STATEMATE APPLIED TO STATISTICAL SOFTWARE TESTING

Pascale THÉVENOD-FOSSE, Hélène WAESELYNCK

LAAS - CNRS
7, avenue du Colonel Roche
31077 TOULOUSE CEDEX, FRANCE

thevenod@laas.fr, waeselyn@laas.fr

ABSTRACT

This paper is concerned with the use of statistical testing as a verification technique for complex software. **Statistical testing** involves exercising a program with random inputs, the test profile and the number of generated inputs being determined according to criteria based on program structure or software functionality. In case of complex programs, the probabilistic generation must be based on a black box analysis, the adopted criteria being defined from **behavior models** deduced from the specification. The proposed approach refers to a hierarchical specification produced in the **STATEMATE®** environment. Its feasibility is exemplified on a **safety-critical module** from the nuclear field, and the efficiency in revealing **actual faults** is investigated through experiments involving two versions of the module.

® STATEMATE is a registered trademark of i-Logix, Inc.

I. INTRODUCTION

The dynamic verification of complex software requires an automatic procedure for the generation of test inputs *and* expected outputs. The paper addresses this issue by studying a pragmatic approach, namely *statistical testing* designed with the aid of a *Computer-Aided Software Engineering (CASE) tool:* STATEMATE. It is worth noting that the statistical approach investigated here is concerned with bug-finding, not reliability: it is based on an unusual definition of random testing originally proposed in [The 89].

Statistical testing involves exercising a program with inputs that are randomly generated according to a given distribution over the input domain, the key of its effectiveness being the derivation of a distribution that is appropriate to enhance the program failure probability. In particular, the generation of random test data based on a uniform distribution over the input domain, which is the probabilistic approach people usually refer to [Dur 84], is not expected to be an efficient way to design statistical testing experiments: revealing input data being unlikely to be uniformly distributed over the input domain, a uniform profile is not relevant to increase the failure probability. In other words, like any test data, the random ones have to be designed by using some model of the target program, whether structural or functional. Indeed, the statistical test sets are defined by two parameters:

(i) the **test profile**, or input distribution, from which the inputs are randomly drawn,

(ii) the **test size**, or equivalently the number of inputs (i.e. of program executions) that are generated;

and both parameters may be determined according to criteria related to software structure or its functionality, the latter solution being more feasible in case of complex programs.

It is increasingly common to use **CASE tools** for the specification and design of complex programs, and a natural issue concerns the use, *for the design of test data*, of the facilities they offer. Through the example of STATEMATE, we show that the benefit that can be derived from such tools is twofold: first, the graphical languages generally supported by the tools to specify expected behavior may serve as the basis for the definition of functional criteria and second, their prototyping facilities provide us with an automatic procedure to generate the correct test outputs.

Section II exposes the principle of statistical testing, together with previous experimental work supporting its expected efficiency. Then, *section III* focuses on the facilities offered by the tool STATEMATE; the method for designing functional statistical test sets from a STATEMATE specification being presented in *section IV*. The feasibility of the approach is exemplified in *section V* on a **safety-critical module** from the nuclear field; experiments in *section VI* confirm its adequacy with respect to actual faults.

II. STATISTICAL TESTING

This section focuses on both the motivation and the theoretical framework of the statistical testing approach previously defined by the authors. Then main conclusions drawn from first experimental investigations are recalled, since they originated the work reported in the next sections.

II.1. Statistical Testing: Why?

Test criteria take advantage of information on the program under test in order to provide guides for selecting test inputs. This information, get during the software development process, relates either to program structure or its functionality (see e.g. [Mye 79, How 87, Bei 90]). In both cases, any criterion specifies a set of elements to be exercised during testing. Given a criterion C, let S_C be the corresponding set of elements. (To comply with finite test sets, S_C must contain a finite number of elements that can be exercised by at least one input item.) For example, the structural criterion "All-Branches" requires that each program branch be executed: C = "Branches" $\Rightarrow S_C$ = {executable program edges}.

Given a criterion C, the usual practice for determining a test set proceeds according to the **deterministic principle**: the tester selects a priori a set of N inputs such that each element of S_C is exercised at least once; and this set is most often built so that each element is exercised *only once*, in order to minimize the test size N. Unfortunately, an acute question still arises from the definition of test criteria: a real **limitation** is due to the imperfect connection of the criteria with the actual faults and, because of the (current) lack of an accurate model for software design faults, this problem is not likely to be solved soon. Hence, exercising only once, or very few times, each element defined by such imperfect criteria is far from being enough to ensure that the corresponding test set possesses a high fault exposure power. And this is the main reason why the efficiency of deterministic testing depends more on the particular test input values chosen than on the criterion retained [Ham 89].

To make an attempt at improving current testing techniques, **two different directions** may be investigated:

(i) one can *search for more pertinent test criteria* that is, criteria under which, for any faulty program, exercising once each element is very likely to produce a failure, whatever the particular input values chosen [Ham 90, Fra 91, Mar 91, Wey 91];

(ii) one can cope with current – imperfect – criteria and compensate their weakness by requiring that *each element be exercised several times.*

The *first direction* is arduous since it can be expected that a pertinent criterion depends thoroughly on the residual faults; these faults being specific to the target program, and unknown before testing. For example, as regards current structural criteria, the most stringent criterion, namely "All-Paths", is not pertinent: residual faults tracked down during testing are generally more subtle in the sense that there revealing input data correspond to a small subset of the input subdomain that exercises a given path. Since "All-Paths" testing is feasible for simple programs only, any more refinement should be unrealistic.

The *second direction* involves larger test sets that should be tedious to determine manually; hence the need for an automatic generation of test sets. And this is the **motivation of statistical testing** designed **according to a criterion**, which aims to combine the information provided by imperfect (but not irrelevant) criteria with a practical way of producing large test sets, that is, a random generation.

II.2. Statistical Testing: How?

When using the probabilistic method for generating test data, the number of times each element k of S_C is exercised is a random variable. **Two factors** play a part in ensuring that *on average* each k is exercised several times, whatever the particular test set generated according to the test profile and within a moderate test duration.

The first factor is the **input probability distribution** which must allow us to increase the probability of exercising the least likely element of S_C. Two different ways of deriving such a proper test profile are possible: either *analytical*, or *empirical*. The first way supposes that the activation conditions of the elements can be expressed as function of the input parameters: then their probabilities of occurrence are function of the input probabilities, facilitating the derivation of a profile that maximises the frequency of the least likely element. The second way consists in instrumenting the software in order to collect statistics on the numbers of activations of the elements: starting from a large number of input data drawn from an initial distribution (e.g. the uniform one), the test profile is progressively refined until the frequency of each element is deemed sufficiently high.

The second factor is the **test size** N which must be large enough to ensure that the least likely element is exercised several times under the test profile derived. The notion of test quality with respect to a criterion recalled below [The 89] provides us with a theoretical framework to assess a test size.

DEFINITION. A criterion C is covered with a probability q_N if each element of S_C has a probability of at least q_N of being exercised during N executions with random inputs. q_N is **the test quality with respect to (wrt) C.**

The quality q_N is a measure of the test coverage wrt C. Let p_k be the probability that a random input exercises the element k under the test profile retained, and $P_C = \min\{p_k, k \in S_C\}$ be the occurrence probability per execution of the least likely element. Then the test quality and the test size N are linked by the relation: $(1-P_C)^N = 1-q_N$. The result of this is that on average each element is exercised several times. More precisely, this relation establishes a link between q_N and the expected number of times, denoted n, the least likely element is exercised: $n \cong -\ln(1-q_N)$. For example, $n \cong 7$ for $q_N = 0.999$, and $n \cong 9$ for $q_N = 0.9999$. Thus, knowing the value of P_C for the test profile derived, if a test quality objective q_N is required the minimum test size amounts to:

$$N \geq \log(1-q_N) / \log(1-P_C) \qquad (1)$$

Based on this, **the principle of the method for designing a statistical test set according to a given criterion C** involves two steps, the first of which being the corner stone of the method. These steps are the following:

(i) *search for an input distribution* which is well-suited to rapidly exercise each element of S_C in order to decrease the test size; or equivalently, the distribution must accommodate the highest possible P_C value;

(ii) *assessment of the test size N* required to reach a target test quality q_N wrt C, given the value of P_C inferred from the first step; relation (1) yielding the test size.

Going back to the imperfect connection of the criteria with the actual faults, it is worth noting that the criterion does not influence random data generation in the same way as in the deterministic approach: it serves as a guide for defining an input profile and a test size, but does not allow for the a priori selection of a (small) subset of input data items. Indeed, the efficiency of the probabilistic approach relies on a single assumption: the information supplied by the criterion retained is relevant to derive a test profile that enhances the program failure probability. And there is a direct link between fault exposure power and random data: from relation (1), *any fault involving a failure probability $p \geq P_C$ per execution according to the test profile has a probability of at least q_N of being revealed by a set of N random inputs*[1]. No such link is foreseeable as regards deterministic test data; and this link should carry more weight than a thorough deterministic selection (thus providing in essence a *perfect* coverage) with respect to *questionable* criteria. This assumption has already been supported by previous experimental work, whose conclusions are recalled below.

II.3. On the Efficiency of Statistical Testing

In [The 91a-c], several current path selection criteria [Rap 85, Nta 88] were used to design **structural statistical test sets** for **four programs from the nuclear field**. For each criterion (from "Instructions" to "All-Paths") and each program, a proper test profile and a test size (for q_N = 0.9999) were defined in accordance with the two steps of the method recalled above. Several test sets were generated based on each profile, and their efficiency was compared to the one of (i) deterministic test sets determined according to the same criteria and, (ii) uniform sets, that is, conventional random test sets generated according to a uniform profile. For this, mutation analysis was used: 2816 mutation faults [DeM 78] were seeded one by one in the source codes, and the efficiency of the sets was assessed in terms of percentage of faults revealed, called mutation score.

Indeed, there is no evidence that simple-order mutations correspond to many faults in software; and this could deny the representativeness of mutation score with respect to the actual fault exposure power of a test set. But some faulty behaviors observed were typical of "subtle" errors produced by real faults, involving in particular [The 91c]: (i) faults turning a combinational function into a sequential one and, (ii) faults producing intermittent failures under unforeseeable conditions. In both cases, the outcome of an execution does not solely depend on the current input supplied: **the exhaustive testing of all input values would not yield a correctness proof** of the program. The production of such subtle errors leads us to conclude that mutation score is not a meaningless metric to compare the relative efficiency of various test sets.

The results confirmed the fact that the effectiveness of deterministic testing and uniform testing depends heavily on the particular input values chosen [Ham 89]. On the contrary, the mutation scores provided by the structural statistical test sets were repeatedly observed, whatever the particular set generated according to a same structural test profile; and these were high, **99.8% of the faults being uncovered**. Both structural deterministic data and uniform random data were far from reaching such a score, failing to reveal several hundreds of mutations. Hence, three main conclusions arose:

(i) the comparison between the uniform and structural statistical sets showed that *the structural analysis did provide a relevant information* to increase the failure probability, even as regards subtle errors loosely connected with the criteria; although derived from the same criteria, the deterministic sets involved too few data to compensate for this imperfect connection;

(ii) the impact of the criteria stringency was deemed not critical in the case of statistical testing; indeed, the *most cost-effective approach* was to retain weak criteria (Instructions, Branches) facilitating the search for a test profile, and to require a high test quality (say, 0.9999) wrt them;

(iii) the efficacy of a *mixed test strategy* combining structural statistical testing and deterministic testing of extremal input values was confirmed: the six mutations not uncovered after completion of the structural statistical tests were typical cases for the latter testing approach.

Following these promising results on the probabilistic generation of test sets, emphasis is placed on testing of larger programs, whose structural complexity prohibits the use of white box criteria. **Functional statistical testing** is investigated, based on criteria related to behavior models of the specification. As a first step [The 92], *finite-state machines* and *decision tables* were used for describing software behavior. The feasibility of the test design approach was exemplified by a real case study: the *safety-critical module from the nuclear field* which encompasses the four programs previously experimented on. The results suggested that functional statistical testing designed from behavior models is efficient: it allowed us to distinguish the important features of the module early in the development process, while still providing a good coverage of the implemented code. Hence, the investigation reported in the following sections focuses on the refinement of the approach, using a CASE tool as an help to the designer of functional test sets; and STATEMATE [Har 90] has been used because of its interesting facilities discussed below.

[1] Since the test profile is derived from the criterion retained, it may have little connection with actual usage (i.e. operational profile): the focus is bug-finding, not reliability assessment.

III. CASE TOOLS FROM A TEST DESIGNER PERSPECTIVE

Software development environments provide technical support for requirement analysis and design, through CASE tools. Such tools may be of great interest from a test designer perspective as they address two main problems related to the generation of functional test inputs for complex software:

(i) functional testing requires a behavioral specification that is sufficiently formal to allow the definition of **coverage criteria**: CASE tools implement methods that force the derivation of such models during specification and design phases (SA/RT techniques);

(ii) generating a large number of test data requires an automatic solution to the **oracle problem**, namely that of how to determine the correctness of the outputs [Wey 82, Bro 92]: CASE tools offering support for an executable specification or prototyping facilities, provides us with an oracle.

III.1. The CASE Tool STATEMATE

STATEMATE [Har 90] is a tool for the specification and analysis of complex reactive systems. A STATEMATE specification involves a hierarchical modeling approach: each function may be decomposed into a group of sub-functions, whose joint behavior is described by a conceptual controller. Hence, each level in the hierarchy consists of two related views (Fig. 1): the functional one, and the behavioral one. An **Activity-chart** (Fig. 1.a) is similar to a conventional Data Flow Diagram: it displays the sub-functions identified (here, functions are called *activities*) and the information flows between them. The control activity associated to these sibling activities is specified by a **Statechart** (Fig. 1.b), which is the behavioral model supported by the tool.

The salient feature of STATEMATE is the emphasis put on the dynamic verification of the specification: the tool provides facilities to program the **model execution,** in interactive or batch mode, and to **instrument** the models in order to collect statistics during execution. For a given test set, output references may be easily obtained from the specification, as well as measures of model coverage.

Before investigating coverage criteria for a STATEMATE specification, let us recall the main features of Statecharts.

III.2. Specifying Behavior by Statecharts

Statecharts [Har 87a] are a graphical language defined by Harel in order to combine the visual appeal of graphics with rigorous mathematical semantics [Har 87b, Pnu 91]. They improve state diagrams by adding the notions of depth, orthogonality and broadcast communication.

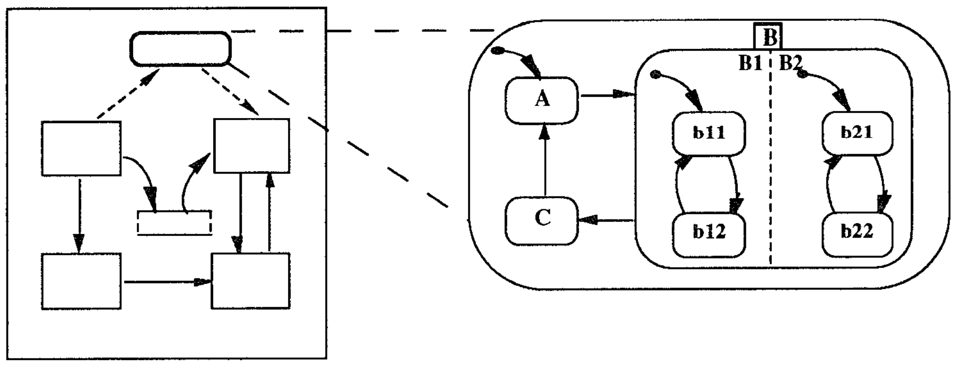
Depth gives the ability to cluster states with common transitions into a superstate. Alternatively, superstates can be defined before they are decomposed into substates, enabling us to adopt a top-down approach by successive refinements. The refinements may be of two types (Fig. 1.b):

(i) *XOR decomposition*, expressed by encapsulation; when we are in state *B1* at a high level of abstraction, we are really in either state *b11* or state *b12* at a lower level of abstraction.

(ii) *AND decomposition*, represented by splitting a state using dashed lines; when we are in state *B*, both **orthogonal** components *B1* and *B2* are active. This decomposition captures the notion of concurrency.

Hence, there are three types of states in a Statechart: **OR states** whose substates are related to each other by XOR, **AND states** having orthogonal components, and **basic states** having no offspring. The set of all active basic states in the Statechart constitutes the system configuration.

The semantics of Statecharts defines the sequence of system configurations taken in response to an external stimulus: starting from a *stable* configuration (no possible evolution unless an external event is generated), the system goes through a succession of *unstable* configurations until a stable configuration is reached. This chain reaction is achieved by a **broadcast mechanism**: any internal modification can be sensed from all active orthogonal components, so that further transitions may be triggered. The broadcasted informations include the states exited or entered, the actions performed when a transition is taken (generation of events, modification of data items...), the status modification of a controlled activity (started, suspended, resumed, stopped), etc.



(a) Activity-chart **(b) Statechart**

Fig.1. Functional (a) and behavioral (b) views of STATEMATE.

IV. DESIGN OF FUNCTIONAL STATISTICAL TESTING

The design of statistical test sets from a hierarchical decomposition of software functionality investigated in previous work [The 92], was based on a multi-level specification combining finite state machines and decision tables. We are now studying the adequacy of the approach wrt a STATEMATE specification.

IV.1. Managing the Complexity

Given a software module and its multi-level specification, the design of random test data must induce an analysis of reasonable complexity. In particular, the detailed analysis of the behavior models according to stringent criteria is presumably intractable, compelling us to use only weak criteria. It is worth noting that this constraint should lead to a less acute problem in the statistical approach than in the deterministic one; and all the more so as several test cases are involved per element to be exercised and as these test cases are unbiased by human choice. Indeed, previous work on *structural* statistical testing has already supported this assumption (section II.3).

Even if a weak criterion is adopted, it would not be realistic to attempt intensive coverage of all the behavior models at the same time. This is because of:

(i) *the module complexity*, which makes the assessment of the element probabilities difficult as numerous correlated factors are involved;

(ii) *the explosion of the test size;* even if these assessments are feasible and tractable in order to derive an input distribution, a prohibitive test size will probably be required to reach a high test quality, as the probability of the least likely element remains very low due to the large number of elements.

To address this issue several distinct test sets may be designed each one focusing on the coverage of a subset of models. To do this, one defines a **partition of the models** into disjoint subsets, each subset gathering one or several behavior models describing functions of the same or consecutive levels in the hierarchy. For each class of models, a specific input distribution can reasonably be derived, that maximises the probability of the least likely element related to the models grouped.

Retaining *weak criteria* and deriving *several input profiles* is the general approach that allows us to manage the complexity for a software module involving a multi-level specification, and to do so whatever the formalism adopted to depict behavioral aspects.

IV.2. Covering a Statechart

Due to the expressive power of Statecharts, even a behavioral description involving a small number of states may constitute a complex model. As a result, the criterion requiring the coverage of all states of the finite state machine equivalent to a given Statechart is expected to be too stringent in the general case: it would imply that every possible configuration, i.e. every possible *combination* of basic states in orthogonal components, be considered. Hence, a weaker criterion, namely the **coverage of the basic states**, should be more realistic.

In the present state of the art, it is not possible to draw from the analysis of the Statecharts the set of equations relating the state probabilities to the input profile. Given a class of models, the search for an adequate input distribution has thus to proceed **empirically** (§II.2): starting from an initial input distribution, e.g. the uniform one, generate several test sets of large size; execute the specification with the test data and count the number of times each basic state is entered; determine the activation conditions of the least exercised basic states in order to improve the input profile. This iterative process is stopped when the frequency of each basic state is deemed sufficiently high.

Since only a subset of the specification is taken into account to determine the input distribution specific to a given class of models, some input variables may not be involved and as a result no probability is obtained for them. Hence, the information deduced from other partition subsets must be included to define a complete input profile. To accomplish this, the inputs that are not involved at a given partition level can be classified according to three types:

(i) *upper level inputs*, conditioning the activation of the considered functions from the upper level Statecharts; their probabilities must provide the most likely activations;

(ii) *lower level inputs*, taken into account in lower level models; their probabilities are set as defined from the corresponding Statecharts;

(iii) *unrelated inputs*, for which a uniform distribution may be used.

As a result, the determination of the input distributions uses a **bottom-up** approach since, from (ii), the input distribution specific to a given level may be partly defined at lower levels. Previous experimental work has shown the feasibility and efficiency of this bottom-up process [The 92]; then, a proposed refinement was to stress the test of the **interactions** between high and low level functions.

IV.3. Studying the Interactions between two Statecharts of Consecutive Level

In a STATEMATE specification, two Statecharts *ctrl_A* and *ctrl_A1* belong to consecutive levels of hierarchy if *ctrl_A* controls the activity whose behavior is described by *ctrl_A1* (Fig. 2). The possible interactions between *ctrl_A* and *ctrl_A1* are the following:

(i) *ctrl_A* modifies the status of the activity *A1* (started, suspended, resumed, stopped);

(ii) *ctrl_A* passes data to *A1*, or conversely *A1* passes data to *ctrl_A* (generally, events for the purpose of synchronization);

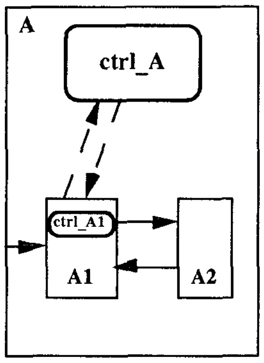


Fig.2. Two consecutive levels of functional decomposition.

(iii) *ctrl_A* manipulates the inputs/outputs of *A1*;

(iv) *ctrl_A* modifies the status of an activity *A2*, and there is a data flow between *A1* and *A2*.

In the case study reported in the next section, we focus on a particular interaction, namely the **initialization** of low level functions (interaction (i), when sub-activities are started). The verification of the initializations implies a suitable probe of the transient period that immediately follows this interaction. An adequate test profile has thus the property to provide likely initializations and to maximize the *conditional* probabilities: Prob [E_k entered/ interaction], {E_k} being the set of basic states in low level functions that may be entered after an initialization.

The verification of the interactions may be carried out apart from the verification of the various functions, through a special purpose test profile; in case the coverage of the upper level Statechart produces frequent initializations of the functions it controls, the verification may be carried out at the same time as the verification of high level functions, by adopting an adequate profile for the lower level parameters. This latter solution is the one we have adopted for the software module studied.

V. REAL CASE STUDY: A SAFETY CRITICAL APPLICATION

The STATEMATE specification has been derived from the high level requirements of the target module.

V.1. High Level Requirements of the Target Module

The module is extracted from a **nuclear reactor safety shutdown system.** It belongs to that part of the system which periodically scans the position of the reactor's control rods. At each operating cycle, 19 rod positions are processed. The information is read through five 32-bit interface cards. Cards 1 to 4 each deliver data about four rod positions; these cards are all created in the same way and are hereafter referred to as *generic* cards. The 5th card delivers data about the three remaining rod positions as well as monitoring data; this card which is therefore processed differently is called the *specific* card.

At each operating cycle, one or more interface card may be declared inoperational: the information it supplies is not taken into account. This corresponds to a *degenerated operating mode:* only part of the inputs are processed. A card identified as inoperational remains in that state until the next reset of the system. In the worst situation all cards are inoperational and the module delivers a *minimal service:* no measure is provided, only routine checks are carried out.

Extensive hardware self-checking is used so that errors when reading a card are unlikely. Nevertheless, for defensive programming concerns, this case is specified: the application is stopped and has to be restarted.

After acquisition, the data are checked and filtered. Three checks are carried out: the corresponding rod sensor is connected, the parity bit is correct and the data is stable (several identical values must be read before acceptance). The stringency of the third check (required number of identical values) depends on the outcome of the preceding checks of the same rod. After filtering, the measurements of the rod positions (in Gray code) are converted into a number of mechanical *steps.* The result of data conversion may be a valid number of mechanical steps or an invalid number or two special limit values.

V.2. STATEMATE Specification

The specification developed with the aid of STATEMATE defines **two levels of functional hierarchy**. The top level Activity-chart describes the information flow between:

- the external environment;
- five activities devoted to the processing of the five interface cards (acquisition, checks and filtering of the corresponding data);
- one activity devoted to the conversion of the 19 measures filtered;
- the Statechart that controls the six preceding activities.

At each operating cycle, all operational cards are processed in parallel; a general synchronization ensures that the conversion starts only after completion of this. When a card switches to inoperational, the corresponding activity is stopped. A read error puts a stop to all activities. The Statechart describing this behavior involves **33 basic states.**

The second level of hierarchy refines the behavior of each of the six activities identified. The processing of an interface card is described by a Statechart of **55 or 44 basic states,** depending on whether it is a generic card or the specific one; it is worth mentioning that the five Statecharts exhibit a regular structure: the processing of the 4 (resp. 3) measures acquired corresponds to the recurrence of identical patterns in orthogonal components. The conversion activity is described by a Statechart with **35 basic states.**

V.3. Determination of the Test Profiles

The design of statistical testing is based on a partition of the behavior models into two subsets, each grouping all models in the same level of hierarchy. Hence, two distinct input distributions must be derived: one to ensure the coverage of the card processing / conversion functions, and one to probe all operating modes. Both profiles are determined in compliance with the empirical process described previously (progressive refinement of the input distribution according to measures of coverage collected on the models).

For the coverage of **low level functions**, no distinction is made between analogous basic states in identical structural patterns; hence, analogous cases identified for the filtering checks and conversion of the 19 measures count for the coverage of the same element. With this simplification, only 22 measures of state coverage are to be collected on the whole models. The input parameters taken into account in this level are:

(i) the measurements of the rod positions acquired (valid or invalid number of mechanical steps, special values);

(ii) the corresponding parity bits;

(iii) the status of the related sensors (connected or not);

(iv) the monitoring data related to the specific card.

A first profile is determined for these parameters in order to ensure a good balance between the basic states, and it is completed by forcing the upper level parameters to their activating values: a full service is delivered, the five cards being operational with no read error. It is worth noting that, under this profile, successive test cases are not selected independently, since the coverage of some basic states requires that the value of a measure be identical to the one read at the preceding operating cycle for the same rod.

The determination of an adequate profile for the **top level controller** depends on the following input conditions:

(i) modification of the status of a card (from operational to inoperational);

(ii) occurrence of a read error;

(iii) occurrence of a reset.

Their probabilities are defined from statistics gathered on a subset of 17 basic states; the coverage of the remaining 16 states of the Statechart being implied by the coverage of the subset considered. Since frequent initializations of low level functions are caused by read errors or resets, their verification may be carried out at the same time as the verification of the top level controller. The test profile adopted for the lower level parameters depends on whether or not the generated input immediately follows an initialization:

- after an initialization, conditional probabilities of reachable basic states are maximized; the subset of basic states that may be entered after an initialization, and their conditions of activations, being determined from the (static) structural analysis of the six low-level Statecharts;
- then, until the next initialization, the probabilities of lower level parameters are the same as in the distribution defined for the coverage of the six Statecharts.

VI. EXPERIMENTAL RESULTS

The experiments involve two versions of the module. REAL is the real version, and STU a version developed by a student from the same high-level informal specification; both versions are written in C language. The size of their object code approximates 20 K-bytes (a thousand lines of source code without comments).

For each version, an experiment proceeds as follows:

(i) apply a test set to the program;

(ii) examine the first output result which differs from the one supplied by the STATEMATE specification;

(iii) identify and fix the corresponding fault(s).

The process is iterated to produce intermediate versions until REAL, STU and the executable specification agree on the whole test set.

VI.1. Overview of the Statistical Test Sets

The whole set of experiments performed on the module involves four types of statistical test sets. The first three types were designed and used in a previous work [The 92]; they are recalled here for comparison purposes. The latter one are the functional test data derived from the STATEMATE specification.

VI.1.1. Uniform Test Set

As the notion of random patterns is often connected to a uniform distribution over the input domain, a **uniform test set**, denoted **U-Set**, has been generated, the definition of the valid input domain being derived from the high level requirements of the module. The U-Set involves a large number of test data, namely 5300 inputs: this is in conformity with the foundation of uniform testing, that is, large test sets generated cheaply.

VI.1.2. Structural Test Sets

Structural statistical testing was shown to be highly efficient in a unit testing phase. But its relevance for larger scale programs may be questioned. One can wonder whether *weak* structural criteria (Instructions, Branches) are sufficient to distinguish relevant input cases for a target module involving the aggregation of several functions; more stringent criteria being no more tractable because of the complexity of the source code.

The test sets were derived from the structure of the **STU version**; few, if any faults being expected to reside in the

REAL one. The complexity of the source code forced us to choose the weakest criterion, namely **instruction testing**, and to proceed empirically to derive an input distribution. Starting from a large number of input data uniformly drawn from their valid range, we progressively refined the test profile until the frequency of each instruction was deemed sufficiently high (the C-compiler supports the automatic insertion of code to count the number of times each basic block of instructions is executed). The final input distribution was very different from the uniform one. A crude estimate of the probability of the least likely block was derived; given a test quality requirement of 0.9999, an upper bound N = 500 on the test size was drawn from relation (1).

Since it is pointless to define a testing method whose efficiency depends heavily on the particular input values selected, rather than on adequate properties of the test data related to the method, **five different structural test sets** denoted **S-Sets** have been generated in order to expose eventual disparities. Each S-Set is composed of 500 inputs, and it has been verified a posteriori that it provides a good coverage of STU (14 executions on average for the least likely blocks).

VI.1.3. Functional Test Sets Derived from Conventional Behavior Models

The modeling approach used in [The 92] involved a hierarchical decomposition of software functionality based on usual behavior models, namely **Finite State Machines (FSM) and Decision Tables (DT)**. The test criteria retained were the coverage of FSM states and of DT rules. Three levels of decomposition were defined from the high level requirements of the target module. The high-level functions (i.e. the various operating modes) and their interactions were described by a FSM M_0 and then refined through two other models: a FSM M_1 which models the checks and filtering performed on one measure, and a DT M_2 which translates the conversion function. For complexity reasons (see §IV.1), two different input distributions were defined from the dynamic analysis of the models:

(i) a first test profile to ensure the rapid coverage of the low-level functions, i.e. of M_1 states and of M_2 rules; under this profile, 85 inputs were required for q_N = 0.9999;

(ii) a second profile to ensure the rapid coverage of the high-level functions (M_0 states), under which 356 inputs were required for q_N = 0.9999.

Hence, a functional statistical test set derived from the three models is composed of 441 inputs: 85 drawn from the first profile, followed by 356 drawn from the second one. **Five different functional test sets**, denoted **F-Sets**, have been experimented on.

VI.1.4. Functional Test Sets Derived from STATEMATE

As Stated in §V.3, two input profiles are involved for the generation of functional test sets from the STATEMATE specification. The derivation of a test size N faces us with the same problem as the derivation of these profiles: without the knowledge of the equations that govern the process of visiting states, the number of inputs drawn from each profile has to be determined empirically.

As a first attempt, we have chosen to take the same sizes as those assessed for the F-Sets, that is: 85 inputs drawn from the profile ensuring the coverage of low level functions, followed by 356 drawn from the profile adequate for the top level controller. **Five new functional test sets**, denoted **STM-Sets**, have been generated. It turns out that each of them ensures a test quality of at least 0.9999 wrt the coverage of basic states: as regards the low level functions the least likely states are activated at least 11 times by each set of 85 inputs; as regards the top level controller, each state is activated at least 17 times by each set of 356 inputs. Hence, it is not deemed necessary to increase the test sizes: as a result the comparison between the functional F-Sets and STM-Sets will involve the same number of data.

VI.2. Actual Faults Uncovered

Thirteen faults have been identified (Fig. 3) in which 12, denoted A, B, ..., L, were found in STU; the last one, Z, resides in our test driver that provides the interface between REAL and the files containing the test sets. Faults A, G and J are **structural faults**, directly linked to the coding of STU. Faults B to F, and I, result from the **lack of understanding of the filtering check requirements** by the student, this function being at the heart of the module. The others are **initialization faults**: either an improper initial value is assigned (K, L) or the initialization is missing (H, Z). The initialization faults are most subtle since their activation depends on the states that follow the wrong initialization. For example, revealing H requires that G be removed and that the specific card be inoperational immediately after a reset; as regards L, its revealing data depends on input conditions related to the first *five* acquisitions of rod positions after a reset. Finally, although Z resides in the test driver that we have developed for REAL, it has a ripple effect on the module: the simulation of hardware reset fails to restore the correct initial context of REAL.

A	wrong operator used in the processing of an output value
B, C, D, E, F, I	the filtering checks, as implemented, do not comply with the specification
G	wrong control flow when the specific card is inoperational
H	initialization missing for variables related to the specific card
J	a variable in a loop is initialized out of loop instead of at each iteration
K	wrong initial state for the filtering process
L	wrong initial state for the filtering process
Z	initialization missing for a variable of our test driver

Fig.3. List of the thirteen faults uncovered.

Figure 4 summarizes the results supplied by the various test sets. The succeeding sections provide the main comments and conclusions related to each type of statistical test sets, emphasis being placed on the STM-Sets (see [The 92] for additional comments on the first three types).

	A	B	C	D	E	F	G	H	I	J	K	L	Z
U-Set N = 5300	✓	—	—	✓	—	✓	✓	✓	—	—	—	—	—
S-Sets N = 500	✓	✓	✓	✓	✓	✓	2	1	✓	✓	✓	—	3
F-Sets N = 441	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—	4
STM-Sets N = 441	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Fig.4. Results supplied by the test sets.

✓	always revealed
i	revealed by i sets out of 5
—	not revealed

VI.3. Inadequacy of the Uniform Distribution

Uniform testing provides the **poorest results**, since it reveals only five of the thirteen faults identified. The U-Set poorly probes STU and REAL, from both a structural and a functional viewpoint. With respect to the structural coverage, four blocks of instructions of STU and one of REAL are never exercised; some are seldom executed (less than three times). As regards the functional coverage, some basic states in the low level Statecharts are seldom or never reached: most faults related to the filtering checks are not revealed.

In conclusion, a test data generation based on a uniform distribution is **definitely not efficient** to design a statistical test experiment. It is often argued that uniform testing gives a best return on investment than other approaches, since a large number of test cases can be generated cheaply. But, such data are unlikely to exhibit a good fault revealing power, so that little improvement is to be expected from a *realistic* increase of the test size. Here, the uniform set is an order of magnitude larger than the other sets: the five faults are found within the first 633 executions; the remaining 4667 executions being garbage.

VI.4. Weakness of Instruction Testing

Nine faults are repeatedly revealed by all the structural sets; other faults (G, H, and Z) are occasionally revealed, usually late in the test experiment. L is never revealed.

Some major module features are not identified by the structural profile: most of the time the system delivers a full service, so that poor coverage is provided as regards the high level functions (degeneration of the operating modes and read errors on the various interface cards). This accounts for the poor results for G and H, since G is linked to degenerated operating modes with the specific card inoperational, and the exposure of H requires that G be fixed. The case of Z is special, because the test sets have been designed to cover STU and Z corrupts REAL from its test driver. However, the S-Sets also provide a good structural coverage of REAL (14 executions on average for the least likely blocks, as for STU).

In conclusion, structural testing is particularly **well-suited in unit testing, but its effectiveness diminishes as the source code under test grows.** The functions supported by the software move away from the instruction level, while finer examination of the structure becomes intractable. Indeed the size of the module under test is a limit above which instruction testing itself is no longer tractable. Moreover, this module belongs to the broad class of reactive systems, and whether or not the static graph of control is a relevant model for such systems is debatable.

VI.5. Promising Features of the Functional F-Sets

The five F-Sets yield better results than the structural sets, despite the fact that they involve a smaller number of inputs (441 versus 500). Hence, the use of behavior models as guides for designing statistical test sets appears to be **meaningful wrt the faults.**

Every fault in STU exposed by some S-Set is repeatedly found by all the F-Sets. The exposure of Z is less accurate, since Z is not revealed by one functional test set, but the result is still better than the one observed for the structural sets. The fault Z is linked to the initialization of the filtering checks: its exposure depends on conditions involving states of both FSM M_0 (describing the operating modes) and FSM M_1 (describing the filtering checks). This justifies the proposed refinement of the approach: stress the test of the interactions between the various levels of decomposition, emphasis being placed on the study of the initialization of low level functions from the high level controller.

VI.6. Efficiency of the Functional STM-Sets

Although they have been derived empirically, the STM-Sets exhibit a high fault revealing power. The twelve faults already identified from the previous experiments (A, ..., K and Z) are repeatedly found by all the sets generated. As regards Z, this result constitutes an improvement. The **two input distributions involved in the STM-Sets exhibit complementary features**: the first subsets of 85 test data reveal the faults related to the bad processing of the measures, notably during the filtering checks, while the second subsets are more appropriate for faults related to the failure of the specific card and for initialization faults.

A new fault, L, is uncovered in the STU program: it corresponds to the assignment of an improper initial value to a variable related to the filtering checks. This fault is rather subtle, because its exposure depends on specific conditions involving *five* successive acquisitions of rod positions after a reset: none of the previous structural and functional test sets fulfilled the required conditions. The test profile derived for the verification of the initializations ensures the coverage of basic states reached during the *first* acquisition; in spite of this, it allows us to raise significantly the failure probability of STU: the five subsets of 356 inputs generated according to this profile expose L, the first failure being observed quite before the end of the test experiments (15th execution in the best case, and 267th in the worst case). This result confirms once again the **efficiency of statistical testing wrt faults that are loosely connected with the criteria retained.**

Finally, the STATEMATE specification has enabled us to identify cases that were poorly covered by the previous F-Sets: the execution of the instrumented models with the five F-Sets shows that some basic states are seldom activated (only twice in the worst case); the reason being that the FSMs and DT derived *manually* from the high level requirements did not specify the corresponding cases. Although no related fault has been uncovered, this argues in favor of the use of a formal, complete specification for the design of functional test data. And for complex software, the development and maintenance of such a specification requires the aid of a CASE tool.

VII. CONCLUSION

For complex software, and in the present state of the art, a testing approach allowing us to generate automatically test sets that exhibit repeatedly high fault revealing powers within reasonable testing times is still a challenge. The work reported here is an attempt to provide a solution to this issue, using **random test data** generated according to distributions over the input domain that are appropriate to **enhance the failure probability** of the program under test. Due to the lack of an accurate fault model, statistical testing could be a more pragmatic approach than the search for "perfect" criteria, and previous work has already confirmed its promising features [Wae 93].

A new trend in software engineering is the emergence of CASE tools that assist the development by:

(i) supporting formal graphical languages for the specification of behavior;

(ii) offering facilities to computerized simulation, thus enabling to execute and evaluate the models of the software.

To our mind, such tools may constitute a valuable aid to implement functional statistical testing for complex software: they provide us with an **oracle**; the possibility of instrumenting the models facilitates the **derivation of test profiles** proper to ensure a rapid coverage of the software functionalities.

In this paper, the benefit of using CASE tools for the design of random data has been exemplified on the **STATEMATE environment**. The experimental results supplied by the real case study reported show the feasibility of the method on a non-trivial application, and support the efficiency of the functional statistical test sets thus designed.

This justifies further research work and two main directions will be investigated in the near future. First, the design of statistical testing from a STATEMATE specification has still to be formalized, since both the test profiles and the test sizes were determined empirically in the case reported. For this, we plan to study how to perform the probabilistic analysis of a Statechart, possibly by considering a restrictive subset of the syntax. Then, the efficiency of statistical test sets thus designed will have to be supported by other experiments performed on various programs: the fault-revealing power will be assessed with respect to actual faults, as well as with respect to mutation faults that will provide us a larger sample of (seeded) faults.

ACKNOWLEDGMENTS

This work was supported in part by the CEC under ESPRIT Basic Research Project no. 6362: "Predictably Dependable Computing Systems (PDCS2)". We wish to thank a lot Pierre BAUMGARTNER for his useful contribution to the STATEMATE specification of the module, within the framework of a student project.

REFERENCES

[Bei 90] B. Beizer, *Software testing techniques*, Van Nostrand Reinhold, New York, Second Edition, 1990.

[Bro 92] D. B. Brown et al, "An automated oracle for software testing", *IEEE Trans. on Reliability*, Vol. 41, no. 2, pp. 272-280, June 1992.

[DeM 78] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *IEEE Computer Magazine*, vol. 11, no. 4, pp. 34-41, April 1978.

[Dur 84] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438-444, July 1984.

[Fra 91] P.G. Frankl and E.J. Weyuker, "Assessing the Fault-Detecting Ability of Testing Methods," *ACM Software Engineering Notes*, vol. 16, no. 5, December 1991.

[Ham 89] R. Hamlet, "Theoretical comparison of testing methods," in *Proc. 3rd IEEE Symposium on Software Testing, Analysis and Verification (TAV-3)*, Key West, USA, pp. 28-37, December 1989.

[Ham 90] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. on Software Engineering*, vol. 16, no. 12, pp. 1402-1411, December 1990.

[Har 87a] D. Harel, "Statecharts : a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274, 1987.

[Har 87b] D. Harel et al., "On the formal semantics of Statecharts," *Proc. 2nd IEEE Symposium on Logic in Computer Science*, IEEE Press, NY, USA, pp. 54-64, 1987.

[Har 90] D. Harel et al, "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Trans. on Software Engineering*, vol. SE-16, no. 4, pp. 403-414, April 1990.

[How 87] W. E. Howden, *Functional program testing and analysis*, Computer Science Series, McGraw-Hill Book Company, 1987.

[Mar 91] B. Marre, "Toward automatic test data set selection using algebraic specifications and logic programming," *Proc. 8th Int. Conference on Logic Programming*, Logic Programming M.I.T. Press, Paris, France, pp. 202-219, 1991.

[Mye 79] G. J. Myers, *The art of software testing*, Wiley, New York, 1979.

[Nta 88] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Trans. on Software Engineering*, vol. SE-14, no. 6, pp. 868-874, June 1988.

[Pnu 91] A. Pnueli and M. Shalev, "What is in a step: on the semantics of Statecharts," *Proc. TACS '91*, Sendai, Japan, September 1991.

[Rap 85] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. on Software Engineering*, vol. SE-11, no. 4, pp. 367-375, April 1985.

[The 89] P. Thévenod-Fosse, "Software validation by means of statistical testing: retrospect and future direction," *Preprints 1st IEEE Working Conference on Dependable Computing for Critical Applications (DCCA-1)*, Santa Barbara, USA, pp. 15-22, August 1989. Published in *Dependable Computing and Fault-Tolerant Systems*, vol. 4 (Eds. A. Avizienis, J-C. Laprie), Springer-Verlag, pp. 23-50, 1991.

[The 91a] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "An experimental study on software structural testing: deterministic versus random input generation," *Proc. 21st IEEE Symposium on Fault-Tolerant Computing (FTCS-21)*, Montréal, Canada, pp. 410-417, June 1991.

[The 91b] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *Journal of Software Testing, Verification and Reliability*, vol. 1, no. 2, pp. 5-25, July-September 1991.

[The 91c] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "Software structural testing: an evaluation of the efficiency of deterministic and random test data," LAAS Report 91.389, December 1991.

[The 92] P. Thévenod-Fosse and H. Waeselynck, "On functional statistical testing designed from software behavior models," *Preprints 3rd IEEE Working Conference on Dependable Computing for Critical Applications (DCCA-3)*, Palerme, Italy, pp. 3-12, September 1992.

[Wae 93] H. Waeselynck, "Vérification de logiciels critiques par le test statistique", Doctoral Thesis, Institut National Polytechnique de Toulouse, LAAS report no. 93.006, January 1993.

[Wey 82] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465-470, 1982.

[Wey 91] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. on Software Engineering*, vol. 17, no. 7, pp. 703-711, July 1991.