Reusing MDA Components: A Design-for-Trust Approach

Yves Le Traon*, Mariano Belaunde* and Jean-Marc Jézéquel**
* France Télécom R&D/MAES/EXA, 2, avenue Pierre Marzin
22 307 Lannion Cedex – France
{yves.letraon, mariano.belaunde}@francetelecom.com

IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
jezequel@irisa.fr

Abstract

In essence, MDA proposes a move away from human interpretation of high-level models, such as design diagrams, into implementations, towards a more automated process where the models are used as first-class artifacts of the development process. The core mechanism for this automation is model transformation. Among the many aspects of the model-driven development process, making model transformations trustable is an obvious target since they impact on the design process reliability. This is even more critical when transformations are to be reused. Ideally, a model transformation program should be designed and tested so that it may be used and reused safely as a MDA component. This paper presents a method for building trustable MDA components. We first define the notion of MDA components, e.g. model transformation programs, as composed of its specification, one implementation and the test cases needed for testing it. The testing-for-trust approach, using the mutation analysis, checks the consistency between specification, implementation and tests. It points out the tests lack of efficiency but also the lack of precision of the executable part of the component’s specification which is captured by the notion of guard in the QVT 2 OMG standard. Thus, we can associate each self-testable component with a value—its level of trustability—that quantifies the test sequence’s effectiveness at testing a given implementation of the component. Our method for achieving this quantification is a version of selective mutation analysis that we adapted to transformation languages, w.r.t. the QVT 2 upcoming standard. Relying on this objective estimation of component trustability, the model transformation developer can then consciously trade reliability for resources to meet time and budget constraints. In this article, we outline the TRL implementation of our methodology.

1. Introduction (à réécrire)

Object-oriented modeling is now mature enough to provide a normalized way of designing software systems in the context of the UML as well as a natural way of encapsulating services into the notion of “component”. This way of modeling could be roughly expressed with the maxim: “the way you think about the system, the way you design it”. However, despite the growing interest due to this incremental way of building software, few works tackle the question of the trust we can put into a component or the question of designing for trustability. Indeed, the trustability is a property that should accompany the OO components expected capability to evolve (addition of new functionality, implementation change), to be adapted to various environments and to be reused. As for hardware systems, we propose to build trust in components through testing. Despite this initial lack of interest, testing and trusting object-oriented systems is receiving much more attention (see http://www.trusted-components.org/ and [1] for a detailed state of the art).

In [8], we presented a pragmatic approach for linking design and test of classes, seen as basic unit test components. Each component is enhanced by the ability to invoke its own tests: components are made self-testable. The approach is conceptual and thus generalized to upper levels: class packages become self-testable by composition of self-testable classes. At any level of complexity, self-testable components have the ability to launch their own tests. While
giving to a component the ability to embed its selftest is a good thing for its testability, estimating the quality of the embedded tests becomes crucial for the component trustability.

Software trustability [6], as an abstract software property, is difficult to estimate directly: one can only approach it by analyzing concrete factors that influence this qualitative property. In this paper, we consider that the truthfulness in the component test suite is the main indirect factor that brings trust into a component. We consider a component as an organic set composed of a specification, a given implementation and its embedded test cases. With such definition the trustability of a component will be based on the consistency between these three aspects. In a “design-by-contract” approach [10], the specification is systematically derived in executable contracts (class invariants, pre/post condition assertions for class methods). If contracts are complete enough, they should be violated when both the implementation is incorrect and the test case exercises the incorrect part of the implementation. Contracts should thus be improved by checking whether they are able to detect a faulty implementation. By improving contracts, the specification is refined and the component’s consistency is improved.

In this paper, we propose a testing-for-trust methodology that helps checking the consistency of the component’s three facets. The methodology is an original adaptation from mutation analysis principles [2]: the quality of a test set is related to the proportion of faulty programs it detects. Faulty programs are generated by systematic fault injection in the original implementation. In our approach, we consider that contracts should provide most of the oracle functions; the question of the efficiency of contracts to detect the presence of anomalies in the implementation or in the provider environment is thus tackled and studied (Section 4). If the generation of a basic test set is easy, improving its quality may require prohibitive efforts. In a logical continuity with our mutation analysis approach and tool, we describe how such a basic unit test set, seen as a test seed, can be automatically improved using genetic algorithms to reach a better quality level.

Section 2 opens on methodological views and steps for building trustable component in our approach. Section 3 concentrates on the mutation testing process adapted to OO domain and the associated tool dedicated to the Eiffel programming language. The test quality estimate is presented as well as the automatic optimization of test cases using genetic algorithms. Section 4 is devoted to an instructive case study that illustrates the feasibility and the benefits of such an approach. Section 5 presents and discusses related works.

2. Test quality for trusting component

The methodology is based on an integrated design and test approach for OO software components, particularly adapted to a design-by-contract approach, where the specification is systematically derived into executable assertions (invariant properties, pre/postconditions of methods). Classes that serve for illustrating the approach are considered as basic unit components: a component can also be any class package that implements a set of well-defined functionality. Test suites are defined as being an “organic” part of software OO component. Indeed, a component is composed of its specification (documentation, methods signature, invariant properties, pre/postconditions), one implementation and the test cases needed for testing it. This view of an OO component is illustrated under the triangle representation (cf. Figure 1). To a component specified functionality is added a new feature that enables it to test itself: the component is made self-testable. Self-testable components have the ability to launch their own unit tests as detailed in [8].

From a methodological point of view, we argue that the trust we have in a component depends on the consistency between the specification (refined in executable contracts), the implementation and the test cases. The confrontation between these three facets leads to the improvement of each one. Before definitely embedding a test suite, the efficiency of test cases must be checked and estimated against implementation and specification, especially contracts.
Tests are build from the specification of the component; they are a reflection of its precision. They are composed from two independent conceptual parts: test cases and oracles. Test cases execute the functions of the component. Oracles – predicates for the fault detection verdict – can either be provided by assertions included into the test cases or by executable contracts. In a design-by-contract approach, our experience is that most of the verdicts are provided by the contracts that are derived from the specification. The fact that contracts of the components are inefficient to detect a fault exercised by the test cases reveals a lack of precision in the specification. The specification should be refined and new contracts added. The trust in the component is thus related to the test cases efficiency and the contracts “completeness”. We can trust the implementation since we have tested it with a good test cases set, and we trust the specification because it is precise enough to derive efficient contracts as oracle functions.

![Measure of Trust based on Consistency](image)

**Fig. 1. Trust based on triangle consistency**

The question is thus to be able to measure this consistency. This quality estimate quantifies the trust one can have in a component. The chosen quality criteria proposed here is the proportion of injected faults the self-test detects when faults are systematically injected into the component implementation. This estimate is, in fact, derived from the mutation testing technique, which is adapted for OO classes. The main classical limitation for mutation analysis is the combinatory expense.

### 3. Mutation testing technique for OO domain

Mutation testing is a testing technique that was first designed to create effective test data, with an important fault revealing power [11]. It has been originally proposed in 1978 [2], and consists in creating a set of faulty versions or mutants of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. In practice, faults are modeled by a set of mutation operators where each operator represents a class of software faults. To create a mutant, it is sufficient to apply its associated operator to the original program.

A test set is relatively adequate if it distinguishes the original program from all its non-equivalent mutants. Otherwise, a mutation score (MS) is associated to the test set to measure its effectiveness in terms of percentage of the revealed non-equivalent mutants. It is to be noted that a mutant is considered equivalent to the original program if there is no input data on which the mutant and the original program produce a different output. A benefit of the mutation score is that even if no error is found, it still measures how well the software has been tested giving the user information about the program test quality. It can be viewed as a kind of reliability assessment for the tested software.

In this paper, we are looking for a subset of mutation operators
- general enough to be applied to various OO languages (Java, C++, Eiffel etc)
- implying a limited computational expense,
- ensuring at least control-flow coverage of methods.

Our current choice of mutation operators includes selective relational and arithmetic operator replacement, variable perturbation and referencing faults (aliasing errors) for declared objects. During the test selection process, a mutant program is said to be killed if at least one
test case detects the fault injected into the mutant. Conversely, a mutant is said to be *alive* if no test case detects the injected fault. The choice of mutation operators is given in Table 1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EHF</td>
<td>Exception Handling Fault</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td>LOR</td>
<td>Logical Operator Replacement</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator Replacement</td>
</tr>
<tr>
<td>NOR</td>
<td>No Operation Replacement</td>
</tr>
<tr>
<td>VCP</td>
<td>Variable and Constant Perturbation</td>
</tr>
<tr>
<td>MCR</td>
<td>Methods Call Replacement</td>
</tr>
<tr>
<td>RFI</td>
<td>Referencing Fault Insertion</td>
</tr>
</tbody>
</table>

Table 1: Mutation operators set for OO programs

**Description of the functionality of mutation operators:**
EHF: Causes an exception when executed. This operator allows forcing code coverage.
AOR: Replaces occurrences of "+" by "-" and vice-versa.
LOR: Each occurrence of one of the logical operators (and, or, nand, nor, xor) is replaced by each of the other operators; in addition, the expression is replaced by TRUE and FALSE.
ROR: Each occurrence of one of the relational operators (<, >, <=, >=, =, /=) is replaced by each one of the other operators.
NOR: Replaces each statement by the *Null* statement.
VCP: Constants and variables values are slightly modified to emulate domain perturbation testing. Each constant or variable of arithmetic type is both incremented by one and decremented by one. Each *boolean* is replaced by its complement.
MCP: Methods calls are replaced by a call to another method with the same signature.
RFI: Stuck-at void the reference of an object after its creation. Suppress a clone or copy instruction. Insert a clone instruction for each reference affectation.

The mutation operators AOR, LOR, ROR and NOR are traditional mutation operators [3,11], the other operators have been introduced in this paper for the object-oriented domain. Operator RFI introduces object aliasing and object reference faults, specific to object-oriented programming:

### 3.1. Test selection process

The whole process for generating unit test cases with fault injection is presented in Figure 2. It includes the generation of mutants and the application of test cases against each mutant. The verdict can be either the difference between the initial implementation’s output and the mutant’s output, or the contracts and embedded oracle function. The diagnosis consists in determining the reason of a non detection: it may be due to the tests but also to incomplete specification (and particularly if contracts are used as oracle functions). It has to be noted that when the set of test cases is selected, the mutation score is fixed as well as the test quality of the component. Moreover, except for diagnosis, the process is completely automated.

The mutation analysis tool developed, called mutants slayer or µSlayer, is suitable for the Eiffel language. This tool injects faults in a class under test (or a set of classes), executes selftests on each mutant program and delivers a diagnosis to determine which mutants were killed by tests as shown in figure 2. All the process is incremental (we do not start again the execution of already killed mutants for example) and is parameterized: the user for example selects the number and types of mutation he wants to apply at any step. The µSlayer tool is available from [http://www.irisa.fr/pampa/](http://www.irisa.fr/pampa/).
3.2. Component and system test quality

The test quality of a component is simply obtained by computing the mutation score for the unit testing test suite executed with the self-test method.

The system test quality is defined as follows:

- let $S$ be a system composed of $n$ components denoted $C_i$, $i \in [1..n]$,
- let $d_i$ be the number of dead mutants after applying the unit test sequence to $C_i$, and $m_i$ the total number of mutants.

The test quality (TQ), i.e. the mutation score $MS$, and the System Test Quality (STQ) are defined as follows:

$$
TQ(C_i, T_i) = \frac{d_i}{m_i} \quad STQ(S) = \frac{\sum_{i=1}^{n} d_i}{\sum_{i=1}^{n} m_i}
$$

These quality parameters are associated to each component and the global system test quality is computed and updated depending on the number of components actually integrated to the system.

In this paper, such a test quality estimate is considered as the main estimate of component’s trustability.

4. Application to MDA Components

To be done

5. Related works (to be completed)

An original measure of the quality of components has been defined based on the quality of their associated tests (itself based on fault injection). For measuring test quality, the presented approach differs from classical mutation analysis [3,11] as follows:

- a reduced set of mutation operators is needed,
- oracles functions are integrated to the component, while classical mutation analysis uses differences between original program and mutant behaviors to craft a pseudo-oracle function.

In this paper, the proposed methodology is based, on a first step, of pragmatic unit test generation and aims at bridging the existing gap between unit and system dynamic tests. In a second step, advanced test optimization techniques, such as genetic algorithms, may help for automatically improving test quality and, consequently, component trustability. To achieve a complete design-for-trust process, the notion of structural test dependencies has been developed for modeling the systematic use of self-testable components for structural system test. In [8], the design-for-testability main methodology is outlined. In this paper, we detailed the testing-for-trust method while [9] describe the automatic production, from UML design models, of an integration test plan that both minimizes the test effort and the test duration for an object-oriented system.

Concerning advanced test generation based on genetic algorithms, genetic algorithms have been recently studied for two different problems. In [7], genetic algorithms are used in a control-flow coverage-oriented way: test sets are improved to reach such a predefined test adequacy criterion. In [12], genetic algorithms are used for performing some kinds of reliability assessment. In this paper, the application of genetic algorithm is coherent with the application of mutation analysis for test qualification. This conceptual continuity, due to the constant analogy of the test selection problem with a “darwinian” analogy, appears if we consider that the µSlayer tool allows both the mutation of programs and the mutation of genes (part of a test “individual”) via the domain perturbation mutation operator.

6. Conclusion

The presented work detailed a method and a tool to help programmers/developers building trustable OO components. This method, based on test qualification, also leads to contracts improvements. The feasibility of components validation by mutation analysis and its utility to test generation have been studied as well as the robustness of trustable and self-testable components in an infected environment. The approach presented in this paper aims at providing a consistent framework for building trust into components. By measuring the quality of test cases, we try to build trust in a component passing those test cases. Future work will focus on demonstrating the correlation between test quality value and software reliability. Some considerations may guide the demonstration:

- in a system, the crossing/coupling power of each trustable class selftest improves the global test coverage of each system component (see the results on component robustness),
- the number of statements executed during the execution of all tests provides some kind of time measure, that is needed for any reliability measure,
- if all selfests of the system are passed successfully, then we can make the pessimistic assumption that any new test execution would provoke a failure.

All these considerations are elements for defining the link between the testing-for-trust proposed approach and a reliability assessment.

References


