Testability Analysis Approach For Reactive Systems

Nguyen Thanh Binh[1] and Chantal Robach[2]

[1] DATIC Laboratory
Danang University of Technology
Danang, Vietnam

[2] LCIS Laboratory
Grenoble INP
Grenoble, France

Abstract

Reactive systems are often designed as two parts: computation and control. The computation part is modeled by operator diagrams, while the control part is modeled by transition-based models. In this paper, we concentrate on analyzing the testability of the control part by using upon transition based models. We first transform transition-based models into Markov chains by augmenting probability information. Then, testability measures are proposed from Markov chains as an estimate of testing effort for reaching state coverage and path coverage.

The approach is applied to a case study and the obtained measures are compared to the testing effort required by a test generation tool. The results show some interesting perspectives.

Keywords: *Testability Analysis, Reactive Systems, Transition-Based Models, Markov Chains.*

1. Introduction

Nowadays, reactive systems are used in many industrial domains: avionics, nuclear, etc… These kinds of systems are increasingly complex. These systems are often designed as two parts: computation part and control part. Some computation parts are physical laws or mathematical algorithms. In general, the computation part is modeled by operator diagrams or equations, while the control part is modeled by a kind of finite state machine (FSM) formalism. To develop these systems, many environments have been proposed as SCADE/SSM, SIMULINK/STATEFLOW. For example, the computation parts can be designed by using SIMULINK or SCADE, while the control parts can be represented by using SSM (Safe State Machine) or STATEFLOW.

In a reactive system development process, verification and validation (V&V) activities play a very important role, because these systems always require a high level of quality and confidence. However, V&V activities are often very difficult and costly, which increases final development cost. V&V are based on: either static analysis, like formal proof, symbolic execution… or dynamic testing, where the objective is to find errors in the system. In addition, in order to reduce validation cost and increase system confidence, testability analysis may be taken into account. In fact, testability measures, if possible previsional, may exhibit a difficulty in testing the system. Therefore, the designer can consider testability as a factor to modify the design in order to improve the ease of testing, *i.e.* to reduce testing cost.

Testability is of high interest by many researchers in different perspectives. Freedman [1] introduced the domain testability for software components by defining observability and controllability notions: observability is the ease of determining if specified inputs affect the outputs; controllability is the ease of producing a specified output from a specified input. His approach is only applied to functional specifications of components by analyzing input and output domains. He also proposed the testability improvement of components by modifying the inputs and outputs.

In [2], the author proposed the PIE (Propagation-Infection-Execution) technique. This technique is based on mutation analysis to predict a location's ability to cause program failure if the location were to contain a fault. The PIE technique measures the probabilities that 1) a location is executed; 2) a change to the source program causes a change in the resulting internal computational state; 3) a forced change in an internal computational state propagates and causes a change in the program's output. This technique is only applied to source code.

Voas and Miller [3] presented another approach analyzing component testability by evaluating the information loss of the component. The information loss is expressed via the domain/range ratio (DRR) of a component specification. The DRR of a component is given by the ratio of the cardinality of the input to the cardinality of the output.

In the communication software area, Petrenko et al. [4] investigated testability of communication software which is

modeled by a composition of finite state machines, then Karoui et al. [5] proposed a testability metric for communication software modeled by relations.

For object-oriented software, many approaches have been presented. Chidamber and Kemerer [6] presented a set of six metrics for object oriented designs: weighted methods per class, depth of inheritance tree, number of children, coupling between object class, response for a class, lack of cohesion in methods. These metrics can be seen as testability measures. Payne et al. [7] proposed the use of software contract for each class in order to improve the software testability. The software contract consists of three essential elements: 1) an invariant expression that defines consistency for the classs state-space; 2) a precondition for each method that defines the conditions under which the method can be invoked; 3) a postcondition for each method that defines what the method does.

Ghosh [8] presented an approach combing the program mutation based on keywords and creation of mutants based on conflict graphs created by performing static analysis of the code. However, this approach can only be applied to concurrent object oriented programs in Java. Baudry and al. [9] are interested in class interactions. They build the class dependency graph by basing it on the class diagram. This graph is used to evaluate the complexity of class interactions, which is seen as testability measures. Kansomkeat and al. [10] proposed a method to measure testability of a class-component based on data flow analysis and considering def and use locations. This method analyzes execution and propagation probabilities from the bytecode in binary class files. The execution probability is the percentage of faulty locations executed. The propagation probability is the percentage of faulty locations for which an input caused incorrect output.

Jungmayr [11] stated that dependencies between components in software have a large effect on testability, so he defined the metrics for software dependencies as well as the concept of test-critical dependencies to identify them and subsequently removing them using dedicated refactoring.

Harman and al. [12] proposed to transform program to make it easier to generate test data for it, it means the improvement of programs testability. This approach is only applied to the source code level.

In [13], [14], the authors proposed the testability analysis for data-flow software. This approach was implemented in the SATAN tool, which can be applied to analyze testability for software designs as well as for source code.

We state that these testability approaches were proposed for applying to certain specific application domains. Each work is based on different sources of the software, like source code, design or specification. However, non of them allows testability of reactive systems to be analyzed by basing them on transition models. Hence, in this work we focus on analyzing transition-based testability. Our testability approach may be a useful guide for testers using transition-based testing.

The paper is organized as follows. In Section 2, we present reactive systems. Some transition-based models are introduced in Section 3. Transition-based coverage criteria are discussed in Section 4. We present a testability analysis approach based on Markov chains in Section 5. A case study is analyzed in Section 6. We finish by the conclusions and future work.

2. Reactive Systems

The term *reactive system* is used to designate systems that permanently interact with their environment and to distinguish them from *transformational systems*. Reactive systems have to react continuously to their environment at a frequency determined by that environment [15]. Moreover, we distinguish between *interactive* and *reactive* systems: the interaction speed of interactive systems (e.g., operating systems, web server) depends on the systems, while the interaction speed of reactive systems depends on the environment. Typical examples of reactive systems are process control in industry, embedded systems in trains, aircrafts…

The main characteristics of these safety critical systems are the followings.

- They are *deterministic*: the execution of a reactive system can be viewed as an infinite sequence of input/output vectors, where, at each step, the output values are completely determined by the past and present inputs, *i.e.* some temporal logics. The same inputs and the same internal state produce the same outputs.
- They are submitted to the *bounded memory* constraint: the output depends on the input and internal state of the system. System states are generally stocked in memory.

2.1 Synchronous reactive systems

The synchronous paradigm is very well recognized for designing critical reactive systems. The synchronous approach supposes that reaction time of the system is null. Synchrony divides time into discrete instants.

The inputs and outputs of the system are described by their flows of values along time. If x is a flow, we will note x_n as its value at the n^{th} instant of the system. A system consumes input flows and computes output flows, possibly using local flows which are not visible from the environment. Local and output flows are defined by equations. An equation $x = y + z$ defines the flow x from

the flows *y* and *z* in such a way that, at each instant *n*, $x_n = y_n + z_n$.
Let *T1* and *T2* be the types of inputs and outputs of the system, and S be the set of internal states of the system. The system can be represented by two following functions:
The output function f: $S \times T1 \rightarrow T2$.
The transition function g: $S \times T1 \rightarrow S$.
There exists an initial state $s_0 : S$, such as:

$$o_k = f(s_{k-1}, i_k)$$
$$s_k = g(s_{k-1}, i_k)$$

where where i_k, o_k, s_k are respectively the input, the output and the internal state of the system at instant k and s_{k-1} is the internal state of the system at instant $k-1$.

2.2 Data-flow and control-flow approaches

Reactive systems are generally described as two parts: computation and control.
The *computation part* is usually described with a system of equations, i.e. a data-flow approach and therefore can be modeled as operator diagrams, in which each operator can be a basic operator or a composition of operators. In operator diagrams, operators are connected by communication channels, data is processed by traversing the diagrams. Note that a diagram may be represented hierarchically: a sub-system may be seen as an operator.
The *control part* means changing the behavior according to external events originating either from sensors and user inputs or from internal program events. System behavior is mainly regular, but can switch instantaneously from one behavior to another. In this case, the system is usually composed of a high level control oriented sub-system which executes different data processing for each state of the system. This control part is generally represented by transition-based models.
System behaviors are called *running modes*. Each mode is a big control law, (*i.e.* a computation part), generally described as data-flow equations. Switching between these modes is described by transition-based models (i.e. control part). Mode-automata [16] has been proposed to describe at the same time data-flow and control-flow parts of the system: data-flow equations are attached to each state (mode) of an automaton. Mode-automata can be combined in order to design hierarchical models.

3. Transition-Based Models

Many transition-based models are used for modeling reactive systems, like finite state machine (FSM), extended finite state machine (EFSM) and statecharts. In this work, we are particularly interested in EFSM. An EFSM [17] is defined as a tuple as follows:

$$M = (S, I, O, s_0, E, T, V)$$

where

- S is a set of states,
- I is a set of inputs,
- O is a set of outputs,
- s_0 is the initial state,
- E is a set of events,
- T is a set of transitions,
- V is a store represented by a set of variables.

All above sets are finite. All the inputs, the outputs and the variables are typed.
A transition of a EFSM is a tuple (s, l, s'), where $s, s' \in S$ are the source state and target state, respectively. The label l is defined as $e[g] = a$ where $e \in E$, g is a guard, *i.e.* a condition (assuming a standard conditional language) that guards the transition from being taken when e is true, and a is a sequence of actions (assuming a standard expression language including assignments). All parts of a label are optional.
In an EFSM, the conditions of the transition include boolean expressions. Expressions are built from inputs, variables and constant values, with the usual arithmetic and relational operators. The actions of the transitions may include assignment to variables.

4. Transition-Based Coverage Criteria

Test generation is usually ruled by an adequacy criterion, providing a measure to justify the effectiveness of test sequences in terms of revealing of errors. Many coverage criteria [18], [19], [20], [21] have been developed for transition based modeling notations. In this section, we introduce the most common transition-based coverage criteria:

- *State coverage*: we need to ensure that every state in the model can be reached and visited at least once by some test sequences.
- *Transition coverage*: we need to ensure that every transition in the model can be traversed at least once by some test sequences.
- *k-transition coverage*: this criterion guarantees all possible transition sequences of length k, $k \in N$, will be tested.
- *Full predicate coverage*: this criterion requires that each clause in each predicate on each transition is tested independently.
- *Path coverage*: every path must be traversed at least once.

The state coverage and transition coverage [18], [19], [21] are the most basic and common criteria. These criteria inspire code-based testing criteria: state coverage is similar to statement coverage, and transition coverage is similar to

branch coverage. Transition coverage is evidently stronger than state coverage.

The *k*-transition coverage was proposed in [20]. If *k* is equal to 1, this criterion becomes transition coverage. If *k* is equal to 2, it is the transition-pair coverage defined in [18], [19], [21]. A set of test sequences covers all transition sequences of a fixed length *k* does not necessarily cover a set of all sequences of length $i \in \{1,\ldots, k-1\}$.

In [18], the authors proposed the full predicate coverage, which is similar to the code-based testing criterion of modified condition/decision coverage. Full predicate coverage is based on the philosophy that each clause should be tested independently, that is, while not being influenced by the other clauses. In other words, each clause in each predicate on every transition must independently affect the value of the predicate.

The path coverage [19] corresponds to exhaustive testing of the control structure of the model. In practice, this is usually not practical because such models typically contain an infinite number of paths due to loops. This criterion is similar to complete sequence coverage proposed in [18]. A complete sequence is defined as a sequence of state transitions that forms a complete practical use of the system. The number of complete sequences may be infinite.

5. Testability Analysis Approach

In this section, we present a testability measurement approach based on Markov chains, which are built from EFSM by adding transition probabilities.

5.1 Markov chains

Transition-based models can be augmented with probabilistic information, so that functions commonly used can be analyzed and tested more thoroughly than other ones. As a basic assumption for usage-based testing, if some functions are used more often, the likelihood that a fault is going to be triggered through such usage is also higher. Therefore, we need to focus on highly used parts in the transition-based models.

EFSM is augmented with probabilistic information regarding frequently used parts in the form of Markov chains. The additional information for the EFSM is the probabilities associated with different state transitions that satisfy the following property:

- From the current state $x_n = i$ at stage n, the probability of state transition to state $x_{n+1} = j$ for the next stage $n+1$ is denoted as p_{ij}, which is independent of the history, that is:

$$P\{x_{n+1} = j \mid x_n = i, x_{n-1} = s_{n-1}, \ldots, x_0 = s_0\} = P\{x_{n+1} = j \mid x_n = i\} = p_{ij}$$

  The complete history is summarized in the current state. The next state is independent of all the past states given the current state. This is called the Markov property.
- Probabilities p_{ij} satisfy the following equations:

$$0 \le p_{ij} < 1, \text{ and } \sum_j p_{ij} = 1$$

- If the above conditions hold for every state in an EFSM, the EFSM forms a Markov chain.

In Markov chains, transitions are probabilistic instead of deterministic. Messages or conditions in the corresponding EFSM are augmented with the associated probability.

A Markov chain can be completely described by a *transition matrix*, $P = [p_{ij}]$. In this matrix, state labels are represented as indices and transition probabilities as entries. Note that the matrix is square and each of its row sums to one.

Markov chains have been used to generate tests, determine when to stop testing, and reason about the outcome of testing in statistical testing [22], [23], [24], [25].

5.2 Control-flow testability measures

EFSMs can be augmented with transition probabilities in forms of Markov chains. In our study, most of the cases, EFSMs are often specified by designers. To build Markov chains, we need to assign transition probabilities to EFSM by basing them on usage model [26]. Usage model characterizes operational use of the software system. Operational use is the intended use of the software in the intended environment. Usage model should be defined in functional specification, usage specification and test specification.

Whittaker and Poore [22] describe three approaches for constructing usage model:

- if no information is available concerning expected use of the software, uniform probabilities can be assigned across the exit arcs for each state, called *uninformed approach*;
- transition probabilities can be assigned based on hypothesized use of the software, called *intended approach*;
- transition probabilities can be assigned based on actual use measurement, called *informed approach*.

These three approaches can cover many situations. The informed approach with actual user sequences is the best. However, this approach is only applied when the software is available. Next best is the intended approach where user sequences are evaluated by hypothesizing runs of the software by a careful and reasonable user. As a last resort, the uninformed approach is used.

In reactive systems analysis, as mentioned previously, we are particularly interested in some variants of EFSM, like SSM in SCADE or STATEFLOW in SIMULINK. In such formalisms, each transition is triggered by a guard represented by a boolean expression. The boolean expression is a predicate on a set of inputs and variables.

In this work, we use a statistical evaluation to compute usage distribution. When no information is available concerning expected use of the software, statistical evaluation is used instead of using uniform probabilities. Statistical evaluation allows computation of the probability that a guard is satisfied, i.e. when the corresponding boolean expression takes the true value. This approach gives more precise information on activation of transition.

Once the Markov chain is built from EFSM and transition probabilities, we can compute its equilibrium, *i.e.* when the Markov chain becomes stationary. In such a state, the stationary probability π_i for being in state i remains the same before and after state transitions over time.

Let $\Pi = [\pi_1, \pi_2, \ldots, \pi_n]$ be the vector of stationary probabilities for the n states. This vector can be found for a given transition matrix P as the unique stochastic vector solution to the eigenvector equation [24]:

$$\Pi = \Pi P$$

The eigenvector Π is sometimes called the Perron eigenvector. This equation is equivalent to the system of equations:

$$\pi_1 = \pi_1 p_{11} + \pi_2 p_{21} + \ldots + \pi_n p_{n1}$$
$$\pi_2 = \pi_1 p_{12} + \pi_2 p_{22} + \ldots + \pi_n p_{n2}$$
$$\ldots$$
$$\pi_n = \pi_1 p_{1n} + \pi_2 p_{2n} + \ldots + \pi_n p_{nn}$$
$$1 = \pi_1 + \pi_2 + \ldots + \pi_n$$

where p_{ij} is the transition probability from state i to state j. The stationary probability π_i indicates the expected appearance rate of state i after the Markov chain reaches the equilibrium. Since each state is associated with some part of the actual software, this information allows testers to determine which parts of the software will get most attention during testing.

We call *state executability* probability π_i. The state executability may allow to predict how easy to generate test cases to activate a specific state i of the software. The basic coverage criterion of FSM is state coverage. For states with low executability, test cases which cover such states are not easy to generate. Therefore, if executability of a state i is very low, computation part associated with state i can be considered as difficult to test, *i.e.* low testability. Otherwise, the computation part associated with state i has high testability in terms of execution.

Moreover, transition-based testing in general consists of generating test cases activating a set of sequences of state transition. The difficulty of generating a test case depends on the difficulty of activation of each transition in the sequence. The difficulty of transition activation is simply the transition probability. Therefore, the difficulty of generating a test case may be measured by the product of probabilities of all transitions activated by that test case. We call *sequence executability* this product of probabilities. The sequence executability may be considered as the testability measure of a sequence.

Hence, we propose the use of two testability measures based on Markov chains:

- *State executability*: predicts how easy it is to execute a state in stationary functioning of the system.
- *Sequence executability*: predicts how easy it is to activate a transition sequence.

Specially, when a Markov chain has an absorbing state (*i.e.* it is impossible to leave that state), the system stays most time in that state.

On the one hand, these measures may help testers to take into account how difficult or easy it is to generate test cases, which cover some specific states or sequences. On the other hand, designers can improve their transition-based models to facilitate generation of test cases.

6. Application

In this section, we apply our approach to analyze the testability of the B01_AUTOMATON[1] SSM (Figure 1), this SSM is designed in the SCADE environment. Then we use the GATeL tool [27] to generate test cases for the B01_AUTOMATON. Finally, we compare our testability measures to the testing cost for generating test cases for the B01_AUTOMATON by the GATeL tool.

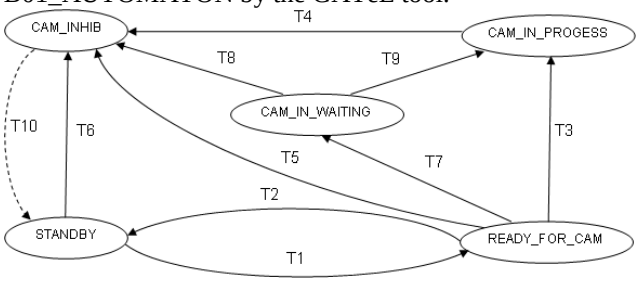

Fig. 1 The B01_AUTOMATON.

6.1 Testability measures of the B01_AUTOMATON

In this automaton, STANDBY is the initial state and CAM_INHIB is the final state. Each transition T_i, $i \in 1, 2, \ldots, 10$ is activated by a boolean expression, which is called as activation condition for transition. For a simplified representation, we don't present all activation

[1] Provided by ASTRIUM for the SIESTA project.

conditions for transition, but only the obtained transition probabilities by performing a statistical evaluation of activation conditions. The transition probabilities of the B01_AUTOMATON are presented in Table 1.

Table 1: Transition probabilities

Transition	T1	T2	T3	T4	T5
Probability	0.063	0.094	0.078	0.5	0.094
Transition	T6	T7	T8	T9	T10
Probability	0.563	0.078	0.25	0.25	0.1

We now transform the automaton into Markov chain by adding transition probabilities. Then we compute testability measures of B01_AUTOMATON: state executability and sequence executability. State executabilities obtained via the eigenvector of Markov chain in Table 2.

Table 2: State executabilities

State	Executability
STANDBY	0.136381
READY_FOR_CAM	0.024797
CAM_IN_WAITING	0.003875
CAM_IN_PROGRESS	0.005811
CAM_INHIB	0.829136

Test sequences of transition are generally determined by the testers or the designers with the use of domain knowledge and experience. For B01_AUTOMATON, we can determine all sequences without loop and one sequence with loop once *Seq6* in Table 3.

Table 3: Test sequences

Sequence	Transition
Seq1	T1, T3, T4
Seq2	T6
Seq2	T1, T5
Seq4	T1, T7, T8
Seq5	T1, T7, T9, T4
Seq6	T1, T2, T6

Sequence executability measures obtained by the product of all the probabilities of transitions of the sequence are presented in Table 4.

Table 4: Sequence executabilities

Sequence	Transition
Seq1	0.002441
Seq2	0.562500
Seq2	0.005859
Seq4	0.001221
Seq5	0.000610
Seq6	0.003296

6.2 Experimentation with the GATeL tool

GATeL [27] is a tool supporting test case generation from Lustre descriptions. GATeL allows us to define test objectives and then test cases are generated in order to satisfy these test objectives. A test objective can be a safety property or declarative characterization of some interesting states of the system under test. Test case generation is handled by solving a constraints system involving program data flows at an arbitrary stage. However, some input data flow values are not involved at each computation stage, they can remain undefined when the constraints system is solved. Thus, an instantiation process on the remaining input variables is performed in order to obtain a better structural coverage. This process requires testers to split interactively the input domain of test cases. Therefore, in this experiment, we measure test case generation cost by basing it on the time for generation and number of splittings done by testers.

We first generate the Lustre code from the B01_AUTOMATON then we define the test objectives satisfying two basic criteria: state coverage and path coverage. For state coverage, we obtained some cost measures in Table 5, where number of test cases is abbreviated as NTC, total time [2] of test generation as TTTG, generation time per test case as TPTC and number of splittings as NS.

Table 5: Testing cost for state coverage

State	NTC	TTTG	TPTC	NS
STANDBY	4	19	4.8	3
READY_FOR_CAM	9	69	7.7	8
CAM_IN_WAITING	3	9	3	2
CAM_IN_PROGRESS	5	50	10	4
CAM_INHIB	8	60	7.5	7

Comparing the testing cost for state coverage with the state executabilities, we find that the generation time per test case corresponds to the state executabilities. For example, state CAM_INHIB is the most executable and the generation time per test case for testing this state is the shortest; state CAM_IN_PROGESS is the least executable and the generation time per test case for testing this state is the longest. However, we find that the number of splittings is proportional to the number of test cases, but not to the testability measures for this case study.

Concerning path coverage, we only define the test objectives for test sequences defined in Table 3. We obtained the results in Table 6.

We can state that the generation time per test case corresponds to the sequence executabilities, for example, sequence *Seq2* is the most executable, so the generation time per test case for this sequence is the shortest;

[2] Generation time is measured in millisecond.

sequence *Seq5* is the least executable, hence the generation time per test case for this sequence is the longest. However, we find that the number of splittings is not related to the number of test cases and the sequence executabilities.

Table 6: Testing cost for path coverage

Sequence	NTC	TTTG	TPTC	NS
Seq1	5	40	8	5
Seq2	3	10	3.3	5
Seq2	2	20	10	4
Seq4	3	40	13.3	5
Seq5	4	60	15	10
Seq6	3	29	9.7	4

This experiment shows that transition-based testability measures correspond to generation time of test cases by the GATeL tool, but do not correspond to splitting cost.

7. Conclusions

Models based testability analysis brings to testers and designers important predictive measures on software quality. In this work, we focus on analyzing testability of reactive systems by basing them on transition-based models. We have presented testability measures based upon Markov chains, which are augmented transition-based models with probabilistic information. We have applied the measurement on the B01 AUTOMATON and we have also compared the obtained testability measures to the testing effort required by the GATeL tool for test generation. The results show a strong link between testability measures and test generation.

This work until now is the first step in testability analysis based on transition models. In the future, we intend to more deeply develop this research and apply it to more complex industrial applications.

References

[1] R. S. Freedman, “Testability of Software Components,” IEEE Transactions on Software Engineering, vol. 17, no. 6, 1991, pp. 553–564.

[2] J. M. Voas, “Pie: A dynamic failure-based technique,” IEEE Transactions on Software Engineering, vol. 18, 1992, pp. 717–727.

[3] J. M. Voas and K. W. Miller, “Software testability: The new verification,” IEEE Software, vol. 12, no. 3, 1995, pp. 17–28.

[4] A. Petrenko, R. Dssouli, and H. K¨onig, “On evaluation of testability of protocol structures,” in Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI, Amsterdam, The Netherlands, The Netherlands, 1994, pp. 111–124.

[5] K. Karoui, R. Dssouli, and O. Cherkaoui, “Specification transformations and design for testability,” in Proceedings of IEEE Global Telecommunications Conference, London, England, 1996.

[6] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” IEEE Trans. Softw. Eng., vol. 20, no. 6, 1994, pp. 476–493.

[7] J. E. Payne, R. T. Alex, and C. D. Hutchinson, “Design-for-testability for object-oriented software,” Object Magazine, vol. 7, no. 5.

[8] S. Ghosh, “Towards measurement of testability of concurrent objectoriented programs using fault insertion: A preliminary investigation,” in SCAM ’02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, Washington, DC, USA, 2002.

[9] B. Baudry, Y. L. Traon, and G. Suny, “Measuring and improving design patterns testability,” in 9th IEEE International Software Metrics Symposium, 2003.

[10] S. Kansomkeat, J. Offutt, and W. Rivepiboon, “Class-component testability analysis,” in Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, 2006.

[11] S. Jungmayr, “Testability measurement and software dependencies,” in Proc. of the 12th International Workshop on Software Measurement, 2002.

[12] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, “Testability transformation,” IEEE Transactions on Software Engineering, vol. 30, 2004, pp. 3–16.

[13] Y. Le Traon and C. Robach, “Testability Measurements for Data Flow Design,” in Proceedings of the Fourth International Software Metrics Symposium, Albuquerque, New Mexico, 1997, pp. 91–98.

[14] T. B. Nguyen, M. Delaunay, and C. Robach, “Testability analysis of data-flow software,” Electron. Notes Theor. Comput. Sci., vol. 116, 2005, pp. 213–225.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous dataflow programing language LUSTRE,” Proceedings of the IEEE, vol. 79, no. 9, 1991, pp. 1305–1320.

[16] F. Maraninchi and Y. R´emond, “Mode-automata: a new domain-specific construct for the development of safe critical systems,” Science of Computer Programming, vol. 46, no. 3, 2003, pp. 219–254.

[17] K. Androutsopoulos, D. Clark, M. Harman, L. Zheng, and L. Tratt, “Control dependence for extended finite state machines,” in Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering, 2009.

[18] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, “Generating test data from state-based specifications,” Journal of Software Testing, Verification and Reliability, vol. 13, 2003.

[19] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan-Kaufmann, 2007.

[20] F. Belli and A. Hollmann, “Test generation and minimization with basic statecharts,” in Proceedings of the 2008 ACM symposium on Applied computing, 2008.

[21] C. Junke, “Critres de tests pour les automates de modes et application au langage scade 6,” in 10es Journes Francophones Internationales sur les Approches Formelles dans l’Assistance au Dveloppement de Logiciels, 2010.

[22] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," ACM Trans. Softw. Eng. Methodol., vol. 2, no. 1, 1993.

Nguyen Thanh Binh graduated from Danang University of Technology in 1997, he got a PhD degree in Computer Science from Grenoble Institute of Technology (Grenoble INP) in 2004. He is currently lecturer in the Information Technology Department, Danang University of Technology, Vietnam. He is dean of IT faculty at Danang University of Technology since 2010. His research interests include software testability, software testing and software quality.

Chantal Robach graduated from the ENSIMAG Computer Science Engineering School — Grenoble in 1973, she got a PhD degree in 1976 and the "Docteur-es-Sciences" degree from the Grenoble Institute of Technology (Grenoble INP), in 1979. She was a junior then senior Researcher at the French National Research Center from 1974 to 1997, and was vice-director of the Computer Science Laboratory (150 persons) from 1992 to 1995. In 1997 she is appointed full Professor at Grenoble INP, head of the LCIS Laboratory from 2002 to 2007 and head of ESISAR « Engineering School on Advanced Systems and Networks » from then.
She directed a research team since 1988, was reviewer for international journals and conferences dedicated to design and test domains (ITC, ETC, ETW, …), published in more than 33 international journals and about 140 international conferences. She has led several expertises for national and international grants attribution, or for french industries. She managed more than 25 industrial grants or contracts, directed about 20 PhD thesis, was involved in 3 international conferences located in France…