Testing Ada 95
Object-Oriented Programs

Stéphane Barbey

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

email: stephane.barbey @ di.epfl.ch
phone: +41 (21) 693.52.43 - fax: +41 (21) 693.50.79

30 September 1995

Abstract. We show some of the specific problems for testing software introduced by the object-oriented features of Ada 95, and focus on specification-based testing, since this strategy is the key strategy for testing object-oriented software. We present a theory for testing software by refinement of an exhaustive test set into a finite test set using three reduction hypothesis. We also show how the Oracle problem can be partially solved using some particular features of Ada 95.

Keywords. Ada 95, Object-Oriented Programming, Testing, Test Sets Selection, Reduction Hypothesis, Oracle

1 Introduction

In this paper, we show some of the specific problems for testing software introduced by the object-oriented features of Ada 95. We consider testing programs developed with an object-oriented method, such as Fusion, OMT, or Booch, and resulting in an object-oriented architecture, i.e. a set of classes and objects which communicate by sending messages to each other. We have shown in [2] how Ada can be used to map such designs.

We focus on specification-based testing, since this strategy is the key strategy for testing object-oriented software. The architecture of object-oriented software leads often to multiple implementations of one specification: we can benefit from the test sets selected for one specification to test the various implementations, thus satisfying to the economics of testing.

In a previous paper [4], we have shown a theory to create finite test sets from a usually infinite exhaustive test set, and eventually (semi)-automatically generated assuming the availability of algebraic specifications. This strategy for selecting test sets has been applied to Ada 83 ADTs. However, the testing methods for object-based applications cannot be applied as is to object-oriented programs, because of the problems introduced by the presence of a state, inheritance and polymorphism[3]. Therefore, we need to adapt this strategy to take into account the object-oriented features of Ada 95, specifically type extension and class-wide programming.

We also take a look at operational methods (the construction of an oracle) and see how a testing harness can be build for Ada 95, by taking advantage of the child units and the generics.

2 Problems Introduced by Type Extension and Class-Wide Programming

In the case of object-oriented software, the smallest unit of testing is the class. In Ada 95, a class is considered as a modularization unit together with a type and its primitive operations. To be perfectly object-oriented, this type has of course to be refinable, i.e. tagged. An object is an instance of a class — in Ada a variable or a constant of the (tagged) type.

The primitive operations of a class could possibly be tested individually. However doing so would shift the focus of testing away from the essence of the class — the interaction between operations. Moreover, in the specification of the class to be tested, operations are rarely independent from each other and the properties of one operation are usually interleaved with the properties of other operations. Each object has a state and the behavior of the operations of the class is expressed in terms of transitions between states. Thus, every test case must be expressed in terms of a combination of operations invocations (possibly a single invocation) that lead to a certain state, and the test of an operation from this state.

Two features added to support object-oriented programming in Ada 95 open new issues in testing Ada software:

- *type extension*

  Inheritance introduces difficulties for incremental testing, assuming that we want to take advantage of the test set selected for the parent when testing a descendant. It is not obvious to build the minimal set of primitive operations of a derived type to test (i.e. those that have a different behavior in descendant classes). It is not possible to only test the operations which are added or overridden. The obvious intuition that only new methods need testing has been proven false in [12]. Inherited operations can depend on overridden operations, and the behavior of the former can change. Moreover, the added and overridden operations can put the object in a state which could not be attained before. Therefore, the behavior of the inherited operations must also be tested in these new states.
- *class-wide objects*

  Dynamic binding hinders testing, because it prevents a static analysis of programs: dispatching calls are undecidable before run-time. Moreover, all the elements in a class hierarchy may not be known when designing a client, since it is possible to add descendants at any time, without even having to recompile the clients of the ancestor types. Moreover, type conversions inside a class hierarchy may cause errors that cannot be detected at compile-time. We have studied different uses of polymorphism in Ada 95 in [1]. Note that this issue is less important in Ada than in other object-oriented programming languages, because the level of granularity of dynamic binding is not the operation (for example, in C++, virtual or not), but the call (dispatching or not), which contributes to limit the cases of dynamic binding.

3 The Theory of Specification-Based Testing

Specification-based testing is a technique to find errors in a program by comparing the results of the execution of tests against its specification. Thus, the specification of the program is the only source for test cases, and those test cases are completely independent from the code of the program.

We postulate the availability of a complete and valid specification (either formal, or not) as a means to determine the expected behavior of the tested entity. From that specification, we determine test cases to verify the possible implementations of this specification, and an oracle to determine the test satisfaction, i.e. whether a test has discovered an error or not.

The desired properties for a test set are:

- valid: the test sets should accept only correct programs;
- unbiased: no test should detect an error in a correct program;
- finite: the test set should contain a finite number of tests;
- decidable: for every test, the decision of whether the test is satisfied or not should be decidable;
- complete: every property of the specification should be exhaustively exercised in the test set.

To select such a test set for Ada 95 object-oriented programs, we will apply an adaptation to object-oriented systems of the theory of testing developed by Bernot, Gaudel and Marre [6] for testing data types by using formal specifications. This theory is decomposed in two parts:

- *a theory of test selection*

  The theory of test selection consists in selecting from a possibly infinite set of formulae, corresponding to all the specified properties of a program, a finite set of ground formulae which is sufficient to prove the property preservations of a program with respect to its specification. Under the hypothesis that a complete specification of a class is available, convenient strategies can be applied to identify the ideal exhaustive test set.

  The reduction of the exhaustive test set to the final test set, i.e. the ground formulae, is performed by applying hypotheses to the program.Those hypotheses, called reduction hypotheses, define selection strategies and reflect common test practices.

The goal of the strategies is to select formulae showing counter-examples to the validity of the program, i.e. to exhibit the presence of errors.

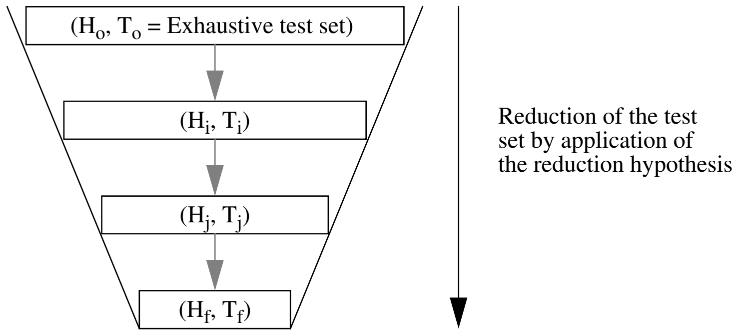

Fig. 1. Iterative refinement of the test context

Thus, as shown in figure 1, we can define test selection as the iterative refinement of a context defined as a couple (H, T) where H is the set of hypotheses, and T is the test set. The process starts from a context (H_o, T_o) which corresponds to the exhaustive test set and no reduction hypothesis. The introduction of a new hypothesis *h* provides a new testing context (H_j, T_j), where $H_j = H_i \cup h$, and T_j is a subset of T_i ($T_j \subseteq T_i$). This refinement of the context (H_i, T_i) into (H_j, T_j) induces a pre-order between contexts ((H_i, T_i) $\leq$ (H_j, T_j)). The context refinement is performed iteratively until we reach a test set (H_f,T_f) that is finite, decidable, and preserves the properties of the exhaustive test set.

- *a theory of test satisfaction*

  Once a finite set of formulae is selected, it is necessary to have a decision procedure to verify that an implementation satisfies a specification, i.e. to verify the success or the failure of every test case. This process is called an oracle [13]. A test set is decidable if, for each test, an oracle can be built to decide whether the test is satisfied or not. In our case, the oracle is based on an *equivalence relationship* that compares the outputs of the execution of a test with the expected results. The oracle decision can be an undecidable problem depending on the complexity of the proof of the equality between expressions in programming languages. For example, such oracle is very difficult to build for concurrent systems [7]. In object-oriented systems, this equivalence relationship can be difficult to build because of data hiding. The equality operation provided by the language is not a satisfying equality relationship, because it does not work well with references: instead of comparing objects, it will compare their access values.

4 The Hypothesis for Test Set Reduction

Three entities come into play when testing a class: its possible states, its defined operations, and the possible values that the parameters of its operations can take, or the possible states that the parameters can take, if these parameters are objects themselves. The exhaustive test set of a class consist of the invocation of all possible operations, starting from all possible initial states, and with all possible parameter values (see figure 2).According to the kind of specification, the tests set can be the tests of all axioms of the specification, instead of all possible calls.

Result := Object.Operation (Arguments) $\Rightarrow$

Final_State (Object) $\equiv$ Expected_Final_State (Object) $\wedge$
Result $\equiv$ Expected_Result

Fig. 2. Structure of a Test

The oracle must then determine it the final state is equivalent to the expected final state, and if the possible results of this operation (values or objects) are equivalent to the expected result. In the case of values (i.e. simple types like Integer, Float, String, enumeration types,...) the predefined equality can be used. For objects however, another equivalence relationship must be provided.

In Ada 95 (see figure 3), such a test for a procedure would consist in the invocation of a primitive operation in which the controlling operand (the object for which the operation is invoked) takes all the possible states, and the other operands all possible values for the other effective parameters. It can easily be generalized to operations in which several operands can be the controlling operand (for example the operation "="), because if an operation has more than one controlling operand, their value must all be of the same specific type (i.e. have the same tag). The results of the operation are obtained from the out parameters.

procedure Operation (Object: **in out** ...; Arguments: ...; Results: [**in**] **out** ...) $\Rightarrow$

Controlling operand *Other operands* *Results*

Is_Equivalent (Final_State (Object), Expected_Final_State (Object)) **and**
Is_Equivalent (Result, Expected_Result)

Fig. 3. Structure of a test (in Ada 95)

The test of a function is a special case of the above test, in which the final and the initial state are the same (as long as no access types are involved in the arguments or in the components of the object), and the result of the function is returned instead of being passed as an out parameter.

Of course, since not all initial states can be created in one single operation invocation, getting into an initial state may imply a sequence of invocations of several operations.

We present three hypothesis to reduce the exhaustive test set, by reducing the number of cases that must be tested for each of those entities, while preserving the integrity of the test set.

4.1 The Regularity Hypothesis

The first hypothesis we apply to the selection of test sets is the regularity hypothesis: if a program behaves correctly for a property instantiated to a set of terms lower in complexity than a defined bound, then it should behave correctly for all possible instantiations. This hypothesis helps limiting the size of the test set by making the assumption that one term of a certain complexity may represent all terms of the same complexity or lower complexity.

This hypothesis helps select and reduce test sets by lowering the number of states of the tested object that must be tested. Therefore, it is used to limit the number of objects of the class under test.

The possible states of an object can be grouped in equivalence classes, each member of which should be behaviorally equivalent. Test sets can be drastically reduced by finding the minimum set of those equivalence classes, since each operation is tested with all possible starting states, i.e. all possible transitions from one state to another (or to the same state) are tested.

For example, test sets for an abstract stack of generic items can be subsumed with two states: Empty and Non-Empty.

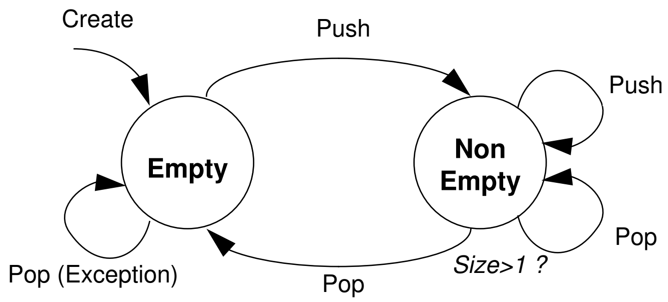

Fig. 4. The reduced set of states and transitions of an abstract stack

Assuming there are two operations defined on this abstract stack, Push and Pop, a test set can be generated by testing all the transactions of the state-based graph (i.e. <-, Create, Empty>, <Empty, Pop, Empty>, <Empty, Push, Non-Empty>, <Non-Empty, Push, Non-Empty>,...). This means that we will not test all possible non-empty stacks (a stack of 12 elements, 13 elements, 14 elements,...), but consider all those stacks to be regular.

4.2 The Uniformity Hypothesis

The uniformity hypothesis helps select and reduce test sets in the presence of class-wide objects and dispatching. The uniformity hypothesis can be stated as follows: if a program behaves correctly for a property A instantiated with a particular term T of a type S, then every instantiation of the property A with terms of the type S behaves correctly. As a consequence, if the property A is correct for one specific instantiation, it is correct for all possible instantiations.

This hypothesis is underlying in random testing, and is used to verify a property with variables of the primitive types (i.e. a type used by the tested type). However, random testing is not applicable in the case of a parameter T belonging to a class-wide type S or one of its descendants, i.e. to a derivation class. In this case, it is even difficult to generate the ideal test set, because of the need to take into account the possibility of newly defined descendants.

For example (see figure 5), in the case of the Stack, the type of the item has no interference with the behavior of the class, even if it is of a class-wide type. In the case of the Alarm System described in the Ada 95 Rationale [5], the operation Manage of a class Manager which would work for all alerts in the Alert hierarchy, would have to be tested for each kind of alert present in the hierarchy (here LowLevel, MediumLevel, EmergencyLevel, HighLevel), and not just for the root type (Alert).

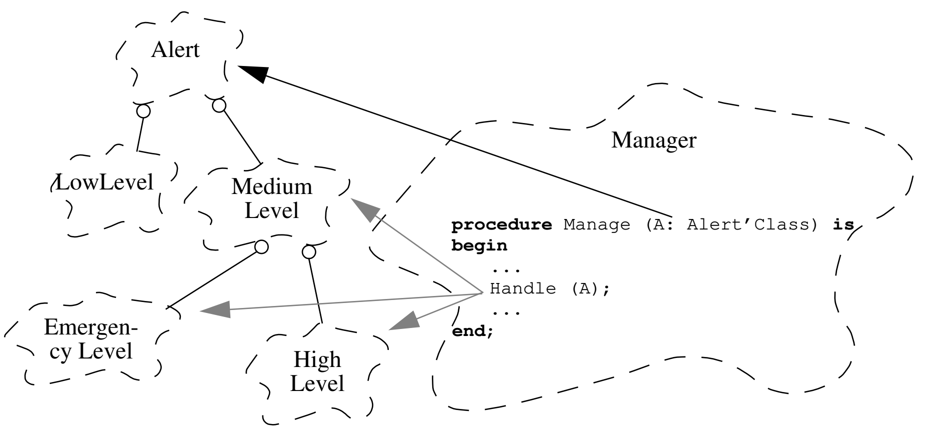

Fig. 5. A Manager for the whole Alert hierarchy

There are several ways to solve this problem. The most used one is to impose a subtype relationship between all types in the class hierarchy. A type S is a *subtype*[1] of T if S provides at least the functionality (c*onformance of behavior*) of T. Objects of type S can be used in any context in which an object of type T is appropriate. The subtype relationship ([9], [11]) ensures that the base type and the derived type have the appro-

1. this is not exactly the Ada subtype.

priate semantics, i.e. that a program that works correctly with objects of the base type has exactly the same effect when substituting objects of the derived type to objects of the base type. Therefore, assuming that the subtype relationship has been carefully tested, it is not necessary to test the operation Manage with the states of all classes of the hierarchy, but just with the states of the root class.

If the requirements of the subtype relationship cannot be achieved, it is possible to provide test sets that test the substitutability for some properties of the hierarchy only, properties that are necessary to ensure substitutability in the context of the tested class (in the example Manager). For best results, it may be useful to take advantage of some limited knowledge of the code, such as information on the presence of dynamic binding.

4.3 The Incrementability Hypothesis

The incrementability hypothesis helps select and reduce test sets in the presence of inheritance by reducing the number of transitions that must be traversed when testing a derived type.

The incrementability hypothesis states that there is a complex, but strong relation between the testing contexts of a tagged type and its derived types, and that we can take advantage of test sets defined for a base type to reduce the new test sets to be selected to test the derived type.

Capitalizing on inheritance can be very useful in minimizing the effort of testing a derived type given the test set developed for the base type. New states and new transitions require testing, whereas pre-existing do not.

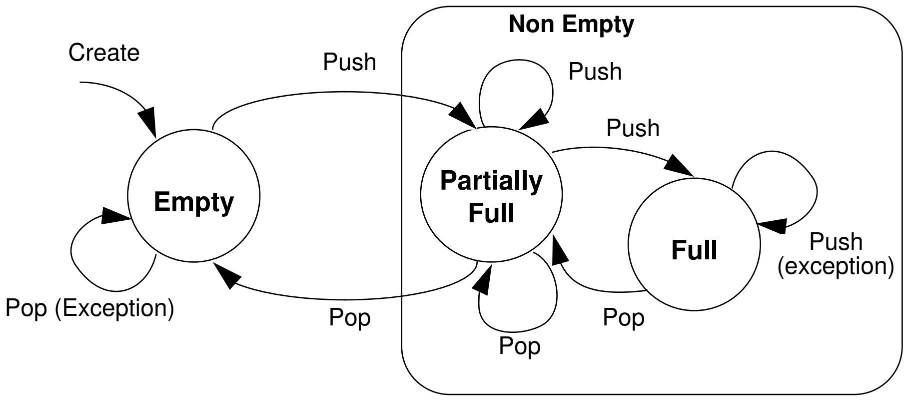

Fig. 6. The reduced set of states and transitions for an unbounded stack

For example, a bounded stack is a refinement of the abstract stack, in which the state Non-Empty is refined into two sub-states, Partially-Full and Full. Thus, some of the test sets defined for Abstract Stack just need to be re-run, because there was no difference in semantics from the parent to the derived type (e.g. <Empty, Pop, Empty>), while in other cases, new test sets need to be selected (e.g. <Full, Push, Full>).

New transitions can also be transitions which were already presenting the specification, but for which new implementations are given in a derived type.

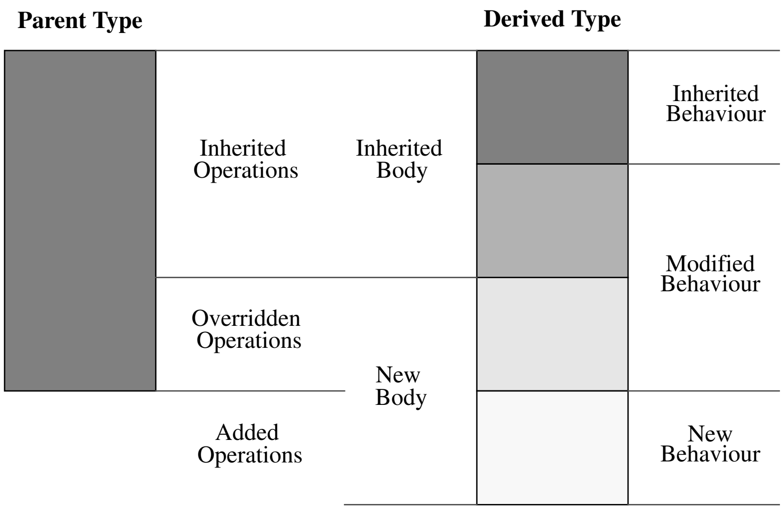

Fig. 7. Incremental testing

In Ada 95 several cases can occur when testing a derived type (see figure 7):

- Operations for which a new body is provided must of course be retested. New test cases must be selected for the added operations. For overridden operations, the test cases selected for the base type must be re-run to prove the subtype relationship, and new test cases may possibly be added to take into account the new properties of the operations. If maintaining the subtype relationship is not expected, then the existing test sets must not be re-run, but a new test set must be provided. The absence of subtyping has of course consequences on the uniformity hypothesis.
- Operations that keep their body do not automatically escape to re-testing, because their behavior may be modified by changes in the behavior of other operations. This case is obviously shown in redispatching. Redispatching occurs when the body of a primitive operation of a type contains a dispatching call to another primitive operation with the same controlling operand.

  For example, on figure 8, the operation Move defined for Vehicle is inherited without modification by Car. However, in its body, it makes a redispatching call to Speed_of. Its behavior for objects of the class Car is not the same as its behavior for objects of the class Vehicle. Therefore, this operation needs re-testing.

```
type Vehicle is tagged
   record
      Passengers: Natural;
      ...;
   end record;
procedure Move (V: in out
Vehicle;
             L: in Location);
function Speed_Of (V: Vehicle)
   return float;

type Car is new Vehicle with ...;
function Speed_Of (C: Car)
   return float;
   -- overridden
-- procedure Move (V: in out Car;
--              L: in Location);
```

```
procedure Move (V: in out Vehicle;
                     L: in Location) is
begin
   ... Speed_of (Vehicle'Class (V))...;
   -- dispatching
end Move;
```

Fig. 8. Example of redispatching

Finally, operations which do not appear in the above cases do not need re-testing: the test performed on the parent type are sufficient to show that those operations meet their specification. In most cases, this set of operations contain most of the inherited operations: for example, adding an iterator to a container class will not imply re-testing the inherited operation.

This algorithm is slightly different if the parent type is an abstract type: abstract types cannot be tested, because objects of those types cannot be declared. However, abstract types can have a specification, and therefore test sets can be generated for them. Those test sets are used for incremental testing in the same way as described above, except that all operations of an abstract types must be tested for the subtype, and not only those with a new behavior, because the operations which kept their behavior could not be tested for the parent type, and their correctness was therefore not exercised.

5 Oracle Generation

There are many ways to solve the Oracle problem. The more general approach is to define the Oracle in terms of external behavioral equivalence such as bissimulation or traces. This approach is for example used in the ASTOOT approach [8].

However, with Ada 95, we can also take advantage of the child unit constructs to get a look inside objects, in a way similar to the strategy explained in [10], except that this approach is non intrusive. This satisfies the requirement expressed sometimes that a tested unit should not be recompiled after having been tested. No recompilation is required for the parent unit when adding a test unit. For complex objects made of an aggregation of sub-objects (components), it is possible to defined several child units with observation operations that give visibility on each sub-object. Of course, extreme care must be take that those observation operations do not affect the behavior to the sub-object.

6 A Test Harness for Ada Programs

The test sets are grouped in a unit that is a child of the unit to be tested, and takes as generic parameter any type that belongs to the type of the tested class. The test set is a set of primitive operations of a new declared tagged type. A non-primitive operation Main performs the tests set.

It is necessary to make the test set package a child unit of the class that it tests so that the possible private operations of the class are visible in the test package and can thus be tested. The equivalence relationship is build inside the test type as a primitive operation of the type, and takes two wide-class parameters to hold the result and the expected result.

```
generic
   type Tested_Unit is abstract new Stacks.Stack_Type with private;
package Stacks.Test_Stacks is
   type Stack_Test_Set is tagged private;
   function Is_Equivalent (L,R:Tested_Unit'Class) return Boolean;
   procedure Test_WWWW (Test_Set: in Stack_Test_Set;
      Stack: in out Tested_Unit'Class; ...; Test_Result: out Boolean);❹
   procedure Test_XXXX (Test_Set: in Stack_Test_Set;
      Stack: in out Tested_Unit'Class; ...; Test_Result: out Boolean);
   ...
   procedure Main (Test_Set: in out Stack_Test_Set'Class);
private
   ...
end Stacks.Test_Stacks;

with Stacks.Test_Stacks;
generic
   type Tested_Unit is new Stacks.Bounded_Stacks.Bounded_Stack_Type with private;
package Stacks.Bounded_Stacks.Test_Bounded_Stacks is

   package Test_Parent_Unit is new Stacks.Test_Stacks (Tested_Unit); ❶

   type Bounded_Stack_Test_Set is
      new Test_Parent_Unit.Stack_Test_Set with private;❷
   function Is_Equivalent (L,R:Tested_Unit'Class) return Boolean;
   procedure Test_XXXX (Test_Set: in Bounded_Stack_Test_Set;
      Stack: in out Tested_Unit'Class; ...;Test_Result: out Boolean);❸
   procedure Test_YYYY (Test_Set: in Bounded_Stack_Test_Set;
      Stack: in out Tested_Unit'Class; ...;Test_Result: out Boolean);❸
   ...
   procedure Main (Test_Set: in out Bounded_Stack_Test_Set'Class);❺
private
   ...
end Stacks.Bounded_Stacks.Test_Bounded_Stacks;
```

Fig. 9. Example of harness for testing stacks

Every test procedure should use the test equivalence in a redispatching way, so that the test developed for a parent type is also valid for the derived types, and makes use of the right equivalence relationship.

```
   procedure Test_WWWW (Test_Set: in Stack_Test_Set;
      Stack: in out Tested_Unit'Class; ...; Test_Result: out Boolean) is
   begin
      ...
      Test_Result := Is_Equivalent (Stack_Test_Set'Class(Test_Set), ...) and ...;
   end Test_XXXX;
```

Fig. 10. Example of test with dispatching equivalence

For example, on figure 10, the Test_WWWW will be valid to test both stack types and bounded stack types, because the equivalence relationship used will be the one defined in the package Stacks.Test_Stacks for stacks types and the one defined in the package Stacks.Bounded_Stacks.Test_Bounded_Stacks for bounded stacks.

The test set for a derived type consists of a child of this class, that takes as generic parameter the type to be tested, like for the ultimate ancestor. However, the test set of the derived type is created by a refinement of the test set of the parent type:

- the test set of the parent type for the derived type is created by instantiating the test set of the parent type with the derived type ❶;
- the test set of the derived type is created by inheriting from the Test_Set type of the parent type ❷;
- the test set of the derived type is then refined by overriding inherited tests (Test_XXXX) or adding new tests (Test_YYYY) ❸; some existing tests (Test_WWWW) ❹ can of course be kept from the parent's test set.
- a new procedure Main has to be defined for the derived test set (that can possibly in turn call the procedure Main defined for the parent type — Test_Parent.Main(...)) ❺. This procedure selects from the inherited test set and the additional tests which must be performed and when.

7 Summary and Conclusion

In this paper, we have shown the bases of a theory for testing object-oriented software. It is based on the refinements of a testing context from an exhaustive test set into a reduced test set, that preserves all the properties of the exhaustive test set. We have shown three hypothesis that help reducing the test set, taking into account the object-oriented features of programming languages, and in particular Ada 95. We have also shown some other features of Ada 95 can help solve the Oracle problem.

We think that this approach to testing object-oriented software has a lots of promises. It has a very strong theoretical ground, it is general, and it is sufficiently easy to be understood by the average programmer. Moreover, it is aimed at reducing the cost of testing — which can be up to sixty percents of the total development cost — by providing minimal test sets.

8 Acknowledgments

The author would like to thank Didier Buchs for its comments on earlier versions of this paper and his thoughtful suggestions. The work reported in this paper is supported by the Swiss National Science Foundation (Nationalfonds), grant number 21-36038.92.

9 References

[1] Stéphane Barbey. Working with Ada 9X classes. In Charles B. Engle, Jr., editor, *TRI-Ada 1994 Conference*, pages 129–140, Baltimore, Maryland, USA, November 6-11 1994. Also available as Technical Report (EPFL-DI-LGL No 94/65).

[2] Stéphane Barbey. Ada 95 as implementation for object-oriented designs. In Charles B. Engle, Jr., editor, *Proceedings of TRI-Ada '95*, Anaheim, California, November 5-10 1995. (to appear).

[3] Stéphane Barbey, Manuel Ammann, and Alfred Strohmeier. Open issues in testing object-oriented software. In Karol Frühauf, editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257– 267, Basel, Switzerland, October 17-20 1994. vdf Hochschulverlag AG an der ETH Zürich. Also available as Technical Report (EPFL-DI-LGL No 94/45).

[4] Stéphane Barbey and Didier Buchs. Testing of Ada abstract data types using formal specifications. In Marcel Toussaint, editor, *Eurospace Ada Europe '94 Symposium Proceedings*, number 887 in LNCS (Lecture Notes in Computer Sciences), pages 76–89, Copenhagen, Danemark, September 26-30 1994. Springer Verlag. Also available as Technical Report (EPFL-DI-LGL No 94/75).

[5] John Barnes, Ben Brosgol, Ken Dritz, Offer Pazy, and Brian Wichmann. *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, MA, USA, February 1995.

[6] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, November 1991.

[7] Didier Buchs. Test selection method to validate concurrent programs against their specifications. In *SQM '95 (Software Quality Management)*, pages 403–414, Seville, Spain, April 1995. (Also Available as Technical Report EPFL-DI-LGL No 95/101).

[8] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[9] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In Norman Meyrowitz, editor, *ECOOP/OOPSLA '90 Conference Proceedings, Ottawa, Canada*, volume 25 of *SIGPLAN Notices*, pages 212–223. ACM SIGPLAN, ACM Press, October 1990.

[10] J. Liddiard. Achieving testability when using Ada packaging and data hiding methods. *Ada User*, 14(1):27–32, March 1993.

[11] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *OOPSLA '93 Conference Proceedings, Washington, DC*, volume 28, pages 16–28, 1515 Broadway New York, NY 10036, September 26 - October 1 1993. ACM SIGPlan, ACM Press.

[12] Dewayne E. Perry and Gail E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January 1990.

[13] Elaine J. Weyuker. The oracle assumption of program testing. In *13th International Conference on System Sciences*, pages 44–49, Hawaii, USA, 1980.