Towards a Statistical Approach to Testing Object-Oriented Programs

Pascale Thévenod-Fosse and Hélène Waeselynck

LAAS - CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 - FRANCE
E-mail: {thevenod, waeselyn}@laas.fr

Abstract

Statistical testing is based on a probabilistic generation of test data: structural or functional criteria serve as guides for defining an input profile and a test size. Previous work has confirmed the high fault revealing power of this approach for procedural programs; it is now investigated for object-oriented programs. A method for incremental statistical testing is defined at the cluster level, based on the class inheritance hierarchy. Starting from the root class of the program, descendant class(es) are gradually added and test data are designed for (i) structural testing of newly defined features and, (ii) regression testing of inherited features. The feasibility of the method is exemplified by a small case study (a Travel Agency) implemented in Eiffel.

1: Introduction

Object-oriented (OO) design corresponds to a recent approach to software construction, the software system being viewed as a collection of entities that interact with each other. OO languages provide powerful mechanisms like entity instantiation, inheritance, polymorphism, that have no equivalent in traditional procedural languages (C, Fortran, ...). Hence, proper verification approaches have to be developed that take into account these specificities.

Testing involves exercising the software by supplying it with input values. Since exhaustive testing is not tractable, the tester is faced with the problem of selecting a subset of the input domain that is well-suited for revealing the (unknown) faults. The selection is guided by *test criteria* that relate either to a model of the program structure or a model of its functionality, and that specify a set of elements to be exercised during testing [2]. For example, for procedural programs, the control flow graph is a classical structural model, and branch testing is an example of a criterion related to this model. In the case of OO programs, further work is needed to determine to what extent such classical criteria may offer useful guidance. This paper belongs to on-going research aimed at adapting structural techniques to the testing of OO programs.

Given a criterion, the methods for generating test inputs proceed according to one of two principles: either deterministic or probabilistic.

The deterministic principle consists in selecting a priori a set of test patterns such that each element 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. But a major *limitation* is due to the imperfect connection of the criteria with the real faults. Because of the lack of an accurate model for software design faults, this problem is not likely to be solved soon, even in the case of traditional procedural programs. Exercising only once, or very few times, each element defined by such imperfect criteria is not enough to ensure a high fault exposure power. Obviously, the problem is expected to get even worse in the case of OO programs.

To make an attempt at improving current testing techniques, one can cope with imperfect criteria and compensate their weakness by requiring that *each element be exercised several times*. This involves large sets of test patterns that may be tedious to derive manually; hence the need for an automatic generation of test data. This is the *motivation of statistical testing* designed according to a criterion: combine the information provided by imperfect criteria with a practical way of producing numerous input patterns, that is, a random generation. This approach should not be confused with blind random testing, which uses a uniform profile over the input domain. Previous work has shown the high fault revealing power of statistical testing for procedural programs (see e.g., [13, 14]). This paper investigates the feasibility of statistical structural testing for small OO subsystems (class cluster). Path selection techniques are combined with the consideration of one specific OO concept: inheritance. This concept should facilitate an incremental testing approach.

The paper is organized as follows. Section 2 recalls the theoretical framework of statistical testing. Section 3 discusses the new testing issues raised by the OO paradigm and reports on current work aiming to address them. Then, a method for designing incremental statistical testing is defined in Section 4. It is a structural class-based approach whose steps are determined according to the inheritance hierarchy. Its feasibility is exemplified by a small case study (A Travel Agency) implemented in Eiffel. Section 5 describes directions for further investigation.

2: Statistical testing

Given a criterion C, let S_c be the corresponding set of elements to be exercised during testing. To comply with finite test sets, S_c must contain a finite number of elements that can be exercised by at least one input pattern. For example, the structural criterion "All-Branches" requires that each program branch be executed: C = "Branches" $\Rightarrow S_c$ = {executable program edges}.

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 input values 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 a function of the input probabilities, facilitating the derivation of a profile that maximizes 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 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 input patterns. 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 pattern 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 equation: $(1-P_c)^N = 1-q_N$. The result of this is that on average each element is exercised several times. More precisely, this equation 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:

$$N_{min} = \ln(1-q_N) / \ln(1-P_c) \qquad (1)$$

The principle of the **method for designing statistical testing** 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:

1. *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;
2. *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) yields the minimum test size.

Going back to the imperfect connection of the criteria with the real 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 patterns. Indeed, the efficiency of the probabilistic approach relies mainly on the assumption that 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_{min} random patterns[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. The experimental results obtained for procedural programs (see e.g., [13, 14]) support this assumption.

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.

3: OO paradigm from a tester's viewpoint

As an in-depth description of the OO paradigm would fall beyond the scope of this paper, we assume a basic knowledge of the underlying concepts (see e.g., [16]). The terms of *object*, *class*, *inheritance* (which can be single, multiple, or repeated), *polymorphism* and *dynamic binding* are used with their usual meaning. We adopt the Eiffel [10] terminology to denote the *features* that can be invoked on one object: *attributes* represent data items associated with the object while *routines* represent computations (either procedures or functions) applicable to it.

We adopt the view of testers to discuss the OO concepts. There is now an agreement among the testing community that these concepts raise a number of problems.

3.1: Structure versus behavior

As already mentioned, test criteria are related to a model of either the program structure (white box model) or of its functionality (black box model).

OO design and implementation is typically used in the case of open, interactive software systems. The black box description of such systems is discussed in [17]. It is argued that the observable behavior of objects cannot be modeled by algorithms, Turing machines or traditional mathematics: the adequate model of computation is the *interaction machine*. Interaction machines extend Turing machines with input actions supporting interaction with an external environment during computation. The richer expressive power of interaction machines has a cost: complete specification of OO system behavior is impossible. Only a subset of all possible computations can be modeled, by providing partial description of interface behaviors. As will be shown below, the irreducibility of object behavior to that of algorithms also has dramatic consequences for structural models.

Structural analysis of procedural programs is usually based on the analysis of control flow graphs that may be supplemented with data flow information. The source code is divided into basic blocks, that is, sequences of instructions that cannot be interrupted. Then, the control flow graph is constructed by making each basic block a node and drawing an arc for each possible transfer of control from one basic block to another. Data-flow annotation may be added to mark the nodes where a variable is assigned a value and the ones where it is used. Such models are consistent with a view of programs acting as closed input/output processes: each input pattern predetermines the path in the graph to be executed; the output is produced as a result of the processing of basic blocks along the selected path. This emphasis on closed algorithms and their transformation effect is questionable in the case of OO programs.

Typically, the body of routines applicable to an object is of small size, but contains calls to routines of other objects, which in turn may invoke distant features. This phenomenon is described in [9]. Having analyzed classes of the InterViews library, the authors found 4,818 invocation chains, most of them involving between two to nine calls in sequence, the longest ones being of length fourteen. This complies with the interactive nature of objects: the structure responsible for the behavior of one routine is distributed among several objects[2]. Moreover, this structure is not restricted to algorithms. Objects carry a state that remembers and influences the effect of routines. The state space can be very large, because of the unfolding of attributes associated with the objects through the invocation chains; it is not a static dimension since objects can be created or deleted during computation. Hence, the structure of OO programs cannot be readily analyzed in terms of algorithmic composition and transfer of control. It is worth noting that this problem is not specific to the OO paradigm: conventional structural analysis would exhibit limitation for any design characterized by highly decentralized flow of control, extensive use of shared data structures, and dynamic allocation of memory, whether or not the implementation language is object-oriented. However, the facilities offered by OO languages further compound the problem.

The class is the basic modular unit into which the source code of an OO program is organized. But a class is a purely static notion: only *objects* can be executed (and tested). This results in the syntactic constructs having a less straightforward operational interpretation than in the case of traditional procedural languages. For example, an attribute declaration within the scope of a class is not equivalent to a variable declaration (several instances may be present at some point of the computation). Owing to polymorphism, the dynamic type of an object variable may be different from the one with which it is declared; hence, the invocation of an object routine is not the direct transfer of control induced by a procedure or function call (the adequate version of the routine has to be selected in accordance with the dynamic type of the object).

Another difficulty in analyzing source code arises through the incremental description of classes throughout the inheritance hierarchy. The code provided for every new class is an addendum to the parent classes, allowing the programmer to introduce new features, rename or redefine

2 But the concept of interaction also goes beyond the boundary of the software system. The processing may be partly delegated to other systems (hardware device, human operator, ...) not encompassed within the structural model.

the old ones, and modify their visibility. Without automated support for *class flattening*, tracing the available features to code fragments rapidly becomes cumbersome as the inheritance tree grows. The composition mechanisms underlying the inheritance relations can be very complex (especially in the case of multiple and repeated inheritance). We are back to the problem of understanding the program behavior in terms of its structure.

This problems has led some authors [8] to deny the accuracy of structural analysis except for testing individual routines in isolation. Their proposed grey box strategy stresses behavior over structure, but structural information is still needed to a certain extent. Moreover, as pointed out in [4], the misuse of some complex aspects of the OO languages (like inheritance) contributes to the introduction of faults. Hence, we believe that although traditional practices cannot be simply transferred to OO programs, structural analysis still has an important role to play in the design of test data.

3.2: Determination of testing levels

The traditional unit and integration levels of testing do not fit well in the case of OO programs. Unit testing of subprograms cannot be mapped onto testing of individual object's routines: taken in isolation, the body of one routine typically consists of a few lines of code; its behavior is meaningless unless analyzed in relation to other routines and their joint effect on a shared state. Hence, any significant unit to be tested cannot be smaller than the instantiation of one class.

A first problem may be due to the presence of classes that are not instantiable, namely abstract and generic classes. An abstract class possesses features that have no default implementation: only (concrete) descendants of the abstract class can be instantiated. Genericity is the definition of a class supporting type parametrisation: an actual parameter must be chosen at instantiation. In both cases, it is not possible to design test data for each possible choice of instantiation.

More acute is the general problem of the complex dependency between classes. As pointed out in [9], the many relationships that exist in an OO program (inheritance, association, dynamic object creation, feature invocation, polymorphism, dynamic binding, ...) imply that one class inevitably depends on another class. It is difficult to determine where to start testing, and there is no obvious order for an integration strategy. The determination of testing levels depends on two factors:

1. The possibility of associating a functional description with a (cluster of) class(es). The instantiation of the class(es) must form a subsystem that is sufficient to generate meaningful behavior patterns.
2. The number of stubs to be constructed. Every server class not included in the target piece of software has to be simulated, which increases the cost of testing.

These factors are not independent. The need for stubs is due to the open nature of objects that achieve a desirable behavior through interaction with an external environment. Minimizing the number of stubs amounts to embodying the interactions within the boundary of the subsystem, in order to approximate to closed behavior.

From what precedes, it must be understood that even a unit test is likely to involve more than one class, as will be exemplified by the case study in Section 4.

3.3: Reusability of test sets

The reuse of test sets may occur in two cases: regression testing after a modification, or testing of a new component that reuses an already tested piece of design.

Regression testing is based on the analysis of dependencies in code. It must be determined which tests to rerun after some modification has been introduced. As already said, many dependencies are likely to exist in an OO program, making this analysis very difficult [9]. The modification of one class may have a ripple effect throughout the OO program, and there is a risk of failing to reveal some faults owing to incomplete regression testing. Clearly, systematic approaches are needed in order to identify the impact of a modification. Such approaches would have to take into account all possible relationships between classes. For example, a problem that is unique to OO programming is the maintenance of an inheritance hierarchy when changes are made to parent classes.

On the other hand, since the inheritance mechanisms promote the reuse of class design, it is expected that testing information can be reused as well for the child class. However, the intuition can be misleading as was established by the theoretical study performed by Perry and Kaiser [11] who applied the adequacy axioms developed by Weyuker [18] to facilities offered by OO languages, and examined their consequences in terms of test requirements.

It is obvious that an inherited routine redefined by child class has to be retested. But Perry and Kaiser showed that the test data previously designed for the parent class may be no longer adequate with respect to either structural or functional criteria. The *antiextensionality axiom* tells us that two programs that compute the same function may have radically different implementations. The *general multiple change axiom* implies that small syntactic changes may have large semantic consequences. Hence the need to develop new test cases. The problem is all the more acute as the OO programming language does not force us to adopt a disciplined use of inheritance. Even unchanged routines require retesting. This is due to the

antidecomposition axiom, stating that adequately testing a program P (with respect to a criterion) does not imply that each component of P is adequately tested (with respect to the same criterion). The child class may provide new contexts of usage that were impossible in the original definition of the routine. Actually, if the design imposes no semantic constraints on the inheritance relation, the only case where retesting is not required is when there are no interactions between the new and inherited features.

A last result of [11] concerns regression testing. It may be thought that, thanks to encapsulation, changing the internal implementation of a class would not affect its clients provided that the interface is left the same. But integration testing is always needed: the *anticomposition axiom* reminds us that adequately testing all components of a program P does not imply that P is adequately tested.

To conclude, it is far from trivial to determine whether some piece of code needs retesting, and if so, to determine whether existing test data can be reused. Moreover, the analysis to be conducted will be different depending on the programming language: as pointed out in [12], the OO languages have a great variance in the facilities that they provide to the programmer. It must be added that common facilities may also have very different implementations. For example, the inheritance mechanisms supported by Eiffel are not the ones offered by C++.

3.4: Previous work on testing OO programs

The problems posed by OO programs are a growing concern of the testing community. We present below an overview of previous work on this area (see [4] for additional pointers). To our knowledge, none of the authors have investigated the issue of statistical testing yet.

Testing from formal specifications has been studied in the framework of algebraic abstract data types (ADT). This is not surprising since many OO concepts originate from ADTs: a class can be seen as the implementation of an ADT. In the ASTOOT approach [6], test cases consist of pairs of sequences of routine invocations that should put objects of the target class into the same abstract state. Each sequence in the pair is executed on a different object, and the paired objects are compared by means of a user-supplied routine that approximates an observational equivalence checker: if the check result is negative, a fault has been revealed. Thus ASTOOT is an approach to the testing of classes that form standalone units, like implementations of stacks and queues. The work initiated by [1] aims at dealing with larger subsystems involving several communicating entities. The specification is done in a language called CO-OPN/2 (Concurrent Object-Oriented Petri Nets) based on algebraic nets. First investigations have consisted in adapting the theory of ADT testing [3] to this new framework. Further work has to be conducted in order to obtain a practical testing strategy. To conclude on ADT testing approaches, a key advantage is that they provide an oracle procedure to determine the correctness of the test results, although there are unavoidably some limiting constraints due to observability problems.

The testing approach presented in [9] involves three complementary models obtained from the reverse engineering of source code:

- The object relation diagram (ORD) displays the inheritance, aggregation and association relationships between the classes;
- The block branch diagram (BBD) is a kind of control and data flow graph for each routine; it also provides information about the interface with other routines;
- The object state diagram (OSD) is a hierarchical, concurrent, communicating state machine that models the object state dependent behavior.

Based on these models, a global testing strategy is sketched. The ORD is used to determine a class test order according to the dependencies between the classes; the aim is to obtain an integration strategy that minimizes the number of test stubs. The BBDs are used to prepare conventional structural test data for the individual routines. The OSDs facilitate the sequential testing of object state behavior. The processing of models is partly supported by tools. The problem of regression testing is also addressed: a "firewall" tool automatically computes the class affected by a change; however, since this tool is based on static analysis, polymorphism and dynamic binding are not taken into account.

The work of [8] is focused on the identification of integration levels. Emphasis is put on behavioral, rather than structural, considerations: grey box analysis is used to identify the most significant paths and interactions in the program. Two intermediate levels are defined in addition to routine and system levels:

- Method/Message Paths, or MM-Paths for short. This is related to the notion of *message quiescence*: an MM-Path starts with a routine (or method, in their terminology) and ends when it reaches a routine which does not issue any message of its own;
- Atomic System Functions, or ASF for short. This is related to the notion of *event quiescence*: an ASF is an input port event, followed by a set of MM-Paths, and terminates by an output port event.

The integration proceeds by testing the MM-Paths first and then testing their interaction in an ASF.

In [7], an incremental testing strategy based on the inheritance hierarchy is proposed. Each class has a testing history involving three integration levels: standard unit testing of individual routines, intra-class integration testing of routines, and inter-class integration testing; at each

level, structural and functional test sets are distinguished. The concern is to have histories be incrementally updated so that many test sets of a child class can be derived from the ones of its parent class. The approach considers only single inheritance, and is oriented toward C++. Depending on the type of child feature (new / virtual_new, recursive / virtual_recursive, redefined / virtual_redefined), appropriate decision is taken at the various testing levels: design new test cases, incorporate parent test cases in the child testing history, rerun or not these incorporated test cases. The decision procedure is defined by explicitly referring to the work in [11] related to adequacy axioms (see Section 3.3).

A synthetic view of the problems addressed by the various authors is provided in Figure 1. For the purpose of comparison, we also show the specific contribution of this paper, to be detailed in the next section.

	Models of OO programs	Contribution to testing issues
[6]	algebraic ADT	Oracle problem: observational equivalence of paired test sequences.
[1]	algebraic nets: CO-OPN/2	Oracle problem: evaluation of ground temporal logic formula. Theoretical investigation of test data selection.
[9]	ORD, BBD, OSD from reverse engineering	Global integration strategy minimizing the number of stubs. Firewall analysis for regression testing.
[8]	MM-Path, ASF	Integration strategy based on behavioral dependence: message and event quiescence.
[7]	–	Reuse of test data according to the inheritance relation.
This paper	static & dynamic BON models	Statistical testing steps based on the inheritance relation (cluster level).

Fig. 1: Overview of the problems addressed by the various authors

4: Statistical testing based on inheritance hierarchy

As a first step, the work presented in this section has concentrated on the question of how to design statistical testing for clusters of directly instantiable classes (no parametrised or abstract classes). We assume two kinds of static relations: single inheritance and client relations.

4.1: Principle

Our goal is to define a compromise between two testing levels: class level (each class is studied in isolation), and cluster level (all the objects involved in a cluster are tested in a single step). The proposed strategy is made up of several testing steps, each one focusing on a subset of classes. Such a progressive testing process is aimed at (i) defining testing steps requiring no stub constructions, and (ii) facilitating the diagnosis which could become very tedious due to complex dependency between classes, in particular in case of polymorphism and dynamic binding. To define the strategy, two questions must be answered:

1. How to divide a cluster into subsets of classes which form meaningful subsystems without stubs, that is, how to define the testing steps?
2. How to design test data specific to each testing step?

4.1.1: Definition of the testing steps. The method for defining testing steps is based on the analysis of the *static dependencies* between classes which are due to both inheritance mechanism and client relations. Starting from the root class of the cluster (which may be a test driver), a subset denoted S_1 is defined: it groups the root class, its server classes and their parent classes. By construction, $SS_1 = S_1$ forms a meaningful subsystem without requiring stubs. It will be the focus of the **first testing step**.

Then, a descendant of a class of SS_1 is selected to define the classes grouped in a new subset S_2 made up of: the selected descendant class, its server classes and all the parent classes which do not belong to SS_1. By construction, $SS_2 = SS_1 \cup S_2$ forms a meaningful subsystem without requiring stubs. It will be used in the **second step** which will focus on testing objects of the new S_2 classes.

The process goes on in a similar way by integrating progressively the descendant classes. Let SS_i denote the subsystem formed by the union of the successive sets S_1, …, S_i: the set S_{i+1} contains a descendant class of an element of SS_i, its server classes and the parent classes that do not belong to SS_i. The subsystem $SS_{i+1} = SS_i \cup S_{i+1}$ will be used in **step (i+1)** focused on testing objects of S_{i+1} classes.

4.1.2: Design of statistical test patterns. In our mind, structural analysis of OO program has an important role to play in the design of test data (Section 3.1). At the cluster level, we recommend that one takes advantage of structural information to define statistical test profiles, as long as the analysis of execution paths is not prohibited by the class dependency complexity. We are conscious that every cluster is not simple enough to enable the structural analysis of execution paths; yet, we believe that the method may apply to a large number of clusters, namely those involving reasonable numbers of calls in sequence (say, less than 8) with typical class routines of small size.

Since the testing method involves several test steps, *structural analysis* is also conducted incrementally on the

expanded control flow graphs associated to the subsystems SS_i. Starting from the control flow graph of the creation routine of the root class, these graphs are built as follows.

Step 1. The nodes of the control graph of the creation routine of the root class may contain a call to a local routine of the root class, or to a distant routine of a class of the set S_1. Then, node expansion is performed, which consists in replacing the calls by partial control flow graphs of the corresponding routines: here, partial means that the only paths that are taken into account are those corresponding to the execution of SS_1 objects. Paths involving objects not in SS_1 classes are ignored because their activation is delayed until the inclusion of the classes in a subsystem SS_i, i>1. The partial graphs thus added may, in turn, contain routine calls and the expansion node process goes on. The control graph G_1 of SS_1 is complete when all call nodes have been expanded.

Step (i+1). Starting from G_i, the control graph G_{i+1} of SS_{i+1} is got by the addition of new execution paths. Partial control graphs obtained at step i are completed with the paths containing calls to routines of the S_{i+1} classes. The added parts may contain routine calls and the expansion node process goes on by taking into account all the calls to routines of any of the SS_{i+1} classes.

Based on the graphs G_i, the coverage of the execution paths provides us with a test criterion that ensures a sound probe of the program structure. At each step i, the *method for designing statistical test patterns* (Section 2) may be applied using S_{ci} = {execution paths of G_i} as the set of elements to be activated during testing. Nevertheless, such test patterns should be quite redundant from one step to the following ones, since $G_i \supset G_{i-1}$. Then, an extreme solution could be to execute only the new execution paths at each step i, that is, to use S_{ci} = {execution paths of G_i that do not exist in G_{i-1}}. But this weak criterion could significantly reduce the test effectiveness since the execution paths of G_{i-1} will not be tested in the new subsystem SS_i: as was stated in Section 3.3, previous paths must be retested in the new context of execution.

Then, at each step i, the solution we propose is to take into account the whole set of execution paths of G_i, and use the notion of test quality q_N to require a less stringent coverage of the paths already executed in the previous steps. This can be done by setting different target test quality objectives, that is, for example:

- $q_N = 0.9999$ for the new paths that must be strongly tested, which means that each path will be executed 9 times on average;
- For the other (retested) paths, $q_N = 0.9$ which means that each path will be executed only 2 times on average.

How to define test profiles and assess test sizes which meet these requirements will be exemplified in Section 4.3, by the case study described below: the Travel Agency.

4.2: Description of the case study

The example studied corresponds to a cluster of classes which implements the management of a provision catalogue provided by a travel agency. The catalogue management consists in creating and updating provisions. A provision is defined by a destination, a departure date, an arrival date, and one or several services. Three types of services are available: normal, comfort, and luxury services. The program has to perform the following operations: creation of a new provision in the catalogue, update of a provision previously recorded, printing of a provision, printing of the whole catalogue. Updating a provision consists of (i) the modification of the destination or of the departure date or of the arrival date, or (ii) a service addition or cancellation.

The program was developed using the BON (Business Oriented Notation) method [15] and implemented in Eiffel [10] (about 670 lines of code). The BON method produces static descriptions and dynamic descriptions of the system.

Static descriptions document the software structure, that is, what the components are and how they are related to each other. The design of the Travel Agency example has led to the definition of 8 classes (see Fig. 2). All of them are instantiable (no parametrised or abstract class). They are linked together by aggregation relationships, or single inheritance relations involving polymorphism (see Fig. 3).

Dynamic descriptions document how objects interact at execution time. First, for each event triggered by something in the external world of the software, a list of the objects that may become involved as part of the software response is identified. Starting from these lists, possible executions (called *scenarios*) are selected to illustrate the important aspects of the system behavior. For each scenario a *dynamic diagram*, which consists of a set of communicating objects is produced. Figure 4 shows an example of scenario and the associated dynamic diagram: a simple rectangle represents an object, a double rectangle refers to a set of objects; a message sent from one object to another is represented by a dashed arrow labeled by sequence numbers. In the dynamic diagrams, the message relations are always potential. For example, in an execution related to Scenario A: message 2 will be sent only if the provision to be updated has been found in the actual catalogue; only one message will be sent by the *PROVISION* object to create either a *SERVICE* object, or a *C_SERVICE* object, or a *L_SERVICE* object, depending on the type of the service to be added in the specific execution.

4.3: Definition of the testing steps

The Travel Agency allows us to illustrate some of the issues discussed in Section 3: class dependency, polymorphism, dynamic binding, and open interactive behavior.

SYSTEM	CATALOGUE_MANAGEMENT
PURPOSE	Creates and updates provisions

Class	Description
TRAVEL_AGENCY	The root class (actually, a test driver)
PROVISION	Provision (destination, departure date, arrival date, one or several services)
SERVICE	Normal service (flight price, flight number, insurance company)
C_SERVICE	Comfort service (one or several stops in addition to a normal service)
L_SERVICE	Luxury service (one or several luxury stops in addition to a normal service)
STOP	Basic stop (town, hotel category)
C_STOP	Comfort stop (identical to a basic stop)
L_STOP	Luxury stop (guide, tour and two meals in addition to a basic stop)

Fig. 2: System chart for the Travel Agency

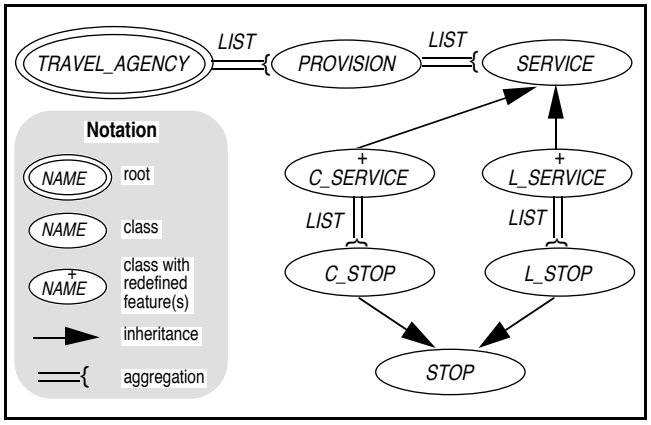

Fig. 3: Static Architecture of the Travel Agency

4.3.1: Static dependency analysis. To define the successive testing steps, we examine the static architecture model (Fig. 3). Since inheritance and client relations induce dependencies between classes, the behavior of any single object would be meaningless. Moreover, we have polymorphism through the services aggregated in the *PROVISION* class. The analysis proceeds as follows:

1. Starting from the root class and retaining only the topmost classes linked together by client relationships provides us with a first subset S_1 of three classes: S_1 = {*TRAVEL_AGENCY*, *PROVISION*, *SERVICE*};
2. To progressively integrate descendants of the elements of S_1, we can choose between two classes: *C_SERVICE* or *L_SERVICE*. The class description (Fig. 2) shows that the former class should involve less features than the later one: hence, we select *C_SERVICE*. Then, starting from this class, we proceed as in step 1 to define a second subset of classes: S_2 = {*C_SERVICE*, *C_STOP*, *STOP*};
3. Finally, starting from the *L_SERVICE* class leads us to define a third subset which contains the two remaining classes: S_3 = {*L_SERVICE*, *L_STOP*}.

Hence, three incremental steps are required to account for the polymorphism of the service features: $SS_1 = S_1$, $SS_2 = S_1 \cup S_2$ and $SS_3 = S_1 \cup S_2 \cup S_3$. SS_1 corresponds to the minimum set of objects that are able to generate meaningful behavior patterns (no test stubs are required).

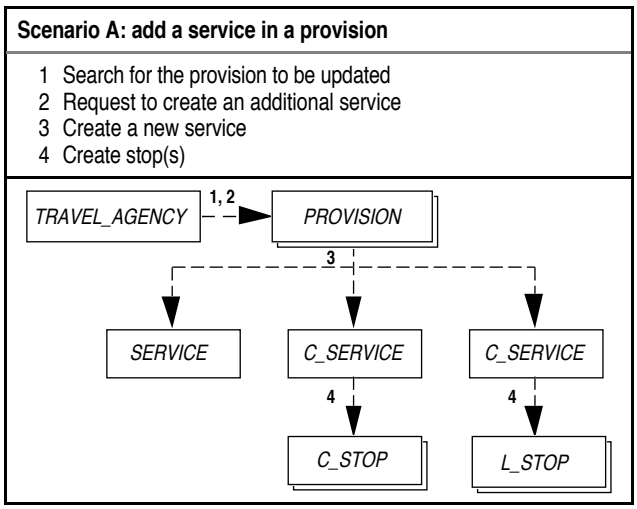

Fig. 4: Example of scenario and associated dynamic diagram for the Travel Agency

4.3.2: Structural analysis. The structural analysis of the routines actually implemented in the Eiffel program aims at providing us with detailed information on (i) all the possible sequences of calls and, (ii) the structural elements (e.g., execution paths) of the routines to be activated in order to provide a complete coverage of the program. Tracing the paths may be difficult when invocation chains of routines traverse the class boundaries. Hence, it is often useful to refer to the dynamic model (Fig. 4) that provides a high level view of the dynamics. The various cases for dynamic binding of *SERVICE* objects have to be taken into account: in each case it is necessary to identify the active part of the software structure, leading to partial control graphs. The analysis is conducted in the following way.

Step 1: analysis of the subsystem $SS_1 = S_1$. The control graph of the creation procedure of the root class *TRAVEL_AGENCY* is first constructed. This graph is made up of 8 execution paths. The nodes of the graph may contain a call to a local routine of the root class, or to a distant routine of the *PROVISION* class. Then, node

expansion consists in replacing the calls by partial control graphs of the corresponding routines: the partial graphs are constructed by ignoring all paths that contain a reference to *C_SERVICE* or *L_SERVICE* classes, since they do not have to be tested in subsystem SS_1. Finally, calls to *SERVICE* class routines are expanded in turn. The control graph G_1 of the subsystem SS_1 is then complete: it is made up of 17 execution paths, each of them involving between one to five routine calls in sequence.

Step 2: analysis of $SS_2 = SS_1 \cup S_2$. Starting from G_1, the control graph G_2 of SS_2 is got by addition of new execution paths. Partial control graphs of the *PROVISION* class routines are augmented with the paths containing calls to routines of *C_SERVICE*. Calls to *C_STOP* routines are then expanded. This leads to 9 additional paths: G_2 is made up of 26 execution paths, each of them involving between one to seven calls in sequence.

Step 3: analysis of the whole system $SS_3 = SS_2 \cup S_3$. In a similar way, starting from G_2, we proceed by addition of paths associated with calls to routines of the S_3 classes. The graph G_3 contains a total of 35 execution paths (9 new paths); the maximum number of routine calls in sequence remains unchanged (7).

4.3.3: Statistical test sets. The class cluster forms an open system that supports interaction with an external operator during computation. The number and type of requests/inputs to be supplied by the operator vary depending on the executed path. This compounds the design of statistical test sets and forces us to proceed in two stages: (i) random choice of a path number, and (ii) random choice of the actual test pattern according to the requests to be supplied when this path is executed. As a result, at each test step, the k execution paths of G_i are first numbered; then the method for designing statistical test patterns (Section 2) is applied to search for an input distribution over the set $\{1, \ldots, k\}$ and assess a test size N_i which gives the total amount of path numbers to be generated. Starting from a sequence of N_i path numbers thus generated, stage (ii) will consist in constructing an actual input pattern for each path number: this pattern is randomly chosen from the input subdomain associated with the path execution, according to a uniform distribution over this subdomain.

As said in Section 4.1.2, the test criteria retained are the coverage of the execution paths of the graphs G_i and, at each step, two different test quality objectives q_N are required depending on whether or not the paths have already been executed during previous steps. The numerical results given below correspond to $q_N = 0.9999$ for new paths, and $q_N = 0.9$ for retested paths. S_{ci} denotes the set of elements to be activated at step i, and p_j the probability of the path numbered j.

Step 1: Subsystem SS_1. The graph G_1 contains 17 paths: $S_{c1} = \{p_1, \ldots, p_{17}\}$. Each path must be strongly tested since it is the first step, and they can easily be made equally likely (i.e., $p_j = 1/17 \ \forall$ j) by choosing a uniform distribution over the set of numbers $\{1, \ldots, 17\}$. For $P = p_j = 1/17$ and $q_N = 0.9999$, relation (1) in Section 2 gives: $N_1 = N_{min} = 152$. Hence, the first testing step is made up of 152 program executions under a uniform distribution over the paths of G_1.

Step 2: Subsystem SS_2. G_2 contains 9 new paths $p_{18}, \ldots, p_{26}$: $S_{c2} = \{p_1, \ldots, p_{26}\}$. Let p_{new} be the probability of each new path, and p_{old} be the probability of each retested path. Since the sum of the probabilities of the 26 paths must be equal to 1, we get:

$$17.p_{old} + 9.p_{new} = 1 \qquad (2)$$

The requirements of $q_N = 0.9999$ wrt new paths and $q_N = 0.9$ wrt retested paths give respectively, from relation (1):

$$N_{min} = \ln(0.0001) / \ln(1-p_{new}) = \ln(0.1) / \ln(1-p_{old}) \qquad (3)$$

Then, replacing p_{old} in (3) by its value deduced from relation (2) gives:

$\ln(0.0001) / \ln(1-p_{new}) = \ln(0.1) / \ln(1-((1-9.p_{new})/17)$, from which we get: $p_{new} = 4/53$, $p_{old} = 1/53$. Then, from (3): $N_2 = N_{min} \cong 120$ test patterns generated according to the distribution defined over the 26 paths of G_2.

Step 3: System SS_3. G_3 contains 9 new paths $p_{27}, \ldots, p_{35}$: $S_{c3} = \{p_1, \ldots, p_{35}\}$. Relation (2) becomes:

$$26.p_{old} + 9.p_{new} = 1 \qquad (2')$$

Replacing p_{old} in (3) by its value deduced from (2') leads us to the following solution: $p_{new} = 4/62$, $p_{old} = 1/62$. Then, from (3): $N_3 \cong 140$ test patterns generated according to the distribution defined over the 35 paths of G_3.

In conclusion, testing of the Travel Agency according to the incremental strategy involves three steps. A complete test set is then divided into three series of test patterns which correspond to a total amount of 412 test patterns.

5 Conclusion and Future Direction

This work is a modest attempt to address the challenging issues of testing OO programs. Having been working on statistical testing for several years, we adopt the view that it is a suitable means to compensate for the tricky link between current test criteria and software design faults. As a first step, we perform a feasibility study of statistical structural testing at a unit level. The example is a class cluster from a Travel Agency software system.

This small case study manages to exhibit some of the problems arising from the OO paradigm. In particular the body of one routine cannot be tested in isolation. This was not obvious from the literature: many authors recommend that each individual routine is tested as a first step of unit testing, using traditional functional and structural methods. Actually, due to the interactive nature of objects, the

smallest meaningful unit is likely to be a (minimal) cluster of classes. The structural analysis at this level requires both class dependency models and control flow models traversing the boundary of the classes; dynamic models helped us to understand the behavior carried by the structure. Assembling this information, it was possible to generate statistical test sets according to a series of input distributions ensuring a suitable probe of execution paths.

Contrary to previous experimental work for procedural programs, we were not able to assess the fault revealing power of the generated sets. The evaluation method that we previously adopted is mutation analysis: in [5], it is shown that, for procedural programs, simple syntactic mutations have the ability to produce complex run-time errors representative of the ones that would have been caused by real faults. Transferring this result to OO programs is questionable. Indeed, we have introduced mutations in the Travel Agency class routines, and they were all revealed by our statistical test sets. But we feel that those mutations are not representative of the problems arising from the misuse of some complex aspects of the OO languages.

The proposed statistical approach, clearly, does not cover all aspects of the OO testing problems. At the unit level, further work is needed to address more complex schemes of (multiple, repeated) inheritance, abstract and parametrised classes, and state dependency problems. At the integration level, practical methods for statistical testing are still to be defined. It is expected that, owing to the fact that the individual clusters have been thoroughly tested through statistical sets, it will be possible to base the analysis on more abstract models and less stringent criteria. For medium-scale procedural programs, retaining weak criteria and deriving several test profiles, each one focusing on the coverage of a subset of a hierarchy of models, was an efficient approach that allowed us to manage complexity [14]. For OO programs, our investigation will be supported by experimental work performed on a case study involving several class clusters.

Acknowledgments. This work is partially supported by the European Community (ESPRIT Project n° 20072: DeVa). We wish to thank Caroline Parent very much for her useful contribution to the Travel Agency case study, within the framework of a student project.

References

[1] S. Barbey, D. Buchs, C. Péraire, "A Theory of Specification-Based Testing for Object-Oriented Software", in *Proc. 2nd European Dependable Computing Conf. (EDCC-2)*, Taormina, Italy, pp.303-320, 1996.

[2] B. Beizer, *Software Testing Techniques,* 2nd Edition, Van Nostrand Reinhold, New York, 1990.

[3] G. Bernot, M-C. Gaudel, B. Marre, "Software Testing Based on Formal Specifications: A Theory and a Tool", *IEE Software Engineering Journal*, 6 (6), pp.387-405, 1991.

[4] R. Binder, "Testing Object-Oriented Software: a Survey", *Software Testing, Verification & Reliability*, 6 (3/4), pp.125-252, 1996.

[5] M. Daran, P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations", in *Proc. ACM Int. Symp. Software Testing and Analysis (ISSTA'96)*, San Diego, USA, pp.158-171, 1996.

[6] R-K. Doong, P. Frankl, "The ASTOOT Approach to Testing Object-Oriented Programs", *ACM Trans. Soft. Engineering and Methodology*, 3 (2), pp.101-130, 1994.

[7] M-J. Harrold, J. McGregor, K. Fitzpatrick, "Incremental Testing of Object-Oriented Class Structures", in *Proc. 14th IEEE Int. Conf. on Software Engineering (ICSE-14)*, Melbourne, Australia, pp.68-80, 1992.

[8] P. Jorgensen, C. Erickson, "Object-Oriented Integration Testing", *Communications of the ACM*, 37 (9), pp.30-38, 1994.

[9] D. Kung et al., "Developping an Object-Oriented Software Testing and Maintenance Environment", *Communications of the ACM*, 38 (10), pp.75-87, 1995.

[10] B. Meyer, *Eiffel - The Language,* Prentice-Hall International, Object-oriented Series, 1992.

[11] D. E. Perry, G. E. Kaiser, "Adequate Testing and Object-Oriented Programming", *Journal of Object-Oriented Programming*, 2 (5), pp.13-19, 1990.

[12] M. Smith, D. Robson, "Object-Oriented Programming – the Problems of Validation", in *Proc. IEEE Conf. on Software Maintenance*, New York, pp.272-281, 1990.

[13] P. Thévenod-Fosse, H. Waeselynck, Y. Crouzet, "An Experimental Study of Software Structural Testing: Deterministic versus Random Input Generation", in *Proc. 21st IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, Montréal, pp.410-417, 1991.

[14] P. Thévenod-Fosse, H. Waeselynck, "Statemate Applied to Statistical Testing", in *Proc. ACM Int. Symp. on Software Testing and Analysis (ISSTA'93)*, Cambridge, USA, pp.99-109, 1993.

[15] K. Waldén, J-M. Nerson, *Seamless Object-Oriented Software Architecture,* Prentice-Hall, The Object-Oriented Series, 1995.

[16] P. Wegner, "Dimensions of Object-Based Language Design", in *Proc. ACM Conf. on Object-Oriented Programming, Systems and Language (OOPSLA'87)*, New York, pp.168-182, 1987.

[17] P. Wegner, "Interactive Foundations of Object-Based Programming", *IEEE Computer*, 28 (10), pp.70-72, 1995.

[18] E. Weyuker, "Axiomatizing Software Test Data Adequacy", *IEEE Trans. Software Engineering*, SE-12 (12), pp.1128-1138, 1986.