Building trust into oo components using a genetic analogy

Benoit Baudry, Vu Le Hanh, Jean-Marc Jézéquel, Yves Le Traon

Building Trust into OO Components using a Genetic Analogy

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

Abstract

Despite the growing interest for component-based systems, few works tackle the question of the trust we can bring into a component.

This paper presents a method and a tool for building trustable OO 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). A component is seen as an organic set composed of a specification, a given implementation and its embedded test cases.

We propose an adaptation of mutation analysis to the OO paradigm that checks the consistency between specification/implementation and tests. Faulty programs, called "mutants", are generated by systematic fault injection in the implementation. The quality of tests is related to the mutation score, i.e. the proportion of faulty programs it detects. The main contribution of this is to show how a similar idea can be used in the same context to address the problem of effective tests optimization. To map the genetic analogy to the test optimization problem, we consider mutant programs to be detected as the initial preys population and test cases as the predators population. The test selection consists of mutating the "predator" test cases and crossing them over in order to improve their ability to kill the prey population.

The feasibility of components validation using such a "Darwinian" model and its usefulness for test optimization are studied.

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 the system, the way you design it". However, despite the growing interest due to this incremental way of building software, few research efforts 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 on 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[Binder99] for a detailed state of the art).

In [1], we presented a pragmatic approach for linking design and testing 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 [2], 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 cases 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 [3,4], 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 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 three facets. The methodology is an original adaptation from mutation analysis principles [5]: the quality of the test cases 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 effectiveness 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 basic test cases set is easy, improving its quality may require prohibitive effort. We describe how such a basic unit test cases set , seen as a test seed, can be automatically improved using genetic algorithms to reach a better quality level. This test improvement stage is called, in this paper, test optimization.

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. The last section briefly presents and discusses related work.

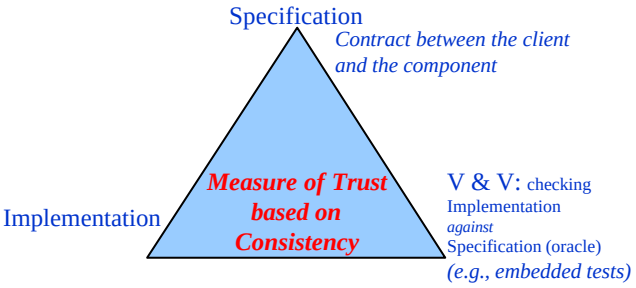


Fig. 1. Trust based on triangle consistency

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 transformed to 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 cases are defined as being an "organic" part of a software OO component. Indeed, a component is composed from 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 which 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 [1].

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 cases set, the effectiveness of test cases must be checked and estimated against implementation and specification, specially contracts. Tests are built from the specification of the component: they are a reflection of its precision. They are composed of two independent conceptual parts: test cases and oracles. Test cases execute the functions of the component. Oracles – predicates for the fault detection decision – 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 decisions are provided by the contracts, that are derived from the specification. The fact that contracts of the components are ineffective 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 effectiveness 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 effective contracts as oracle functions.

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 combinatorial expense. As it will be detailed in section 4, an incremental approach combined with the fact that OO methods code is often (or should be) small, makes the approach realistic and useful.

3. Mutation testing for OO domain

Mutation testing is a testing technique which was first designed to create effective test data, with an important fault revealing power [6,7]. It has been originally proposed in 1978 [5], and consists of creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a test cases 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 cases set is *relatively adequate* if it distinguishes the original program from all its non-equivalent mutants. Otherwise, a *mutation score(MS)* is associated with the test cases 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.

Table 1: Mutation operators set for OO programs

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

Our current choice of mutation operators includes selective relational and arithmetic operator replacement, variable perturbation, but also 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 cases detect the injected fault. The choice of mutation operators is given in Table 1.

Functionality of each of the mutation operators:

EHF: Causes an exception when executed. This semantically large mutation operator allows us to force code coverage.

AOR: Replaces occurrences of "+" by "-" and vice-versa.

arithmetic operator	replaced by
+	-, *
-	+, / (or div)
*	/ (or div), +
/	*, -
Div	-, mod
Mod	-, div

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: Constant 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.

MCR: 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 [8, 9, 6], the other operators having been introduced in this paper for the object-oriented domain. The data perturbation operator VCP allows to disturb state of data and to obtain a sensitivity analysis of program similar to [7]. Operator RFI introduces object aliasing and object reference faults, specific to object-oriented programming:

- reference to an object is stuck-at "**void**",
- object duplication instructions (**clone/copy**) are suppressed,
- each reference affectation of an object is preceded by the duplication of this object.

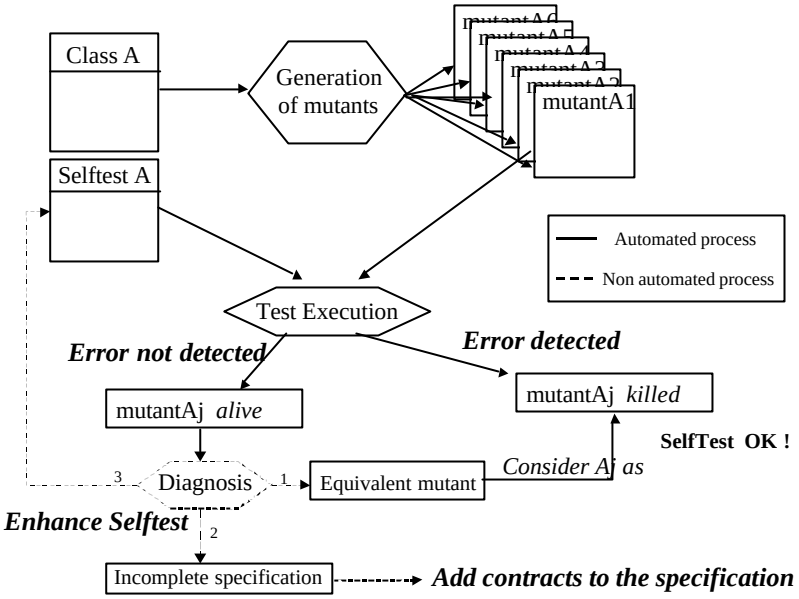


Fig. 2 The mSlayer tool in the global testing-for-trust process

The faults due to the RFI operator are more difficult to detect than the those due to other operators. Then, more complex and specific to OO domain errors have been taken thought about, such as switching an object with a brother object (common ancestor) for modeling polymorphic errors. In this paper, such operators were not used, nor implemented.

A reduced set of mutation operators is needed in terms of hard to detect errors (a < replaced by a <= is harder to detect than a replacement by a >): it means that some mutation are less meaningful than others and should be avoided.

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, the application of test cases against each mutant. One can choose to use as a decision the difference between the result of the initial implementation and the mutant result. One can choose to use contracts and embedded oracle function to make the decision. The analysis consists on the determination of 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 analysis, the process is completely automated.

The mutation analysis tool, called mutants slayer or μSlayer, is dedicated to 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 an analysis to determine which mutants were killed by tests. The process is incremental (for example, we do not start again the execution on already killed mutants) and is parameterized: for example, the user 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/.

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 cases to C_i, and m_i the total number of mutants.

The test quality, i. e. the mutation score MS, for C_i being given a unit test sequence T_i is defined as follows:

$$TQ(C_i, T_i) = \frac{d_i}{m_i}$$

The System Test Quality (STQ) is defined relatively to the d_i and m_i as follows:

$$STQ(S)= \frac{\sum_{i=1}^{n} d_i}{\sum_{i=1}^{n} m_i}$$

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

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

3.3. 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. Implementing such test cases into a self-testable class is a low-cost task. Our experiments showed 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 cases which are the predators of the mutant population. From this initial population, how can we mutate the "predators" test cases and cross them over in order to improve their ability to kill mutants programs. One of the major difficulty 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 effectiveness of a test case.

a) Genetic algorithms

Genetic algorithms [10] have been first developed by John Holland [11], 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 cold weather…) reproduce and thanks to crossover and mutation, the next generation will fit better. This is just how a genetic algorithm works: it uses an objective criteria to select the fittest individuals in one population, it copies them and creates new individuals with pieces of the old ones.

This objective criteria used to go from one generation to the other is one of the interesting points of genetic algorithms, but there are others. As we will see, these algorithms are computationally simple, they improve rapidly and they work at the population level, not on a single individual.

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 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
-mutation

- Reproduction copies the individuals which are going to participate in crossover: they are chosen according to their F(x) value. The choice can be seen as spinning a roulette wheel where each individual has a slot proportional to its fitness value. We spin the wheel as many times as the size of the population, and so we have a new population which is going to participate to crossover. This new population is made of individuals of the old one, and the number of each type of individual is proportional to its fitness (there are many of the fittest and few of the ones with a low fitness).
- Crossover : the members of the population after reproduction are mated randomly, then every pair is crossed, to create as many new pairs, like this : first, you choose, at random, an integer value k between 0 and the size n of an individual less one. Secondly, you create two new individuals A' and B' with a pair (A,B), A' is made of the k first genes of A and n-k last genes of B, and B' is made of the k first genes of B and the n-k last genes of A.
- The 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. mutation on one or several individuals
6. back to step 2.

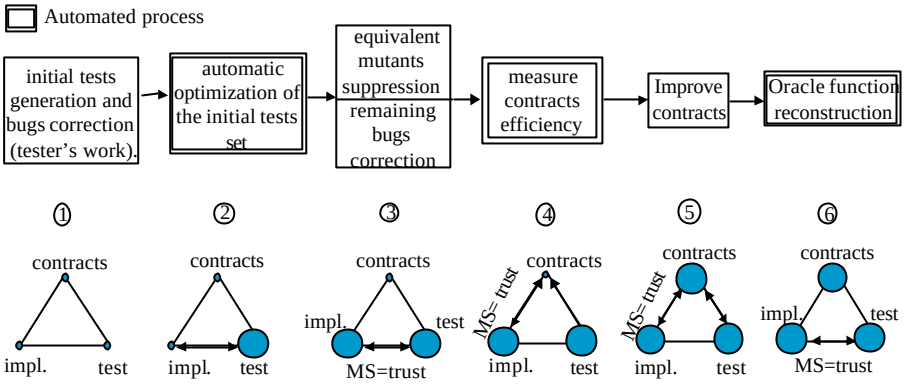


Fig 3. The global testing for trust process

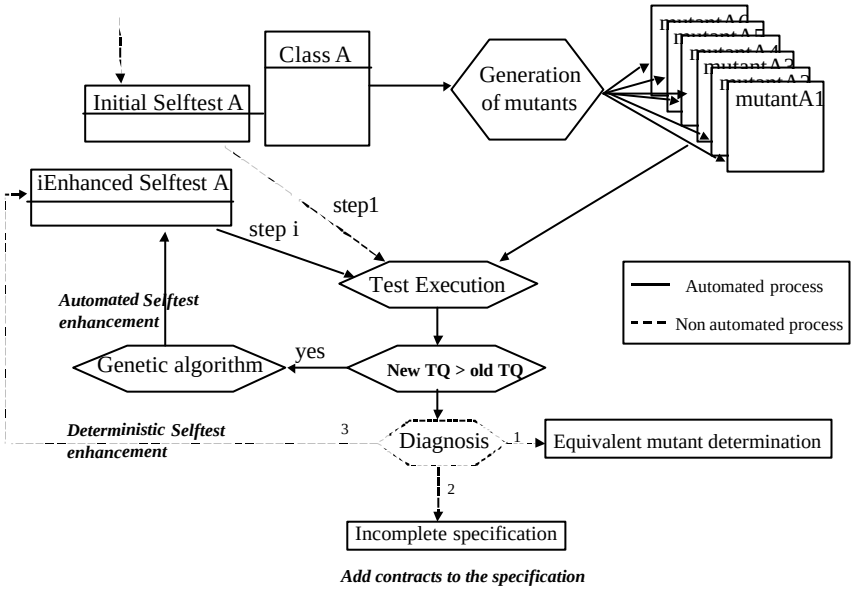


Fig 4. The global testing-for-trust process with automated genetic test enhancement Case study

b) Genetic algorithm for test 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 case, and genes are test method calls. We consider a test as a sequence of initialization and method calls. The initialization, the same for every test, prepares the system to accept the method calls.

Notations

Test : 1 test = 1 gene = [an initialization sequence, several method calls]

Gene : G = [I , S] and S = $(m_1(p_1),\ldots,m_n(p_n))$

Individual : An individual is defined as a finite set of genes = $\{G_1,\ldots,G_m\}$. It is equivalent to what we call a test case.

The size of an individual m should not change, whereas a gene's size m might change(by calling more or less methods in one test). Moreover, as we will see later, the size of the population changes, we will have to make it grow from one generation to the other.

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:

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

$ind_1 = \{G_{1\,1}, \ldots G_{1\,i}, G_{1\,i+1}, \ldots G_{1\,m}\}$ $\quad ind_2 = \{G_{2\,1}, \ldots G_{2i}, G_{2\,i+1}, \ldots G_{2\,m}\}$

$ind_3 = \{G_{1\,1}, \ldots G_{1\,i}, G_{2\,i+1}, \ldots G_{2\,m}\}$ $\quad ind_2 = \{G_{2\,1}, \ldots G_{2i}, G_{1\,i+1}, \ldots G_{1\,m}\}$

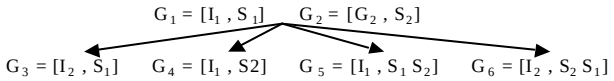
- Mutation : we use two mutation operators. The first one changes the method call parameters values in one or several genes.

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

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

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 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.



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.

Thanks to the automation of a part of test generation, we give a six steps process for the global design of trustable components as shown Figure 3.

1. At first, the programmer writes an initial selftest that reaches a given initial Mutation Score (MS).
2. This step aims at automatically enhancing the initial selftest. We propose to use genetic algorithms for that purpose (Fig. 4.), but any other technique could be used. The used oracle function is the comparison between the testing object states.
3. During the third step, the user has to check if the tests do not detect errors in the initial program. If errors are found, he must debug them.
4. The fourth steps consists in measuring the contracts quality thanks to mutation testing. We use the embedded contracts as an oracle function here.
5. Then a non-automated step consists of improving contracts to reach an expected quality
6. At last, the process constructs a global oracle function. To do this, it executes all the tests on the initial class, and the object's state after execution is the oracle value.

c) 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 might 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 faulty tests are detected by the compiler.

4. 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.

4.1. 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 effectiveness 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.

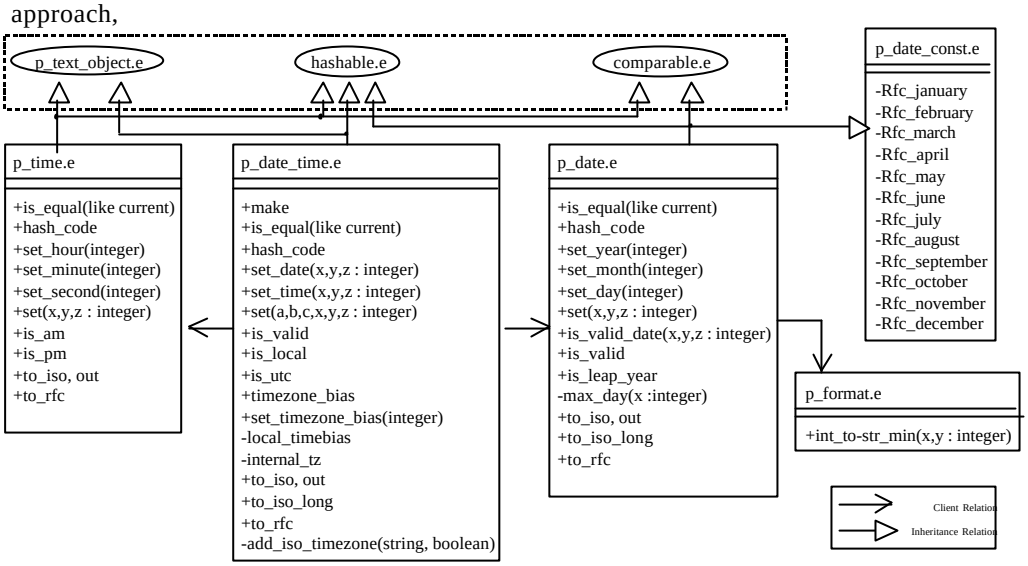


Fig. 5 Classes of package "date-time"

The two first aspects concern the feasibility of the overall approach in terms of effort. Concerning point 2, to make clear the differences between test generation effort and contracts improvement, each generated test data is associated to a specific oracle assertion in the test program: these dedicated oracle assertions allow a perfect decision. Faults in the program are thus systematically detected if test data exercise them. Then, in a second stage, when a 100% mutation score is reached, these dedicated oracle assertions are suppressed and the proportion of mutants detected by contracts is measured. This provides a simple way of estimating the effectiveness 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 effectiveness 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 verifies its own correctness but also some of its neighboring providers components.

The analysis focus 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.

4.2. Results and lessons learned

Results for the three first points are presented in Table 2. For the classes that are studied here, this first stage of generation allowed the elimination of approximately 60 to 70% of the generated mutants. It corresponds to the test seed that can be used for automatic improvement through genetic algorithms optimization (see Section III.3). Figure 5 presents the curves of the mutation score as a function of the number of generated predators (one point represents a generation step). To avoid the combinatorial expense, we limit the new mutated generation to the predators that had the best own mutation score (good candidates). We also only mutate test genes including method calls corresponding to non-detected mutants. 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.

The mutation score has been improved by analyzing the mutants one by one: specific tests cases were written for alive mutants to reach a 100% mutation score.

Concerning point 2, the results show that even if contracts are improved, they are still local properties and they cannot completely replace these deterministic dedicated oracle assertions in the selftest program. Indeed, a given test suites may lead the component under test in a particular global state, and local contracts cannot easily check the global state correctness. At the end of the improvement process, the self-testable component has a considerable greater capacity to detect faults (between 64 and 90 % in the case of mutation faults for this study). As a result, this approach points out methods whose contractual definition is too weak.

Equivalent mutant detection is not automatic and requires a human decision to compare between the line of code initial and the mutant line. For the studied example, effort to determinate equivalence is almost negligible compared to the effort to produce test cases: the methods have a well-defined semantics; methods have a low complexity and equivalence is easy to determine by simple comparison between initial and mutant code.Concerning the improvement of contracts, the results of the initial quality of contracts used as oracles are given in Table 2. The table recapitulates the initial effectiveness of contracts and then the final level they reached after improvement.

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 8 to 70 for p_date_time). The fact that all 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, Table 3 gives the percentage of mutants detected by the client class selftest **p_date_time** when **p_date** and **p_time** are faulty. This percentage 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. However, the results show that 60-80% of faults related to the external environment are locally detected by the selftest of a component.

Table 2. Main result

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

The study shows that the majority of mutants are detected easily by an initial test case (60-70% approximately). This reveals that even if a 100% score is not an objective, the approach provides a useful index to estimate the quality of basic unit tests. Indeed, if “programmers love writing test” [Beck98], estimating test quality provides an interesting satisfaction... The genetic algorithms applied to an initial test case aims at improving it, considered as a seed, by composing and applying some kind of mutation on the test cases themselves.

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 on the number of mutants that can be generated for a line of code.

The generation of mutants as well as the test execution are automated processes. Moreover, during test generation steps, only the last generated test cases are applied to 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.

On the studied example, for the three classes, test generation and contract improvement required 6 person-days for contract improvement.

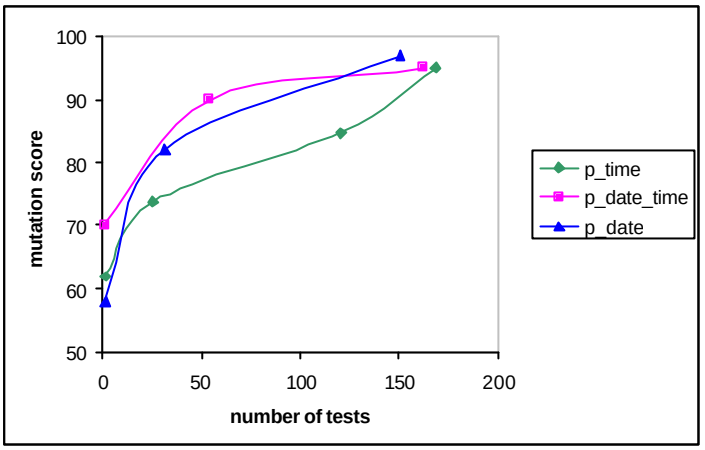


Fig.6 . Genetic algorithm results for test optimization

Table 3. p_date_time robustness in an infected environment

Infected component	**P_date**	**p_time**
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%**	**74%**

The most significant execution times are due to test execution on mutants (mutant generation being made once for all). The test execution time of 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 the Small Eiffel GNU compiler: only modifications need to be recompiled. For 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 use of genetic algorithms significantly reduces the test generation effort since only a simple initial test case seed is needed for automated test improvements,
- the computational expense of genetic algorithms still remains a problem,
- 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.

5. Related work

While electronic devices have set of measures characterizing their quality (reliability, performance, use-domain, speed scale), no real consensus exists to measure such quality characteristics for software components. Binder details the existing analogy between hardware and OO software testing and suggests an OO testing approach close to the "built-in-test" and "design-for-testability" hardware notions [12]. In this paper, we go even further than Binder suggests, and detail how to create self-testable OO components, with an explicit analogy with the "built-in-self-test" hardware terminology. Moreover, 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 [6, 8] as follows:

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

Besides, the test problem may be seen from a pragmatic point of view, and some simple-to-apply methodology can be found in the literature, which are based on an explicit test philosophy [13]. 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 [1], the design-for-testability main methodology is outlined. In this paper, we detailed the testing-for-trust method while [14,15] 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 [16], genetic algorithms are used in a control-flow coverage-oriented way: test sets are improved to reach such a predefined test adequacy criterion. In [17], genetic algorithms are used to perform some kind 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 has the robustness of trustable and self-testable components into 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 (the revealing power of the test cases [Voa92]), we seek to build trust in a component passing those test cases.

References

[1] Y. Le Traon, D. Deveaux and J-M. Jézéquel, "Self-testable components: from pragmatic tests to a design-for-testability methodology," In proc. of TOOLS-Europe'99. TOOLS, June 1999, Nancy (France) 96-107.

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

[3] B. Meyer. Applying ``design by contract". IEEE Computer, pages 40--51, oct 1992.

[4] J-M. Jézéquel, M. Train and C. Mingins. Design-Patterns and Contracts. Addison-Wesley, 0ctober 1999. ISBN 0-201-30959-9.

[5] 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.

[6] J. Offutt, J. Pan, K. Tewary and T. Zhang "An experimental evaluation of data flow and mutation testing," Software Practice and Experience, v 26, n 2, February 1996.

[7] J. Voas et K. Miller, "The Revealing Power of a Test Case", Software Testing, Verification and Reliability, vol. 2, pp. 25-42, 1992.

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

[9] A. J. Offutt, "Investigation of the software testing coupling effect", ACM Transaction on Software Engineering Methodology, vol. 1, pp. 3-18, 1992.

[10] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, 1989.

[11] J. H. Holland, "Robust algorithms for adaptation set in general formal framework", ", In proc. of the 5th International Symposium on Software Reliability Engineering (ISSRE'94), October 1994, Monterey (California), 143-151.

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

[13] K. Beck and E. Gamma, "Test-Infected: Programmers Love Writing Tests," Java Report, July1998, 37-50.

[14] T. Jéron, J-M. Jézéquel, Y. Le Traon, and P. Morel, “Efficient Strategies for Integration and Regression Testing of OO Systems”, In proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE’99), November 1999, Boca raton (Florida), 260-269.

[15] Y. Le Traon, T. Jéron, J-M. Jézéquel and P. Morel, “Efficient OO Integration and Regression Testing”, to be published in IEEE Transactions on Reliability, March 2000.

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

[17] 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), November 1999, Boca Raton (Florida)