Testing techniques selection based on ODC fault types and software metrics

Domenico Cotroneo, Roberto Pietrantuono, and Stefano Russo

Department of Computer and Systems Engineering, Federico II University of Naples,
Via Claudio 21, 80125 Naples, Italy.
E-mail: {cotroneo, roberto, Stefano.Russo}@unina.it

Abstract

Software testing techniques differ in the type of faults they are more prone to detect, and their performance varies depending on the features of the application being tested. Practitioners often use informally their knowledge about the software under test in order to combine testing techniques for maximizing the number of detected faults.
This work presents an approach to enable practitioners to select testing techniques according to the features of the software to test. A method to build a testing-related base of knowledge for tailoring the techniques selection process to the specific application(s) is proposed. The method grounds upon two basic steps: *i)* constructing, on an empirical basis, models to characterize the software to test in terms of *fault types it is more prone to contain*; *ii)* characterizing testing techniques with respect to *fault types they are more prone to detect* in the given context.
Using the created base of knowledge, engineers within an organization can define the mix of techniques so as to maximize the effectiveness of the testing process for their specific software.

Keywords: Software Testing

1. Introduction

During software testing, engineers have to cope with several heterogeneous software faults, which have to be detected and fixed before releasing the product. How many faults are in the software, and of what type, depends on several factors, such as the techniques adopted along the development cycle, the developers' skills, and the size and complexity of the system under test. In modern complex software systems, it is often the case that applying just one testing technique is not sufficient to meet the desired quality with acceptable resources. Indeed, each technique pursues a specific goal, resulting more or less prone towards some type of fault, more or less suitable for a specific kind of system, or for a particular kind of organization (e.g., techniques based on software model-checking may perform well in highly concurrent and self-contained systems; but it may require skilled personnel in the organization, and may present limits for large "system of systems" with many off-the-shelf third-party components entailing unknown interactions). In order to achieve high effectiveness, a proper combination of testing techniques should be employed.
However, how to select and combine the most proper testing techniques is still an open issue in software engineering communities, and in industrial practice as well. The challenge is to figure out what are the most effective techniques for the specific software being tested, and what techniques are more prone to detect a specific type of faults. This is currently left to the tester's intuition and expertise, resulting each time in different and unpredictable outcomes. Such practice often leads to immature testing processes: decisions are made based on subjective abilities, and the knowledge that may derive from past experiences in the organization remains implicit in the people's mind, unstructured, and, ultimately, not fully exploited. Improving the testing process demands for making such knowledge explicit and available, in order to structure procedures and policies by which the testing team can make decisions on an objective, and possibly, quantitative basis.
This paper addresses the issue of techniques selection in software testing process. A method is defined to allow a testing team characterizing techniques adopted in the organization and their best applicability

conditions according to the features of the software under test.
The method is based on two complementary steps. The former consists in constructing a set of predictive models to characterize a software application considering the perspective of the tester, who is ultimately interested in figuring out how many faults are present in the application and of what type. The procedure for such a construction is based on empirical analyses, whose objective is to infer the relationship between the number of faults of different types present in a software application and features characterizing it. The types of fault considered are as intended by a well-known fault classification scheme, i.e., the Orthogonal Defect Classification (ODC) (Chillarege et al., 1992), whereas *features* are intended as attributes characterizing the software product, usually referred to as product metrics (to distinguish them from process metrics). Example of such metrics are the widely-adopted size and complexity metrics, e.g., total number of lines, and number of lines containing comments or declarations (Ostrand et al., 2005), McCabe's cyclomatic complexity and Halstead's metrics (Fenton and Pfleeger, 1998).
The output of this characterization is a set of models to estimate the number of faults of each ODC type an application is expected to contain starting from its metrics.
The latter step of the method defines a procedure, based on controlled experiment, to characterize testing techniques in terms of ODC fault types they are more prone to detect. Once the set of techniques of interest for the organization is chosen, the characterization consists in evaluating their performance with respect to several factors, including: ODC fault types by which the system is potentially affected, detection cost, and detection effectiveness.

With these product and technique analysis, the method intends to relate testing techniques to software features through a characterization of the software in terms of expected fault types and number.
To use the method, engineers of a given organization would first characterize the software to test, by means of metrics and predictive models, obtaining an estimate of faults affecting it, and then they would select the most effective combination of techniques with respect to that characterization.
In the following, the basic actions to implement the method are first outlined. Then, we show how to apply the method in practice. In the firsts step, a set of applications is selected with known ODC faults (the "code" faults are considered) in order to build predictive models via empirical analysis. In the second step, techniques characterization is performed considering four widely used testing techniques, namely Functional testing, Statistical testing, Robustness testing, and Stress testing. The latter analysis is carried out with respect to the following real-scale software applications: the MySQL DBMS; the Apache Web server; the Samba networking applications suite.
Results obtained from our application of the method to the case studies, other than showing the feasibility of the method itself, can be used as starting base of knowledge for a company. This is then refined with time, by iteratively adding knowledge produced within the company.
In addition they also constitute a source of empirical data for researchers. Indeed, due to the lack of empirical analyses in real-world large software about testing techniques and fault types, as well as about fault types and software metrics, results may be useful to refine and/or validate theories on software testing and on fault prediction area.
The rest of the paper describes in detail the proposed method: the next subsection highlights the main contribution of this work. Section 2 discusses other works in the literature related the mentioned method's steps. Section 3 introduces the fault classification scheme adopted throughout the paper. Section 4 presents the method, defining the procedure to build predictive models for software characterization and to characterize testing techniques. A description of how we applied the method to create the models, based on the case study applications, is then presented in Section 5. Section 5.4 shows how we used the method result on one more simple case study, while Section 6 highlights the limitations of the approach, and Section 7 that concludes the paper.

1.1. Contribution

This Section highlights the contribution of our method as compared with the current state of the art in testing techniques selection methods.
Combining testing techniques has been clearly shown to be beneficial, both from theoretical analyses

(Littlewood et al., 2000), (Wagner, 2005) and from practical experiences (e.g., (Laitenberger et al.,2002), (Selby 1986), (Musuvathi and Engler, 2003)). Nonetheless, the problem of how to select and combine techniques within a V&V process received less attention in the past (Bertolino, 2004). Several studies have been made that compare techniques, but very few analyze the real applicability conditions of a technique and the relevant parameters involved in its application, so that choosing a technique remains hard (Vegas and Basili, 2005).
Some well-known methods conceived for a broader process analysis support also the V&V process improvement. For instance, in 1992 (Chillarege et al., 1992) proposed the mentioned orthogonal defect classification, ODC, which, more than a classification scheme, is a method for process improvement based on defect analysis. As for the V&V process, the information on triggers activating faults in the various phases is used as feedback to give some judgment on their effectiveness. Triggers are used to find discrepancies between the faults found during each phase and the ones expected to be found, hence to evaluate the test process.
Defect analysis is also the central concept of the Root Cause Analysis (RCA), whose feedback may be used for improving the V&V process. Indeed, RCA provides enough details on each defect, but it requires substantial investment and it is based on qualitative reasoning, not lending itself to measurement and quantitative analysis. ODC concept is, in fact, an improvement of such analysis, in that it tries to provide more quantitative information to support decisions. One more interesting approach using the defect features to provide feedback is proposed by (Damm et al., 2004), through the "Faults-Slip-Through" (FST) analysis. In such analysis, faults are classified according to whether they slipped through the phase where they should have been found; based on this, the phase that needs improvement in the process is identified. This is not easy to say, because one should identify in which phase a given fault should have been found, which may be very unclear. FST has been also used in conjunction with ODC (Damm and Lundberg, 2005), in order to give indications also on the activities that need improvement.

These methods are therefore not specifically conceived to select testing techniques, but to figure out which phase (and, to some extent, which activity) needs to be improved in a process, by using historical data on defects. Based on that, some works tried to improve the concept of defect analysis to drive also the technique selection, e.g., the works by (Barret et al., 1999), and by (Wojcicki and Strooper, 2007). Specifically, (Barret et al., 1999) first proposed to use mapping matrix to optimize the process by analyzing V&V techniques and their ability of finding types of defects. Starting from a set of modified ODC triggers, authors mapped testing method to defects found by each of the triggers, populating a matrix with qualitative labels (Low, Medium ,High), indicating that a testing method has been prone toward a specific defect trigger. This represents an instantiation of the ODC approach to a specific process. The work by (Wojcicki and Strooper, 2007) extends the work by (Barret et al., 1999), by defining a way to use the mapping matrix, enriched with cost information. They outline four steps concerned with the selection of cost-effectiveness information about testing methods toward defect types, with usage of the matrix to "maximize" completeness, and with the selection of techniques, among the one assuring completeness, that minimize effort; the final step is to update cost-effectiveness information.
These two papers describe a way to organize the information about defect types and testing techniques in a process, introducing possible improvements; however, they just outlines a flow of high-level steps and the underlying idea, but not much is said about how to practically implement these steps (e.g.: how to obtain and quantify the needed information about the testing techniques, what is the objective criteria to assign labels in the matrix, what are the alternatives): diving into each of those steps is the actual challenge.
One work closer to our view of technique selection is the one by (Vegas and Basili, 2005). It proposes a characterization schema to aid techniques selection, in order to provide developers with a catalogue containing information to select techniques. This intends to identify relevant information and, differently from the previously mentioned works, to provide a quantitative basis to select testing techniques. Their final result is a scheme for storing the information about technique to be used for future selection.

With respect to all these studies, the work presented in this paper contributes in several aspects. The main contributions are summarized in the following points.

- **Product *vs.* Process**. In all the described works, except the one by (Vegas and Basili, 2005), the decisions are based on defect types encountered during the process, and possibly on testing techniques performance applied in the process in past iterations, i.e., they are based entirely on the process knowledge. None of them includes the product features in the decision about which technique should be selected.
  One novelty of our work is that we characterize the defect types expected to be found *in the product under test*, to find the best combination of techniques. In this sense, the main contribution lies in the combination of a *product-oriented* approach, since it aims at exploiting the knowledge extractable from a specific product (defects expected *in the product*, not *in the process phase*), with a *process-oriented* approach, as the ODC one, since it uses historical in-process data to iteratively enrich the characterizations. Our aim is to relate the knowledge regarding testing techniques with the knowledge about the software to test, in order to address the differences that a technique may exhibit when applied to diverse software applications. Therefore, all the mentioned methods can be used in conjunction with our approach.
  The work by (Vegas and Basili, 2005) includes some product features (such as the size, programming language), correctly identifying them as factors to take into account in the selection, but: *i)* there is no specific procedure to include this quantitative information in a measurement-based selection process; their entire solution is not intended to give a selection procedure, but more oriented to provide a scheme for storing relevant information that may help in the selection: the selection, even though supported, remains a quite subjective judgement by the tester (also because many attributes are qualitative); *ii)* most importantly, they do not deal with defect type that can be found in that specific product (their information on defect type refers to the one detectable by the technique); *iii)* our work supports many software product's features compared to that scheme that can potentially affect the fault type introduced in the program; on the other hand, their work considers some "qualitative" information that can be included in our approach to complement it, and to conceive a combined approach accounting for both quantitative and qualitative procedures.

- **Characterizing of techniques**. The mentioned approaches characterize techniques with respect to some features of interest (e.g., defects they found in the past, the detection phase, the usability of the technique); (Vegas and Basili, 2005) extended these features, by combining subjective aspects of interest with consumer/producer's view. The main difference with respect to this point is that we decided to split the problem in two subproblems, which may individually be treated quantitatively by statistical analyses; predicting faults from metrics, and characterizing techniques with respect to faults. Indeed, the useful features considered by others can be integrated in our approach to further help the decision. Moreover, to carry out the techniques characterization we also foresee the possibility, as we experimented, to conduct controlled experiments and then using historical data to iteratively refine results; this allows us, differently from the described works, to give a preliminary characterization of new techniques never used in the process, and thus help introducing new techniques in the process by assessing their performance.

- **Fault Prediction**. To include the defect type expected in a product under test as an attribute for the selection, the step introduced by our method adopts *fault prediction* techniques to help characterizing the product to test. This is an important difference, since it represents the union of two areas that evolved independently so far: the *techniques characterization* and the *fault prediction* areas. This is the key to treat the problem through statistical methods, rather than through subjective interpretation of historical data. Note that this approach does not exclude other sources of information (as the one in the scheme by (Vegas and Basili, 2005)), but it provides a quantitative support (on product-based knowledge) to the tester's decision that he can for sure complement with other information.

- **Combination of Techniques**. The works by (Wojcicki and Strooper, 2007) and (Barret et al., 1999), as ours, include the possibility to combine techniques (statically, before the process beginning), unlike the contribution of (Vegas and Basili, 2005). This crucially affects the technique performance

((Littlewood et al., 2000)). As also argued by (Wojcicki and Strooper, 2007), Basili's characterization scheme is closer to the repositories idea rather than a process for technique selection.

- **Runtime Support**. We support the runtime selection of a technique, i.e., the ability to dynamically support the selection of the best technique accounting for a how the testing process is evolving. Our method, by monitoring the type of defects fixed along the process, is able to quantitatively suggest which is the best technique for that product at that specific time during the V&V process evolution. The best technique as scheduled through static information, maybe the best one at the beginning of the testing process; but as the testing proceeds, the technique may loose its detection power (since it exercise always the same portion of the code), and needs to be changed. Works on adaptive test case selection (e.g., (Cai et al., 2007)) do not use any type of information on fault type and on past techniques performance, but just adapt the selection of test cases based on the current detection rate.

Grounding on these features, applying this approach is expected to improve the ability in detecting software faults. The crucial assumption to obtain this improvement is the following: looking for the best combination of testing techniques for a software application depends upon: *i)* how much testers know about the effectiveness of techniques they can adopt, *ii)* what they know about the peculiarities of the software under test, and *iii)* how they can map the knowledge of techniques to the knowledge of that specific software. Consequently, the main contribution is to provide support to testing practitioners in addressing these issues, corresponding to "how to gain knowledge and how to exploit it". In Section 5.4 we show that exploiting a base of knowledge actually allows an improvement. The approach is compared against the pure Functional testing approach, which is the most adopted method in practice. In Section 5.5 and 6 we also discuss the main limitation of the approach, that is the "initialization" phase requiring an effort to build the required knowledge for the first time, mainly for companies without historical data available. The proposed strategy implies a "process" change to a certain degree, not merely a strategy for testing one product; hence, this incurs a start-up effort.

2. Related work

The previous Section presented the contribution with respect to the current state of the art in terms of testing techniques selection method. As further related work is worth to mention our previous work, the Ph.D Thesis in (Pietrantuono 2009); that work does not formalize the method for selecting techniques, but it reports preliminary materials on experiments on fault prediction (with a limited set of metrics) and on techniques performance.
This Section presents other works related to the two different areas which our method is based on: *i)* Fault prediction, and *ii)* Testing techniques characterization.

2.1. Fault Prediction

As first step of our method, we investigated *ODC fault types-metrics* relationship in software applications. In the past, several studies used software metrics to assess faults content (Fenton and Pfleeger, 1998). Statistical techniques such as regression and classification, are adopted in order to build predictive models, also known as fault-proneness models. Such models allow developers to focus their attention on fault-prone software modules. In (Chidamber and Kemerer, 1994) Object-oriented metrics were proposed as predictors of faults density; Subramanyam et *al.* (Subramanyam and Krishnan, 2003) presents a survey on eight empirical studies showing that OO metrics are significantly correlated with faults. Basili et *al.* (Basili et al., 1996) focused on validating OO metrics for fault prediction. Other studies using metrics investigated design metrics able to predict modules more prone to failures (Binkley and Schach, 1998; Ohlsson and Alberg, 1996). In (Gokhale and Lyu, 1997) authors used a set of 11 metrics and an approach based on regression trees to predict modules more prone to contain faults. In (Nagappan et al., 2006) authors mine metrics to predict the number of *post-release* faults in five large projects. For the statistical technique adopted, the latter is the work closest to our, since we use similar technique to reduce the initial set of metrics, complemented with another one.

Differently from these studies, the hypothesis that we investigated to build predictive models is that values of some metrics can be related to the *type* of faults present in the software, according to the ODC, other than the amount of faults (or the faulty modules).
A good survey on fault prediction is in the study by (Catal and Dirl, 2009), which reports the trends in selecting metrics, in selecting statistical method, and datasets across years.

2.2. Testing Techniques Characterization

As a result of applying the second step of the method, we conducted a comparison of techniques with respect to several factors. In the literature, several studies dealt with the analysis of software testing techniques. Much of the literature, since the eighties, is about theoretical comparison of test case selection criteria. One of the main issues addressed in these papers is the appropriateness of the adopted comparison relations: the first question authors attempted to answer is "how to correctly compare the effectiveness of testing strategies?". One of the first comparison relations was used, in 1982, by Rapps and Weyuker (Rapps and Weyuker, 1985), known as the "subsumption relation", where a criterion C_1 *subsumes* a criterion C_2 if for every program P, every test suite that satisfies C_1 also satisfies C_2.
They compared several data-flow and control-flow test case selection strategies by this relation. However, this criterion was recognized to be inadequate (many testing techniques were incomparable using subsumption) or even misleading, as pointed out in (Hamlet, 1989). Successively, the "power relation" was introduced by Gourlay (Gourlay, 1983), which defined a criterion C_1 to be at least as powerful as C_2 if, for every program P, when C_2 detects a failure in P, then so does C_1. Although this made a positive step, the incomparability problem was not solved. The introduction of the "BETTER relation" attempted to address the problems of both the previously introduced relations (Weyuker et al., 1991). Authors first defined the notion of a test case being *required* by a criterion C to test a program P, if every test set that satisfies C for that program must include that test case. Then, they defined the relation, stating that criterion C_1 is BETTER than criterion C_2 if for every program P, any failure-causing input *required* by C_2 is also required by C_1. Although this attempt, authors also stated that very few techniques are comparable with this relation, leaving the incomparability problem unsolved. In (Duran and Ntafos, 1984) authors addressed the problem by a different view, followed then by others (Hamlet and Taylor, 1990). They address the problem by using a probabilistic measure. Along this trend, authors in (Frankl and Weyuker, 1993) introduced the *properly covers* and *universally properly covers* relation, which uses the probability that a test suite exposes at least one fault. While with these theoretical relations, authors compared many testing criteria, the main problem of this approach is the practical applicability of these results in real development projects. Weyuker (Weyuker, 2008) surveys past work about testing techniques comparison: she lists a set of assumptions that should be satisfied in theoretical comparison studies, but that are not always verified in practice; the main issue is that the "idealized" versions of the compared testing techniques are never the actual versions used in practice, due to the numerous underlying assumptions.
On the other hand, there are very few empirical studies that compare the effectiveness of testing techniques in real-scale software systems. Many of them used generally small programs, specifically written for experimentation (Duran and Ntafos, 1984; Thevenod-Fosse et al., 1991; Frankl et al., 1997; Grindal et al., 2006); some examples of large projects are present in (Avritzer and Weyuker, 1996; Avritzer and Weyuker, 1999). The lack of such studies is mainly due to the difficulties and cost associated with this kind of analysis; also in (Weyuker, 2008) it is recognized that there is a profound need for empirical study in this field.
The study presented in this paper does not aim at empirically comparing techniques: however, results derived from our experimentation may contribute to enlarge the body of knowledge about testing strategy, since it is an experience with real-world large software projects where four state-of-the-practice techniques are empirically evaluated. Results of the existing empirical analyses may be compared, and also combined, with ours. However, the main difference is that this work focuses on presenting an approach whose ultimate aim is to practically support organizations in planning testing policies; the obtained empirical data are only the result of the application of the method.

3. Fault Classification

Fault types play a relevant role in this study, since they are of primary importance in both steps of the method. A software fault is a development fault that may cause a deviation from the expected (i.e., correct) program behaviour. To evaluate techniques with respect to fault types, a classification scheme is required. In this paper we adopt the well-known Orthogonal Defect Classification (ODC) (Chillarege et al., 1992) to distinguish fault types.
The classification comes from a preliminary collection of faults observed in the MVS (Multiple Virtual Storage) system, schematized in a level particularly close to the programming one. Since its introduction, the ODC has been adopted in several large projects (Bhandari et al., 1994; Bassin et al., 2001; Dalal et al., 1999) for process tracking and improvement. Faults, in this scheme, are classified into non-overlapping (orthogonal) classes, in which the choice is uniquely identified according to the semantic of their fix, i.e., a fault is characterized by the change in the code that is necessary to fix it. It was conceived to keep the classification process simple, with little room for confusion, and with few classes that are distinct and mutually exclusive. The choice of the ODC is ultimately due to its orthogonality feature, to its closeness to the actual mistakes made by programmers, and to its wide adoption in real large projects.
The ODC includes the following fault types:

- **Assignment**: values assigned incorrectly or not assigned.
- **Checking**: missing or incorrect validation of data or incorrect loop or conditional statements.
- **Interface**: errors in the interaction among components, modules, call statements, parameters list.
- **Algorithm**: incorrect or missing implementation that can be fixed by reimplementing an algorithm or data structures.
- **Function**: affects a sizable amount of code and refers to capability that is either implemented incorrectly or not.
- **Timing/Serialization**: missing or incorrect serialization of shared resources.
- **Build/Package/Merge**: Describe mistakes in library systems, management of changes, or version control.
- **Documentation**: This defect type affects both publication and maintenance notes, it has a significant meaning only in the early stages of software life cycle (specification and high level design).

The procedure presented in the following Sections aims at: *i)* constructing empirical models to relate *features* of the software to the ODC fault types in order to enable ODC faults prediction, and at *ii)* comparing testing techniques and characterizing their performance with respect to ODC faults.

4. Base of Knowledge Construction

4.1. Models Construction

*4.1.1. **Objectives***

In testing a software application, what is primarily needed to best select testing techniques is a way to characterize the specific software application(s) under test. "Characterize" in this context means extracting the features of the application that are mainly of interest to the tester, i.e., how many faults affect the program, and of what type.
This depends on many factors, e.g., how the software has been developed, what programming techniques have been adopted, how complex and large the code is. Since many of software features can be expressed by means of common software metrics, we construct predictive empirical models to estimate the number of ODC fault types starting from a set of software metrics. The conjecture is that metrics are able to capture (and to describe) those software features that mainly determine (or are related to) what types of faults affect a software application. If such a relation exists, testers estimate the number of faults of each type in the software. This will help in selecting testing techniques, once the second step of the method is completed.

4.1.2. ***Procedure***

In the following, the steps to build the models are outlined.

i. *Metrics Selection and Extraction*: a set of software metrics suitable for fault prediction is selected. Much literature has been produced to find a "best" set of metrics to predict faults (cf. with Section 2). In many cases, metrics provide good prediction results also across several different products; however, it is difficult to claim that a given set of metrics is general enough to be used even with very different products, as also discussed in (Nagappan et al., 2006). On the other hand, they are undoubtedly useful within an organization that iteratively collects in-process faults data. Hence, engineers have to select the set of metrics more suitable for their organization, also depending on the type of software they produce (e.g., Chidamber & Kemerer's – CK – metrics perform well for OO software (Chidamber and Kemerer, 1994), (Basili et al., 1996)). (Catal and Dirl, 2009) identifies several set of metrics, at several level of granularity (e.g., method-level, class-level, component-level, process-level). Although there are studies using every type of metrics, the most used and reliable ones are the method-level metrics, followed by the class-level ones – (Catal and Dirl, 2009) count more than 64% of the surveyed papers adopting method-level metrics. These express the complexity and the size of the product under analysis, the most common ones being the McCabe's cyclomatic complexity, the lines of code, the Halstead's metrics, and the object-oriented CK metrics.
Thus, a practical way to apply this step is to consider such metrics as a starting point (Table 1 provides a list, with also some important class-level metric), which are more guaranteed by past studies to give good results; other metrics specific to the company or to the product/process features may be then added to refine the prediction accuracy.

Table 1: Software metrics considered*

Metrics	**Description**	**Metrics**	**Description**
CountDeclClass	Number of Classes	MaxNesting	Maximum nesting level of control constructs
CountDeclFunction	Number of Function	CountPath	Number of unique paths through a body of code
CountLine	Number of all lines	SumCyclomatic	Sum of cyclomatic complexity
CountLineBlank	Number of blank lines	SumEssential	Sum of essential complexity
CountLineCode	Number of lines containing source code	AvgVolume	Average Halstead's Volume
CountLineComment	Number of lines containing comments	AvgLength	Average Halstead's Length
CountLineInactive	Number of lines inactive from the view of preprocessor	AvgVocabulary	Average Halstead's Vocabulary
CountStmtDecl	Number of declarative statements	AvgDifficulty	Average Halstead's Difficulty
CountStmtExe	Number of executable statements	AvgEffort	Average Halstead's Effort
RatioCommentToCode	Ratio of the number of code lines to the number of comments line	AvgBugs	Average Halstead's Bugs Delivered
Header File	Number of Header files	VVolume	Variance of Halstead's Volume
Code File	Number of Code files	VLength	Variance of Halstead's Length
CountLineCodeDecl	declarative source code declarative source code	VVocabulary	Variance of Halstead's Vocabulary
CountLineCodeExe	Number of lines containing executable source code	VDifficulty	Variance of Halstead's Difficulty
AvgCyclomatic	Average cyclomatic complexity	VEffort	Variance of Halstead's Effort
MaxCyclomatic	Maximum cyclomatic complexity	VBugs	Variance of Halstead's Bugs Delivered
*These metrics refer to our application of the method; thus, there may be some metrics that are specific to the context in which we applied the method, e.g., Header file, which may not have sense if referred to programs written in non-C languages.			

Metric values are then extracted from the selected applications, possibly by means of tools for static code analysis.
Note that over-descriptive metrics (it is common to have strongly correlated metrics having redundant information) can be reduced before building the prediction model (step *iv*) by one of the available techniques (such as in (Luo et al., 2010), (Nagappan et al., 2006), (Wang et al., 2011)).

In our application of the method we considered metrics of Table 1, opportunely reduced by the technique of the principal component analysis (Jolliffe 2002), described in the point *iv*.

ii. *Sample Selection*: to evaluate relations between metrics and ODC fault types, it is needed a set of software applications of which one can know the metrics value and the number of ODC fault types they contain. If there is no application with classified ODC faults available in the organization, they may be taken from the literature in order to build the initial base of knowledge, and then refined over time when the applications in the organization become available. Of course, the best set of metrics may also change with time when new samples are added, and predictive models are re-built.

iii. *Building Regression Models*: the successive step is the definition of statistical models to predict the presence of faults of each ODC type, by using software metrics as predictors. To this aim, fault prediction literature proposes several methods, the most used ones being statistical methods and machine learning approaches, such as decision/regression trees, (multiple) linear regression and logistic regression, Bayesan network, Naive Bayes, or a combination of them.

There are works which adopt binary classification for fault prediction: binary classification consists in learning a model from known data samples in order to classify an unknown sample as "fault-prone" or "fault-free" (samples being software modules).

These are not conceived to provide an estimate of the number of faults, unlike regression-based methods. In order to use the information on fault types proneness to select the techniques, any method could be used to choose depending on the features of data.

However, to make it easy the ability of switching techniques at runtime (as faults of any type are removed) the information on the number of fault per type could be more useful with respect to binary classification outcome.

Without excluding any method for fault prediction, we suggest methods providing the faults estimate as number of faults, such as multiple linear regression or regression trees.

The regression tree model is a goal-oriented technique, which attempts to predict numerical attributes (in this case the number of faults from metrics) by constructing a tree (similarly to decision trees), which is then used as a predicting device. The approach consists in recursively partitioning the predictor variable space (the metrics) into homogeneous regions such that within each region the distribution of the response variable conditional to the predictor variables is independent of the predictor variables (Gokhale and Lyu, 1997). At each step the construction algorithm search through all possible binary splits of all the predictor varaibles until the overall deviance is minimized.

The tree is then used for prediction, simply tracking down from the top node to the final leaf by comparing metric values with thresholds at each node.

Linear regression is a method that models the relationship between a dependent variable Y, independent variables X_i, $i = 1, \ldots, n$, and a random term ϵ. A linear regression model can be written as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots \beta_n X_n + \epsilon$, where β_0 is the intercept ("constant" term), the β_i are the coefficients of independent variables X_i, and n is the number of variables. The ϵ term represents the unpredicted or unexplained variation in the response variable; it is conventionally called the "error", regardless of whether it is really a measurement error or not, and it is assumed to be independent of the X_i variables. In our case, the X_i variables acting as predictors are the extracted software metrics, while Y represents the number of faults of a given type (since we consider five fault types, there will be five regression models).

One difficulty associated with the models construction is determined by the high inter-correlation among metrics: this causes the problem of *multicollinearity*. For instance, it is clear that a metric counting the lines of code can be strongly correlated with a metric counting the number of blank lines; this correlation leads to an inflated variance in the estimation of the dependent variables. To overcome this problem, two techniques are alternatively considered: the *Principal Component Analysis* (PCA) (Jolliffe 2002), or the *Stepwise regression* (Hocking 1976). The first one transforms original data into uncorrelated data (it removes the first-order correlation); it computes new variables, called Principal Components (PCs), which are linear combination of original variables, such that all principal components are uncorrelated. From these new variables, a subset able to explain the most of variance of original data is selected. Typically, a very small percentage of original variables (e.g., 10%) are able

to explain from 85% to 90% of the original variance. Each of the principal components is expressed as:

$$PC_i = \sum_{j=1}^{m} a_{ij} X_j \; with \, 1 \leq i \leq m \tag{1}$$

where X_js are the original variables, m is the number of variables and a_{ij}s are coefficients expressing the weights that the jth original variables has on the ith principal component ($-1 < a_{ij} < 1$, where values close to $|1|$ indicate that the jth variable, X_j, has high impact on the ith principal component PC_i). With this technique, once the m PCs are obtained, a subset of them containing as much variance as possible is selected. The chosen PCs will be the predictor variables of the regression model. The second technique, i.e., the stepwise regression, considers an initial partial set of variables and then attempts to add a new variable that improves the model (it performs some statistical tests to measure the improvement) or to eliminate a variable whose coefficient is not statistically significant; this leads to a minimum subset of variables that provide the best model (it yields a sub-optimal solution, since it is an heuristic technique).

While the first technique has the advantage of using (first-order) uncorrelated variables, a problem is that the principal components do not have a physical meaning; they are a combination of original variables. The stepwise regression, on the other hand, does not use totally uncorrelated variables, but it preserves the physical meaning of predictors. To have the best regression models, engineers may compare results from both techniques. Specifically, per each model, we propose to compare statistics resulting from the PCA-based model with the ones resulting from the stepwise-based model, and then to choose the best one in terms of *predictive power* and *significance*.

Thus, for each fault type (i.e., for each model) the steps are the following: *i)* from the original metrics, apply the PCA: a set of principal components (PCs) of the same size of the metric set is obtained; *ii)* from the obtained PCs, select the first k such that their variance is at least of 95%; *iii)* apply multiple linear regression using these PCs as predictors and the number of faults of the given type as response variable, obtaining the model M; *iv)* from the set of the original metrics, apply the stepwise regression (e.g., with a tool such as Matlab ©), obtaining another model M'; *v)* from both the resulting models, collect the following statistics: the significance of the hypothesis test of Fischer, known as *F-test*, (i.e., the *p-value*), which indicates how much confident the outcomes given by the model are; the value of the adjusted *coefficient of determination*, known as adjusted R^2: the R^2 value represents the explanatory power of the model (i.e., how well the dependent variables outcomes are likely to be predicted by the model); the *adjusted* R^2 value, that accounts for the degree of freedom of the independent variables and for the sample population, capturing the effect due to the introduction of a new variable in the model; *vi)* for comparing the two models, consider a model only if its confidence is greater than at least 95% (i.e., p-value less than 0.05 – the standard adopted values are 0.1, 0.05 or 0.01); if both the models satisfies this condition, consider the model with the highest adjusted R^2.

The output of this step is represented by the regression models, by which testers may get fault types estimate of the application to test. It is important to remark that, even though constructed models may yield accurate predictions, testers should not have the ambition to treat them as generalizable, since results are always tied to the specific applications chosen as samples, and to the context in which they are obtained (Basili et al., 1999). On the other hand, by systematically adopting the method within an organization, the number of samples would increase with time, obtaining iteratively better and better prediction models, and improving generalizability within the target context.

4.2. Testing Techniques Characterization

4.2.1. ***Objectives***

Planning testing activities requires knowing the features of the application to test, as well as the features of techniques that could be adopted, in order to judge if they will be more or less effective for the software being tested. With predictive models for software characterization being available by the first step of the method, the procedure described in this Section is intended to have a characterization of the testing techniques that can be adopted in a given organization.

The objective of this second step is to evaluate techniques in different settings and with respect to several factors, so as to have an estimate of their best applicability conditions.
This approach, based on running controlled experiment, allows for the seamless introduction of new technique, where all the other approaches for techniques selection consider only historical information to characterize the performance of a technique, making it hard to include new techniques in the process. Whenever the application of this approach is not possible (e.g., for the cost it entails –see Section 5.5 for a discussion on cost) historical information can also be used to characterize the performance of the technique with respect to fault types.

Running the experiment as outlined in this Section, the goal is to evaluate techniques with respect to: (i) *number of total faults detected*, (ii) *detection cost*, and (iii) *fault types they are more prone to detect*, we call it the *attitude towards each fault type*.
Evaluating the attitude towards fault type is the most relevant aspect for our purpose. Indeed, the number of faults detected per type is a finer-grained information than the total number of faults: knowing that one technique detects more faults than another may be not sufficient to choose the best technique, especially in the late phase of testing. A technique may be able to detect more faults of a type that is not (or no longer) present in the software. If the software, at any given testing time, is especially affected by faults of one type, testers may decide to apply a technique that is more prone towards that type, rather than a technique that detects, in absolute, more faults.
The evaluation of the number of total faults detected may help if testers decide to (or must) ignore the information on fault types (e.g., there is still not enough skill on ODC in the organization). Evaluating detection cost helps making decision considering the available budget for testing.
The procedure adopted defines how to plan and conduct a controlled experiment to characterize techniques with respect to the stated objectives. The experiment is expected to provide feedback on how each technique behaves in different environmental conditions. In particular, the mentioned response variables are evaluated with respect to the following factors: the *amount of faults* that is initially present in the software (i.e., if it is a heavily bugged software, or not), the *testing technique* adopted for the testing session, and the *software application type*. Different distributions of fault types initially present in the software (i.e., different percentages of each type of fault) are also considered.
In summary, by this step, testers should be able to answer the following questions about techniques to characterize in their organization:
Fault detection ability

1. Which of the compared techniques detects the greatest percentage of faults, regardless their type, with respect to those initially present?
2. Does the percentage of detected faults depend on the initial amount of faults in the program?
3. Does the percentage of detected faults depend on the chosen software type?

Fault detection cost

1. Which of the techniques detects the greatest number of faults, standing the same testing time (i.e., the greatest *#faults/hour ratio* – assuming the personnel effort as measure of cost, i.e., "person hour")?
2. Does the *#faults/hour* ratio (namely, the *detection rate*) depend on the initial amount of faults?
3. Does *detection rate* depend on the chosen software type?

Attitude towards fault types

1. Given a type of fault, which of the available techniques detects the greatest percentage of faults of that type (i.e., are faults of a given type more easily detectable with a particular technique than with another one) ?
2. Is a given technique more prone to detect some types of faults than others?

The following Sections describe the steps to undertake in order to answer these questions.

4.2.2. ***Procedure***

The application of the characterization procedure has to be carried out via a controlled experiment aiming at evaluating techniques behavior in different conditions. This, within an organization, leads to constructing a base of knowledge regarding techniques performance in the specific context; it is carried out once, with an extra-effort to perform the experiment. The procedure consists of the following steps:

i. *Selecting Testing Techniques*. The firs step is to choose the testing techniques to characterize. This depends on the organization. Among the plethora of testing techniques, a company can start by selecting the techniques already used in its context. Note that this choice does not exclude the possibility to introduce new techniques in the process; rather, it is true the opposite: one of the main novelty of our approach is that it is the first one allowing for a characterization of new techniques never used within the organization, and this can be done by including in this step a new technique in the experimental planning.

ii. *Choosing the target applications*. The applications used for the analysis is also dependent on the organization. They should be representative of the target context, where the testing techniques will be applied. If the techniques are to be applied to different kinds of products (different in terms of applicative target, code size and complexity, development styles) within the same organization, then it is preferable to choose samples in order that each line of product is adequately represented.

iii. *Experimental planning*. An experimental plan has to be derived, by adopting the Design of Experiments (DoE) methodology. The DoE (Juristo and Moreno, 2001) is a systematic approach to the investigation of a system or a process. A series of measurement treatments are designed, in which planned changes are made to one or more system (or process) *input factors*. The effect of these changes on one or more *response variables* is then assessed. The DoE aims at planning a minimal list of treatments in order to get statistically significant answers.

The first step in planning such treatments is to formulate a clear statement of the objectives of the investigation. Then, the next step is concerned with the choice of *response variables*, which identify what variables have to be measured during the experiment. Since the goal is to figure out if certain factors affect, and how, response variables, the subsequent step regards the choice of *factors* of interest for the experiment's objective.

A particular value of a factor is usually called *level*. A factor is said to be *controllable* if its level can be set by the experimenter, whereas the levels of an *uncontrollable* or observable factor cannot be set, but only observed.

The identification of response variables, factors and levels is followed by the definition of a list of treatments. Each treatment is obtained by assigning a level to each controllable factor.

In this step, response variables, factors, and levels relevant for the study are defined. Factors should include at least the testing techniques to characterize, the number of faults initially present in the application, and the software type. For each of these factors, levels should be established. For instance: levels for the testing techniques factor are the type of technique. Levels for the initial number of faults can be low, medium, and high; the number of faults for the medium level may be determined either by historical information or estimated by fault prediction (i.e., predicting expected faults from metrics of the application under test), and then varied of a percentage of $k\%$ to determine the low and high level as $medium + k\%$, with the goal of evaluating the impact of their variation on response variables. The type of initial faults has also to be set in this phase, i.e., the fault types distribution. Percentage for each type may be determined similarly, i.e., either historically or by prediction.

Response variables should include at least indices mentioned above regarding: the attitude toward fault types (e.g., the number or the percentage of faults detected per type), the cost (e.g., the time or the fault detection rate), and the absolute fault detection ability (e.g., the total number and percentage of faults detected).

Finally, given these factors and response variables, the list of treatments to perform for meeting the defined objectives is produced via the support of DoE aiding tools.

To give an idea of how many treatments are likely to be produced by a particular design, consider the following two examples. If practitioners have to characterize three testing techniques, considering

three type of applications, and two levels of initial fault content (low and high), the minimal number of treatments to get some statistically significant answers, as suggested by a statistical tool amounts to 6. Similarly, in a scenario with four techniques, on four applications, three levels of initial fault content, the number of treatments is at least: 9. Minimal here means that the effect of interactions among factors are neglected (i.e., a fractional design is adopted, cf with Section 5).

iv. *Injecting faults*. In order to compare techniques with respect to fault types and faults number, the amount and types of faults present for each treatment need to be injected. This implies performing a fault injection campaign, where, per each treatment, faults of each type are deliberately inserted into the software. Before running the treatment, the application under test is therefore prepared by inserting faults of different types according to what prescribed by the experimental plan.

v. *Running treatments*. After fault injection, the treatment is carried out by applying the testing technique to the target application as foreseen by the plan (e.g., Functional testing on Application 1). A treatment is a testing session: to apply the treatment, test cases are first generated, according to the technique foreseen by the plan, and then they are run.
Each treatment tests the effectiveness of one technique applied to a software application with a given number of faults, and a given combination of fault types.
Note that depending on the adopted technique, test cases writing and/or execution can be either manual or automated to some extent. For comparing techniques, it is important to have the same conditions; hence, either including the test case generation for all the technique, or focusing only on test case execution for all the techniques.

vi. *Recording data*. During each testing session the following data are recorded: the number of detected faults (per each type), the time when they are detected, and the identifiers of test cases exposing them.

vii. *Analyzing results*. Collected data are finally analyzed statistically in order to outline how factors affect the response variables. To this aim, data analysis techniques need to be adopted in this phase, depending on the type of data obtained from the characterization experiment. Examples range from basic techniques, such as analysis of average differences, hypothesis tests, parametric and non-parametric statistics, to more complex (multivariate) analyses, such as (multiple) linear and/or logistic regression, factorial analysis, (multivariate) analysis of variance, cluster analysis.
For instance, a simple and effective way to proceed in order to evaluate the variation of the output variables in response to the variations of the analyzed factors is to conduct an Analysis of Variance (ANOVA) to figure out if factors and/or their interaction impact the response variables significantly; then, to evaluate the impact, if any, post-hoc analysis (e.g., *HSD Tukey* test) on the differences among mean values could be performed. Note that such an approach should be applied only if data meet the assumption imposed by ANOVA, and this needs be verified by proper tests, such as the one of the normality and homoscedasticity (Montgomery, 2001).
After numerical interpretation of achieved results by statistical analyses, the physical interpretation takes place, by relating data to the questioned objectives, obtaining the desired characterization, in terms of attitude toward fault types, cost, and type-unrelated detection ability.

By following the procedures above, testers obtain models to predict ODC fault types from software metrics, and then a characterization of techniques with respect to the same fault types (and to other variables), to be used for techniques selection. The following Sections detail how we applied the two steps to build an actual base of knowledge.

5. The Method in Practice

5.1. Premises

To apply the method, the first step requires choosing, as samples, a set of software applications with known ODC faults. Applications are better selected if they belong to the reference context, e.g., within a company, where the method is adopted. This requires the availability of bug repositories for sample applications, by which an ODC-based classification can be carried out. In our implementation of the method, we selected a set of applications with known ODC faults taken from the literature, specifically from one of

the most important field studies on ODC faults on open source programs (Duraes and Madeira, 2006).This choice does not refer to any specific context; on the other hand, it provides practitioners that want to apply the method with an initial set of software applications to be used as sample for models construction. If applications in the organization are available, we advise adopting the method described above, considering the following text as an example on how to implement the method. On the contrary, if such applications are not available, we advise adopting our results as starting base of knowledge, to be then refined with future developed applications within the company.
Similarly, characterization of techniques requires software application that should be representative of the target context, where the testing techniques will be applied. If the techniques are to be applied to different products (in terms of applicative target, code size and complexity, development styles) within the same organization, then it is preferable to choose samples in order that each kind of application is adequately represented. For the scope of this study, we select widely-used and real-scale software applications (and the most common techniques as well) to validate the characterization step. In this way, results may be taken as an initial base of knowledge that an organization may wish to adopt, as in the former step. Again, this would be then refined with time by extending the experimental plan with additional treatments.
The goal of the following Sections is not only to show the applicability of the proposed method on case-studies; results of the conducted application constitute a source of empirical data on testing techniques performance on realistic and large applications, and on software metrics-fault content relationship – thus the choice of adopting applications unrelated to a specific context is to capture the interest of more people in software engineering community, and to share empirical data with a wider audience. Finally, the choice of adopting open-source applications with publicly available bug reports gives also the possibility to researcher to repeat the process of building the base of knowledge, and validate our results.

5.2. Models Construction

We applied the procedure steps to obtain five predictive models, one per each ODC fault type. Note that we consider five out of the eight ODC classes; the applications on which we apply our procedure are provided with bug reports classified according to the ODC but considering only those programming-related five types, since the other three types are typically either negligibly few, or do not affect directly the delivered service – e.g., wrong/missing documentation (Duraes and Madeira, 2006). As for software metrics to consider, Table 2 reports the 32 metrics that we have selected, being them the most used and reliable ones (cf. with Section 4.1.2). To recall, their meaning, the first subset of metrics is related to the "size" of the program in terms of lines of code (e.g., total number of lines, and number of lines containing comments or declarations) and files. These metrics provide a rough estimate of software complexity, and have been used as simple predictors of fault-prone modules (Ostrand et al., 2005). Other metrics are introduced in order to improve fault prediction models, the most notable being the McCabe's cyclomatic complexity and the Halstead's metrics. These metrics are based on the number of paths in the code and the number of operands and operators, respectively.
Product metrics are typically correlated with each other. For instance, *CountLine* and *CountLineBlank* are likely to be strongly correlated with each other, and this leads to an inflated variance in the estimation of the dependent variables, thus to the mentioned problem of multicollinearity.
To overcome this problem, we adopted the approach similar to the one described in (Nagappan et al., 2006) (i.e., by stepwise regression and PCA, as described in Section 4.1.2), in order to reduce the number of predictors for the subsequent regression models.
The chosen software applications are made of a set of eleven applications with known ODC fault types: they were the subjects of the study in (Duraes and Madeira, 2006), where a classification from bug reports was carried out by authors for *fault injection* -related purposes. Applications are shown in Table 3.

Metrics for these applications have been extracted by using a tool for static code analysis[1].
Once we have obtained predictors (i.e., metric values) and response variables (i.e., ODC fault types), we

[1]We used the source code analysis & metrics tool Understand 2.0, available at: *http://www.scitools.com/products/understand/*

Table 2: Software Metrics considered

Type	Metrics	Description
Program size	*CountDeclClass, CountDeclFunction, CountLine CountLineBlank, CountLineCode , CountLineComment CountLineInactive, CountStmtDecl, CountStmtExe RatioCommentToCode, C Header File, C Code File CountLineCodeDecl, CountLineCodeExe*	Metrics related to the amount of lines of code, comments, declarations, statements, and files
Program complexity	*Average Complexity, Maximum Cycl. Complexity, Maximum Nesting level, CountPath, Sum of Cycl. Complexity Sum of Essential Complexity*	Metrics related to the CFG of functions and methods
(Software Science) Halstead's Metrics	*Mean and Variance of: Volume, Length, Vocabulary Difficulty, Effort, Bugs Delivered*	Metrics based on operands and operators

Table 3: Faults Content per Type in the chosen applications

Software	**Fault Types**					Description
	Assig.	*Check.*	*Interf.*	*Algor.*	*Func.*	
Linux	21	24	5	31	12	Linux Kernel 2.0.39 and 2.2.22
Bash	2	0	0	0	0	Bourne Again Shell
Joe	20	35	11	12	0	Tex Editor
Pdftohtml	11	1	0	8	0	Pdf to html converter
Firebird	1	1	0	0	0	Relational DBMS
ScummVM	18	6	3	42	5	Interpreter for adventure engines
ZSNES	2	0	0	1	0	Emulator for x86
Vim	53	56	16	111	13	Linux editor
FreeCvi	6	7	4	28	8	Strategy game
MinGW	6	23	3	28	0	Minimalist GNU for Windows
CDEX	2	2	6	1	0	CD digital audio data extractor

have built the regression models. The first step to build regression model is the normalization of predictor variables due to their different orders of magnitude. Metrics have been normalized by the following method:

$$x'_i = \frac{x_i - min_i\{x_i\}}{max_i\{x_i\} - min_i\{x_i\}} \tag{2}$$

where x'_i is the normalized value of x_i, comprised between 0 and 1. Among the methods listed in Section 4.1.2 for fault prediction, we choose methods providing the number of faults per type as output, in order to ease the subsequent step of techniques selection according to fault type. We preliminarily evaluated regression trees obtaining bad performance. Such a method would require more samples for having acceptable performance. Thus, we switched to multiple linear regression, with the stepwise method and with the PCA compared as suggested in Section 4.1.2.

For the multiple linear regression with preliminary PCA, we adopted the same method of (Nagappan et al., 2006). In order to compare the models obtained by the PCA and by the stepwise regression algorithm, we have first carried out the PC transformation, obtaining 32 PCs. PCA-based regression models have been then constructed with a number of PCs, as independent variables, able to explain at least the 95% of the original data. Stepwise-based models have been obtained by the MATLAB® algorithm implementation.

Table 4 reports the statistics obtained for each of the five models obtained with both methods. In the second column the technique that has been tried for building the model is shown; the one selected is in boldface. The third column reports the number of variables used as predictors (for PCA, the percentage of explained variance, at least 95%, is also reported). The fourth column shows the obtained models: in the case of PCA, the independent variables PC_i are obtained as in the Equation 1; whereas, for stepwise regression, the algorithm determined the variable(s) X_j to consider. In the latter case, variables correspond to metrics: for the *Assignment* model it is the *Average Halsteads Effort* (X_{26}); for the *Checking* model as well as for the *Interface* model, they turned out to be the *RatioCommentToCode* (X_{10}), that is the ratio of the code lines to the number of comment lines, and the variance of the Halstead's *Difficulty* metric (X_{31}); for the *Algorithm* model they are the *RatioCommentToCode* (X_{10}) and *Variance of Halstead's Volume* (X_{28}) for the *Function* model, they are again the *RatioCommentToCode* (X_{10}), *MaxNesting* (X_{16}), i.e., the maximum nesting level of control constructs, and *SumComplexity* (X_{18}), i.e., the sum of cyclomatic complexity. For

Table 4: Regression Models and their Predictive Power. Boldface indicates the model selected. X_is are metrics selected by the stepwise regression: X_{26}= *Average Halsteads Effort*; X_{10}=*RatioCommentToCode*; X_{31}= *Variance of Halsteads Difficulty*; X_{28}= *Variance of Halstead's Volume*; X_{16}= *MaxNesting*; X_{18}= *SumComplexity*. PC_is are principal components computed by the PCA.

ODC Type	**Technique**	**# of Variables**	**Model**	R^2	**Adjusted R^2**	**F-test, p-value**
Assignment	Stepwise	1	$Y = 47.4898 * X_{26} + 4.5815$	0.769	0.7433	29.9651 p = 0.0004
Assignment	**PCA**	4 (95.38 %)	$Y = 3.0288 * PC_1 - 12.6068 * PC_2 + +5.5420 * PC_3 - 14.8154 * PC_4 + 12.9090$	0.8704	0.7841	10.079, p = 0.00785
Checking	**Stepwise**	2	$Y = 45.8743 * X_{10} + 55.2594 * X_{31} - 18.35$	0.7546	0.6933	12.3057 p = 0.0036
Checking	PCA	8 (99.63 %)	$Y = 0.7721 * PC_1 - 13.9042 * PC_2 - 1.7838 * PC_3 - 15,7697 * PC_4 - 14.8209 * PC_5 - 29.7702 * PC_6 - 25.7013 * PC_7 - 62.7401 * PC_8 + 14.0909$	0.97917	0.89589	11.7567 p = 0.08072
Interface	**Stepwise**	2	$Y = 10.4686 * X_{10} + 16.195 * X_{31} - 4.3205$	0.79609	0.74512	15.6168, p = 0.001728
Interface	PCA	4 (95.38 %)	$Y = 0.4579 * PC_1 - 3.920 * PC_2 + +0.9752 * PC_3 - 2.544 * PC_4 + 4.3636$	0.5802	0.3003	2.0732 p =0.2027
Algorithm	Stepwise	2	$Y = 42.91 * X_{10} + 107.84 * X_{28} - 10.3804$	0.83	0.7959	20.498 p = 0.0007
Algorithm	**PCA**	4 (95.38 %)	$Y = 8.5038 * PC_1 - 22.9087 * PC_2 + +15.7448 * PC_3 - 38.8943 * PC_4 + 23.8182$	0.88604	0.81007	11.66297, p = 0.00541
Function	**Stepwise**	3	$Y = 11.5111 * X_{10} - 7.3998 * X_{16} + +15.9779 * X_{18} - 2.0038$	0.86888	0.81268	15.46215, p = 0.00181
Function	PCA	6 (98.40 %)	$Y = 0.8987 * PC_1 - 1.6490 * PC_2 + 2.053 * PC_3 - 8.0278 * PC_4 - 0.89921 * PC_5 - 12.1691 * PC_6 + 3.4545$	0.8849	0.7123	5.1269 p =0.0677

PCA-based models, it is possible to intuitively figure out what are the original metrics most affecting the result, by analyzing the composition of each PC included in the model as predictor: the coefficient value $a_{i,j}$ represents in fact how much a given metric j contributes to the i-th PC.

Typically, what describes the goodness of linear multiple regression models, and that has been considered for selecting between the two techniques, is: *i)* the significance of the hypothesis test of Fischer, known as *F-test*, (i.e., the *p-value*), which indicates how much confident the outcomes given by the model are, and *ii)* the value of the *coefficient of determination*, known as R^2, which represents the explanatory power of the model (i.e., how well the dependent variables outcomes are likely to be predicted by the model). These values are reported in the last columns of Table 4.

In particular, the fourth column reports the R^2 value, which is computed as the ratio of the regression sum of squares to the total sum of squares. A larger value indicates a better predictive power, since it means that more variability is explained by the model with respect to the total variability.

Models based on PCA method shows higher R^2 values than stepwise-based models. The fifth column reports the *adjusted* R^2 value, that accounts for the degree of freedom of the independent variables and for the sample population. It captures the effect due to the introduction of a new variable in the model, measuring if the new variable adds a sufficient explanatory contribution to counterbalance the loss of one degree of freedom. In other words, it is a measure of the robustness of the model. In this case, PCA-based models have not always higher values: indeed, the greater number of variables implies a bigger difference between the R^2 and the *adjusted* R^2 values. Thus, considering the *adjusted* R^2 instead of the R^2 index, the predictive powers of models are quite similar among each other. Then, in the sixth column, the *F-test* value is reported; it tests the null hypothesis that all regression coefficients are zero at a given significance level: if a model obtained with either PCA or stepwise is not significant at least at $\alpha = 0.05$ (i.e., at 95 % confidence level), we discard it and evaluate the other technique. Stepwise-based models show very high levels of confidence (more than 99.7%). Also the models for the *Assignment* and for the *Algorithm* fault type, obtained via PCA, have a high confidence (greater than 99%). In the described case, models show a satisfactory accuracy; however, testers adopting the method must not not have the ambition to treat them as generalizable, and should consider results as tied to the specific applications chosen as samples, and to the context in which they are obtained (Basili et al., 1999).

5.3. Techniques Characterization

5.3.1. Testing Techniques Selection

We applied the second step of the procedure by characterizing the following testing techniques: *Functional Testing*, *Statistical Testing*, *Robustness Testing* and *Stress Testing*. In general, these techniques are conceived to pursue different goals, and their application aims at improving some specific aspect of the software being developed. In the context of this work, the main goal is to characterize their behavior with regard to the type of faults they are more prone to detect, and to figure out how their overall performance varies depending on different conditions. It is worth noting that the chosen techniques are intended for the *system testing* stage, hence after unit and integration testing. To have a set of techniques able to cover different types of faults, we selected two techniques, namely, Functional and Statistical testing, for verifying if the system does what is required to do (i.e., "demonstrative testing"), and two techniques, namely robustness and stress, for verifying that the system does not do what is not expected (i.e., "destructive testing"). For system test stage, these are the most used ones.

A brief description of the selected techniques follows.

Functional Testing

In *Functional testing*, engineers derive test cases from specifications and test the system against what it is required to do. It is often referred to as "specification-based testing" or "black-box testing" (since there is no view of the code), and it is described in any books on software testing (e.g., (Beizer, 1995; Pezze and Young, 2007; Lewis, 2008)). Functional testing is typically the base-line technique for designing test cases. There are various criteria to define functional tests. Starting from the input domain of the program, test cases can be derived randomly (i.e., random testing (Duran and Ntafos, 1984)) through a brute force generation directly from the specification, or, most often, systematically, i.e., by formalizing the test case derivation process into elementary steps (Pezze and Young, 2007). In the latter case, given the unmanageable number of input combinations, the way by which input values are combined determines the strategy: the most common way is by using "the partition principle", i.e., by separating the input space in k subdomains (i.e., classes), and taking input values from them. In partition testing, the elements of a subdomain are supposed to be tightly related to each other: therefore, inputs from a subdomain are selected randomly according to a uniform distribution (Weyuker and Jeng, 1991). Partition testing is considered in this work as Functional testing strategy.

Statistical Testing

The goal of *Statistical Testing* is to improve reliability of the software, intended as the probability that the application does not fail during its mission time. Similarly to Functional testing, Statistical testing does not require any knowledge of the code. The term *Statistical Testing* is used differently in the literature, depending on how the distribution is chosen, e.g., to satisfy an adequacy criterion expressed in terms of functional or structural properties (Poulding and Clark, 2010), or selecting test cases by sampling from a distribution over the input domain reflecting the expected operational profile (Whittaker and Thomason, 1994; Kallepalli and Tian, 2001; Thevenod-Fosse, 1991; Thevenod-Fosse and Waeselync, 1991). (Pezze and Young, 2007), as well as (Frankl et al., 1998), refer to the latter acceptation as *operational profile -based testing* (OP-based testing), or simply *operational testing*; in this work, we also adhere to this terminology in order to avoid ambiguity. OP-based testing is based on the idea that some functionalities will be more exercised than others, and thus deserve more testing effort. The operational profile is built by assigning probabilities of occurrence to functionalities, or set of functionalities. This can be also supported by analytical models, such as Markov chains, as in (Whittaker and Thomason, 1994), (Trammell, 1995); but basically the accuracy of the operational profile depends on the accuracy of the information on how the system will be used (e.g., information derived from interviews with domain experts, from historical field data, from documentation, and/or from the literature data) (Pietrantuono et al., 2010).

Robustness Testing

Robustness testing aims at evaluating the software application's reaction to exceptional and unforeseen inputs (Kropp et al., 1998). Koopman et *al.* (Koopman and DeVale, 2000) describe one of the most successful robustness testing techniques, and the corresponding tool: *Ballista*. This technique adopts a data-type based fault model to generate invalid inputs, and generates test cases by combining invalid param-

eter values of invoked function's data type. Similar approaches to generate invalid inputs are adopted in (Kanoun et al., 2005); in this case, valid inputs are intercepted and replaced with invalid ones, by using a data-type based fault model, as well as by fuzzing and bit-flips. Tools that use parameters corruption are RIDDLE (Arlat et al., 2002), and MAFALDA (Ghosh et al., 1998). In the context of this work, test cases are obtained by combining invalid input combinations based on data type, which is the most common way to perform robustness testing, adopting the same technique as in Ballista (Koopman and DeVale, 2000).

Stress Testing

Finally, Stress testing is a technique to evaluate the application's ability to react to heavy loads (Lewis, 2008); unlike robustness testing, stress testing does not use "exceptional inputs"; it uses normal input values, with excessive load. The goal is to observe the performance of the system under excessive loads, and to see if there are bugs in the management of these situations. It is particularly useful for long-running, mission and business critical applications. Test cases are generated by considering the functionalities of the application and providing high values for inputs representing the load for that specific application (cf. with Section 5.3.4).

5.3.2. Case Study Applications

As anticipated, we choose to adopt widely-used and well-known software applications. Thus, criteria adopted to select applications are the following: they should have *i)* a different mission (which implies different requirements, design choices, development styles), *ii)* a significant and realistic size (since we expect that the techniques that we compare do not show significant and realistic differences if applied to small systems), *iii)* a considerable spreading in real-world scenarios.

Based on these criteria, we have selected the following software applications: *i)* a popular database management systems (DBMS), namely MySQL; *ii)* a well-known Web server, *Apache*, and *iii)* a networking application for Linux/Windows interoperability, that is *Samba*[2]. All these applications belong to open projects, and are provided with the source code.

The chosen applications are widely used in real-world scenarios, including business- and safety-critical contexts. MySQL is one of the most used DBMSs, accounting for about 40% of installations among IT organizations[3]. Apache is the most popular web server on the Internet, installed on more than 65% of machines running websites in the last two years[4]. Samba is the standard Windows interoperability suite of programs for Linux and Unix, that runs on most Unix and Unix-like systems, such as GNU/Linux, Solaris, AIX and the BSD variants, including Apple's Mac OS X Server, and is standard on nearly all distributions of Linux.

Table 5 reports the applications' characteristics. MySQL (v. 5.1.36) is made up of more than 330K Lines of Code (LoC) distributed among 216 files and a little over than 8K functions. Apache 2 accounts for more than 100K LoC distributed among 218 files and nearly 1800 functions. Samba (v. 4.0.0) enumerates approximately 225K LoC, distributed in 621 files, and 5160 functions.

As may be noted, Apache has less lines of code than the other two, but it has the highest average cyclomatic complexity; the number of files of MySQL is comparable with Apache (216 *vs.* 218), indicating that MySQL files are in the average bigger in terms of LoC, with many more functions per code file (a ratio of 61.27 *functions/code file vs.* 12.38 for Apache and 11.14 for Samba). Although MySQL is the biggest software, in terms of LoC, it has the lowest cyclomatic complexity and the lowest *LoC/functions* ratio (about 42 LoC per function, *vs.* 61 LoC per function for Apache and 44 for Samba), indicating a different programming style, especially with respect to Apache. MySQL programmers use many small functions, in relatively few files (obtaining also a small cyclomatic complexity); Samba programmers use functions of approximately the same size of MySQL (in terms of LoC per Function), but distributed in much more files; whereas Apache programmers put more code into a function, with higher cyclomatic complexity, and use relatively few files. These differences may have an effect on performances of techniques in terms of detection effectiveness. Thus, we consider software applications as a controllable factor of the experimental study, whose effect on response

[2]In particular, the adopted applications have been: MySQL 5.1.36, available at http://www.mysql.com/, Apache 2, available at: http://www.apache.org/ , and Samba 4.0.0, available at http://www.samba.org/

[3]MySQL Market Share, http://www.mysql.com/why- mysql/marketshare/, accessed on March 2011

[4]http://news.netcraft.com/archives/2010/06/16/june-2010-web-server-survey.html, accessed on June 2011

variables has to be assessed.

Table 5: Features of the Case-study Applications

	LoC	Code Files	Header Files	Functions	Avg Cyclomatic Complexity
Apache	104,403	139	72	1722	6.25
MySQL	337,752	132	84	8088	3.89
Samba	225,587	463	158	5160	4.97

5.3.3. Experiment Design

In the DoE approach, the number of treatments required to estimate the effect of factors and of all their interactions on response variables (namely a *full factorial design*) is determined by the number of controllable factors and the number of levels assigned to them (e.g., given k factors and l levels, the number of treatments is $n = l^k$). Typically, a full factorial design requires a large number of treatments; hence designers choose to reduce this number by ignoring the analysis of interactions among factors (e.g., three-way interactions) and by adopting optimization algorithms, as we did in this work. Design reduced in such a way is referred to as *fractional factorial design*. Reducing a design by still having statistically significant answers is an optimization problem whose solutions are found heuristically by adopting some optimality criteria. In our case, the D-Optimality criterion has been chosen, since it is particularly useful when standard fractional factorial designs would require too many runs for the amount of resources available. The optimality criterion used in D-Optimal designs is the maximization of the determinant of the information matrix, which results in minimizing the generalized variance of the parameter estimates (Montgomery, 2001). Given the available resources (16 treatments), we heuristically found by a support tool[5] the list of treatments reported in Table 8.

This plan ignores the three-way interactions, i.e., the effect of the interactions among more three factors on the response variable. Even though the plan could be obtained manually (Montgomery, 2001), practitioners are advised to adopt an automatic tool for generating the test plan, and then to follow the suggested number of treatments to have significant results.

In our plan, we considered three categorical factors. They are: the testing technique, the number of faults initially present in the software, and the software type. In each treatment, the initial conditions of the distribution of fault types are also set (by fault injection), i.e., the percentages of fault of a given type initially present in the application. Table 6 shows the mentioned factors along with levels that they can assume. Table 7 reports values of fault types distribution. Factors can vary among the following levels:

- Testing technique can assume four distinct levels, i.e., Functional, Statistical, Stress, and Robustness testing;
- Software type can assume three levels: MySQL, Apache, Samba;
- Initial number of faults can assume three levels: low, medium and high. The number of faults for the medium level for the sample software applications has been determined by using a linear regression model predicting the number of faults from software metrics, with the goal of injecting realistic quantities of faults. The model has been built by using the 11 sample software programs and the *size/complexity* metrics also adopted in the first step of the method (namely, the programs in Table 3, and the 20 metrics reported in the first 20 rows of Table 1). Due to the correlation among metrics, the *stepwise* technique (Hocking 1976) (specifically, the MATLAB® implementation) has been adopted in order to select the most relevant (and less correlated) subset of metrics for predicting the number

[5]The adopted tool is *JMP®*, which is statistical software for data analysis, DOE, and Six Sigma, from SAS, available at: *http://www.jmp.com/*

of faults. The resulting model has the following expression: $Y = 0.04601 * X_1 + 23.17263$, where X_1, which is the number of C Header files, turned out to be most relevant metric, sufficient to provide the best estimates without any additional metrics. The model is characterized by an R^2 value of 0.8389, meaning that it explains the 83% of the variability of original data, and a F-value of 7.81007 (i.e., its significance level is $p\text{-}value = 0.01471 < \alpha = 0.05$). An estimate of the number of faults has been obtained by extracting metrics for MySQL, Apache, and Samba[6], and by using them in the model as independent variables. The low and high levels have been derived by varying the medium value of +/- 65 %.

The distribution of fault types initially present in a software application vary as described in Table 7. These values have been determined based on typical faults content of each ODC type observed in real applications in several past studies (Duraes and Madeira, 2006; Christmansson and Chillarege, 1996). Each quantity is allowed to vary between +/- 10% of the estimated value. For instance, since the average value of *Assignment* fault type observed in real applications is 21%, we have considered 11% and 31% as the low and high level of that factor. Note that, by considering past empirical studies, we are setting realistic combinations of initial faults distribution, instead of arbitrary, unlikely, and indefinitely numerous combinations.

Table 6: Factors and their levels. Legend: F = Functional Testing, ST = Statistical Testing, SS = Stress Testing, R = Robustness Testing

Levels					
Categorical Variables	**Technique**	*F*	*ST*	*SS*	*R*
	Faults #	*Low*	*Medium*		*High*
	Software	*MySQL*	*Apache*		*Samba*

Table 7: Initial Fault Type Distribution

Initial faults content of type:		*Range*	
		From	To
Continuous Variables	Assignment (21.4%)	11%	31%
	Checking (25%)	15%	35%
	Interface (7.3%)	0%	17%
	Algorithm (40.1%)	30%	50%
	Function (6.1%)	0%	16%

As response variables, the following are considered: *the percentage of faults detected*, *the fault detection rate*, *the percentage of faults detected per each fault type*. The first one captures the ability of detecting faults; the total number of faults detected is recorded, but the percentage with respect to those initially present is considered for the analysis, so as to eliminate the bias due to the initial number of faults injected. The second one captures the cost of the testing session, considering the number of faults detected per hour by the selected technique. The third one is similar to the first one, but referring to the ability of detecting a specific type of fault (thus, there is one response variable per fault type, i.e., 5).
Table 8 summarizes the combinations of levels of each factor and the fault types distribution that have to be set for each treatment.

5.3.4. Experimental Setup and Treatments Execution

This paragraph provides details about the execution of the treatments listed in Table 8, including the experimental setup and test cases execution. As already mentioned, fault injection is preliminarily performed to prepare the application for a treatment; then the treatment is executed by running test cases.
Fault Injection.

[6]We used the source code analysis & metrics tool Understand 2.0, available at: *http://www.scitools.com/products/understand/*

Table 8: The experimental Plan. The combination of levels of the factors, per each treatment, is reported

Treatment #	*Initial Fault Content per Type*					Testing Technique	Initial Fault Content	Software Type
	Assignment	Checking	Interface	Algorithm	Function			
1	31%	19%	0%	50%	0%	Functional	Medium (27)	MySQL
2	11%	15%	17%	50%	7%	Statistical	Low (11)	Samba
3	31%	23%	0%	30%	16%	Stress	Low (9)	MySQL
4	31%	35%	0%	34%	0%	Statistical	High (43)	Apache
5	18%	15%	17%	50%	0%	Stress	Medium (26)	Apache
6	17%	35%	17%	30%	1%	Statistical	Low (9)	MySQL
7	11%	35%	4%	50%	0%	Robustness	Medium (27)	MySQL
8	31%	23%	0%	30%	16%	Statistical	Medium (30)	Samba
9	11%	15%	8%	50%	16%	Statistical	High (45)	MySQL
10	31%	15%	0%	50%	4%	Robustness	Low (9)	Apache
11	22%	15%	17%	30%	16%	Robustness	High (45)	MySQL
12	31%	22%	17%	30%	0%	Functional	High (49)	Samba
13	11%	35%	0%	50%	4%	Stress	High (49)	Samba
14	11%	35%	0%	38%	16%	Functional	Low (9)	Apache
15	11%	35%	8%	30%	16%	Robustness	Medium (30)	Samba
16	11%	26%	17%	30%	16%	Functional	Medium (26)	Apache

Fault injection emulates the presence of faults in the software, by deliberately applying modifications to the code and creating faulty versions of the application. Injected faults have been extensively used in the past for testing techniques analysis, especially under the umbrella of mutation testing (Jia and Harman, 2010). This technique is quite close to our objectives (it injects faults performing simple mutations based on the "coupling effect" in order to evaluate test suites), but it adopts operators that are not necessarily representative of real faults introduced by programmers.

For our purposes, that is relating techniques to the type of faults made by programmers, it is important to adopt an injection approach that considers the following issues: *i)* the representativeness of injected faults, i.e., how much faithfully the injected faults represent real mistakes made by programmers; *ii)* the representativeness of the distribution of fault types, i.e., how much faithfully the injected quantities of faults of each type are realistic.

There are several techniques and tools to inject software faults. The most relevant way is via code changes, i.e., modifications of the code either at binary (Carreira et al., 1995), or at source level (Duraes and Madeira, 2006). To satisfy the representativeness requirements, we choose to inject via code changes at source code level, by adopting the G-SWFIT technique (Duraes and Madeira, 2006). This technique, in fact, exploits a set of fault operators derived from the ODC scheme, which, as discussed in Section 3, has been shown to represent well common programming mistakes (Chillarege et al., 1992; Chillarege, 1995). With regard to the representativeness of fault types distribution, we consider a distribution adhering to the one actually observed in field studies about the presence of ODC fault types in several programs (Duraes and Madeira, 2006; Christmansson and Chillarege, 1996) (cf. with Table 6).

The size of the applications under analysis suggested us adopting an automated fault injection tool. The G-SWFIT-based tool that we adopt is named SAFE (SoftwAre Fault Emulation)[7], developed by our research group and described in (Cotroneo et al., 2009). This tool is able to automatically analyze the source code, build an abstract syntax tree representation, and then identify suitable locations to inject faults, according to code patterns and constraints of ODC fault types. In each treatment, we have created a new faulty version of the application, i.e., we have not reused faulty versions generated in previous treatments, but we have repeated the fault injection campaign from scratch, randomizing the place where faults are injected.

Finally, note that during the treatments execution, we considered exclusively faults injected by us. A failure of a test case caused by a natural fault (if any) is ignored, in order to not alter the *"detection rate"* response variable. Indeed, the time to remove an injected fault (i.e., the debugging time) is almost immediate (and approximately the same) for all the applications, since we are able to track the injected faults; but if we considered natural faults, the debugging time would be not the same for all the applications (depending on the number of faults and on the difficulty to find them); hence the *detection rate* would be biased from

[7] *http://www.mobilab.unina.it/SFI.htm*

how many natural faults are present. As the DoE approach suggests, we limit the potential effect of this uncontrollable factor on the detection rate response variable, by ignoring natural faults.

Test cases Generation and Execution.

The generated plan imposes four testing sessions with Functional testing, five with Statistical testing, three with Stress testing, and four with Robustness testing. Each session (i.e., a treatment) executes test cases in a different, randomly selected, order, to minimize the effect due to the execution order.

Test Cases Writing

Test cases writing is carried out once, before treatments execution. Each treatment is preceded by a preliminary phase to set up the environment and test cases to run.

In some cases, tests writing required a little work, either for the support of available tools, or for the availability of test cases; in other cases we wrote test cases from scratch. In particular, both Samba and MySQL provide an extensive test suite and a valuable support for executing tests. The available facilities allow executing test suites with easy-to-use scripts, to modify them and to write new test cases easily; Apache, instead, required more effort, since its support to testing is growing, but still limited. Due to these differences in the preliminary setup phase, the setup times have not been considered in the analysis.

In detail, functional test cases for MySQL and Samba are the ones provided with the corresponding distributions, i.e., they are the test cases created by developers, with little modifications. The distributions also provide scripts to execute test cases, as well as easy-to-use utilities (such as *smbTorture* for Samba). With regard to Apache, functional test cases have been written from scratch, with the support of the tool *JMeter*[8], by adopting the same criterion as for MySQL and Samba test suite, .i.e., inputs partitioning: identification of functionalities and their inputs from documentation, identification of classes of values, random input selection from classes.

For Statistical testing, the operational profiles have been estimated from real usage data about the most used functionalities, obtained either from the literature (e.g., (Matias and Filho, 2006), (Grottke et al., 2006)), or from documentation information. To generate test cases, the functionality to test is first selected according to a distribution adhering to the estimated profile; then the test case for that functionality (with only valid inputs) is selected randomly, according to a uniform distribution, from the available test suite.

Test cases for stress testing are defined manually, by listing the provided operations and setting different levels of load. For MySQL we used *MySQLslap*[9], that is a diagnostic program to emulate load for a MySQL server. Load is imposed in terms of concurrent clients, and number of operations. For Samba, we used the mentioned *smbTorture* utility, which allows setting the load in terms of number of parallel connections, and number of operations. For Apache, the above mentioned tool *JMeter* has been used to set load levels, by configuring the number of parallel threads connecting to the server, the request delay, the ramp-up period, the type of page (static with plain text, with pictures and/or videos, dynamic).

For robustness test cases, we provide invalid input parameters while invoking the application functionalities, adopting a "Balista-like" approach to select invalid values (Koopman and DeVale, 2000). These test cases are generated manually, by modifying the existing test suite. In the case of MySQL and Samba, we exploited the facilities provided with the distribution to write new test cases; whereas for Apache, we again made use of *JMeter*.

Test Cases Execution

Table 9 summarizes the number of test cases that have been executed in the experiment. The total number of executed test cases in all the treatments amounts to 2748 test cases, executed in 256 h of experimental time. Table 9 shows test cases breakup for each application and for each technique.

The number of test cases for the chosen applications basically depends on the size of the application under test. For instance, for MySQL, which approximately counts 300 KLoC, the test suite size for functional testing amounts to 568 test cases, whereas for Apache, whose size is in the order of 100KLoC, the functional test suite is composed of 160 test cases.

To fairly compare the techniques' effectiveness, we need to assure that the same *effort* is spent for applying

[8]Apache Jmeter is an open source software to load test functional behavior and measure performance, available at: http://jakarta.apache.org/jmeter/

[9]MySQLslap, available at: http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html

Table 9: Experiment details

	Apache	*MySQL*	*Samba*	**Total # of test cases**	**# of Treat.**
Functional	160	568	256	984	4
Statistical	164	575	261	998	5
Stress	78	210	82	370	3
Robustness	96	188	110	394	4
Total	*498*	*1541*	*709*	*2748*	*16*

each technique, i.e., technique is more effective than another if it detects more faults standing the same effort. For this reason, a limit of at most two days of calendar time (approximately 16h) has been imposed for each testing session.
Clearly, different techniques require different times to execute test cases; thus, the number of test cases are different from technique to technique, standing the same available time. For instance, the execution of robustness test cases requires more often to be stopped, since even when the application correctly handles the invalid inputs (i.e., it has no robustness flaws) the application often stops and exits, requiring the system restarting. Hence, the number of robustness test cases is lower than the others; we carried out preliminary estimates of the average execution times of one test case, per each pair application/testing technique, in order to tune the number of test cases to execute in two days. Note that to judge the cost of the application of each technique, absolute time cannot be considered, since it has been limited to two days. Instead, the *detection rate* is considered as response variable, i.e., the number of detected faults per hour.

Table 10: Results of experiments. For each treatment the collected data for each response variable are reported

	Effectiveness per Fault Type					*Cost*	*Absolute Effectiveness*	
	Assignment	*Checking*	*Interface*	*Algorithm*	*Function*			
Treatm. #	**Detected Faults**	**Detected Faults**	**Detected Faults**	**Detected Faults**	**Detected Faults**	**Detection Rate (Faults/h)**	**Detected Faults**	**% of Detected**
1	4 (80%)	3 (75%)	4 (80%)	9 (69.23%)	- (-)	1.176471	20	74.07 %
2	1 (100%)	1 (50%)	1 (50%)	3 (60%)	1 (100%)	0.4375	7	63.63 %
3	2 (66.67%)	2 (100%)	- (-)	2 (66.67%)	0 (0%)	0.5	6	66.67 %
4	11 (84.61%)	12 (80%)	- (-)	11 (73.33%)	- (-)	1.789474	34	79.07 %
5	2 (40%)	2 (50%)	3 (75%)	8 (61.53%)	- (-)	1.153846	15	57.69 %
6	1 (50%)	1 (33.33%)	2 (100%)	2 (100%)	- (-)	0.461538	6	66.67%
7	1 (33.33%)	4 (44.44%)	1 (100%)	3 (7.14%)	- (-)	0.5625	9	33.33 %
8	6 (66.67%)	6 (85.71%)	- (-)	7 (77.78%)	2 (40%)	1.5	21	70.00 %
9	3 (60%)	4 (57.14%)	1 (25%)	14 (63.63%)	4 (57.14%)	1.3	26	57.78 %
10	1 (33.33%)	1 (100%)	5 (62.50%)	3 (60%)	- (-)	0.454545	5	55.56 %
11	2 (20%)	3 (42.86%)	3 (37.50%)	2 (15.38%)	1 (14.29%)	0.684211	13	28.89 %
12	13 (86.67%)	6 (54.54%)	- (-)	8 (53.33%)	- (-)	1.428571	30	71.42 %
13	2 (40%)	12 (70.59%)	- (-)	12 (48%)	1 (50%)	1.5	27	55.10 %
14	1 (100%)	2 (66.67%)	- (-)	2 (66.67%)	2 (100%)	0.583333	7	77.78 %
15	0 (0%)	3 (30%)	3 (100%)	1 (11.11%)	1 (20%)	0.5	8	26.67 %
16	2 (66.67%)	6 (85.71%)	3 (75%)	5 (62.50%)	2 (50%)	1.285714	18	69.23 %

5.3.5. Experimental Results and Data Analysis

In this Section, data collected during the experiments are analyzed with respect to the outlined investigation goals. A summary of obtained results is in Table 10.
The first column reports the treatment number. From column two to six, it is reported the number (and the percentage) of detected faults per type. Note that some of these columns have no value in some cells, meaning that, in the corresponding treatment, the value for that variable had to be 0 (e.g., in the first treatment, the number of function faults to inject is 0; thus no function faults could be detected). The seventh column reports the detection rate. The last two columns report, respectively, the total number and the percentage of faults detected.
In the following, we report results of the analysis on the obtained experiment data by ANOVA followed by a post-hoc Tukey analysis for each response variable; then, a further analysis on testing technique is reported

to better highlight the proneness of each technique toward fault types. Finally, results of a Multiple ANOVA are shown.

Experiment Data Analysis
For each response variable, we first run ANOVA models, so as to investigate the impact of factors on response variable; then a post-hoc analysis is performed to better highlight differences among levels of a factor (e.g., among testing techniques).
As for the experiment data, we carry out a three-factors ANOVA, by selecting the most appropriate model (considering the number of available treatments). In particular, among all the possible three-factor models, we neglect the full (i.e., *saturated*) model, which accounts for all the main effects, all the two-way interactions and the three-way interaction. Then, we compare the significance of all the possible three-factors two-way interaction models, i.e., of those models with some or all the two-way interactions. Considering a significance of 0.05, we select only the statistically significant model(s). If more than one model is significant, we select the most significant one (i.e., the one with the lowest $p - value$). If none of such models is significant, we consider the three-factors one-way model: $y_{i,j,k} = \mu + \alpha_i + \beta_j + \gamma_k + e_{i,j,k}$, with $e_{i,j,k}$ being the error and μ the mean.
After ANOVA, we run a post-hoc analysis, performing the HSD (*honest significant difference*) Tukey's test on differences among means of factor's levels.

Fault Detection Ability.

ANOVA
The first point is to investigate the effectiveness of techniques in detecting faults. Since the variable "number of detected faults" is affected by the number of faults that are initially injected, we evaluate the percentage of detected faults with respect to those initially injected. Table 11 reports results of ANOVA. The most significant model turned out to be the one with two-way interaction between the initial fault content factor and the software type. Analyzing the factors effect, the only influencing factor on detected faults is the testing technique, with a $p - value$ of 0.011. Results confirm that the initial number of faults and the software type do not affect the response variable, and that neither their interaction affects the response variable, being $p - value$s greater than 0.1.
Table shows also the assumption verification. In particular, ANOVA can be applied if the following assumptions are verified: *i)* independence of experiments, obtained with the application of the DoE approach; *ii)* Homoscedasticity of variances; *iii)* Normality of Residuals. For Homoscedasticity condition we performed the following tests: *O'Brienn Test, Brown-Forsythe Test, Levene Test, Bartlett Test*, to the test the null hypothesis that variances of levels of variables are the same. Whereas for normality, we apply the Shapiro-Wilk test, in which the null hypothesis states that residuals come from a normal distribution. In both cases, accepting the hypothesis means that the assumptions are verified. As for Homoscedasticity, note that by performing 4 tests, we can neglect potential cases, if any, in which one of the tests is not verified. In this case, all the tests are accept the null hypothesis. Moreover, ANOVA requires the number of data points for the response variable to be balanced with respect to the factors. This is also verified, since we have 4, 5, 3, 4 points for testing technique; 6, 5, 5 for initial fault content and 6, 5,5, for software type.
Post-hoc Tukey Analysis
To better investigate differences among factor levels' means, we run the HSD Tukey's test for each factor. Table 12 reports data aggregated by *testing techniques*, *software type*, and *initial faults content*. Observing data aggregated by testing techniques, it is worth noting that, in the average, *i)* Functional and Statistical testing detect a greater percentages of faults than Stress and Robustness testing; *ii)* Stress testing seems to be closer, from this point of view, to Functional and Statistical testing than to Robustness testing, and *iii)* Robustness testing detects fewer faults than the others. HSD Tukey tests report that:

- the difference between Functional and Statistical testing (i.e., 5.69%) is not significant ($p - value = 0.7606$), as well as the difference between Functional and Stress testing (which is 13.30 % at $p - value = 0.2339 < 0.05$);

- Functional testing detected 37.01% more faults than Robustness testing ($p - value = 0.0003 < 0.05$);
- the difference between Statistical testing and Stress testing (7.61%) is not significant ($p - value = 0.6333$); whereas the difference between Statistical testing and Robustness testing (31.32%) is significant with $p - value = 0.0008 < 0.05$;
- Stress testing detected 23.71% more faults than Robustness testing ($p - value = 0.0166 < 0.05$).

Hence, all the techniques behave better than Robustness testing significantly in the total number of faults detected. It is worth to note that there is a non-negligible differences between Functional and Stress testing and between Statistical and Stress testing, even though they are not statistically remarkable. Functional and Statistical testing have very similar performances. The slight difference, even if not significant, can be explained by considering that Statistical testing aims to improve reliability, exercising mainly the part of code more likely exercised at runtime; Functional testing covers a wider portion of the code (a similar explanation makes sense for Stress testing, since it exercises some common functionalities with overloaded inputs, but does not span over the code as Functional testing).

Table 11: Results ANOVA for the *Percentage of detected faults*. Text in boldface indicates that the factor is significant

Model: *DetFaults* = Technique + Faults + Software + Faults*Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	11	3924.445	6.7162	**0.0404**
Error	4	212.482		
Total	15	4136.928		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	3	2420.072	15.186	**0.0119**
2: Faults	2	68.754	0.6472	0.5708
3: Software	2	298.169	2.806	0.173
2-3 Interaction	6	258.669	1.217	0.426
Normality of Residuals. Shapiro-Wilk Test: *p-value*				0.9775
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	0.7776	0.7386	0.6134	0.8715
Faults	0.2419	0.0692	0.3046	0.3859
Software	0.6485	0.5618	0.8839	0.7049

Table 12: Results of detection effectiveness, grouped by factors

Factors	**Average % of Faults**
Grouped by Testing Technique	
Functional	73.13%
Statistical	67.43%
Stress	59.82%
Robustness	36.11%
Grouped by Software Type	
MySQL	54.47%
Apache	67.87%
Samba	57.36%
Grouped by Initial Number of Faults	
Low	66.06%
Medium	55.17%
High	58.45%

In the second part of Table 12, data are aggregated by *software type*. By adopting further hypothesis tests, we can observe that the differences in faults percentage detected in Apache and Samba (i.e., 10.51 %), Apache and MySQL (i.e., 13.4 %), and Samba and MySQL (i.e., 2.89 %) are not statistically significant ($p - value => 0.1$ for all the tests). This means that software type has not affected in this experiment the ability of testing techniques to detect faults, confirming the ANOVA results. Similarly, the last part of Table

12 reports the same data aggregated by the *initial number of faults* factor. We can derive that the difference in faults percentage detected with an initial low faults content and an initial high faults content (7.61%) is not statistically significant ($p - value = 0.22 > 0.1$). This also confirms the ANOVA result. Hence, the only factor affecting the *detection effectiveness* turned out to be the *testing technique*.

Fault Detection Cost.

ANOVA

The variable measuring fault detection cost is the *fault detection rate*. Table 13 reports results of ANOVA. The most significant model turned out to be the one with two-way interaction between the initial fault content and the software type factors. Analyzing the factors effect, the only influencing factor on detection rate is the initial faults content, with a $p - value$ of 0.047, whereas neither testing technique nor software type, nor the interaction between fault content and software type affect the response variable, being $p - value$s greater than 0.05.

Post-hoc Tukey Analysis

Table 14 reports data aggregated by the three factors: *testing techniques*, *software type* and *initial number of faults*. Observing data aggregated by testing technique, results of hypothesis tests on differences among means show that none of the differences among the techniques is not statistically significant (Functional - Robustness = 0.568, $p - value = 0.3343$; Statistical - Robustness = 0.547, $p - value = 0.3222$; Stress - Robustness = 0.500, $p - value = 0.4988$; Functional - Stress= 0.067, $p - value = 0.9973$; Statistical - Stress = 0.046, $p - value = 0.990$; Functional - Statistical = 0.0208, $p - value = 0.999$). The most noticeable difference, even though not significant, is between Functional and Robustness, and Statistical and Robustness; the lowest rate of robustness can be explained by considering that it requires testing to be stopped more often. In the second part of Table 14, data are grouped by software type; we can investigate the dependence of detection rate on software type. It is shown that the difference between the highest mean detection rate (observed in Samba) and the lowest one (observed in MySQL), 0.2924 faults/hour, is not significant ($p - value = 0.5397 > 0.1$), as for the other combinations; hence, there has been no difference in the detection rate due to the software type. Finally, the last part of Table 14 reports the same data aggregated by the factor initial number of faults. We observe that applications with the highest initial number of faults experienced, in the average, a higher detection rate than those with low initial fault content (the average difference has been 0.8530 faults/hour, significant at $p - value = 0.0043 < 0.05$). It is clear that at the beginning of the test, in software with a high initial faults content more faults are exposed and the initial rate is very high; then, as faults are removed, the behavior in the two cases should be almost the same. Other differences are not significant (Low-Medium=0.542, $p - value = 0.0506$; $Medium - High = 0.3106$; $p - value = 0.3189$).

Attitude Towards Fault Type.

ANOVA

The last point is to investigate the effectiveness in detecting faults of different types. This point is important for aiding the selection of the proper testing techniques, once software has been characterized in terms of expected fault types. The response variables are the percentage of detected faults per type, rather than the number of faults per type, since the latter is affected by the number of faults initially injected.

ANOVA results are reported in Table 15. For each response variable, the ANOVA model is reported in the Table along with its significance; where two-way interaction are not present, it means that the model was not sufficiently significant to estimate such a potential effect, and thus the analysis focuses on main effects of the considered factors. The ANOVA results show that the testing technique has an impact, being the hypothesis significant for the *Assignment*, *Algorithm* and *Function* fault type response variables. Results also show an impact of the interaction between technique and software type on the *Checking* fault type, even though the overall model has a significance slightly higher than 0.05 (it is 0.06, i.e., 94 % of confidence). Finally, the models for *Algorithm*, *Function* and *Interface* fault type are additive model, meaning that there

Table 13: Results ANOVA for *detection rate*. Text in boldface indicates that the factor is significant

Model: $DetRate$ = Technique + Faults + Software + Faults*Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	11	3.21576	7.4419	**0.0337**
Error	4	0.15713		
Total	15	3.37289		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	3	0.7122	6.2896	0.0539
2: Faults	2	2.12852	27.0921	**0.0047**
3: Software	2	0.25578	3.2557	0.1448
2-3 Interaction	6	0.14331	0.9121	0.5345
Normality of Residuals. Shapiro-Wilk Test: *p-value*				0.8185
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	0.8177	0.6653	0.6888	0.8982
Faults	0.0785	**0.0003**	**0.0121**	0.0513
Software	0.3572	0.1066	0.4235	0.5626

Table 14: Results of cost response variable, grouped by factors

Factors	**Avg. Faults Detection Rate (Faults/h)**
Grouped by Testing Technique	
Functional	1.118522409
Statistical	1.097702429
Stress	1.051282051
Robustness	0.550313995
Grouped by Software Type	
MySQL	0.780786596
Apache	1.053382582
Samba	1.073214268
Grouped by Initial number of Faults	
Low	0.48738345
Medium	1.029755171
High	1.340451128

are no sufficient point to estimate interactions. In the *Function* fault type case, all the factors have an impact.

Post-hoc Tukey Analysis

Finally, Table 16 reports data aggregated by the three factors for each of the *percentage of detected fault per type* response variable. In the first part of the Table, it is reported, in each row, the percentage of faults per type detected grouped by technique. From results, it is clear that, in the average, Functional testing is the fairest technique (it detects comparable percentages of distinct fault types), and that Robustness and Stress testing are more prone to detect some kinds of fault rather than others. Similarly, some fault types are more easily detectable with a technique than with others (e.g., *Assignment* faults with Functional testing, *Interface* with Robustness testing).

On these results, the *Tukey* test on the differences among mean values is applied. The statistically significant results are:

- Functional testing detects 61.67% more Assignment faults than Robustness, with $p - value = 0.0247$.
- Functional testing detects 39.57% more Algorithm faults than Robustness, with $p - value = 0.0254$.
- Statistical testing detects 51.54% more Algorithm faults than Robustness, with $p - value = 0.0035$.

At $\alpha = 0.1$,

- Statistical testing detects 50.58% more Assignment faults than Robustness, with $p - value = 0.0976$.
- Stress testing detects 35.32% more Algorithm faults than Robustness, with $p - value = 0.0534$.

Table 15: Statistics of ANOVA for the *percentage of faults type*. Text in boldface indicates that the factor is significant. Legend: *DF*: Degree of Freedom; *SS*: Sum of Squares; *Technique*: Testing Technique; *Faults*: Initial # of Faults; *Software*: Software Type; Y_{Type}: percentage of detected fault type response variable

Percentage of Detected Assignment Fault				
Model: Y_{Ass} = Technique + Faults + Software + Faults*Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	11	9250.772	7.761	**0.0313**
Error	4	434.4		
Total	15	9684.172		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	3	5817.5	17.897	**0.0088**
2: Faults	2	1045.514	4.824	0.0859
3: Software	2	507.537	2.342	0.212
2-3 Interaction	4	1480.104	3.415	0.130
Normality of Residuals. Shapiro-Wilk Test: *p-value*				0.3384
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	0.4427	0.3536	0.1797	0.4116
Faults	0.4123	0.1251	0.6215	0.7003
Software	0.6189	0.2120	0.6987	0.8355

Percentage of Detected Interface Fault				
Model: Y_{Int} = Technique + Faults + Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	7	5576.3	1.9701	0.3775
Error	2	806.695		
Total	9	6385.0		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	3	502.3839	0.4142	0.7628
2: Faults	2	4616.304	5.708	0.149
3: Software	2	632.375	0.782	0.561
Normality of Residuals. Shapiro-Wilk Test: *p-value*				0.3684
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	0.4692	0.6354	0.3580	0.7942
Faults	0.3195	0.5789	0.3643	0.6938
Software	0.37	0.7151	0.1745	0.5515

Percentage of Detected Function Fault				
Model: Y_{Fun} = Technique + Faults + Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	6	9652.434	23.757	**0.0409**
Error	2	135.432		
Total	8	9787.867		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	2	3088.061	22.801	**0.042**
2: Faults	2	3494.908	25.805	**0.0373**
3: Software	1	2831.048	41.807	**0.0231**
Normality of Residuals. Shapiro-Wilk Test: *p-value*				0.3671
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	-	**0.0113**	0.0748	**<0.001**
Faults	0.4757	**0.0055**	0.2479	0.1809
Software	0.8704	0.2038	0.6327	0.5871

Percentage of Detected Checking Fault				
Model: Y_{Chec} = Technique + Faults + Software + Technique*Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	13	7346.390	16.0861	0.06
Error	2	70.260		
Total	15	7416.651		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	3	403.546	3.8291	0.2140
2: Faults	2	1033.308	14.7069	0.0637
3: Software	2	1420.163	20.2129	**0.0471**
1-3 Interaction	6	5504.946	26.1169	**0.0373**
Normality of Residuals. Shapiro-Wilk Test: *p-value*				**0.0013**
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	0.437	0.243	0.2744	**<0.001**
Faults	0.4276	0.3567	0.3567	**0.0020**
Software	0.3388	0.3464	0.3464	**0.0011**

Percentage of Detected Algorithm Fault				
Model: Y_{Alg} = Technique + Faults + Software				
Model Stats.	*DF*	*SS*	*F-ratio*	*p-value*
Model	7	7788.461	5.5269	**0.0140**
Error	8	1610.496		
Total	15	9398.957		
Factors Effect	*DF*	*SS*	*F-ratio*	*p-value*
1: Technique	3	5832.135	9.6569	**0.0049**
2: Faults	2	758.563	1.8840	0.2136
3: Software	2	452.342	1.1235	0.3715
Normality of Residuals. Shapiro-Wilk Test: *p-value*				0.7852
Homoscedasticity. Tests *p-value*				
Factors	O'Brienn	Brown Forsythe	Levene	Bartlett
Technique	0.5766	0.4498	0.3452	0.3282
Faults	0.0508	**0.0123**	**0.0364**	0.0829
Software	0.9603	0.9585	0.9738	0.9644

Other values are worth to mention, such as the ability of Robustness and Stress testing in detecting more *Interface* faults than other type of faults, as well as the proneness of Stress testing toward *Checking* fault type. Similarly the poor value of Robustness testing for *Function* faults is also evident.
The second part of the Table shows the mean values of detected faults per type grouped by the initial faults content factor. No statistical evidence have been observed among all the differences, but the difference in the *Interface* fault type between the initial faults content set to medium level and the same factor set to high level(54.75%, $p - value = 0.0167$).

Table 16: Results of detection effectiveness per fault type, grouped by the Testing Technique factor

Technique	**% of Detected Faults**				
	Assignment	**Checking**	**Interface**	**Algorithm**	**Function**
Functional	83.34%	70.48%	77.5%	62.93%	75.00%
Statistical	72.25%	61.24%	58.33%	74.95%	65.71%
Stress	48.89%	73.53%	75.00%	58.73%	25.00%
Robustness	21.67%	54.33%	75.00%	23.41%	17.15%
Grouped by Software Type					
MySQL	51.67%	58.80%	68.50%	53.68%	23.81%
Apache	64.92%	76.48%	70.83%	64.81%	75.00%
Samba	58.67%	58.17%	75.00%	50.04%	52.50%
Grouped by Faults Content					
Low	70.00%	70.00%	70.83%	70.67%	66.67%
Medium	47.78%	61.81%	86.00%	48.22%	36.67%
High	58.26%	61.03%	31.25%	50.77%	40.48%

The final part of the Table shows response variable values grouped by software type. Also in this case, no statistically significant difference has been detected in the response variable with respect to all the possible pairs of factor levels.

Analysis of Testing Techniques *vs.* Detected fault type
Besides Tukey post-hoc analysis on factors' averages, in order to have a clearer picture on the relative ability of techniques with respect to fault types, we carry out a further analysis comparing mean differences, by a *t-student test*, among the percentage of detected fault types given a testing technique. We highlight the following results:

- Functional testing detects percentages of faults of each type that do not differ significantly among each other (as may be seen in the first row of Table 16); the only exception is the *Assignment* faults, that are significantly higher than the others (e.g., the difference between the percentage of *Assignment* faults and *Algorithm* faults, i.e., 20.41%, is significant at $p - value = 0.0196 < 0.05$. This means that Functional testing is quite fair with respect to the different fault types; moreover, these percentages are higher than the other techniques (as discussed in the analysis of the response variable "*total number of faults detected*").

- Robustness testing detects 53.33% more faults of *Interface* type than *Assignment*, 51.59% more than *Algorithm*, 57.85% more than *Function*, and 20.67% more than *Checking* ($p - value < 0.05$ for all the differences);

- Robustness testing also detects more faults of *Checking* type with respect to other types (e.g., $p - value = 0.0851 < 0.1$ for the difference with *Algorithm*, $p - value = 0.0937 < 0.1$ for the difference with *Function*);

- Functional testing and Statistical testing detect many more *Function* faults than Robustness testing (the differences are 57.85% and 48.56%, respectively, with $p - value = 0.0741 < 0.1$ for the former, and $p - value = 0.0633 < 0.1$ for the latter);

Results of Table 16 tell us that Functional testing has a good average attitude towards each fault type, but also that its best performances are obtained with the *Assignment* and *Function* faults. Statistical testing shows a slightly higher attitude towards *Algorithm* faults, and is fair with respect to the other types, for which detection is around at 60 %. Stress testing focuses its attitude on the *Interface* and *Checking* type, whereas Robustness is particularly tailored for *Interface* type, and secondarily for the *Checking* fault type.

MANOVA: Detection Ability and Cost.
It is finally also worth investigating if some factor can affect both response variables of the detection effectiveness and cost "contemporary". We adopt the one-way *multivariate analysis of variance* (MANOVA) to

Table 17: Results of Multivariate Analysis of Variance. Text in boldface indicates that the factor is significant

Hypothesis Test	**Testing Technique**	**Initial # of Faults**	Software Type	Test. Technique Initial # of Faults
Wilk's Lambda	**0.0033478**	**< 0.0001**	0.639354	0.5283410
Pillai's Trace	**0.0343945**	**0.001700**	0.606809	0.4655856
Hotelling-Lawley	**0.0014306**	**< 0.0001**	0.648965	0.4348407
Roy's Largest Root	**0.0001547**	**< 0.0001**	0.406813	0.1740403

evaluate the impact of a each categorical factor on the two response variables. We adopt several hypothesis tests to verify if a factor significantly impacts the pair of response variables, commonly used for MANOVA. They are: the *Wilks' Lambda*, the *Pillai's trace*, the *Hotelling-Lawley trace*, and the *Roy's largest root*. Results are reported in Table 17. Columns 2 to 4 report results of the tests per each relevant factor. It is evident that we can focus the attention on the first two factors, because they have individually influence on the pair of response variables analyzed, unlike the software type factor. Hence, by a two-way MANOVA, we further analyzed if the interaction of these two factors also have influence on the two response variables. The last column of Table 17 reports results of this investigation, showing that the hypothesis that the interaction affects the pair of response variables is rejected. Thus, the testing techniques and the initial amount of faults have influence also on the pair of the considered response variables, but their interaction does not have any effect on it.

5.3.6. Final Remarks

Summarizing, results have shown that Functional testing is able to detect a greater percentage of faults than Stress and Robustness testing, and it seems to be the fairest technique with respect to the fault types it is prone to detect; its difference with Statistical testing is not significantly relevant. Statistical testing also detects more faults than Robustness testing, as well as than Stress testing, and it also turned out to be a "fair" technique with respect to fault types. Functional and Statistical testing also highlighted a better ability in detecting faults of *Function* type, in contrast to Stress and Robustness testing that were not able to deal with such kind of faults. Moreover, Functional testing shows a particular attitude towards *Assignment* faults (83 % of detected faults). Stress testing also detects a good percentage of faults, and shows good performances in detecting *Checking* and *Interface* faults. Robustness testing detects fewer faults than the others, and presented also the lowest detection rate. An explanation is that Robustness testing goal is to detect the "hardest", uncommon faults, even if few, rather than many common faults. However, it is resulted as the best technique to detect faults of *Interface* type (for instance, it is very good at detecting faults on passed parameters), and had a good attitude also towards *Checking* faults. This makes it an important technique, since it complements the other ones in the detection of faults that are hard to be exposed by the other techniques.Hence, Stress and Robustness testing do not exhibit "fair" performances with respect to fault types; they are focused on faults such as those in validating data and in parameters exchanged among modules. On one hand, this confirms that Functional or Statistical testing are the best starting point to detect faults due to their detection ability and their fairness (typically, the information on fault types affecting the software is not available); on the other hand, such techniques expose faults whose activation is more common, i.e., the easiest faults to detect: once they detected a certain number of such faults, a technique able to expose more subtle faults, like Robustness testing, can help in improving the final detection effectiveness.

The characterization of techniques completes the construction of the testing-related base of knowledge, through which testers have the possibility to tailor the techniques selection process to the application. Testers may first apply predictive models to obtain an estimate of fault types affecting the software; then, by using results of techniques characterization, they are driven in the selection of the best mix of techniques towards those types of fault. In the next Section, a simple example on how to use these results is reported. It is finally worth to point out that the objective of the presented experiment is to show how it is actually possible to create a knowledge base, taking some programs and doing the prescribed activities according to the outlined steps; the purpose is not to asserting a general conclusion on techniques performance by an empirical analysis (even though data can be used as source for empirical analyses). Testers will not have to

use the same number of data points to build the knowledge base and apply our method; they will have to address a trade-off between the desired accuracy and the resources employed. The accuracy of the base of knowledge will be tied to how many points practitioners can adopt in their contexts, also considering that these may be iteratively increased with time.

5.4. Exploiting the Knowledge

While in the previous Sections we experimented the method to build the knowledge, in this Section it is showed how the latter can be exploited to have more faults detected. To this aim, a *C* program has been chosen and tested by selecting techniques according to the proposed approach, as suggested by the built base of knowledge. Then, the result, in terms of total number of detected faults is compared against the sheer Functional testing approach, which is the most adopted method in practice, to show the positive effect of combining techniques with the proposed strategy. The application to test has been selected from the publicly available repository *Software-artifact Infrastructure Repository* [10], which is a platform thought for experimenting testing techniques performance (Do et al., 2005). Among the available ones, we selected *Grep* (General Regular Expression Print), a *Unix* utility of about $15KLoC$, whose main static features are summarized in Table 18. There are five versions available in the repository, each one with also a fault-seeded version. For the purpose of the evaluation, we have adopted the most mature one, i.e., the version $v5$.
Among the four characterized testing techniques, we have considered Functional testing and Robustness testing as set of techniques available to the tester. Thus, the application of the method is expected to tell: *i)* if it is worth to combine these techniques, and, if yes, what technique we should use first; *ii)* (if and) when it is convenient to switch to the other technique.
The evaluation starts with the method being applied to the fault-seeded version of the program, which includes 19 faults in the code. The adopted procedure is as follows:

a. Extracting the metrics, i.e., characterizing the application under test, in order to obtain an estimate of the ODC fault content within the program.
b. Obtaining the ODC faults estimate, by using the regression models of the constructed base of knowledge (step 1 of the method). For the models obtained in our case, a PCA transformation must be also carried out on the original metrics. In fact, three out of the five regression models use PCs as predictor variables (cf. with Table 4).
c. Estimating the performance of available testing techniques for the specific application under test. Let us denote the number of estimated ODC faults as $\#Faults_j$, with j being the type of fault. By using results of the techniques characterization (obtained in the step 2 of the method), we define an "attitude index" $A_{i,j}$, expressing how much the technique i is prone to find a fault of type j. For the purpose of this evaluation, we consider simply the percentages of detected faults as reported in Table 16: $A_{i,j} = V_{i,j}/100$ where $V_{i,j}$ is the percentage of faults of type j detected by the technique i (cf. with Table 16) - note that, in general, other more complex approaches could be used to determine the attitudes from historical information. From these indexes, it is immediate to estimate the number of faults that a technique i is expected to detect at time t; we call it detectable faults, $\#DFaults_i$:

$$\#DFaults_i(t) = \sum_{j=1}^{5} \#Fault_j(t) * A_{i,j} \quad with\ i = 1 \ldots N \tag{3}$$

where $\#Fault_j(t)$ is the number of faults of type j expected in the program at a given time t, and N is the number of techniques. The fault estimates at time 0 derives from the previous step.
d. From Equation 3, the technique that should be used to start the testing session is computed. The testing session starts with the technique having the highest $\#DFaults$ value.

[10] The repository contains software programs of different size and features made available for experimentation, and are equipped with test cases, as well as with faulty and non-faulty versions; available at *http://sir.unl.edu/*

Table 18: Grep Metric Values

Metric Values(Normalized Values)					
#1 *CountDeclClass*	0.00 (0.00)	*#12* *C CodeFile*	3,00 (-6,64612E-05)	*#23* *Avg. Vocabulary*	216,0000000 (0,218099488)
#2 *CountDeclFunction*	146 (-0,03045067)	*#13* *CountLineCodeDecl*	1586 (-0,004791206)	*#24* *Avg. Difficulty*	46,5384600 (0,247494814)
#3 *CountLine*	15633 (-0,0023375)	*#14* *CountLineCodeExe*	4242 (-0,007438433)	*#25* *Avg. Effort*	12178129,0000000 (0,53816094)
#4 *CountLineBlank*	1953 (-0,00337229)	*#15* *AverageComplexity*	0,63 (0,077353109)	*#26* *Avg. Bugs Delivered*	8,1673600 (0,353639893)
#5 *CountLineCode*	6864 (-0,003865979)	*#16* *Max Cycl.Complexity*	478,00 (0,414220877)	*#27* *Var. Volume*	1,70E+10 (1,423869295)
#6 *CountLineComment*	3669 (0,000542667)	*#17* *MaxNestingLevel*	9,00 (0,333333333)	*#28* *Var. Length*	1,42E+08 (1,390617104)
#7 *CountLineInactive*	1985 (0,001467102)	*#18* *CountPath*	1513533337,00 (-0,004078525)	*#29* *Var. Vocabulary*	269915,7 (0,122494121)
#8 *CountStmtDecl*	1071 (-0,006130615)	*#19* *Sum of Cycl.Complexity*	2150,00 (-0,001436081)	*#30* *Var. Effort*	1,85E+15 (2,655125361)
#9 *CountStmtExe*	5059 (-0,001407738)	*#20* *Sum of Ess.Complexity*	985,00 (-0,001336288)	*#31* *Var. Difficulty*	8336,604 (0,3768503956)
#10 *RatioCommentToCode*	0,53 (0,629824561)	*#21* *Avg. Volume*	11478,0000000 (0,085708909)	*#32* *Var. Bugs Delivered*	703,3081 (1,689453117)
#11 *C Header File*	10,00 (-8,52604E-06)	*#22* *Avg. Length*	3770,3076000 (0,321863723)		

Table 19: Principal Components from Metrics

PC_1	PC_2	PC_3	PC_4
-0.619	-0.195	0.784	0.852

e. During the testing session, each detected fault is classified according to the ODC scheme and recorded, along with the test case number that exposed it, and then it is removed; by Equation 3, the $\#DFaults_i$ values are updated. If the $\#DFaults$ value of the current technique becomes less then the value of another technique, a switching occurs to the latter; testing proceeds always with the technique having the highest $\#DFaults$.

f. At the end of the testing, the number of faults detected and the number of test cases employed to detect them is recorded.

After the application of the method to test the program, a new testing session is carried out on the same version, by using the Functional testing alone. Results, in terms of number of detected faults and number of employed test cases, are compared with our method's ones.

Applying this procedure, the following results have been obtained. Metrics extraction provides the values summarized in Table 18. From the PCA transformation, 32 principal components are obtained from the 32 original metrics. The models of Table 4 indicate that at least the first 4 PCs are needed (cf. with the model for *assignment* and *algorithm* faults) to carry out the predictions.; these are reported in Table 19. By applying the models, the ODC estimates at time t_0 are obtained, as showed in Table 20, column two. The next step is to figure out what is the technique that should be used to start the testing session, given the

Table 20: ODC Fault Estimation for Grep

Estimation at time t_0		**Estimation at time of switching**	
ODC Fault type	**# of Fautls**	**ODC Fault type**	**# of Faults**
Assignment	5	*Assignment*	0
Checking	12	*Checking*	9
Interface	8	*Interface*	8
Algorithm	2	*Algorithm*	1
Function	2	*Function*	0

initial ODC estimate. Applying Equation 3, the technique to use turned out to be the *Functional testing*, since the $\#DFaults$ value, i.e., the actual detectable faults, is the greatest one ($\#DFaults_{Functional} = 5 * 0.8334 + 12 * 0.7048 + 8 * 0.6417 + 2 * 0.6293 + 2 * 0.75 = 20.5168$, against $\#DFaults_{Robustness} = 5 * 0.2167 + 12 * 0,5433 + 8 * 0.875 + 2 * 0.2341 + 2 * 0.1715 = 15.4143$). Hence testing starts by selecting test cases with the Functional testing criterion.
As testing proceeds, the expected number of faults varies, since more and more faults are detected, classified, and removed. Table 21a shows the testing session progress. In the first column, it is reported the number of test case exposing a fault; in the second column the ODC type of the fault removed; in the third column the updated value of $\#DFaults$ for Functional and Robustness testing. In the fourth column the "severity" of the detected bug, intended as impact of the bug activation on program's behavior. As for the latter we distinguish two classes, based on the observed failures: *major*, when the caused the program crash, and *normal*, where the bug causes a bug to output a result different from the expected one.
For instance, the first row indicates that after 8 test cases, the first failure has occurred, with *normal* severity, caused by an *Assignment* fault; this is removed, and new $DFaults$ values are computed by Equation 3.

Table 21: Results

(a) Testing Session with the Proposed Method

Test Case #	**ODC Type**	$DFaults_{Fun}$	$DFaults_{Rob}$	**Severity**
6	Assignment	19.6834	15.1976	Normal
8	Assignment	18.85	14.9809	Normal
9	Function	18.1	14.8094	Normal
10	Checking	17.3952	14.2661	Normal
11	Assignment	16.5618	14.0494	Normal
21	Function	15.8118	13.8779	Normal
33	Algorithm	15.1825	13.6438	Major
59	Checking	14.4777	13.1005	Normal
66	Assignment	13.6443	12.8838	Normal
69	Assignment	12.8109	12.6671	Normal
71	**Checking**	**12.1061**	**12.1238**	Normal
76	Checking	11.4013	11.5805	Normal
80	Checking	10.6965	11.0372	Major
81	Interface	10.0548	10.1622	Normal
236	Algorithm	9.4255	9.9281	Major
262	Checking	8.7207	9.3848	Normal
264	Interface	8.079	8.5098	Normal

(b) Testing Session without the Proposed Method - Functional Testing

Test Case #	**ODC Type**	**Severity**
8	Assignment	Normal
10	Assignment	Normal
71	Function	Normal
74	Assignment	Normal
81	Algorithm	Normal
98	Assignment	Normal
103	Checking	Normal
146	Assignment	Normal

The test suite is made up of an initial set of 610 test cases for Functional testing, and 300 test cases for Robustness testing. As Table 21a shows, after 71 test cases, eleven faults, of various type, are detected and removed; at this point, the $\#DFaults$ value of Robustness testing becomes greater than the one of Functional testing. This indicates that for the types of faults remained in the program, Robustness testing is likely to detect more faults from that point on. Residual faults at this time are in Table 20, column 4. Following this suggestion, the testing proceeds with Robustness testing. Switching to Robustness testing allows uncovering other 6 faults, ending up with 17 faults removed, with 371 test cases (from 264 to 371 no other fault is detected).

In order to compare this strategy with Functional testing alone, an additional testing session is carried out. This session adopts a test suite of 671 functional test cases. Such effort is considered equivalent to the effort allocated to the combined strategy, since it is made up of 71 test cases (as the ones employed for Functional testing in the previous case), and of 600=300 · 2 test cases, derived by considering the effort for a robustness test case as roughly double as the effort for a Functional test case. This indication has been derived by some preliminary tests, which suggested us that the time to execute a robustness test case is higher because of the frequent interruptions also in case of no faults detected.
Table 21.b shows the results obtained by the application of the sole Functional testing. In this case, 8 out of 19 faults have been removed with 671 functional test cases, whereas with the combined approach 17 out of 19 with 71 functional test cases and 300 robustness test cases.
The obtained result highlights the benefits of applying a strategy for combining techniques: in this case the number of residual *interface* faults, made robustness testing able to detect effectively faults as compared to continuing with functional testing.
From the severity point of view, we can observe, from this case, that combining techniques may increase the possibility to reveal bugs with different severity; for instance, considering robustness testing makes the approach find more crash bugs, because that kind of technique is based on unexpected and abnormal conditions as inputs and is more likely to activate such type of bugs.
As final considerations, it is also worth to point out that the strategy does not necessarily suggest to switch among techniques, e.g. when the initial technique turns out to have the highest $\#DFaults$ value for the entire testing session. It should be also noted that in this experiment the attitude index of techniques is computed in the simplest way, i.e., by considering the percentage of faults detected: alternatively to this approach, more complex strategies could be put in place to analytically find the best policy by adopting stochastic models, such as Markov Decision Processes (MDP), and its variants (e.g., partial observable MDP). In such a case, the *objective* would be the maximization of the number of faults; *states* would correspond to detected faults of each type, *actions* to the testing technique to choose, and a *policy* would be the best set of actions (techniques choice) maximizing the objective. Strategies for the best exploitation of obtained results will be the main subject of our future work.

5.5. Effort for Applying the Method

This section briefly discusses the cost incurred by practitioners willing to adopt this method. It should be immediately noted that applying such a method for testing just one single product is not worth. The method is intended to be included within a V&V process of an organization. It is indeed worthwhile when, within a company, more than one product will be tested since the underlying idea is to introduce the ability to structure the knowledge on testing techniques and software products within a company and to be able to exploit it for future products. This may be especially effective in product families.
As any process change, this implies a startup cost expectedly compensated in the medium-long term (such as in the mentioned cases of process/defect analysis works, i.e., ODC, RCA, FST, as well as in characterization scheme of Basili (Vegas and Basili, 2005)). Such a cost, as well as benefits, may be highly variable depending on the context and on the features and initial conditions of the organization adopting the method. For instance, there may be the extreme case in which a company does not have any type of historical data available, and does not collect data, in which case the cost of starting tracking data, and/or relying on external sources of data is much different from companies with a usual defect tracking system and/or with available programs as historical data. In the same way for the benefits, which should be regarded not only referred to the testing improvements. For instance, the benefits coming from defects tracking and defect analysis activity regards not only the savings for the greater number of faults detected, but also the overall process quality.
More in detail, the initialization cost is related to building the base of knowledge. The factors mostly impacting the effort/benefits obtainable are: the ability of a company to track historical data about detected faults, including their type; the availability of programs as historical sample data; the number of testing technique to characterize; the size of available programs on which the characterization is carried out; the ability of extracting metrics; the competence in dealing with the presented prediction models and with DoE techniques. Specifically, step 1 requires practitioners to have data on fault types occurred in a set of

applications of which they can extract the metrics (hopefully, company's internal products). In turn this requires *i)* collecting fault types data along the process, i.e., of recording the type of fault fixed during a testing campaign, and *ii)* extracting metrics from source code of products (e.g., adopting an automatic tool). For step 1, recording data on fault types is the most important process change. Tracking faults data is a practice suggested also for other purposes (related to the overall process quality measurement and improvement), for instance by CMMI certification at a certain levels of quality. Thus, the effort required by this step is negligible for organizations already collecting this data; while it corresponds to the cost of organizing an activity, along the process, for recording fault occurrence for organization not adopting this practice (in this case the resulting benefit is not only related to the V&V phase). The cost of formulating models to make the actual predictions from data is negligible (since there exist many tools for data analysis building models automatically). If an organization does not have an initial base of products, it can rely on external sources (as the one provided in this paper) or can start building its base of knowledge collecting the mentioned data across products development.

The second step requires the characterization of techniques. This cost depends basically on the number of techniques that testers want to included in the process. The more the techniques, the more the experiments to carry out to characterize them. As in the case of prediction models of step 1, it is important to remark that this cost is sustained once, at the beginning of the process. Then, all the produced knowledge will be iteratively updated with data coming from the V&V of new products, and thus exploited to have more and more effective testing processes for upcoming products. The accuracy of predictions (of faults and of technique performances) increase with time, and thus also the benefits (as compared to the initial cost) in detecting more faults in less time. To figure out when one can start applying the method (i.e., when the knowledge is statistically significant), the statistics of the model should be considered. Thus, for fault prediction models the significance (the confidence) of the obtained model, which should be at least less (greater) than 0.05 (95%), and the goodness of fit, measured by R^2 and adjusted R^2 values with meaning already explained (a rule of thumb can indicate an adjusted R^2 at least greater than 0.7, with R^2 being greater than the adjusted one). As for the techniques characterization, the needed information is about the relative effectiveness of a techniques with respect to fault types (i.e., we need to know if a technique is more prone toward a fault type or another); thus hypothesis tests that we performed (in our case $F - test$ and t-test) may indicate that we can have a confident application of the technique when the confidence is at least greater than 95%. From that point on the accuracy will increase as new products (i.e., samples) are developed and tested in the organization.

6. Threats to Validity

This Section describes the main threats to validity that a practitioners applying this method must be aware of. Since the approach relies on empirical data, the first point concerns with the generalization of results obtained in one context to other contexts.

While the approach easily generalizes, care must be taken in interpreting results deriving from applying the approach within an organization, since one application of the approach is similar to instantiating an empirical study. Thus, practitioners must be aware that results obtained in one context (referring to "context" as an organization with its own skills, personnel, products, tools adopting a specific V&V process) are hardly applicable to other contexts, both regarding the ODC fault types predictive models, and regarding the techniques characterization: to apply them in different contexts, they should consider relationships of the applications examined with the particular environment or projects in which the results might be applied. Indeed, if the same results obtained in one environment are confirmed in others, the testers may draw also more general conclusions, but this is outside the scope of this paper. Sometimes this aspect may be underestimated. For instance, fault prediction method is one of the most reliable ones to support decisions via quantitative analyses on the process; due to its success and to the quantitative support, people may be led to think that results are general, only because statistically valid; however, as pointed out in many papers, generalization, without specific solutions, of results from one context to another should be avoided. Other than generalization, some other issues may threat the validity of the application of the method. For

instance, the number of treatments may represent a limiting point. However, this number is always a trade-off between lab resources and results accuracy: in our case study, the adequacy of the plan is supported by the measures provided by the tool JMP, such as D-efficiency, by which the plan is generated.
Finally, it is important to remark that the approach must be regarded as a quantitative support to tester's decision; however, being the method based on empirical data, it cannot totally substitute tester's decision, but it should be regarded as a way to reduce the impact of subjective judgment. Empirical models will never give the deterministic guarantee about the goodness of the strategy's performance, but only a probabilistic indication; thus, to judge if models are "sufficiently" accurate for decision support or not depends on which confidence the tester decides to accept; when deciding about techniques, s/he has to take into account that available models are valid at a given confidence level. With empirical models, it may also happen that the process history on which models are built is no longer representative of a context; if significant changes in development process occur, this should be managed by human judgment.
As for fault injection and test cases execution, we mentioned potential threats in Section 5 and discussed the way by which we limited their effect. They may be helpful for who adopts the outlined approach. Hereafter, they are summarized:
Representativeness of injected faults, i.e., how much faithfully the injected faults represent real mistakes made by programmers, and the injected quantities of each type are realistic. We coped with this by adopting the ODC scheme, along with G-SWFIT, and taking quantities from well-established literature.
Location, i.e., the effect that the place where faults are injected can have on the outcome: to minimize this, we adopted an automatic fault injection tool, that looks for location where it is possible to inject, and then we have not reused faulty versions generated in previous treatments, but we have repeated the fault injection campaign from scratch in each treatment, randomizing the place where faults are injected. This choice limits, even if not eliminates, this side effect.
Test Case (TC) execution; the order of execution of test cases affects the outcome; in our experiment, each testing session (i.e., a treatment) executes tests in a different, randomly selected, order. We advise making a similar choice to limit this threat. Moreover, the effect of the experimenter has to be also taken into account. In our case, all the treatments are executed by the same team: considering how we conducted the experiment, the effect of the experimenter ability can be considered negligible. Actually, *i)* TC derive from developers' test suite, hence their generation is only slightly affected by the ability of the experimenter, who just adapted them; *ii)* TC setup time and debugging time (which are manual, and possibly affected by the experimenter ability) are excluded from response variable analysis, for the reasons explained in Section 5; *iii*) the actual TC execution, as well as the fault injection process, is automized, and the experimenter just launched the test session and observed what happened.

7. Conclusions

Given a software under test, there exists a relationship between its features (in terms of the value of common software metrics) and the types of faults it contains, and between these and the attitude of testing techniques to reveal them. Although the real nature of these relationships is in general not fully understood yet, based on this assumption we devised a pragmatic method to let testing engineers to characterize a software under test in terms of the estimated number of faults of different types it contains, and then to select the techniques more prone to detect the greatest number of faults of the various types for the given application. Since no software testing technique suits all needs, all application types and all contexts, the choice of the techniques should be tailored to the specific context; the proposed method allows to mix techniques so as to maximize the number of faults revealed for the software under test among those estimated to be present. Thus, the method provides engineers within an organization with a practical way for planning testing activities and efforts. As for fault types, the method refers to the well-known ODC classification; indeed, this work is the first one, to the best our knowledge, that relates testing techniques to ODC fault types and these to software metrics. As for testing techniques, we considered techniques of functional testing, statistical testing, robustness testing, and stress testing.
The method consists of two complementary steps. The first one is to construct, for the given application, a set of simple regression models to predict the number of faults it contains for the ODC fault categories

of interest. This step requires an initial estimate of the faults number, that the testing engineer in an organization can derive in several ways, such as from previous versions of the software, or from other existing systems in the same product line, or in similar product lines. For the purpose of showing how to construct such models, in this paper we used a set of applications from the technical literature, whose ODC fault types are known. The second step consists of a procedure to characterize testing techniques of interest in terms of ODC fault categories they are more prone to detect. The final result is a prediction of how many faults of each ODC category every technique is able to detect for the given application; this provides the basis for planning testing activities and efforts.

The limitation of the approach consists in the startup cost needed to create the base of knowledge within a context, represented also by the need to achieve a sufficient accuracy of the prediction models before applying the method (and thus ultimately dependent on the number of samples on which the steps are applied). This cost is balanced with time by the iterative application of the method, as typically required by any process change.

As result of applying the approach, the paper provides the additional contribution of extending the body of empirical studies about software metrics and testing, analyzing some techniques widely used in the industrial practice with respect to their adequacy and effectiveness in relation to ODC fault types. Hence, this study can be viewed also as an initial source of information about the relationship between software metrics and ODC fault types, and between the considered testing techniques and the ODC scheme. From this point of view, an aspect that we intend to improved in the near future is about the application of the characterization to a wider set of testing techniques than the ones adopted here.

References

[Arlat et al., 2002] Arlat, J., Fabre, J., Rodriguez, M., Salles, F., 2002. Dependability of COTS microkernel -based systems. IEEE Transactions on Computers. 51(2), 138–163.

[Avritzer and Weyuker, 1999] Avritzer, A, Weyuker, E.J., 1999. Metrics to assess the likelihood of project success based on architecture reviews. Empirical Software Engineering Journal. 4(3), 197–213.

[Avritzer and Weyuker, 1996] Avritzer, A., Weyuker, E.J., 1996. Deriving workloads for performance testing. Software Practice and Experience. 26(6), 613–633.

[Barret et al., 1999] Barret, N., Martin, S., Dislis, C., 1999. Test Process Optimization: Closing the Gap in the Defect Spectrum. In Proceedings of the International Test Conference. pp. 124–129.

[Basili et al., 1996] Basili, V.R., Briand, L.C., Melo, W.L., 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering. 22(10), 751–761.

[Basili and Selby, 1987] Basili, V.R., Selby, R.W., 1987. Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering. SE-13(12), 1278–1296.

[Basili et al., 1999] Basili, V.R., Shull, F., Lanubile, F.,1999. Building Knowledge through Families of Experiments. IEEE Transactions on Software Engineering. 25(4), pp. 456–473.

[Bassin et al., 2001] Bassin, K., Biyani, S., Santhanam, P., 2001. Evaluating the Software Test Strategy for the 2000 Sydney Olympics. Proceedings of the 12th International Symposium on Software Reliability Engineering.

[Beizer, 1995] Beizer, B., 1995. Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley and Sons.

[Bertolino, 2004] Bertolino, A., 2004. Guide to the Knowledge Area of Software Testing. Software Engineering Body of Knowledge. IEEE Computer Society, February. http://www.swebok.org.

[Bhandari et al., 1994] Bhandari, I., Halliday, M.J., Chaar, J., Chillarege, R., Jones K., Atkinson, J.S., Lepori-Costello, C., Jasper, P.Y., Tarver, E.D., Lewis, C.C., Yonezawa, M., 1994. In-process improvement through defect data interpretation. IBM System Journal. 33(1), 182–214.

[Binkley and Schach, 1998] Binkley, A.B., Schach, S., 1998. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. Proceedings of the Intl. Conference on Software Engineering, 452–455.

[Cai et al., 2007] Cai, K., Gu, B., Hu, H., Li, Y., Adaptive software testing with fixed-memory feedback. Journal of Systems and Software 80(8), 1328–1348.

[Catal and Dirl, 2009] Catal, C., Diri, B.m 2009. A systematic review of software fault prediction studies. Elsevier, Expert Systems with Applications. 36 (4).

[Carreira et al., 1995] Carreira, J., Madeira, H., Silva, J.G., 1995. Xception: Software Fault Injection and Monitoring in Processor Functional Units. IEEE Transactions on Software Engineering. 24(2).

[Chidamber and Kemerer, 1994] Chidamber, S.R., Kemerer, C.F., 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. on Software Engineering. 20(6), 476-493.

[Chillarege, 1995] R. Chillarege, 1995. Orthogonal Defect Classification, in Handbook of Software Reliability Engineering, IEEE CS Press, McGraw-Hill, , chapter 9.

[Chillarege et al., 1992] Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.Y., 1992. Orthogonal Defect Classification-A Concept for In-Process Measurements. IEEE Transactions on Software Engineering. 18(11), 943–956.

[Christmansson and Chillarege, 1996] Christmansson, J., Chillarege, R., 1996. Generation of an Error Set that Emulates Software Faults. Proceedings of the 26th IEEE Fault Tolerant Computing Symposium, 304–313.

[Cotroneo et al., 2009] Cotroneo, D., Natella, R., Pecchia, A., Russo, S., 2009. An Approach for Assessing Logs by Software Fault Injection. Supplemental Proceedings IEEE International Conference on Dependable Systems and Networks, A15–A20.

[Dalal et al., 1999] Dalal, S., Hamada, M., Matthews, P., Patton, G., 1999. Using Defect Patterns to Uncover Opportunities for Improvement. Proceedings of the Intl. Conference Applications of Software Measurement.

[Damm et al., 2004] Damm, L.O., Lundberg, L. Wohlin, C.,2004. Determining the Improvement Potential of a Software Development Organization through Fault Analysis: A Method and a Case Study. Proceedings of the 11th International Conference on Software Process Improvement, Lecture Notes in Computer Science 3281, Springer-Verlag, 138–149.

[Damm and Lundberg, 2005] Damm, L.O., Lundberg, L., 2005. Identification of Test Process Improvements by Combining Fault Trigger Classification and Faults-Slip-Through Measurement. In Proceedings of the International Symposium on Empirical Software Engineering. pp. 152–161.

[Do et al., 2005] Do, H., Elbaum, S., Rothermel, G.. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empirical Software Engineering. 10, 4 (October 2005), 405–435.

[Duraes and Madeira, 2006] Duraes, J.A., Madeira, H., 2006. Emulation of Software Faults: A Field Data Study and a Practical Approach. IEEE Transactions on Software Engineering. 32(11), 849–867.

[Duran and Ntafos, 1984] Duran, J.W., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering. 10(7), 438–444.

[Fenton and Pfleeger, 1998] Fenton, N.E., Pfleeger, S.L., 1998. Software Metrics: A Rigorous and Practical Approach, Brooks/Cole.

[Frankl et al., 1997] Frankl, P.G., Weiss, S.N., Hu, C., 1997. All-Uses versus Mutation Testing: An Experimental Comparison of Effectiveness. Journal of Systems and Software 38(3), 235–253.

[Frankl and Weyuker, 1993] Frankl, P.G., Weyuker, E.J., 1993. A formal analysis of the fault detecting ability of testing methods. IEEE Transactions on Software Engineering, 202–213.

[Frankl et al., 1998] Frankl, P.G., Hamlet, R.G., Littlewood, B., Strigini, L., 1998. Evaluating Testing Methods by Delivered Reliability. IEEE Transactions Software Engineering. 24(8), 586–601.

[Gokhale and Lyu, 1997] Gokhale, S., Lyu, M.R.,1997. Regression Tree Modeling for the Prediction of Software Quality. Proceedings of International Conference on Reliability and Quality in Design, 31–36.

[Ghosh et al., 1998] Ghosh, A.K., Schmid, M., Shah, V., 1998. Testing the Robustness of Windows NT Software. Proceedings of the 9th IEEE Intl. Symposium on Software Reliability Engineering, 231–236.

[Gourlay, 1983] Gourlay, J.S., 1983. A mathematical framework for the investigation of testing. IEEE Transactions on Software Engineering, SE-9(6), 686–709.

[Gokhale and Lyu, 1997] Gokhale, S.S., Lyu, M.R., 1997. Regression Tree Modeling for the Prediction of Software Quality. Proceedings of the third ISSAT.

[Grindal et al., 2006] Grindal, M., Lindstrom, B., Offutt, J., Andler, S.F., 2006. An Evaluation of Combination Testing Strategies. Empirical Software Engineering. 11(4), 583–611.

[Grottke et al., 2006] Grottke, M., Li, L., Vaidyanathan, K., Trivedi, K.S., 2006. Analysis of Software Aging in a Web Server. IEEE Transactions on Reliability. 55(3).

[Hamlet, 1989] Hamlet, D., 1989. Theoretical comparison of testing methods. Proceedings of the 3rd Symposium on Testing, Analysis and Verification, 28–37.

[Hamlet and Taylor, 1990] Hamlet, D., Taylor, R., 1990. Partition testing does not inspire confidence. IEEE Transactions on Software Engineering. 16(12), 1402–1411.

[Hiller et al., 1998] Hiller, M., Christmansson, J., Rimèn, M., 1998. An experimental comparison of fault and error injection. Proc. of the 9th IEEE International Symposium on Software Reliability Engineering, 369–378.

[Hocking 1976] Hocking, R.R., 1976. The Analysis and Selection of Variables in Linear Regression. Biometrics. 32(1), 1–49.

[Jia and Harman, 2010] Jia, Y., Harman, M., 2010. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering.

[Jolliffe 2002] Jolliffe, I.T., 2002. Principal Component Analysis, second ed.. Springer.

[Juristo and Moreno, 2001] Juristo, N.J., Moreno, A.M, Basics of software engineering experimentation. Kluwer.

[Kallepalli and Tian, 2001] Kallepalli, C., Tian, J., 2001. Measuring and modeling usage and reliability for statistical Web testing, IEEE Transactions on Software Engineering. 27(11), 1023–1036.

[Kanoun et al., 2005] Kanoun, K., Crouzet, Y., Kalakech, A., Rugina, A., Rumeau, P., 2005. Benchmarking the Dependability of Windows and Linux using PostMark Workloads. Proceedings of the 16th International Symposium on Software Reliability Engineering.

[Koopman and DeVale, 2000] Koopman, P., DeVale, J., 2000. The exception handling effectiveness of POSIX operating systems. IEEE Transactions on Software Engineering. 26(9), 837–848.

[Kropp et al., 1998] Kropp, N.P., Koopman, P., Siewiorek, D.P., 1998. Automated Robustness Testing of Off-the-Shelf Software Components. Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing.

[Laitenberger et al.,2002] Laitenberger, O., Freimut, B., Schlich, M. Combination of Inspection and Testing Technologies. Fraunhofer IESE 2002.

[Lewis, 2008] Lewis, W.E., 2008. Software Testing Continuous Quality Improvement, third edition. Auerbach Publications.

[Littlewood et al., 2000] Littlewood, B., Popov P., Strigini L., Shryane, N., 2000. Modelling the Effects of Combining Diverse Software Fault Detection Techniques. IEEE Transactions on Software Engineering. 26(12), 1157–1167.

[Luo et al., 2010] Luo, Y., Ben, K., Mi, L., 2010. Software metrics reduction for fault-proneness prediction of software modules. Proc. of the IFIP international conference on Network and parallel computing, in Chen Ding, Zhiyuan Shao, and Ran Zheng (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 432–441.

[Musuvathi and Engler, 2003] Musuvathi, M., Engler, D., 2003. Some Lessons from Using Static Analysis and Software Model Checking for Bug Finding. Electronic Notes in Theoretical Computer Science, 89, 378–404.

[Matias and Filho, 2006] Matias, R., Filho, P.J., 2006. An Experimental Study on Software Aging and Rejuvenation in Web Servers. Proceedings of the 30th Intl. Computer Software and Applications Conference. 189–196.

[Montgomery, 2001] Montgomery, D.C., Design and Analysis of Experiments, fifth ed. John Wiley & Sons Inc.

[Nagappan et al., 2006] Nagappan, N., Ball, T., Zeller, A., 2006. Mining Metrics to Predict Component Failures. Proceedings of the 28th international conference on Software engineering, 452–461.

[Ohlsson and Alberg, 1996] Ohlsson, N., Alberg, H., 1996. Predicting fault-prone software modules in telephone switches. IEEE Transactions on Software Engineering. 22(12), 886–894.

[Ostrand et al., 2005] Ostrand, T., Weyuker, E.J., Bell, R., 2005. Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering. 31 (4), 340–355.

[Perry and Stieg, 1993] Perry, D.E., Stieg, C., 1993. Software faults in evolving a large, real-time system: a case study. Proceedings of the European Software Engineering Conference, Lecture Notes in Computer Science, vol. 717, 48–67.

[Pezze and Young, 2007] Pezze, M., Young, M., 2007. Software Testing and Analysis: Process, Principles and Techniques. John Wiley & Sons Inc.

[Pietrantuono et al., 2010] Pietrantuono, R., Russo, S., Trivedi, K.S., 2010. Software Reliability and Testing Time Allocation: An Architecture-Based Approach. IEEE Transactions on Software Engineering. 36(3), 323–337.

[Pietrantuono 2009] Pietrantuono, R., 2009. Reliability-oriented Verification of Missione-critical Software Systems. PhD. Thesis. Univerista' degli Studi di Napoli Federico II.

[Poulding and Clark, 2010] Poulding, S., Clark, J.A., 2010. Efficient Software Verification: Statistical Testing Using Automated Search. IEEE Transactions on Software Engineering. 36(6), 763–777.

[Rapps and Weyuker, 1985] Rapps, S., Weyuker, E.J., 1985. Selecting software test data using data flow information. IEEE Transactions on Software Engineering. SE-14(4), 367–375.

[Riebisch et al., 2003] Riebisch, M., Philippow, I., Gotze, M., 2003. UML-Based Statistical Test Case Generation. Objects, Components, Architectures, Services, and Applications for a Networked World. 2591, 394–411.

[Selby 1986] R. W. Selby, 2006. Combining Software Testing Strategies: An Empirical Evaluation. In Proceedings of the ACM/SIGSOFT IEEE Workshop on Software Testing, 82–90.

[Subramanyam and Krishnan, 2003] Subramanyam, R., Krishnan, M.S, 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Trans. on Software Engineering. 29(4), 297–310.

[Thevenod-Fosse et al., 1991] Thevenod-Fosse, P., Waeselynck, H., Crouzet, Y., 1991. An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation. IEEE Fault-Tolerant Computing: the 21st International Symposium, 410–417.

[Thevenod-Fosse, 1991] Thevenod-Fosse, P.,1991. Software Validation by Means of Statistical Testing: Retrospect and Future Direction, in: Avizienis, A., Laprie, J.C. (Eds.), Dependable Computing for Critical Applications. Springer, 23–50.

[Thevenod-Fosse and Waeselync, 1991] P. Thevenod-Fosse, H. Waeselynck, 1991. An Investigation of Statistical Software Testing. Software Testing, Verification and Reliability, 1(2), 5–26.

[Trammell, 1995] Trammell, C., 1995. Quantifying the Reliability of Software: Statistical Testing Based on a Usage Model. Proceedings of the 2nd IEEE International Software Engineering Standards Symposium, 208–218.

[Vegas and Basili, 2005] Vegas, S., Basili, V., 2005. A Characterization Schema for Software Testing Techniques. Empirical Software Engineering. 10, 437–466.

[Wagner, 2005] Wagner, S., 2005. Software Quality Economics for Combining Defect-Detection Techniques. Proceedings of NET.OBJECT DAYS 2005 (NODE'05), 559–574.

[Wang et al., 2011] H. Wang, T.M. Khoshgoftaar, N. Seliya, 2011. How Many Software Metrics Should be Selected for Defect Prediction? Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference.

[Weyuker, 2008] Weyuker, E., 2008. Comparing the Effectiveness of Testing Techniques. Formal Methods and Testing, 271–291.

[Weyuker and Jeng, 1991] Weyuker, E.J., Jeng, B., 1991. Analyzing partition testing strategies. IEEE Transactions on Software Engineering. 17(7), 703–711.

[Weyuker et al., 1991] Weyuker, E.J., Weiss, S.N., Hamlet, D., 1991. Comparison of program testing strategies. Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification, 1–10.

[Whittaker and Thomason, 1994] J.A., Whittaker, M.G. Thomason, 1994. A Markov chain model for statistical software testing. IEEE Transactions on Software Engineering. 20(10).

[Wojcicki and Strooper, 2007] Wojcicki, M.A., Strooper, P., 2007. An Iterative Empirical Strategy for the Systematic Selection of a Combination of Verification and Validation Technologies. Proceedings of the fifth International Workshop on Software Quality. IEEE Computer Society.

[Wood et al., 1997] Wood, M., Roper, M., Brooks, A., Miller, J., 1997. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. In Proceedings of the 6th European Software Engineering Conference. pp. 262–277.