Issues and Theory for Unit Testing of Object-Oriented Software

Stéphane Barbey, Didier Buchs and Cécile Péraire

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

ABSTRACT. *This paper presents an overview of the work performed at the software engineering laboratory of EPFL in the field of testing object-oriented software. It starts with a discussion of the problems specific to object orientation by comparing testing of traditional and of object-oriented systems. It emphasizes the key points of the object-oriented paradigm, encapsulation, inheritance and polymorphism. Then, it shows the adaptation of a general theory of unit testing to object-oriented software. The test selection process is performed by applying three reduction hypothesis (regularity, uniformity and incrementality) to an exhaustive test set. The test satisfaction process is performed using either an oracle built in terms of external behavioral equivalence, or on an ad hoc basis.*

KEYWORDS. *Object-oriented systems, unit testing, test reduction hypotheses, oracle generation.*

1. Introduction

Object-oriented development is viewed by many as the silver bullet that will solve the software crisis. The confidence found amongst its practitioners is sometimes such that "object-oriented" has become a thaumaturgy. However, while the use of object-oriented development methods has increased the quality of software by leading to better, modular architectures, it does not provide correctness by itself, because, alas, "*software is created by error-prone human*" (Boehm, in [24]). Thus, testing software remains an important task, even in the presence of object-orientedness. Since object-oriented technology promotes reuse, it could be argued that the need for meticulously tested components is even more important.

Furthermore, contrary to some belief, traditional testing methods cannot be applied directly to object-oriented software. Some aspects of the nature of the object-oriented paradigm, such as encapsulation, inheritance and polymorphism, introduce problems that were previously not found in structured software and require adapting of the testing methods used.

In this paper, we will first examine what the key points of an object-oriented architecture are in order to identify the constructions that need testing. We will then expose the issues in testing the identified constructions, classes, hierarchies, and subsystems with respect to the most important aspects of the object-oriented paradigm: encapsulation (visibility problems), inheritance (incremental testing problems) and polymorphism (undecidability problems).

Next, we will propose a solution to those issues with an adaptation to the object-oriented systems of the theory of specification-based testing developed at the LRI by Bernot, Gaudel and Marre. We will especially focus on the process of test selection.

All along this paper, some examples will be given, usually written in the Ada 95 object-oriented language.

2. The dimensions of object-orientation

In this section, we will introduce what the main constituents of an object-oriented architecture are.

2.1 Objects and classes

An object is an item that represents a concept which is either abstract, or depicts an entity of the real world [4]. It is usually made up of two kinds of properties: a set of features and a set of operations. The features can be values or other objects and constitute the state of the object while the operations are the subprograms that represent its behavior and can give information on or alter the state of the object.

A class is the template from which objects are instantiated. It encapsulates the properties of its instances and can hide the data structures and other implementation details that are not to be available outside of the class. The non-hidden properties form the interface of the class. They are usually operations only. Therefore programmers can manipulate objects only by invoking these public operations and do not have to care about the data representation of the class. This separation between the specification of a class and its implementation is very important: the same specification can lead to multiple implementations. Since classes encapsulate a complete abstraction, they are easily isolated and can be reused in many applications.

For instance, in the development of a security system the design of a class *Alert* could consist of features such as time and location where an alert is raised, and operations such as *Handle*, which implements the actions to be performed upon an alert.

2.2 Class hierarchy (derivation class)

A descendant is a class derived from one (single inheritance) or more (multiple inheritance) parent classes (its ancestors). It inherits their properties, features and operations, and can be refined by adding new properties or modifying operations (overriding them) to give them a new implementation. In some inheritance schemes, it is even possible for a descendant to decide not to inherit all the operations of its parent. The set of all classes derived directly or indirectly from a given class (the root class) forms a hierarchy.

For example (see figure 1), we could create a hierarchy of alerts, which is based on their priorities. The operation *Handle* could have a different implementation for each class in the hierarchy to specify actions specific to the priority of the alert. This alert-handling system example is inspired from [1]. In the inheritance scheme in which all properties are inherited, each instance of a class belonging to a hierarchy owns a common set of properties which are the properties defined for the root class. Hence, it is possible to work with objects knowing only the hierarchy to which their class belongs, and not the class itself. In programming languages this possibility is often realized by a mechanism called polymorphism, which makes it possible for a name, i.e. a variable or a reference, to denote instances of various classes in the same hierarchy. Since there are different possible implementations of an operation within a hierarchy, it is impossible for the compiler to choose which implementation must be used when invoking an operation on an object denoted by such a name. This choice is postponed until run-time, when it will be made according to the class of the object denoted by the name. This mechanism is called dynamic binding

2.3 Subsystems

A system is a collection of classes and objects used to implement an application. Since systems can be very big, they are usually decomposed into subsystems.

Subsystems are logical collections of classes and objects that collaborate to provide a set of related services. Each subsystem defines an interface that the rest of the system will use to invoke its services. Subsystems of a general nature can have a high degree of reusability.

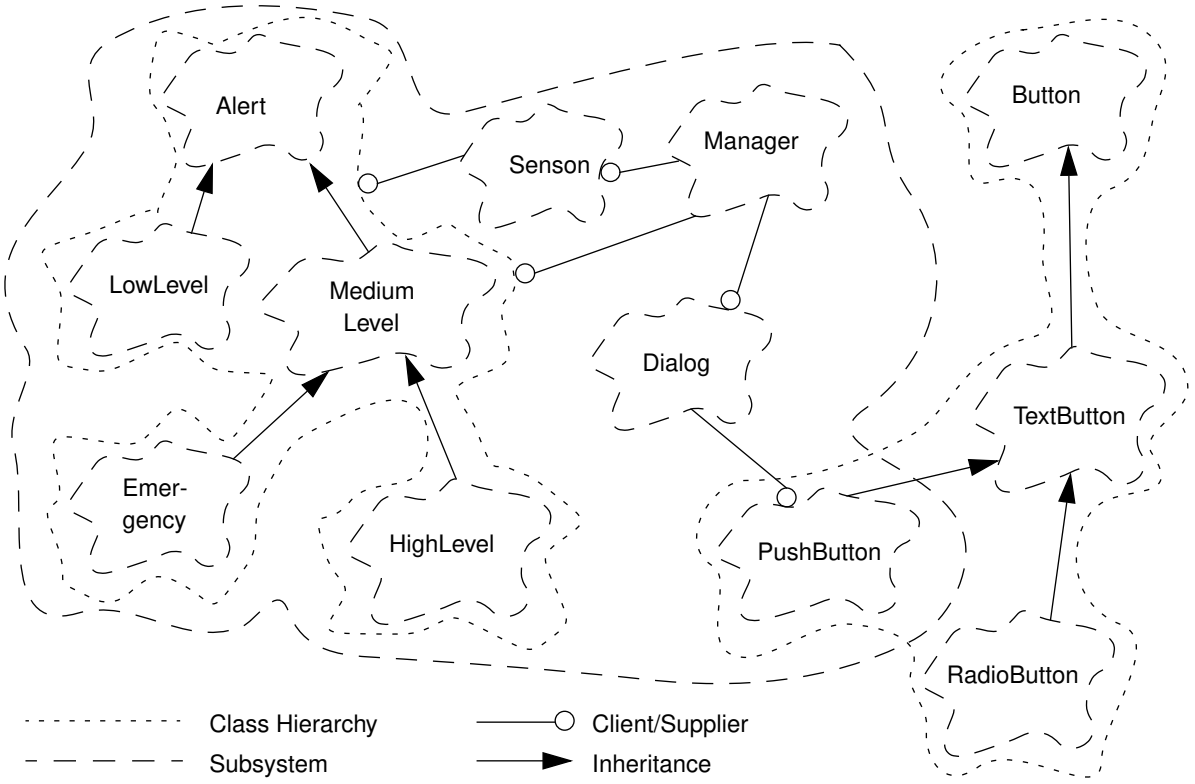

Fig. 1. The dimensions of object-oriented programming

For instance (figure 1), we could imagine an alert-handling subsystem made up of an instance of *Sensor*, which detects the apparition of *Alerts* of any priority[1] and informs an instance of *Manager*, which logs the alert and brings up to the display of the action officer an instance of a *Dialog* box consisting of several instances of *Pushbuttons*[2] representing the different actions that can be undertaken, e.g. ringing the alarm or handling the alert, in which case the instance of *Manager* invokes the operation *Handle* defined for the occurred alert regardless of its implementation. The interface of this subsystem will consist of the public operations of the class *Manager*, which can activate and deactivate the subsystem, and possibly proceed to an audit of the logged alerts.

Such a subsystem could then be reused by bigger systems, for instance an air traffic control system or the management of a nuclear power plant.

3. Testing object-oriented software

Within object-oriented approaches, testing has received less attention than analysis, design, and coding. This lack of interest is mainly caused by a strong belief that the object-oriented technology is the ultimate answer to the software crisis, i.e. applying an object-oriented development method will eventually lead to quality code. Moreover, there is also much trust in some values of the traditional software engineering methods, i.e. applications developed with an object-oriented development method are close enough to applications developed with a traditional method that they can be tested using traditional testing methods. For example, although Rumbaugh and al.[25] assert their willingness to apply the object-oriented technology to all stages of the software development life cycle, they deny the importance of a specific testing for object-oriented software:

[1] The whole Alert hierarchy is in the subsystem.

[2] The rest of the Button hierarchy is not in the subsystem.

"Both testing and maintenance are simplified by an object-oriented approach, but the traditional methods used in the phases are not significantly altered.

However, an object-oriented approach produces a clean, well-understood design that is easier to test, maintain, and extend than non-object-oriented designs because the object classes provide a natural unit of modularity."

J. Rumbaugh et al., [25]

In spite of this confidence, recent studies show that "object-orientedness" is not such a silver bullet.

"The Self group has produced 100'000 lines of C++ and 40'000 lines of Self. We have at least as many bugs in the Self code as in the C++ code, and these bugs are usually found by testing.

I believe that language features are no substitute for testing."

D. Ungar, [27]

Object-orientedness does not by itself ensure the production of correct programs. Although an object-oriented design can lead to a better system architecture and an object-oriented programming language enforce a disciplined coding style, they are by no means shields against programmers' mistakes or a lack of understanding of the specification.

Therefore object-oriented systems still need testing. As a matter of fact, this phase of the software life cycle is even more important for object-oriented software than for traditional software, as the "object-oriented" paradigm promotes reuse: software components will be used and re-used in various contexts, even significantly different from those that the component's developer had in mind, thus the need for robust and well-tested components. In [3], Birss affirms that because of the increased level of usage, reusable components need two to four times more testing that unique components.

3.1 The testing process

Testing is the process of finding programmers errors or program faults in the behavior or in the code of a piece of software. It is used to give confidence that the implementation of a program meets its specifications, although it cannot ensure the correctness of a program. There are several steps in testing of program.

The first step is usually the unit test. To perform this step the tested software is divided into the smallest possible units that can be tested in isolation, which are called basic units [9]. Two tests are usually applied to a test unit, specification-based testing and program-based testing.

- Specification-based testing, also known as black-box testing, is aimed at testing a program against its specification only, i.e. regardless of the way it is coded. It is usually performed by submitting a set of possible inputs, the test data, to the program being tested and by comparing the result to the specification. The decision of whether a test is satisfied or not is made by an oracle [2].
- Program-based testing, also known as white-box testing is a kind of test complementary to specification-based testing. It consists in examining the internal structure of a piece of code to select test data or to gain confidence from certain aspects of the code (e.g. coverage of statements, of paths, of conditions,...).

The other steps include integration testing, system testing, and acceptance testing. These steps are not covered by this paper.

Most work in testing has been done with "procedure-oriented" software in mind, and some good methods of testing have been developed [21]. However, those methods can not be applied as is to object-oriented software, because the architectures of those systems differ on several key issues.

3.2 Classes- vs. procedure-oriented testing

In this section, we will compare the architectures of procedure-oriented systems to object-oriented systems.

In the structured architecture, a program is decomposed functionally into subprograms, which independently implement the services of the program. The basic unit of test is a subprogram, which can consist of or make use of other (possibly embedded) subprograms. Bigger units of testing are assembled from already tested subprograms (bottom-up integration), or alternatively, subprogram stubs in a tested subprogram are replaced by subprograms to be tested (top-down integration). Data handling is accomplished by subprograms, which may not be related, and which can be scattered throughout the system, hence the difficulty in creating a complete test unit. Subprograms communicate either by passing parameters, or by using global variables.

As shown in section 2, the object-oriented architecture is indeed very different from the procedure-oriented architecture.

- There are no global data and data handling is not shared between units. A class contains all the properties that can affect the state of its instance.
- An operation is not a testable component, i.e. it can only be tested through an instance of a class.
- It is impossible, unlike in procedure-oriented (traditional) testing, to build either a bottom-up or a top-down testing strategy based on a sequence of invocations since there is no sequential order in which the operations of a class can be invoked unless inspired by common sense (for instance invoking the operation that creates an object before invoking any other operation of the object).
- Every object carries a state. The context in which an operation is executed is not only defined by its possible parameters, but mainly by the values of the features of the object by which it is invoked (i.e. its state). Furthermore, any operation can modify these values. Therefore, the behavior of an operation cannot be considered independent from the object for which it is defined.

Most of the work in testing object-oriented software has been concerned with testing classes. It is not possible to consider a subprogram as the basic unit of testing anymore. For object-oriented approaches, the basic unit of organization is the class construct. Although the operations of a class could admittedly be tested individually, it is impossible to reduce the testing of a class to the independent testing of its operations, i.e. the tests cannot be designed, coded, and performed as if the operations were "free-floating" subprograms instead of being all related to an object.

Also, testing an object-oriented system must concentrate on the state of the objects and check that the state remains consistent when executing the operations of an object. However, if not directly, traditional techniques can still be applied for some phases of the test plan, especially functional and structural testing; for instance, coverage measures can still be used to determine the effectiveness of a test suite.

3.2.1 Example

To be more concrete, a simple case of testing taken from [21] will be examined.

> *"The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, equilateral."*
>
> *G. J. Myers, [21]*

What Myers had in mind when suggesting this exercise to his readers was to show the difficulty of writing a test case that would adequately test a unit such as the following procedure-oriented (Pascal) unit.

```
type
    Kind: (scalene, isosceles, equilateral, error);

function Kind_of (Length1, Length_2, Length_3: Integer): Kind
begin
   ...
end;
```

Fig. 2. "Procedure-oriented" test unit

In the context of object-oriented programming, the unit to test would probably more look like the following class (in C++ and Ada 95).

```
enum kind
   {scalene, isosceles, quilateral,
    error};

class Triangle : public Shape
{
   public:
      Triangle   (int length1,
                  int length2,
                  int length3);
      ~Triangle ();

      virtual int length_of (int side);

      virtual kind kind_of();

      virtual void homothety (int factor);


      bool similar(
              Triangle *a_triangle);

   private:
      // data representation of a triangle
};
```

```
package Triangles is

   type Kind is
      (scalene, isosceles, equilateral,
       error);

   type Triangle is new Shape
      with private;

   function Create
      (Length1, Length2, Length3: Integer)
      return Triangle;


   function Length_Of (T: Triangle)
      return Integer;
   function Kind_Of (T: Triangle)
      return Kind;
   procedure Homothety
      (T: in out Triangle;
       Factor: in Integer);
   function Similar (T1, T2: Triangle)
      return Boolean;

private
   -- full declaration of Triangle
end Triangles;
```

Fig. 3. "Object-oriented" test unit

The noticeable changes from the first architecture to the second are:

- encapsulation/hiding

  All the operations and related types, even some which are not useful (like *Homothety* or *Similar*) to solve the problem, are encapsulated in the class. The operation *Kind_of* is not an isolated algorithm. It cannot be tested without being invoked by an instance of *Triangle*, the data structure of which is hidden. The only way to access it is by invoking selector operations defined in the interface, such as *Length_of*.

- inheritance

  The class *Triangle* inherits from the class *Shape*. *Triangle* can therefore rely on *Shape* for both its representation (data structure) and its behavior (operations implementation).

- operations binding

  The operations of the class *Triangle* are tightly bound: it is not possible to apply the operations *Length_of* and *Kind_of* to an object triangle before *Create* is invoked to create an instance. Hence, it becomes necessary to monitor the changes that take place in the features according to the operations. Therefore, testing a class requires writing scenarios to test the possible interactions of the operations onto its instances. Of course, exhaustive testing (i.e. writing all possible

scenarios, which is often infinite) is out of the question, therefore the tester will have to limit the number of scenarios while maximizing their efficiency. C. Turner describes a technique for writing scenarios in [26].

Besides these technical issues, another reason to consider the testing of object-oriented software different from the traditional testing is the move in the software life cycle. The traditional development life cycle (waterfall) is challenged by continuous development methods (prototyping). Testing does not occur anymore as the steady last step in the development, but as a step that occurs repetitively for every refinement of a software component. To be efficient, a tester has to write flexible test code and stress incremental testing, to make a maximum reuse of existing test code.

4. Problems introduced by object-oriented programming for testing

The three architectural issues raised in the previous section (encapsulation, inheritance, and polymorphism) will now be examined in more details.

4.1 Encapsulation

The notion of class is a blessing for testing, since this notion involves encapsulation. The test unit is clearly designed, which effectively simplifies the testing because the determination of a test unit becomes easier. The class can thus be tested in isolation of the rest of the system. i.e. the context in which the test is made does not affect the testing of the class.

4.1.1 Hiding

Hiding is a fundamental property of object-oriented programming. Programmers do not need to worry about the internals of a class, since they only use the interface to communicate with the objects. However, the resulting opacity hinders testing, since the coherence of the state of the object cannot be checked after invoking an operation of the class. In the presence of *hiding*, the only way to observe the state of an object is through its operations; there is therefore a fundamental problem of observability, as part of the testing relies on the tested software itself.

There are many workarounds to that problem.

The first and most obvious solution is to modify the tested class to add operations which provide the tester with visibility on the hidden features of the objects, e.g. test points [15]. This solution is usually not satisfactory since it is intrusive and it involves an overhead on the tested class. Moreover, it is not guaranteed that the class will have the same behavior in the presence of the tested code.

A refined approach is to define the added operations in a descendant class, which is derived from the tested class, and serve to test the derived class. This method captures the functionality of test points while being non-intrusive. However, this method is useless if the derived class does not have complete visibility on the state of the inherited properties. For example, the internal (called private) properties of a C++ class are not visible to its descendants.

Besides those two general methods, several programming languages support language-specific mechanisms to break encapsulation by providing either:

- family-related constructs

  C++ has intrusive friends subprograms to define operations that are not operations of the class but have visibility on all its features. The child units of Ada 95 are non-intrusive package extensions, which have a complete view onto the private part of their parent package.

- low-level constructs

  Smalltalk's inspectors and Eiffel's class Internal provide low-level operations to examine all the features of an object. These operations break encapsulation by accessing the physical object structure.

- unchecked type conversion

  If the type system of the programming language is weak, or if the language provides unchecked type conversion, it is possible to break encapsulation by writing another class, the data structure of which is a clone of the tested class except that all its properties are public and by casting instances of the tested class to instances of the clone class to access their features freely.

Of course, all those solutions are "hacks". The best solution is to accept encapsulation, and to base the test set selection on the results accessible from the interface of the class rather than to try to break into it. Thus, a test set should depend on the specification of the class only. This way, it can be reused (e.g. in regression testing), even if the internal properties of the class have changed.

4.1.2 Non-instantiable classes

On top of the problem of accessing the state of an object is the problem of testing non-instantiable classes. Non-instantiatable classes are classes from which instances cannot be created because they do not contain enough information.

There are three kinds of non-instantiatable classes: abstract classes, the implementation of which is not completely defined (and must be subsequently defined in derived concrete classes), generic (parameterized) classes (some components of which are unspecified to be more general, for instance the type of the items to be put in a class Stack), and mixin classes (abstract subclasses that can be derived from any base class to create a new class).

Since instances of those classes cannot be created, it is impossible to test them as is. However, since those classes have a specification, it is possible to define test sets (for black box testing) for them. Those test sets can then be exercised on the concrete descendant classes.

4.2 Inheritance

As mentioned in section 2, inheritance is a mechanism that allows for a class, the derived class, to inherit properties, features and operations, from one (single inheritance) or many classes (multiple inheritance), its base classes. The derived class can then be refined by modifying or removing the inherited operations, or adding new properties.

Since the derived class is obtained by refinement of its parent class, it seems natural to assume that a base class that has been tested can be reused without any further re-testing of the inherited properties. This intuition is however proved false in [22], with the help of Weyucker's axioms for adequate testing: some of the inherited properties need re-testing in the context of the derived class. Therefore, to take advantage of the testing already completed for the base class to test the derived class, i.e. to do incremental testing, and not to re-test the whole set of inherited properties, we must diagnose which of those properties need re-testing, i.e. the minimal set of properties the behavior of which is different from the parent class.

As the inheritance scheme differs from one language to another, there is no language-independent method to define this set. In the following discussion, we will examine the most commonly found schemes of incremental modifications. An algorithm that determines this minimal subset for the C++ programming language, assuming a white-box testing strategy, can be found in [11].

4.2.1 Strict inheritance

Strict inheritance is the simplest scheme of inheritance. A derived class is a strict heir of its parent class if it keeps the exact inherited behavior of its parent. The inherited properties cannot be modified (e.g. overridden); the derived class can only be refined by adding new properties.

Once more, despite intuition, strict inheritance involves re-testing of some of the inherited properties. As discussed before, the advantage of encapsulation is that clients do not have direct access to the data structure of the objects. Inheritance does however break encapsulation: the derived class has access to the features of the base class, and can modify them. While encapsulation builds a wall between the class and its clients, it does not prevent the derived class to mess up with the inherited features.

Thus, although the specification and the code of the inherited operation are the same in the base and the derived class, the increments in the derived class can lead to changes in the execution of the inherited operations: the added operations can have an impact on the state of the object, so that portions of code that were previously unreachable, and were thus not tested, may become reachable in the context of the subclass, and thus need testing.

4.2.2 Syntactical subtyping

The second, and most common, scheme of inheritance is syntactical subtyping, which only allows signature-compatible modifications. Besides the properties of strict refinement, syntactical subtyping allows for the redefinition (overriding) of the inherited properties (i.e. to give a new implementation to an inherited operation, to be used in the derived class). Overriding occurs when the behavior of an inherited operation is not appropriate in the context of the derived class.

Overriding does of course imply re-testing the overridden operation. If the programmer has felt the need to give a new implementation to an operation, the latter will not reproduce the exact behavior of the inherited code. Moreover, a side effect of overriding is that it also implies the re-testing of all the operations that invoke the overridden operation as part of their implementation, no matter if they are inherited from the class in which the overridden operation was first defined or in a latter subclass in the inheritance hierarchy: since those operations make use of an operation the behavior of which has been modified, their own behavior is also modified and they need re-testing.

In the example on figure 4, inspired from Interviews [16], a class Button has two operations, Redraw and Choose, the implementation of which makes use of Redraw. In the class PushButton, which is derived from Button, the operation Redraw is overridden, but not Choose. However, since the implementation of Choose makes use of an overridden operation, it will have to be re-tested, although its code is not modified.

4.2.3 Subclassing

Subclassing is a scheme of inheritance in which the derived class is not considered as a specialization of the base class, but as a completely new abstraction which bases part of its behavior on *part of* another class. This scheme is also called implementation inheritance.

The derived class can therefore choose not to inherit all the properties of its parent. As a consequence of subclassing, the test suite defined for the subclass must therefore be modified to suppress all references to the removed properties. The tester must also take care that no operation makes a call to a removed operation. This is easy to achieve in program-based testing.

Subclassing can also introduce other problems that will be discussed in the section on polymorphism.

4.2.4 Multiple inheritance

Multiple inheritance does not introduce problems other than the problem exposed above, unless the properties inherited from the various base classes “step on each others feet”.

```
with Coordinates; use Coordinates;
with Displays; use Displays;
package Buttons is

   type Button is tagged private;
   procedure Redraw (B: in out Button; C: in Coord);      -- (1)
   procedure Choose (B: in out Button);

   type PushButton is new Button with private;
   procedure Redraw (P: in out PushButton; C: in Coord);  -- (2)

private
   ...
end Buttons;

package body Buttons is

   procedure Choose (B: in out Button) is
      C: Coord := New_Coord(B);
   begin
      ...
      Redraw (Button'Class (B));   -- dynamic binding: the implementation of Redraw
                                   -- is selected according to the specific type
                                   -- of B (i.e. either (1) or (2)).
      ...
   end Choose;

end Buttons;
```

Fig. 4. Operation overriding in Ada 95

This case occurs when two or more base classes have homograph operations (i.e. with similar name and profile), and the language resolves the ambiguity of choosing by implicitly selecting which one of the homograph operation is inherited for the derived class (for example by a precedence rule like in CLOS [12]).

```
; definition of class first
(defclass first-class
   ()
   (first-slot) )

(defmethod a-method ((an-object first-class))
   (format t "Method of First"))

(defmethod first-method ((an-object first-class))
   (a-method an-object))

; definition of class second
(defclass second-class
   ()
   (second-slot))

(defmethod a-method ((an-object second-class))
   (format t "Method of Second"))

; definition of class third
; (multiple inheritance of second-class and first-class, nothing else)
(defclass third-class
   (second-class first-class)
   ())
```

Fig. 5. A case of multiple inheritance in CLOS

The homograph operation inherited from one of the base classes overrides the other homograph operations, even in the classes where those other operations have been defined. Therefore, it will be used for all calls of this operation, changing the behavior of operations that makes use of it.

In the above example (figure 5), a class third-class is built by deriving from first-class and second-class. Both first-class and second-class have an homograph operation a-method. In the class third-class, the operation first-method, which is inherited from first-class, and used to call a-method defined in first-class will call a-method defined in second-class. Therefore, any call to first-method with an instance of third-class will result in outputting "Second Method".

Note that in most object-oriented languages that support multiple inheritance (C++, Eiffel), such ambiguities must be explicitly resolved by the programmer himself.

4.3 Polymorphism

Polymorphism is the possibility for a name, i.e. a variable or a reference (called polymorphic name below) to denote instances of various classes. It is usually constrained by inheritance, in which these various classes must belong to a hierarchy of classes made up of a root class or its descendants. If the inheritance scheme is subtyping, the denoted objects all have at least the properties of the root class of the hierarchy. Thus, an object belonging to a derived class could be substituted into any context in which an instance of the base class appears, without causing a type error in any subsequent execution of the code. (This principle is known as Liskov's substitution principle [17]). Note that this hierarchy of classes can be as vast as the totality of all classes in the system if all classes are descendants of an unique root class (e.g. Smalltalk's Object, Eiffel's Any).

We will now examine different uses of polymorphism that can affect the correctness of a program and cause trouble to testing.

4.3.1 Undecidability of dynamic binding

Polymorphism brings undecidability to program-based testing. Since polymorphic names can denote objects of different classes, it is impossible, when invoking an operation of a polymorphic name, to predict until run-time, what the code is, that is about to be executed, i.e. whether the original or a redefined implementation will be selected. (This is called dynamic binding.)

Therefore, to gain confidence in an operation a statement of which is a call to a dynamically invoked operation, it becomes necessary to make assumptions regarding the nature of the polymorphic parameters, i.e. a "hierarchy specification" must be provided for each of their operations, which specifies the expected minimal behavior of their operation and all their possible redefinitions. This "hierarchy specification" is only a subset of the specification of the operation: the exact behavior expected from an operation is usually not wanted in its redefinitions (thus the need to override the unwelcome behavior with a proper one).

These assumptions are the assertions of the operation: the conditions that must be respected by every implementation of the operation (the original implementation and its redefinitions). Assertions can be refined for the specification of an overridden operation, but cannot be relaxed. (A mechanism to implement assertions is part of the Eiffel programming language [20].)

4.3.2 Extensibility of hierarchies

A similar problem occurs when testing (specification-based and program-based) an operation one or more parameters of which are polymorphic. As testing an operation consists in checking its effects when executed for various combinations of actual parameters, a test suite must ensure that all possible cases of bindings are covered (see figure 6).

However, given a polymorphic call or an operation with one or more polymorphic parameters, it is impossible to plan a test in which you check the operation with parameters of all the possible classes, because a hierarchy of classes is freely extensible: it is possible, at any moment, to add a derived class to a hierarchy, without even causing the re-compilation of the considered operation.

In this case too, the use of assertions can facilitate the task of the testers.

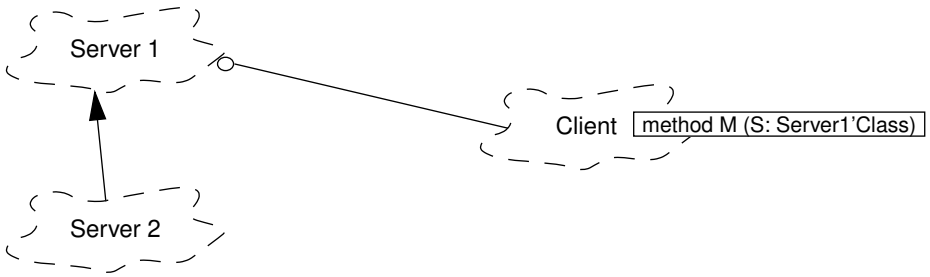

Fig. 6. retesting clients when adding subclasses of a server

4.3.3 Heterogeneous containers and type casting

Heterogeneous containers are data structures the components of which may belong to various classes, constrained in the same way as polymorphic names.

However, some objects do not have the same set of properties as the root class of the hierarchy to which they belong. To let those objects make full use of their properties, it is possible to cast the objects contained in the heterogeneous data structures to any class in the hierarchy.

This can lead to two common faults:

- an object being casted to a class to which it does not belong, and being therefore unable to select a feature or invoke an operation, because it lacks these properties.
- if the scheme of inheritance is subclassing, an object being not coasted to its class and being compelled to invoke a message that was removed for its class.

Since these typing faults can usually not be caught at compile-time, care must be taken when designing test suites not to overlook them.

5. A Theory of specification-based testing of object-oriented software

Specification-based testing is a technique to find errors in a program by comparing the results of the execution of tests against its specification.

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 test procedure (see figure 7) is usually a three steps process. It starts with a given requirement, usually to find faults in a program by comparing it against a specification, but other requirement include testing the robustness of the program, its performance,... It is decomposed in

- a test selection step, in which the test cases that will validate or not the requirement are generated,
- a test execution step, in which the test cases are executed, and
- a test satisfaction step, in which the results obtained during the test execution phase are compared to the expected results, this last step is commonly performed through the help of an oracle.

The results of this comparison give the result of requirement, yes if it is validated, no it is not, and undecided if the oracle was unable to analyze the result or if the test execution did not return a result.

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.

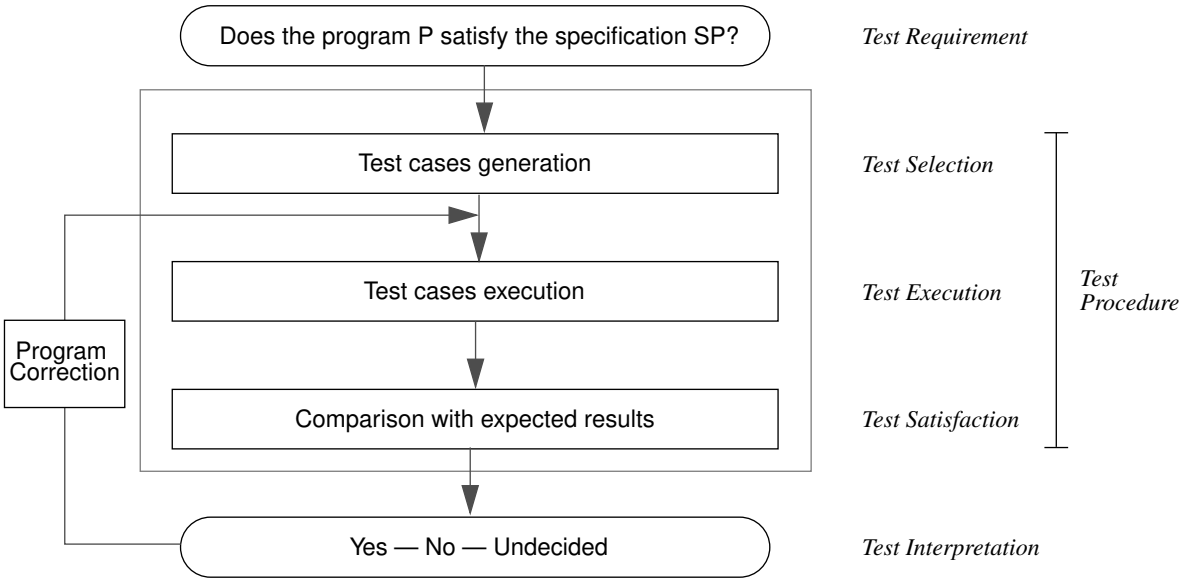

Fig. 7. The test procedure for specification-based testing

To select such a test set for object-oriented programs, we will apply an adaptation to object-oriented systems of the theory of testing developed by Bernot, Gaudel and Marre [2] for testing data types by using formal specifications. This theory is decomposed in two parts:

- *Test selection*

  The theory of test selection consists in selecting from a possibly infinite set of formulae, corresponding to all the behaviors of a program, a finite set of ground formulae which is sufficient to prove the property preservation of a program with respect to its specification. This infinite set of formulae is called the exhaustive test set. 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 (see figure 8). 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. Validity and unbias are necessary properties of the reduced test sets, i.e. after the application of the hypothesis, all unsatisfied tests should reveal program errors and no satisfied test should hide errors..

  Thus, as shown in figure 8, 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 selection 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

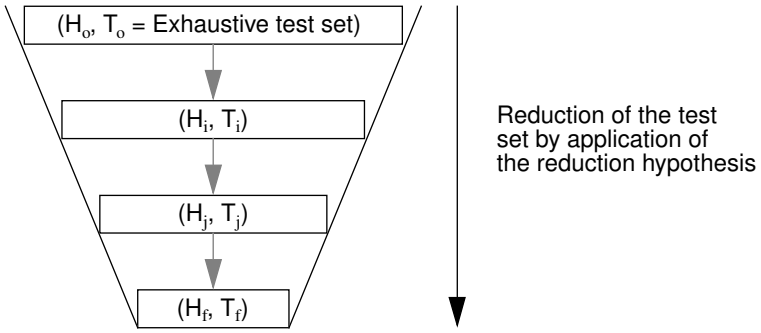

Fig. 8. Iterative refinement of the test context

- *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 [28]. 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 [5]. In object-oriented systems, this equivalence relationship can be difficult to build because of data hiding.

  A test set is decidable if, for each test, an oracle can be built to decide whether the test is satisfied or not. The best way to show the equivalence of two objects is to have a well-founded observation theory. 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 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.

5.1 Test selection

The adaptation required to this theory for the selection of test cases is of two kinds. First of all, we must study the sources of test sets, i.e. the specification of object-oriented software, from which the generation of the exhaustive test set is performed. Secondly, the reduction hypothesis must be enhanced, principally to take polymorphism into account.

5.1.1 Sources for test selection

The quality of the test cases is heavily dependent on the quality of the specification from which they are derived. There are several sources to build a test set, for dynamic testing (testing where the tested unit is executed, and the results are compared against the expected result). The main sources are:

- Models

  The test sets are generated from the models developed during analysis and design (object model, operation model, etc.). Those models are usually vague or incomplete, not to mention that their semantics is weakly defined and can lead to inconsistencies. However the foremost object-oriented modelling techniques (OMT/Rumbaugh, Jacobson, Booch, Fusion, etc.) use those kinds of models extensively. Object models have been used as sources by [23] with the OMT method, using simple testing techniques (range checks, boundaries analysis, equivalence classes). The expected results were entered by the tester as part of the test sets manually. (They do not appear in the models.) There was no inheritance, not subtyping/substitution in the described experiment.

- Specifications

  Class specifications are the main and most complete source of test sets, because these specifications are easily mapped to the tested unit, and do not require extra work to gather the information.

This self-containness makes it easy to generate test sets automatically. It is also easier to infer the expected results from the specification (than from models). Class specifications need not be formal. They can be written in natural language. However, the advantage of formal specifications is that it is easier to check their completeness and their consistency. [8][6] give an overview of the various formal specification languages for object-oriented systems. Several experiences have been made in testing with formal specification, a good summary of the state of the art can be found in [10].

- Programs

  While maintaining the idea of testing the behavior of the program, the idea behind program-based testing is to select the test inputs to ensure a maximum coverage of the control flow (paths, conditions, etc.) of the program text. This approach gives very good results in practice for specific types of errors. However, program-based testing is not perfectly suited for object-oriented software, because it does not support well incremental testing and because it requires more work in the case of multiple implementation of one specification —which happens frequently—, whereas a single test case is needed for specification-based testing.

- Random

  In random-based testing, the test inputs are generated randomly (regardless of the input domain). This leads to an easy and inexpensive test generation, which can give a good coverage of the whole input domain and good results in some specific cases (syntax checkers, etc.). However, this method is generally considered as weak, because random tests do generally not give a good coverage of the domain partitions. Random test generation for object-oriented systems has been studied in [13].

- Experience

  Finally, most programmers turn to (previous-)experiences based testing. This is the most used method. Catalogues have been created that archive those experiences (see for example the (C-based) catalogue in [19]). Automatic generation may be possible with the appropriate tool that try to find coincidences between the catalogue and the tested unit.

All those sources can serve to select tests, and are usually complementary: the errors caught by with one source cannot necessarily be easily found with another source. However, the only source that really fits the needs of our theory is the specification, because it is the one for which completeness and consistency is more easily obtained. Completeness is the key point for the generation of the exhaustive test set.

5.1.2 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 9). According to the kind of specification, the tests set can be the tests of all axioms of the specification, instead of all possible calls.

$$\text{Result} := \text{Object.Operation (Arguments)} \Rightarrow$$
$$\text{Final_State (Object)} \equiv \text{Expected_Final_State (Object)} \wedge$$
$$\text{Result} \equiv \text{Expected_Result}$$

Fig. 9. 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 10), 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.

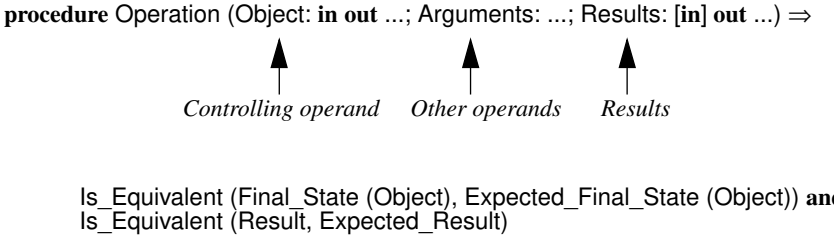

Fig. 10. Structure of a test (in Ada 95)

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 usually implies 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 (see figure 11).

5.1.3 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.

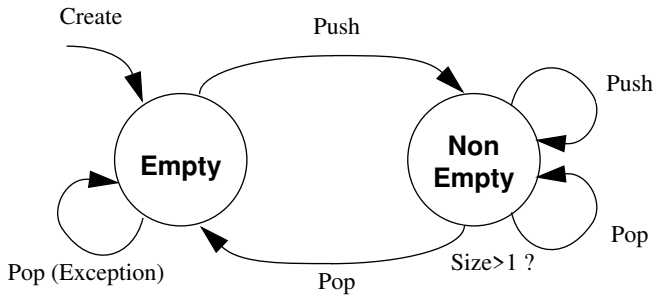

Fig. 12. The reduced set of states and transitions of an abstract stack

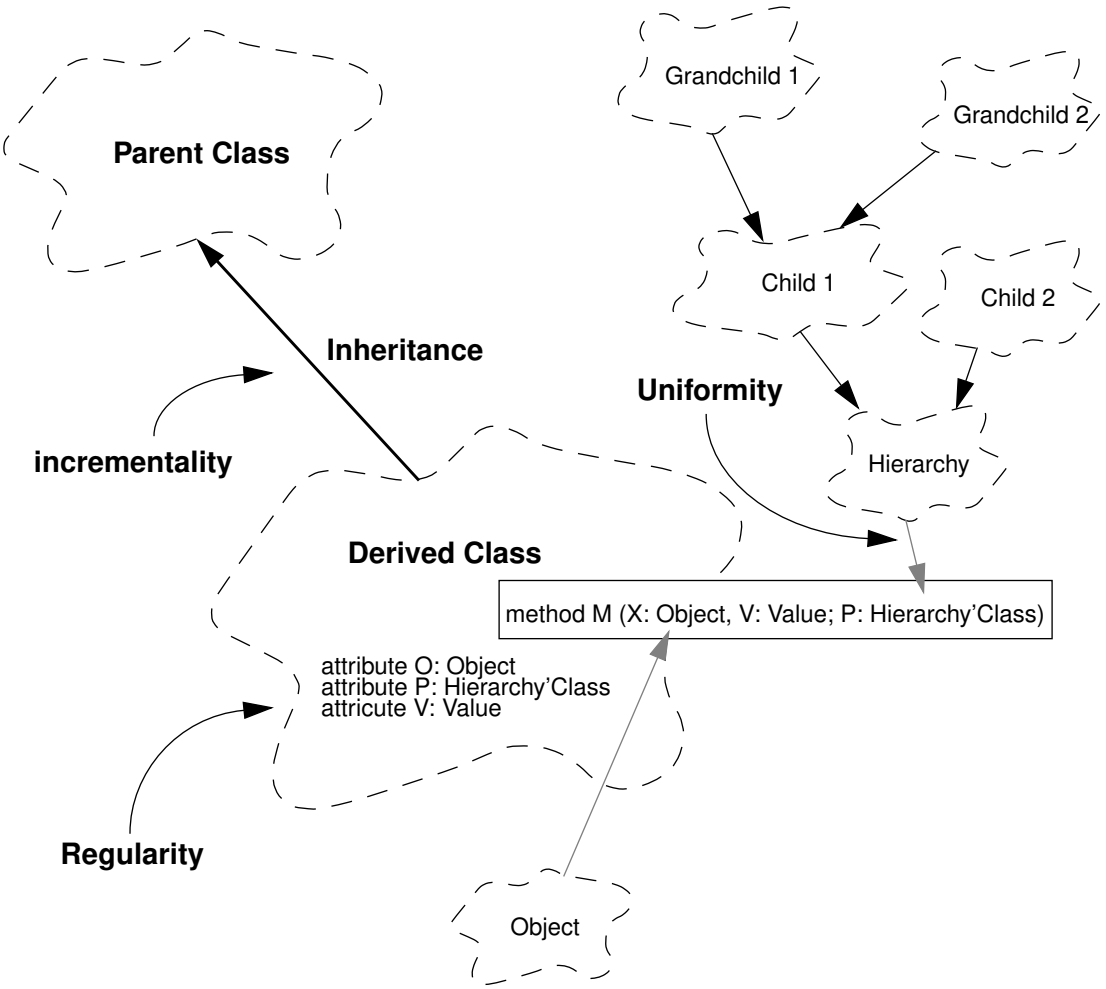

Fig. 11. Hypotheses for reducing the exhaustive test set

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.

5.1.4 The uniformity hypothesis

The uniformity hypothesis helps select and reduce test sets in the presence of polymorphic references and dispatching (i.e. dynamic binding). 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 polymorphic 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 13), in the case of the Stack, the type of the item has no interference with the behavior of the class, even if it is a polymorphic reference. In the case of the Alarm System described in the Ada 95 Rationale [1], 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).

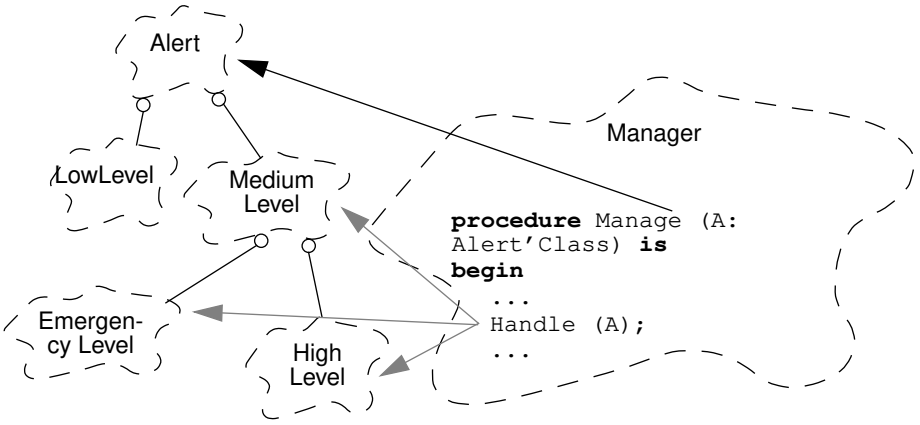

Fig. 13. A Manager for the whole Alert hierarchy

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*[3] 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 ([14], [18]) ensures that the base type and the derived type have the appropriate 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.

5.1.5 The incrementality hypothesis

The incrementality 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 incrementality hypothesis states that there is a complex, but strong relation between the testing contexts of a base 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

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

[3] this is not exactly the Ada subtype.

derived type (e.g. <Empty, Pop, Empty>), while in other cases, new test sets need to be selected (e.g. <Full, Push, Full>).

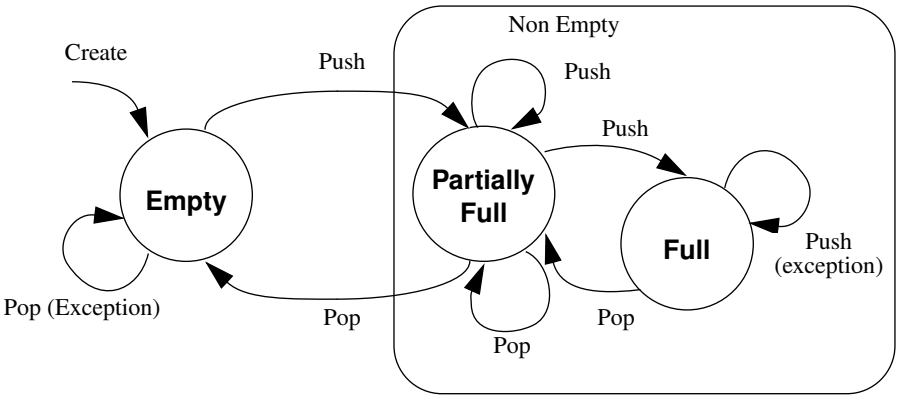

Fig. 14. The reduced set of states and transitions for an unbounded stack

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

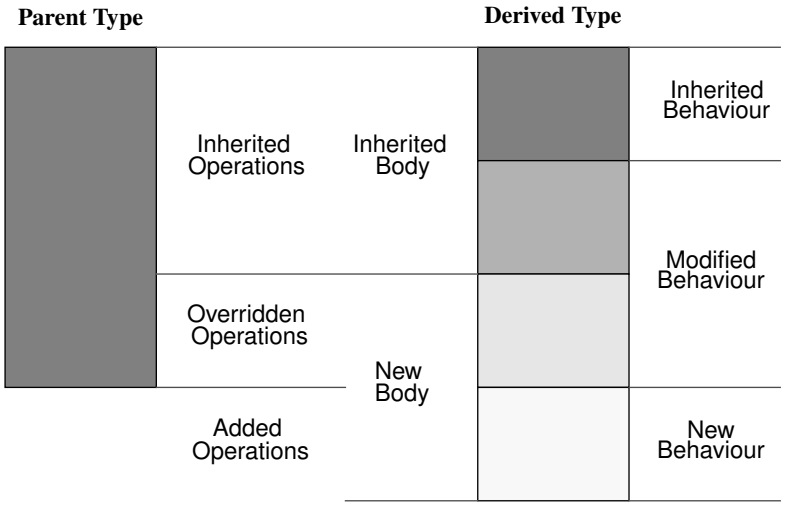

Fig. 15. Incremental testing

In Ada 95 several cases can occur when testing a derived type (see figure 15):

- 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 16, 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
   ...
   if Speed_of (Vehicle'Class (V))> 80.0 then
                        -- dispatching
   ...
end Move;
```

Fig. 16. 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.

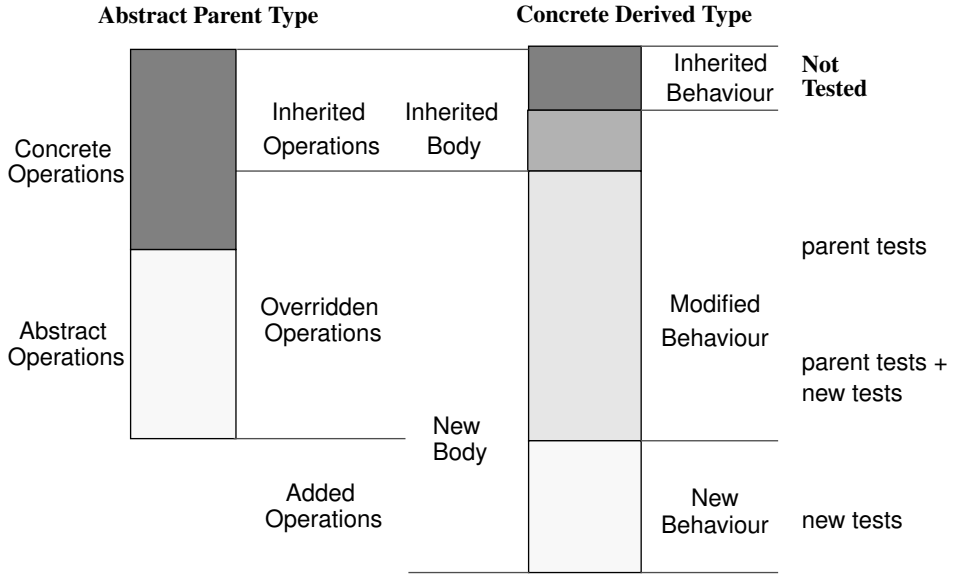

Fig. 17. Incremental testing for abstract classes

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.

5.2 Oracle generation

The oracle is a program that must decide whether the evaluation of formulae of the test set is successful or if this test set reveals errors. The evaluation is performed using the equality for functional approaches and traces for concurrency or state based approaches. The oracle cannot handle all possible formulae that are proposed as tests, this is why oracle hypothesis must be taken into account to limit the test selection to decidable test formulae. In the context of object oriented language, and for CO-OPN, the oracle must handle formula such as process logic or traces of method calls in order to take into account the state of the objects.

5.2.1 Oracle for algebraic specifications

Given the correspondence of the program constructs and the algebraic specification language, we are able to build an oracle. For instance, in the case of exceptions, both sides of the axiom equality must be encapsulated in order to capture the exception and translate it in a boolean value used to establish the validity of the axiom (the axiom is defined using a boolean value instead of an exception). For functions or procedures, the oracle is a direct translation of the axioms into conditional statements. The problem of the oracle is solved by building the equality of non-observable sorts on the equality defined on the primitive sorts, rather than by direct comparison between values of the tested abstract data type, which may be non available. For an equality between two non-observable values, a composition of operations the result of which is observable (an observable context), is added "on top of" each member of the equality. Therefore, the observation of a non-observable equality is performed through a set of observable equalities corresponding to the addition of observable contexts "on top of" the right and the left member of the non-observable equality.

5.2.2 Oracle for object specifications

Similarly we have to build an oracle for the program under test which calls the methods following the events appearing in the test formulae. Problems are encountered when branching have to be preserved in the formulae, it means that we need to linearize the formulae. The result of the evaluation is then that the formula is satisfied or not by the program. Correct programs have the same answer as the specification to the test formulae.

5.2.3 Oracle for Ada 95 programs

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 bisimulation or traces. This approach is for example used in the ASTOOT approach [7].

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 [15], except that this approach is non intrusive. This satisfies the requirement expressed sometimes that a tested unit should not be re-compiled after having been tested. No re-compilation 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. Conclusion

The benefits from using object-oriented methods can be substantial. However, these methods do not guarantee correctness. Thus it is essential to develop a good testing methodology such that some of the current problems can be alleviated.

Although the test of object-oriented software has recently been recognized as an important subject of research, all dimensions of object-oriented programming relevant to testing have yet to be cov-

ered. Previous work has focused on identifying the basic unit of testing, the class, and finding the specific problems caused by encapsulation and inheritance. We further analyzed the problems that have already been identified. We also addressed other dimensions such as hierarchies and analyzed the problems that can arise from polymorphism with respect to testing.

We have shown the bases of a theory for the specification-based testing of 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.

The key points of the method presented in this paper are the use of subtyping in the uniformity hypothesis and the further reduction obtained by introducing incremental testing from a parent to a subtype. We plan to continue our work on this subject and particularly to express formally our adaptation of the testing theory.

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.

Therefore, it has both a theoretical and a practical value, and we can hope that it will eventually lead to a preeminent contribution to the validation and verification of object-oriented programs.

7. Reference

[1] J. Barnes, B. Brosgol, K. Dritz, O. Pazy, and B. Wichmann. *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, MA, USA, Feb. 1995.

[2] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, Nov. 1991.

[3] B. Birss. Testing object-oriented software. *SunProgrammer - The Newsletter for Professional Software Engineers*, 1(3):15–16, Fall/Winter 1992.

[4] G. Booch. *Object-Oriented Design with Applications*. Benjamin-Cummings, 1991.

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

[6] M. Dodani. The many faces of formal methods for specifying object-oriented software. *ROAD*, 1(6):36–40, Mar.-Apr. 1995.

[7] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Apr. 1994.

[8] H.-D. Ehrich, M. Gogolla, and A. Sernadas. Objects and their specification. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification - 8th Workshop on Specification of Abstract Data Types*, volume 655 of *Lecture Notes in Computer Sciences*, pages 40–63, Douran, France, Aug. 1991. Springer Verlag.

[9] S. P. Fiedler. Object-oriented unit testing. *Hewlett Packard Journal*, 40(1):69–74, Apr. 1989.

[10] M.-C. Gaudel. Testing can be formal, too. In *Proceedings of TAPSOFT 95*, 1995. (to be completed).

[11] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceeding of the 14th international conference on Software Engineering*,

pages 68–79, Melbourne, Australia, May 11-15 1992. ACM Press.

[12] G. Kiczales, J. des Riviers, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[13] S. Kirani. *Specification and Verification of Object-Oriented Programs*. PhD thesis, University of Minnesota, Dec. 1994.

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

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

[16] M. A. Linton, P. R. Calder, and J. M. Vlissides. Interviews: A C++ graphical interface toolkit. Technical Report CSL-TR-88-358, Stanford University, July 1988.

[17] B. Liskov. Data abstraction and hierarchy. In L. Power and Z. Weiss, editors, *OOPSLA '87 Addendum to the Proceedings*, pages 17–34, Orlando, Florida, May 1987. ACM Press. Published as Volume 23, Number 5 of SIGPLAN Notices.

[18] B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. In A. Paepcke, editor, *OOPSLA '93 Conference Proceedings, Washington, DC*, volume 28, pages 16–28, Sept. 26 - Oct. 1 1993.

[19] B. Marick. *The Craft of software testing: Subsystem testing including object-based and object-oriented testing*. Series in Innovative Technology. Prentice Hall, 1994.

[20] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[21] G. J. Myers. *The Art of Software Testing*. Business Data Processing: a Wiley Series. John Wiley & Sons, 1979.

[22] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, Jan. 1990.

[23] R. M. Poston. Automated testing from object models. *Communications of the ACM*, 37(9):48–58, Sept. 1994. Special issue on testing object-oriented software.

[24] P. Rook, editor. *Software Reliability Handbook*. Elsevier Applied Science, 1990.

[25] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[26] C. D. Turner and D. J. Robson. The testing of object-oriented programs. Technical Report TR-13/92, Computer Science Division, SECS, University of Durham, England, Nov. 1992.

[27] D. Ungar. Position paper for the OOPSLA '93 workshop on object-oriented testing. Sept. 1993.

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