Robustness and Diagnosability of OO Systems Designed by Contracts

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

Abstract

While there is a growing interest for component-based systems in industry,little effort has so far been devoted to quality evaluation of these systems. This paper presents the definition of measures for two quality factors, namely robustness and "diagnosability" for the special case of OO systems for which thee approach known as Design by Contract has been used. The main steps in constructing these measures are given, from informal definitions of the factors to be measured to the mathematical model of the measures. To fix the parameters, experimental studies have been conducted, essentially based on applying mutation analysis in the OO context. Several measures are presented that reveal and estimate the contribution of contracts quality and density to the overall quality of a system in terms of robustness and "diagnosability".

1. Introduction

Contracts are elements of formal specification associated to programs in a way that is acceptable to practicing developers [Meyer00]. Contracts have reactive support in Object Oriented languages such as the UML or Eiffel, and can be easily supported in Java (iContracts) or C++.

The objective of this paper is to bridge the gap between intuitive understanding of what contracts (or to enlarge the scope of the paper, use of assertions) improve in the software and a quantitative, and hopefully accurate, estimate of these improvements. We voluntarily restrict the problem domain to the design by contract approach and propose two measurements: one of the robustness, and the other of what we call "diagnosability" of the software [Le Traon98]. Indeed, these factors are related to a contract-based design approach: software with embedded contracts can detect internal anomalies during execution (the system is thus more robust) and helps in pinpointing the fault location (the faulty state can be expected to be close to the fault cause). At this level of understanding, the diagnosability factor can be roughly defined as the degree to which the software allows an easy and precise location of a fault when detected.

This paper mainly reports on the empirical validation of an axiomatization of the behavior of these robustness and diagnosability factors. This validation seems to be successful since the case studies reveal that the measures correspond closely to the intuition. The proposed robustness and diagnosability measures offer an easy way of comparing designs, as well as a method for appraising contract efficiency in terms of both robustness and diagnosis effort and preciseness. These estimates allow the effort which must be devoted to contract quality and quantity in order to reach a certain level of robustness and diagnosability to be predicted. The measures presented, which are based on a generic axiomatization of their expected behavior, can be generalized to classical procedural programming.

Section 2 opens with a presentation of the design by contract approach and an intuitive analysis of some expected benefits of the approach: robustness and diagnosability improvement. Section 3 concentrates on the definition of robustness, axiomatization of the expected measurement behavior and the calibration of the model parameters on several case studies. Concerning the calibration of the model parameters, we use a particular adaptation of mutation analysis to the OO paradigm. Section 4 is devoted to diagnosability analysis, along the same lines as that given for robustness.

This paper is quite dense, but we prefer detail each measurement elaboration, since we believe that any realistic software measurement cannot be too simplistic.

2. The Problem Domain: Design by Contract

2.1. Design by contract

The notion of software contract has been defined to capture mutual obligations and benefits among classes. Experience tells us that simply spelling out unambiguously these contracts is a worthwhile design approach [Jezequel97], that B. Meyer cornered the *Design by Contract* approach to software construction [Meyer92]. This design by contract approach has a sound theoretical basis in relation to partial functions, and provides a methodological guideline for building robust, yet modular and simple systems. In some ways, design by contract is the exact opposite of defensive programming [Liskov86] where it is recommended to protect every software module by as many checks as possible.

Defensive programming makes it difficult to precisely assign responsibilities among modules, and has the additional malevolent side-effect of increasing software complexity, which eventually leads to a decrease in reliability.

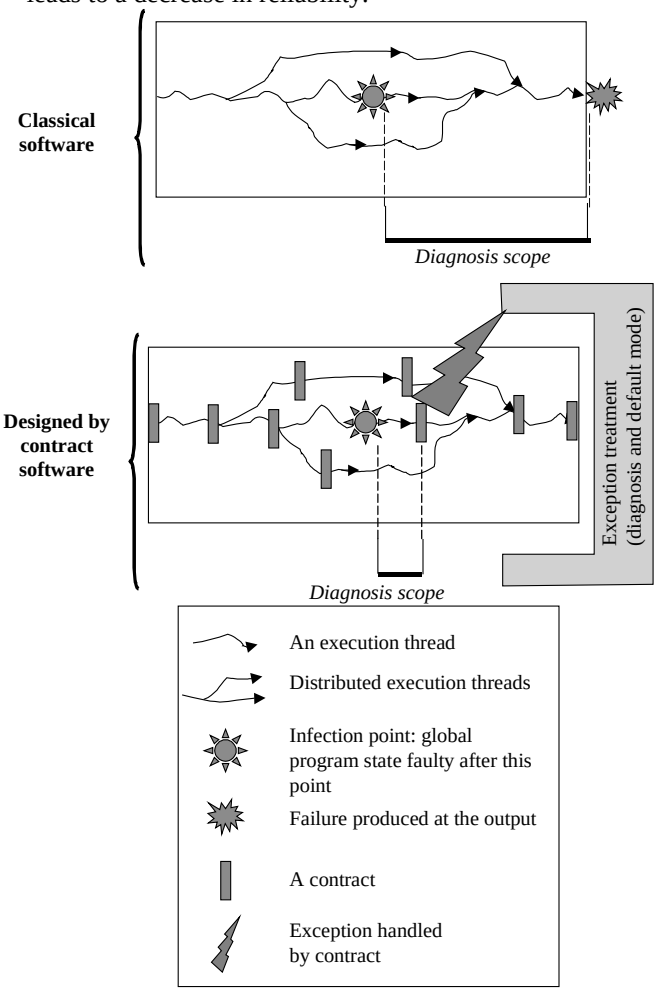

Fig. 1. **Contracts for early detection of a fault**

The *design by contract* approach prompts developers to specify precisely every consistency condition that could go wrong, and to explicitly assign the responsibility of its enforcement to either the routine caller (the client) or the routine implementation (the contractor). Along the line of abstract data type theory, a common way of specifying software contracts is to use boolean assertions called pre-and post-conditions for each service offered, as well as class invariants for defining general consistency properties. A contract carries mutual obligations and benefits: the client should only call a contractor routine in a state where the class invariant and the precondition of the routine are respected. In return, the contractor promises that when the routine returns, the work specified in the postcondition will be done, and the class invariant is still respected.

A failure to meet the contract terms indicates the presence of a fault, or bug. A precondition violation points out a contract broken by the client: the contractor does not then have to try to comply with its part of the contract, but may signal the fault by raising an exception. A postcondition violation indicates a bug in the routine implementation, which does not fulfill its obligations.

The Design by Contract approach is smoothly integrated into the type system of an OO language through the notion of subcontracting as provided by the inheritance mechanism. That is, dynamic binding can be viewed through the perspective of a routine subcontracting its actual implementation to a redefined version. Redefinition is then a semantics-preserving transformation in the sense that the redefined routine must at least fulfill the contract of the original routine, and optionally does more (e.g., accepting cases that would have been rejected by the original contractor or returning a ‘‘better” result than originally promised). In other words, it means that in a subclass, preconditions may only be weakened (accept more) and postconditions strengthened (do more).

2.2. Contracts for Robustness and Diagnosability

In this paper, we focus on measuring the benefit of a design by contract approach. The main impact of contracts on final-product software quality is two-fold:

- since contracts can be compiled in such a way to raise exceptions when violated, they behave as classical, but more meaningful, assertions: a faulty program state during execution (due to a fault or bug) can thus be automatically detected, and the failure that would have certainly occurred can be avoided. Intuitively, it shows

that contracts participate to software robustness. The question is: to what degree do contracts contribute to software robustness, depending on their "strength" and number?

when a contract is violated, it indicates the part of the code where a wrong program state has been detected. With no contract embedded in the software, the failure would have been detected elsewhere, perhaps at the system output. Since the scope of diagnosis (the number of statement in which the fault must be located) is reduced when contracts catch the presence of a fault, it can be seen that contracts help the diagnosis task. The question is: to what degree do contracts reduce the diagnosis effort depending on their "strength" and number.

As presented in Figure 1, a design-by-contract software should allow early detection of faults (during their propagation to the outputs) and help locating the faulty part of the software by reducing the "diagnosis scope" in which a fault must be located. What is of great interest with contracts is the fact that their efficiency is not dependent on possibly distributed execution of the software. While faults are really difficult to locate in a distributed execution environment, using contracts there is a good probability if pinpointing where the fault occurred.

2.3. Measurement construction

The literature emphasizes the difficulty in constructing valid measurements [Fenton86, Shepperd93, Briand96, Kitchenham95]. In this paper, since the measured factors first appear as quite abstract and unclear, we choose to make the axiomatization of the measures [Shepperd93]. Figure 2 illustrates the process of the measurement construction: the factor to be measured is first informally defined and significant and measurable attributes are identified (with intuitive and hopefully convincing arguments and assumptions). Then the intuitive properties of the factor behavior must be expressed using the chosen attributes: this is what we call axioms. An axiom is an expected and understandable property of the measurement that also has a meaning in the mathematical model. It makes the connection between the intuitive/real world and the formal one for the theoretical validation. A formal model of measurement, that is richer than the expressed axioms (otherwise, the formal model is of no interest), can then be proposed. The role of axiomatization is twofold: it provides a unifying framework for evaluating the various measures that can be proposed, and thus helps one to separate the search for a measurement from the statement of the intuition (the expected behavior).

Since the axioms formalize the essential expected properties of any measurement, they define which systems should be comparable and the generic characteristics these measures must satisfy. The axioms constitute the basis of the theoretical evaluation, which was carried out for checking the consistency of the proposed measurement with the real world. The theoretical evaluation precedes empirical evaluation since it is less time-consuming and more appropriate to show that the model is internally consistent: it is used to detect that no pathological structures exist for which the model produces inappropriate behavior.

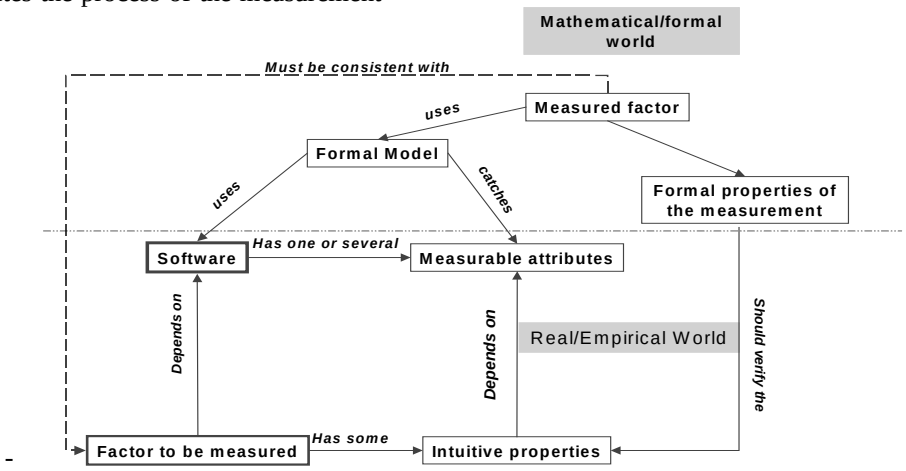

Fig. 2. The measurement elaboration

3. Measuring robustness improvement

In this section, we want to model the relationship between a component's robustness and its contracts. Therefore we propose a measure of contracts efficiency, and we show the improvement in robustness brought by contracts.

3.1. Definitions

Definition- Robustness**:** *Robustness expresses the degree to which the software is able to recover from internal faults that would otherwise have provoked a failure.*

In this paper, we consider that the main attribute that is significant for robustness is the capability of the software to detect an internal faulty state.

Definition - Isolated Robustness (Rob_i**):** *The isolated robustness Rob_i of a component C_i in a system S is defined as the probability that a fault internal to C_i is detected by C_i when it is known that this fault would provoke a failure. Conversely, the "weakness" $Weak_i$ of the component is equal to the probability that the fault is not detected.*

The detection mechanisms we focus on are executable contracts and other assertions. Many components cannot directly be executed (abstract or generic classes). Nevertheless, they may still be equipped with their own contracts that can detect their internal failure.

Definition - Global robustness ($\Re$**):** *The global robustness $\Re$ of a system composed of a set of interconnected components is defined as the probability that an internal fault is detected by any one of the components.*

Without loss of generality, we can consider a component in isolation, and the interconnection rules are those classically defined in OO systems, e.g. in a UML context.

It has to be noted that an internal fault in a component plugged into a system can be detected either by the component itself or by one of its clients or children. Intuitively, the global robustness cannot be directly deduced by the knowledge of local components robustness. We argue that a relationship exists between local and global robustness but that additional information on the architecture is needed to obtain the global robustness. The proposed model extracts the main attributes from a UML model to compute the global robustness based on local robustness measures(the information we need could also be re-constructed through static analysis of programs written in Java, C++, Eiffel…). This consideration leads to the definition of the local robustness of a component plugged into a system.

Definition - Local Robustness ($RobInS_i$**):** *The local robustness $RobInS_i$ of a component C_i in a system S is defined as the probability that all fault is detected either by C_i when it is known that this fault would provoke a failure.*

Both isolated and local robustness are measurements local to a component, while the global robustness concerns the whole system. This explains the decomposition of axiomatization into local and global axioms

a) Axiomatization

The axioms formalize the essential expected properties of any measurement. They define what should be comparable and the generic characteristics that these measures must satisfy. Based on the definitions given in the previous paragraph, three sets of axioms are provided: global axioms, local axioms (for local and isolated robustness) and axioms linking global and local measures. They constitute the basis of the theoretical evaluation which was carried out.

Measures profiles:

Rob_i: Component $\rightarrow$ Real over [0..1]

$\Re$: System Architecture $\rightarrow$ Real over [0..1]

Since all robustness measurements are probabilities, they are bounded between 0 and 1, a value of 1 meaning perfect robustness (internal faults are always detected) and 0 indicating a non-robust system or component.

Local robustness axioms:

LRA1 - Component comparison. *All components of a system are comparable in terms of local and isolated robustness.*

LRA2 – Component with no contracts (or assertions). *Components which have no contracts (or assertions or other fault detection mechanisms) have an isolated robustness value of 0.*

The following axioms concern the intuitive behavior of the measures under some design operations: system concatenation, contracts addition and contracts improvement

- *Concatenation*: models any operation that allows the connection of two systems to produce a new one (for example using inheritance, client/provider dependencies).
- *Contract addition*: operation consisting of adding a contract to a system component (pre/post conditions, class invariants).
- *Contract improvement*: operation consisting of adding a new clause to an existing contract to check the consistency of a previoulsly non verified property of a component. A contract is thus improved iff it checks more properties of the component.

LRA3 - System concatenation. *The isolated robustness of a component included in a system S1 is unmodified by concatenation to a system S2 and its local robustness cannot decrease.*

LRA4 –Contract (assertion) addition. *In a system, the local and isolated robustness of a component cannot decrease by the addition of a contract to a component in the system.*

LRA5 – Contract improvement. *The improvement of a contract of a component C_i in a system must increase its isolated and local robustness. The other components local (and obviously isolated) robustness cannot decrease.*

Global robustness axioms:

GRA1 - System comparison. *Two systems are always comparable in terms of robustness.*

GRA2 - System concatenation. *The global robustness of a system obtained by concatenation of two systems S1 and S2 cannot be lower than the lowest robustness of S1 and S2.*

GRA3 - Observation point addition*. For any system, its global robustness cannot decrease by addition of an observation point.*

Inter-level axioms:

The knowledge of local and isolated robustness measures must be sufficient to deduce the system global robustness.

IRA1 - Inter-level relationship. *An injective function relates local robustness to global robustness.*

3.2. Assumptions and mathematical model

A component isolated from the system will have a basic robustness corresponding to the strength of its embedded contracts. A component plugged into a system has a robustness enhanced by the fact that its clients bring their contracts to help the fault detection. The notion of Test Dependency is thus introduced to determine the relationship between a component and its clients in a system.

Definition- Test dependency: *A component class C_i is test-dependent on C_j if it uses some objects from C_j. This dependency relation is noted: C_i R_{TD} C_j*

For example, on figure 3, component C is test-dependent on D, and Components A and B are test-dependent on C.

Definition - Det_i^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_i^j.

In next section, we give a way to estimate the robustness of a component and the probability Det_i^j.

Even though the test dependency relationship is transitive, we only consider faults that are detected by a component directly dependent on the faulty one. Inheritance is a special case of Test Dependency whose impact on robustness measurement is still unclear to us. For our experiments we have thus considered two hypothesis: the pessimistic one for which we consider that the Det_i^j probability is 0 for inheritance, and the optimistic one for which we consider that the Det_i^j probability is the same for inheritance as for the client relationship.

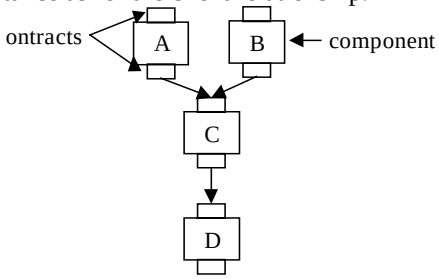

Fig. 3. Example for test-dependency

The robustness $RobInS_i$ (= 1- $WeakInS_i$) of the component C_i in the system S is the probability a fault in the component C_i is detected either by its own contracts or by the components it interacts with. To calculate this probability, we calculate $WeakInS_i$. The probability $WeakInS_i$ is the probability that a fault due to C_i is not detected

locally by C_i multiplied by the probability that the fault is not detected by the clients of C_i.

$$WeakInS_i = Weak_i \cdot \prod_k (1 - Det_k^i), k / C_k R_{TD} C_i$$

Finally, the global robustness $\Re$ of the system is thus equal to:

$$\Re = 1 - Weak = 1 - \sum_{i=1}^{n} Prob\_failure(i) \cdot WeakIntoS_i$$

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.

3.3. Experimental model parameterization

a) Mutation analysis for OO Domain

Mutation testing is a testing technique which was first designed to create effective test data, with an important fault revealing power [Offutt96, Voas92]. It was originally proposed in 1978 [DeMillo78], and consists in 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 set of test cases is *relatively adequate* if it distinguishes the original program from all its non-equivalent mutants. Otherwise, a *mutation score (MS)* is associated with the set of test cases set to measure its effectiveness in terms of the percentage non-equivalent mutants detected. 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.

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.

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.

For experiments, our choice of mutation operators includes selective relational and arithmetic operator replacement, variable perturbation, but also referencing faults (aliasing errors) for declared objects, these operators are detailed in [Baudry00]. The operators introduced for the object-oriented domain are the following:

- MCP (Methods Call Replacement): Replace methods by a call to another method with the same signature.
- RFI (Referencing Fault Insertion): Nullify 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, most common in object-oriented programming.

b) Estimating Contracts efficiency: a case study

To compute the contract quality of classes in a system, that is the isolated robustness of the class, we used two mutation analyses.

A first analysis uses behavioral differences between the initial class and a mutant class. During this first analysis, a test kills a mutant if the execution results of a test on the initial class and on a mutant class are different. This analysis is incremental: first we compute the mutation score of an initial test cases set, if this score is not satisfying, we write new test cases to improve the score. At the end of this analysis we have a good test cases set for each class in the system able to kill at least 90% of the class' mutants.

For the second mutation analysis, we consider all the mutants of a class alive and we try to kill them again, using only the class contracts. For this second analysis, we execute all the test cases we have written during the first analysis on all the mutants. We say that a test kills a mutant if the execution of the test on the mutant class raises an exception. The mutation score of a class' test cases set at the end of this analysis is the percentage of mutants the class' contracts are able to detect. For our experiments, we consider this score as the *isolated robustness* of the class. If this initial robustness value is not satisfying, we can improve the contracts until we reach a good robustness.

Once we have the isolated robustness of each class, we can measure Det_i^j. This value, for a component C_i, is measured by injecting faults in C_i's providers. Then, we execute C_i tests using C_i and its faulty providers. The percentage of killed mutants is the Det_i^j.

The parameters of the model of robustness are easily fixed using mutation analysis:

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.

$Prob\_failure(i) = 1/n$, n being the number of classes in the system.

Starting from a system in which each class has an associated test case set, the aims of the case study are the following:

1. To appraise the initial effectiveness of contracts and improve them using this approach,
2. To estimate the robustness of a component with embedded selftest in terms of detecting faults due to supplier classes.

We used the Pylon library (http://www.eiffel-forum.org/archive/arnaud/pylon.htm) as a case study. It is a small, portable, freely available Eiffel library for data structures and other basic features. The class diagram is composed of 50 classes and 134 relations. This library is complex enough to illustrate the approach and obtain interesting results. The way in which the various classes used in this package interact is presented in Annex. The mutation analysis tool used, called *mutant slayer* or μSlayer, is dedicated to the Eiffel language. This tool injects faults in a class under test (or a set of classes), executes tests 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 restart 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).

Concerning the improvement of contracts, initial contracts killed an average of 58.5% of mutants, after improvement the mutation score reached an average of 7.5%.

The isolated robustness of classes is significantly improved (the best improvement is from 25% to 100%). 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. At the end of the improvement process, the contractable component has a considerably greater capacity to detect faults (between 72% and 100% in the case of mutation faults for this study). As a result, this approach highlights methods for which the associated contracts are too weak.

To measure Det_i^j values, we generate the mutants for a class, and compute the mutation score for the clients' set of test cases on the mutants of methods the client uses. For example, if class P is a provider for class C, we generate mutants for class P, we select the mutants of methods used by C and we compute the mutation score for the set of test cases of C on the selected mutants. This mutation score corresponds to the percentage of mutants of P the contracts of C are able to detect and this is what we call Det_i^j.

For these measures, all the classes in the system have their improved contracts (average isolated robustness 87%), and the values are ranged from 50% to 84%.

3.4. Results

To illustrate the interest of a design-by-contract approach for robustness improvement, we applied it *a posteriori* to three real world case studies in the telecommunications and compiler software domains.

- A Telecommunications Switching System: Switched multimegabits data service (SMDS) is a connectionless, packet-switched data transport service running on top of connected networks such as the Broadband Integrated Service Digital Network (B-ISDN), which is based on the asynchronous transfer mode (ATM). A detailed description of an SMDS server design and implementation can be found in [Jéron99]. The class-diagram is composed of 37 classes, with a high connectivity degree (72 relations).
- The Pylon library, which has already been presented above .
- The InterViews library, composed of 146 classes and 420 relations.

For these three systems, we show the evolution of global robustness with the improvement of isolated components' robustness. To illustrate this evolution, we consider that the Det_i^j probability depends on the component's robustness as follows:

$$Det_i^j = K \quad Rob_i$$

Using the measures in tables 2 and 3, we fix the coefficient K: *K=0.8*. The robustness evolutions for the three systems are shown Figure 5. For these evolutions, we make the pessimistic assumption that inheritance dependencies have no impact on global robustness.

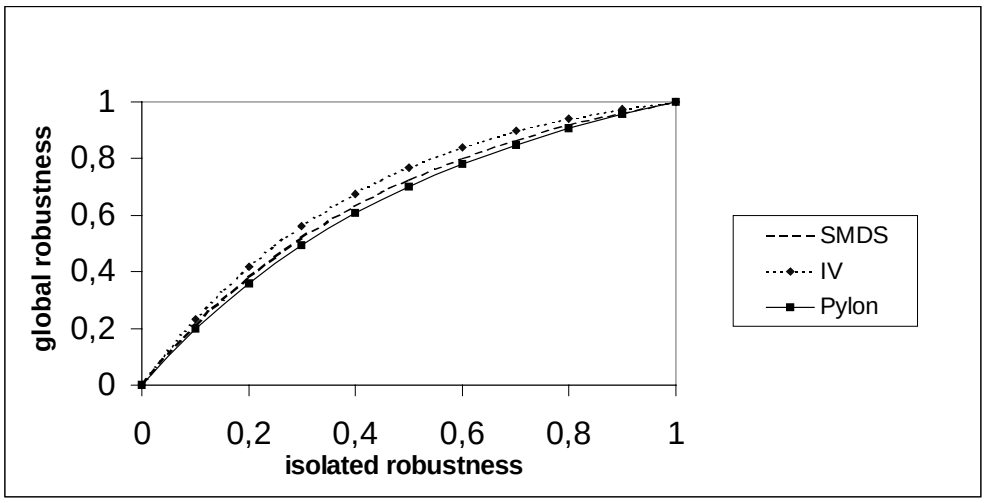

Fig. 5.Evolution of global robustness for three systems

With these results, we see that using no contracts implies that the system is not robust, but that adding simple contracts improves the global robustness rapidly. For example, in the InterViews system, if the components isolated robustness is 0.4, the global robustness is almost 0.7. Moreover, the three curves show that improving the isolated robustness from 0.8 to 1, which corresponds to the most costly improvements, is not interesting in terms of global robustness improvement: for InterViews the global robustness is already 0.94 when isolated components' robustness is 0.8.

The slight differences between the systems correspond to different dependency densities. Indeed, the local robustness of a component can be increased by its clients' contracts, and improving local robustness improves global robustness. Thus, the more relationships there are between components, the more local robustness can increase, and the bigger is the global robustness.

We have measured evolutions of the SMDS robustness considering different influences of inheritance dependencies for global robustness. We have considered three cases, the first case corresponds to the pessimistic assumption under which we ignore inheritance. In the second case, we have considered that inheritance is less important than other dependencies for global robustness, and took a coefficient K=0.2 for Det_i^j. Finally, we considered that inheritance is as important as other dependencies. The measures showed that the maximum difference between values is only 3%.

All we can say now is that the three cases bound the true robustness value, and that future work should help us understand more precisely the importance of inheritance for local and global robustness.

4. Measuring Diagnosability

A failure may be observed during the software development as well as the maintenance stage. Given the occurrence of a failure, diagnosis requires a set of additional symptoms in order to determine the faulty part of the system which causes the detected failure. *Diagnosis* is thus defined as the task of locating faulty parts of a system when a failure is detected. The notion and definition of what we call "diagnosability" are introduced in this section: the formal part of the measurement process is not presented. We just briefly comment the main results obtained using a diagnosability measurement and its impact on system quality.

To analyze the diagnosability attribute, one needs to understand the main methods used for locating faults in the software after they have been detected.

4.1. Diagnosis practices in the software domain

A first way of locating faults consists of performing some cross-checking between information resulting from test executions. Such systematic cross-checking of test results and executed paths may lead to semi-automated diagnosis strategies [Khalil98]. Along these lines, most diagnosis reported works are based on the program slicing techniques. These techniques focus on the software code at the unit and integration levels. Various slicing methods exist [Weiser84, Weiser82, Kamkar95, Korel97, Agrawal95] which basically consist in extracting

from the program a set of statements which can be executed independently (this corresponds to a *slice* of the program). The fault localization consists in executing the program slice by slice and in analyzing each slice result. The main limitation of this technique is its cost in terms of human effort. Indeed, each slice implies the determination of an oracle and, because the slices have no simple functional meaning, it often needs human intervention.

Another classical way for locating faulty statements consists of inserting assertions in the program for detecting some internal faulty state during execution. The systematic use of assertions before and after procedure calls may be very efficient for detecting and locating faults. Design by contract is a generalization of this principle. However, the effort for defining and inserting assertions may be important because it implies a good understanding of the internal meaning of the procedure and expected values of the data. Some works have focused on the way of inserting assertions when needed in the program, with testability criteria [Voas95].

Confronted with the problem of diagnosis, which remains a non-automated task, it would be useful to appraise the probable difficulty of locating faults in the software beforehand. Such predictor estimate is called *diagnosability (see* [Le Traon98] for data flow designs), and provides a way of improving the design quality.

To illustrate the importance of such measurement, we concentrate on the impact of executable contracts/assertions on OO system diagnosability. Measures are generic enough to be adapted to classical procedural programming.

4.2. Diagnosability: analysis of the notion

The localization effort is related to the size of the sets of suspected components or statements. The bigger the suspected sets are, the more difficult the diagnosis is. Moreover it is intuitively more difficult to distinguish a faulty statement among ten statements than to determine it among two statements. Applying the same reasoning, the diagnosis preciseness obtained by a test strategy is higher if the fault is located among two statements rather than among ten statements. These intuitive considerations can be expressed by three propositions:

- the diagnosability is composed of the localization effort and the diagnosis preciseness,
- the localization effort and the diagnosis preciseness are closely connected,
- the diagnosability depends on the capacity to isolate statements in the structure either by applying a test strategy covering this structure or by using “watchdogs”, e.g.; contracts and other assertions.

The underlying attribute expressing the localization of faulty statements among a set of suspected statements is called *indistinguishability*. To conclude, the *diagnosis effort* and *the preciseness* can both relate to the number of indistinguishable statements in which the faulty statement has to be localized.

4.3. Definitions

In this section, we detail the definition of diagnosability. The refinement of the intuitive diagnosability definition into a set of behavioral axioms is not presented here for conciseness reasons. The process used for producing a diagnosability measurement is similar to the one used for robustness. Since the intuitive aspects of diagnosability have been already discussed, diagnosability and its related attributes can be defined.

Definition (Informal)- Diagnosability: *Diagnosability expresses the localization effort as well as the precision allowed by a test strategy on a given system.*

The effort depends on the selected test strategy. However, for sake of simplicity, we do not consider multiple paths diagnosis strategies (the problem has been studied in detail in [Le Traon98]) since we focus on the impact of contracts on diagnosability. While for robustness our basic measurable attribute was a component and its contracts, our entry points to diagnosability measurement are the statements executed when a failure occurs or when a contract (or assertion) detects the fault. The notion of component is forgotten here, since we concentrate first on a particular software execution.

Assumptions/propositions:

- the software is assumed to be faulty: there exists an execution of the system that would provoke a failure if it were not detected by a contract.
- contracts are assumed to be correct
- a fault can be modeled as an invalid state in the global program state, which differs from the expected one after the execution of a statement. We call this statement the faulty statement, even if it is not necessarily the cause of the failure (that for example can be an omitted statement),
- the main diagnosis task consists of locating this divergence point,
- as a consequence, if an execution flow is faulty on multiple points, the diagnosis will point out the first divergence point (faults that compensate each other are considered as negligible for a global estimate).

Definition – Execution flow: *An execution flow is a partially ordered set of statements and contracts (or assertions) that is executed by a given program. Implicitly, we only consider flows that would provoke a failure.*

The statements are partially ordered because – and particularly in an OO system- some of the execution may be distributed on several threads. To simplify the mathematical model, we only consider a non distributed flow. The exhaustive mathematical modeling of such flows is not presented because it does not significantly modify the results of the measurements while it makes the model much more complex. In our case, an execution flow is equivalent to a dynamic slice of the system.

Definition - Indistinguishable statements: *Two statements are indistinguishable from each other if they are bounded by consecutive contracts in an execution flow (or the entrance and output of the flow).*

Definition - Indistinguishability set: *An indistinguishability set corresponds to a set of indistinguishable statements.*

Definition - Size of an indistinguishability set: *The size of an indistinguishability set is equal to the cardinality of this set. The smallest possible size is 1; the case when the suspected set is made up of only one statement.*

For local measurements (attached respectively to a statement and to an execution flow), the diagnosability is more directly expressed in terms of diagnosis effort (in that case, a diagnosability improvement corresponds to a reduced diagnosis effort) while the global diagnosability measure (attached to a system) is related to the precision of diagnosis.

Definition - Local diagnosis effort (δ): *The local diagnosability δ of a statement Stat in an execution flow F is the probable effort needed for determining that Stat is faulty in F when the number of statements, the number and distribution of contracts and their efficiency is known.*

Definition – Local diagnosis effort for a flow (δeff): *The global diagnosis effort for a flow F is the probable effort needed for pointing out the faulty statement, knowing that a fault is detected in F, and knowing the number of statements, the number of contracts and their efficiency.*

Definition - Global diagnosability of a system (Δ): *The global diagnosability of a system S is the probable degree of preciseness obtained depending on the density and quality of the embedded contracts.*

From these measurement definitions, derived from the informal ones, one can deduce the measure expected profiles [Le Traon00]. A diagnosis effort is a ratio extensive measure (operations such as addition are possible since it is a counting measure). In our study the measurable attribute associated to the diagnosis effort is the size of the indistinguishability set in which the faulty statement must be located (using scrutation with program slicing for example). Local diagnosability and diagnosis effort should thus express the probable size of an indistinguishability set. These measurements thus take their domain value into $[1..+\infty]$. Note that a good diagnosability corresponds to a low diagnosis effort. For the global diagnosability Δ, a difficulty comes from the fact that there is no general relationship between the size of the execution flow and the size or architecture of the system. What we want to measure is the degree of diagnosis precision obtained with a certain proportion/quality of contracts in the system, compared to the same system with no contracts (as an absolute reference value). Δ is also a ratio measurement but it is intensive, in the sense that values can not be easily combined. For Δ, a good diagnosability corresponds to a 1 value and the worse one to 0 (when no contracts or assertions allow the reduction of the diagnosis scope).

Measures profiles:

δ: Statement $\times$ Flow $\rightarrow$ Real over $[1..+\infty]$

δeff: Statement $\rightarrow$ Real over $[1..+\infty]$

Δ: System $\rightarrow$ Real over $[0..1]$

The detail of the axiomatization and measure definition are not given in this paper, we just comment the diagnosability results in relation with the density of contracts in the design and their quality (in terms of probability to detect the faulty state of the system).

4.4. Results and conclusions

Results show that the introduction of contracts quickly enhances the global diagnosability of the system. Besides the Δ values are stable between [0.17, 1] range of density of contracts. So, the addition of many contracts (high contract density) does not significantly improve a system global diagnosability :

$\Delta \approx 0.6$ with contracts efficiency equal to 0.2

and a contract density $\in [0.17, 1]$

$\Delta \approx 0.9$ with contracts efficiency equal to 0.4

and a contract density $\in [0.17, 1]$

Finally, the quality of the contracts is more important than their number, since it is the only way to make the upper bound for diagnosability increase.

The conclusions we can deduce from this measurement are the following:

- a 0.2 contract/assertion density is enough to reach the upper bound of diagnosability for a given contract average efficiency. It has to be noted that in good OO designs the size of methods is often small, and includes a small number of statements. A 0.2 contract density corresponds to a good and possible density for an OO system. In most cases, the use of assertions in the body of methods is thus useless. This result could not be easily predicted without a mathematical model.
- Quality is better than quantity. For the same contracts density, the diagnosability is highly sensitive to the quality of contracts. It is better to put the effort on a good design with high encapsulation and well defined interfaces (supporting clearer properties derivable into contracts) than to put the effort on defensive assertions, that often are unclear and dependent on the code.

Design by contract is thus a very efficient way of improving the diagnosability and robustnesss of a system and its general quality.

5. Conclusion

The work presented here focused on the definition of measures for two quality factors in a particular problem domain: the use of a design by contract approach for designing and implementing OO systems. The main steps in the construction of measures elaboration have been given, from informal definitions of the factors to be measured (robustness and diagnosability) to the mathematical model of the measures. To fix the parameters, experimental studies have been conducted, essentially based on applying mutation analysis in the OO context. Several measures have been presented that estimate the contribution of contract quality and density to the overall quality of a system in terms of robustness and diagnosability. Finally, our results confirm that the quality of contracts is more important than their quantity.

Acknowledgements

We thank Clémentine Nebut and Franck Allion for computing the data about the Pylon library.

References

[Agrawal95] H. Agrawal, J. Horgan, S. London, and W. Wong, “Fault Localization using Execution Slices and Dataflow Tests,” presented at 6th International Symposium on Software Reliability Engineering, Toulouse (France), 1995.

[Baudry00] B. Baudry, Y. Le Traon, H. Vu Le, “Testing-for-Trust: the Genetic Selection Model applied to Component Qualification”, In proceedings of TOOLS’2000 (Technology of Object Oriented Languages and Systems), pp. 108-119, June 2000.

[Briand96] L. C. Briand, S. Morasca, and V. R. Basili, “Property-Based Software Engineering Measurement,” *IEEE Transactions on Software Engineering*, vol. 22, pp. 68-86, 1996.

[DeMillo78] 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.

[Fenton86] N. E. Fenton and R. W. Whitty, “Axiomatic approach to Software Metrication through Program Decomposition”, in *The Computer Journal*, vol. 29, 1986, pp. 330-339.

[Jéron99] 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) , pp. 260-269, Boca raton (Florida), November 1999.

[Jézéquel97] J-M. Jézéquel and B. Meyer, “Design by Contract: The lessons of Ariane”, *Computer*, vol. 30, No. 1, pp. 129-130, January 1997.

[Kamkar95] M. Kamkar, “An Overview and Comparative Classification of Program Slicing Techniques,” *Systems Software*, vol. 31, pp. 197-214, 1995.

[Khalil98] Khalil, M., Le Traon, Y. and Robach, C., “Automated Strategies for Software Diagnosis”, Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE’98), Paderborn (Germany), November 1998.

[Kitchenham95] B. Kitchenham, S. L. Pfleeger, N. Fenton, “Towards a Framework for Software Measurement Validation”, *IEEE Transactions on Software Engineering*, vol.21, No. 12, pp. 929-943, December 1995.

[Korel97] B. Korel, “Computation Of Dynamic Program Slices For Unstructured Programs”, *IEEE Transactions on Software Engineering*, vol. 23, pp. 17-34, 1997.

[Le Traon98]Y. Le Traon, F. Ouabdesselam and C. Robach, “Software Diagnosability”, Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE’98), Paderborn (Germany), November 1998.

[Le Traon00]Y. Le Traon, F. Ouabdesselam and C. Robach, “Analyzing Testability on Data Flow Designs”, to be presented. to the International Symposium on Software Reliability Engineering 2000 (ISSRE’00), San José (CA), October 2000.

[Liskov86] B. Liskov and J. Guttag, “Abstraction and Specification in Program Development”, MIT Press/McGraw-Hill, 1986.

[Meyer92] B. Meyer. “Applying “Design by contract””. *IEEE Computer*, Vol. 25, No. 10, pp. 40--52, October 1992.

[Meyer00] B. Meyer, "Toward More expressive contracts", *Journal of OO Programming (JOOP)*, July-August 2000, pp. 39-43.

[Offutt96] 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, February 1996.

[Shepperd93] M. Shepperd and D. Ince, “Derivation and Validation of Software Metrics”. Oxford, 1993.

[Voas92] J. Voas et K. Miller, “The Revealing Power of a Test Case”, *Software Testing, Verification and Reliability*, vol. 2, pp. 25-42, 1992.

[Voas95] J. M. Voas, “Software Testability Measurement for Assertion Placement and Fault Localization,” presented at 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95), Saint-Malo (France), 1995.

[Weiser82] M. Weiser, “Programmers Use Slices When Debugging,” *Communication of ACM,*, vol. 25, pp. 446-452, 1982.

[Weiser84] M. Weiser, “Program Slicing,” *IEEE Transactions on Software Engineering*, vol. 10, pp. 352-357, 1984.

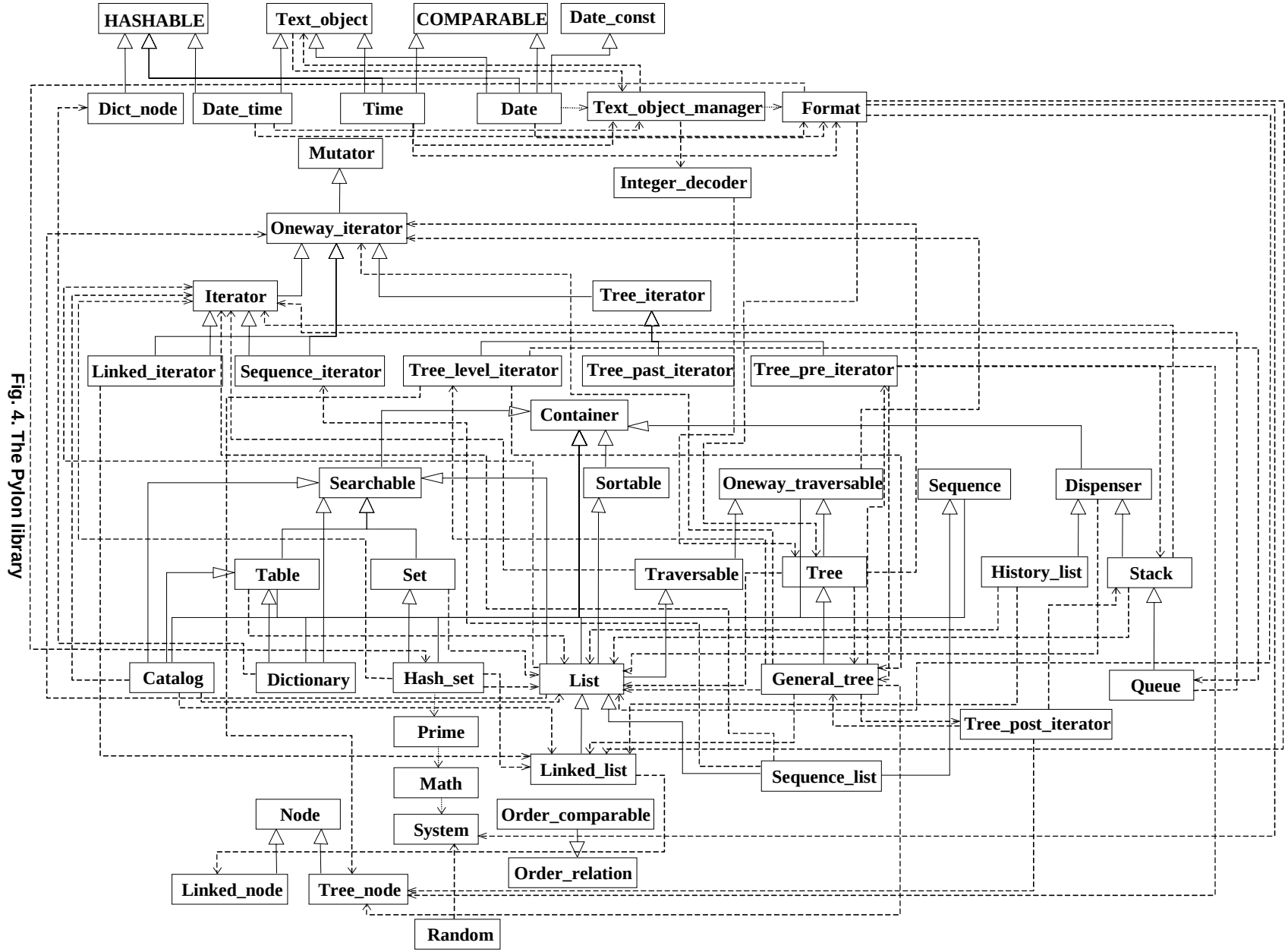

Fig. 4. The Pylon library