# Analysis and Prediction of Mandelbugs in an Industrial Software System

Gabriella Carrozza*, Domenico Cotroneo†, Roberto Natella†, Roberto Pietrantuono†, Stefano Russo†
*Consorzio SESM scarl, Via Circumvallazione Esterna di Napoli, 80014, Naples, Italy
†Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Naples, Italy
Email: gcarrozza@sesm.it, {cotroneo, roberto.natella, roberto.pietrantuono, sterusso}@unina.it

***Abstract*—Mandelbugs are faults that are triggered by complex conditions, such as interaction with hardware and other software, and timing or ordering of events. These faults are considerably difficult to detect with traditional testing techniques, since it can be challenging to control their complex triggering conditions in a testing environment. Therefore, it is necessary to adopt specific verification and/or fault-tolerance strategies for dealing with them in a cost-effective way.**

**In this paper, we investigate how to predict the location of Mandelbugs in complex software systems, in order to focus V&V activities and fault tolerance mechanisms in those modules where Mandelbugs are most likely present. In the context of an industrial complex software system, we empirically analyze Mandelbugs, and investigate an approach for Mandelbug prediction based on a set of novel software complexity metrics. Results show that Mandelbugs account for a noticeable share of faults, and that the proposed approach can predict Mandelbug-prone modules with greater accuracy than the sole adoption of traditional software metrics.**

***Keywords*-Fault prediction; Fault tolerance; Mandelbugs; Software metrics.**

## I. Introduction

Mission-critical software systems, i.e., systems whose failure may lead to severe loss in terms of cost or safety, have strict dependability requirements. In these systems, testing activities are fundamental to detect *software faults* (also referred to as *defects* or *bugs*) and to reduce the probability of failures at operational time. However, there are faults that are considerably more difficult to detect and that tend to elude testing, such as: faults related to interactions with the environment (e.g., hardware devices and other software systems), faults related to the timing of events and to process scheduling, resource leaks and data corruption. These kinds of faults are often referred to as *Mandelbugs* in the literature, due to the complexity of the conditions that cause their manifestation [1]. According to several studies, these faults have a significant impact in terms of software failures and costs [2], [3], [4], [5].

Analyzing Mandelbugs in complex software systems, and predicting where these bugs are located in the system, is of practical interest for dealing with them during development and testing. Experience has shown that such faults are difficult to spot with traditional testing techniques, since it can be challenging to control their complex triggering conditions in a testing environment. Therefore, Mandelbugs require specific strategies. Examples are fault tolerance strategies that mask faults, for instance by reinitializing the software state and retrying the failed operation [6], [7], [8], [9], and verification techniques able to spot hard-to-trigger faults, such as code reviews and model checking [10], [11], [12], in addition to traditional dynamic testing. Since these strategies can significantly increase the cost of development and testing, it is necessary, in order to be cost-effective, to focus them on the most problematic parts of the system. Planning V&V efforts and fault-tolerance according to Mandelbugs prediction can reduce the cost of applying such strategies in large and complex software systems. However, to the best of our knowledge, previous studies on fault prediction in complex software [13], [14], [15] did not consider the problem of predicting Mandelbugs.

In this paper, we investigate how to predict the location of Mandelbugs in complex software systems. The main contributions are: *i)* an empirical analysis of Mandelbugs in an industrial mission-critical software system, *ii)* an approach for predicting Mandelbugs in such systems, based on a set of novel software complexity metrics. The considered system is an industrial software system consisting of about 1.3M lines of code and 23 software modules, and developed by Selex-SI, a Finmeccanica company producing mission-critical systems for several domains (such as avionic, naval, defense). Results shows that Mandelbugs account for a noticeable share of faults, and that the proposed approach can predict Mandelbug-prone modules with greater accuracy than the sole adoption of traditional software metrics.

The paper is organized as follows. Section II provides background and related work on the characterization of Mandelbugs and on fault prediction. Section III describes the industrial case study of this work. Section IV analyzes the type and the distribution of bugs in the case study. Section V describes and evaluates the proposed fault prediction approach. Sections VI and VII conclude the paper.

## II. Background and related work

### A. Mandelbugs

The study by Gray in 1984 [3] was the first work in the published literature pointing out that often the nature of software failures is "transient". In analogy to the Heisenberg uncertainty principle, faults behind transient failures are referred to as *Heisenbugs*, since they do not manifest themselves when trying to debug them, due to perturbations introduced by debugger (e.g., initialization of unused memory, influence on CPU scheduling and timing of events).

Conversely, non-transient bugs are referred to as *Bohrbugs* (in analogy to the Bohr atom model), since they are easy to diagnose once detected.

Subsequent studies further analyzed the extent and the features of bugs according to this view, but adopting different terminologies with slightly different meanings, such as *environment-dependent* vs. *environment-independent* [16] and *deterministic* vs. *non-deterministc* bugs [17], or by focusing on *concurrency bugs* [18], [19] as the class of faults causing transient failures. Performing such studies has proven to be a daunting task, since data about transient failures is hard to collect by their nature.

In a recent taxonomy of software faults [1], [5], Heisenbugs are included in the more general class of *Mandelbugs*, where the former are the bugs that change their behavior when probed using a debugger, and the latter include all the bugs whose activation condition is related to timing and to complex interactions with the system state as a whole, including hardware, OS and other software such as middleware, virtual machines, libraries and remote services. Mandelbugs include the subclass of *aging-related bugs*, that are responsible for a phenomenon increasingly being studied, known as *software aging* [4]. Software aging typically causes an increasing failure rate and/or degraded performance in long-running software systems. This can be due, for instance, to the accumulation of errors in the system state and to resource leaks such as physical memory.

Notwithstanding the difficulties in finding evidences of Mandelbugs, field data studies provided some evidence that Mandelbugs account for a significant part of bugs in complex software. The analysis of field failures in Tandem systems [3], [20] showed that most software failures were transient. More recent analyses found that although Bohrbugs represent the majority of software faults, Mandelbugs account for a significant share (in the 20-40% range) in open-source [16] and in NASA space software [5], [21]. The study of Mandelbugs has been exploited in [9] to analyze software recovery strategies, and in [22] to analyze how testing for aging-related bugs affects availability during operation.

### B. Fault prediction

Fault prediction aims to identify which modules (e.g., files or functions) in a given software system are more prone to software faults, in order to support the planning of testing and maintenance activities. To this aim, a set of measurable attributes, namely *software metrics*, is extracted for each module (from its source code or from its development process), in order to relate them to the presence of faults in the module. First, metrics are collected *from previous projects or previous versions of the current project*, along with information on bugs found in each module, in order to train a *fault prediction model*. Then, metrics are collected from the system under analysis, in order to identify fault-prone modules using the learned fault prediction model.

Early research in this area was focused on the definition of metrics able to measure the complexity of a software module and, in turn, its likelihood to be faulty (such as the McCabe's and Halstead's metrics) [23], [24]. Fault prediction approaches have then evolved by adopting *machine learning* and *data mining* algorithms and techniques, in order to establish a more accurate relationship between sets of software metrics and faults, using *classifiers* and *regression models* [13], [25], [26]; these approaches are surveyed in [27]. Subsequent studies confirmed the feasibility and effectiveness of fault prediction using public-domain datasets from real-world projects, such as the NASA Metrics Data Program, and using several regression and classifiers [14], [28]. Other recent studies were on the definition of software metrics collected from early lifecycle data such as textual requirements [29], and on transferring prediction models across different projects and companies [30], [31]. However, to the best of our knowledge, only few studies considered the problem of discriminating between fault types in fault prediction [32], [33], and none of them took into account the difference between Mandelbugs and Bohrbugs, for which different V&V and fault tolerance strategies are needed.

## III. System overview

In this work, we analyze an industrial software system developed for the military domain. The goal of the system is to support the EU Military Staff (EUMS) in the task of rapidly set-up an *Operations Centre* (OPC) where a joint civil/military response is required, and where no national head quarters can be identified, once a decision on such an operation has been taken. The system supports command-level activities and cross-command level activities of the EU OPC in conducting civilian and military operations on behalf of the European Union. More specifically, the system has been designed to: *i)* enable and improve command and control of military and/or civilian organizations; *ii)* support operational planning undertaken by the OPC; *iii)* improve the information exchange; *iv)* increase Situational Awareness; *v)* be the tool to improve collaborative efforts between planning, execution, assessment and maintenance of *Situational Awareness*; *vi)* enable the OPC to deliver timely and informed decisions to the deployed forces, to receive reports from them, to produce and send reports to higher levels. Additionally, the system acts as support tool for preplanning activities when the OPC is not activated.

The system is developed by outsourcing the implementation of modules to external providers, then integrating them at Selex-SI. Testing is performed jointly with SESM on single modules as well as on their integration. Results of testing are reported back to providers in the form of problem reports. The system is made up of about 1.3 *MLoC* across 12 components; in turn, some components include sub-components. In our analysis, we handle a component and its sub-components as different entities (i.e., the code of a component not included in any sub-component is considered a distinct entity), and we refer to both components and sub-components as *modules*. In total, the system consists of 23 modules. We discarded 4 modules from the analysis, as information about them was not available to us.

## IV. Fault classification

This section focuses on the classification and analysis of problem reports of the case study. We describe in detail the criteria adopted for fault classification, and we analyze the distribution of faults across fault types and modules.

### A. *Fault types*

In order to examine faults in a rigorous way, we classified them among two classes according to the definitions by Grottke *et al.* [5]. They concern the conditions related to the fault activation and the error propagation:

- **Mandelbug**: a bug whose activation and/or error propagation are "complex" where complexity is caused by
  1) a time lag between the fault activation and the failure occurrence, or
  2) the possible influence of indirect factors:
     a) interactions of the software application with its *system-internal environment* (hardware, operating system, or other applications);
     b) timing of inputs, events, and operations (relative to each other, or in terms of the system runtime or calendar time);
     c) sequencing of inputs, events and operations (the inputs could have been run in a different order, and at least one of the other orders would not have led to a failure).
- **Bohrbug**: a bug which can easily be isolated and which manifests consistently under a well-defined set of conditions, because its activation and error propagation lack "complexity" as defined before.

In most cases, the complexity of the triggering conditions of Mandelbugs makes them difficult to isolate, and significantly increases the efforts for systematically reproducing the failures caused by these faults [3], [20]. Mandelbug is the complementary antonym of Bohrbug (i.e., each fault is either a Mandelbug, or a Bohrbug). Aging-related bugs are a subtype of Mandelbugs (i.e., aging-related bugs is always considered a Mandelbug), since they exhibit a time lag between their activation and the occurrence of a failure.

It is important to note that these definitions do not focus on the circumstances of one specific manifestation of the bug (e.g., the one that made the testers notice its presence, or that helped them locate it in the code), but rather on its *potential manifestation characteristics* and its inherent features. For example, even if a developer is able to reproduce a failure in a well-controlled environment, the underlying fault is classified as a Mandelbug if its manifestation can result in a transient failure at the user site because one of the criteria of complex fault activation or error propagation applies.

### B. *Classification procedure*

The bugs analyzed in this study were obtained by inspecting problem reports provided by the V&V team responsible for the case study system. Data were collected by testers using the *Mantis* bug tracker, during the integration testing phase. Each problem report included a *summary* and a *detailed textual description* of the software problem, the *module* affected by the problem, the *type* of problem (e.g., bug or feature request), an indication of whether the problem has been successfully *reproduced*, and the *steps to reproduce* the problem (if available).

Given a problem report, in order to classify the related bugs into Bohrbugs and Mandelbugs, we conducted a manual analysis by examining the textual descriptions and, where available, the steps to reproduce the failure occurrences, and additional information attached to the problem reports (e.g., system logs and screenshots). In order to classify faults in a rigorous way, we defined a classification procedure consisting of the following steps:

1) Each problem report was first examined to make sure that it was related to an actual bug. We removed from the analysis those problem reports turning out to be requests for software enhancements, reports related to the aesthetic of user interfaces or to the online help of the system, and problems rejected by the development team as out of the scope of system requirements. Problem reports not related to bugs were flagged as not classifiable (NOC) and were not further analyzed.
2) We then looked in the report for any information on the activation conditions of the bug (e.g., the set of events and/or inputs required to trigger the fault), its error propagation (e.g., how the bug affected the program state and how error states propagated through the system), and the failure behavior (e.g., the effects perceived by the users). Problem reports related to the same fault were grouped and analyzed as a unique bug (e.g., the textual description indicated that the reported problem was caused by the same underlying bug as another report already included in our study).
3) A bug was classified as a Mandelbug (MAN) if we found indications that one of the types of "complexity" of the activation and/or error propagation applied to it, among the ones included in the definition of Mandelbugs: (i) there is a time lag between fault activation and the failure occurrence; (ii) the bug requires an interaction with the system-internal environment in order to manifest itself; (iii) the timing and (iv) sequencing of inputs/events/operations have influence on the fault activation and/or error propagation.
4) If there was evidence that the "complexity" criteria above are not applicable to the bug (i.e., it is not a

Mandelbug), we classified it as a Bohrbug (BOH).

5) Sometimes, a report did not contain sufficient details to classify the underlying bug. It was then assigned to the class of bugs of unknown type (UNK).

The classification of reports was in a first round carried out by an intern at SESM, using the criteria described above. In a second round, three of the authors independently reviewed the classification. Discordant results among the three authors and doubtful cases were discussed among authors. When an agreement was not reached or not enough information could be obtained from the V&V team, the report was classified as UNK.

### C. Analysis of bugs

The problem reports considered in this analysis were collected between July 2011 and January 2012. In total, 463 problem reports were inspected. Among these reports, 202 of them (43.63%) turned out not to be actual bugs, and were flagged as NOC. The remaining 261 problem reports were classified into Bohrbugs and Mandelbugs, accounting respectively for 78.93% and 14.56% of bugs, as depicted in Figure 1. The percentage of Mandelbugs is noticeable, although it is smaller than other projects reported in the literature (their percentage is between 20% and 40% in NASA space projects [5], [21]), since the project considered in this study has been analyzed during its integration testing phase: Bohrbugs manifest themselves under less complex conditions than Mandelbugs, and it is reasonable to expect that a high number of Bohrbugs are detected and fixed during testing activities. As observed in [22], the analysis of bug data found during testing can be exploited to identify the best trade-off between verification and fault tolerance strategies against Mandelbugs in terms of availability.

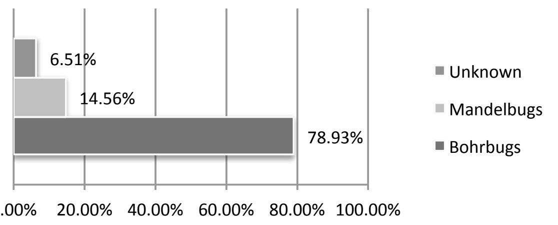

Figure 1. Distribution of problem reports into Bohrbugs and Mandelbugs.

The distributions of both Bohrbugs and Mandelbugs across modules is shown in Figure 2. In some cases (23 Bohrbugs and 5 Mandelbugs), problem reports involved more than one module; in these cases, we counted one fault for each module affected by the problem. It can be seen that the distributions of both types is uneven across modules, that is, there are modules that are more faulty than others. In particular, there are modules having a relatively high number of Mandelbugs, such as C4, C6 and C7, while at most 1 Mandelbug was found for several modules. This situation has been observed in many studies on complex software systems, where few software modules account for most faults and failures (it is often informally referred to as "20-80 rule") [2], [13], [25]. This fact motivates the adoption of fault prediction strategies for identifying on which modules testing should focus [13], [14]. A good testing strategy is to test these modules first and with greatest emphasis, in order to get the greatest payoff from V&V activities. Moreover, predicting such Mandelbug-prone modules allows to plan specifically tailored testing and/or fault-tolerance strategies.

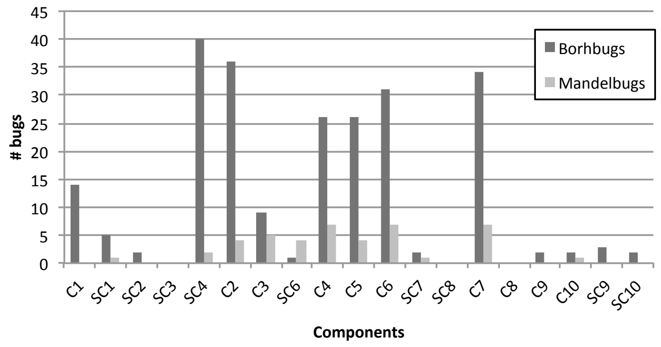

Figure 2. Distribution of Bohrbugs and Mandelbugs across modules.

## V. Fault prediction

To predict Mandelbug-prone modules, there are two main aspects that need to be taken into account: the definition of the *software metrics* (to characterize the features of a software module and relate them to faults) and the *fault prediction algorithm* to adopt (to establish such relationship in a quantitative way). In the following, we first discuss the proposed software metrics and fault prediction algorithms. We then evaluate prediction on the considered case study.

### A. Software complexity metrics

A software module can be characterized by using both product-oriented and process-oriented metrics (Section II-B). In this study, we focus on product-oriented metrics, since they have a wider applicability than process-oriented ones, which may not be always available. This is the case of the considered case study (Section III), in which process-oriented metrics are not available as the modules of the system are developed by third-party organizations. We adopted a set of software metrics that are well-known and commonly used by researchers and practitioners, such as McCabe's and Halstead's metrics. Software metrics were automatically extracted by using the Understand tool for static analysis, and are listed in Table I (more details about the definition of these metrics can be found in [34]).

In addition to traditional software metrics, we introduce a set of novel metrics that we expect to be related with the presence of Mandelbugs in the code. These metrics focus on the indirect factors that cause the activation and manifestation of Mandelbugs (according to the definition of Mandelbugs discussed in Subsection IV-A):

Table I
TRADITIONAL SOFTWARE METRICS USED FOR BUG PREDICTION.

| Type | Metrics | Description |
|---|---|---|
| *Program size* | *AvgLine*, *AvgLineBlank*, *AvgLineCode*, *AvgLineComment*, *CountDeclFunction*, *CountInput*, *CountLine*, *CountLineBlank*, *CountLineCode*, *CountLineCodeDecl*, *CountLineCodeExe*, *CountLineComment*, *CountOutput*, *CountPath*, *CountSemicolon*, *CountStmt*, *CountStmtDecl*, *CountStmtExe*, *RatioCommentToCode* | Metrics related to the amount of lines of code, declarations, statements, and files |
| *McCabe's cyclomatic complexity* | *AvgCyclomatic*, *AvgCyclomaticModified*, *AvgCyclomaticStrict*, *AvgEssential*, *Cyclomatic*, *CyclomaticModified*, *CyclomaticStrict*, *Essential*, *MaxCyclomatic*, *MaxCyclomaticModified*, *MaxCyclomaticStrict*, *MaxNesting*, *SumCyclomatic*, *SumCyclomaticModified*, *SumCyclomaticStrict*, *SumEssential* | Metrics related to the control flow graph of functions and methods |
| *Halstead metrics* | *Program Volume*, *Program Length*, *Program Vocabulary*, *Program Difficulty*, *Effort*, *N1*, *N2*, *n1*, *n2* | Metrics based on operands and operators |
| *Object-oriented metrics* | *CountClassBase*, *CountClassCoupled*, *CountClassDerived*, *CountDeclClass*, *CountDeclClassMethod*, *CountDeclClassVariable*, *CountDeclInstanceMethod*, *CountDeclInstanceVariable*, *CountDeclMethod*, *CountDeclMethodAll*, *CountDeclMethodDefault*, *CountDeclMethodPrivate*, *CountDeclMethodProtected*, *CountDeclMethodPublic*, *MaxInheritanceTree*, *PercentLackOfCohesion* | Metrics based on object-oriented constructs |

- **Concurrency**: In most cases, concurrency-related bugs manifest themselves only under a specific thread schedule (e.g., two or more threads are scheduled such that they are interleaved during the execution of some critical sections lacking proper synchronization). Therefore, this kind of bugs are triggered by indirect factors and can lead to transient failures. Since concurrency-intensive programs can be more prone to this kind of faults, we collect metrics based on concurrency-related constructs, such as the number of synchronized methods and blocks in a class.
- **I/O**: Some Mandelbugs are triggered by interactions with the system-internal environment, which includes the hardware and other software. Since these interactions pass through I/O interfaces of the system, and since incorrect interactions increase the likelihood of Mandelbugs, we collect metrics based on the usage of I/O libraries, such as the *java.io* and *java.net* packages.
- **Exception handling**: The use of exception handling code in a program is aimed to deal with events (e.g., external errors) that may occur at runtime. Faults in exception handling code are difficult to spot and to debug, since they are triggered by rare events and by interactions with the system-internal environment. Therefore, we collect metrics based on the usage of exception handling constructs, such as *try-catch* blocks.

The proposed metrics are summarized in Table II. Metrics are tailored to the Java language, which has been adopted in the case study system, although they can potentially be extended to other programming languages such as C++. These metrics are computed by inspecting the source code and looking for keywords that denote the usage of specific programming constructs (e.g., synchronization constructs) and libraries. Metrics related to *import* directives (*ImportIO*, *ImportNet*, and *ImportConcurrent*) are computed for each source file, while the remaining metrics are computed for each class in the module; then, the values of metrics for each class or file are averaged in order to obtain an individual value for the whole module. The result of this process is a vector of metrics, where each value denotes the average value of a metric across files or classes in the module. The same process is adopted for collecting traditional metrics.

Table II
PROPOSED SOFTWARE METRICS.

| Name | Description | Domain |
|---|---|---|
| *CountTryBlocks* | Number of *try* blocks | $\mathbb{N}$ |
| *AtLeastOneTryBlock* | At least one *try* block is present in the class | $\{0,1\}$ |
| *CountCatchBlocks* | Number of *catch* blocks | $\mathbb{N}$ |
| *StaticLibraryMethodCalls* | Number of calls to static methods of the *System* package and of the *Runtime.getRuntime()* object | $\mathbb{N}$ |
| *AtLeastOneStaticLibraryMethodCall* | At least one call according to *StaticLibraryMethodCalls* is present in the class | $\{0,1\}$ |
| *UniqueStaticLibraryMethodCalls* | Number of calls according to *StaticLibraryMethodCalls*, where calls to the same method are counted only one time | $\mathbb{N}$ |
| *LibraryMethodCalls* | Number of calls to methods of the *java.io* and *java.net* packages | $\mathbb{N}$ |
| *AtLeastOneLibraryMethodCall* | At least one call according to *LibraryMethodCalls* is present in the class | $\{0,1\}$ |
| *UniqueLibraryMethodCalls* | Number of calls according to *LibraryMethodCalls*, where calls to the same method are counted only one time | $\mathbb{N}$ |
| *ClassThread* | The class is a descendant of the *Thread* class, or it implements the *Runnable* interface | $\{0,1\}$ |
| *CountSynchronizedMethods* | Number of synchronized methods | $\mathbb{N}$ |
| *CountSynchronizedBlocks* | Number of synchronized blocks | $\mathbb{N}$ |
| *CountLockCalls* | Number of calls to instances of the *Lock*, *Semaphore*, *ReentrantLock* classes | $\mathbb{N}$ |
| *CountWaitNotify* | Number of calls to *wait()*, *notify()*, *notifyAll()*, *await()*, *signal()*, *signalAll()* | $\mathbb{N}$ |
| *ImportIO* | The file imports the *java.io* package | $\{0,1\}$ |
| *ImportNet* | The file imports the *java.net* package | $\{0,1\}$ |
| *ImportConcurrent* | The file imports the *java.concurrent.util* package | $\{0,1\}$ |

## B. Fault prediction algorithms

Since the relationship between software metrics and (Mandel)bugs is not self-evident and may depend on the specific system under analysis, we adopt machine learning algorithms to infer this relationship, which are widely applied to knowledge discovery problems in industry and science [35]. In this context, a data sample is represented by a module in which bugs may reside. In particular, we consider two approaches for fault prediction, respectively *classification* and *regression*.

In the case of *classification*, the aim of the fault predictor is to divide modules among a finite set of mutually exclusive classes. In the first approach that we consider in this study, modules are classified between two classes, namely "Mandelbug-prone modules" (i.e., modules with at least one Mandelbug) and "Mandelbug-free modules" (i.e., modules without Mandelbugs). If "Mandelbug-prone" are correctly identified, the V&V team could exploit classification by focusing verification efforts on modules classified as "Mandelbug-prone". However, other choices are possible. We also consider the case in which modules are classified among three classes, namely "no Mandelbugs", "exactly one Mandelbug", and "more than one Mandelbug". In this scenario, the V&V team would have more detailed information for planning verification activities, since more efforts could be devoted to "modules with more than one Mandelbug" compared to "modules with exactly one Mandelbug". To perform classification, a *classification algorithm* learns a predictive model from known data samples (e.g., data from previous projects or previous releases of the system), which is then adopted to classify unknown samples (e.g., modules under development, for which V&V is being planned). There exist several classification algorithms, which make different assumptions about the underlying data that can affect the effectiveness of fault prediction. Therefore, we consider several algorithms in this study, which are well-known and often adopted for prediction purposes [15]:

- *Decision Trees*;
- *Support Vector Machines* (SVM);
- *Bayesian Networks*;
- *Naive Bayes*;
- *Multinomial Logistic Regression*.

In the case of *regression*, the fault predictor ranks modules with respect to the (expected) number of Mandelbugs that are present in each module. Since verification activities are limited by time and budget constraints, testers could focus their efforts on top-ranked modules: for instance, if testers expect that most of Mandelbugs are located in 20% of modules (according to the "20-80" rule of thumb), then V&V can be focused on the top-20% modules according to the ranking produced by a regression model. In a similar way to classification, a regression model is learned from known data samples, which is then adopted to rank a set of unknown samples. We consider several regression models:

- *Linear Regression*;
- *Regression Trees*;
- *Support Vector Regression* (SVR).

When adopting machine learning for prediction purposes, it is important to pay attention to which attributes (i.e., software metrics in our context) are adopted to analyze data. In fact, using many attributes does not necessarily improve the effectiveness of prediction, and, in some cases, using too many attributes can even reduce performance since attributes tend to be correlated and redundant (this effect is referred to as *multicollinearity*) [26]. Therefore, we include during fault prediction a *feature selection* pre-processing step, where a subset of software metrics is selected before training a classifier or regressor. We adopt a *stepwise regression* algorithm for selecting features [36]: at each iteration, the algorithm evaluates a candidate subset of features with respect to the goodness-of-fit of a multiple linear regression model, and it adds and/or removes a feature from the subset until the subset cannot be further improved. We consider both the case in which a subset of features is selected before prediction, and the case in which feature selection is not performed.

## C. Evaluation

***Prediction by classification.*** We first evaluate the effectiveness of fault prediction based on classification. We evaluate the ability of a classifier to correctly predict fault-proneness of unforeseen data instances, by training a classification model using some of the available data (*training set*), and by using it to classify the remaining data (*test set*). The evaluation has been performed using the *$k$-fold cross-validation* approach [14], [26], with $k = 3$; cross-validation has been repeated 15 times and results have been averaged.

For each data sample in the test set, the predicted class is compared with the actual class of the sample. Given a target class (e.g., the class of "Mandelbug-prone" modules), we compute the following set of performance indicators, that are commonly adopted in machine learning studies [35]:

- **Precision (Pr)**: Percentage of *true positives* (TP) among modules that are classified as belonging to the target class (*true positives* and *false positives* (FP)):
$$Precision = TP/(TP + FP)\ . \qquad (1)$$
- **Recall (Re)**: Percentage of *true positives* among modules that actually belong to the target class (*true positives* and *false negatives* (FN)):
$$Recall = TP/(TP + FN)\ . \qquad (2)$$
- **F-measure (F)**: Harmonic mean of *precision* and *recall*:
$$F - measure = (2 \cdot Pr \cdot Re)/(Pr + Re)\ . \qquad (3)$$

The higher the precision and the recall (ideally, $Pr = Re = 1$), the higher the quality of the predictor, since it avoids false positives (e.g., V&V efforts are not wasted on Mandelbug-free modules) and false negatives (e.g., all Mandelbug-prone modules are identified). Improving precision, i.e., reducing the number of false positives, often results in worse recall, i.e., increasing the number of false negatives, at the same time. Therefore, the F-Measure is often adopted to evaluate the trade-off between precision and recall.

Table IV provides the results of cross-validation in the case of 3-classes classification, in which modules are classified among the "0 Mandelbugs", "1 Mandelbug", and ">1 Mandelbugs" classes. The columns of the table provide, for each class, the precision, the recall and the F-measure for that class. The rows provide the values of these measures for each classification algorithm. Each algorithm is evaluated in 4 cases: (i) only traditional metrics are used, without feature selection; (ii) only traditional metrics are used, with feature selection; (iii) both traditional and proposed metrics are used, without feature selection; (iv) both traditional and proposed metrics are used, with feature selection. To simplify the comparison of algorithms, the table provides, in the last columns, the average for each measure weighted by the percentage of samples in each class. The set of metrics selected by feature selection are showed in Table III, along with the $R^2$ and *Adj*-$R^2$ measures indicating the goodness-of-fit of a multiple linear regressor using these features [26].

Table III
FEATURE SELECTION WITH RESPECT TO 3-CLASSES CLASSIFICATION.

| **Metrics** | **Selected features** | $R^2$ | **Adj**-$R^2$ |
|---|---|---|---|
| Traditional metrics | *CountDeclMethodAll*, *CountClassCoupled*, *CountDeclClassVariable*, *RatioCommentToCode*, *CountLineBlank*, *CountLineCodeDecl* | 0.8784 | 0.7973 |
| Traditional and proposed metrics | *CountClassDerived*, *UniqueLibraryMethodCalls*, *CountSynchronizedBlocks*, *CountClassCoupled*, *CountTryBlocks*, *CountDeclClassVariable* | 0.8857 | 0.8095 |

It can be seen that, for the classes "0 Mandelbugs" and ">1 Mandelbugs", the measures are between 0.6 and 0.9 in most of the cases, that is, the classifiers are quite effective at identifying modules belonging to these classes. Therefore, classifiers are useful to identify the modules that are least and most prone to Mandelbugs, respectively. Instead, for the "1 Mandelbug" class, the values are much lower: classifiers in most cases make mistakes when they classify modules in this class. This result may be due to the fact that modules in the "1 Mandelbug" class are often confused with modules in the "0 Mandelbugs" and in the ">1 Mandelbugs" since they exhibit features that are close to both these classes.

We evaluated whether feature selection and the proposed metrics can actually provide benefits to fault prediction, by comparing the weighted average of F-measures (last column of Table IV) of the different combinations of classifiers and features. We performed a *hypothesis test* to evaluate whether the differences between the best classifier and the other ones are *statistically significant*, i.e., the differences are unlikely due to random errors. According to [37], we adopted the non-parametric *Wilcoxon signed-rank test* procedure, which tests the *null hypothesis* that the differences between two sets of measures have null median, and which can be used to compare the performance of different classifiers [38]. For each classifier and feature set (i.e., a row of Table IV), we collected the averaged F-measure for each "test" fold of each cross-validation repetition, thus obtaining a distribution of values for each classifier and feature set, that we compared using the Wilcoxon signed-rank test. Since multiple comparisons are being performed, we adjusted p-values using the Benjamini-Hochberg procedure [39], which allows controlling the false rejection probability while retaining a high power of the tests.

In the last column of Table IV, we highlight in **bold** the best results (according to the Wilcoxon signed-rank test, with a $90\%$ confidence level) for each combination of feature selection and metrics (i.e., first, we compare algorithms when using neither feature selection nor novel metrics; we then compare algorithms when only feature selection is used; and so on). For some combinations, more than one classifier may be the best one (i.e., there are not statistically significant differences). Moreover, we compared (using the Wilcoxon signed-rank test) the best algorithms from different combinations, in order to identify if feature selection and/or the proposed metrics actually improve the F-measure of classification (the best overall F-measure is **underlined** in the table). We found that the F-measure is higher when using feature selection (respectively, Bayes Networks when only traditional metrics are adopted, and SVM and Naive Bayes when both traditional and novel metrics are adopted), which points out the presence of redundant features and the need for feature selection. Moreover, the proposed metrics provide an additional gain, since the best results are obtained when both feature selection and the novel metrics are adopted. This highlights that *the presence of constructs related to concurrency, I/O and exception handling can provide hints about the occurrence of Mandelbugs*.

In a similar way, in Tables V and VI we provide the results of cross-validation in the case of 2-classes classification, in which modules are classified as "Mandelbug-prone" or as "Mandelbug-free". For both classes, measures are between 0.6 and 1 in almost all cases. In this scenario, we do not try to identify modules that have exactly 1 Mandelbug, thus avoiding the low performance for the "1 Mandelbug" class; this choice leads to a performance improvement of classification for the other two classes. *The prediction of Mandelbug-proneness seems to be the most suitable strategy for practical applications*, since it leads to more reliable results than predicting at a more detailed level (e.g., discriminating between modules with 0, 1 and >1 Mandelbugs). In the best case, when using the SVM classifier along with

Table IV
CROSS-VALIDATION OF 3-CLASSES CLASSIFICATION.

| **Features** | **Classifier** | **Target class: number of Mandelbugs** | | | | | | | | | **Weighted average** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $0$ **Mandelbugs** | | | $1$ **Mandelbug** | | | $> 1$ **Mandelbugs** | | | | | |
| | | *Pr* | *Re* | *F* | *Pr* | *Re* | *F* | *Pr* | *Re* | *F* | *Pr* | *Re* | *F* |
| *All metrics* | Dec. Trees | 0.772 | 0.637 | 0.689 | 0.283 | 0.333 | 0.280 | 0.779 | 0.750 | 0.746 | 0.682 | 0.608 | **0.627** |
| | Mult. Logistic Reg. | 0.784 | 0.533 | 0.623 | 0.128 | 0.222 | 0.159 | 0.602 | 0.750 | 0.658 | 0.623 | 0.536 | 0.550 |
| | Bayes Net. | 0.702 | 0.726 | 0.707 | 0.109 | 0.156 | 0.126 | 0.950 | 0.800 | 0.858 | 0.653 | 0.638 | **0.636** |
| | SVM | 0.700 | 0.570 | 0.622 | 0.071 | 0.133 | 0.092 | 0.725 | 0.717 | 0.693 | 0.588 | 0.525 | 0.540 |
| | Naive Bayes | 0.612 | 0.904 | 0.729 | 0.000 | 0.000 | 0.000 | 0.722 | 0.467 | 0.557 | 0.525 | 0.625 | 0.549 |
| *Selected metrics* | Dec. Trees | 0.683 | 0.644 | 0.660 | 0.072 | 0.089 | 0.079 | 0.771 | 0.967 | 0.857 | 0.590 | 0.621 | 0.600 |
| | Mult. Logistic Reg. | 0.668 | 0.533 | 0.587 | 0.155 | 0.267 | 0.193 | 0.957 | 0.867 | 0.899 | 0.644 | 0.567 | 0.591 |
| | Bayes Net. | 0.711 | 0.970 | 0.820 | 0.022 | 0.022 | 0.022 | 1.000 | 0.833 | 0.876 | 0.654 | 0.758 | **0.684** |
| | SVM | 0.651 | 0.874 | 0.744 | 0.000 | 0.000 | 0.000 | 0.727 | 0.633 | 0.661 | 0.548 | 0.650 | 0.584 |
| | Naive Bayes | 0.682 | 0.793 | 0.731 | 0.000 | 0.000 | 0.000 | 0.822 | 0.650 | 0.709 | 0.589 | 0.608 | 0.588 |
| *All metrics, with novel metrics* | Dec. Trees | 0.740 | 0.615 | 0.662 | 0.231 | 0.267 | 0.222 | 0.762 | 0.717 | 0.721 | 0.650 | 0.575 | **0.595** |
| | Mult. Logistic Reg. | 0.798 | 0.533 | 0.618 | 0.132 | 0.222 | 0.162 | 0.572 | 0.700 | 0.616 | 0.617 | 0.517 | 0.532 |
| | Bayes Net. | 0.682 | 0.719 | 0.695 | 0.096 | 0.111 | 0.099 | 0.906 | 0.800 | 0.842 | 0.628 | 0.625 | **0.620** |
| | SVM | 0.812 | 0.622 | 0.688 | 0.100 | 0.156 | 0.120 | 0.645 | 0.750 | 0.681 | 0.637 | 0.567 | **0.579** |
| | Naive Bayes | 0.632 | 0.978 | 0.767 | 0.000 | 0.000 | 0.000 | 0.867 | 0.467 | 0.603 | 0.572 | 0.667 | **0.582** |
| *Selected metrics, with novel metrics* | Dec. Trees | 0.710 | 0.615 | 0.655 | 0.243 | 0.267 | 0.236 | 0.767 | 0.883 | 0.805 | 0.637 | 0.617 | 0.614 |
| | Mult. Logistic Reg. | 0.724 | 0.548 | 0.616 | 0.165 | 0.333 | 0.216 | 0.879 | 0.717 | 0.775 | 0.658 | 0.550 | 0.581 |
| | Bayes Net. | 0.665 | 0.859 | 0.747 | 0.017 | 0.022 | 0.019 | 0.973 | 0.700 | 0.806 | 0.620 | 0.663 | 0.625 |
| | SVM | 0.778 | 0.993 | 0.871 | 0.000 | 0.000 | 0.000 | 0.793 | 0.867 | 0.819 | 0.636 | 0.775 | **0.695** |
| | Naive Bayes | 0.804 | 0.807 | 0.801 | 0.267 | 0.267 | 0.261 | 0.817 | 0.800 | 0.796 | 0.707 | 0.704 | $\underline{\mathbf{0.698}}$ |

the proposed metrics and feature selection (see Table V), performance metrics are always greater than $0.9$, with an average F-measure equal to $0.941$, leading to a reliable enough classification for practical purposes. Moreover, by looking at the weighted average of F-measures (in the last column in the table), it can be seen that both feature selection and the proposed metrics are necessary to achieve the best performance: the Wilcoxon signed-rank test showed that performance metrics are higher with statistical significance. This result confirms that the proposed metrics are helpful to improve the prediction of Mandelbug-prone modules.

Table V
FEATURE SELECTION WITH RESPECT TO 2-CLASSES CLASSIFICATION.

| **Metrics** | **Selected features** | $R^2$ | **Adj-**$R^2$ |
|---|---|---|---|
| Traditional metrics | *CountInput*, *N2*, *Vol*, *AvgEssential* | 0.8536 | 0.8004 |
| Traditional and proposed metrics | *CountInput*, *Essential*, *N2*, *CountWaitNotify*, *Vol*, *CountClassBase*, *ImportIO* | 0.9843 | 0.9705 |

***Prediction by regression.*** Finally, we evaluated the effectiveness of fault prediction when using a regression model. In this scenario, the aim is to rank modules with respect to their number of Mandelbugs: this is achieved by estimating the number of Mandelbugs in a module using a regression model and by ordering the modules according to this estimate. In order to evaluate the accuracy of the ranking, we adopt cross-validation to train regression models using a subset of data and to evaluate the accuracy of models with respect to the remaining data, repeating this process several times. The ranking produced by the regression model is compared to the correct ordering of modules, by using the Pearson correlation coefficient [26]. Moreover, since the tester would focus verification on the topmost modules in the ranking provided by the regression model (according to the observation that most faults are located in few modules), we evaluated the accuracy of regression in predicting which modules have the highest number of Mandelbugs, assuming that the tester is going to focus on the top-20% modules in the ranking. We evaluated the following two measures:

- *"TOP-20%"*: The ratio between the number of Mandelbugs located in the *predicted* 20% topmost modules and the total number of Mandelbugs *in the test set*. This ratio represents the expected percentage of Mandelbugs that can be detected when focusing on the predicted topmost modules, *among the whole set of Mandelbugs in the system.*
- *"Normalized TOP-20%"*: The ratio between the number of Mandelbugs located in the *predicted* 20% topmost modules and the number of Mandelbugs in the *actual* 20% topmost modules. This ratio compares the number of Mandelbugs that can be detected when focusing on the predicted topmost modules *with the number of Mandelbugs that could be detected in the case of perfect prediction.*

Table VII shows the results of cross-validation for regression models. The best results (according to the Wilcoxon signed-rank test) are obtained using both the proposed metrics and feature selection. In particular, when the linear regression model or SVR is adopted, (i) the expected correlation coefficient of predicted ranking is higher than $0.7$, (ii) about $60\%$ of the whole set of Mandelbugs can be detected when focusing verification on the predicted $20\%$ of modules (as indicated by the "TOP 20%" measure), and (iii) the

Table VI
CROSS-VALIDATION OF 2-CLASSES CLASSIFICATION.

| | | Target class: Mandelbugs-proneness | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Features** | **Classifier** | **Mandelbug-free** | | | **Mandelbug-prone** | | | **Weighted average** | | |
| | | *Pr* | *Re* | *F* | *Pr* | *Re* | *F* | *Pr* | *Re* | *F* |
| *All metrics* | Dec. Trees | 0.698 | 0.652 | 0.665 | 0.589 | 0.629 | 0.598 | 0.651 | 0.642 | **0.636** |
| | Mult. Logistic Reg. | 0.742 | 0.607 | 0.664 | 0.592 | 0.724 | 0.648 | 0.676 | 0.658 | **0.657** |
| | Bayes Net. | 0.701 | 0.748 | 0.706 | 0.676 | 0.571 | 0.593 | 0.690 | 0.671 | **0.656** |
| | SVM | 0.724 | 0.630 | 0.670 | 0.601 | 0.695 | 0.642 | 0.670 | 0.658 | **0.658** |
| | Naive Bayes | 0.654 | 0.659 | 0.650 | 0.548 | 0.533 | 0.533 | 0.608 | 0.604 | **0.599** |
| *Selected metrics* | Dec. Trees | 0.882 | 0.770 | 0.819 | 0.754 | 0.867 | 0.804 | 0.826 | 0.813 | **0.812** |
| | Mult. Logistic Reg. | 0.864 | 0.800 | 0.828 | 0.774 | 0.838 | 0.802 | 0.825 | 0.817 | **0.817** |
| | Bayes Net. | 0.833 | 0.807 | 0.815 | 0.776 | 0.790 | 0.778 | 0.808 | 0.800 | **0.799** |
| | SVM | 0.751 | 0.867 | 0.803 | 0.798 | 0.629 | 0.697 | 0.772 | 0.763 | **0.757** |
| | Naive Bayes | 0.810 | 0.815 | 0.808 | 0.763 | 0.743 | 0.746 | 0.789 | 0.783 | **0.781** |
| *All metrics, with novel metrics* | Dec. Trees | 0.680 | 0.622 | 0.643 | 0.549 | 0.600 | 0.565 | 0.622 | 0.613 | 0.609 |
| | Mult. Logistic Reg. | 0.789 | 0.622 | 0.685 | 0.627 | 0.771 | 0.681 | 0.718 | 0.688 | 0.683 |
| | Bayes Net. | 0.633 | 0.622 | 0.622 | 0.549 | 0.543 | 0.537 | 0.596 | 0.588 | 0.585 |
| | SVM | 0.809 | 0.711 | 0.751 | 0.694 | 0.781 | 0.727 | 0.759 | 0.742 | **0.741** |
| | Naive Bayes | 0.702 | 0.770 | 0.730 | 0.672 | 0.571 | 0.609 | 0.689 | 0.683 | 0.677 |
| *Selected metrics, with novel metrics* | Dec. Trees | 0.802 | 0.711 | 0.747 | 0.672 | 0.752 | 0.701 | 0.745 | 0.729 | 0.727 |
| | Mult. Logistic Reg. | 0.956 | 0.926 | 0.938 | 0.916 | 0.943 | 0.927 | 0.938 | 0.933 | **0.933** |
| | Bayes Net. | 0.762 | 0.807 | 0.775 | 0.738 | 0.648 | 0.671 | 0.751 | 0.738 | 0.729 |
| | SVM | 0.976 | 0.919 | 0.943 | 0.917 | 0.971 | 0.940 | 0.950 | 0.942 | **0.941** |
| | Naive Bayes | 0.859 | 0.911 | 0.875 | 0.883 | 0.781 | 0.817 | 0.870 | 0.854 | 0.850 |

regression model identifies on average 83% of Mandelbugs that could be identified in the case of perfect prediction (as indicated by the "Normalized TOP 20%" measure). For these models, the average $R^2$ measure on the test sets is $R^2_{\text{linear}} = 0.482$ and $R^2_{\text{SVR}} = 0.571$, respectively; although the models do not precisely fit data, they can still be useful to point out which are the "TOP 20%" most Mandelbug-prone modules in the system. A similar result was observed in [26] where failure-prone components were ranked using regression models. We conclude that *fault prediction can be useful to identify a small set of modules in which most of Mandelbugs are located*, thus improving V&V planning.

Table VII
CROSS-VALIDATION OF REGRESSION.

| | **Regressor** | **Pearson coef.** | **TOP 20%** | **Norm. TOP 20%** |
|---|---|---|---|---|
| *All metrics* | Linear Regr. | 0.241 | **31%** | **46%** |
| | Regr. Trees | 0.115 | 12% | 19% |
| | SVR | 0.352 | **34%** | **50%** |
| *Selected metrics* | Linear Regr. | 0.516 | 38% | 51% |
| | Regr. Trees | 0.343 | 23% | 33% |
| | SVR | 0.617 | **50%** | **74%** |
| *All metrics, with novel metrics* | Linear Regr. | 0.227 | **30%** | **45%** |
| | Regr. Trees | 0.115 | 12% | 19% |
| | SVR | 0.326 | **32%** | **50%** |
| *Selected metrics, with novel metrics* | Linear Regr. | 0.770 | **59%** | **83%** |
| | Regr. Trees | 0.230 | 14% | 22% |
| | SVR | 0.707 | **61%** | **83%** |

## VI. THREATS TO VALIDITY

Observed results are limited to the considered software system, although the analysis is comforted by the extensiveness of the study accounting for more than 400 problem reports. We observed that the percentage of Bohr/Mandelbugs is comparable to past studies, while the pinpointed relationships with software metrics is new, and needs to be confirmed on different systems in future work.

To interpret the results, it should be considered that the manual classification, even if driven by the defined procedure, may introduce a bias, e.g., due to unclear reports, or to misinterpretations. This can affect the measurement of the Bohr/Mandelbugs proportion, and thus the identification of Mandelbug-prone modules. To limit this bias, we exclude those reports not fully clear and/or with not sufficient information, and then, on the resulting set, a cross-validation step is carried out: results were validated by three of the authors, discussing reports with the V&V team where possible, and excluding a small part of reports (6.51%) that were not entirely clear with respect to the adopted criteria.

## VII. CONCLUSION

In this paper, we investigated how to predict the location of Mandelbugs in complex software systems. We found that Mandelbugs account for a noticeable share of bugs, and that there are components more prone to Mandelbugs than others. This fact motivates the use of fault prediction to focus V&V activities and fault-tolerance mechanisms. We also found that using both traditional software metrics and a novel set of metrics (based on concurrency, I/O and exception handling constructs), fault prediction can achieve high accuracy. Future work will be aimed to combine prediction with V&V and fault-tolerance strategies.

## ACKNOWLEDGMENT

This work was supported by the projects "Iniziativa Software CINI-Finmeccanica", "Embedded Systems in Critical Domains" (CUP B25B09000100007), and "SVEVIA:

Innovative methods and techniques for Software VErification and ValIdAtion of near-realtime complex systems" (PON02_00485_3487758). The authors are grateful to Rosa Spina, Mauro Russo and Cristofaro Gozzolino for their help during the preparation of this work.

## References

[1] M. Grottke and K. S. Trivedi, "Fighting bugs: Remove, retry, replicate, and rejuvenate," *IEEE Comp.*, vol. 40, no. 2, pp. 107–109, 2007.

[2] E. Adams, "Optimizing preventive service of software products," *IBM Journal of Research and Development*, vol. 28, no. 1, pp. 2–14, 1984.

[3] J. Gray, "Why do computers stop and what can be done about it?" Tandem Computers, Tech. Rep. 85.7, 1985.

[4] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. Intl. Symp. Fault-Tolerant Computing*, 1995, pp. 381–390.

[5] M. Grottke, A. Nikora, and K. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. Intl. Conf. Dep. Sys. and Netwks.*, 2010, pp. 447–456.

[6] Y. Wang, Y. Huang, and C. Kintala, "Progressive retry for software failure recovery in message-passing applications," *IEEE Trans. Comp.*, vol. 46, no. 10, pp. 1137–1141, 1997.

[7] S. Maffeis and D. Schmidt, "Constructing reliable distributed communication systems with CORBA," *Comm. Mag.*, vol. 35, no. 2, pp. 56–60, 1997.

[8] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *Op. Systems Review*, vol. 39, no. 5, 2005, pp. 235–248.

[9] K. S. Trivedi, R. Mansharamani, D. S. Kim, M. Grottke, and M. Nambiar, "Recovery from failures due to Mandelbugs in IT systems," in *Proc. Pacific Rim Intl. Symp. Depend. Comp.*, 2011, pp. 224–233.

[10] V. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M. Zelkowitz, "The empirical investigation of perspective-based reading," *Emp. Softw. Eng.*, vol. 1, no. 2, pp. 133–164, 1996.

[11] G. Holzmann, "The model checker SPIN," *IEEE Trans. on Soft. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, 2008, pp. 267–280.

[13] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. on Soft. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[14] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. on Soft. Eng.*, vol. 33, no. 1, pp. 2–13, 2007.

[15] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic review of fault prediction performance in software engineering," *IEEE Trans. on Soft. Eng.*, 2011, on press.

[16] S. Chandra and P. M. Chen, "Whither generic recovery from application faults? A fault study using open-source software," in *Proc. Intl. Conf. Depend. Sys. Netwks.*, 2000, pp. 97–106.

[17] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *Proc. Intl. Conf. Softw. Eng.*, 2010, pp. 485–494.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics." in *Proc. Intl. Conf. Arch. Supp. Prog. Lang. Op. Sys.*, 2008, pp. 329–339.

[19] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues, "A study of the internal and external effects of concurrency bugs," in *Proc. Intl. Conf. Depend. Sys. Netwks.*, 2010, pp. 221–230.

[20] I. Lee and R. Iyer, "Software dependability in the Tandem GUARDIAN system," *IEEE Trans. Softw. Eng.*, vol. 21, no. 5, pp. 455–467, 1995.

[21] R. Chillarege, "Understanding Bohr-Mandel bugs through ODC triggers and a case study with empirical estimations of their field proportion," in *Wksp. Softw. Aging Rejuv.*, 2011.

[22] M. Grottke and B. Schleich, "How does testing affect the availability of aging software systems?" *Perf. Eval.*, in press.

[23] V. Basili and B. Perricone, "Software errors and complexity: an empirical investigation," *Comm. ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[24] N. Fenton and S. Pfleeger, *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[25] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. on Soft. Eng.*, vol. 22, no. 12, pp. 886–894, 1996.

[26] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. Intl. Conf. Soft. Eng.*, 2006, pp. 452–461.

[27] C. Catal and B. Diri, "A systematic review of software fault prediction studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346–7354, 2009.

[28] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. on Soft. Eng.*, vol. 34, no. 4, pp. 485–496, 2008.

[29] Y. Jiang, B. Cukic, and T. Menzies, "Fault prediction using early lifecycle data," in *Proc. Intl. Symp. on Software Reliability Eng.*, 2007, pp. 237–246.

[30] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project Defect Prediction—A Large Scale Experiment on Data vs. Domain vs. Process," in *Proc. Europ. Soft. Eng. Conf.*, 2009, pp. 91–100.

[31] B. Turhan, T. Menzies, A. B. Bener, and J. D. Stefano, "On the Relative Value of Cross-company and Within-company Data for Defect Prediction," *Empirical Soft. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.

[32] B. Caglayan, A. Tosun, A. Miranskyy, A. Bener, and N. Ruffolo, "Usage of multiple prediction models based on defect categories," in *Proc. Intl. Conf. on Predictive Models in Soft. Eng.*, 2010.

[33] A. Mısırlı, B. Çağlayan, A. Miranskyy, A. Bener, and N. Ruffolo, "Different strokes for different folks: a case study on software metrics for different defect categories," in *Wksp. on Emerg. Trends in Soft. Metrics*, 2011, pp. 45–51.

[34] Scientific Toolworks Inc., *What metrics does Understand have?*, http://www.scitools.com/documents/metrics.php.

[35] I. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, 2011.

[36] N. Draper and H. Smith, *Applied regression analysis*. Wiley-Interscience, 1998.

[37] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Machine Learning Res.*, vol. 7, pp. 1–30, 2006.

[38] K. Killourhy and R. Maxion, "Comparing anomaly-detection algorithms for keystroke dynamics," in *Proc. Intl. Conf. Dep. Sys. and Netwks.*, 2009, pp. 125–134.

[39] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *J. Roy. Stat. Soc., Ser. B*, pp. 289–300, 1995.