Automated Generation of Statistical Test Cases from UML State Diagrams

Philippe Chevalley* and Pascale Thévenod-Fosse
LAAS-CNRS, 7 avenue du Colonel Roche
31077 Toulouse Cedex 4, France
{chevalle, thevenod}@laas.fr

Abstract

The adoption of the object-oriented (OO) technology for the development of critical software raises important testing issues. This paper addresses one of these issues: how to create effective tests from OO specification documents? More precisely, the paper describes a technique that adapts a probabilistic method, called statistical functional testing, to the generation of test cases from UML state diagrams, using transition coverage as the testing criterion. Emphasis is put on defining an automatic way to produce both the input values and the expected outputs. The technique is automated with the aid of the Rational Software Corporation's Rose RealTime tool. An industrial case study from the avionics domain, formally specified and implemented in Java, is used to illustrate the feasibility of the technique at the subsystem level. Results of first test experiments are presented to exemplify the fault revealing power of the created statistical test cases.

1. Introduction

Software testing involves executing a program on a set of test case input values and comparing the actual output results with the expected results. A large number of testing techniques have been defined for programs developed according to hierarchical approaches and written in procedural languages (see e.g., [1]). But, it is now well recognized that these techniques must be revisited to take account of the characteristics of the object-oriented (OO) technology (see e.g., [2, 14, 15]). Even if this technology is more and more used in industrial cases, testing methods for OO programs are still to be improved. The development of high-quality OO software requires appropriate testing to ensure that the software meets its specification. This statement becomes crucial for safety-related software where failures may be a synonym of injuries or financial loss. This paper presents results from an ongoing research work focused on the automatic generation of test cases from Unified Modeling Language (UML) specifications.

The UML [4] is a semiformal language that comprises nine types of graphics, called diagrams. They are used to describe different aspects of a system including static, dynamic, and use-case views. The testing technique we investigate is based on the information provided by one type of diagrams, called state diagram: it is based on Harel's Statecharts [10], and is widely used to represent the dynamic behavior of objects. The UML is supported by several Computer-Aided Software Engineering (CASE) tools in current use for specifying and designing industrial applications. Our aim is to study to what extent the facilities offered by such tools may provide a practical support for the generation of adequate test cases. The paper investigates the automation process of a probabilistic method for generating test inputs, called statistical functional testing [25], with the aid of the Rational Software Corporation's Rose RealTime tool [20].

Statistical testing involves exercising a program with input values that are randomly generated according to a given probability distribution over the input domain. When the focus of testing is fault removal, that is, bug-finding rather than reliability assessment, the effectiveness of the method depends on the adequacy of the distribution in revealing faults with high probability. Here, we will use the information provided by the UML state diagrams to define a probability distribution that is well suited to rapidly trigger all the transitions of the diagrams. This approach belongs to the family of statistical functional testing techniques, as defined in [25]. It should not be confused with (i) random testing — a blind approach that uses a uniform probability over the input domain [7], or (ii) statistical operational testing — that is, when the test samples are randomly drawn from an input distribution representative of operational usage in order to determine

* The author is with Rockwell-Collins France as a CIFRE fellow.

0-7695-1372-7/01 $10.00 © 2001 IEEE

whether or not a piece of software is ready for use (reliability assessment) [17, 21].

The paper is organized as follows. In Section 2, we first discuss related work on test cases generation from UML state diagrams. Section 3 recalls the principle of statistical testing, and introduces the testing technique under investigation. Then, Section 4 presents the case study used to illustrate the feasibility of the proposed approach. It is extracted from a research version of an avionics system provided by the Advanced Technology Center of Rockwell-Collins — the mode control logic part of a Flight Guidance System [16] — and implemented in Java. Section 5 presents the testing technique and gives first experimental results. Future work is outlined in Section 6.

2. Definitions and Related Work

This paper uses the following definitions [19]. A *test* or *test case* is a general software artifact that includes test case input values (or input values, for short), expected outputs for the test case, and any inputs that are necessary to put the software system into the state that is appropriate for the input values. A test case input value comes from the *test requirements* which define specific things that must be satisfied or covered during testing: for example, reaching statements are the requirements for statement coverage. A *testing criterion* (e.g., statement coverage) is a rule or a collection of rules that impose test requirements on a set of test cases. A *testing technique* guides the tester through the testing process by including a testing criterion and a process for creating test case input values. Testing criteria may be related to the coverage of either a model of the program structure (e.g., the program control flow graph) or a model of its functionality (e.g., finite state machines that may be used to describe some software functions). The former case defines structural (or white box) testing techniques, and the latter case defines functional (or black box) testing techniques [1].

There is a general agreement among the testing community that the OO development process and associated concepts raise a number of problems from the perspective of testing (see e.g., [2, 14, 15]). Among the questions that must be answered, let us quote: How to determine the unit and integration testing levels (decentralized architecture of objects)? Which models and associated testing criteria should be used to produce effective input values? How to solve the controllability and observability problems that are increased by object encapsulation? etc. This paper is related to the second issue. Our aim is to investigate a functional testing technique based on UML state diagrams, emphasis being put on defining an automatic way of producing both the input values and the expected outputs. In previous work that addresses this topic, we identified two different axes depending on the information represented by UML state diagrams.

The first axis is centered on use case documents [9, 11, 22]. Based on these documents, Frölich and Link [9] explain how test cases can be automatically generated with the aid of Artificial Intelligence (AI) methods. Their use cases are systematically transformed into UML state diagrams, which then represent the behavior of the system under test. State diagrams are further more mapped to a planning language and then a planning tool (called *graphplan*) yields the different test cases as solutions to a planning problem. The testing criterion retained is the coverage of every transition of the state diagram. The work of [22] is focused on a similar process. The authors also propose to convert use cases into formal scenarios using annotated state diagrams and then to derive test cases from these state diagrams in a systematic manner. The method, called SCENT (for SCENario-based validation and Test of software), derives test cases to cover every transition. The work presented in [11] addresses the issue of testing distributed components and their interactions specified in UML state diagrams using commercial modeling tools such as Rational Rose. This technique is similar to the two previous ones: state diagrams represent scenarios of use cases and the testing criterion is transition coverage.

The second axis is the most similar to ours in that it is centered on UML state diagrams that are used to represent the behavior of an object [3, 12, 19]. Binder's book [3] addresses a wide spectrum of concerns. Among them, the author considers the UML from a testing perspective: a detailed analysis of each UML diagram is presented and generic test requirements for each diagram are identified, which can be used to develop application-specific test cases. As regards the state diagrams, emphasis is put on the use of the FREE (Flattened Regular Expression) state model that is consistent with all requirements of the state diagrams: according to this model, in case of inheritance the state diagram of a class is expanded to represent the behavior of inherited features. Then, conventional state-based testing criteria (see e.g., [5]) can be applied to the expanded diagrams. In [12], the authors propose a transformation method from state diagrams into extended finite state machines and flow graphs. The transformation consists in flattening the hierarchical and concurrent structure of states and eliminating broadcast communications, while preserving both control and data flow in the UML state diagrams. Then, it is shown that conventional control and data flow testing criteria (see e.g., [8]) can be applied to the transformed models. Unlike the previous techniques, the work of [19] is concentrated on the definition of testing criteria related to the coverage of UML state diagrams, without flattening transformation.

Four testing criteria are defined from the change event enabled transitions: (1) transition coverage, (2) full predicate coverage, (3) transition-pair coverage and, (4) complete sequence coverage where a complete sequence is a sequence of state transitions that form a complete practical use of the system. To evaluate their criteria, the authors have developed a proof-of-concept test data generation tool, called UMLTest, which is integrated with the Rational Rose tool. An empirical study conducted on a small size program (400 lines of C code) has demonstrated the feasibility of the proposed testing criteria. More empirical studies are still required to evaluate and compare the effectiveness of the testing techniques associated with each of the four criteria.

To conclude on the work pointed out in this section, it is worth noting that all the testing techniques involve a *deterministic* process for creating test case input values: it consists in selecting a priori a set of test cases such that each element defined by the chosen testing criterion is covered once. As far as we know, a probabilistic process, such as the statistical approach presented in the next section, has not yet been investigated from UML diagrams.

3. Statistical Functional Testing

Statistical testing involves exercising a program with input values that are randomly generated according to a given probability distribution over the input domain. Previous work related to procedural programs [25] has shown that the information provided by testing criteria may be used as guides for determining input distributions from which effective test case input values are produced. Section 3.1 recalls the motivation and the principle of the approach, called *statistical structural or functional testing*, depending on whether the testing criteria are related to the coverage of a structural or a functional model of the software. Based on this background, Section 3.2 introduces the testing technique we are currently investigating, that is statistical functional testing designed from UML state diagrams with the support of the Rational Rose RealTime tool.

3.1. Background

Given a testing criterion, the conventional method for generating input values proceeds according to the deterministic process. But a major limitation is due to the imperfect connection of the criteria with real faults, so that exercising each element defined by such criteria once (e.g., each transition of a finite state machine) may not be sufficient to ensure a high fault exposure power. Yet, the criteria provide us with relevant information about the target piece of software. A practical way to compensate for criteria weakness is to require that each element be exercised several times. This involves larger sets of input values that have to be automatically generated: it is the motivation of *statistical testing* designed according to a testing criterion [25]. In this approach, the information provided by criteria is combined with an automatic way of producing input values, that is, a random generation.

When using the statistical approach for generating input values, the activation number of each element defined by the testing criterion is a random variable. Two factors play a part in ensuring that on average every element is exercised several times, whatever the particular input values randomly generated according to the input distribution and within a moderate test duration. The first factor is the input probability distribution, from which the input values are randomly drawn; the second factor is the test size, i.e., the number of input values that are generated. Both of them have to be determined according to the chosen testing criterion.

The *determination of the input distribution* is the corner stone of the method. The aim is to search for a probability distribution that is proper to rapidly exercise every element defined by the criterion. Given a criterion C – say, transition coverage – let S be the corresponding set of elements – the set of transitions. Let P be the occurrence probability per execution of the least likely element of S. Then, the distribution must accommodate the highest possible P value. Depending on the complexity of this optimization problem, the determination of the distribution may proceed either in an analytical way or in an empirical way (see e.g., [24, 25] for detailed examples). The first way supposes that the activation conditions of the elements can be formulated as a function of the input parameters. Then their probabilities of occurrence are a function of the input probabilities, facilitating the derivation of an input distribution that maximizes the frequency of the least likely element. The second way is based on simulations to tune progressively the input distribution until the frequency of each element is deemed sufficiently high (see Section 3.2).

The *test size* N must be large enough to ensure that each element of S is exercised several times under the defined input distribution. The assessment of a minimum test size N is based on the notion of test quality with respect to the testing criterion. This quality, denoted q_N, is the probability of exercising at least once the least likely element of S. As a result, the test quality q_N and the test size N are linked by the relation: $(1-P)^N = 1-q_N$. Thus, knowing the value of P for the derived input distribution, if a test quality objective q_N is required, the minimum test size N is given by Relation (1).

$$N = \ln(1-q_N) / \ln(1-P) \qquad (1)$$

Returning to the motivation of statistical testing – exercising several times each element defined by the criterion, let n be the average number of times the least likely element is exercised. Then $n = P.N = P.\ln(1-q_N) / \ln(1-P)$ from Relation (1). For small values of P (i.e., $P < 0.1$, which is a realistic assumption for non trivial applications), $\ln(1-P) \cong -P$. Thus, we get Relation (2) that establishes a link between q_N and n. For example, $n \cong 3$ for $q_N = 0.95$, and $n \cong 5$ for $q_N = 0.99$.

$$n \cong - \ln(1-q_N) \qquad \textbf{(2)}$$

A number of experiments have already been conducted on *procedural programs* coming from various critical application domains. They confirmed the soundness of the statistical testing techniques, which have repeatedly exhibited a higher error detection power when compared with deterministic testing techniques and random (uniform) testing (see e.g., [25] that reports on a sample of these experiments). The main conclusion arising from this previous work is that statistical testing is a suitable means to compensate for the tricky link between testing criteria and software design faults. As regards *OO programs*, a first empirical investigation of statistical functional testing was successfully conducted on the control program of a production cell [26]: the program was developed using the Fusion method [6] and implemented in Ada 95. But because no CASE tool supports the Fusion method, this work was based on functional models (automata) manually drawn from the information got from Fusion analysis and design documents.

3.2. CASE Tools Applied to Statistical Functional Testing

Several CASE tools in current use provide support for graphical behavioral modeling. The expected benefits of using a CASE tool for supporting the generation of test cases are noticeable. First, no specific model of the software is required for the purpose of testing. Second, since most of these tools offer prototyping or simulation facilities, they provide the test designer with an automatic solution to generate the expected outputs (assuming that the models are correct). These benefits hold for any testing technique, whether deterministic or statistical.

In the case of statistical functional techniques, the analytical determination of the input distribution may be very tedious as soon as the activation conditions of the elements to be covered during testing are highly dependent on each other. Moreover, it becomes impossible when the model complexity prohibits us from getting the set of equations relating the element probabilities to the input distribution. For many industrial applications, the practical limitation of an analytical search is rapidly reached, even at the component (subsystem) level.

A pragmatic way to proceed is then to conduct an *empirical search* of the input distribution. It is based on the following principle: starting from an initial input distribution (e.g., the uniform one), several large sets of input values are generated; the models are executed with these values to count the number of times each targeted element is covered; the execution results are analyzed to determine the activation conditions of the least covered elements in order to improve the input distribution. This iterative process is stopped when the frequency of each element is deemed sufficiently high. Note that the empirical process does not yield the actual value of the probability P of exercising the least likely element. Hence, the test size N cannot be assessed from Relation (1). In that case, Relation (2) is used to tune the test size N according to a test quality objective q_N.

The feasibility of such an empirical procedure requires the use of a CASE tool that provides facilities to program model execution and to instrument the model in order to collect statistics about coverage measures during execution. A first investigation was centered on the STATEMATE environment [24]. The test criterion used was the coverage of the basic states (states having no offspring) of the Statecharts. This work showed the feasibility of the empirical procedure on a case study from the nuclear field, and the results of test experiments supported the high fault revealing power of the test cases. Yet, for real-time software systems (such as the avionics case study presented in Section 4) that have to react to small changes of their environment (e.g., a pilot changes the position of a two-way switch), it should be important to focus on all the transitions between the states of the behavioral models, rather than just the basic states. Obviously, using transition coverage as the testing criterion makes the empirical search of the input distribution an order of magnitude more complex – or even unfeasible, as we will see in Section 5. Hence, how to design another – practical and automatic – procedure is the aim of our investigation conducted with the aid of the Rational Rose RealTime tool.

4. Case Study: an Avionics System

As support of our theoretical work, the Advanced Technology Center of Rockwell-Collins provided us with a research version of an avionics system, the mode control logic of a Flight Guidance System (FGS) for a General Aviation class aircraft. The FGS compares the measured state of an aircraft (present position, speed, attitude) to the desired state and generates pitch and roll guidance commands in order to minimize the difference between the measured and desired states. An FGS can be broken into the mode control logic and the flight control laws. The case study focuses on the *mode control logic*. This system accepts commands from the pilots, the Flight Management System (FMS), and information about the current state of the aircraft to determine which system modes are active. The active modes in turn determine which flight control laws are used to generate the pitch and roll guidance commands. When engaged, the autopilot translates these commands into movement of the aircraft's control surfaces.

From the SCR (Software Cost Reduction) specification of this industrial example [16], we developed a UML model in the Rose RealTime environment. This model is made of 115 classes, including 14 classes with a functional behavior that determines the bulk of the model. In this real-time environment, these 14 classes are modeled with capsules, which are a specialization of the general concept of a UML class [23] and their behavior is modeled with UML state diagrams.

The Java implementation to be tested has been developed at the Advanced Technology Center from the SCR specification and is also made of 115 Java classes (6500 LOC). It is designed as an automaton that is updated at each cycle of execution. A cycle may be divided into three activities. First, input variables are set from values provided by the aircraft buses. Second, the program reacts to input values. Third, output variables are produced and sent on the aircraft buses.

4.1. UML State Diagrams

The functional behavior of this avionics system has been modeled with a total of 14 concurrent UML state diagrams. Figure 1 depicts as an example a subset of four diagrams referring to the *Aircraft Data Sources* and *Autopilot* classes of the FGS case study: two diagrams represent the behavior of the super-states Aircraft Data Sources and Autopilot, and the other two describe the behavior of the composite states DISENGAGED and ENGAGED. The *Aircraft Data Sources* state diagram consists of two basic states, SPEED_OK and TOO_FAST, with SPEED_OK as the initial state. These states are *or*-states, that is, being in the *Aircraft Data Sources* state diagram implies being either in SPEED_OK or in TOO_FAST but not in both. A Boolean variable *Overspeed* is set according to the active state. This variable is involved in the triggering of transitions in the *Autopilot* state diagram. The autopilot has two composite *or*-states, DISENGAGED and ENGAGED. In the DISENGAGED state, the autopilot is idle and information is displayed to the flight crew to indicate whether the autopilot has been disengaged in a bad condition (the Warning substate) or in a normal condition (the Normal substate). In the ENGAGED state, the autopilot elaborates guidance commands to drive the aircraft's control surfaces. This is the goal of the Normal substate. In the Sync substate, the automatically generated flight guidance is overridden by manually generated commands from the pilots.

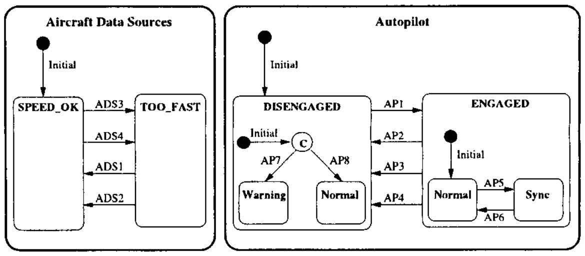

Figure 1. State diagrams for the Aircraft Data Sources and Autopilot classes

4.2. Description of Events

The triggering of transition represents the response of the system to events generated externally by the environment or internally by the system itself. An event occurs when the value of any variable changes. Tables 1 and 2 show the events that trigger the labeled transitions of Figure 1. The formalism used in these tables is derived from the SCR formal notation [16]. The event "@T(c)" means that the Boolean expression c has to change from false to true. Reciprocally, the event "@F(c)" is defined as @F(c) = @T($\neg$ c), which means the Boolean expression c has to change from true to false. In addition to an event, a guard expression may condition the triggering of a transition so that the transition can only be taken if that expression is true. A guard is defined after the keyword *WHEN*. For example, transition AP1 means that the autopilot changes from DISENGAGED to ENGAGED when the event "@T(AP_Engage_Switch = ON) WHEN (AP_Disconnect_Bar = UP)" occurs. This event corresponds to the flight crew pressing the button requesting the engagement of the autopilot from OFF to ON when the button requesting the disconnection of the autopilot is in the position UP.

The UML state diagrams shown in Figure 1 look simple, but the triggering of their transitions is subject to events that are not so trivial. Events occur on a change of value that is expressed with the notation "@T(c)" and "@F(c)". Let us take an example to understand that some transitions may be difficult to trigger due to this particular definition of events. Consider an event E defined with the expression "@T(c) WHEN d" where c and d are Boolean expressions. If the expression c becomes true when the guard d is false, the event does not occur.

Table 1. Aircraft Data Sources transition table

Label	Previous State	Events	New State
ADS1	TOO_FAST	@T(Indicated_Airspeed <= Vmo AND NOT Above_Transition_Altitude)	SPEED_OK
ADS2	TOO_FAST	@T(Indicated_Mach_Number <= Mmo AND Above_Transition_Altitude)	SPEED_OK
ADS3	SPEED_OK	@T(Indicated_Airspeed > (Vmo + 10) AND NOT Above_Transition_Altitude)	TOO_FAST
ADS4	SPEED_OK	@T(Indicated_Mach_Number > (Mmo + 0.03) AND Above_Transition_Altitude)	TOO_FAST

Table 2. Autopilot transition table

Label	Previous State	Events	New State
AP1	DISENGAGED	@T(AP_Engage_Switch = ON) WHEN (AP_Disconnect_Bar = UP)	ENGAGED
AP2	ENGAGED	@T(AP_Disconnect_Bar = DOWN)	DISENGAGED
AP3	ENGAGED	@T(AP_Disengage_Left = ON) OR @T(AP_Disengage_Right = ON)	DISENGAGED
AP4	ENGAGED	@T(GA_Pressed)	DISENGAGED
AP5	Normal	@T(SyncLeft = ON OR SyncRight = ON)	Sync
AP6	Sync	@F(SyncLeft = ON OR SyncRight = ON)	Normal
AP7	Entered(DISENGAGED)	@T(GA_Pressed) AND (NOT Overspeed')	Warning
AP8	Entered(DISENGAGED)	NOT(@T(GA_Pressed) AND (NOT Overspeed'))	Normal

At the next cycle, if d becomes true and c does not change, the event does no more occur, even if the expression c is true. Indeed, the expression c was true at the previous cycle and is still true at the current cycle, while the event E request a change of value from false to true, which is not the case. This example shows that the event E uses *both* the previous and current values for the expression c. If we want to trigger the event E at this stage, it will take at least two cycles since the expression c must first change to false and then return to true.

Moreover, a lot of events in the case study are based on two-way switches (ON and OFF) that provoke an event only when their status changes from OFF to ON (changes from ON to OFF have no effect). For instance, the engagement of the autopilot is requested when the switch *APEngage* is set to ON whereas it is not disengaged when this switch is set to OFF. Other switches are used to disengage it. Then, some changes provoke an event whereas others provoke nothing because of this particular representation of switches. Finally, the activation conditions of the transitions may depend on several variables. For instance, the Sync state can be requested only when one of the two switches (*syncLeft* and *syncRight*) is set to ON when both previous values were OFF. Once the Sync state is activated, setting the second switch to ON has no effect.

5. Generation of Statistical Test Cases

To study the feasibility of the statistical approach, we have used a subsystem of the FGS case study. It involves 77 classes and 4 UML state diagrams (those of Figure 1), with a total of 12 transitions. This subsystem is controlled by 11 input variables. An input variable is either a Boolean variable, which represents the position of a two-way button (or switch) that may be operated by the flight crew (e.g., to request the engagement of the autopilot), or a float variable that represents an air data elaborated by other devices (e.g., the indicated airspeed or the pressure altitude).

5.1. Determination of an Input Distribution

For this subsystem, we first used the empirical procedure (Section 3.2) for determining a probability distribution over the input domain, using transition coverage as the testing criterion. Simulation experiments were conducted from a uniform distribution over the input domain: several large sets of input values were generated using different initializations of the random generator; the models were executed with these sets to collect measures of the transition coverage provided by each of them. The results showed that most transitions are never or seldom exercised under the uniform distribution, which means that this distribution does not supply a sound probe of the modeled transitions. The activation conditions of the least (or never) covered transitions were analyzed to improve the distribution, and many try-and-backtrack loops were thus performed each time with a modified input distribution that should increase the triggering probability of the least covered transitions at the previous step. Unfortunately, we observed no significant improvement all along the empirical procedure. The reason for this is that by defining a probability distribution over the input domain, we act on the occurrence probability of values (e.g., the probability that a Boolean variable has the value true), but not on the occurrence probability of value changes (e.g., a Boolean variable changes from false to true). Since the events that trigger the transitions are related to changes, little improvement may be expected using the empirical procedure conducted over the input domain.

Yet, the analysis guided us to define an algorithm, called *functional distribution algorithm*, outlined in Figure 2. It allows us to significantly increase the frequency of changes that may trigger one of the enabled outgoing transitions, i.e., one of the outgoing transitions of the current active state(s) of the model: the algorithm selects the enabled transition that was the least triggered by the previous test case input values.

Input	A set of state diagrams S representing super-states of the (sub)system under study.
Output	An input value generated according to the functional distribution.
Step 1	Select a super-state $S_0 \in S$ with the probability T_0 / T, where T_0 is the number of transitions in S_0 and in its potential composite states, and T is the total of transitions in all the state diagrams of the (sub)system.
Step 2	From all the transitions enabled in the chosen super-state and in its potential composite states, select the outgoing transition that has been the least triggered.
Step 3	For the selected transition, select randomly a variable implied in the event that triggers the transition.
Step 4	If the chosen variable is a Boolean, *change* its current value. Otherwise, generate a random input value within the domain of the variable to *change* its current value.

Figure 2. Functional distribution algorithm

For example, Table 3 presents a sample of the coverage measures we obtained. It gives the minimum and the maximum numbers of time every transition was triggered during the first 300 input values of: (1) five different sequences of input values generated according to the uniform distribution over the input domain and, (2) five different sequences of input values generated by the functional algorithm. These results show that in the latter case (functional distribution), every transition is triggered at least 3 times by each of the five test sequences. From Relation (2), it means that a test size of N = 300 corresponds to a test quality of $q_N \cong 0.95$ (Section 3.1). To reach the same test quality, the uniform distribution would require a much larger test size.

Table 3. Transition coverage measures: simulation results for 5 sequences of 300 input values generated according to each distribution (minimum & maximum activation numbers)

Transition Name	Uniform Distribution	Functional Distribution
ADS1	1 – 5	3 – 8
ADS2	2 – 5	3 – 5
ADS3	1 – 5	4 – 8
ADS4	1 – 3	3 – 6
AP1	3 – 8	16 – 20
AP2	1 – 4	4 – 6
AP3	0 – 2	4 – 6
AP4	0 – 2	5 – 10
AP5	0 – 4	3 – 5
AP6	0 – 2	3 – 5
AP7	0 – 2	3 – 5
AP8	2 – 5	11 – 15

However, the functional distribution does not provide a "balanced" coverage of the transitions. This is due to the fact that the activation conditions of some transitions are dependent on each other: some transitions are reachable only when others have been triggered. For example (see Figure 1), a single transition AP1 enters the ENGAGED state of the autopilot while three different transitions (AP2, AP3 and AP4) exit this state; as a result, AP1 will always be exercised more often than AP2 (or AP3 or AP4), whatever the input distribution.

Finally, it is worth noting that there is a difference between the number of input values (N = 300) and the number of triggered transitions (about 130 under the functional distribution). It is due to the fact that some events are not easy to produce and may request several successive input values to occur (see Section 4.2). Yet, input values that do not actually provoke the triggering of a modeled transition are not "garbage": when testing the program, they allow us to verify that it actually does nothing when it is supposed to do nothing.

5.2. Test Generation Process

To automize the test case generation in the Rose RealTime environment, we propose to extend the UML model representing the program under test with two capsules. These capsules, named *generator* and *collector*, are respectively responsible for the generation of the input values and the collection of transition coverage measures (Figure 3). The functional distribution algorithm is implemented in the generator capsule. The collector capsule supplies the generator capsule with the coverage measures needed at step 2 of the functional algorithm, to identify and select the least triggered enabled transition.

In addition, the transition coverage measures guide the empirical determination of the test size: given a test quality objective, Relation (2) gives the minimum number of times *n* each transition has to be triggered: hence, the generator capsule may stop the production of new input values as soon as the predefined value *n* is reached. Then, two files are automatically created. One file contains the expected outputs produced by simulation of the model with the input values. Another file represents a Java class that contains the input values translated into Java instructions. This class is a test driver especially created for managing the test process of the Java program. During the test experiments, outputs produced by the Java program are automatically compared to the expected outputs provided by the model.

Returning to the subsystem on which we exemplify our testing technique, the 12 transitions may be triggered according to values of input variables and internal variables. An internal variable is any function of input variables, active states or other internal variables. The test generation process consists of selecting values of input variables according to the algorithm. Internal variables are not directly controlled by the generated test cases but via input variables.

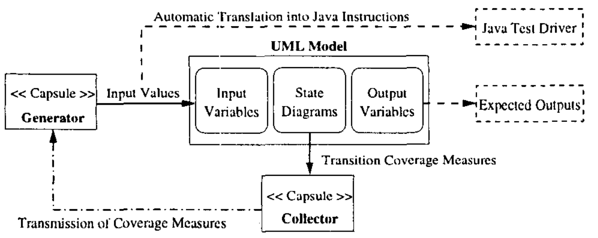

Figure 3. Test generation process

5.3. Results of First Test Experiments

Test experiments have been conducted on the Java program to get a hint on the adequacy of our test cases. In these experiments, we used ten different sequences of 500 input values: five of them were generated according to the functional distribution algorithm, and the other five according to a uniform distribution over the input domain, for purposes of comparison.

First, running the 10 sequences revealed no fault in the Java implementation, which always supplied output values identical to the expected ones provided by the UML model. This is in conformity with the test experiments conducted on the same FGS case study by Offutt [18]: no faults were found in the subsystem investigated.

Then, mutation analysis has been used to get an insight into the error detection power of the test cases. Mutation analysis consists in creating a sample of faulty programs from the program under test, by seeding it with faults called mutations. Mutations are single-point, syntactically correct changes introduced one by one in the original program. A mutant is then a copy of the original program containing one mutation. If test cases cause a mutant to produce different outputs than the expected ones, the mutant is said *killed* by the test cases. The number of mutants killed by a set of test cases may give an empirical assessment of their effectiveness. In [18], Offutt evaluates his testing technique on the overall FGS case study with a set of 155 mutants. With in mind to compare both techniques (in future work), Offutt provided us with his mutants. 46 of these mutants are related to the subsystem we consider. This sample of mutants is small and non-exhaustive, but it gives us a first hint on the adequacy of our test cases.

Table 4 summarizes the results of the experiments. For each input distribution, it shows the number of mutants *killed by each of the 5 sequences* of input values, as a function of the test size. The results are in favor of a best efficiency of the test sequences generated according to the functional distribution algorithm. Also, one can note that the number of killed mutants stabilize around N = 300, which suggests that requiring a test quality objective of $q_N \cong 0.95$ with respect to the transition coverage criterion could be a reasonable stopping rule to tune the test size (to be confirmed by further experiments).

Table 4. Number of mutants killed by each of the five sequences of input values

Test Size	Uniform Distribution	Functional Distribution
100	16	31
200	28	41
300	31	41
400	32	41
500	32	41

The 5 mutants that are not repeatedly killed under the functional distribution have been analyzed. Analysis results show that these 5 mutants deal with constant values, that is a constant value is replaced by another value. For instance, a mutant changes the correct statement "c > 0.813" with the faulty statement "c > 0.815" where c is defined in the range [0.0-1.0]. This mutant can only be killed if the random value generated for c falls within the small sub-domain]0.813-0.815]. This type of faults highlights the already known weakness of statistical testing with respect to faults related to extremal/special values. Previous work reported on the importance of using additional (deterministic) test cases specifically aimed to reveal this type of faults [25].

6. Conclusion and Future Directions

The paper presents a statistical testing technique for the generation of test cases from UML specifications. The approach represents a challenge for the testing of complex critical systems. This first investigation is concentrated on the automation process of the technique with the aid of the Rational Software Corporation's Rose RealTime tool. An automated test generation process is proposed that produces sequences of test cases (including both the input values and the expected outputs) for Java programs. The testing criterion used to guide the selection of input values is the coverage of the transitions of the UML state diagrams. Based on this criterion, we present a generic algorithm for the probabilistic (random) generation of input values that allows us to produce sequences of test cases that trigger several times every transition.

Returning to the related work outlined in Section 2, it is worth noting that transition coverage of the state diagrams is not a very stringent requirement: it is the least demanding of the four testing criteria defined in [19]; and, unlike [3, 12] we do not flatten the state diagrams to use testing criteria related to the coverage of expanded behavioral models. According to the motivation of statistical testing, here the principle consists of: (1) the use of models and testing criteria of reasonable complexity to guide the selection of input values in order to facilitate the definition of a test generation algorithm (and the analysis required to implement the algorithm) and, (2) the requirement of exercising several times – using different input values – every element identified by the criterion to compensate for the lack of detailed behavioral information supported by the models (and a fortiori by the testing criterion). On the opposite, improvements of the deterministic testing techniques are based on refinements of the behavioral models (e.g., using flattening transformation) and/or of the testing criteria (e.g., transition-pair coverage instead of transition coverage). But, the cost of such improvements is an additional complexity of the analysis required to create the test case input values, which may become impossible to automatize.

The feasibility of our approach is exemplified on a subsystem of a research version of an avionics system, the FGS case study. Results of first test experiments are in favor of a high effectiveness of the generated test cases. However, further extensive experiments must be conducted to get a more valuable assessment of the testing technique (including the stopping rule to tune the test size), and to compare it with other (deterministic) approaches, in terms of cost/effectiveness.

Ongoing work is concentrated on such experiments. They will be conducted on the whole FGS case study. Mutation analysis based on larger samples of seeded faults will be used to get a more reliable feedback on the efficiency of different techniques. In particular, it will involve mutation faults defined to target object-oriented concepts (such as those described in [13]). This future work will allow us to analyze the strengths and weaknesses of statistical functional testing applied to OO programs. Then, we will investigate the possibility of extending our framework with structural testing techniques to target the specificities of the Java programming language.

Acknowledgements

We are grateful to David E. Statezni of Rockwell-Collins, Inc. for his advice and review comments on earlier versions of this paper. We also thank Jeff Offutt of George Mason University for having supplied us with the mutants involved in our test experiments.

References

[1] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, Second Edition, 1990.

[2] R.V. Binder. Testing Object-Oriented Software: a Survey. *Journal of Software Testing, Verification & Reliability,* 6:125-252, 1996.

[3] R.V. Binder. *Testing Object-Oriented Systems.* Addison-Wesley, 1999.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Object Technology Series. Addison Weysley Longman, Inc., October 1998.

[5] T.S. Chow. Testing Software Design Modeled by Finite State Machines. *IEEE Transactions on Software Engineering,* SE-4(3):178-186, 1978.

[6] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development – The Fusion Method.* Object-Oriented Series, Prentice Hall, 1994.

[7] J.W. Duran and S.C. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438-444, 1984.

[8] P.G. Frankl and E.J. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE Transactions on Software Engineering*, 14(10):1483-1498, 1988.

[9] P. Fröhlich and J. Link. Automated Test Case Generation from Dynamic Models. In *Proceedings of the Fourteenth European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, Springer, pages 472-491, Sophia Antipolis and Cannes, France, June 2000.

[10] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231-274, 1987.

[11] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-Based Integration Testing. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 60-70, Portland, Oregon, USA, August 2000.

[12] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test Cases Generation from UML State Diagrams. *IEE Proceeding-Software*, 146(4):187-192, 1999.

[13] S. Kim, J.A. Clark, and J.A. McDermid. Class Mutation: Mutation Testing for Object-Oriented Programs. In *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, Erfurt, Germany, October 2000.

[14] D. Kung, P. Hsia, J. Gao (eds). *Testing Object-Oriented Software*. IEEE Computer Society, 1998.

[15] J.D. McGregor. An overview of testing. *Journal of Object-Oriented Programming*, 9(8):5-9, 1997 (first issue of a monthly column in the Journal).

[16] S.P. Miller. Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR. In *Proceedings of the Second Workshop on Formal Methods in Software Practice (FMSP'98)*, Clearwater Beach, Florida, USA, March 1998.

[17] J. Musa. *Software Reliability Engineering*. McGraw-Hill, 1999.

[18] A.J. Offutt. Generating Test Data From Requirements/Specifications: Phase III Final Report. Technical Report, George Mason University, Fairfax, USA, November 1999.

[19] A.J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *Proceedings of the Second International Conference on the Unified Modeling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, Springer, pages 416-429, Fort Collins, USA, October 1999.

[20] Rational Software Corporation. *Rational Rose RealTime User's Guide*. March 1999.

[21] P. Runeson and B. Regnell. Derivation of an Integrated Operational Profile and Use Case Model. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE'98)*, pages 70-79, Paderborn, Germany, November 1998.

[22] J. Ryser and M. Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In *Proceedings of the Twelfth International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, December 1999.

[23] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. Technical Report, ObjecTime limited and Rational Software Corporation, March 1998.

[24] P. Thévenod-Fosse and H. Waeselynck. STATEMATE Applied to Statistical Software Testing. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 99-109, Cambridge, USA, June 1993.

[25] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. Software Statistical Testing. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, Springer, pages 253-272, 1995.

[26] H. Waeselynck and P. Thévenod-Fosse. A Case Study in Statistical Testing of Reusable Concurrent Objects. In *Proceedings of the Third European Dependable Computing Conference (EDCC-3)*, volume 1667 of *Lecture Notes in Computer Science*, Springer, pages 401-418, Prague, Czech Republic, September 1999.