HAL
open science

Design by Contract to improve Software Vigilance

Yves Le Traon, Benoit Baudry, Jean-Marc Jézéquel

To cite this version:

Yves Le Traon, Benoit Baudry, Jean-Marc Jézéquel. Design by Contract to improve Software Vigilance. IEEE Transactions on Software Engineering, 2006, 32 (8), pp.571–586. inria-00542784

HAL Id: inria-00542784

https://inria.hal.science/inria-00542784v1

Submitted on 3 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design by Contract to Improve Software Vigilance

Yves Le Traon, Benoit Baudry, *Member*, *IEEE*, and Jean-Marc Jézéquel, *Member*, *IEEE*

Abstract—Design by Contract is a lightweight technique for embedding elements of formal specification (such as invariants, pre and postconditions) into an object-oriented design. When contracts are made executable, they can play the role of embedded, online oracles. Executable contracts allow components to be responsive to erroneous states and, thus, may help in detecting and locating faults. In this paper, we define Vigilance as the degree to which a program is able to detect an erroneous state at runtime. Diagnosability represents the effort needed to locate a fault once it has been detected. In order to estimate the benefit of using Design by Contract, we formalize both notions of Vigilance and Diagnosability as software quality measures. The main steps of measure elaboration are given, from informal definitions of the factors to be measured to the mathematical model of the measures. As is the standard in this domain, the parameters are then fixed through actual measures, based on a mutation analysis in our case. Several measures are presented that reveal and estimate the contribution of contracts to the overall quality of a system in terms of vigilance and diagnosability.

Index Terms—Object-oriented design methods, programming by contract, diagnostics, metrics.

1 Introduction

Several works advocate the use of assertions to improve software quality [1], [2], [3], [4], [5], but very few are actually interested in measuring this improvement. This paper aims at bridging the gap between intuitive understanding of what assertions improve in software and a quantitative, accurate estimate of these improvements. We focus on Design by Contract as a method to place assertions at specific locations in an object-oriented (OO) system. Design by Contract was introduced by Meyer [6] as a lightweight technique for embedding elements of formal specification into an OO design through pre and postconditions of methods and class invariants. In practice, Design by Contract is acknowledged to be a reasonable trade-off between the full extent of formal specifications (as a complete description of the behavior of the system) and the effort acceptable to developers [7]. More than making the classes' interfaces explicit, the observed benefits of this approach are to help in detecting and locating faults since executable contracts can play the role of embedded and online oracles [8].

In the usual sense, *vigilance* can be defined as the quality or state of being wakeful and alert. This notion can be extended to the software domain as the ability of a system to dynamically detect an erroneous internal state. Based on this informal definition, we can say that contracts contribute to making a system more "vigilant" about the correctness of its internal execution state. Vigilance is a useful building block for setting a robustness mechanism, a mechanism for recovering from an error before a failure occurs can be attached to each contract violation. A property related to vigilance is *diagnosability*, which represents the effort needed to locate a fault in a system knowing that this fault has caused a failure.

This paper proposes a model that captures the impact of design by contract on vigilance[1] and diagnosability. It extends the preliminary work of [9] by providing a complete mathematical model and a more detailed analysis of the impact of design-by-contract and by illustrating the whole approach with precise examples. We define vigilance and diagnosability in the context of a contract-based design approach since software with embedded executable contracts can detect internal anomalies during execution (the system is thus more vigilant) and help in pinpointing the fault location (the contract that detects the erroneous state is expected to be close to the fault).

The elaboration of the models for vigilance and diagnosability measures follows a precisely defined framework. An important part of this elaboration consists of expressing the measure and isolating each parameter on which it depends. This process is given here only for vigilance and is published in [10] for diagnosability. Once the parameters are isolated, a mathematical definition of the measure can be established. Several experiments have been conducted to measure values for these parameters on a case study. The case study is written in Eiffel, which is an object-oriented language with native support for the implementation of contracts. The most important criterion is the ability of contracts to detect errors. This is called the efficiency of contracts and it is estimated by adapting mutation analysis [11]. The idea is to create faulty versions of the system by injecting errors. When a contract is violated during the execution of a faulty system, it means that the contract has detected an error. With values for every parameter, it is possible to estimate the impact of contracts on vigilance and diagnosability. The interest of these estimates is to allow a prediction of the effort that the programmer must devote to

- *Y. Le Traon is with France Télécom R&D, 2, Avenue Pierre Marzin, 22 307 Lannion Cedex, France. E-mail: yves.letraon@rd.francetelecom.com.*
- *B. Baudry and J.-M. Jézéquel are with IRISA, Université de Rennes1, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France. E-mail: {bbaudry, Jean-Marc.Jezequel}@irisa.fr.*

Manuscript received 2 Feb. 2005; revised 24 Nov. 2005; accepted 20 June 2006; published online 7 Sept. 2006.
Recommended for acceptance by B.G. Ryder.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0023-0205.

1. Vigilance is called robustness in [9].

writing contracts in order to reach a certain level of vigilance and diagnosability.

Section 2 opens on a presentation of the design by contract approach and an intuitive analysis of some expected benefits of the approach: vigilance and diagnosability improvement. Section 3 concentrates on vigilance definition, definition of the expected measure properties, and calibration of the model parameters on several case studies. Section 4 presents experiments to tune the parameters for the vigilance measure. Section 5 is devoted to diagnosability analysis, along the same lines as for vigilance. Last, Section 6 discusses related work.

2 The Problem Domain: Design by Contract and Measures Elaboration

Since fault, error, and failure are notions used throughout the whole paper, we take the definitions given in [12]: A *fault* designates the cause of an error, an *error* is the part of the system which is liable to lead to a failure, and a *failure* is the deviation from the delivered service compliance with the specification. This section introduces the design by contract methodology and the underlying theory. We also summarize elements of the Object Constraint Language (OCL) that is used for illustration in the paper. Finally, a generic framework for measures elaboration is proposed.

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 unambiguously spelling out these contracts is a worthwhile design approach [13], which Meyer named the *Design by Contract* approach to software construction [14]. Building on the idea of defensive programming [15], where it is recommended to protect every software module by as many checks as possible, the design by contract approach provides a methodological guideline for building vigilant, yet modular and simple systems.

The *design by contract* approach prompts developers to precisely specify 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 postconditions 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 hold. In return, the contractor promises that, when the routine returns, the work specified in the postcondition will be done and the class invariant still holds.

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

Since the work presented in this paper studies the impact of adding contracts to a program, we give a precise definition of contract and also make clear two important notions: contract correctness and completeness.

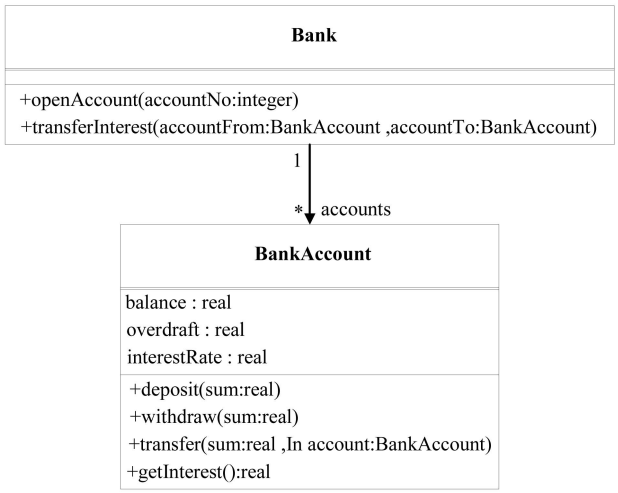


Fig. 1. Simple class diagram of a Bank.

Contract. *A method contract is a set of assertions that are evaluated before and after the execution of one method. There is one contract for each method in the program that is composed of a pre and a postcondition and of the invariant of the class.*

Correct contract. *A contract is correct if it is never violated by a correct call of a correct implementation. A precondition in a correct contract is never violated by a legal input for the method. A postcondition in a correct contract is never violated if there is no fault in the method.*

Complete contract. *A contract is complete for a method if the invariant and the precondition completely define the domain of the legal inputs for the method and if the postcondition and the invariant always detect an erroneous state after the method's execution.*

Concerning these definitions, several important points have to be noticed. First, an empty contract (no pre or postcondition and no invariant) is correct. Second, as specified in the introduction, contracts offer a way to embed some elements of formal specification, not necessarily all; completeness is not necessarily the objective when writing contracts for a class. When writing contracts, it is important to embed the most important properties but also to write assertions that are not too complex: Writing them has to be simpler than writing the method. The work presented in this paper investigates the benefits of putting more or less effort into completing contracts (under the assumption that the contracts are correct). We study the benefits of the contracts' improvement on two factors: vigilance and diagnosability. A contract is said to be efficient if it has a high probability of detecting an erroneous state. So, the more complete a contract is, the more efficient it is.

2.2 The UML and the OCL

In this paper, the Unified Modeling Language (UML) is used to specify the system architecture and contracts are expressed using the Object Constraint Language (OCL) [16]. The OCL [16] is a formal language to express constraints on UML diagrams. It can be used to describe invariant on classes and pre and postconditions on methods. The different types of constraints as well as some OCL constructs are illustrated here, with a small example displayed in Fig. 1. *Invariants* can be expressed for both the BANK and BANKACCOUNT classes. The type of the

instance to which an invariant applies is written with the *context* keyword. A constraint labeled *inv* is an invariant. For example, an invariant constraint on the class BANK-ACCOUNT specifying that the balance must be greater than or equal to the overdraft for any instance of BANK-ACCOUNT is written:

```
context BankAccount inv :
  self.balance >= self.overdraft.
```

OCL enables the navigation of associations starting from a specific object. The role name is used to designate the set of objects on the other side of the association. For example, an invariant constraining any BANK instance to have at least one account is designated in the following way:

```
context Bank inv :
  self.accounts-> size > 0.
```

For pre and postconditions, the constraints are labeled with the keywords *pre* and *post*. In postconditions, OCL also enables access of the state before the operation, using the *@pre* operator. A precondition for the deposit method in BANK-ACCOUNT is that the sum passed as a parameter is positive and a postcondition states that the balance at the end of this method is equal to the balance before the method call plus the sum. These constraints are expressed as follows:

```
context BankAccount : deposit(sum : real) : void
  pre sum > 0
  post self.balance = self.balance@pre + sum.
```

2.3 Contracts for Vigilance and Diagnosability

In this paper, we focus on measuring the benefit of a design by contract approach for vigilance and diagnosability. When contracts are executable, their main impacts on the final product quality are twofold:

- Since contracts raise exceptions when violated, a faulty program state can be automatically detected during execution (due to a fault or bug). A specific mechanism can then be implemented to avoid the failure that would have certainly occurred. The intuition is that contracts can help improve software robustness by enhancing the vigilance of the system to erroneous states. The question is: How much do contracts help software vigilance, depending on their efficiency and number?
- When a contract is violated, it points out the part of the code where a faulty program state has been detected. With no contract embedded in the software, the failure might be detected later, maybe at the main outputs of the system. Since the scope of diagnosis (the number of statements in which the fault must be located) can be reduced when contracts are used to catch faults, contracts help in the diagnosis task. The question is: How much do contracts reduce the diagnosis effort depending on their efficiency and number?

Fig. 2 illustrates the difference between software with no contract (top of the figure) and the same designed by contract. Considering an execution thread and an infection point, in the first case, the error is propagated along the execution thread and produces a failure at the output of the program. In the second case, contracts are checked that can detect the error and raise an exception, thus preventing the propagation of the fault. Our intuition is that software designed by contract might allow an early detection of fault (during its propagation to the outputs) and help locate the faulty part of the software by reducing the "diagnosis scope" in which the fault must be located. When contracts are violated and a erroneous state detected, a recovery mechanism can be used to deal with this error and put the software in a safe state.

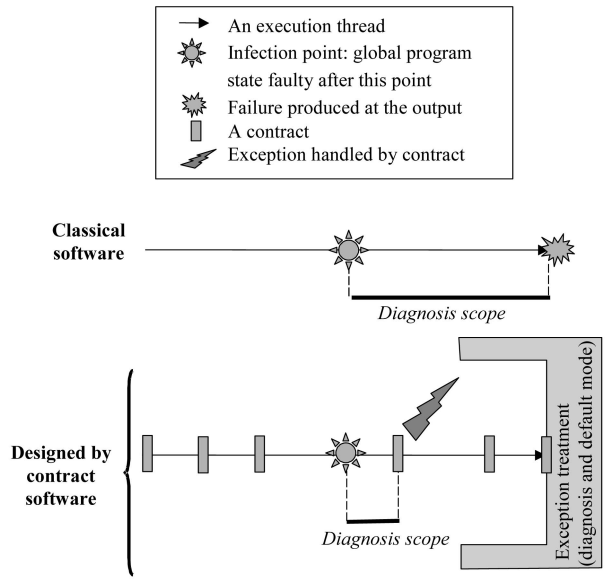


Fig. 2. Contracts for early detection of a fault.

2.4 Measure Elaboration

The literature insists on the difficulty of elaborating valid measures [17], [18], [19], [20]. In this paper, since the factors to be measured (namely, vigilance and diagnosability) first appear as quite abstract and unclear, we choose to make a property-based definition of the measures [20]. The usual steps of the measure elaboration are:

1. Identifying the factor to be measured.
2. Identifying the attributes upon which the factor depends.
3. Expressing intuitive properties.
4. Defining a formal model of the measure.
5. Checking the properties in the formal model.

The factor to be measured (1) is first informally defined and significant and measurable attributes (2) are identified (with intuitive and hopefully convincing arguments and assumptions). These attributes can be measured on a program. The number of contracts in the software and the mean number of statements between two contracts are examples of attributes. Then, the intuitive properties (3) of the factor's behavior must be expressed: These are called axioms in [20]. These properties express how the factor behaves when the software evolves (for example, the addition of a contract, system concatenation).

Based on the measurable attributes, a formal model of the measure can then be proposed (4): It is richer than the expressed properties (in the other case, the formal model is of no interest). The theoretical evaluation (5) is carried out so as to check the consistency of the proposed measure with the expected intuitive properties. This theoretical evaluation precedes empirical evaluation since it is less time-consuming

```
class BankAccount{
   float balance;
   public void deposit (float sum){
      balance=balance-sum;
   }...
}

context BankAccount::deposit
   pre self.amount>0
   post self.balance=self.balance@pre+amount
```

Fig. 3. Example for a fault, an error, and a failure.

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.

3 Measuring Vigilance Improvement

This section presents a model of the relationship between a component's vigilance and its contracts. A measure of the contracts' efficiency is proposed and the improvement in vigilance brought by contracts is explained.

3.1 Definitions

In the context of this study, a *fault* is a statement or a set of statements (or even omission of statements) that can lead to an error. An *error* is the state of a particular object (i.e., the set of the values of its attributes) in the system that can lead to a *failure*. For example, let us look at a faulty version of the BANKACCOUNT class (Fig. 3). In the `deposit` method, the statement `balance` = `balance` − `sum` should be `balance` = `balance` + `sum`. This statement is a fault: When it is executed, the `balance` attribute of the BANKACCOUNT class will be assigned a wrong value. The state of the object after execution of this method is said to be an error. This error will actually cause a failure since the value of balance is not consistent with the expected value expressed in the postcondition (`self.balance` = `self.balance@pre` + `sum`).

The term *component* is also very much used in the following definitions. In this work, it designates either a class in an object-oriented system or a set of classes grouped in a package with a well-defined interface (in terms of required and provided services). Now, let us give a first informal definition of vigilance.

Vigilance (informal definition). *Vigilance expresses the probability that the system contracts dynamically detect erroneous states that would have otherwise provoked a failure.*

Isolated Vigilance ($\mathrm{Vig_i}$). *The isolated vigilance* $\mathrm{Vig_i}$ *of a component* $\mathrm{C_i}$ *in a system S is defined as the probability that an internal error in* $\mathrm{C_i}$ *is detected by* $\mathrm{C_i}$. *Conversely, the "weakness,"* $\mathrm{Weak_i}$, *of the component is equal to the probability that the error is not detected.*

The notion of *internal error* is introduced in this definition. An internal error in $\mathrm{C_i}$ is an erroneous state of the component $\mathrm{C_i}$. The detection mechanisms we focus on are executable contracts. So, an error is said to be detected if a contract is violated because of this error. For example, the fault in Fig. 3 causes an internal error in BANKACCOUNT that is detected by the postcondition for the `deposit` method. Some components cannot directly be executed (e.g., abstract or generic classes). Nevertheless, they may still be equipped with their own contracts that can detect their internal errors. For example, there can be concrete methods in abstract classes and, when a class that inherits from the abstract class is executed, the contracts of the abstract class can detect errors in the concrete parts of the class.

Global Vigilance (V). *The global vigilance V of a system composed of a set of interconnected components is defined as the probability that an internal error is detected by any of the components. Conversely, the "weakness," Weak, of a system is equal to the probability that the error is not detected* $(V(S) = 1 - Weak(S))$.

It has to be noted that an error in a component plugged into a system can be detected either by the component itself or by one of its client or subclasses. Intuitively, the global vigilance cannot be directly deduced by the knowledge of local components vigilance. We argue that a relationship exists between local and global vigilance, but that additional information on the architecture is needed to evaluate the global vigilance. The proposed model extracts the main attributes from a UML class diagram to compute the global vigilance based on local ones. This consideration leads to the definition of the local vigilance of a component plugged into a system.

Local Vigilance (LocVig(C, S)). *The local vigilance, LocVig(C, S) of a component C in a system S is defined as the probability that an internal error in C is detected either by itself or by any other component of the system.*

We distinguish between local and isolated vigilance because an error in $\mathrm{C_i}$ might not be detected by $\mathrm{C_i}$'s contracts. This error might then propagate through the system and might be detected by contracts in another component. The vigilance of a component plugged into a system is thus different from the vigilance of the same component outside any particular context. Both isolated and local vigilance measure the likelihood of detecting an error in a single component, while the global vigilance concerns the whole system.

3.2 Expected Properties

Expected properties define what should be comparable and generic characteristics that these measures must satisfy. Based on the definitions given in the previous paragraph, two sets of properties are provided: global and local properties (in that case for local and isolated vigilance). They set up the basis for the theoretical evaluation.

Measures' profiles:

- Vig_i: Component → Real over [0..1] : the isolated vigilance of a component.
- $Weak_i$: Component → Real over [0..1] : the isolated weakness of a component. $Vig_i = 1 - Weak_i$.
- $LocVig(C, S)$: Component × System → Real over[0..1]: the local vigilance of a component in a system S.
- $LocWeak(C, S)$: Component × System → Real over [0..1] : the opposite of the local vigilance of C in S: the local weakness.
- V: System Architecture → Real over [0..1] : the global vigilance of a system.

Since all vigilance measures are probabilities, they are bounded between 0 and 1, a 1 meaning perfect vigilance (internal errors are always detected) and 0 a nonvigilant system or component.

3.2.1 Local Vigilance Properties

LVP1—Component comparison. *All components of a system are comparable in terms of local and isolated vigilance.*

LVP2—Component with no contracts (or assertions). *Components that have no contracts (or assertions or other fault detection mechanisms) have an isolated vigilance value of 0.*

The following properties concern the intuitive behavior of the measures when applying usual design operations: system concatenation, contracts addition, and contracts improvement. These operations correspond to the following actions:

- 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/postconditions, invariants).
- Contract improvement: Operation consisting of adding a new clause to an existing contract to check the consistency of a property that is not yet verified. A contract is thus improved iff it checks more properties of the component. Let us take the `withdraw` method in the BANKACCOUNT class (Fig. 1). A precondition for this method can be:

```
context BankAccount :: withdraw(sum : float) :
    void
pre sum > 0
```

Improving this precondition consists of adding a new clause, for example:

```
context BankAccount :: withdraw(sum : float) :
    void
pre sum > 0 and sum ≤ balance − overdraft
```

LVP3 System concatenation. *The isolated vigilance of a component included in a system S_1 is unmodified by concatenation to a system S_2 and its local vigilance cannot decrease.*

LVP4 Contract (assertion) addition. *In a system, the local and isolated vigilance of a component cannot decrease by addition of a contract in the system.*

LVP5 Contract improvement. *The improvement of a contract of a component C_i in a system must increase its isolated and local vigilance. The other components local (and obviously isolated) vigilance cannot decrease.*

3.2.2 Global Vigilance Properties

GVP1—System comparison. *Two systems are always comparable in terms of vigilance.*

GVP2—System concatenation. *The global vigilance of a system obtained by concatenation of two systems S_1 and S_2 cannot be lower than the lowest vigilance of S_1 and S_2.*

GVP3—Contract addition. *For any system, its global vigilance cannot decrease by addition of a contract.*

3.3 Assumptions and Mathematical Model

A component isolated from the system will have an isolated vigilance corresponding to the efficiency of its embedded contracts. A component C_i plugged into a system has a vigilance enhanced by the fact that its clients (other components that use C_i) bring their contracts to help the fault detection. The notion of *dependency* is thus introduced to determine the relationship between a component and its clients in a system.

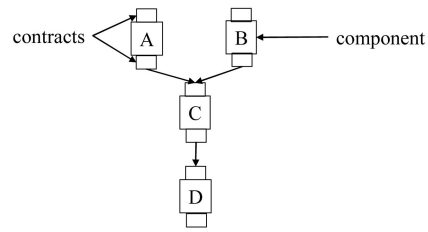


Fig. 4. Example for dependency.

Dependency. *A component C_i is dependent on C_j if it uses services from C_j. This dependency relationship is noted: $C_i\ R_D\ C_j$. For example, in Fig. 4, component C is dependent on D ($CR_D\ D$), and components A and B are dependent on C ($AR_D\ C$ and $BR_D\ C$).*

$\mathbf{Det(C_i, C_j)}$. *If $C_i\ R_D\ C_j$, then the probability that C_i's contracts detect an error in C_j is noted $Det(C_i, C_j)$.*

We notice that, according to this definition, Det(C, C) is the isolated vigilance of C. In the next section, we give a way to estimate the vigilance of a component and probability $\mathrm{Det(C_i, C_j)}$.

The dependency relationship is not necessarily transitive. In Fig. 4, A depends on C and C depends on D, but it is not possible to decide whether A depends on D without looking at the code. If A uses services of C which do not call any service of D, then A does not depend on D. Since we focus on the model to evaluate the local vigilance, we can consider only errors that are detected by a component *directly dependent* on the faulty one. The results obtained from a model are thus an estimate of the actual local vigilance that would be obtained by taking into account all dependencies. The local vigilance $\mathrm{LocVig(C_i, S)}$ $(= 1 - \mathrm{LocWeak(C_i, S)})$ of the component $\mathrm{C_i}$ in system S is the probability that an error in the component $\mathrm{C_i}$ is detected either by contracts of $\mathrm{C_i}$ or by the contracts of the components that use $\mathrm{C_i}$. To get this probability, we evaluate $\mathrm{LocWeak(C_i, S)}$. $\mathrm{LocWeak(C_i, S)}$ is the probability that an error in $\mathrm{C_i}$ is not detected locally by $\mathrm{C_i}$ multiplied by the probability that the error is not detected by the clients of $\mathrm{C_i}$. This last probability is the product of $1 - \mathrm{Det(C_k, C_i)}$ for all k such that $\mathrm{C_k\ R_D\ C_i}$.

$$\mathrm{LocWeak(C_i, S)} = \mathrm{Weak_i} \cdot \prod_{\mathrm{k}} (1 - \mathrm{Det(C_k, C_i)}), \mathrm{k/C_k R_D C_i}.$$

Finally, the global vigilance V of the system is thus equal to:

$$\begin{aligned} \mathrm{V(S)} &= 1 - \mathrm{Weak(S)} \\ &= 1 - \sum_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{Prob_err(C_i, S)} \cdot \mathrm{LocWeak(C_i, S)}, \end{aligned}$$

where $Prob_error(C_i, S)$ is the probability the failure in S comes from the component $\mathrm{C_i}$. This probability can be approximated by the component's complexity.

3.4 Properties Theoretical Evaluation

The theoretical evaluation is used to detect that no pathological structures exist for which the model produces inappropriate behavior. It aims at ensuring that the mathematical model fulfills the measure specification captured in the form of properties. In this section, we illustrate this step on property GVP3. The demonstration aims at showing that the formal model ensures that the addition of a contract in a system S does not decrease its global vigilance. Let us call S' the system after adding a contract, the proof given below aims at proving that $V(S') \geq V(S)$.

The addition of a contract in a component, for example, C_1 in Fig. 5, changes its isolated vigilance and the local vigilance of its q server components, i.e., $\mathrm{LocVig}(C_k) \mid C_1\ R_D\ C_k$. In Fig. 5, the addition of a contract in C_1 changes the isolated vigilance of C_1 as well as $\mathrm{LocVig}(C_2)$, $\mathrm{LocVig}(C_3)$, and $\mathrm{LocVig}(C_q)$.

Let us first consider the addition of a contract in C_1 and C_1', the component after the contract addition. The isolated vigilance of C_1' is greater than or equal to the isolated vigilance of C_1:

$$\mathrm{Det}(C_1', C_1') \geq \mathrm{Det}(C_1, C_1). \quad (1)$$

The probability that C_1' detects an error in one of its servers is greater than or equal to the probability that C_1 detects the error: $\forall\ k \mid C_1\ R_D\ C_k$, $\mathrm{Det}(C_1', C_k) \geq \mathrm{Det}(C_1, C_k)$, so:

$$\prod_k (1 - \mathrm{Det}(C_1', C_k)) \leq \prod_k (1 - \mathrm{Det}(C_1, C_k)),$$

i.e.,

$$\forall\ k \mid C_1\ R_D\ C_k, \mathrm{LocWeak}(C_k, S') \leq \mathrm{LocWeak}(C_k, S). \quad (2)$$

The servers of C_1 are $[C_2, C_3 \ldots C_q]$ and $V(S')$ is the following:

$$V(S') = 1 - \sum_{i=(q+1)}^{|S'|} (\mathrm{Prob_Err}(C_i', S') \times \mathrm{LocWeak}(C_i', S')) + \sum_{i=1}^{q} (\mathrm{Prob_Err}(C_i', S') \times \mathrm{LocWeak}(C_i', S')).$$

For $i > q$, $\mathrm{LocWeak}(C_i', S') = \mathrm{LocWeak}(C_i, S)$ because the addition of a contract has no influence on the vigilance of components with an id $> q$ since C_1 does not depend on them. Moreover, since the number of components in S' does not change, $|S'| = |S|$ and $\mathrm{Prob_Err}(C_i', S') = \mathrm{Prob_Err}(C_i, S)$. The vigilance $V(S')$ can be rewritten:

$$V(S') = 1 - \sum_{i=(q+1)}^{|S|} (\mathrm{Prob_Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S)) + \sum_{i=1}^{q} (\mathrm{Prob_Err}(C_i', S') \times \mathrm{LocWeak}(C_i', S')).$$

Since we have made the assumption that $C_1 R_D C_2$, $C_1 R_D C_3$, and $C_1 R_D C_q$, then, from (2) we can deduce that $\forall\ k \in [2\,..\,q] \mathrm{LocWeak}(C_k, S') \leq \mathrm{LocWeak}(C_k, S)$. Moreover (1) states that $\mathrm{Det}(C_1', C_1') \geq \mathrm{Det}(C_1, C_1)$, i.e., $\mathrm{LocWeak}(C_1, S') \leq \mathrm{LocWeak}(C_1, S)$. Then,

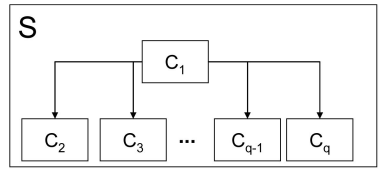


Fig. 5. Adding a contract in C_1.

$$\sum_{i=1}^{q} (\mathrm{Prob_Err}(C_i', S') \times \mathrm{LocWeak}(C_i', S')) \leq \sum_{i=1}^{q} (\mathrm{Prob_Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S))$$

and:

$$\sum_{i=q+1}^{|S|} (\mathrm{Prob_Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S)) + \sum_{i=1}^{q} (\mathrm{Prob_Err}(C_i', S') \times \mathrm{LocWeak}(C_i', S')) \leq \sum_{i=q+1}^{|S|} (\mathrm{Prob_Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S)) + \sum_{i=1}^{q} \mathrm{Prob_}(\mathrm{Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S)).$$

Finally:

$$V(S') \geq 1 - \sum_{i=q+1}^{|S|} (\mathrm{Prob_Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S)) + \sum_{i=1}^{q} (\mathrm{Prob_Err}(C_i, S) \times \mathrm{LocWeak}(C_i, S)),$$

i.e., $V(S') \geq V(S)$.

4 Model Parameterization and Case Study

In Section 3.3, three parameters have been isolated for the computation of the vigilance: $\mathrm{LocVig}(C_i, S)$, $\mathrm{Det}(C_i, C_j)$, and $\mathrm{Prob_Err}(C_i, S)$. This section reports experiments that were conducted to estimate the value of these parameters in a system. The idea is to inject errors in the system and check how many of them the contracts are able to detect. The proportion of errors detected by the contracts is an estimation of the local vigilance of a component. Mutation analysis is used as a systematic process for fault injection and the estimation of actual values for Weak_i and $\mathrm{Det}(C_i, C_j)$.

4.1 Mutation Analysis

Mutation analysis was initially proposed by DeMillo et al. in [11] to create effective test data, with an important fault revealing power [21], [22]. This analysis consists of creating a set of faulty versions of a program, called *mutants*, with the ultimate goal of designing a set of test cases 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. A mutant is a copy of the program under test into which one fault has been injected.

A set of test cases is *adequate* if it distinguishes the original program from all its nonequivalent mutants.

Otherwise, a *mutation score (MS)* is associated with the test cases set to measure its effectiveness in terms of the percentage of the revealed nonequivalent mutants. A mutant is considered *equivalent* to the original program if there exists 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 quality of the test cases. It can be viewed as a kind of reliability assessment for the test cases.

During the test selection process, a mutant program is said to be *killed* if at least one test case detects the fault injected into the mutant. Conversely, a mutant is said to be *alive* if no test case detects the injected fault (living mutant). To kill mutants, an oracle is needed for the test set. For all the experiments described in this paper, we used two oracles for the mutation analysis. They are defined below.

Behavioral difference oracle. *This oracle is very specific to the mutation analysis and is based on the assumption that the original program is correct. When running a mutation analysis for a class, let Out_o be the set of outputs when running the tests with the original class, let Out_m be the set of outputs for tests with a mutant class. If $Out_m \neq Out_o$, then the mutant is killed by the tests set. This is called the behavioral difference oracle for mutation analysis.*

For the case studies, we used a perfect behavioral difference oracle to have a reference for estimating the contract efficiency. This perfect oracle compares the objects of the mutant program and the objects of the initial program in depth (deep_equal in Eiffel). Since the comparison is made on the full internal state of the objects, it is, by definition, the most efficient oracle function, hence a perfect oracle. When nonequivalent mutants survive the perfect oracle, it simply means that our test set is incomplete.

Contracts as oracles. *When running tests for a system with executable contracts, if a contract is violated, an exception is raised and the system can stop. In the context of a mutation analysis, if the system stops because of a contract violation when running the tests on a mutant program, the mutant is killed. This is how contracts are used as oracles in mutation analysis.*

In our experiments, the choice of mutation operators (see [23] for other possible operators for OO) includes selective relational and arithmetic operator replacement, variable perturbation, but also referencing faults (aliasing errors) for declared objects:

- EHF (Exception Handling Fault): Causes an exception when executed.
- AOR (Arithmetic Operator Replacement): Replaces occurrences of "+" by "−" and vice versa.
- LOR (Logical Operator Replacement): 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 (Relational Operator Replacement): Each occurrence of one of the relational operators $(<, >, <=, >=, =, /=)$ is replaced by each one of the other operators.
- NOR (No Operation Replacement): Replaces each statement by the *Null* statement.
- VCP (Variable and Constant Perturbation): 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.
- 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 affectation. Operator RFI introduces object aliasing and object reference faults, most common in object-oriented programming.

The process of creating mutants with all the operators listed above, executing every test case with the original program and all the mutants, collecting the verdicts for all test cases, and computing the mutation score has been automated in a prototype tool. More details about mutation analysis and our implementation of this process can be found in [24]. The study is in Eiffel since design by contract is natively supported by Eiffel and there exist libraries with contracts. We could not apply OO operators since they were not available at the time we started our studies. Moreover, currently, the only available mutation tool implementing OO operators is dedicated to Java (MuJava).

4.2 Mutation Analysis for Estimating the Efficiency of Contracts

The goal of the study is to check the contracts efficiency only on mutants that are killed by one test case at least, using the behavioral difference oracle (deep-equals on object programs). We thus defined a two-steps experimental process that applies mutation analysis for two different purposes:

1. Generation of a set of test cases that has a high mutation score.
2. Evaluation of the efficiency of the contracts for a given class.

In the first step, we used mutation analysis to guide the generation of test cases for each class. The generation of unit test cases for a class consisted of generating a set of mutants and of incrementally building a set of test cases that could kill every mutant. The process started with an initial set of test cases written by hand and that covered all nominal cases for the class. Then, the mutation score for these test cases was computed using the behavioral difference as an oracle. If the mutation score was not satisfactory, we looked at the living mutants. We isolated the equivalents and added new test cases (by hand) to kill as many of the living mutants as possible. At the end of this process, a set of test cases is obtained with a mutation score between 90 percent and 100 percent for each class.

The second mutation analysis aimed at evaluating the efficiency of the contracts for a class. The intuition is that, if contracts for a class are complete, they should kill all the mutants killed at Step 1. If the contracts are not complete, the mutation score is an estimate of their ability to detect an error. This mutation analysis used the contracts as an oracle. We suppressed the equivalent mutants and the nonequivalent living mutants since there is no chance for a living mutant (with a behavioral difference oracle) being killed by a contract. We executed the test cases generated during the previous analysis on all the mutants that were killed by these test cases. For instance, if we consider that there are 10 equivalent mutants plus 10 mutants alive among 100 mutants, we reexecute the test cases against the 80 killed mutants using contracts as an oracle. If only 20 mutants are killed, the efficiency of the contracts is 25 percent (20/80). If, after the improvement of contracts, 60 mutants are killed, the efficiency of the contracts is 75 percent.

```
+BankAccount:: transfer(real sum, BankAccount account){
    balance=balance-sum;
    account.deposit(sum);
}

context BankAccount::transfer(real sum,BankAccount account)
pre sum>0 and balance-sum>=overdraft
post balance = balance@pre - sum



Mutant1 (NOR)
public void transfer(float sum, TP2Bank.BankAccount account){
   account.deposit(sum);
}
Mutant2 (NOR)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance-sum;
}
Mutant3 (AOR)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance+sum;
   account.deposit(sum);
}
Mutant4 (VCP)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance-sum;
   account.deposit(sum-1);
}
Mutant5 (VCP)
public void transfer(float sum, TP2Bank.BankAccount account){
   balance=balance-sum-1;
   account.deposit(sum);
}
```

Fig. 6. Example for isolated vigilance computation.

4.3 Process for the Estimation of Parameters

Two parameters have to be estimated to compute the global vigilance for an OO system designed by contract: the local vigilance of classes and $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$. The isolated vigilance of a class corresponds to the proportion of errors (in the class) that the contracts of this class can detect. This vigilance corresponds to the efficiency of the contracts for a class. It is estimated using mutation analysis as described in previous section. For example, Fig. 6 displays the `transfer` method of class BANKACCOUNT, with its associated pre and postcondition. This method moves the sum from the current BANKACCOUNT instance to `account`. The postcondition is correct but incomplete in the sense that it does not check the new balance of `account`. Five mutants for this method are given in the bottom part of the figure. The postcondition for `transfer` can kill mutant1, mutant3, and mutant5. Let us notice that the invariant ($\texttt{balance} \geq \texttt{overdraft}$) can also kill mutant5 with the method call, $\texttt{transfer}\ (\texttt{balance} - \texttt{overdraft}, \texttt{bAcc1})$. The mutation score is 60 percent in this particular case. To obtain the isolated vigilance for the BANKACCOUNT class, it is necessary to generate mutants for every method in the class and compute a global score.

To measure $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$ for a component $\mathrm{C_i}$, the process consists of injecting faults in a component $\mathrm{C_j}$ used by $\mathrm{C_i}$ ($\mathrm{C_j}$ is a *provider* for $\mathrm{C_i}$). Then, we execute $\mathrm{C_i}$'s tests using $\mathrm{C_i}$ and each of its faulty providers. The percentage of killed mutants in $\mathrm{C_j}$ by $\mathrm{C_i}$ is the $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$. For example, the test cases for BANK have to be executed with mutants of BANKACCOUNT. Let us consider the mutants for BANKACCOUNT given in Fig. 6 and the `automaticTransfer()` method in BANK (Fig. 7). This method allows an automatic

```
+Bank::automaticTransfer(BankAccount from, BankAccount to){
  if (from.getBalance()>threshold){
    from.transfer(from.getBalance()-threshold, to);
  }
}

context Bank::automaticTransfer(BankAccount from, BankAccount to)
post from.getBalance() + to.getBalance() =
      from.getBalance()@pre + to.getBalance()@pre
```

Fig. 7. Example for $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$ computation.

transfer from `accountFrom` to `accountTo` when the balance of `accountFrom` is greater than a given threshold. The postcondition checks that the sum of the balances of both accounts is the same before and after a call to `automaticTransfer()`. Let us consider a test case that calls `automaticTransfer()`, the `transfer` method of BANKACCOUNT will also be called and the postcondition of `automaticTransfer()` will kill mutants 2 and 4 of `transfer` that were not killed previously. Det(BANK, BANKACCOUNT) is thus equal to 40 percent.

A third parameter is necessary to compute the global vigilance, $\mathrm{Prob_error}(\mathrm{C_i}, \mathrm{S})$. We make the assumption that the probability that an error occurs in a class is the same for every class in the system. Thus, $\mathrm{Prob_error}(\mathrm{C_i}, \mathrm{S}) = 1/\mathrm{n}$, n being the number of classes in the system.

4.4 A Case Study

We use the Pylon library (http://www.nenie.org/eiffel/pylon/) as a case study for estimating the range of values for isolated vigilance and $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$. 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 used mutation analysis tool, called *mutants slayer* or μSlayer, is designed for 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 are killed by the test cases.

Starting from a system in which each class has an associated set of test cases and initial executable contracts (the ones provided by the programmer of the Pylon library), the case study first aims at studying the bounds of values that isolated vigilance can take (min, max, average). It thus consists of measuring the initial efficiency of the contracts and then the efficiency of the improved contracts. These improvements are incremental: We compute the efficiency of the initial contracts and then we add assertions into the contracts that have a low mutation score or add contracts to method without one. We go on like this until we cannot improve the contracts anymore. The results for the improvement of the contracts are given in Table 1. The table summarizes the initial efficiency of the contracts and the final level they reach after improvement. The isolated vigilance of classes is significantly improved (the best improvement is from 25 percent to 100 percent). The fact that not all errors are detected by the improved contracts reveals the limit of contracts as oracle functions. There are several reasons for this. The first one is that contracts are unable to detect errors disturbing the global state of a component. It is thus not possible to improve a contract so it can detect such an error. For example, a *prune* method of a stack cannot have trivial local contracts checking whether the element removed had been previously inserted

TABLE 1
Main Results for Contracts Improvement

	Min.	Max.	Avg.
% mutants killed (initial contracts)	17%	83%	58.5%
% mutants killed after contracts improvement	72%	100%	87.5%

by a put. In that case, even a class invariant would not be able to capture this property since it would have to check that, for all the elements in the stack, a call putting the element to the stack must have occurred before the evaluation of the invariant.

Another reason why contracts cannot be improved to kill all mutants is that some contracts become very complex in the improvement process (large number and complex assertions) and, even if they could become complete oracles, there is also great chance to introduce errors in them. As we said in Section 2.1, contracts do not aim at completely describing the behavior of the object, but aim at embedding important elements of the specification. In the process of contracts improvement, we thus have taken this into account and have stopped improving contracts that were becoming too complex. The way the contracts have been improved is similar to the levels of detail proposed in [25].

At the end of the improvement process, the contract-enabled component has a considerably greater capacity to detect errors (between 72 percent and 100 percent in the case of mutation faults for this study). As a result, this approach points out methods for which the associated contracts are not efficient enough.

To measure $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$ values, we generate the mutants for all provider classes $\mathrm{C_j}$ and we compute the mutation score for the $\mathrm{C_i}$'s test cases set on the mutants concerning the methods $\mathrm{C_i}$ uses. For example, in Fig. 8, LINKED_LIST uses LINKED_NODE and the LINKED_LIST class calls all the methods of LINKED_NODE. The process to compute Det(LINKED_LIST, LINKED_NODE) consists of generating all the mutants for LINKED_NODE and running the test cases for LINKED_LIST. The obtained mutation score corresponds to the proportion of LINKED_NODE mutants the contracts from LINKED_LIST are able to kill. This score is an estimate for Det(LINKED_LIST, LINKED_NODE).

For these measures, all the classes in the system have their contracts improved (average isolated vigilance 87 percent) and the results are shown in Table 2.

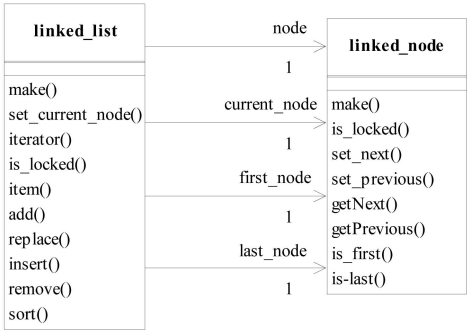


Fig. 8. Excerpt from the Pylon library.

TABLE 2
$\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$ Measures

	Min.	Max.	Avg.
% mutants of provider killed by client's contracts	50%	84%	69%

4.5 Results and Limitations of the Model

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

1. A Telecommunication 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 [26]. The class-diagram is composed of 37 classes, with a high connectivity degree (72 relationships between classes).
2. The Pylon library, which has already been presented above.
3. The InterViews (IV) library, composed of 146 classes and 420 relationships.

For these three systems, we show the evolution of the global vigilance with the improvement of isolated components' vigilance. To illustrate this evolution, we make the assumption that the $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j})$ probability depends on the component's vigilance as follows: $\mathrm{Det}(\mathrm{C_i}, \mathrm{C_j}) = \mathrm{K} \cdot \mathrm{Vig_i}$, where $\mathrm{K} < 1$ is a coefficient that expresses the loss of vigilance efficiency for the client component $\mathrm{C_i}$ when detecting an erroneous state propagated by a provider component $\mathrm{C_j}$. This assumption is based on the observations done when checking the efficiency of contracts to detect faults propagated by providers.

Using the measures in Table 2, we estimate the coefficient K as $K = 0.8$. The global vigilance evolutions for the three systems are shown Fig. 9.

With these results, we see that using no contracts (i.e., isolated vigilance equals 0) implies that the system is not

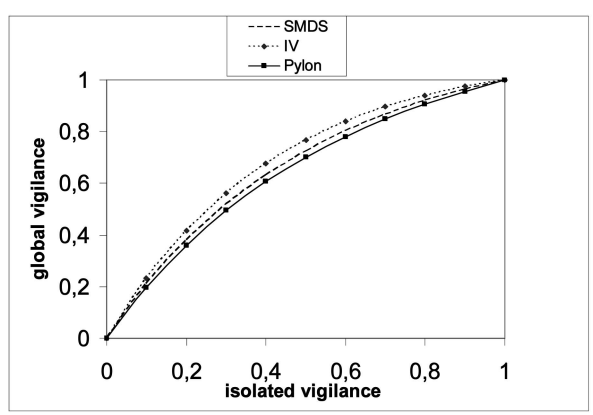


Fig. 9. Evolution of global vigilance for three systems.

vigilant, but that adding simple contracts (that are not very efficient) improves the global vigilance rapidly. For example, in the InterViews system, if the components isolated vigilance is 0.4, the global vigilance is almost 0.7. Moreover, the three curves show that improving the isolated vigilance from 0.8 to 1, which corresponds to the most costly improvements, is not interesting in terms of global vigilance improvement: For InterViews, the global vigilance is already 0.94 when the isolated components' vigilance is 0.8.

The slight differences between the systems correspond to different coupling values (dependency densities). Indeed, the local vigilance of a component can be increased by its clients' contracts and improving local vigilance improves global vigilance. So, the more coupling there is between components, the more local vigilance is increased and the higher the global vigilance. It is well-known that coupling is not a desirable feature for a system since it is a factor of complexity and is error-prone. A good design aims at reducing the coupling and, thus, limits the global vigilance. However, coupling cannot be completely avoided and, thus, contracts allow us to make maximum use of this otherwise negative property to improve the global vigilance.

In [27], Briand et al. analyze the sensitivity of the mathematical model we propose in [9] (vigilance having been improperly called robustness). The model is studied and generalized under the assumption that each component has the same isolated vigilance Vig. The following simplified formula is obtained:

$$\mathrm{LocVig}(\mathrm{C_i}, \mathrm{S}) = 1 - (1 - \mathrm{Vig})^{*}(\mathrm{K} \times (\mathrm{Vig})^{\mathrm{cl}},$$

where cl is the number of clients of component $\mathrm{C_i}$.

Briand et al. then assign a constant value to the average number of clients for the components in a system. The local vigilance is then also a constant LocVig. Under the assumption that each component has the same probability 1/n of being erroneous, the global vigilance V of a system is:

$$\mathrm{V} = \sum_{\mathrm{i}=1}^{\mathrm{n}} (1/\mathrm{n} \cdot \mathrm{LocVig}(\mathrm{C}, \mathrm{S})) = \mathrm{LocVig}.$$

With this simplified model, Briand et al. study the sensitivity of the model to the number cl of clients of a class and the coefficient K. They take values for cl between 1 and 6 and values for K between 0.2 and 0.8. Their results show that, for these values, the model is actually sensitive to both parameters: There is an important variation for the global vigilance depending on the value of these parameters. However, our experimental measures show that, in practice, K varies only between 0.5 and 0.8. Since we have observed only small variations of K in our studies, the results obtained with our model are still relevant considering the variations of K. Since the number cl is not a parameter of our model, the sensitivity analysis of cl has no relevance to the model we propose (it is suited to the simplified model of Briand et al.). Precisely because the coupling is important for vigilance, the dependencies for each class are counted to obtain local vigilance values for each class. For the three case studies shown in Fig. 9, the number of clients has been calculated for each class of the class diagram.

5 Measuring Diagnosability

A failure may be observed during the software development as well as the maintenance stage. *Diagnosis* is defined as the task that consists of locating faulty parts of a system when a failure is detected. In the presence of contracts, diagnosis is simplified in the following way: First, there is no need to look at statements after the violated contract in the execution thread. Second, in practice, locating the faulty statement consists of proceeding backward from a violated contract. The efficiency of diagnosis is thus linked to vigilance since the greater the local vigilance, the more errors will be detected by contracts and the closer the fault might be to the violated contract.

Confronted with the problem of diagnosis, which remains a nonautomated task, it would be useful to appraise the probable difficulty of locating faults in the software beforehand. Such predictor estimation is called *diagnosability* (see [28] for a study of diagnosability in data flow designs) and provides a way of improving the design quality. Diagnosability refers to the ability to locate a fault that has been detected during testing, i.e., the fault localization effort. We study this factor in the context of designed by contracts systems.

5.1 Diagnosis Practices in the Software Domain

The diagnosability of a program depends on its internal structure and on the set of test cases (used to detect faults). To analyze it, one needs to understand the main methods used for locating faults in the software after they have been detected by tests.

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 [29], [30], [31]. Along this line, 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 [32], [33], [34], [35], [36] which basically consist of 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 of 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 possible technique for locating faulty statements consists of inserting assertions in the program for detecting some internal erroneous 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 [1]. We focus on the diagnosability of the software which has been designed using such a technique. We analyze the impact of executable contracts/assertions on system diagnosability.

5.2 Diagnosability Measure

Diagnosability, defined as the fault localization effort, is related to the size of the sets of suspect statements. The larger are the suspected sets, the more difficult the fault localization. It is intuitively more difficult to distinguish the faulty statement among 10 statements than among two

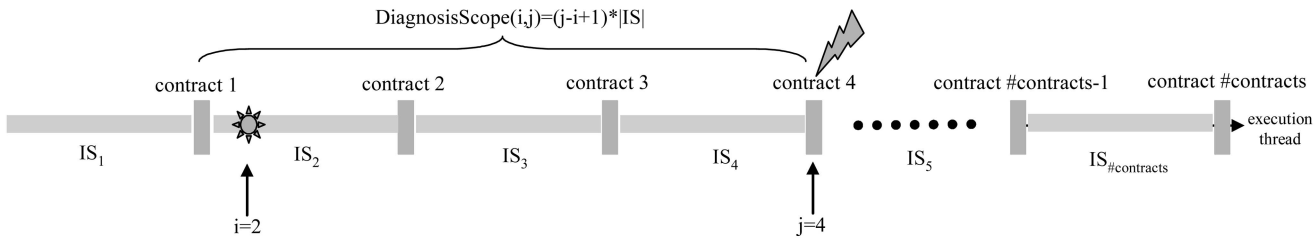


Fig. 10. Indistinguishability sets and contracts along an execution thread.

statements. A set of statements which are equally suspect is said to be indistinguishable. Thus, the underlying attribute expressing the localization of faulty statements among a set of suspect statements is called *indistinguishability*.

The input for the diagnosability measure is the set of statements executed (the execution thread) when a failure occurs or when a contract detects an error. This section starts with the mathematical definition of the diagnosability measure. Then, the tuning of the parameters (vigilance of the system, number of indistinguishable statements, execution thread) is explained and the evolution of the diagnosability of a system is computed for different values of the parameters.

The model for the measure is based on the following assumptions:

- The software is assumed to be faulty: There exists an execution of the system that provokes a failure if the error is not detected by a contract.
- Contracts are assumed to be correct.
- If an execution thread is faulty on multiple points, the diagnosis will point out the first divergence point (faults that compensate for each other are considered as negligible for a global estimate).

The core attribute we need to express diagnosability is the notion of indistinguishability set, defined as follows:

Indistinguishability set (IS). *An indistinguishability set corresponds to a set of statements bounded by consecutive contracts in an execution thread.*

5.2.1 Local Diagnosability Measure

To compute the diagnosability of a system, we first define the probability that a faulty statement is detected by a contract in any execution thread T composed of *#contracts* contracts. Since we do not distinguish statements in an indistinguishability set, this probability is the same for every statement in such a set. This probability depends on the number of contracts along an execution thread, #contracts.

$\mathbf{Det_i^j}$. *$Det_i^j (i \in [1 \ldots \#contracts]$ and $j \in [i \ldots \#contracts])$ is the probability that a faulty statement in an indistinguishability set IS_i is detected by the contract j knowing that no intermediary contract has detected this erroneous state. IS_i is the ith indistinguishability set along an execution thread (bounded by two consecutive contracts as illustrated in Fig. 10).*

We have the following constraint on Det_i^j probabilities:

$$\sum_{j=i}^{\#\mathrm{contracts}} \mathrm{Det}_i^j + \mathrm{Pfailure} = 1,$$

where Pfailure is the probability that no contract has detected the faulty statement and a failure occurs. Let p_k be the probability that the contract k detects the erroneous state due to a faulty statement in IS_i. Since, for a contract k, the nondetection of the erroneous state by any previously executed contracts is independent events, we have:

$$\mathrm{Det}_i^j = p_j \cdot \prod_{k=i}^{j-1} (1 - p_k).$$

The second parameter is the *diagnosis scope*. The *diagnosis scope* for a detected fault is the number of statements executed between the indistinguishability set that contains the fault and the error detection point. Since all statements of the indistinguishability set which contains the fault are suspected, they are included in the diagnosis scope. When contracts are embedded in the program, the diagnosis scope for every statement in one particular indistinguishability set is the same. Based on this notion of diagnosis scope, we define the *diagnosis effort*. For a fault in an indistinguishability set IS_i, this effort is the probable number of statements which are suspect, knowing the probability each contract has to detect the erroneous state along the execution thread.

Diagnosis effort of $\mathbf{IS_i}$ ($\boldsymbol{\delta_i}$). *The local diagnosability δ_i of any statement in an indistinguishability set IS_i along an execution thread T is the effort needed for determining that IS_i is faulty in T. It is equal to:*

$$\delta_i = \sum_{j=i}^{\#\mathrm{contracts}} \mathrm{DiagnosisScope}(i,j) \times \mathrm{Det}_i^j + \mathrm{Pfailure} \times \mathrm{DiagnosisScope}(i, \#\mathrm{contracts}),$$

where:

$$\mathrm{Pfailure} = 1 - \sum_{j=i}^{\#\mathrm{contracts}} \mathrm{Det}_i^j$$

and

$$\mathrm{DiagnosisScope}(i,j) = \sum_{k=i}^{j} |IS_k|.$$

Indeed, if the fault in IS_i is detected by contract j, then the diagnosis scope associated to the indistinguishability set IS_i is equal to the number of statements between the faulty one to the contract's detection. The last term of δ_i equation is added to measure the diagnosability scope when no contract detected the faulty statement (a failure thus occurs at the outputs of the execution thread).

Global diagnosis effort for an execution thread ($\boldsymbol{\delta}$). *The global diagnosis effort for an execution thread T is the probable*

TABLE 3
Contracts Distribution

	#statements	#contracts	#stats / #contracts
List Class	1391	505	2,75
Virtual Meeting Server	2291	1171	1,96
JUnit Auto-Test	19419	10801	1,8
Loading JDK	111730	40751	2,74
Jtree	1970745	885001	2,22

effort needed for pointing out the faulty statement, knowing that a fault is detected in T:

$$\delta = \sum_{\mathrm{i=1}}^{\#\mathrm{contracts}} (\mathrm{Pfaulty_i} \times \delta_\mathrm{i}).$$

The diagnosis effort for an execution thread is the probable diagnosis scope knowing the probabilities $Pfaulty_i$ each $\mathrm{IS_i}$ has of containing the faulty statement.

In practice, it is too hard to directly calculate $Pfaulty_i$ and δ, so we make an approximation based on the following additional assumptions:

1. The contract distribution in an execution thread is uniform. Each IS has the same size |IS| (= #stats div #contracts, where #stats is the number of statements in the program).
2. The closer a contract is to the faulty statement in $\mathrm{IS_i}$, the higher the probability that it has to detect the fault (i.e., $\mathrm{p_k}$ decreases when k grows, $\mathrm{k} \in [\mathrm{i..\#contracts}]$).
3. The contracts have an equal probability p of detecting a fault coming from the statements directly preceding them (the contracts being written to check these statements).
4. Each statement has the same probability of being faulty, equal to 1/#stats.

The first assumption seems less realistic since it implies that all the distances between method calls and returns in a thread are equal. Nevertheless, experiments have been conducted on a set of OO programs to look at the actual distribution of method calls along an execution thread as an estimation of the contracts distribution. Under the assumption that there is a precondition at the beginning of each method and a postcondition at the end, the distribution of method calls along an execution thread can indeed approximate the contracts distribution. We measured this distribution for five systems. The execution threads had sizes from 1,400 statements to almost two millions. The exact characteristics for each system are given in Table 3 and the curves of distribution are given in Appendix A. From those experiments, it appears that contracts are actually homogeneously distributed in these systems and that assumption 1 is valid and can be taken into account to estimate the impact of contracts on the diagnosability of systems.

The second assumption fits the intuition and has not been experimentally verified. The first and third assumptions lead to a simplified measure that will reflect the global impact of contract improvement on the diagnosis effort. Assumption 3 is used to get the impact on diagnosability by varying the average efficiency of contracts. Assumption 4 is reasonable in a predictive approach since we must equally suspect any part of the software to be faulty.

To be realistic, to model assumption 2 and be consistent with assumption 3, we consider that if the first executed contract has a probability p of detecting the erroneous state, the second has only $\alpha.\mathrm{p}$ probability of detecting it, the third one $\alpha^2.\mathrm{p}$, and so on. The probability p corresponds to the efficiency of the contract that directly follows the faulty statement and α is a constant *absorption coefficient* ($\alpha \in [0..1]$). If α is equal to 1, it means that assumption 2 is not verified, all contracts being equivalent. If α is equal to 0, it means that only the first executed contract can detect the fault. So, the probability a contract j detects an error due to a fault in $\mathrm{IS_i}$ is equal to the probability $\alpha^{\mathrm{j-i}}.\mathrm{p}$, which is the probability that contract j detects the fault while all the preceding contracts have failed. This probability is:

$$\left(\prod_{k=i}^{j-1}(1-\alpha^{k-i}p)\right).$$

We thus have:

$$\begin{aligned}\mathrm{Det_i^j} &= p_j \cdot \prod_{\mathrm{k=i}}^{\mathrm{j-1}}(1-\mathrm{p_k}) = \alpha^{\mathrm{j-i}}\mathrm{p} \cdot \prod_{\mathrm{k=i}}^{\mathrm{j-1}}(1-\mathrm{p_k}) \\ &= \alpha^{\mathrm{j-i}}\mathrm{p} \cdot \prod_{\mathrm{k=i}}^{\mathrm{j-1}}(1-\alpha^{\mathrm{k-i}}\mathrm{p}).\end{aligned}$$

Fig. 10 presents an execution thread with contracts. If there is a fault in $\mathrm{IS_2}$ that is detected by contract 4, the DiagnosisScope(2, 4) is equal to $3 * |\mathrm{IS}|$. More generally, we have $\mathrm{DiagnosisScope(i,j)} = (\mathrm{j-i+1}) * |\mathrm{IS}|$. The local diagnosis effort can be deduced as follows:

$$\begin{aligned}\delta_\mathrm{i} &= \left[\sum_{\mathrm{j=i}}^{\#\mathrm{contracts}} \mathrm{DiagnosisScope(i,j).Det_i^j} \right. \\ &\quad \left. + \left(1 - \sum_{\mathrm{j=i}}^{\#\mathrm{contracts}} \mathrm{Det_i^j}\right) \times (\#\mathrm{contracts} - \mathrm{i} + 1)\right] \\ &= |\mathrm{IS}| \left[\sum_{\mathrm{j=i}}^{\#\mathrm{contracts}} (\mathrm{j-i+1}).\mathrm{Det_i^j} \right. \\ &\quad \left. + \left(1 - \sum_{\mathrm{j=i}}^{\#\mathrm{contracts}} \mathrm{Det_i^j}\right) \times (\#\mathrm{contracts} - \mathrm{i} + 1)\right].\end{aligned}$$

5.2.2 Model Tuning

These last equations are easily computable and depend on three parameters: p, α, and #contracts. The question now is how to count the number of contracts and how to evaluate their efficiency and the absorption coefficient, which are the main parameters of the model. In any system, an appropriate instrumentation of the code would lead to an exact counting for #stats and #contracts for a given execution thread.

Section 4.3 shows that contracts' efficiency varies between 0.17 for less efficient ones to 1 for the best one. An average for a reasonable effort is around 0.87. This efficiency corresponds to the probability p that a contract directly following a faulty IS in an execution thread detects the error.

Concerning the absorption coefficient, Fig. 11 gives the various curves that reveal the sensitivity of the model to this coefficient. We study the case in which there are 10 contracts between the faulty statement in an $\mathrm{IS_i}$ and the end of the execution thread ($\#\mathrm{contracts} - \mathrm{i} = 10$). Given an absorption coefficient, each curve displays the probable number of

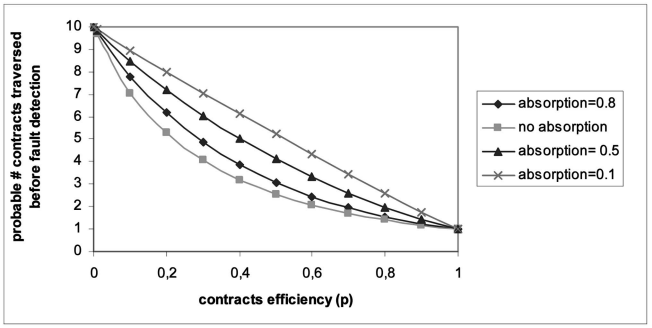


Fig. 11. Tuning for 10 contracts.

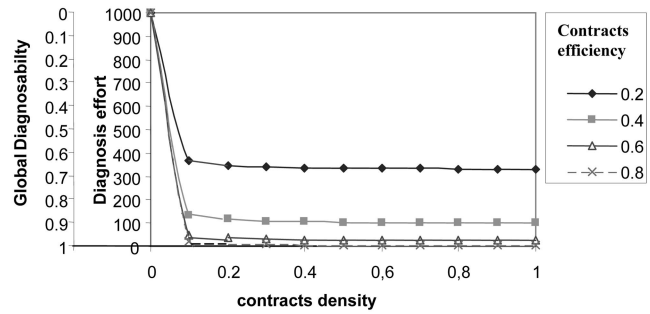


Fig. 12. δ and Δ results—Main results.

contracts which are executed before the error is detected as a function of the contracts' efficiency (p). For contracts with efficiency of 0.4, the probable number of traversed contracts before detection is between 3.1 and 6.1 (delta is of three contracts). A significant loss of precision for the global diagnosability measure can be due to a bad estimation of the absorption coefficient (particularly if contracts' efficiency is between 0.2 and 0.7). We calibrate the absorption coefficient to 0.8, which corresponds to the mean value observed during experiment with the Pylon Eiffel library.

5.2.3 Global Diagnosability Measure

For local measure (attached, respectively, to a statement and to an execution thread), the diagnosability was only expressed in terms of diagnosis effort (in that case, a diagnosability improvement corresponded to a reduced diagnosis effort). At global level, it is also useful to have a ratio to simplify comparison from one program to the other. The global diagnosability measure (attached to a system) thus corresponds to the accuracy of the diagnosis and is equal to 0 if the diagnosis effort is equal to the complete execution thread, while it is equal to 1 when the diagnosis effort is restricted to 1 statement.

Global diagnosability of a system (Δ)**.** *The global diagnosability of a system S is the probable degree of diagnosis accuracy obtained depending on the embedded contracts density and efficiency. It is computed from the global diagnosis effort* δ *as follows:* $\Delta = 1 - \delta/\#stats$.

With the assumptions already given, the diagnosis effort curve (for an execution thread and a given average contract efficiency, function of the contract density—number of contracts per statement) is proportional to the size of the execution thread.

Fig. 12 presents the same results with both scales in terms of "absolute" diagnosis effort and in terms of global diagnosability. Diagnosis effort curves are given for various contracts' efficiency with a 1,000-statement thread—the absorption coefficient is equal here to 0.8 (it corresponds to the factor K used to compute vigilance in Sections 3 and 4).

The global diagnosability is thus directly obtained by any execution thread and, for example, by those given in Fig. 12. The gap between local diagnosis effort values and global diagnosability is thus bridged thanks to a useful property of the model: The global diagnosability of a system only depends on its contracts' efficiency and density as shown in Fig. 12.

5.3 Discussion on Diagnosability

Let us interpret the results of Fig. 12. First, we remark that the introduction of contracts quickly enhances the global diagnosability of the system. Second, the addition of many contracts (high contract density) does not significantly improve a system global diagnosability (which is upper bounded around 0.6 with 0.2 efficient contracts or around 0.9 with 0.4 efficient contracts). Third, the efficiency of the contracts (isolated vigilance) is more important than their number since it is the only way to make the upper bound to diagnosability increase.

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

- A 0.2 contract density is enough to reach the upper bound of diagnosability for a given contract average efficiency. This density corresponds to the execution of one contract every five statements (on average). This is realistic in an OO system where methods are often small and where delegation is intensively used. Indeed, each delegation implies a method call and, thus, the execution of its contracts. 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 density, the efficiency of contracts highly changes the diagnosability. It is important to put the effort into the design of well-defined interfaces and efficient contracts.

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

6 Related Work

To our knowledge, a few works have tackled the issue of estimating the qualitative impact of design by contract for software construction. Briand et al. have studied the impact of contracts for testability of object-oriented systems in [25]. Nordby et al. [5] have proposed a development methodology based on contracts and showed that it speeds up the testing and integration phases. Other works have focused on contracts as a means for testing [37], [38] or proving [39].

In [25] and [27], Briand et al. investigate the instrumentation of analysis contracts in object-oriented systems for helping the detection and isolation of faults. First, they provide the guidelines they have followed to add contracts to the system they want to study and give details about the translation of contracts written in OCL to executable contracts written with a commercial tool for Java. The rest of the paper consists of studying the impact of the analysis contracts as a substitute for test oracles and as a factor for helping diagnosability. The case study consists of an ATM system written in Java. The study of contracts as test oracles consists of seeding faults in the system, running test cases

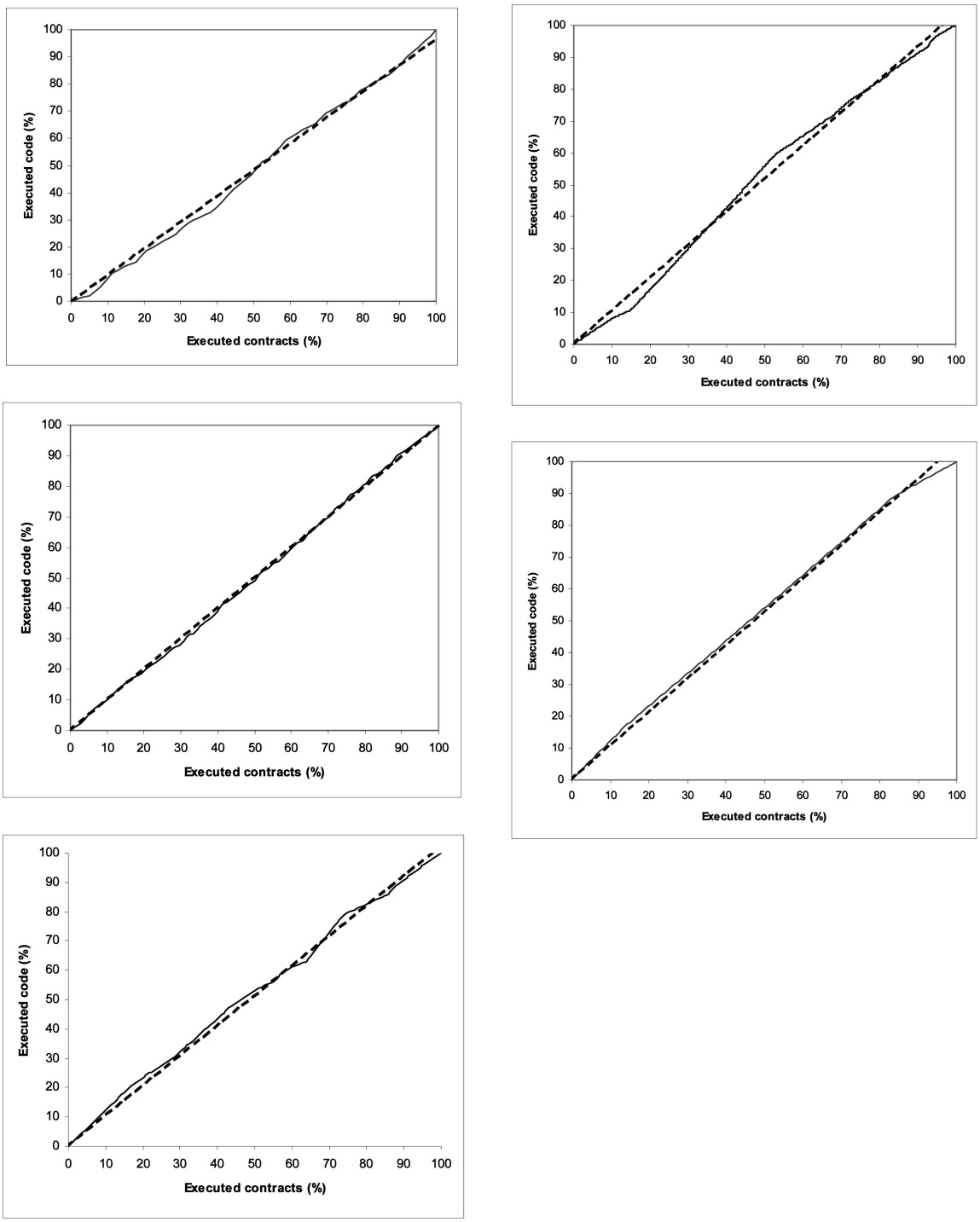


Fig. 13. Distribution of method calls along an execution thread in five systems.

on the faulty versions, and measuring the effectiveness of failure detection by contract violation. They seed 14 different types of faults, but the fault seeding is done by hand and the study is run with only 69 faulty versions of the system. For the diagnosability analysis, they run the test cases on the faulty versions of the system with contracts as oracles and, without contracts, using a manual oracle. Then, in the case where a fault has been detected by a contract and the explicit oracle, they look at the execution thread to determine whether contracts can actually help the diagnosis. The whole is very detailed in the paper, but the lack of systematic approach for the mutation analysis and of an abstract model for both factors makes the conclusions difficult to generalize.

In [5], Nordby et al. propose a methodology to use contracts in the different phases of the development process. They distinguish two types of contracts: weak and strong. Strong contracts require that the client satisfies a specific condition, the postcondition then states the outcome only in the legal situations. For weak contracts, the client has no obligation (precondition is true), the postcondition then has to take into account the outcome even in the case of a

meaningless call. Both types of contracts are used at different moments during the development of the software: Strong contracts help in writing unit components and finding internal errors, while weak contracts are used when the component is released for integration. In [5], the authors propose a systematic method to weaken strong contracts. Then, results are given for an industrial case study. Strong contracts enable rapidly detecting faults when writing two components. Contracts have then been weakened and no error was introduced during this operation.

Other works on contracts focus on the methodology to improve software quality through testing or proof. In [39], Meyer proposes a general framework to prove object-oriented classes designed by contract. The idea is that contracts defined for classes correspond to minimal specification of the behavior. It should then be proven that the implementation actually conforms to the specification. In the paper, he takes an Eiffel library as an example and suggests a solution to prove effective implementations of abstract classes. Then, he lists the different issues that arise to actually prove the classes. The first one is to be able to derive abstract mathematical properties from the model contracts. The second one is the scalability of the approach: It should be applicable on a larger scale than a class, for example, for a component.

Several works study contracts for testing, in particular to automatically derive an oracle function. In [37], the authors derive contracts from a formal specification for CORBA components. Several rules are defined to prove the consistency between the contracts obtained after derivation and the formal specification. The contracts are then complete and can be used as efficient embedded oracle functions for testing. In [38], the authors use contracts written with JML to derive oracles for test cases in the JUnit format. They make no assumption about the completeness of contracts and the technique they propose is more to facilitate the design and the refactoring of JUnit test cases.

A secondary contribution of this work is the adaptation of mutation analysis to evaluate the efficiency of contracts (how complete they are). In other words, we have proposed a way to estimate the completeness of the specification of the program under test. Some works related to this issue have used mutation analysis on models to evaluate the coverage (a sort of completeness) provided by a specification in the context of model-checking. In [40], Hoskote et al. first proposed a measure for estimating the coverage for model-checking using mutation operators defined on finite State Machines. The idea of this work was to evaluate the validity of formulas that are expressed on a State Machine. Chockler and Kupferman extended this work in [41] for coverage of Kripke structures.

7 Conclusion

Although there are several works that advocate the use of assertions to improve software quality or the application of Design by Contract to improve the design of a system, few studies actually focus on measuring the impact of such approaches. The work presented in this paper focuses on measuring the impact of Design by Contract on two quality factors: vigilance and diagnosability.

This study first consisted of precisely defining the measures of vigilance and diagnosability, expressing the expected properties for these measures, and in identifying the measurable parameters that had to be taken into account. An important parameter was the efficiency of contracts. We thus propose a definition of contract efficiency related to contract completeness and an adaptation of mutation analysis to measure this efficiency.

Experimental work allowed us to tune the models for the measures with values coming from an Eiffel case study. We then computed the evolution of vigilance and diagnosability of several OO systems as a function of the efficiency of their contracts. It appeared that efficient contracts significantly contributed to improving the quality of the systems, but also that their efficiency is more important than their quantity. Furthermore, identifying clear interfaces to which precise and efficient contracts can be attached seems a good trade-off to improve the quality of software. Further studies should focus on expressing rules and define guidelines to efficiently write contracts when designing an OO system.

Appendix A

The curves presented in Fig. 13 represent the distribution of method calls along an execution thread in five systems: Unit Testing, Virtual Meeting, Junit Auto-Test, Loading JDK, Jtree. In each case, the black curve is the actual distribution and the dotted one the linear regression.

References

[1] J.M. Voas and L. Kassab, "Using Assertions to Make Untestable Software More Testable," *Software Quality Professional,* vol. 1, no. 4, 1999.

[2] D.S. Rosenblum, "A Practical Approach to Programming with Assertions," *IEEE Trans. Software Eng.,* vol. 21, no. 1, pp. 19-31, Jan. 1995.

[3] M. Carrillo-Castellon, J. Garcia-Molina, E. Pimentel, and I. Repiso, "Design by Contract in Smalltalk," *J. Object Oriented Programming,* vol. 8, no. 7, pp. 23-38, 1996.

[4] R.B. Findler and M. Felleisen, "Contract Soudness for Object-Oriented Languages," *Proc. Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '01),* Oct. 2001.

[5] J.E. Nordby, M. Blom, and A. Brunstrom, "On the Relation between Design Contracts and Errors: A Software Development Strategy," *Proc. Int'l Conf. and Workshop Eng. of Computer-Based Systems (ECBS '02),* Apr. 2002.

[6] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall, 1992.

[7] B. Meyer, "Towards More Expressive Contracts," *J. Object Oriented Programming,* pp. 39-43, 2000.

[8] J.-M. Jézéquel, D. Deveaux, and Y. Le Traon, "Reliable Objects: A Lightweight Approach Applied to Java," *IEEE Software,* vol. 18, no. 4, pp. 76-83, July/Aug. 2001.

[9] B. Baudry, J.-M. Jézéquel, and Y. Le Traon, "Robustness and Diagnosability of Designed by Contracts OO Systems," *Proc. Software Metrics Symp. (Metrics '01),* Apr. 2001.

[10] Y. Le Traon, F. Ouabdessalam, C. Robach, and B. Baudry, "From Diagnosis to Diagnosability: Axiomatization, Measurement and Application," *J. Systems and Software,* vol. 65, no. 1, pp. 31-50, 2003.

[11] R. DeMillo, R. Lipton, and F. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34-41, Apr. 1978.

[12] J.-C. Laprie, *Dependability: Basic Concepts and Terminology,* 1992.

[13] J.-M. Jézéquel and B. Meyer, "Design by Contract: The lessons of Ariane," *Computer,* vol. 30, no. 1, pp. 129-130, Jan. 1997.

[14] B. Meyer, "Applying Design by Contract," *Computer,* vol. 25, no. 10, pp. 40-51, Oct. 1992.

[15] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* MIT Press/Mc Graw Hill, 1986.

[16] OMG, Object Constraint Language Specification, http://www.omg.org/docs/ad/97-08-08.pdf, 2002.

[17] L. Briand, S. Morasca, and V.S. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. Software Eng.,* vol. 22, no. 1, pp. 68-86, Jan. 1996.

[18] N.E. Fenton and R.W. Whitty, "Axiomatic Approach to Software Metrication through Program Decomposition," *The Computer J.,* vol. 29, no. 4, pp. 330-339, 1986.

[19] B. Kitchenham, S.L. Pfleeger, and N. Fenton, "Towards a Framework for Software Measurement Validation," *IEEE Trans. Software Eng.,* vol. 21, no. 12, pp. 929-944, Dec. 1995.

[20] M. Shepperd and D. Ince, *Derivation and Validation of Software Metrics.* New York: Oxford Univ. Press, 1993.

[21] A.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, 1996.

[22] J.M. Voas and K. Miller, "The Revealing Power of a Test Case," *Software Testing, Verification, and Reliability,* vol. 2, no. 1, pp. 25-42, 1992.

[23] Y.-S. Ma, Y.-R. Kwon, and A.J. Offutt, "Inter-Class Mutation Operators," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '02),* Nov. 2002.

[24] B. Baudry, Y. Le Traon, J.-M. Jézéquel, and V.L. Hanh, "Trustable Components: Yet Another Mutation-Based Approach," *Proc. First Symp. Mutation Testing,* Oct. 2000.

[25] L. Briand, Y. Labiche, and H. Sun, "Investigating the Use of Analysis Contracts to Support Fault Isolation in Object Oriented Code," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA '02),* June 2002.

[26] T. Jéron, J.-M. Jézéquel, Y. Le Traon, and P. Morel, "Efficient Strategies for Integration and Regression Testing of OO Systems," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '99),* Nov. 1999.

[27] L.C. Briand, Y. Labiche, and H. Sun, "Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code," *Software Practice and Experience,* vol. 33, no. 7, 2003.

[28] Y. Le Traon, F. Ouabdessalam, and C. Robach, "Software Diagnosability," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '98),* Nov. 1998.

[29] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving Test Suites for Efficient Fault Localization," *Proc. Int'l Conf. Software Eng. (ICSE '06),* May 2006.

[30] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," *Proc.f European Conf. Object-Oriented Programming (ECOOP '05),* July 2005.

[31] J.A. Jones and M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault Localization Technique," *Proc. Automated Software Eng. (ASE '05),* Nov. 2005.

[32] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 352-357, 1984.

[33] M. Weiser, "Programmers Use Slices when Debugging," *Comm. ACM,* vol. 25, no. 7, pp. 446-452, 1982.

[34] M. Kamkar, "An Overview and Comparative Classification of Program Slicing Techniques," *J. Systems and Software,* vol. 31, no. 3, pp. 197-214, 1995.

[35] B. Korel, "Computation of Dynamic Program Slices for Unstructured Programs," *IEEE Trans. Software Eng.,* vol. 23, no. 1, pp. 17-34, Jan. 1997.

[36] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault Localization Using Execution Slices and Dataflow Tests," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE '95),* Oct. 1995.

[37] P. Fenkam, H. Gall, and M. Jazayeri, "Constructing Corba-Supported Oracles for Testing: A Case Study in Automated Software Testing," *Proc. Automated Software Engi. (ASE '02),* Sept. 2002.

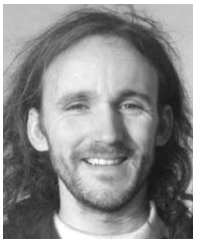
[38] Y. Cheon and G.T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way," *Proc. European Conf. Object-Oriented Programming (ECOOP '02),* June 2002.

[39] B. Meyer, "A Framework for Proving Contracts-Equipped Classes," *Proc. Int'l Workshop Abstract State Machines,* Mar. 2003.

[40] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage Estimation for Symbolic Model Checking," *Proc. 36th Ann. Conf. Design Automation (DAC '99),* 1999.

[41] H. Chockler and O. Kupferman, "Coverage of Implementations by Simulating Specifications," *Proc. Theoretical Computer Science,* Aug. 2002.

Yves Le Traon received the engineering degree and the PhD degree in computer science from the Institut National Polytechnique de Grenoble, France, in 1997. From 1998 to 2004, he was an associate professor at the University of Rennes. He is now a research engineer at France Télécom Research and Development, where he is an expert on MDA and validation and an associate member of the IRISA research laboratory in the Triskell team. His research interests include OO testing, design for testability, model-driven validation, and software measurement.



Benoit Baudry received the PhD degree in computer science from the University of Rennes, France, in 2003. He first worked at CEA (French government nuclear agency) before joining INRIA in 2004. He is now a researcher in software engineering on the Triskell team of the IRISA lab. His research interests concern software testing, fault localization, design for testability, and software modeling in the context of model-driven software development. He is a member of the IEEE and the IEEE Computer Society.

Jean-Marc Jézéquel received the PhD degree in computer science from the University of Rennes, France, in 1989. He joined the CNRS (Centre National de la Recherche Scientifique) in 1991. Since October 2000, he has been a professor at the University of Rennes, leading an INRIA research team called Triskell. His interests include model-driven software engineering based on object-oriented technologies for telecommunications and embedded systems. He is a member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.