Trustable Components: Yet Another Mutation-Based Approach

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

This paper presents the use of mutation analysis as the main qualification technique for:
- *estimating and automatically enhancing a test set (using genetic algorithms),*
- *qualifying and improving a component's contracts (that is the specification facet)*
- *measuring the impact of contractable robust components on global system robustness and reliability.*

The methodology is based on an integrated design and test approach for OO software components. It is dedicated to design-by-contract, where the specification is systematically derived into executable assertions called contracts (invariant properties, pre/postconditions of methods). 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. 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

The *Object-Oriented* approach offers both strong encapsulation mechanisms and efficient operators for software reusability and extensibility. In a component-based approach using a *design-by-contract* methodology, the following considerations make mutation analysis useful for several analysis levels:
- in a design-by-contract approach [5,10], components integrate "contracts" that are systematically derived from the specification. Contracts behave as executable assertions that automatically check the components consistency (pre-postconditions, class invariants). Based on mutation analysis, the efficiency of contracts can thus be estimated by their capacity of rejecting faulty implementation, and the enhancement of contracts can be guided. Then, and also based on a particular application of mutation analysis, the contribution of each component to the global system robustness and reliability can be estimated.
- Components, to be reusable, are considered as an "organic" set of a specification, an implementation and embedded tests. With such self-testable component definition, all the difficulty consists of automatically improving embedded tests based on the basic test cases written by the tester/developer. Being given these basic test cases, we consider mutants programs as a population of preys and, conversely, a test set as a particular predator. This analogy leads to the application of genetic algorithms to enhance the original population of predators using as a fitness function the mutation score.
- Trustability [4] is finally the result of the global packaging of a design-by-contract approach, component self-testability and mutation analysis for both tests & contracts improvement and qualification are

In this paper, we propose a testing-for-trust methodology that helps checking the consistency of the component's three facets, i.e., specification/ implementation and tests. The methodology is an original adaptation from mutation analysis principle [1]: the quality of a tests 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 anomalies in the implementation or in the provider environment is thus tackled and studied (Section 4). If the generation of a basic tests 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 tests 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). Section 5 is devoted to an instructive case study that illustrates the feasibility and the benefits of such an approach. Section 6 presents a robustness measure, for a software component, based on a mutation analysis.

2.   Test quality for trustable components

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 [6].

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 of two independent conceptual parts: test cases and oracles. Test cases execute the functions of the component. Embedded 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 contracts derived from the specification. The fact components' contracts 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.
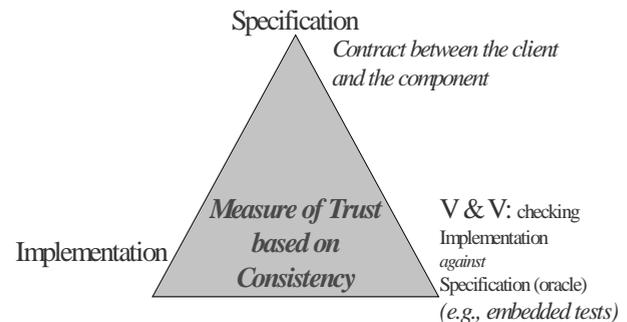

Fig. 1. Trust based on triangle consistency

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 combinatorial expense.

The global component design-for-trust process consists of 6 steps that are presented in figure 2.
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, 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 step 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.
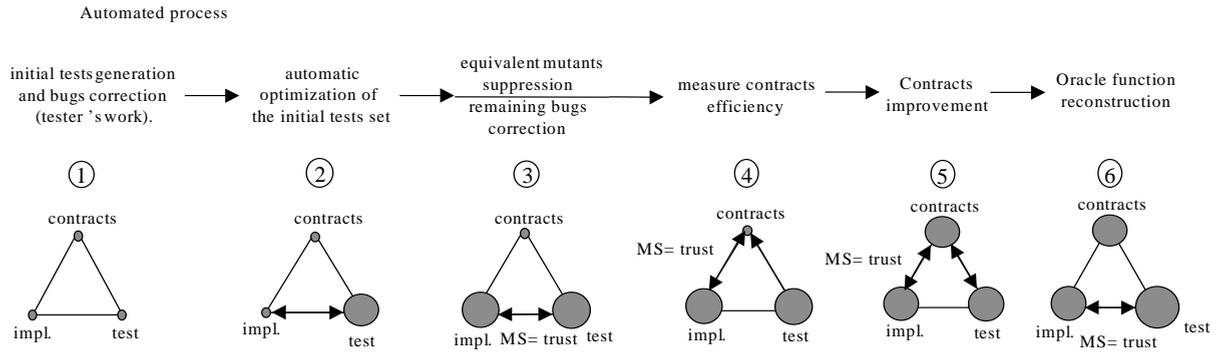
Fig. 2.The global testing-for-trust process

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 [1] , and consists in creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a tests 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 tests 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.

A mutation analysis seems well adapted to the Object-Oriented domain for the following reasons:
- methods body of a well designed OO component are generally shorter than for a procedural implementation, most of the control predicates being dispatched on the system dependencies: combinatorial explosion of a mutation analysis is thus limited;
- in OO paradigm, the executed program is an object with a state (attributes values and recursively states of the referenced objects): in classical mutation analysis, the oracle is obtained by comparison between the explicit outputs of the original program and the mutant. In the case of OO programming, an oracle can easily be built by comparing the states of the initial program with the state of the mutant one (a deep comparison of the object states). In fact, to avoid the problem of stateless programs (or if the injected fault does not affect the state of the object under test) the object states that will be compared are the testing programs themselves: the testing program is an object, where all queries method calls on the class under test are caught by attributes of the testing class. With this solution an efficient oracle function compares testing objects attributes. This integrated mechanism significantly enlarge the spectrum of programs concerned by a mutation analysis (no specific instrumentation of the source code is needed)

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 is the following:

EHF: Causes an exception when executed

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.

The operators introduced for the object-oriented domain are the following:

- MCP (Methods Call Replacement): Methods calls are replaced by a call to another method with the same signature.
- RFI (Referencing Fault Insertion): Stuck-at void the reference of an object after its creation. Suppress a clone or copy instruction. Insert a clone instruction for each reference assignment. 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 includes the generation of mutants and the application of test cases against each mutant. The decision 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 on alive mutants consists in determining the reason of 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. 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/.

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. Test cases generation : genetic algorithms for test generation

In this section we present the results obtained after using a genetic algorithm as a way to automatically improve the basic test cases set in order to reach a better Test Quality level with limited effort. We begin with a population of mutant 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 mutate 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.

Genetic algorithms [2] have been first developed by John Holland [3], 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.

For test optimization, the problem is modeled as follows:

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 = {$G_1,…,G_m$}

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 that manipulate the individuals and genes in our problem:

- Reproduction: selection of individuals that will participate to the next generation guided by the individuals' mutation score.
- Crossover: we select at random an integer i between 1 and individual's size, 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.
- Mutation: we use two mutation operators. The first one changes the method call parameters values in one or several genes. 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 Variable and Constant Perturbation 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, or by switching the genes initialization sequences.

The genetic algorithm is applied until the Quality Test (i. e. the mutation score of the whole set of individuals) level is no more improved.

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.

This study proceeds in two stages to help 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.

The aims of this case study were:
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 detect faults due to external infected provider classes.

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 crosschecking 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 three classes: **p_date_time.e, p_date.e** and **p_time.e.**

For 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 3 presents the curves of the mutation score growth as a function of the number of generated predators (one plot represents a generation step). To avoid the combinatorial expense, we limit the new mutated generation to the predators that 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.

Table 1. Main results

	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%**

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 1. The table recapitulates the initial efficiency of contracts and then the final level they reached after improvement.

Table 2. p_date_time robustness in an infected environment

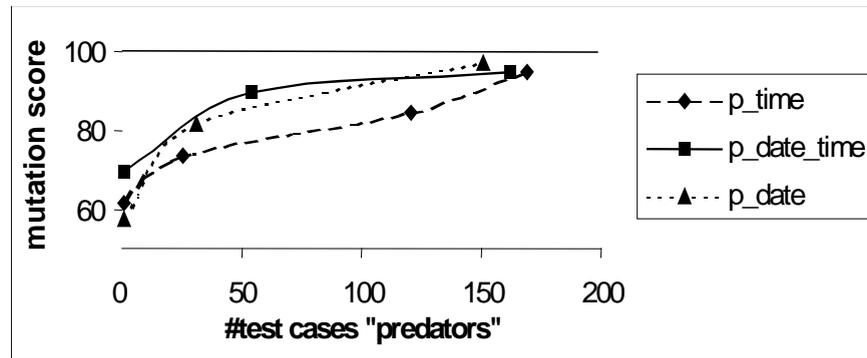| Infected component	P_date	p_time
Total number of methods	19	12
Number of used/infected methods	14	11
# generated mutants	350	161
# equivalents	33	8
# killed mutants	195	114
% killed mutants	**61%**	**74%**

Fig.3. Genetic algorithm results for test optimization

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 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.

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 2 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.

6. Reliability and Robustness of a designed by contract system

Based on mutation analysis, we propose a first approximation of the initial failure rate that could be used in a reliability model for initializing some of the initial constant parameters [8,9]. We do not look for a new reliability model but the argumentation aims at bridging the gap between testing and initial reliability/robustness of a system in a design-by-contracts approach, with self-testable components.

Embedded contracts, as executable assertions derived from a specification, provide a mechanism to detect faults before they provoke a failure. We analyze the initial reliability of a system, tested using mutation analysis, and the robustness reached by a system using contractable components versus no contractable ones.

Let C_i, $i \in [1..n]$ be the n components of a system.

Let F_i^0, the initial failure rate (after validation steps), i.e., the probability that a failure occurs in the component in the next statement execution. The initial reliability R_i^0 of the component is thus: $R_i^0 = 1 - F_i^0$

In a mutation analysis approach, the test cases have been executed against all the mutant programs. Recall that we consider a component as an organic set of a specification (contracts), an implementation and the embedded test sets. A component has a good behavior if

its tests are able to detect failure coming from the implementation. So, the number of killed mutants represents a number of successful behavior of the component, since the known injected faults have been successfully detected by the selftest. To measure the initial reliability, the assumption is the following: we assume that if a new test case is executed, then a failure will certainly occur. With such an assumption if the number of statements executed before the faulty code is infected is $Nstat_i$, then the initial failure probability is

$$F_i^0 = 1 / Nstat_i .$$

In this paper, we estimate $Nstat_i$ by multiplying the number of statements executed in the correct program by the number of killed mutants. It provides a satisfying approximation of the number of executed statements. So we have: $Nstat_i \cong \#killed\_mu\tan ts_i \times K_i$, where K is the number of statements executed by the test set on the program.

The global initial reliability of a system composed of n components can thus be estimated in two ways depending on fault independence assumption. First, we can approximate the reliability by considering that failure events occurring in the system are independent. With such a (pessimistic) assumption, the initial reliability R^0 of a system is equal to : $R^0 = \prod_{i=1}^{n} R_i^0$.

Another consideration would lead to a more realistic model, by considering that one statement will be executed at a time (indeed this no more true for parallel and distributed software). Under that optimistic assumption, we have the following initial reliability and failure rate:

$$R^0 = 1 - F^0 = 1 - \frac{1}{\displaystyle\sum_{i=1}^{n} Nstat_i} = 1 - \frac{1}{\displaystyle\sum_{i=1}^{n} \frac{1}{F_i^0}}$$

Both models provide boundaries of the initial reliability and failure rates.

Independently of the way these factors are measured, the robustness Rob_i of a component C_i is defined here as the probability that a fault is detected, assuming that this fault would provoke a failure if not detected by a contract or an equivalent mechanism. Conversely, the "weakness" $Weak_i$ of the component is equal to the probability that the fault is not detected. This probability corresponds to the percentage of faults detected by contracts. Indeed, if the component has been designed by contracts, then the detected fault can be retrieved, and a mechanism (such as exception handling and processing) will prevent a failure to occur. In the case of a component C_i with no contracts, its robustness is equal to 0: $Rob_i = 1 - Weak_i = 0$.

A component isolated from the system has a basic robustness corresponding to the strength of its embedded contracts. A component plugged into a system has robustness enhanced by the fact that its clients will add their contracts to the fault detection. The notion of Test Dependency is thus introduced for determining the relation between a component and its client and heirs in a system.

Test dependency : A component class C_i is *test-dependent* from C_j if it uses some objects from C_j or inherits from C_j. This dependency relation is noted:

$$C_i \ R_{TD} \ C_j$$

If $C_i \ R_{TD} \ C_j$, then the probability that C_i contracts detect a fault due to C_j is noted Det_j^i. To estimate this probability, one can use the proportion of mutants detected by C_i while C_j is infected. Even though the test dependency relation is transitive, we only consider faults that are detected by a components directly dependent from the faulty one.

The robustness $Rob\_intoS_i (= 1 - Weak\_intoS_i)$ of the component into the system S –and so enhanced by the client components contracts- is thus :

$$Rob\_intoS_i = 1 - (Weak_i \cdot \prod_k (1 - Det_i^k)), \quad k/C_k \ R_{TD} \ C_i$$

Finally, the robustness $Rob \cdot$ of the system is thus equal to:

$$Rob = 1 - Weak = 1 - \sum_{i=1}^{n} Prob\_failu\,re(i) \times Weak\_intoS_i$$

where *Prob_failure(i)* is the probability the failure comes from the component C_i knowing that a failure certainly occurs. This probability is approximated by the component's complexity.

To conclude, considering that a fault detected by a contract allows the service continuity, the initial reliability of the system is also enhanced as follows: $R_{new}^0 = 1 - F_{new}^0 = 1 - (F^0 \cdot Weak) = R^0 + F^0 \cdot Rob$.

Fixing the values

The parameters of this model of robustness and initial reliability are easily fixed using mutation analysis.

$$R_i^0 = 1 - F_i^0 \text{ with } F_i^0 = 1 / Nstat_i$$

$Rob_i = 1 - Weak_i = $ percentage of mutants detected by contracts.

$Det_i^j = $ percentage of mutants in C_j detected by C_i contracts.

Prob_failure(i) $= 1/n$

Illustration

To estimate (roughly) the gain in robustness of a system, we consider the SMDS system composed of 37 components, already studied in [7] for optimizing integration testing. Here, we apply our robustness estimator for this system. The detailed model is not needed to understand the application to robustness. We fix the values as follows to appraise the global improvement in robustness of the system due to a systematic use of contracts:

$$Rob_i = 1 - Weak_i = 0.85, \qquad Det_i^j = 0.7 \qquad \text{and}$$

$Prob\_failure(i) = 1/n = 1/37$.

As a mean estimator, we consider that a test case is composed of mean 100 tests and that 200 mutants are generated for each of the component. A test approximately executes 10 statements.

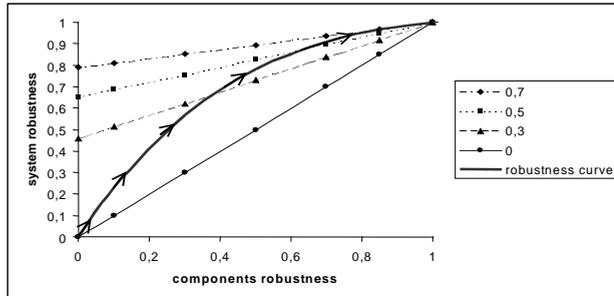The initial failure rate of the system composed of 37 components is thus equal to $F^0 = 1.35 \cdot 10^{-7}$.

Fig. 4. System Robustness depends on Components Robustness

Figure 4 first shows four evolutions of the SMDS system robustness depending on the components robustness. The four curves correspond to four different Det_i^j values. This figure shows that contracts, by enhancing a component's robustness and by enhancing the Det_i^j value, improve the global system's robustness.

Since it is obvious that a relationship exists between Det_i^j and the robustness, the figure 4 also displays the "robustness curve". To draw this curve, we have considered that Det_i^j is related to the robustness by a linear function such as: $Det_i^j = K \cdot Rob$. Here we have taken K equal to 0.8. So this curve really corresponds to the real global robustness evolution: during the development phase, the programmer will start with basic weak contracts and then enhance them. So during this period, the robustness of components and Det_i^j will grow together and so will the system's robustness.

7. Conclusion

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 [12]) we seek to build trust in a component passing those test cases. The analysis also shows that a design-by-contract approach associated to the notion of embedded selftest significantly improves the robustness, and indirectly the reliability, of a final-product.

References

[1] 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.

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

[3] 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.

[4] 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.

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

[6] 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.

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

[8] M. Lyu, "Handbook of Software Reliability Engineering", McGraw Hill and IEEE Computer Society Press, 1996, ISBN 0-07-0349400-8.

[9] J. D. Musa, A. Iannino, K. Okumoto, "Software Reliability: Measurement, Prediction, Application", McGraw Hill, 1987, ISBN 0-07-044093-X.

[10] B. Meyer, "Applying design by contract", *IEEE Computer*, Vol. 25, No. 10, pp. 40-52, 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] J. Voas, "PIE: A Dynamic Failure-Based Technique", *IEEE Transactions on Software Engineering*, Vol.18, pp. 717-727, 1992.