# Constructing Self-Testable Software Components

Eliane Martins
eliane@ic.unicamp.br

Cristina Maria Toyota
cristina.toyota@tecban.com.br

Rosileny Lie Yanagawa
lie@cpqd.br

*Institute of Computing (IC)*
*State University of Campinas (Unicamp)*
*Campinas – SP – Brazil*

## Abstract

*Component-based software engineering techniques are gaining substantial interest because of their potential to improve productivity and lower development costs of new software applications, yet satisfying high reliability requirements. A first step to address such high reliability requirements consists in reusing reliable components. To merit the attribute "reliable", a component should be extensively validated. As far as testing is the technique most commonly used for validation, this means that reusable components should well tested. For tests to be applied efficiently and on time, a component should be testable. This paper presents an approach to improve component testability by integrating testing resources into it, and hence obtaining a self-testable component. A prototyping tool, Concat, was developed to support the proposed approach. The tool is intended for OO components implemented in C++. Some preliminary results of an empirical evaluation of the fault detection effectiveness of the proposed testing approach are also discussed.*

**Keywords**: component testability – design for testability – self-testable component - OO testing

## 1. Introduction

Software components are gaining substantial interest in the development of new applications. The main motivation is the possibility to reduce development time and cost by using components developed by third-parties or even available in the commercial market (the so called Commercial-Off-The-Shelf or COTS components). A first step for a successful reuse is to use reliable components. To guarantee such quality, among others, a component should undergo extensive validation. This means that components should be well tested, as far as testing is the technique most commonly used for validation.

As pointed out by different authors [9, 41], a reusable component should be tested many times: by their producers, during development or maintenance, and by their consumers, every time they are reused. For that reason, testing should not be the cause of a reduction in productivity in component-based development. The use of testable components is one of the means to efficiently achieve a well-tested software, yet guaranteeing the high reliability required.

Software testability encompasses all aspects that eases software testing, from the quality of its specification, design, code, and tests, to the availability of test support. Design for testability techniques have long been using to improve the testability of hardware components. These techniques, used in conjunction with the self-testing concept, have lead to built-in self-testing (or BIST) hardware components, which embed test pattern generation capabilities.

In this paper we present an approach for the construction and use of self-testable components. A self-testable component contains, in addition to its implementation, a specification from which test cases can be derived. Our concern in this study was OO components that are a unique class. A prototyping tool was developed to support some activities of the proposed approach to show its feasibility. This prototyping tool, named Concat [33], supports the construction and use of self-testable components implemented with the C++ language.

The text is organized as follows: next section presents some basic concepts which are used in the remainder of the text. Section 3 describes the proposed approach. Section 4 presents preliminary results obtained with the empirical evaluation carried out to assess the effectiveness of the test selection technique used. Section 5 compares our approach to some related work. Section 6 concludes the text, presenting also some future work.

## 2. Background

In this section we present an overview of the concepts and terminology used throughout this paper. The section ends with a



0-7695-1101-5/01 $10.00 © 2001 IEEE

brief description of a self-testable component.

## 2.1. Components

There are different definitions for components in the literature. We will consider here the one from C. Szyperski [32, ch4.1.5]: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"*.

A component is then a reusable software artifact. In OO context, which is our concern here, the basic reusable unit is a class. A class can be reused in different ways [6, c11.1]: by inheritance (as is the case with abstract classes), by parameterization (as is the case with generic or template classes) or by composition (an attribute is declared as a class).

Whatever the reuse mechanism, adequate testing from both producer and consumer is required for reuse to be successful. Testing aspects are described next.

## 2.2. Software testing objectives and activities

Software testing is the execution of a code in the presence of pre-selected inputs with the goal to reveal faults [27]. A software fault (or bug) is introduced into the software due to errors committed by developers during the development process. The execution of a faulty software may lead to the occurrence of failures, in which case the delivered services no longer complies with the specification. When performing testing the following issues should be considered:

- The testing level: software systems should go through at least three testing level [4, 41]: (i) **unit testing**, in which individual components are tested in isolation; (ii) **integration testing**, in which the subsystems formed by integrating the individually tested components are tested as an entity; and (iii) **system testing**, in which the system composed by already tested subsystems is tested as a whole. Our focus is the unit testing of components, as this is a first step a user should made before using a component in an application.
- The test criteria: test input selection is generally guided by *criteria* [40] derived from an abstract representation of the software (*test model*). They define the elements of the test model that should be covered by the tests. Most commonly used test models represent either the implementation or the specification of the software system. When a test model represents an implementation, we have implementation-based (or white-box) testing techniques. A criterion in this case might be: cover all nodes of a control flow graph. When a test model represents a specification, we have specification-based (or black-box) testing techniques; a criterion in this case might be: all transitions coverage on a finite state machine, in case this is used as a test model. In this study we used specification-based testing since our purpose is to allow a component's consumer to test whether the component addresses each of its required features. The test model used as well as the criterion for test generation are given in section 3.2.
- Test case design: a test case consists of test inputs and the expected outputs. Test inputs may be obtained according to the testing criteria presented above. The mechanism to produce the expected outputs is the *oracle*. Various approach exist for oracle development (see [6, c18] for a good survey on this subject). In this study the oracle is based on the component's contract. Contracts establish what the consumer should do to use a component, and what the producer should implement to provide the services required [31, c5.1.3]. The design-by-contract method [25, 26] is commonly used to specify contracts in the form of assertions, or logical conditions. For OO components, assertions can be associated to methods (pre and post conditions) as well as to a class (class invariant). Assertions are used in this study as a partial oracle, as will be presented in section 3.3.
- The test support: it may be necessary to define support for test execution. Among other things, this support comprises *test drivers* and *test stubs*. A test driver (or simply driver) activates the component under test, providing it with the necessary inputs and other information necessary for its proper functioning. A stub is a partial, temporary implementation of a component whose services are used by the component under test. The use of a formal specification model may be useful for the automatic construction of drivers and stubs. In this study we use the component specification to derive test drivers, as is explained in 3.4.

Testability is relative to the ease with which the activities mentioned above are executed, and is an important quality factor since a component should be tested many times. These aspects are presented next.

## 2.3. Components testability

According to IEEE std. 610.12 [23], testability can be defined as: "(i) *the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met*. (ii) *the degree to which a requirement is stated in terms that permit establishment of test criteria and the performance of tests to determine whether those criteria have been met*". This definition is more concerned with the ease to establish test criteria and to have them met, so it can be viewed as a measure of how hard it is to satisfy a particular testing criterion.

Another definition of testability states it in terms of the easy with which faults are revealed during testing, that is, the capacity of a system to fail during testing, indicating the presence of faults [36]. So testability is viewed as a measure of the likelihood that a system hides faults during testing.

As pointed out by different authors [8, 18, 19, 28], a testable software might possess a set of attributes, among them, observability, controllability and understandability. Roughly speaking, the more we can observe the states and outputs of a component, the more we can control it during testing; and the more information we have about a component, more easy it is to effectively testing it.

From the above it can be concluded that embedding inside a component its specification and other facilities allowing its monitoring and testing, is a way to improve components testability. This is especially true for OO software, whose intrinsic characteristics such as encapsulation and information

hiding reduce both observability and controllability. That is the reason why some studies proposed the use of built-in test capabilities to obtain a self-testable component. This is explained in the item that follows.

### 2.4. *Self-testable components*

Design for testability (DFT) techniques have been using for integrated circuit (IC) engineers to reduce test cost and effort yet achieving the required quality level. With DFT techniques, extra pins are added to an IC to allow access to internal circuit points during testing, thereby enhancing controllability and observability [KBJ+96]. The built-in test (BIT) approach is a DFT technique that introduces standard test circuits and pins that allows a card or an IC to be put in test mode, and also to transmit test inputs and capture the outputs. An extension to that approach consists in integrate inside the IC mechanisms for test inputs generation, which avoids the use of an external tester. This is the built-in self-test concept, or BIST.

Some authors have proposed the use of these hardware approaches to improve software testability. D.Hoffman proposes an approach [21] that augments a module's interface with testing features – these are the access programs, corresponding to the test access ports (TAP) used in BIT approaches in hardware. Integrating assertions to the source code is also proposed to improve controllability and observability. Test suites can be the same used in previous testing (since the purpose is mainly regression testing), but they can also be manually generated. Test suites should be described in a specific notation from which drivers are automatically generated. The advantage with respect to the hardware is that testing features may not be part of the final version: its insertion can be controlled by the use of compiler directives.

R.Binder adapts this approach to the OO context, proposing the construction of **self-testing classes** [Binder94]. A self-testing class is composed by the class (or component) under test (CUT) augmented with built-in test (BIT) capabilities and a test specification. With the BIT capabilities it is possible to: access test facilities, control and observe an object's internal state as well as monitor intermediate results by the use of assertions. The test specification (t-spec) is used for test generation and also as a test oracle.

The author proposes different architectures for a self-testing class. In this study we use the one that is based on a driver generator, which uses the information in the t-spec to generate a specific driver. The specific driver then activates the CUT, apply the test inputs and analyses test results. Our aim was show the feasibility of the approach . Next section presents how we implement such approach.

## 3. Proposed Approach

### 3.1. *Methodology*

The methodology is composed of two parts; one is performed by the component producer and the other by its consumer.

The component producer perform three tasks for developing a self-testable component:

- Construct the test model.
- Develop the t-spec from the test model and insert it into the component source code.
- Instrument component source code to introduce built-in test mechanisms.
-

To use a self-testable component, a consumer should perform the following tasks:

- Generate test cases based on the t-spec.
- Compile the component in test mode.
- Execute tests.
- Analyze the results obtained, to determine in what extent does the component provide the specified services.

The Concat tool [33] was developed to provide the built-in test interface and also to support test generation. Its main purpose was to demonstrate the feasibility of the proposed approach, as well as to validate the test generation strategy. The remainder of this section details the above tasks.

### 3.2. *Test specification construction*

In the context of this work we opted for a specification-based testing technique, so the test model represents a component's specification. This presents various advantages: (i) integrating the component's specification to it improves understandability, thereby increasing the component's testability; (ii) the specification can serve not only for test case generation, but also for oracle development; (iii) test selection is, to a certain extent, implementation language independent, which allows tests to be generated for abstract classes, for example, to be later incorporated to a subclass test suite; (vi) updating the specification when the component is modified becomes easier when the specification is integrated with the component; (vii) the specification quality can be improved, since incompleteness, ambiguity and inconsistency can be detected by the tester and then removed.

It is worth noting that the fact that we are concerned only with specification-based testing does not mean that we consider other tests useless. On the contrary, we assume that a component has been extensively validated before deployment. Embedded test facilities are intended mostly to facilitate tests when the component is reused or altered.

According to Beugnard et al [1, c.5, 3], a component specification should describe its interface and behavior. The interface description includes input and output features and their related value space (domain); in OO context, this comprises methods signatures: type and order of arguments, return type). Behavior description comprises not only the expected outputs in response to messages sent to an object but also non-functional aspects such as time and space efficiency, safety, security, among others.

Different types of specifications were used for OO testing. For example, Doong and Frankl present the ASTOOT approach [13], based on an algebraic specification, where the operations on abstract data types are described in terms of axioms. Another model commonly used is based on finite state machines [7, 22, 24], as they are adequate to represent an object's behavior. Barbey et al used a language, COOPN/2 (Concurrent Object Oriented Petri Nets), based on a combination of Petri nets and algebraic models [2]. These models are useful to describe the functional requirements. For non-functional requirements, however, less work has been done at that moment. One notation developed to express such requirements is NoFun [17], which

allows the representation of software attributes such as: time and space efficiency, reusability, maintainability, reliability and usability.

In this work we used the transaction flow model (TFM), defined by B.Beizer for concurrent systems testing [4; 5], to represent functional behavior aspects. A transaction is a unit of work seen from a user's point of view. A transaction is a sequence of operations (or tasks or else, activities) that can be performed by a human, by a system or a device. S.Siegel adapted this model for the context of unit testing of a class [29, ch.11]: in this case, though, a transaction describes an allowable sequence of method invocations from creation to destruction. The TFM represents then the different ways an object can be created, the different tasks it can perform and the different ways that can be used to destroy it. A TFM is represented as a directed graph; the nodes represent public features (attributes or methods) of the class, encapsulating all references to other objects as well as internal features. A link connect two nodes A and B if task A is immediately followed by task B. Therefore an individual transaction is a path through the TFM from birth to death of an object.

This model can be obtained from the use cases defined during requirements specification phase, so it is useful for our validation testing purposes. For objects presenting a finite set of allowable methods sequences, this model is useful. Our main reason to use such model is that it scales up easier than finite state machine models, which are more commonly used in OO testing. Therefore, it can be used for components having more than one object (or even other sub-components), as it can show the sequencing of activities performed by several objects as well. Another strength of this model is its capability to represent concurrent behavior, but this feature was not considered in our studies yet.

As an example, suppose the class Product in Figure 1 representing a product in the stock control system of a warehouse, in C++ notation. The product is obtained from a Provider, which is another class of this system that does not matter for this example, so it is not shown here. Suppose the use case scenario that follows, relative to the addition of a new product into the stock database: 1. Create a a Product object. 2. Obtain data about this product from the database. 3. Remove the product from the database. 4. Destroy the object.

Figure 2 shows the TFM for that class, where the path corresponding to the use case described is highlighted. After the development of the test model, the t-spec representing it is created and integrated to the component under test.

| | |
|---|---|
| Class Product { | /* Update methods */ |
| int qty; | void UpdateName( char* n); |
| char *name; | void UpdateQty(int q); |
| float price; | void UpdatePrice(float p); |
| Provider *prov; | void UpdateProv(Provider* prv); |
| public: | /* Access method */ |
| Product( ); | void ShowAttributes( ); |
| Product(int q; char* n; float p; Provider* prv); | /* Insert/Delete from database */ |
| Product(char* n); | int InsertProduct( ); |
| ~Product( ); | Product* RemoveProduct( ); |

**Figure 1. Example class Product.**

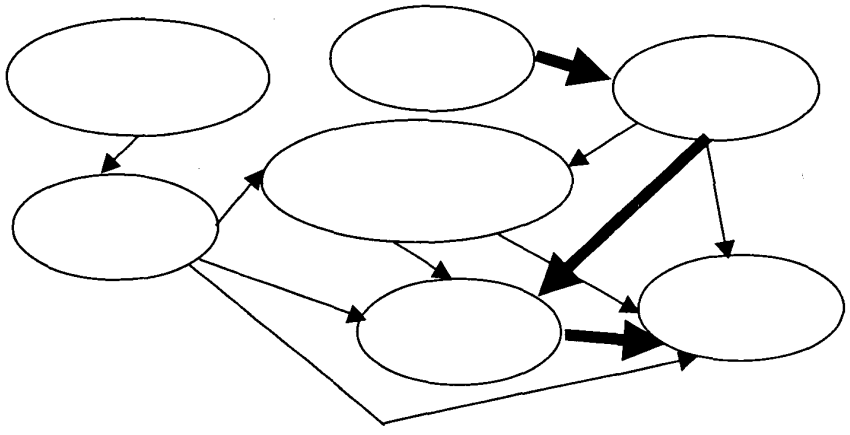

**Figure 2. TFM of class Product. The path corresponding to the example scenario is highlighted.**

| | | | |
|---|---|---|---|
| **Class** ( 'Product', | | **Parameter** ( m5, | // identifies the method |
| No, | // indicates whether the class is abstract or not | 'n', | // parameter name |
| <empty>, | // indicate that the class has no superclass | String, | // parameter type. Same as for attributes |
| <empty>) | // list all source files needed to compile the class | ['p1', 'p2', 'p3'] ) | |
| **Attribute** ('qty', | | ... | |
| range, | // allowable types: range, set, string, object, pointer | | // Description of the test model: |
| 1, | // for range types, indicates the lower limit | **Node** (n1, | // node identifier |
| 99999) | // for range types, indicates the upper limit | No, | // Starting node? |
| ... | | 1, | // number of outgoing edges |
| **Method** ( m1, 'Product', | // method identifier and name | [m1, m2] ) | // list of methods that constitute the node |
| <empty>, | // return type | ... | |
| constructor, | // method category relative to test reuse | **Edge** (n1, | // starting node |
| 0) | // number of parameters | n4) | // ending node |
| ... | | ... | |

**Figure 3 – Test specification ( t-spec) format.**

```
class BuiltInTest {
public:
BuiltInTest ( ) { }
~BuiltInTest ( ) { }
virtual void InvariantTest ( ) { }
                    // checks class invariant by calling the macro defined in the tool's macro library (ClassInvariant)
virtual void Reporter (FILE *logfile) { }
}                   // stores object state into a log file which can be used for post test analysis
```

**Figure 4 – Format of the BuiltInTest class.**

The t-spec corresponding to the example model is presented in Figure 3. This specification contains also a description of the component's interface, as shown in the figure.

For further details concerning this model, the references [4, ch. 4, 5, ch.6, 29, ch.11] can be consulted. Examples from models created in the context of this project can be seen in [32, 42].

### *3.3. Class instrumentation*

The instrumentation is extra software that is introduced into the class to increase its controllability and observability (c.f. 2.3). The instrumentation, also designated as built-in test (BIT) capabilities, comprises [8]: assertions, a reporter method and a BIT access control, as explained in 2.4. These capabilities are available through an abstract class, BuiltInTest, presented in Figure 4. This superclass contains the interfaces of two methods: InvariantTest and Reporter, and it was created to guarantee a built-in test interface independent from the target class interface. The target class, which is part of the component under test, inherits these capabilities, that should be redefined by the user.

A set/reset method could also be defined, to set an object to a predefined internal state, independent of the object's current state. This kind of method is not used in this study since the test of each transaction sets the object to a initial state (by using one of its constructors) and terminates by destroying it.

Concat implements the assertions relative to class invariant and methods pre and post conditions as macros, as shown in Figure 5. These are used to check whether the state of an object during a test session is valid or not. As they are useful for error detection, they serve as a partial oracle; manually derived oracles are also used in complement.

The BIT features can only be accessed if the class is in test mode, which is set by the user through BIT access control capability. This control capability prevents the misuse of BIT services; for the moment it consists in a compiler directive which includes or excludes BIT capabilities.

```
// Concat's macro library: contains the macros
// necessary for introducing assertions into a class.
// The user should provide the logic expression which
// defines the predicate to be evaluated.

#define ClassInvariant ( exp ) \
if ( ! (exp))
throw "Invariant is violated!";

#define PreCondition ( exp ) \
if ( ! (exp))
throw "Pre-condition is violated!";

#define PostCondition ( exp ) \
if ( ! (exp))
throw "Post-condition is violated!";
```

**Figure 5 – Macros used for assertion definition.**

### *3.4. Driver generation*

To use a self-testable component a test infra-structure is necessary which support the following functions: test driver generation, test history creation and maintenance, test retrieval, test execution, test result checking and test report. The current implementation of the Concat tool supports the first three functions; the test selection strategy used for that purpose is presented in the first item. Test history creation, maintenance and retrieval is partially implemented, and is the subject of the second item of this section. The user manually performs the other functions.

**3.4.1. Test selection strategy.** Test selection is entirely performed by the *Driver Generator*, implemented as part of the test-infra-structure provided by Concat [33]. The *Driver Generator* creates test cases according to the transaction coverage criterion that requires exercising each individual transaction at

least once. In other words, each test case exercises a path containing sequences of methods corresponding to the creation, processing and destruction of an object. Although being the weakest criterion among the ones presented in [5, c.6.4.2], nevertheless it is useful to reveal faults in transactions, specially those used less frequently, such as error-recovery transactions, for example. Since it is a specification-based test criterion, it is also useful to find situations that the designer did not consider, or transactions that are missing.

Values of input parameters for each method are also generated, by randomly selecting a value from the valid subdomain. Currently, this is implemented only for numeric types and strings, using information about range or set of possible values contained in the t-spec, as shown in Figure 3. Structured type parameters (including objects, arrays, and pointers) must be completed manually by the tester. For template classes, it is necessary that the tester indicate a set of possible types that he/she wants to use to create an instance of that class.

Figure 6 shows the structure of a test case. Each test case is implemented as a template function in C++, to allow its reuse when testing a subclass, as explained in next item.

```
// test cases are defined as template functions
    template <class ClassType>
// a test case is named by the Driver Generator as
// TestCase<id number>
    void TestCase0 (ClassType* CUT) {
// create variable that contains the current method
// name, for documentation purpose only
        char* CurrentMethod = new char [30];
// opens the log file
        ofstream LogFile("Result.txt", ios::app);
        if (!LogFile) cout << "Error opening log file! \n";
        else cout << "Log file created! \n";
// calls methods that are part of the exercised transaction;
// class invariant is checked before a method is
// called and after its return.
    try {
        CUT -> InvariantTest( );
        CurrentMethod = "Method1(321, 594, \"Mary\")";
        CUT -> Method1(321, 594, "Mary");
        CUT -> InvariantTest( );
        LogFile << "TestCaseTC0 OK! \n";
        LogFile.flush( );
// calls reporter method to store the object's internal state
        CUT -> Reporter ("Result.txt");
        LogFile << "\n";
        LogFile.close( );
        delete CUT;
    }
// in case an exception is raised, the name of the called method is
// stored in the log file
    catch (Error& er) {
        LogFile << "TestCaseTC0 \n";
        LogFile.flush( );
        Cout << ". . .";
        LogFile << er.msg << "\n"
        LogFile << "Method called: " << CurrentMethod << "\n";
        LogFile.flush( );
        CUT -> Reporter ("Result.txt");
        LogFile << "\n";
        LogFile.close( );
    }
        . . .
}
```

**Figure 6 – Example of test case format.**

The methods are called inside a try-block, so that an exception generated when an assertion is violated could be captured and treated by the driver. The (specific) *driver* is an executable test suite. Therefore, test cases can be used in different test suites. A test suite is considered as "executable" after being completed with the values of structured parameter types as well as any global data and stubs. Figure 7 shows an example of an executable test sequence.

```
# include <iostream.h>
# include "CUT.cc"              // file containing the instrumented
                                // class under test
# include "TestSuite.cc"        // file containing the test suite
# define TestMode               // test mode selection

    void main ( ) {

// create instance of the CUT, passing it as parameter
// for the test cases
        CUT* obj0 = new CUT;
        TestCase0 (obj0);

            . . .
        CUT* objN = new CUT;
        TestCaseN (objN);
}
```

**Figure 7. Executable test suite structure.**

3.4.2. **Tests reuse.** Another issue that should be taken Into account is the reuse mechanism for the components test resources. Since in this study a component is supposed to be a unique class, reuse can occur by inheritance, by parameterization or by composition, as explained in 2.1. In the last case, the class is not modified: its instances are used as attributes or parameters of methods. In this case, test resources can be reused without modifications. When using a generic class, all the user should do is to indicate which parameter types are to be used during tests, as mentioned in the previous section. So, the concern here is how to reuse test resources when subclasses are created.

Harrold et al. [20] proposed an incremental class testing technique based on the inheritance hierarchy. Each class possesses a testing history that associates each test case with the feature it tests. To generate tests for a subclass, its parent's testing history is incrementally updated to reflect differences from the parent, i.e., modified (redefined) or newly defined features. With this technique it is possible to identify new features of a subclass for which new test cases must be generated, as well as the modified one's that must be retested. In this technique, inherited features that are not modified in the context of the subclass need retest only if they interact with modified or newly defined features. The technique assumes a C++ language model, but imposes some constraints on inheritance mechanism: (i) only single inheritance is considered, that is, each class has only one parent; (ii) modifications to an inherited method cannot alter its signature (i.e., a modified method contains the same argument's list of the parent's); and (iii) the visibility of a feature in the subclass is at least as restrictive as in the parent class, and possible types of visibility are: private (visible only in the parent class, hidden in the subclass), protected (visible in the parent and in the subclass) and public (visible to all).

We used this approach in our study with a slight modification: instead of associating a test case with a class

feature (attribute or method), we associated it with a transaction. In this way it is possible to reuse test cases generated from a parent's transaction: if the transaction in the subclass contains only methods inherited without modification (except for the constructor and destructor methods, which for this reason are not part of a test case), there's no need to regenerate the test case for that transaction. In case an attribute is modified, the methods using it are considered as modified, since we are supposing a further constraint: attributes are not part of a class's public interface, being accessible only through methods. Of course, we are also assuming that specification changes imply that the tester updates assertions and t-spec.

## 4. Empirical evaluation

This section presents results of the preliminary evaluation of the fault revealing effectiveness of the test selection strategy. For this empirical evaluation we used a class from the Microsoft Foundation Class (MFC) library, CObList, which implements a linked list, and one derived class, CSortableObList, obtained through the Internet, which implements an ordered linked list. These classes were chosen because: (i) the classes in MFC library already contain assertions, and (ii) this library was used by a real-word application that was also used as a case study [42].

To assess fault-revealing power of test sets generated according to a given criterion, the most commonly adopted method is mutation analysis [12]. Mutation analysis consists in the injection of simple syntactic faults into the source code, representing the most common mistakes committed by programmers. It is expected that a test set capable of revealing such simple faults would also be capable of revealing more complex ones. Programs with injected faults are called *mutants*. Mutants are executed with the test set to determine whether their behavior is different from the original program. Mutants that behave differently are considered as *killed by* the test. The product of the mutation analysis is a measure called *mutation score*, which indicates the percentage of mutants killed by a test set. Mutants are obtained by applying *mutation operators* that introduce the simple changes into the source code; for example, increment a constant by one yield one mutant. So, if we want to evaluate test sets generated according to our test selection strategy, we have to select the mutation operators most applicable to our case.

At the time we perform such experiments, existing mutation operators were applicable to procedural programs, either at unit or at integration level as well as to specifications in the form of finite state machines [14], statecharts [16], Petri Nets [15] and Estelle [30]. Since we are not testing methods in isolation, the unit test operators were not applicable. The same with specification operators, since we are not using none of the mentioned models. So we decided to use the interface mutation operators [10], because they focus on integration faults, thereby being more adequate to our study, since the tests considered method's interactions. Interface mutation is aimed at representing fault models relative to the interaction between two routines $R_1$ and $R_2$. Therefore, supposing that $R_1$ calls $R_2$, they affect the points where global variables and formal parameters are used inside $R_2$, as well as the points where values are returned from $R_2$ to $R_1$ (as in return statements in C, for example), thus local variables and constants in $R_2$ that affects the returned values are also considered. In this study, interface mutation operators were used to introduce faults affecting the interactions among methods that are part of a transaction. Our aim was to study whether the generated test cases were able to reveal these faults.

The operators used in the experiments reported here are presented in Table 1. They are a subset of the so-called essential operators, determined to reduce time and cost of the mutation analysis [35].

**Table 1. Interface mutation operators applied.**

| Operator | Description |
|---|---|
| IndVarBitNeg | Inserts bitwise negation at non-interface variable use |
| IndVarRepGlob | Replaces non-interface variable by $G(R_2)$ |
| IndVarRepLoc | Replaces non-interface variable by $L(R_2)$ |
| IndVarREpExt | Replaces non-interface variable by $E(R_2)$ |
| IndVarRepReq | Replaces non-interface variable by RC |
| Where<br>$G(R_2)$: set of global variables used in $R_2$;<br>$L(R_2)$: set of local variables defined in $R_2$;<br>$E(R_2)$: set of global variables not used in $R_2$;<br>RC: set of required constants, containing some special values such as NULL, MAXINT(greatest positive integer), MININT(least negative integer), and so on;<br>Non-interface variables $\in L(R_2) \cup E(R_2)$. | |

There exists a tool, Proteum/IM [11], to support mutants generation. However, in our study faults were manually inserted, since the tool was developed for the C language, and the classes we tested were in C++. One risk of this approach was the possibility of a potential bias due to the fact that the same person prepared the self-testable classes and the mutants. But actually this risk did not exist for the following reasons: (i) test cases were automatically generated, so the possibility that knowledge of test cases could impact the insertion of faults did not exist; and (ii) fault insertion was based on a set of clearly defined rules, according to the definition of the mutation operators. For example, the IndVarRepGlob operator requires the replacement of a method's local variable by a global variable (or class attribute) not used in this method.

Each mutant was created as a separate class, and they were individually compiled, to assure that all faulty classes compiled cleanly. Then the test sequence generated by Concat was applied to the mutants. The mutant was considered killed if one of the following situations occurred: (i) the program (driver + mutant class) crashed while running the test cases; (ii) an exception was raised due to assertion violation, during a mutant execution, given that this was not the case with the original program; and (iii) the output of the program that finished execution was different of the output of the original program (these outputs were validated by hand before experiments began).

Two experiments were performed. In the first one, faults were inserted into methods of the class CSortableObList, and the tests were applied to objects of that class, to observe their fault revealing power. Table 2 presents a summary of the results obtained with this first experiment. A total of 233 test cases were generated for this class, for a test model composed of 16 nodes and 43 links. This is the number of new test cases; the class reused 329 test cases from its superclass.

The first part of the table shows the number of mutants generated for each operator, that are applied to each method of

the target class that were considered in the experiment. The second part presents, for each operator: the total number of

**Table2. Results obtained for the CsortableObList class.**

| Method | Mutation Operator | | | | | |
|---|---|---|---|---|---|---|
| | IndVarBitNeg | IndVarRepGlob | IndVarRepLoc | IndVarREpExt | IndVarRepReq | Total |
| Sort1 | 4 | 64 | 99 | 104 | 9 | 280 |
| Sort2 | 7 | 25 | 25 | 25 | 25 | 107 |
| ShellSort | 24 | 25 | 25 | 25 | 28 | 127 |
| FindMax | 7 | 25 | 11 | 25 | 25 | 93 |
| FindMin | 7 | 25 | 11 | 25 | 25 | 93 |
| #mutants | 49 | 164 | 171 | 204 | 112 | **700** |
| #killed | 42 | 152 | 168 | 197 | 93 | **652** |
| #equivalent | 0 | 3 | 0 | 1 | 15 | **19** |
| Score | 85.7% | 94.4% | 98.2% | 97% | 95.8% | **95.7%** |

**Table 3. Results obtained for the CobList class.**

| Method | Mutation Operator | | | | | |
|---|---|---|---|---|---|---|
| | IndVarBitNeg | IndVarRepGlob | IndVarRepLoc | IndVarREpExt | IndVarRepReq | Total |
| AddHead | 2 | 21 | 3 | 13 | 3 | 42 |
| RemoveAt | 0 | 30 | 25 | 25 | 2 | 82 |
| RemovHead | 3 | 15 | 5 | 8 | 4 | 35 |
| #mutants | 5 | 66 | 33 | 46 | 9 | **159** |
| #killed | 2 | 43 | 23 | 29 | 4 | **101** |
| #equivalent | 0 | 0 | 0 | 0 | 0 | **0** |
| Score | 40% | 65.1% | 69.7% | 63% | 44.4% | **63.5%** |

mutants generated, the number of mutants killed, the number of mutants that are equivalent to the original program, and the mutation score, calculated as the ratio between the number of mutants killed and the number of non-equivalent mutants. A mutant is considered equivalent to the original program if there is no input data on which the mutant and the original program produce different output. The determination of equivalent mutants is a non-decidable problem, so they were obtained manually, by analyzing the mutants that were alive after the tests. The mutation score is thus a value in the interval [0, 1]; the most close to 1, the better, meaning that the test set is more effective to reveal the injected faults. The results show that the test strategy, although not based on the source code, is effective to reveal the types of faults that were inserted. It is worth noting that from the 652 mutants killed, 59 were due to assertion violation.

Although no definite conclusions can be drawn based on these preliminary results, they show that assertions, besides improving testability, help to improve fault-revealing effectiveness. The results also show that assertions alone do not constitute an effective oracle.

The second experiment were aimed to observe the effectiveness of the test set generated for the CSortableObList class, in revealing faults inserted into its base class. Thereby, the mutation operators presented in Table 1 were applied to the CObList class. A summary of the results obtained is shown in Table 3.

The test set for the CSortableObList class was generated according to the hierarchical incremental technique, as explained in section 3.4.2. According to this technique neither the specification-based nor the implementation-based test sets have to be rerun if they only test interactions among methods inherited without modifications. In our context, this means that a test case for a transaction which is composed only by inherited methods (constructor and destructor methods excluded) are not included in CSortableObList's test set.

Only test cases exercising transactions containing modified or new methods are included in the test set, either by reusing test cases from the base class, in case the modification in the subclass did not change the specification, or regenerated, in case of new methods. Therefore the low scores in Table 3 indicate that not retesting a transaction in the context of the subclass, although cost effective in terms of test productivity, can be dangerous, as many faults may not be revealed. This situation can occur, for example, when an application reuses components from a commercial library, and a new release of the library substitutes the old one.

## 5. Related works

Design for testability techniques and the self-testing concept have been using in hardware for a long time; however, only recently is this subject gaining more attention of the software community, although software testability concepts are not new. Here we present works that are most close to ours.

Voas et. al present a tool, ASSERT++ [37], to help the user place assertions into his/her programs to improve testability of OO software. The tool comprises two utility programs: one to support assertion insertion and the other that suggests points where the assertions might be inserted, according to testability

scores obtained. Their work is complementary to ours, in that we are not concerned with the best place to introduce assertions. Moreover, they are only concerned with improving controllability and observability, and so, test case generation and other built-in facilities, such as reporter methods, were not considered in their work.

Wang et al describe how to construct testable OO software by embedding test mechanisms into the source code [38, 39]. In fact, they include test cases aimed at covering implementation-based criteria applied to the methods of a class. These test cases can be inherited, as they are implemented as member functions (or methods), and therefore, allowing tests of a base class to be reused by the derived classes. The main differences with our approach are: (i) they use implementation-based criteria; (ii) test cases are integrated into the class, instead of its specification; and (iii) the use of assertions and other built-in capabilities to enhance controllability and observability is not mentioned. An advantage of their approach is that they don't need a test case generator. However, although implementation-based criteria are very useful during development and maintenance, the built-in test cases may become useless if the subclass introduces significant modifications to a method's implementation, even if its external behavior remains the same. Moreover, it is not always possible to embed test cases into abstract classes.

Le Traon et al also present self-testable OO components which embed their specification (documentation, methods signature and invariant properties) and test cases [34]. Their approach has been implemented in Eiffel, Java, Perl and C++ languages, and in the aforementioned reference they detailed the Eiffel implementation, for which the design by contract [26] support offered by that language makes the introduction of built-in test capabilities straightforward. Test cases are generated manually and embedded into the component. Assertions available in Eiffel are used as oracle, but manually generated oracles can also used in complement, in case post-conditions and invariants are not sufficient to express functional dependencies between methods. One interesting point in their approach is that a test quality estimate can be associated to each self-test, either to guide in the choice of a component, or to help reaching a test adequacy criteria when generating test cases. Test quality estimation is based on mutation analysis; the authors presented mutation operators applicable to different OO languages. Test case selection can be driven either by quality or by the maximum number of test cases desired. Another important aspect of their work is that they also present a strategy for the use of self-testable components for integration and regression testing. As the previous approach, test cases generated during class development are embedded into the class, and not the test model, as in our approach. Since test cases are created manually, they don't need a test case generator, but they need a mutant generator to evaluate test quality.

## 6. Conclusion and future works

The aim of this work is to provide an approach for building self-testable components. By integrating a test specification and built-in test mechanisms comprising assertions and reporter methods, we seek to increase component's testability, an important step to improve reliability of the component itself and of the applications using it. The steps for the production of self-testable component have been presented. A prototyping tool that supports some of these steps was developed. It is focused on OO components developed in C++. One step that is automated is test selection, performed by the consumer using test information contained in the component. The main aspects of our testing selection strategy are: (i) it is based on the component specification; (ii) it combines different techniques that cover not only behavioral aspects but also values of method's parameters; (iii) it allows test case reuse when deriving tests for a subclass, by the use of the hierarchical incremental technique. Preliminary results of an empirical validation of the test selection strategy using mutation analysis are also presented. Although incipient, the results showed that the test strategy has a good potential to reveal methods' interaction faults. They also served to point out the need to retest inherited features in the context of a subclass, even if they don't interact with modified or newly introduced features, among other reasons, to avoid that faults introduced during base class maintenance remain unrevealed in the subclass.

As future works, in the short term, further empirical evaluations are being envisaged, to help improve the approach. We are also extending this approach for components having more than one class; so instead of method's interactions inside a class (intraclass testing), we focus on interactions between classes (interclass testing). In the long term, we envisage the construction of an infrastructure supporting the development and use of self-testable components that is generic enough to be applicable to components developed using different languages.

## 7. Acknowledgements

This work was partially supported by grants from the Brazilian Research Council (CNPq). We would also like to thank the reviewers, whose comments greatly contributed to the improvement of this text.

## 8. References

1. F.Bachmann, L.Bass et al. "Technical Concepts of Component Based Software Engineering". Technical Report CMU/SEI-2000-TR-008, 04/2000.
2. S.Barbey, D.Buchs, C.Péraire. "A Theory of Specification-Based Testing for Object-Oriented Software", in Proc. 2nd. European Dependable Computing Conference (EDCC-2), Italy, 1996, pp303-320.
3. A.Beugnard, J.Jezequel, N.Plouzeau, D.Watkins. "Making Components Contract Aware", IEEE Computer, July/1999, pp38-45.
4. B.Beizer. "Software Testing Techniques". International Thomson Computer Press, 2nd. Edition, 1990.
5. B.Beizer. "Black-Box Testing". John Wiley & Sons, 1995.
6. R.V.Binder. "Testing Object-Oriented Systems: Models, Patterns and Tools". Addison-Wesley, 2000.
7. R.V.Binder. "The FREE Approach to Testing Object-Oriented Software: an Overview", report of RSBC, 1994, available on the Web: www.rsbc.com.
8. R.V.Binder. "Design for Testability in Object-Oriented Systems", Comm. of ACM, 37(9), Sept./1994, pp87-101.
9. R.V.Binder. "Testing for Reuse: Libraries and Frameworks", in Journal of Object-Oriented Programming (JOOP), 6(6), Aug/1996.

10. M.E.Delamaro. "Interface Mutation: an Adequacy Criterion for Integration Testing". PhD Thesis, Physics Institute, S.Paulo State University at S.Carlos, 1997. (in Portuguese)
11. M.E.Delamaro, J.C.Maldonado, A.M.R.Vincenzi. "Proteum/IM 2.0: An Integrated Mutation Testing environment", in Proc. of Mutation 2000 Symposium, USA, Oct/2000.
12. R.A.DeMillo, R.J.Lipton, F.G.Sayward. "Hints on test data selection: Help for the Practitioner Programmer". IEEE Computer, 11(4), 1978, pp34-43.
13. R.K.Doong, P.Frankl. "The ASTOOT Approach to Testing Object-Oriented Programs", in ACM Trans on Software Engineering and Methodology, 3(2), April/1994, pp101-130.
14. S.C.P.F.Fabbri, J.C.Maldonado, P.C.Masiero, M.E.Delamaro. "Mutation Analysis Testing for Finite State Machines", in Proc. 5th. International Symposium on Software Reliability Engineering (ISSRE'94), USA, Nov/1994, pp220-229.
15. S.C.P.F.Fabbri, J.C.Maldonado, P.C.Masiero, M.E.Delamaro. "Mutation Analysis Applied to Validate Specifications Based on Petri Nets", in FORTE'95 – 8th. IFIP Conference on Formal Descriptions Techniques for Distributed Systems and Communication Protocols, Canada, Oct/1995, pp329-337.
16. S.C.P.F.Fabbri, J.C.Maldonado, T.Sugeta, P.C.Masiero. "Mutation Testing Applied to Validate Specifications Based on Statecharts", in Proc. 9th. International Symposium on Software Reliability Engineering (ISSRE'99), USA, Nov/1999, pp210-219.
17. X.Franch. "The Convenience for a Notation to Express Non-Functional Characteristics of Software Components", in Proc. 1st. Workshop on Component-Based Systems, Switzerland, 1997. Available on the Web: www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html.
18. R.S.Freedman. "Testability of Software Components". IEEE Trans. Software Eng., 17(6), June/1991, pp553-564.
19. J.Gao. "Testing Component-Based Software". Proc. of STARWest, San Jose, USA, 1999. Available on the Web: www.engr.sjsu/edu/gaojerry/report/star-test.htm.
20. M.-J.Harrold, J.McGregor, K.Fitzpatrick. "Incremental Testing of Object-Oriented Class Structures, in Proc. 14th. IEEE Int. Conference on Software engineering (ICSE-14), 1992, pp68-80.
21. D.Hoffman. "Hardware Testing and Software Ics". Proc. Northwest Software Quality Conference, USA, Sept/1989, pp234-244.
22. D.Hoffman, P.Strooper. "The Testgraph Methodology", in Journal of Object-Oriented Programming (JOOP), Nov-Dec/1995, pp35-41.
23. IEEE Standard Glossary of Software Engineering Terminology ANSI/IEEE 610.12, 1990, IEEE Press.
24. D.Kung, J.Gao, P.Hsia, Y.Toyoshima, C.Chen, Y.S.Kim, Y.K.Song. "Developping an Object-Oriented Software Testing and Maintenance Environment", in Comm. of the ACM, 38(10), Oct/1995, pp75-87.
25. B.Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
26. B.Meyer. "Applying Design by Contract". IEEE Computer, Oct./1992, pp40-51.
27. G.J.Myers. "The Art of Software Testing". John Wiley & Sons, 1978.
28. R.S.Pressman, *Software Engineering, a Practitioner's Approach*. McGraw-Hill, 1997, 4th. Edition.
29. S.Siegel. "Object-Oriented Software Testing: a Hierarchical Approach". John Wiley & Sons, 1996.
30. S.R.S. Souza, J.C.Maldonado, S.C.P. F.Fabbri, W.Lopes de Souza. "Mutation Testing Applied to Estelle Specifications", in Proc. of 33th. Hawaii Intern. Conf. On System Science, mini-track on Distributed Systems Testing, Hawaii, Jan/2000.
31. C.Szyperski. "Component Software. Beyond OO Programming". Addison-Wesley, 1998.
32. C.M.Toyota, E.Martins. "Reuse in OO Testing: a Methodology for the Construction of Sekf-Testing Classes", in IX International Conference on Software Technology – Software Quality, Curitiba, Brazil, June/1998. (in Portuguese)
33. C.M.Toyota. "Improving Class Testability using the Self-Testing concept". Master Dissertation, Institute of Computing, State University of Campinas, June/2000. (in Portuguese)
34. Y.Traon; D.Deveaux; J.-M.Jézéquel. "Self-testable Components: from Pragmatic Tests to Design-for-Testability Methodology". in Proc. 1st. Workshop on Component-Based Systems, Switzerland, 1997. URL: www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html.
35. A.M.R.Vincenzi, J.C.Maldonado, E.F.Barbosa, M.E.Delamaro. "Essential Interface Operators: a Case Study", in XIII Brazilian Symposium on Software Engineering, S.Catarina, Brazil, Oct/1999, pp373-391.
36. J.M.Voas; K.W.Miller. "Software Testability: the New Verification". IEEE Software, March/1995, pp17-28.
37. J.M.Voas, M.Schmid, M.Schatz. "A Testability based Assertion Placement Tool for Object-Oriented Software". Technical Report – Information Technology Laboratory, NIST CGR 98-735, Jan/1998. Available on the Web: http://www.rstcorp.com.
38. Y.Wang, G.King, I.Court, M.Ross. "On Testable Object-Oriented Programming", in ACM Software Engineering Notes, 22(4), July/1997, pp84-90.
39. Y.Wang; G.King; H.Wickburg. "A Method for Built-in Tests in Component-based Software Maintenance". Proc. 3rd. European Conference on Software Maintenance and Reengeneering (CSMR), Holanda, March/1999, pp186-189.
40. E.J.Weyuker. "Axiomatizing Software Test Data Adequacy". IEEE Trans. Software Eng. SE-12, 12, Dec/1986, p1128-1138.
41. E.J.Weyuker. "Testing Component-Based Software: a Cautionary Tale". IEEE Software, Sept-Oct/1998, pp54-59.
42. R.L.Yanagawa. "Evaluating the approach for construction and use of self-testable components". Master Dissertation, Institute of Computing, State University of Campinas, December/2000. (in Portuguese)