties are identical.
10  );
11
12  var
13      pp_ok : bool;
14      ok : bool;      -- true if the environment property
15                      -- is respected by the mutant.
16
17  let
18      -- compare the two results.
19      verdict = (ok = pp_ok);
20
21      pp_ok   = PropEnvClim (
22          Go, TooCold, TempOK, TooHot
23      );
24
25      ok = __NOM_MUTANT__ (
26          Go, TooCold, TempOK, TooHot
27      );
28  tel;
```

we can determine, using the LESAR tool [20], if the mutant and the original properties are equivalent: they are equivalent if the `verdict` variable is always `true`.

Some other strategies can be used to describe the environment clause of the test node including the original environment property or a simplified one, etc.

Experiences have been made using the following mutation operators:

```
LOR = (and, or, xor)
VR = (once_from_to,
      always_from_to,
      not_between_and
)
VR = (TooHot, TempOK, TooCold)
```

Here is a mutant obtained by replacing the `TempOK` variable with the `TooCold` variable in the `assert_2` equation:

```
1   node PropEnvClim (
2   Go , -- position de l'interrupteur
3   TooCold , -- ambient temp. < thermostat
4   TempOK , -- ambient temp. = thermostat
5   TooHot : bool -- ambient temp. > thermostat
6   )
7   returns (
8   ok : bool -- ''true'' if the environment
9    -- properties are respected
10  ) ;
11  var
12  assert_1 : bool ;
13  assert_2 : bool ;
14  let
15  ok = assert_1 and assert_2 ;
16  assert_1 = ( ( TooHot or TempOK or TooCold ) and
17  # ( TooHot , TempOK , TooCold ) ) ;
18  assert_2 = ( once_from_to ( TempOK , TooCold , TooHot )
19   and once_from_to ( TooHot  , TooHot , TooCold ) ) ;
20  tel
```

With these operators, we generated 35 mutants. There are 2 equivalent mutants due to the replacement of the `or` operator by the `xor` operator. All non-equivalent mutants were killed: the mutation score obtained is equal to 1 because our case study is quite simple.

6. Conclusion and Future Work

This is a short presentation of our work on a very simple example. It must be validated on larger formal specifications from real industrial case studies. Nevertheless, we can present some limits of our approach and the needed future work.

Our mutation technique process can be used on assertion clauses in a node if the assertions only use input variables of this node: the assertions can be viewed as simple environment properties.

By using a mutation operator table, we have an implicit fault model. By extending the semantics of the mutation operator language, we may use a more precise specification fault model for the SCADE/LUSTRE language or the specific application domain.

We may need to replace the LUTESS test sequence generator with the GATEL one in order to manipulate integer data types and, in the future, precise real data types.

In the future, a new and specific mutant generator must be written in order to extend the functionalities needed to make a full mutation analysis for the LUSTRE language.

7. Acknowledgment

The authors would like to thank Odile LAURENT of AIRBUS industry (Toulouse), Christel SEGUIN of CERT-ONERA research center (Toulouse), Jean-Sébastien CRUZ of MBDA industry (Paris).

And also, we would like to thank the ESTEREL TECHNOLOGIES company for providing the SCADE SUITE within their academic program to our institutions.

References

[1] G. Al-Hayek. *Vers une Approche Unifiée pour la Validation et le Test de Circuits Intégrés Spécifiés en VHDL*. PhD thesis, Institut National Polytechnique de Grenoble, 1999.

[2] P. E. Ammann and P. E. Black. A Specification-Based Coverage Metric to Evaluate Test Sets. In *In Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE '99)*, pages 239–248. IEEE Computer Society, Nov. 1999.

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

[4] P. E. Black, V. Okun, and Y. Yesha. Mutation Operators for Specifications. In *Automated Software Engineering*, pages 81–88, 2000.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*, 2002.

[6] DO-178B/ED-12B. *Software Considerations in Airborne Systems and Equipment*. RTCA/EUROCA, Dec. 1992.

[7] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Specification-based Testing of Synchronous Software. In *In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000), Berlin, GMD Report 91*, pages 123–139, Apr. 2000.

[8] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a Specification-driven Testing Environment for Synchronous Software. In *Proc. 21st International Conference on Software Engineering, ACM Press*, pages 267–276, May 1999.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programing language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[10] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: From formal verification to automatic testing. In *Asian Computing Science Conference*, pages 1–12, 1999.

[11] K. N. King and A. J. Offutt. A Fortran Language System for Mutation based Software Testing. *Software-Practice and Experience*, 21, 1991.

[12] D. R. Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):569–571, Oct. 1999.

[13] B. Marre. GATeL: a method and a tool for the interactive generation of test sequences.

[14] T. MathWorks. Simulink Reference (version 5), Apr. 2003.

[15] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, Mar. 1993.

[16] 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.

[17] A. J. Offutt and R. h. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000*, pages 2183–2187, 2000.

[18] G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. *Software Tools for Technology Transfer*, 5(2—3), Mar. 2004.

[19] I. Parissis. *Test de logiciels synchrones spécifiés en LUSTRE*. PhD thesis, Université Joseph Fourier, Grenoble (France), Sept. 1996.

[20] C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes Lustre : Le système Lesar*. PhD thesis, Université Joseph Fourier, June 1992.

[21] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow language LUSTRE. In *Proceedings of the conference on Software for citical systems*, pages 112–119, New Orleans, Louisiana, United States, 1991.

[22] M. Scholivé and C. Robach. *Simulation-based fault injection and testing using the mutation technique*, chapter 5. Kluwer Academic Publishers, 2003.

[23] E. Technologies. Efficient Development of Safe Avionics Software with DO-178B Objectives Using SCADE Suite, July 2005.

[24] E. Technologies. SCADE User's Manual, July 2005.

[25] W. Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, West Lafayette, Dec. 1993.