Testing-for-Trust: the Genetic Selection Model Applied to Component Qualification

Benoit Baudry, Vu Le Hanh and Yves Le Traon
IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Benoit.Baudry, vlhanh, Yves.Le_Traon }@irisa.fr

Abstract

This paper presents a method and a tool for building trustable OO components. The methodology is based on an integrated design and test approach for OO software components. It is particularly adapted to a design-by-contract approach, where the specification is systematically derived into executable assertions (invariant properties, pre/postconditions of methods). This method, based on test qualification using fault injection (mutation analysis), also leads to contracts improvements. We consider a component as an organic set composed of a specification, a given implementation and its embedded test cases. 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 contracts. As an advanced way of enhancing the tests set, the application of a genetic algorithm is presented as complementary of mutation analysis for test qualification. In this paper, the genetic selection model, a "darwinian" analogy, is thus used for the problem of efficient tests selection.

The feasibility of components validation by mutation analysis and its usefulness for test generation are studied as well as the robustness of trustable and self-testable components into an infected environment.

1 Introduction

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.

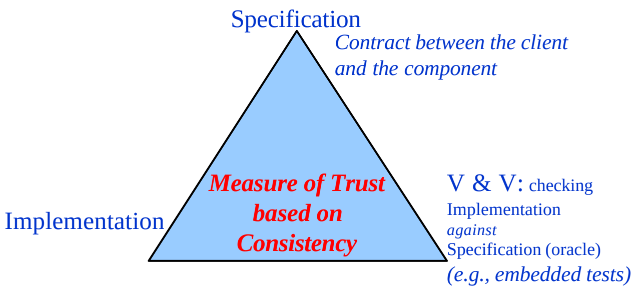
Specification
Contract between the client and the component
Measure of Trust based on Consistency
Implementation
V & V: checking Implementation against Specification (oracle) (e.g., embedded tests)

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.

Type	Description
EHF	Exception Handling Fault
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
NOR	No Operation Replacement
VCP	Variable and Constant Perturbation
MCR	Methods Call Replacement
RFI	Referencing Fault Insertion

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/.

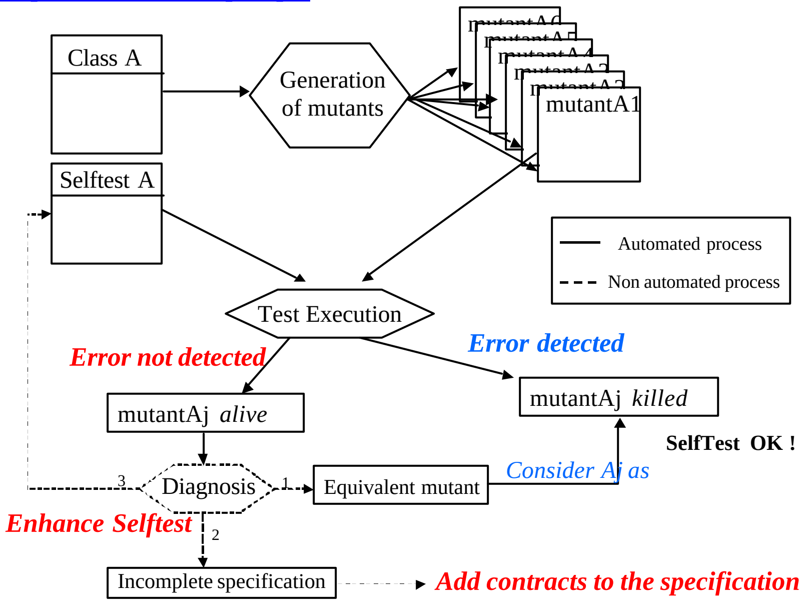

Fig. 2.The µSlayer tool in the global testing-for-trust process

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 :

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

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 Test cases generation : genetic algorithms for test generation

In this paper, we argue that writing a first set of component test cases is easy, and most developers do such basic testing. Memorizing such test cases into a self-testable class is a low-cost task. Our experiments taught us that such test cases easily reach 60% of Test Quality (see the following case study).

Then improving test quality implies a particular and specific supplementary testing effort. In this section we investigate the use of genetic algorithms as a pragmatic way to automatically improve the basic test cases set in order to reach a better Test Quality level with limited effort. Indeed, the basic test cases set carries information that can be optimized to create better test cases, by some cross-checking and "mutation" of the test cases themselves. So, at the beginning we have a population of mutants programs to be killed and a test cases pool. We randomly combine those test cases (or "gene pool") to build an initial population of test sets which are the predators of the mutant population. From this initial population, how can we mute the "predators" test cases and cross them over in order to improve their ability to kill mutants programs? One of the major difficulties in genetic algorithms is the definition of a fitness function. In our case, this difficulty does not exist: the mutation score is the function that estimates the efficiency of a test case.

4.1 Genetic Algorithms

Genetic algorithms [4] have been first developed by John Holland [5], whose goal was to rigorously explain natural systems and then design artificial systems based on natural mechanisms. So, genetic algorithms are optimization algorithms based on natural genetics and selection mechanisms. In nature, creatures which best fit their environment (which are able to avoid predators, which can handle coldness…) reproduce and, due to crossover and mutation, the next generation will fit better. This is just how a genetic algorithm works: it uses an objective criterion to select the fittest individuals in one population, it copies them and creates new individuals with pieces of the old ones.

To write a genetic algorithm we need to code individuals as a finite string of genes (genes can be bits, letters…). We also have to define a fitness function F which, for every individual x among a population, gives F(x), the value which is the quality of the individual regarding the problem we want to solve. This corresponds to the function we want to maximize.

Moreover, a genetic algorithm uses three operators: reproduction, crossover and mutation.

- Reproduction makes copies of the individuals which are going to participate to crossover, they are chosen according to their F(x) value.
- Crossover: two individuals are chosen, a position on the string of genes is selected at random, and the tails of the two strings are exchanged.
- Mutation operator modifies one or several genes' value. (e.g. if an individual is a bit string, mutation means changing a 1 to 0 and vice versa)

The reproduction and crossover operators are so powerful in improving the search, that the mutation operator usually plays a secondary role.

When we have those three operators and the fitness function, a genetic algorithm is easy to compute:

1. choose an initial population
2. calculate the fitness value for each individual
3. compute the reproduction operator on this population, this gives the new population
4. crossover
5. mutate one or several individuals
6. go back to step 2.

4.2 Genetic algorithms for tests generation

In this section we explain what is an individual in our specific problem and what fitness function and operators we use.

An individual is a test set, and genes are tests. We consider a test as a couple of initialization and method calls. The initialization prepares the system to accept the method calls.

Notations

Test: 1 test = 1 gene

Gene: G = [an initialization sequence, several method calls] = [I , S]

Individual: An individual is defined as a finite set of genes = $\{G_1,\ldots,G_m\}$

The function we want to maximize is the one we use as the fitness function; in our problem, it is the mutation score.

Here are the three operators adapted to our problem:

- <u>Reproduction</u>: the slot for each individual in the wheel roulette is proportional to its mutation score.
- <u>Crossover</u>: let's select at random an integer i between 1 and m-1, then from two individuals A and B, we can create two new individuals A' and B'. A' is made of the i first genes of A and the m-i last genes of B, and B' is made of the i first genes of B and m-i last genes of B.

$$\begin{tikzcd}[column sep=small]
{ind_1 = \{G_{1\,1}, \ldots G_{1\,i}, G_{1\,i+1}, \ldots G_{1\,m}\}} & {} \arrow[d] & {ind_2 = \{G_{2\,1}, \ldots G_{2\,i}, G_{2\,i+1}, \ldots G_{2\,m}\}} \\
{ind_3 = \{G_{1\,1}, \ldots G_{1\,i}, G_{2\,i+1}, \ldots G_{2\,m}\}} & {} & {ind_2 = \{G_{2\,1}, \ldots G_{2\,i}, G_{1\,i+1}, \ldots G_{1\,m}\}}
\end{tikzcd}$$

- <u>Mutation</u>: we use two mutation operators. The first one changes the method call parameters values in one or several genes.

$$G = [I , S] \longrightarrow G = [I , S_{mut}]$$

$$S = (m_1(p_1),\ldots,m_i(p_i),\ldots m_n(p_n)) \longrightarrow S_{mut} = (m_1(p_1),\ldots,m_i(p_{i\,mut}),\ldots m_n(p_n))$$

This mutation operator is important, for example if there is an if-then-else structure in a method, we need one value to test the if-branch and another one to test the else-branch, in this case it is interesting to try different parameters for the call. Moreover, in practice, we can use µSlayer's VCP operator to implement this operator.

The second mutation operator makes a new gene with two genes either by adding, at the end of a gene, the method calls of the other gene (this is how the size of a gene can change), or by switching the genes initialization sequences.

$$\begin{tikzcd}[column sep=small]
{} & {G_1 = [I_1 , S_1] \quad G_2 = [I_2 , S_2]} \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & {} & {} \\
{G_3 = [I_2 , S_1]} & {G_4 = [I_1 , S_2]} & {G_5 = [I_1 , S_1 S_2]} & {G_6 = [I_2 , S_2 S_1]}
\end{tikzcd}$$

This operator is important to make tests for bigger execution cases. We said earlier that, in genetic algorithms, the mutation operator plays a secondary role, but our mutation operators play an important role. Indeed, they are the only operators that change the mutation score of a gene; the other operators just reorganize the genes to change the global mutation score of individuals and the population.

Global process

First we try to compute the program under test with the initial population of test sets given to our process. We eliminate some tests that do not fit with the program specification (wrong value for method call, unexpected method call…) and have a set of potential predators. Then we compute this set with mutant programs, and this will give us the mutation score of every test. With these scores we can compute our genetic algorithm.

The algorithm will give a new population of better predators. We shall add several old predators to this population (to avoid stopping on local maximums), and then start a new round of the process with this new population. The process stops when the population is no more improved from one round to the other, based on the Quality Test value. As presented in Figure

3, the test optimization steps are just plugged in the testing-for-trust process. The genetic algorithm is applied as soon as the Quality Test (i. e. the mutation score) level is no more improved.

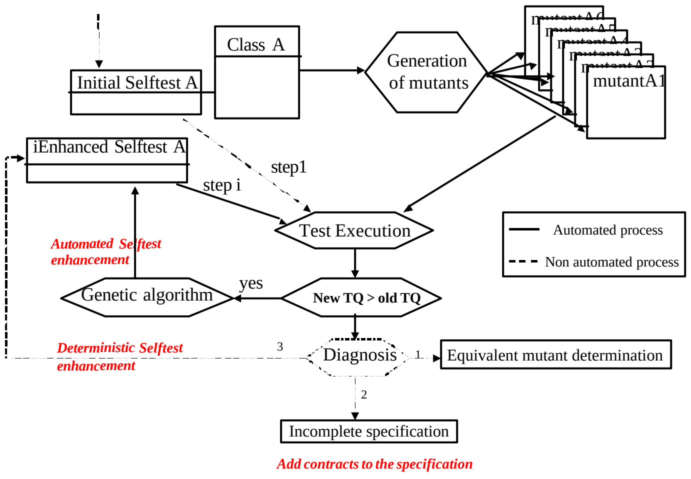

Fig.3. The global testing-for-trust process with automated genetic test enhancement

Algorithmic cost

The expensive part of this process, is running the tests on every mutant program. Indeed, there are usually many mutants (275 mutants for p_time.e for example). However, it is not as expensive as we could think, because in any given step of the process, we only run the tests (genes) that have been changed by the mutation operators. Indeed, for the other tests we know their mutation score from the previous turn, so we do not need to compute them.

The other operations of the global process are not expensive. The genetic algorithm is just a random reorganization of genes and several mutations. The compiler detects the faulty tests.

5 Case study

In this case study, the class package of the Pylon library (http://www.eiffel-forum.org/archive/arnaud/pylon.htm) relating to the management of time was made self-testable. These classes are complex enough to illustrate the approach and obtain interesting results. The main class of this package is called p_date_time.e. The way in which the various classes used in this package interact is presented in Figure 4.

This study proceeds in stages for better isolating the efforts of test data generation compared to those of oracle production. In real practice, the contracts – that should be effective as embedded oracle functions - can be improved in a continuous process: in this study, we voluntarily separate test generation stage from contract improvement one to compare the respective efforts. The last stage only aims to test the capacity of contracts to detect faults coming from provider classes. We call that capacity the "**robustness**" of the component against an infected environment.

5.1 Aims of the study

The aims of this case study are the following:

1. estimating the test generation with genetic algorithms for reaching 100% mutation score,
2. appraising the initial efficiency of contracts and improve them using this approach,

3. estimating the robustness of a component embedded selftest to detects faults due to external infected provider classes.

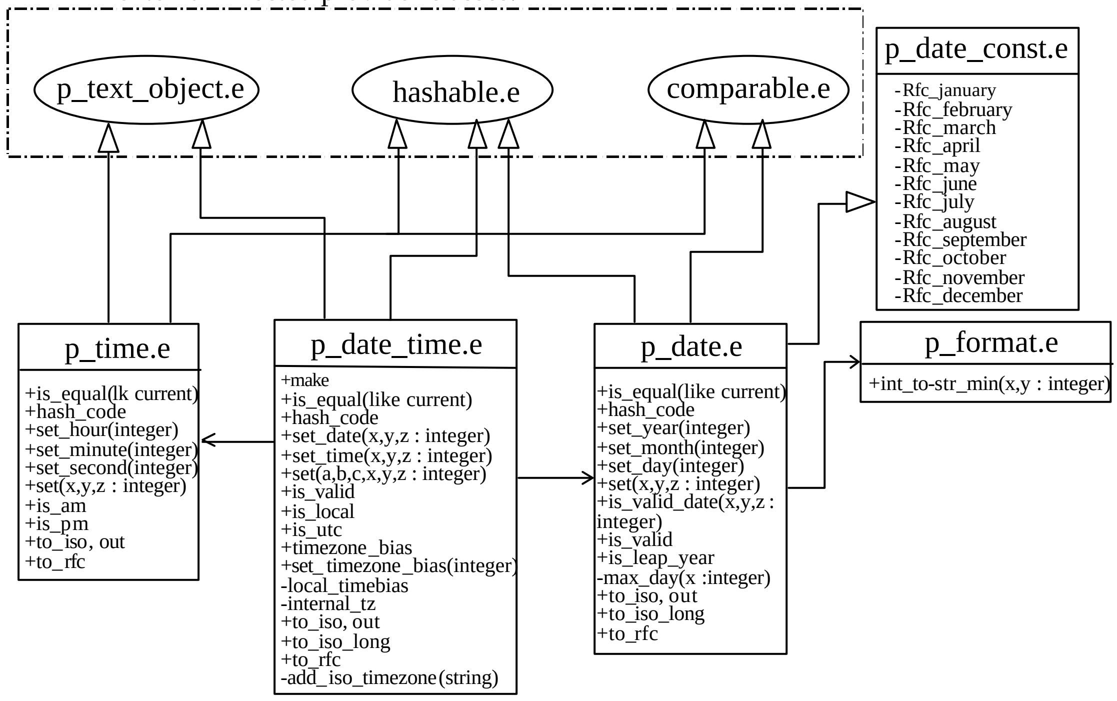

Fig 4.Classes of the package "date-time**"**

Concerning point 2, when a 100% mutation score is reached, the proportion of mutants detected by contracts is measured. This provides a simple way of estimating the efficiency of embedded contracts as oracle functions. This also reveals the ability of the component to be “clever” enough to detect its own defects. The contracts are then improved systematically, and the new efficiency of contracts estimated.

The last point aims at estimating whether a self-testable system, with high quality tests, is robust enough to detect new external faults due to integration or evolution. Indeed, each component’s selftest checks its own correctness but also some of its neighboring provider’s components. These cross-checking tests between dependent components increase the probability to detect faults in the global system. So the intuition is that 100 tests method calls per class in a 100 classes system make a high fault revealing power test of 10 000 tests for the whole system. The question is thus to estimate whether a selftest has or not a good probability to detect a fault due to one of its infected provider.

The analysis focuses on the classes **p_date_time.e, p_date.e** and **p_time.e** (see Figure 4). In fact, the classes **p_date_const.e** and **p_format.e** do not have a great number of methods, and carry especially the values for constants.

5.2 Results and lessons learnt

On the classes that are studied here, this first stage of generation allowed to eliminate approximately 60 to 70% of the generated mutants. It corresponds to the test seed that can be used for automatic improvement through genetic algorithm optimization (see Section III.3). Figure 5 presents the curves of the mutation score growth in function of the number of generated predators (one plot represents a generation step). To avoid the combinatory expense, we limit the new mutated generation to the predators which have the best own mutation score (good candidates). The new generation of predators was thus target-guided (depends on the alive mutants) and controlled by the fitness function. Results are encouraging even if the CPU time remains important (2 days of execution time for the three components to reach more than 90 percent mutation score on a Pentium II). The main interest is that the test improvement process is automated.

Then, the mutation score has been improved by analyzing the mutants one by one: equivalent mutants were suppressed and specific test cases were written for alive mutants to reach 100% mutation score.

Concerning the improvement of contracts, the results on the initial quality of contracts used as oracles are given in Table 2. The table recapitulates the initial efficiency of contracts and then the final level they reached after improvement.

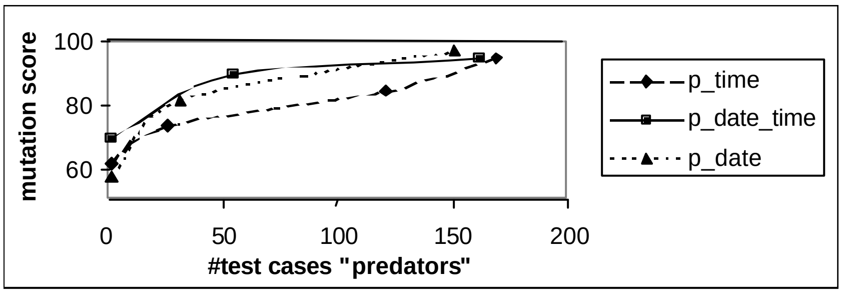

Fig.5. Genetic algorithm results for test optimization

	p_date	p_time	p_date_time
Total number of generated mutants	673	275	199
# equivalents mutants	49	18	15
% mutants killed by initial contracts	**10,3%**	**17,9%**	**8,7%**
% mutants killed after contracts improvement	**69,4%**	**91,4%**	**70,1%**

Table 2. Main results

Infected component	**p_date.e**	**p_time.e**
Total number of methods	19	12
Number of used/infected methods	14	11
# generated mutants	350	161
# equivalents	33	8
# killed mutants	195	114
% killed mutants	**61,51%**	**74,50%**

Table 3. p_date_time selftest robustness in an infected environment

The addition of new contracts thus improves significantly their capacity to detect internal faults (from 10 to 70 % for p_date, from 18 to 91 for p_time and from 9 to 70 for p_date_time). The fact that all the faults are not detected by the improved contracts reveals the limit of contracts as oracle functions. The contracts associated with these methods are unable to detect faults disturbing the global state of a component. For example, a prune method of a stack cannot have trivial local contracts checking whether the element removed had been previously inserted by a put. In that case, a class invariant would be adapted to detect such faults. However such contracts improvements are not always trivial to express and the effort spent for that task may be prohibitive compared to the gain in terms of test quality: dealing with test quality and contracts improvement is a difficult trade-off.

Concerning the robustness of a component against an infected component, **p_date.e** and **p_time.e** have been infected and **p_date_time** client class selftest launched. Table 3 gives the percentage of mutants detected by the client class selftest **p_date_time**. It gives an index of the robustness of **p_date_time** against its infected providers. The numbers of methods used by **p_date_time**, and thus infected by our mutation tool, are given as well as number of generated mutants for each provider class. The results show however that 60-80% of faults related to the external environment is locally detected by the selftest of a component.

The complexity of mutation analysis applied is linear with the number of statements in the methods. In fact, the maximum number of applicable mutation operators is an upper bound to the number of mutants that can be generated for a line of code.

The generation of mutants as well as the tests execution are automated processes. Moreover, during tests generation steps, only the last generated tests are applied on the already alive mutants: since the number of alive mutants decreases after each step of test generation, the global process speed increases with the improvement of test quality.

The most significant execution times are due to tests execution on mutants (mutant generation being made once for all). The test execution time on a mutant is short compared to the compilation time of the mutant. In an incremental process, the test execution time is shortened, since only new tests are applied to alive mutants. The compilation time is particularly short in the case of incremental compilation – as for example for Small Eiffel GNU compiler: only modifications need to be recompiled. On this example, the compilation and the execution mean time is close to 3 seconds per mutant, on a Pentium II machine.

The main lessons of this case study can be summarized in four points:

- the systematic use of a mutation tool for obtaining a test quality value is useful has a first index of trust since it provides a basic estimate that is not only "black and white" valued,
- the systematic improvement of tests and contracts implies a significant supplementary effort,
- a 100% Test Quality gives to the component a high ability to detect internal and external anomalies,
- the computational expense is reasonable for OO programs when the test qualification process through mutation is incremental.

6 Related works

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.

7 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 selftests 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

[1] Robert V. Binder, "Testing Object-Oriented Systems :Models, Patterns, and Tools", Addison-Wesley, October 1999. ISBN 0-201-80938-9.

[2] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection : Help For The Practicing Programmer," IEEE Computer, Vol. 11, pp. 34-41, 1978.

[3] R. DeMillo et A. Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Transactions On Computers, Vol. 17, pp. 900-910, 1991.

[4] D. E. Goldberg, "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989. ISBN 0-201-15767-5.

[5] J. H. Holland, "Robust algorithms for adaptation set in general formal framework", Proceedings of the 1970 IEEE symposium on adaptive processes (9th) decision and control, 5.1 –5.5, December 1970.

[6] William E. Howden and Yudong Huang, "Software Trustability", In proc. of the IEEE Symposium on Adaptive processes- Decision and Control, XVII, 5.1-5.5, 1970.

[7] B. F. Jones, H.-H. Sthamer and D. E. Eyres, "Automatic structural testing using genetic algorithms", Software Engineering Journal, pp. 299-306, September 96.

[8] Yves Le Traon, Daniel Deveaux and Jean-Marc Jézéquel, "Self-testable components: from pragmatic tests to a design-for-testability methodology", In proc. of TOOLS-Europe'99, TOOLS, Nancy (France), pp. 96-107, June 1999.

[9] Yves Le Traon, Thierry Jéron, Jean-Marc Jézéquel and Pierre Morel, "Efficient OO Integration and Regression Testing", to be published in IEEE Transactions on Reliability, March 2000.

[10] B. Meyer, "Applying design by contract", IEEE Computer, pp. 40-51, October 1992.

[11] J. Offutt, J. Pan, K. Tewary and T. Zhang, "An experimental evaluation of data flow and mutation testing", Software Practice and Experience, Vol. 26, No. 2, pp. 165-176, February 1996.

[12] S. A. Wadekar, S. S. Gokhale, "Exploring cost and reliability tradeoffs in architectural alternatives using a genetic algorithm", In proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton (Florida), pp. 104-113, November 1999.