Object-Oriented Testing: Aspects to Test

El Houssain Htite, Rachida Dssouli, Gregor v. Bochmann and A. Ghedamsi

Université de Montréal
Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences, C.P. 6128, Succ "A"
Montréal, (Québec) Canada H3C 3J7

Email: {htite, dssouli, bochmann, ghedamsi} @iro.umontreal.ca
fax: (514) 343-5834

Abstract

Test sequence selection for conformance testing of communication protocol implementations has been an active research area during the last decade. Many testing methods were developed to produce test sequences for testing software and hardware systems. They test systems represented by single Finite State Machines (FSMs). In the object-oriented approach, a specification of system is seen as a dynamic set of Finite State Machines. In order to test this type of systems, other methods need to be applied. Very little work has been done in such a direction. In this paper, we first define a fault model for object-oriented specifications, then we study different aspects, which influence object-oriented testing such as: inheritance, attributes, relationships and dynamic configuration. We also propose a test selection methodology that can help to test object-oriented implementations.

Keywords: Object-oriented testing, fault model, test selection methodology.

1. Introduction

The conformance testing of implementations is important during the development of communication protocols. It consists of selecting some test sequences from the specification and execute them on the implementation in order to check any abnormal behavior. The use of Formal Description Techniques (FDTs) to specify protocols has a benefit impact to develop many test selection methods [Chow 78, Gone 70, Nait 81, Sabn 88]. A lot of FDTs are based on the finite state machine formalism [Koh 78]. In this formalism, the specification of a system is characterized by a set of states, a set of inputs, a set of outputs, an initial state and a set of transitions which connect states.

The object-oriented approach based on the Entity-Relationship formalism [Chen 76, Mond 90, Boch 90, Boch 92] came with new concepts such as: the inheritance, the relationships and the dynamic configuration. It possesses the advantage of being natural. Each physical component of a real system can be represented in the specification by an entity. A lot of research work has been done on the development of object-oriented programming languages and object-oriented design techniques [Boch 91]. We believe that the new concepts of object-oriented approach will change the test selection process.

In this paper, we are interested in testing object-oriented implementations and the aspects influencing such a process. We use the "black-box" testing and examine the limitation of the application of existing test selection methods. The remaining of the paper is organized as follows. In Section 2, we introduce the object-oriented approach. In Section 3, we define a fault model for object-oriented specifications. Section 4 describes the new aspects to consider in the testing process. Section 5 introduces our test selection methodology. In Section 6, we discuss the different results, the limitations of the approach and we give some conclusions and suggest directions for future work.

2. Overview of the object oriented approach

In structured programming, a system is seen as a set of procedures and a set of data. Little change in the data structure risks to introduce deep changes in the organization of one or several procedures [Mas 89]. The object-oriented programming, which is based on new concepts came to avoid such a drawback by regrouping data and procedures in the same entity called object.

An object-oriented specification is described by objects. Each object is an instance of a class or an object type. A class is a conceptual model describing objects. The object creation must be done by respecting the conceptual class model. The specification of a class contains three parts: a static part, a dynamic part and an interface part (Figure 1). The static part defines data, also called attributes, which are local to each instance of a class (object). The dynamic part contains procedures which manipulate data. These procedures are used to describe the object behavior. The interface part allows a list of operations through which the object can be invoked by other objects. The interface is a unique way to insure communication between objects. For example, an item in a store can be modeled by a class having the attributes: name, quantity and price. This class can offer the following operations: add, remove, price-up-date and store-item-value.

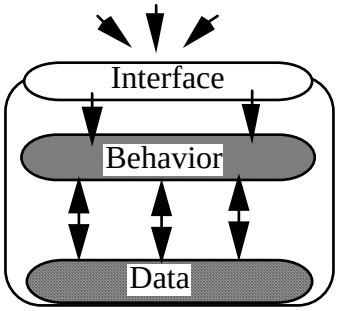


Figure 1 : the object concept

The computation between objects is performed by rendezvous mechanism. The caller invokes the called object for a specific operation, sometimes with parameters. The invoked object executes the behavior related to the selected operation depending on the parameters and the state of an object. When the called object completes the execution, it returns the control and the result (if any) to the caller object.

A class can also model relationships. A relationship is an association relation between objects. It can be represented in the object by using attributes for referring associated objects or by a separate object. When the relationship is an object, it offers implicit operations, which return the information about the objects under the corresponding relationship.

The inheritance aspect is one of the fundamental characteristics in the object-oriented approach. It helps in developing new classes from old ones. A new class, called sub-class, inherits all attributes, operations and behavior of the old class (super-class). The sub-class also can be extended by adding other attributes, operations and behavior. In the case in

which the inherited operations and behavior are not appropriate for the sub-class, the inheritance mechanism allows the possibility to redefine them. In some object-oriented languages, the inheritance can be single or multiple. It's single when the sub-class can have only one super-class and it's multiple when it can have many super-classes.

3. Fault model

During the testing process, it is important to take into consideration different possible faults in the purpose of detecting them when they manifest in an implementation. The set of faults, which could occur into an implementation with respect to its specification, is very big. Sometimes the description of these faults is very complex. In addition, several faults could provoke the same observable error. To classify faults, we use a fault model describing their effects [Boch 91a]. This description is a high level of abstraction, containing several classes, where each one of them includes several types of faults. It has for consequence to reducing the number of possibilities to take into consideration.

3.1. Fault model for Finite State Machine

The type of faults considered in this model are the following [Boch 91a]:

a)-Output fault: it occurs when the output of a transition in the implementation differ from the one in the corresponding transition of the specification.
b)-Transfer fault: it occurs when the terminal state of a transition in the implementation is different from the one in the corresponding transition of the specification.
c)-Transfer fault with additional states: same types of faults can only be modeled by additional states with transfer faults which lead to those additional states.
d)-Additional or missing transitions.

3.2. Fault model for Software

The types of software faults can be classified into two classes [Boch 91a]: the process faults and data faults [Sari 87, Ural 91]. Some of those faults are:

a)-Sequencing faults, such as: wrong logic expression to control loops.
b)-Arithmetic manipulation faults.
c)-Calling wrong function.

d)-Wrong specification of data type.
e)-Wrong initial value.
f)-Wrong referencing variable.
g)-Defining a variable without subsequent usage.

3.3. Fault model for Object-Oriented specifications

In the Entity-Relationship model, a specification is described as a set of classes. Each class could represent an object or a relationship. One fault model for object-oriented specifications must be able to take into consideration the different characteristics of the object-oriented approach. We will define in the following, one fault model for the object-oriented specifications. This model is described by two classes, where each class defines a type of fault. These classes are: the class of configuration faults and the class of behavior faults.

a)-The class of configuration faults

This class concerns all faults caused by the absence or existence in plus of objects and/or relationships. All objects and relationships described by the specification have to exist in any of its implementations. The absence of an object or a relationship could imply the non-conformance of the implementation. Two sub-classes are considered (Figure 2):

1)-Object existence fault: one case is when a specified object doesn't exist in the implementation. The other case is when there is an object in plus in the implementation.

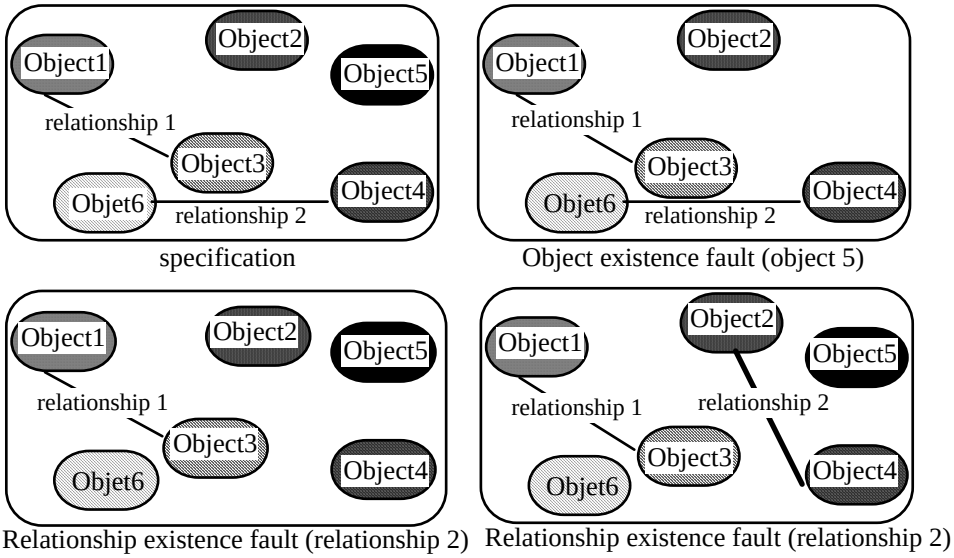


Figure 2: class of configuration faults

2)-Relationship existence fault: one case is when a specified relationship doesn't exist in the implementation. The other case is when there is a relationship in plus in the implementation. The case when the relationship doesn't attach the right objects can be seen as a combinations of faults where the first fault is the absence of a relationship and the second fault is the existence in plus of another relationship.

This class of faults can be manifested during the initialization of the implementation. It represents a type of static configuration faults, but it can also take place during the execution of some object behaviors (dynamic configuration faults). Indeed, some object behaviors could introduce some changes in the configuration of the implementation by creating or by destroying other objects and/or relationships. It concerns the situation where these creations or destructions didn't happen correctly.

b)-Class of behavior faults

This class of faults includes faults related to the behavior of objects. We propose the following classification:

1)-Wrong operation or object call: the interface of an object is formed by a set of operations. The invocation of an object must be done by specifying one operation among this set. The faults of this sub-class concerns the invocations (point c in software fault model). For example, the object A is invoked by object B for operation X but it must be

invoked for operation Y (the object A offer operation X and Y). An other possible case is when the object A invokes object B for operation Z but it must invoke object C (the two objects B and C offer the operation Z).

2)-Incorrect operation parameters: this fault occurs when the right object is invoked with the right operation but one or more parameters of the operation are erroneous (points d and e in the software fault model).

3)-Illegal operation sequence: to execute a behavior, an object could invoke one or several objects by a sequence of operations. Sometimes the order of the wanted sequence is very important, and any permutation of operations could generate undesirable behaviors (point a in the software fault model).

4)-Output fault: point a in FSM fault model.

5)-Transfer fault: point b in FSM fault model.

6)-Transfer fault with additional states: point c in FSM fault model.

7)-Additional or missing transitions: point d in FSM fault model.

4. Test selection from object-oriented specifications

In this section, we present different aspects to consider during the test selection phase. For each aspect, examined separately, we study the means and the restrictions in order to insure an adequate test generation.

4.1. Test selection for the behavior of object types

The specification of each object type defines the operations and the behavior of any instance of this type. We consider only the objects where their behaviors could be modeled by FSMs. We will process the case of entities that modelise relationships in the following sub-sections. Two kinds of objects are considered: the objects directly accessible by the test system (i.e., objects A and B in Figure 3) and the objects which are not (i.e., objects C, D and E).

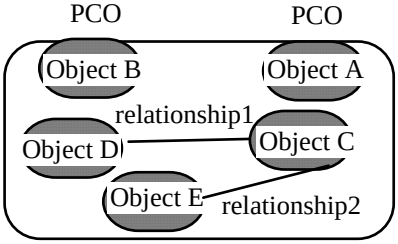


Figure 3: implementation under test

Consider the FSM of Figure 4, where the behavior of object type A is modeled. The object type offers the operations A1 and B1. At the invocation of the object for the operation B1, the object returns the output O2 and transfers to state e2. One sequence of invocations, formed by (A1, B1), brings back the object to its initial state e1. The transition C1 joining the states e5 and e4 represents an invocation for object C. This transition may not be observable depending on the type of the adopted test. For example, in the black-box testing context (see Figure 3), the transition C1 is not observable.

The test selection could be realized under one test select method, such as: TT [Nait 81], DS [Gone 70], UIO [Sabn 88] and W [Chow 78], depending on the nature of the FSM (strongly connected, completely specified) and the fault detection level that we want to achieve. For example, the TT method insures only the detection of output faults. We see in the following sub-sections that other factors can influence this choice depending on the accessibility to the object under test.

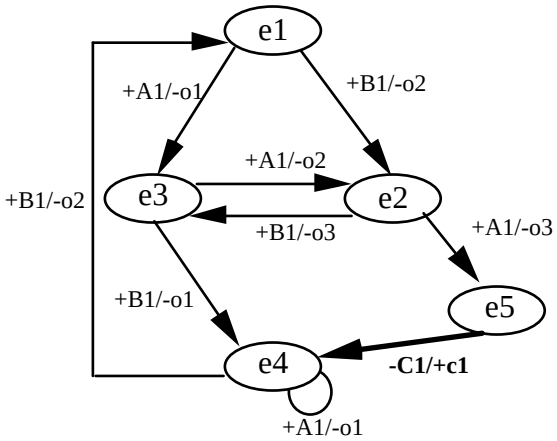


Figure 4: behavior of an object type

4.2. Test selection for parameters

Other tests have to be selected to cover the variation of operation parameters. Face to the impossibility to make an exhaustive test, it is preferable to operate with representative values for each parameter. In order to help in the determination of these representative values, two classes of values are to be considered: the class of valid values and the class of invalid values [Gama 91, Vauc 91]. The first class represents the case where the operation must be executed correctly, while the second class corresponds to the opposite case. The determination of each class of values could be done by different manners. The partition class method [Myer 79] consists in dividing the domain of possible values of parameter into a finite number of equivalence classes. This method makes the hypothesis that the use

of one value or another in the same equivalence class would generate the same behavior. Another method called, limit value [Gama 91, Vauc 91], consists in selecting limit values of each parameter as representative values. For example, if the parameter is specified into the interval [1..100], then one could consider the representative values: 1 and 100 for the valid values and 1-ε and 100+ε for the invalid values, where ε represents the smallest positive value not equal to zero.

Now that it is possible to determine representative values for each parameter, the selection of test cases to cover the variation of parameters in an operation have to contain the maximum number of representative valid values in order to minimize the number of test cases. Other test cases will be generated to cover the representative invalid values. Take the example of the operation A1 of object type A which contains two parameters r and s. Suppose that the representative valid values for r and s are respectively r_1, r_2 and s_1, s_2. The representative invalid values are r_i and s_i. The test cases to cover the parameters of the operation A1 are:

1)-The test cases for valid values:

$A1(r_1,s_1)$

$A1(r_2,s_2)$

2)-The test cases for invalid values:

$A1\ (r_i, s_1 \text{ or } S_2)$

$A1\ (r_1 \text{ or } r_2, s_i)$

It is important to note that the selection of test cases for the invalid values covers one single invalid value at once. This choice is explained by the goal of tests that consists in detecting the maximum number of faults and consequently, a test case which includes several invalid values risk to mask some faults.

Another factor to take into consideration is the dependence between parameters in the operation. The selection of tests to cover the variation of parameters cannot be done by the same manner. Indeed, despite that the test cases are selected for representative valid values, they could test only the invalid behaviors and hence, a wrong coverage of normal behavior. To resolve this problem, we introduce the following method [htite 93]:

Given the operation B1 of object type A which contains three parameters r, s and t. We suppose that only the parameters r and s are dependent. We also suppose that R_V, R_i and

T_V, T_i are the sets of representative valid and invalid values of parameters r and t. For each element R_V^j of the set R_V, we associate two subsets S_V^j and S_i^j, where S_V^j contains the representative valid values of parameter s, which respects the dependence relation with the value R_V^j of parameter r. Similarly, S_i^j contains representative valid values of parameter s, which doesn't respects the dependence relation. We also consider the set S_i which is constituted by representative invalid values. The selection of test cases could be done by the following way:

1)-The valid test cases: for each value R_V^j, we select the test cases until the coverage of all values in subset S_V^j. Valid values of the parameter t are also assigned in those test cases, to profit to drive the most possible representative valid values (T_V).

2)-The invalid test cases: for each value R_V^j, we select test cases to cover all the values of subset S_i^j. The values of parameter t will be chosen from the set T_V. In addition, we select separately other cases to cover the sets of invalid values R_i, S_i and T_i.

4.3. Test conformance for object type instances

In an implementation, several instances of an object type can take place. These instances have to offer the same operations and the same behaviors as their type. The test conformance of instances consists to insure that the implementation of instances are done correctly. Theoretically, this verification is simple; it is sufficient to select tests from an object type (section 4.1.) by adding the test cases to cover the variation of parameters (section 4.2.). Then, we apply this test suite to any instance of this object type. All instances should react in the same manner by respecting the specification of their object type. The test conformance can be lengthy and expensive especially, if the number of instances is large. to recover from this inconvenience, one could adopt a test by range. The range could have two shapes:

1)-Random range**:** a certain number of instances will be chosen by a random manner to do the test conformance.

2)-Range by class**:** an equivalence relation will be defined on the set of object type instances. This relation can be defined on several criterias such as the value of an attribute. From each equivalence class one candidate will be chosen to do the test conformance. This

election can be random or fixed (for example, the smallest value of an attribute in the class).

4.4. The inheritance test

The inheritance is a fundamental characteristic of the object-oriented approach. One object type can inherit from one or several other types. This mechanism of inheritance permits the recuperation of attributes, interfaces and behaviors from the inherited object types. The inheriting object type can also offers, in addition, other operations and behavior (extension). The inheritance test consists in insuring that the relation of inheritance is rightly implemented. Suppose that the object type B (Figure 3) inherits from object type A. To verify if this inheritance is rightly implemented, it is sufficient to execute the selected test suite S_a, for the type A (section 4.1. and 4.2.), on the object B. The object B must have to react on the same manner as the object A, given this test suite. It is clear to see the benefit of using the inheritance mechanism in the protocol specifications. A specification using this mechanism allows to recuperate behaviors already defined without being obliged to redefine them a second time. The utilization of the inheritance has an other benefit, but this time during the test selection. This advantage consists in the reusability of tests. Indeed, the test suite S_b to test the implementation of object type B will be included in the suite Sa which also might contain other test cases for the proper behavior of object type B (extension).

4.5. Other aspects to test

The attributes and the relationships between objects are other aspects to examine. We will present them in this section in order to identify the difficulties in their tests. An attribute of an object can have several significances. It would represent a condition or a state, it could be the means to distinguish objects of the same type or it would be the modelisation of a relationship between objects. Since the access to attributes is indirect, one will examines the effect of these attributes in order to verify their implementations. In order to do this test, we suppose that the behavior of the object, which we want to test its attributes, is correctly implemented. Several scripts are to consider following what the attribute could represent:

a)-Attribute condition or attribute state

The test of this type of attribute could be insured by the behavior of the object. The execution of the part of behavior (which is assumed correctly implemented) that use the attribute and generate an output, allows to test the implementation of this attribute. This verification will be repeated several times to cover all representative values of the attribute. For example, if the attribute represents the two states no-active and active, one will execute the behavior of the object to verify that it is in no-active state, then one will invoke the object to take it to the active state and finally one will execute again the behavior for insuring that the object is this time in the active state.

b)-Test of relationships

The test of relationships presents the same difficulties as the test of attributes. However the test of some relationships which only represent the modelisation of operation parameter dependencies (section 4.2.) could be relatively easy. So verify with some test cases that the dependence of parameters are rightly respected returns to insure the existence of these relationships (Figure 5). The other test cases with valid values of parameters which doesn't respect the dependence will attempt to verify the nonexistence of unspecified relationships.

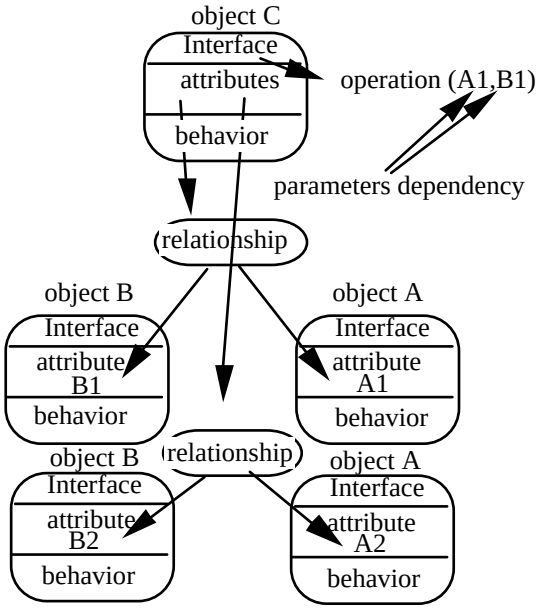


figure 5: test of relationships.

In the conclusion, and under the hypothesis that the behavior of involved objects are correctly implemented, the test of parameters as well as the one of dependencies between parameters are the instruments that could help to test some type of attributes and relationships.

5. Test selection Methodology

We will present in the following, a test selection methodology for object oriented specifications. This selection supposes that the execution of tests and the observation of the implementation outputs will be through service access points; this constitutes a "black-box" test.

Our object oriented model to test is formed by several objects and relationships (Figure 3). The behavior of each object is modeled by one FSM. The execution of tests, to be selected, are taking place on the points of control and observation (PCOs). Outside those points, no observation is possible. In this model, a PCO is a special object of the specification, it plays the role of interfacing between the outside of specification and the other internal objects. The methodology of selection that we suggest consists of the following test selections:

-Test cases to verify the behavior of objects, conformance tests, attribute tests, relationship tests and the inheritance tests.

-Test cases to verify the configuration.

5.1. Test selection for objects behavior

The selection of tests for the object behaviors has as a goal the verification of the conformity of object behaviors to their specifications. Following the test selection method adopted, it requires to cover all transitions of objects FSM. The coverage of object behaviors are not without problem, indeed only the behavior of PCO objects (directly accessible) could be covered efficiently.

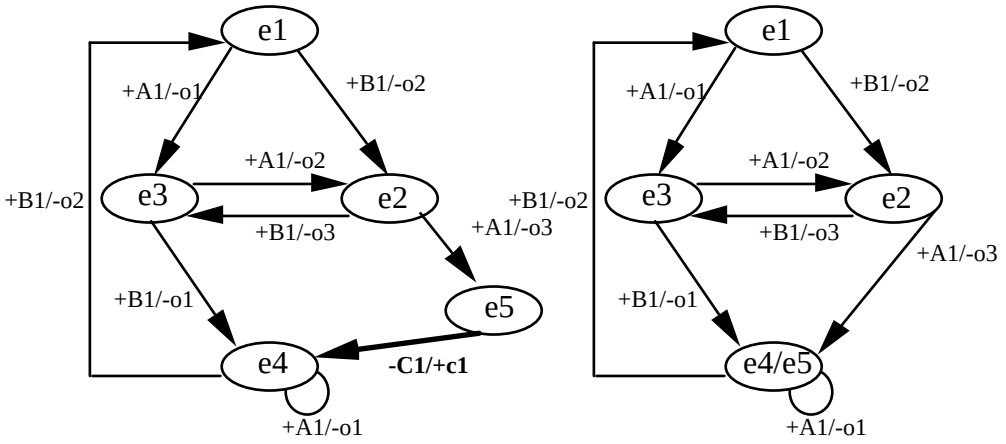
Figure 3 gives an example of an implementation. It composes of five objects A, B, C, D and E which only objects A and B are directly accessible. The behavior coverage of objects A or B can be done easily, since any input to the implementation will be immediately consummated by the object in question and any output of the object destined to the outside will be directly observable. However, the transitions of objects A and B (PCO objects) which represents the invocations of internal objects couldn't be directly observable. The

access to internal objects is not direct, their behaviors couldn't be directly observable either.

a)-Test selection for objects directly accessible

The FSM of the Figure 6a models the behavior of the object A. The transition C1 represents one operation offered by the internal object C. The object A invokes the object C for this operation when it is in the state e5. This internal transition of the object A is invisible from the outside. The FSM of the figure 6b represents the observable behavior of the object A, it's on this FSM that the test selection could be done.

For objects directly accessible the classical test selection methods (TT, UIO, DS, W) can be applied depending on the nature of FSM of the observable behavior (completely specified, strongly connected). At a time that the choice is done on the test selection method, we select the suite of tests to which we add other test cases to cover the parameters (valid values, invalid values) with keeping into account the dependencies between them. In the case of inheritance between PCO objects we would reuse tests selected for the ancestor objects.



a: complete behavior of the object A b: observable behavior of the object A

Figure 6: complete and observable behavior of the object A

b)-Test selection for non accessible objects and relationships

consider the object C in Figures 3. This object is led to communicate with others objects (the objects A, B, D and E). The coverage of accessible object behaviors will imply automatically the coverage of a part of object C behavior. In their executions, the objects A

and B could invoke the object C to execute some behavior. Figure 7 shows the different parts of object C behavior.

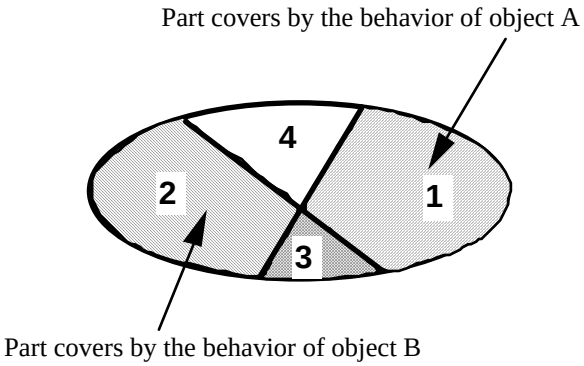


Figure 7: coverage of behavior of non accessible object C

The coverage of parts 1 and 2 is insured respectively by the behavior of objects A and B took separately. As for the coverage of part 3, which represents the intersection of parts 1 and 2, it will be insured at the same time by the object A and B. The coverage of the part 4 could be insured by other non accessible objects (for example, the objects D and E) or by combining the behaviors of accessible objects. Two cases however are to consider for the non accessible object under test (Figures 8):

1- the object under test could be invoked directly by the accessible object (the object C, Figure 8-a)

2-the object is invoked by the accessible object via one or several non accessible objects (the object E, Figure 8-b)

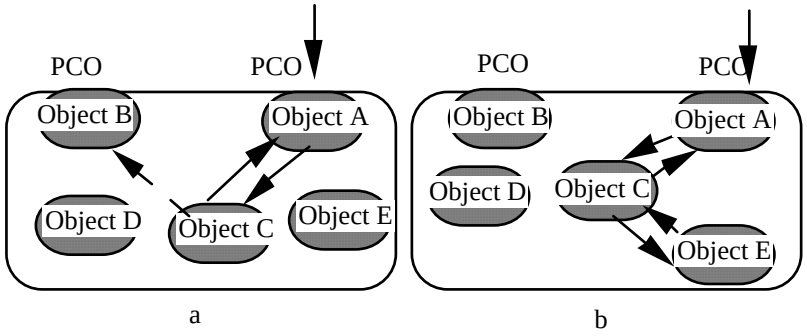


Figure 8: coverage of non-accessible objects

In order to insure the coverage of non accessible object behavior, one is forced, in both cases to assume the hypothesis that the behaviors of accessible objects and intermediate non accessible objects are correctly implemented. The idea consists in seeing every non

accessible object, the consequence of execution of tests selected for the objects PCO and complete these tests in order to cover the totality of behavior of non accessible object.

Like it was mentioned in the section 4.2., some relationships which model the dependence of parameters could be implicitly tested in keeping into account the parameters. For the others relationships which doesn't concern the dependence of parameters, the idea is to activate the behavior of PCO objects which will use these relationships and which would generate an observable output.

5.2. Test selection for the configuration

The implementation under test in its processing run incurs some changes in its configuration. Indeed, some objects behavior can create objects or relationships, while others will be destroyed. The test of configuration has for task to insure that the creations and the destructions of objects or relationships take place correctly. The same hypothesis, which considers the behaviors of implied objects are implemented property, stays valid. The idea to insure this test is to activate the behavior of PCO objects which invokes the objects or relationships which are destroyed (to verify that they are correctly destroyed), the same for the objects or relationships created (in order to insure that they are correctly created).

6. Concluding discussion.

We suggested in this paper a methodology of selection test from oriented object specifications. We apply this methodology on the example of PCS "Personal Communication Service" specification [Bala 90, Regn 90, Desb 92]. This example allowed to place into evidence the problem of test selection for non accessible objects, the relationships and the attributes. We assume the hypothesis that the behavior of intermediate objects are correctly implemented. This explains by the partial observation that we have in our model of test and which doesn't allows to make the internal transitions observable. However if we had other points of observation on the internal objects we would reduce the impact of this hypothesis.

In order to raise this restriction, we suggest in this conclusion some ideas which constitutes a logical sequence to our work. Face the complexity of examined aspects and which summarizes to a partial observation of internal objects, the introduction of some

instrumentations in the object which can help to test other objects would increase the observability of systems under test. In this optic, one would imagine that each type object would be formed by two facets. The first facet used to describe the normal behavior of the object; while the second facet would give the means to test other object types. The objects like this constructs can be tested mutually to establish a final verdict. The advantage of a such system is to allow at the same time to make the tests with more observation and to insure the diagnosis for the location of faults.

An other possible difficulty is the case non deterministic FSM (NFSM). The selection of test from NFSM is a classic problem. G. Luo in [Luo 92] suggests a method to permit the selection of tests. The idea of this method, is to make some transformations in the NFSM, under certain hypotheses, in the goal to make this NFSM deterministic in the input /output sense. So the selection can be done on this resultant machine.

An other point that was not treated is the architecture of test system. Indeed the coordination of tests and their synchronizations require to put in place a superior level of testers. In [Meer 91] we find a presentation of this problem.

Acknowledgments

The authors would like to thank CITR for its financial support.

References

[Bala 90]: K. Balasubramanya and G. Rochlin, Universal Personal Telecommunications: Concepts and Requirements, IEEE, ICC'90, pp 0228-0232, 1990.

[Boch 90]: G. V. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval and N. Williams: MONDEL, Object-Oriented databases Specification language, Technical Report from CRIM, Departemental Publication #748, Department IRO, University of Montreal, 1990.

[Boch 91]: G. v. Bochmann, S. Poirier and P. Mondain-Monval: Object-oriented design for distributed systems and OSI standards, Departmental Publication #768, Department IRO, Université of Montreal, April 1991.

[Boch 91a]:G.v. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi and G. Luo, "Fault models in testing (invited paper)", IFIP Transactions, Protocol Testing Systems IV (the Proceedings of IFIP TC6 Fourth International Workshop on Protocol Test Systems), Ed. by Jan Kroon, Rudolf J. Heijink and Ed Brinksma, 1992, North-Holland, pp.17--30.

[Boch 92]: G. V. Bochmann, S. Poirier and P. Mondain-Monval: Object-oriented design for distributed systems and OSI standards, Proc. of IFIP Int. Conf. on Upper Layer Protocols, Architectures and

Applications, Vancouver, May 1992. A shorter version is also included in the proceedings of the Int. Workshop on ODP, Berlin, Oct. 1991.

[Chen 76]: P. P. Chen, The Entity-Relationship Model: Toward a Unified View of Data, ACM Trans. on Database Systems, Vol. 1, N 1, pp. 9-36, March 1976.

[Chow 78]: T. Chow: Testing Software Design Modeled by Finite-State Machines, IEEE Trans. Software Engineering, Vol.SE-4, pp.178-187, March 1978.

[Desb 92]: D. Desbiens: Modelization and Specification of the Personal Telecommunication Services, Master thesis, Department IRO, University of Montreal, 1992.

[Gama 91]: P. Gamache: Inteligent tests generation adapted to communication protocols, Master thesis, Department IRO, University de Montreal, March 1991.

[Gone 70]: G.Gonenc: A Method for the Design of fault Detection Experiments, IEEE Transactions on Computer, Vol 19, N 6, pp. 551-558, 1970.

[Htite 93]: E. Houssain Htite: Test generation for personal communication service, Master thesis, Department IRO, University of Montreal, 1993.

[Koh 78]: Z.Kohavi: A switching and Finite Automata Theory, Mc Graw Hill, 1978.

[Luo 92]: G. Luo, G. V. Bochmann and A. Das: Test Generation for Concurrent Programs Modeled by Communicating Non-deterministic Finite State Machines, Departmental Publication #823, department IRO, University of Montreal, May 1992.

[Mas 89]: G. Masini, A. Napoli, D. Colnet, D. Léonard et K. Tombre: The Object Languages, interEditions, iia, Paris 1989.

[Meer 91]: J. de Meer, V. Heymer, J. Burmeister, R. Hirr and A. Rennoch: Distributed Testing, 4th International Workshop on Protocol Test Systems, Part IV, Oct 1991.

[Mond 90]: P. Mondain-Monval, G. V. Bochmann: An object-Oriented Software Design Methodology, Progress Report Document N 7 for CRIM/BNR Project, June 1990.

[Myer 79]: G. J. Meyers: The Art of Software Testing, Wiley-interscience publication, John Wiley and Sons, pp. 177, 1979.

[Nait 81]: S. Naito and M.Tsunoyama: Fault Detection for Sequential Machines by Transition Tours, Proceeding of the 11th IEEE fault Tolerant Computing Symposium, IEEE Computer Society Press, pp. 238-243, 1981.

[Regn 90]: J. Régnier and W. H. Cameron, Bell-Northern Research, Personal Communication Services: The New Pots, Globecom 90, San Diego, December 1990.

[Sabn 88]: K. K. Sabnani and A. T. Dahbura: A protocol Test Generation Procedure, Computer Networks and ISDN Systems, Vol 17 (4), pp. 285-297. 1988.

[Sari 87]:B. Sarikaya, G.v. Bochmann and E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Trans. on SE, April 1987, pp.518-531.

[Ural 91]: H. Ural and Bo Yang, " A Test sequence selection method for protocol testing", IEEE Transaction on Communications, April 1991.

[Vauc 91]: J. Vaucher, G. v. Bochmann, B. Lefebvre, S. Desmarais and P. Gamache: MMS project "L'informatique intelligente appliquée à l'implantation et au test de logiciels industriels", Intelligence Artificielle et Sciences Cognitives au Québec, Vol.3, N 3, pp.45-58, 1991.