UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CRISTIANO WERNER ARAUJO

Bug Prediction in Procedural Software Systems

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Ingrid Nunes

Porto Alegre
August 2017

CIP — CATALOGING-IN-PUBLICATION

Araujo, Cristiano Werner

Bug Prediction in
Procedural Software Systems / Cristiano Werner Araujo. – Porto Alegre: PPGC da UFRGS, 2017.

90 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Ingrid Nunes.

1. Bug prediction. 2. Procedural programming. 3. Static code metrics. I. Nunes, Ingrid. II. Título.

“ For of him, and through him, and to him, are all things: to whom be glory for ever”

— ROMANS 11:36

THANKS

I would like to thank first my wife, for her support and patience in all my nights working and do not giving her the attention she deserves. Also, to all my family, especially to my mother, whom listened to my complaints and discussed with me about machine learning and statistics; and for my sister-in-law for her revision of my Portuguese.

Thanks for the folks back in Parks, thanks everyone for their patience on my everlasting subject: masters; and namely thanks to Alex, Leandro Lucas and Pedro for the inspirations, ideas and discussions.

I also want to thank my Prosoft group colleges, for the help and tips, especially for Jhonny and Vanius. Also for my advisor which corrected me with patience and taught me the way into the academic research.

Thanks also for my friends from church, namely Ricardo, Matheus, Cristiano, Victoria, Giovana for their conversations about software and academia. And last but not least, I want God for his sovereign path and the lessons he taught me through this dissertation.

ABSTRACT

Information regarding bug fixes has been explored to build bug predictors, which provide support for the verification of software systems, by identifying fault-prone elements, such as files. A wide range of static and change metrics have been used as features to build such predictors. Many bug predictors have been proposed, and their main target is object-oriented systems. Although object-orientation is currently the choice for most of the software applications, the procedural paradigm is still being used in many—sometimes crucial—applications, such as operating systems and embedded systems. Consequently, they also deserve attention. This dissertation extends work on bug prediction by evaluating and tailoring bug predictors to procedural software systems. We provide three key contributions: (i) comparison of bug prediction approaches in context of procedural software systems, (ii) proposal of the use of software quality features as prediction features in the studied context, and (iii) evaluation of the proposed features in association with the best approach found in (i). Our work thus provides foundations for improving the bug prediction performance in the context of procedural software systems.

Keywords: Bug prediction. procedural programming. static code metrics.

Predição de *Bugs* para Sistemas Procedurais

RESUMO

Informação relacionada a concertos de *bugs* tem sido explorada na construção de preditores de *bugs* cuja função é o suporte para a verificação de sistemas de software identificando quais elementos, como arquivos, são mais propensos a *bugs*. Uma grande variedade de métricas estáticas de código e métricas de mudança já foi utilizada para construir tais preditores. Dos muitos preditores de *bugs* propostos, a grande maioria foca em sistemas orientados à objeto. Apesar de orientação a objetos ser o paradigma de escolha para a maioria das aplicações, o paradigma procedural ainda é usado em várias — muitas vezes cruciais — aplicações, como sistemas operacionais e sistemas embarcados. Portanto, eles também merecem atenção. Essa dissertação extende o trabalho na área de predição de *bugs* ao avaliar e aprimorar preditores de *bugs* para sistemas procedurais de software. Nós proporcionamos três principais contribuições: (i) comparação das abordagens existentes de predição de *bugs* no contexto de sistemas procedurais, (ii) proposta de uso dos atributos de qualidade de software como atributos de predição no contexto estudado e (iii) avaliação dos atributos propostos em conjunto com a melhor abordagem encontrada em (i). Nosso trabalho provê, portanto, fundamentos para melhorar a performance de preditores de *bugs* no contexto de sistemas procedurais.

Palavras-chave: predição de defeitos, programação procedural, métricas estáticas de código.

LIST OF ABBREVIATIONS AND ACRONYMS

AMC	Average Method Complexity
AST	Abstract Syntax Tree
BTS	Bug Tracking System
Ca	Afferent Couplings
CAM	Cohesion among Methods of Class
CBM	Coupling between Methods
CBO	Coupling between Object Classes
Ce	Efferent Couplings
CK	Chidamber & Kemerer
CPL	Compiler Warnings
CPS	Standard Code Warnings
CPW	Wall Code Warnings
CPX	Wextra Code Warnings
DAM	Data Access Metrics
DIT	Depth of Inheritance Tree
DSL	Domain Specific Languages
DT	Decision Tree
DUP	Duplicated Code
EDC	External Duplicated Code
FN	False Negative
FP	False Positive
GCC	GNU Compiler Collection
GQM	Goal Question Metric
GY	Gyimothy Approach

IC	Inheritance Coupling
IDC	Internal Duplicated Code
IDE	Integrated Development Environment
JU	Jureczko Approach
KI	Kim Approach
KO	Koru Approach
LCOM	Lack of Cohesion in Methods
LCOMN	allowing Negative value
LOC	Lines of Code
LOCM3	Lack of Cohesion in Methods
LR	Logistic Regression
LWIP	Light Weight IP
MFA	Measure of Functional Abstraction
MOA	Measure of Aggregation
MO	Moser Approach
NB	Naive Bayes
NN	Neural Network
NOC	Number of Children
NPM	Number of Public Methods
OO	Object-Oriented
PCA	Principal Component Analysis
PRC	Constants Preprocessor Count
PRE	Preprocessor Count
PRI	Includes Preprocessor Count
PRK	Keys Preprocessor Count
PRM	Macros Preprocessor Count

RFC	Response for a Class
RF	Random Forest
RQ	Research Question
SAC	CppCheck Static Analyser
SAL	Static Analysers
SAU	Uno Static Analyser
SCM	Software Configuration Management
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
UML	Unified Modeling Language
VCS	Version Control System
WMC	Weighted Methods per Class
XML	Extensible Markup Language

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In software development, a significant amount of time is dedicated to bug corrections. The bugs—or defects—have a variety of root causes. They are caused by many factors, such as wrong API (application user interface) usage, wrong or incomplete specification and other factors (KO; MYERS, 2005). To identify bugs development processes include steps for testing each piece of software delivered. In each test step of the process, previous parts are combined, and more sophisticated tests are applied.

The further in the integration process the bug is found, the higher the cost (BLACK, 2003). Consequently, the earlier a bug is found, the cheaper the correction is. This fact brings motivation for better software testing. An ideal test must always lead to a specific piece of code that contains defects.

In practice, finding and locating a bug is a hard task. The relationship among multiple modules, different interfaces, and many abstractions layers may complicate it. Ensuring that when a test fails the outcome is a defect detection with its context is not an easy task. This requires a precise specification, which is not always available. Nevertheless, some code patterns are known (SOMMERVILLE, 2010) and when applied, can be used to reduce bug incidence even before running tests, hence reducing testing costs even further. This results in semi-automated methods based on static and run-time analysis for quality improvement. One example of such methods is code lints.

More complex approaches use code change and code metrics for predicting the location of bugs (HASSAN; HOLT, 2005) (ZIMMERMANN; NAGAPPAN; ZELLER, 2008) (KIM et al., 2007). These approaches lead to the construction of *bug predictors*, which take as input information of a given software system, typically collected from issue trackers and software repositories, and build a prediction model that is used to indicate fault-prone elements of the system, such as classes, files or modules. Often, results can also include the fault probability. The output of bug predictors can be used to drive decisions regarding the effort demanded to test, review and verify software systems, and the prioritisation of which entities should be tested.

Previous work focused mostly, if not only, on object-oriented (OO) systems. Object orientation is the choice for many software systems, such as web and mobile applications. However, the procedural paradigm is still being used in many—sometimes crucial—software systems, such as operating systems, embedded systems, and scientific computing applications. These applications deserve attention not only because they must

be maintained, but also because they are often long-lived systems and procedural languages lack some mechanisms (e.g. inheritance and polymorphism) that improve code quality. These factors may cause the maintenance and evolution of those systems to be even harder. In this work, we address the gap between bug prediction and procedural software systems. In next section, we describe in more detail the problem and limitations of existing bug prediction approaches, and in Section 1.2 we present the work we performed to fill this gap.

1.1 Problem Statement and Limitations of Related Work

As introduced, most of the existing bug predictors focus on OO systems and leverage OO code-extracted information such as OO metrics or were investigated and validated in the context OO systems. Explored static code metrics are often limited to those part of traditional metric suites, e.g. that proposed by Chidamber and Kemerer (CK) (CHIDAMBER; KEMERER, 1994) and Halstead (HALSTEAD, 1977), which capture structural code properties. Other studies are based on text change, which uses version control systems (VCSs) information, hence are independent of the programming language used (MOSER; PEDRYCZ; SUCCI, 2008). Some approaches (D'AMBROS; LANZA; ROBBES, 2010) also use previous process information, e.g. a number of post-release bugs and number of commits to determine current release stability.

Approaches that use metrics as a predictor (BASILI; BRIAND; MELO, 1996) (ZIMMERMANN; PREMRAJ; ZELLER, 2007) typically rely on CK metrics suite (CHIDAMBER; KEMERER, 1994). Because CK metrics suite is focused on OO systems, procedural systems cannot be well measured with them. There is a simpler and widely known set of metrics described by Sommerville (SOMMERVILLE, 2010). Most known examples are Lines of Code, Cyclomatic complexity, fan-in and fan-out.

Because procedural systems have different characteristics and are usually adopted to implement systems of a particular nature—e.g. embedded systems, operating systems, and scientific computing—the code structure may differ from OO software, causing procedural software to be potentially different. Consequently, bug predictors can be tailored to procedural software systems. This can be achieved by exploring particularities of procedural languages, extracting information, such as metrics, that captures particular structures present in systems implemented in procedural languages, like C. Our goal in this work is thus to improve bug prediction in the context of procedural software systems.

More specifically, given that procedural programming languages are mostly used to develop low-level applications, such as drivers and scientific applications, we focus on bug prediction in these application domains.

1.2 Proposed Solution and Overview of Contributions

In this work, we address the aforementioned problem by first presenting a study conducted to evaluate approaches that rely on static code metrics (RADJENOVIć et al., 2013) in the context of procedural software systems. We evaluate only static source code metrics despite the existence of evidence that source code change metrics are a valuable source of information for bug prediction (MOSER; PEDRYCZ; SUCCI, 2008). The reasons for performing this analysis are: (i) static code metrics, which are useful to be incorporated to bug predictors, can be combined with change metrics; and (ii) there are cases in which a version control system is not available to extract change metrics, due to for example third-party development. Approaches were evaluated from two perspectives: (i) *degree of applicability*, by measuring the amount of OO-specific information they use; and (ii) *effectiveness*, by measuring their precision, recall and F-measure with a set of procedural software systems. Effectiveness was evaluated with the subset of metrics applicable to procedural systems, and with this subset together with metrics adapted to the procedural paradigm. For building our dataset, we selected a range of procedural software systems from many application domains, both open-source and proprietary, including operating systems and tools, bare-metal environments (software that does not require the support of a host operating system), and embedded commercial applications. Static code metrics and defects were extracted from each target system. Prediction was performed using learning techniques applied by the evaluated approaches.

We then investigate the use of code quality tools and analysis of bad programming practices to obtain information regarding source code problems and use it to build bug predictors for procedural software systems. We specify a set of four features associated with code issues, namely presence of duplicated code, issues of static analysers, compiler warnings, and use of preprocessors, and evaluate how effective such features are for bug predictions. We used as baseline an existing set of metrics used for bug prediction (KORU; LIU, 2005), which performed best in our investigated context. In order to evaluate the effectiveness of the features, our evaluation consists of an empirical study that assessed accuracy metrics obtained with different subsets of features. This evaluation

allowed us not only to evaluate the effectiveness of our proposed features, but also identify the sets of features with highest results, and thus are the best candidates to be used to build bug predictors in our target context.

Therefore, the main contributions of this work are the following: (i) comparison of existing bug prediction approaches in our investigated context; (ii) extension of features used in bug prediction to the context of procedural software systems and; (iii) evaluation of the improvement provide by the proposed features. Moreover, we also provide a rank of individual attributes based on the machine learning scores used.

1.3 Outline

The remainder of this dissertation is arranged as follows. Existing bug prediction approaches are described in Chapter 2, presenting the related work using source code changes, static source code metrics, combination and comparison of both, and studies regarding using bug prediction models across multiple systems. In Chapter 3 we present the study performed to find which approach is more suitable for procedural software systems. Proposed features to enhance bug prediction in the context of procedural software systems are presented in Chapter 4 and an empirical study using these proposed features is described in Chapter 5. Finally, we summarise our contributions and future work in Chapter 6.

2 DEFECT PREDICTION APPROACHES

Defect prediction, or bug prediction, consists of predicting whether a given entity has a probability of having bugs based on extracted information about system. This can be done using a snapshot of the system or analysing the way it changes over time. In addition, the outcome can be a binary decision (*with or without bug*) or a score, pointing out the probability that a given entity has of having defects. Entities can have different granularity levels, which can be functions, classes, packages, modules and entire programs. The list of bug prediction approaches presented here are not exhaustive, a more complete survey is presented by Jaechang (JAECHANG, 2017).

In this chapter existing bug prediction approaches are presented. First, we introduce a machine learning background in Section 2.1. In Section 2.2, the use of static source code metrics for defect prediction is presented. Next, the use of process and change metrics are presented in Section 2.3. Combinations of both methods are presented in Section 2.4 and comparisons of change and static source code based predictors are presented in Section 2.5. Cross-project defect prediction studies are presented in Section 2.6. Finally, we present the approaches selected for evaluation on this work in Section 2.7.

2.1 Machine Learning Background

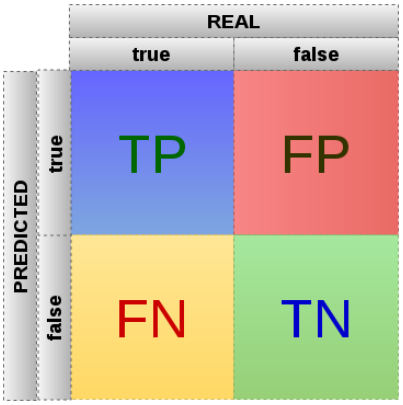
Because most of bug prediction approaches use machine learning classification algorithms, we next describe the metrics used to evaluate their performance. The concepts of TP (true positive), FP (false positive), FN (false negative) and TN (true negative), used to calculate the machine learning scores, are presented in Figure 2.1.

Precision is the fraction of all classified files that are classified as defective. It is calculated as follows: Precision $= TP/(TP + FP)$, where TP is the correctly classified defective files and FP is non-defective files classified as defective.

Recall is the fraction of all files that should be classified as defective that are classified as defective. It is calculated as follows: Recall $= TP/(TP + FN)$, where TP is the correctly classified defective files and FN is the defective files classified as non-defective.

F-measure is a score that combines recall and precision. It is the harmonic mean between

Figure 2.1: Relations of predicted and real classes for machine learning scores calculation.



them, calculated as follows: F-measure $= (2 \cdot Precision \cdot Recall)/(Precision + Recall)$.

All presented scores are in $[0, 1]$, were the closer to one, the better the classification. Other scores (e.g. accuracy) are presented by Alpaydin (ALPAYDIN, 2010) and are not within the scope of this work. Next, we present the bug prediction approaches.

2.2 Defect Predictors based on Static Code Metrics

With the rise of static source code metrics as a measurement for software systems complexity, one of their first applications were to use them to predict defects. In this section we discuss bug predictors based on static source code metrics. First we present the approaches that are the inception for bug prediction, providing evidence of the correlation between bugs and complexity. Later, we present an approach that was applied to a large open source project also has this characteristic and last we discuss an approach which uses metrics for bug prediction in five different systems.

Based on the CK metric, Basili et al. (BASILI; BRIAND; MELO, 1996) performed an experiment to evaluate if such metrics could be used as an indicator for bug presence. In the first step of the experiment, the participants—undergraduate students—wrote a medium size software in the C++ language. System metrics were then calculated and stored. In the second step, the software was evaluated, and the errors found were reported back to the students. After fixing the errors, the metrics were calculated again. The difference of metrics at each step were used to evaluate if the set of metrics could be correlated with buggy code. The conclusion was that this correlation is true.

In parallel, Binkley and Schach (BINKLEY; SCHACH, 1998) evaluated a similar

set of metrics for three systems. The systems used were implemented in different languages: C, Cobol and C++. Some metrics used were different for each case. In the C and Cobol systems, run-time behaviours—accessing external data elements for example—were used as metrics. With the C++ system the metrics used are similar to the CK metrics. When available, they used CDM (Coupling dependency metric), fan-in, fan-out and the number of clients (run-time information). The coupling dependency metric was a combination of three other metrics.

- Referential dependency: measures the dependency on declarations.
- Structural dependency: measures the dependency on internal organisation.
- Data integrity dependency: measures the vulnerability of the data in a module from changes by other modules.

The Spearman coefficient was used to correlate each system metrics with the defects detected at run-time. The metrics with greater correlation with defects were the ones that indicate the coupling degree (CDM, fan-in, fan-out and RFC). Therefore, the study concluded that coupling metrics are a better predictor of bugs.

Instead of comparing three different systems with different metrics, Olague et al. (OLAGUE et al., 2007) compare the same system—Rhino, a JavaScript interpreter written in Java—using three sets of OO metrics: CK, MOOD and QMOOD. The metrics were computed for each release of Rhino and paired with that release reported defects. Afterwards, a univariate binary logistic regression was done to determine which metrics were the most significant. The study concluded that the coupling dependency metric was the best estimator for bug presence. No limitations for the study were presented, but because only one system was evaluated, there is no evidence for generalisation.

Koru and Liu (KORU; LIU, 2005) used five systems from NASA which only the metrics are available, it is found on in the PROMISE data repository (SHIRABAD; MENZIES, 2005). They built J48 regression trees using the WEKA (HALL et al., 2009) software suite for machine learning. One finding was that the module size chosen is related to the accuracy and recall of the prediction model. Modules too small do not have enough variance. Hence, the differences between buggy and not buggy entities are unclear. A large module, in turn, contains a significant portion of code to be evaluated. Based on this, they advise against fine granularity bug prediction, using classes instead of methods, for example. A suggestion is made on aggregating (through sum) the method metrics in a class metrics. The F-measure, the harmonic measure between precision and recall, was

Table 2.1: Metrics and hypotheses used by Gyimothy et al.

Name	Description	Hypothesis
WMC (weighed methods per class)	Number of methods in classes weighed by complexity	The larger the function is the higher the bug density
RFC (response for a class)	Methods called from method	A large response increase bug probability
DIT (depth of inheritance tree)	Information from other classes	The deeper the inheritance the higher the bug chance
NOC (number of children)	Number of class descendants	High NOC reduces bug probability
CBO (coupling between object classes)	Number of other classes that are used by a class	High coupling correlates with defect density
LCOM (Lack of Cohesion on Methods)	Measures the methods similarity (variables sharing)	High coupled classes are more defect prone
LOC (Lines of code)	Number of lines not empty or commented	The class size is proportion to bug rate

used for model evaluation. They found F-measures from 0.16 to 0.56 at the class level, meaning that in the best case, the accuracy is slightly better than randomness, which has an average f-measure of 0.5. In one of the cases, KC1, they had the method information and error count, they used the information and obtained an improvement in the predictor performance. The F-measure found was 0.65 with binary decision and 0.56 with classification. The classes used for classification were buggy and not buggy.

Gyimothy et al. (GYIMOTHY; FERENC; SIKET, 2005), instead of using only the source code as input, retrieved information from Bugzilla, an issue tracking system. The source code metrics were extracted from a targeted project, the Mozilla project[1]—a Browser and E-mail suite, developed in C++. Metrics extraction was performed using the Colombus tool, developed and presented by Ference et al. (FERENC et al., 2002). The prediction techniques used went further than previous methods, using, aside from linear regression, neural networks and decision trees machine learning techniques as prediction engines. These techniques are compiled and presented by Alpaydin et al. (ALPAYDIN, 2004). The claimed difference from previous work was to use of a system that has millions of lines of code, but still a single system was evaluated. Table 2.1 shows the metrics used and the related hypotheses associated with the use of each for measuring defects.

A model for each metric and a model using all metrics was then generated. Three metrics were used for evaluating the models: accuracy, completeness and correctness. Correctness measures how much of the predicted classes are buggy. Completeness measures how many faults in total are predicted. Accuracy is the percentage of total correct

[1]<http://www-archive.mozilla.org>

Table 2.2: Metrics used by Zimmermann et al.

Metric Name
Fan-Out
Methods LOC
Total LOC
Nesting Depth
Number of Parameters
Number of Classes
Number of Static Methods
Number of anonymous type declarations
Number of interfaces

predictions of the total buggy classes. As result, they concluded that CBO and all metrics were the best models, staying roughly 5% better than other metrics. The best completeness was achieved by all metrics, LOC and CBO. Logistic regression had better accuracy, but decision trees better completeness. The neural network produced was the model with the lowest performance.

Similarly to Binkley and Schach, Zimmermann et al. (ZIMMERMANN; PREMRAJ; ZELLER, 2007) used the Spearman coefficient for evaluating the correlation between metrics collected from pre-release and post-release defects. The Eclipse defects and its source code metrics were used. Used metrics are enumerated in Table 2.2.

These metrics were computed for each file and module. Next, bug occurrence in each module and file were then predicted using a linear regression model. Finally, prediction was correlated using the Spearman coefficient. It was found that there is correlation between complexity metrics and post-release defects.

Another static source code metric used is clone metric, Kamei et al. (KAMEI et al., 2011) analysed three versions of the Eclipse system using clone metrics. They found that "clone metrics did not improve the performance of a fault prediction model", nevertheless they found that the performance of clone metrics is better in the component level. They extended the work of Monden et al. (MONDEN et al., 2002), which found that modules with some small clones are more reliable.

These approaches showed that there is correlation between source code metrics and bug occurrence. Using the correlation information, some bad smells—code patterns that can lead to errors—were found. This can improve performance enforcing programmers to avoid those when writing code. Furthermore, existing code can be analysed and potential sources of defects can be localised and refactored. None of these methods, thought, analysed the change of the source code over time. Also, the methods used

simple linear regression, which has limitations approximating non-linear functions. Nevertheless, the area of software and maintenance engineering increased from simple lints to advanced bug predictors based on these works.

2.3 Defect Predictors based on Change Code Metrics

As pointed out in the previous section, the first defect predictors were based only on static code metrics. An alternative to analysing the code statically is to analyse the way it changes. This is done by seeking correlation between defects and changes over-time, and specifically, the moment where the defect was introduced. Given this scenario, the next set of approaches we present are based on *change metrics*. These metrics typically include the number of lines added, removed and changed. Other information used is, for example, meta-data extracted from Version Control Systems (VCS) and Bug Tracking Systems (BTS), such as Git [2] and Bugzilla [3].

One of the first approaches on code change was proposed by Munson and Elbaum (MUNSON; ELBAUM, 1998). A large embedded system, QTB, was used as case study. This system has 300K LOC and 3700 functions and uses C as the programming language. System complexity metrics based on module control flow graph and number of operands were computed into a single metric. This particular metric is denominated *code delta*. The complete set of metrics is shown in Table 2.3. Metric compositions were then used for comparing different builds. One limitation presented by the authors is that changed modules with approximately the same size in the metrics composition would pass unnoticed. For this, the number of *code churn* must be evaluated. They define code churn as the sum of deleted and inserted lines of code. Comparing code churn, code delta, change requests and people involved in each version, the greatest correlation with software faults found was code churn. Thus, the study concluded that code churn is a better predictor for defects, among the investigated projects.

Instead of defining code churn as the sum of deleted and inserted lines of code, Nagappan and Ball (NAGAPPAN; BALL, 2005) split code churn in two categories: lines churned and lines removed. They used the Windows Server 2003 (SP1) as case study, using code churn to analyse the most defect-prone binaries. The post-release defects were collected for each release. The metrics presented in Table 2.4 were computed for

[2] <https://git-scm.com/>
[3] <https://www.bugzilla.org/>

Table 2.3: Metrics used by Elbaum and Munson.

Name	**Description**
Comm	Total comment count
ExStmt	Executable statements
NonEx	Non executable statements
N1	Total number of operands
$\eta 1$	Unique operands
N2	Total number of operators
$\eta 2$	Unique operators
$\eta 3$	Unique operators with overloading
Nodes	Number of nodes in the module control flow graph
Edges	Number of edges in the module control flow graph
Paths	Number of distinct paths in the module control flow graph
MaxPath	Maximum path length in the module control flow graph
Path	Average path length in the module control flow graph
Cycles	Total cycle count in the module control flow graph

the time between releases. They were associated with each other and validated using a cross check correlation with a 0.01 significance. A predictive model was created using statistical regression and discriminant analysis was used to classify the binaries. It was found that code churn is a good predictor for code defects. They used R^2, a measure of model fitness, and Root MSE, a measure for model deviations (DRAPER; SMITH, 2014). Values found were 0.811 and 1.301, respectively. These values indicate that the model found fits to the data. Validity threats pointed out are the untraceable changes in the machine and the analysis of a single software system.

In contrast to Nagappan and Ball's work, Hassan and Holt (HASSAN; HOLT, 2005) performed a case study in six Open Source projects (NetBSD, OpenBSD, FreeBSD, KDE, KOffice and Postgres) instead of using a single closed source application. The goal was to give managers a top ten list of the most error-prone modules, therefore pointing out the modules for improved test efforts. The output is a dynamic list containing the top ten currently most error-prone modules. A requirement is also the need for keeping the developers routine unchanged. The objective is to present a short response time, instead of long term planning. Four policies for cache replacement were evaluated: most frequently modified, most recently modified, most frequently fixed and most recently fixed. As performance metric, besides hit rate, an adjusted hit rate was used, where the module must be in the top ten list at least 24 hours before a bug happens on it. Furthermore, the average prediction age was used to determine how early each replacement heuristic identified a defective module. The commits analysed were also classified into three classes: fault repairing modifications, general maintenance modifications and feature introduction

Table 2.4: Metrics used by Nagappan and Ball.

Absolute Metrics
Total LOC
Churned LOC
Deleted LOC
File count
Weeks of churn
Churn count
Files churned
Relative Metrics
Churned LOC / Total LOC
Deleted LOC / Total LOC
Files Churned / File Count
Churn Count / Files Churned
Weeks of churn / File Count
(Churned LOC + Deleted LOC) / Weeks of Churn
Churned LOC / Deleted LOC
(Churned LOC + Deleted LOC) / Churn Count

modifications. The heuristics based on modification frequency obtained the best performance. Moreover, using modification commits was a better predictor than fix commits.

Similarly to Hassan and Holt (HASSAN; HOLT, 2005), Kim et al. (KIM et al., 2007) proposed a cache architecture to bug detection. Using the locality concept—common in cache systems—the following concepts were used.

- Changed entity locality: changed entities may present defects soon.
- New entity locality: new entities may present defects soon.
- Temporal locality: if an entity introduced a defect recently, it might present defects soon.
- Spatial locality: related entities will present defects in the same time span.

The spatial locality is measured by entity co-change. If two entities are often changed together, they are related and changes to one may cause or trigger defects in the other. Furthermore, the concepts of BugCache and FixCache are introduced. FixCache is a cache that is changed when a fix is committed, and the temporal and spatial entities related to the bug commit that introduced this bug are added to the cache. The hypothetical BugCache adds the commit and surroundings when a bug is committed. This approach

Table 2.5: Systems evaluated by Sunghun et al.

System	Implementation Language	Number of Files	Number of Entities
Apache HTTP 1.3	C	154	2.113
Subversion	C	255	3.963
PostgreeSQL	C	598	8.569
Mozilla	C/C++	396	8.203
JEdit	Java	420	5.429
Columba	Java	1428	8.428
Eclipse	Java	3330	33.214

is unfeasible, otherwise the bug would not be committed, so FixCache is the only viable approach for bug detection in this context. The FixCache must have a heuristic for detecting which commit introduced the bug. Based on the aforementioned localities, a FixCache containing the most defect-prone entities is built. Granularity used in the model was function and files. A 73–95% accuracy was found in the file level using 10% of files in the cache. At function/method level, the accuracy was 46%-72% with 10% of the functions/methods level in the cache. The best predictor found was recently used entities, in contrast to Hassan and Holt's work (HASSAN; HOLT, 2005), which found that frequency is a better defect indicator.

2.4 Defect Predictors using Heterogeneous Code Metrics

The approaches discussed in Section 2.3 showed that code change is associated with defects. In Section 2.2, the presented approaches also demonstrated that static code metrics are also related to bugs. Because both sets of approaches were useful for bug prediction, combining both change metrics and static code metrics in the same model can potentially increase accuracy. In this section we discuss work that follows this approach, integrating both change metrics and static code metrics.

Sunghun et al. (KIM; WHITEHEAD JR.; ZHANG, 2008) used both approaches to bug forecasting. The proposed process has the following stages: (i) extract file level changes; (ii) identify commits that are bug fixes; (iii) identify bug introducing changes; (iv) extract features from all changes; and (v) train classifier using all changes.

The features used are VCS meta-data, code metrics and text (containing the text of all possible artefacts in the process). Commit meta-data used were the author, hour and size of changes. Text (commit messages, source code and file name) is also mined, using

all words and operators, including comments and numbers. The Understand software suite[4] is used for extracting source metrics, which are supplied as features. After feature extraction for each change, the features and corresponding status (clean or buggy) are feed to a machine learning algorithm. The machine learning algorithm used in this case is Support Vector Machine (SVM) (CORTES; VAPNIK, 1995). An average accuracy of 73% was found in the projects evaluated. Given that all previous changes can be computed and cached, a new commit has a low computational cost for prediction. A limitation is that if no difference is found between fix and new feature, the prediction will be of a fix or a new feature, therefore not helping bug forecasting. Some next steps towards a better classifier were proposed by the authors. Among them are using online machine learning techniques, using more features, using semantic analysis and tweaking machine learning methods for this problem.

Instead of using machine learning algorithms, Ambros et al. (D'AMBROS; LANZA; ROBBES, 2010) used statistic methods like PCA (Principal Component Analysis) (ABDI; WILLIAMS, 2010), regression models (DRAPER; SMITH, 2014), R^2 (DRAPER; SMITH, 2014) and Spearman coefficient (YULE, 1968) for building and validating models. Five systems were used as case study; four are part of the Eclipse ecosystem (JDT Core, PDE UI, Equinox and Mylyn) and the other is the Apache Lucene. All of them are written in Java. The previously mentioned systems were used to create a corpus that can be used as benchmark for others defect predictors. The corpus was created using a meta-model to describe the source code (in the FAMIX language). Moreover, the corpus contained a model of the change history, saving all the commits meta-data together with the change-set. Commit log messages were used to link classes to commits. The metrics used were then combined to form new metrics, among them are the introduced OO metrics and change metrics. Entropy measures were also used, they were based on the change metrics. Given that a lot of data were examined, many conclusions were found, detailed as follows.

1. Metrics based on code churn are more efficient than static metrics.

2. Bug-fixes and code metrics are more data-wise (less cost) than code churn.

3. OO and traditional metrics have nearly the same explanatory power, but are better together.

[4] <https://scitools.com/>

4. Bug prediction based on a single metric is unstable, with large variance among projects.

5. Use of string matching without validation in commits messages decreases accuracy.

6. Combining all metrics (source code and change) can improve prediction.

2.5 Comparisons and Measurements

The introduced approaches focused on proposing new bug prediction techniques. There is also work on comparing such techniques, in order to evaluate them and also possibly enhance or combine them. This section discusses comparisons of bug predictors using static and change metrics.

Zimmermann et al. (ZIMMERMANN; NAGAPPAN; ZELLER, 2008) followed the same research line of Nagappan and Ball (NAGAPPAN; BALL, 2005), presented in Section 2.3. The goal of their work is to understand what makes a module defect-prone and how can we avoid it. First of all, four characteristics are discussed, namely Complexity, Problem Domain, Evolution and Process. All of them are related to introducing defects into code. The first is related to the number of details that programmers can pay attention in the same context. The second is related to the fact that some problems are harder than others. Evolution is related to architecture changes due to constant refactoring and always changing requirements with the related code. At last, the development process can compensate for the previously cited problems. For example, if a requirement is changed, but process ensures unit and integration tests, new bugs due to requirement change are detected earlier.

Their own implementation was compared to the work of Basili et al. (BASILI; BRIAND; MELO, 1996) and with the prior work of Nagappan and Ball (NAGAPPAN; BALL, 2005). The first used static code metrics to predict bug density in academic projects, and the second used source code change for forecasting defects for five Microsoft applications (Internet Explorer HTML rendering module, Internet Information Services, Process Messaging Component, Microsoft Direct X, Microsoft NetMeeting). In the first approach, in each project, a source code metric (or a set of) was found to have high correlation with defect density. Notwithstanding, the metric(s) which had correlation to defects were not common among projects, although some intersections were found. The second approach used code churn as an estimator for defects, as previously discussed in

Table 2.6: Change metrics used by Moser et al.

Change Metric
Revisions
Bug Fixes
LOC Added
Average LOC Added
Max LOC Deleted
Code Churn
Max Change-set (files changed together)
Age
Refactorings
Authors
Max LOC Added
LOC Deleted
Average LOC Deleted
Average Code Churn
Average Change Set
Weighted (by LOC added) Age

Section 2.3. The conclusion found is that metrics do correlate with defect density, but also code churn. Because each project has a different approach and performance for each technique, the authors propose that new metrics and more process data (instead of only source code information) should be used in bug prediction. Furthermore, more combined approaches and granularity change (more specific) are encouraged.

Instead of comparing previous implementations, Moser et al. (MOSER; PEDRYCZ; SUCCI, 2008) used static code metrics and change metrics from the Eclipse project. They analysed the behaviour of models built using the static code metrics and change metrics. The goal is to determine which one has a better predictive power. The set of metrics used is the same used by Zimmermann et al. (ZIMMERMANN; PREMRAJ; ZELLER, 2007), 31 in total. As change metrics, 18 metrics were used, they are enumerated in Table 2.6.

Using these metrics, they built the predictors using the following techniques: Naive Bayes, Logistic Regression and J48 Decision Tree. These three techniques are presented by Alpaydin E. (ALPAYDIN, 2004).

They built three predictors, one for static code metrics, other for change metrics and a third combining both. Among the nine predictors built, the best performance was with the J48 Decision Tree using only change metrics. Nevertheless, the performance of the other predictors were also satisfactory. One limitation cited by the authors is that only one project was analysed; therefore no further conclusions can be extracted.

To select which metrics perform better in the bug prediction context, several work

performed a feature selection. Catal and Diri performed a statistical feature selection, using the Chi-Squared technique in same dataset used by Koru et al. (KORU; LIU, 2005). Shivaji et al. (SHIVAJI et al., 2013) performed a feature selection using Gain Ratio, Chi-Squared, Significance, and Relief-F feature rankers. They used the same metrics used by Kim et al. (KIM; WHITEHEAD JR.; ZHANG, 2008) and reported gains of more than 20% in accuracy, precision, recall and f-measure over the existing approaches. Gao et al.(GAO et al., 2011) used genetic algorithms to select the features, they found that removing up to 85% of features does not reduce predictor performance and may even increase it in some cases.

Hall et al. (HALL et al., 2012) presented a systematic literature review comparing 208 articles in the bug prediction context. They found that 108 of it used Eclipse data, moreover 136 used Object-Oriented systems. They found that 23% of the articles used files as granularity for the prediction. Regarding metrics, seventeen used LOC; forty-two used Object-Oriented metrics and 71 used process metrics or SCM. The more used machine learning algorithm used were: Logistic Regression(56), Naive Bayes(33) Random Forest(13). They found that in most of the study, the performance achieved is related to the system size. Moreover, they found that feature selection improves bug prediction performance.

2.6 Cross-Project Defect Prediction

A limitation found in all previously presented approaches the result generalisation. This limitation implies that no method can be applied in different projects. Furthermore, one cannot use other projects for creating a bug predictor for a new project. This section introduces work proposed in this area addressing these limitations.

Jureczko and Madeyski (JURECZKO; MADEYSKI, 2010) have addressed this issue analysing 48 releases of 15 open software systems, 27 releases of 6 commercial systems and 17 academic software projects. They tried to cluster software projects in categories with the same characteristics regarding defect prediction. K-Means (FORGY, 1965) and Kohonen's neural network (KOHONEN, 1982) were used for clustering the projects. As features for clustering, they used static code metrics. After clustering they built a defect predictor for each cluster and for the all projects. All predictors were built using step-wise linear regression and all metrics. They found two significant clusters, but most of the projects were uncovered by it. Some improvements suggested are removing

outliers from clusters and narrowing down the clusters founded.

Later, Zhang et al. (ZHANG et al., 2014) obtained better results building a cross-project predictor. They used projects from SourceForge[5] and GoogleCode[6] as basis. Projects with few commits and short existing time were cut off. Project metrics were calculated splitting the project in frames of six months. A regular expression was used for marking which commits were defective. Context from the project (source code language, use of issue tracking, number of files) was used as features in the predictor. Precision, recall, false positive rate, f-measure, g-measure and AUC (area under the curve) were used for evaluation. For matching metrics for each cluster, they used rank transformation. Static code metrics were extracted using the Understand tool. They achieved similar precision for a within project predictor and a universal predictor, proving that a universal defect predictor can be built. F-measure ranged from 0.50 for within project predictor and 0.46 for universal predictor to 0.55 using project context and process metrics. In an external project, the F-measure ranged from 0.26 to 0.62.

Finally, Lewis et al. (LEWIS et al., 2013) implemented a bug predictor across all Google Projects. The predictor pointed out which files were more bug-prone on the code-review system. They evaluated how developers behaved across four months using the tool. The study concluded that the tool was not ready for use by the developers and that the most required feature was "actionable messages," hence, a way of developers to fix a bug-prone file.

2.7 Approaches Selected for Evaluation

In Table 2.7, we detail the set of approaches selected for comparison in our study. The proposed bug predictors are based on static code metrics—approaches that directly evaluated the correlation between metrics and defects rather than proposed predictors were excluded. They vary mainly in two aspects: (i) used metrics (this table overviews used metric suites); and (ii) investigated learning techniques. These approaches are those evaluated in this work, and hereafter they are referred to as the acronyms introduced in Table 2.7. Two criteria were used for narrowing the approaches presented before: (i) approaches within fifteen years since their publication; and (ii) approaches that used static source code metrics. We included in the study approaches that *also* use change

[5] <http://sourceforge.net/>
[6] <https://code.google.com/>

Table 2.7: Summary of Investigated Approaches.

Approach	**Acronym**	**Metric Suites**	**Learning Techniques**
Gyimothy et al. (GYIMOTHY; FERENC; SIKET, 2005)	GY	CK (CHIDAMBER; KEMERER, 1994)	Decision Trees, Linear Regression, Neural Networks
Jureczko and Madeyski (JURECZKO; MADEYSKI, 2010)	JU	CK (CHIDAMBER; KEMERER, 1994), QMOOD (BANSIYA; DAVIS, 2002), Tang et al. (TANG; KAO; CHEN, 1999), Martin (MARTIN, 1994), Henderson-Sellers (HENDERSON-SELLERS, 1996)	Linear Regression
Kim et al. (KIM; WHITEHEAD JR.; ZHANG, 2008)	KI	Metrics provided by the Understand (TOOLWORKS, 2017) tool	SVM
Koru and Liu (KORU; LIU, 2005)	KO	Halstead (HALSTEAD, 1977), McCabe (MCCABE, 1976)	Decision Tree, K-Star, Random Forests
Moser et al. (MOSER; PEDRYCZ; SUCCI, 2008)	MO	CK (CHIDAMBER; KEMERER, 1994), traditional OO metrics	Decision Trees, Logistic Regression, Naive Bayes

metrics (KIM; WHITEHEAD JR.; ZHANG, 2008), but only static metrics were taken into account. Moser et al. (MOSER; PEDRYCZ; SUCCI, 2008)'s approach extended that proposed by Zimmermann et al. (ZIMMERMANN; PREMRAJ; ZELLER, 2007), by including change metrics and exploring other learning techniques. In our evaluation, we use the static code metrics as well as learning techniques used by Moser et al. They used a subset of metrics from Zimmermann et al.'s dataset because all code metrics would "*involve overly complex models and not yield better performance as most of the measures are highly correlated with each other*."

2.8 Final Remarks

In this chapter, we presented the state-of-art of bug prediction related to the procedural software systems context. We presented approaches using static and change source code metrics, besides mixed techniques and comparisons. Next, we present a comparison of bug prediction approaches that used static source code metrics. Finally, we introduced a set of approaches selected for the study we conduced, described in the next Chapter.

3 EVALUATION OF EXISTING BUG PREDICTION APPROACHES

Given the lack of provision of bug predictors dedicated to procedural software systems, we performed a study to fulfil this gap. We used existing bug prediction approaches with procedural software systems, also making required adaptations in this process, and evaluated and compared the obtained performance. In Section 2.7 we presented a set of bug prediction approaches. These approaches are compared and evaluated regarding their effectiveness and adaptability for procedural software systems in the study presented in this chapter.

This chapter is structured as follows. In Section 3.1 we introduce the study settings, the targeted systems, and the methodology applied. In Section 3.2 we present the results we obtained. Finally, in Section 3.3 we discuss the results found.

3.1 Study Settings

Next, we present the methodology used in this study. We used two research questions as guidelines and analysed both the applicability and performance of approaches in the context of bug prediction for procedural software systems. To address the questions, we perform a machine learning prediction in five different systems. We next detail our study.

3.1.1 Goal and Research Questions

In order to design our study, we followed the GQM (*goal-question-metric*) paradigm proposed by Basili et al. (BASILI; SELBY; HUTCHENS, 1986). According to it, the first step is to specify the goal of the study, which is the following according to the GQM template: *to assess the effectiveness of existing bug prediction approaches in the context of procedural software systems, evaluate existing bug prediction approaches based on static code metrics from the perspective of the researcher in the context of 8 open source and proprietary software projects*. Based on our goal, we derived two research questions presented as follows.

RQ-1: *How bug prediction approaches based on static code metrics can be applied to procedural software systems?* Given that some approaches consider OO-specific

metrics, we investigate the amount of metrics that can be used with procedural software systems and, from metrics that cannot be used, which can be adapted to our context.

RQ-2: *What is the effectiveness of bug prediction approaches based on static code metrics, possibly adapted, with procedural software systems?* Considering the investigated approaches and the set of metrics that can be extracted from procedural software, possibly with adaptations, we measure and compare the effectiveness of each approach. We evaluate both the set of metrics that can be extracted *as-is*, and also a set including adapted metrics.

The metrics used to answer these research questions are detailed in the next section together with our study procedure.

3.1.2 Procedure

Our study procedure is composed of three main steps. We first analysed each investigated approach in order to verify whether their metrics can be used in our study. In the second step, we prepared our dataset, by performing two activities: (i) extraction of defects; and (ii) extraction of static code metrics. Last, we executed all the approaches with our target systems and measured their performance. We next provide details regarding our procedure.

Metric Adaptations. In order to answer our RQ1, we identified all metrics used by each approach, and verified whether they can be extracted from procedural systems. In the cases that they cannot, we adapted the metric calculation using the following mapping between OO concepts and procedural structures. In OO systems, there are classes with attributes and methods, with visibility modifiers. In procedural systems, there are source files (in C, files *.c), which contain global variables and functions, and header files (in C, files *.h), which contain function declarations and possibly global variables. In order to adapt metrics, we map: (i) classes to a combination of header and source files; (ii) public attributes and methods to variables and functions, respectively, declared in header files; and (iii) private attributes and methods to variables and functions, respectively, declared only in source files. Header files are thus considered similar to public interfaces of classes. Inheritance is not mapped, given that there is no similar concept in procedural languages, like C.

For evaluating the applicability of each approach to procedural systems, we measured the following scores.

Applicability with No Adaptations Ratio (A-score$_{NA}$) is the fraction of metrics that can be extracted from procedural systems with no adaptations. It is calculated as follows:

$$\text{A-Score}_{NA} = \frac{|M_P|}{|M|}$$

where M_P is the set of metrics that can be extracted from procedural systems with no adaptations, and M is the set of all metrics used by the approach.

Applicability with Adaptations Ratio (A-score$_{WA}$) is the fraction of metrics that can be extracted from procedural systems with or without adaptations. It is calculated as follows:

$$\text{A-Score}_{WA} = \frac{|M_P \cup M_A|}{|M|}$$

where M_P is the set of metrics that can be extracted from procedural systems with no adaptations, M_A is the set of metrics that can be extracted from procedural systems with adaptations, and M is the set of all metrics used by the approach.

a *Defect and Metric Extraction.* We used commits indicated as *fixes* to identify defects in our target systems, which is an approach typically used in similar work. When a certain file is modified in a fix, it counts as one defect in that file. In order to mine commits, we used two approaches, depending on the tools available (only VCS, or VCS and issue tracker). For projects in which an *issue tracker* was available, we searched for commit messages that contained the issue id of issues that are bugs (and not features). Therefore, the issue category was used to identify commits that are fixes. For projects in which we had no access to an issue tracker, we searched for commit messages that matched a regular expression, which is a method adopted in previous work (JURECZKO; MADEYSKI, 2010; KIM; WHITEHEAD JR.; ZHANG, 2008; ZHANG et al., 2014). Regular expressions were selected for *each project* according to *message patterns* adopted by developers, e.g. in Linux, the regular expression includes "*fix*" and its variants. We have not addressed the bug criticality of the defects because of two issues: (i) defect criticality cannot be extracted directly from commits and (ii) most of the evaluated approaches did not use the bug criticality in their analysis.

Extraction of static source code metrics was performed using the Understand (TOOLWORKS, 2017) static code analysis tool. Metrics for pure C not available in the Under-

stand, or those we adapted, were extracted using: (i) implemented and available scripts[1]; and (ii) open-source tools, namely Cflow, CTags, and CCCC[2]. Cflow provides a call-graph for a C file, which can be parsed and used for extracting the fan-in and fan-out metrics. Complementary, CTags provides the functions and variables available both in header and source files, used for computing the public and private attributes and methods. CCCC, in turn, is a tool for metric measurement for C and C++. The dataset created combining static source code metrics and defect is available online.[3]

Prediction and Evaluation. The evaluation of the effectiveness of each approach was made by building a predictor for our dataset using machine learning algorithms adopted by each investigated approach. Details of how these algorithms were executed are available elsewhere[3], as well as the used dataset. We then measured results with common machine learning scores, also used by most of the evaluated approaches (thus being used as a baseline), and used 10-fold cross-validation. The Scikit-Learn Framework (PEDREGOSA et al., 2011) was used for prediction and score calculation. The scores described in Section 2.1 are applied for evaluation. Now that we have described our procedure, we proceed to the presentation of the target procedural systems of our study.

3.1.3 Target Systems

In order to build our dataset, we selected known open-source procedural systems as well as proprietary systems to which we have access. The latter have the advantage of having an accessible issue tracker, from which we can extract reported bugs and associated commits. Some open-source systems have available issue trackers, but we could not trace bug fixes to the files that changed in commits. In total, our study involved eight target systems, as listed in Table 3.1, from which three are proprietary. In order to be selected, systems had to satisfy two requirements. The first is that selected systems must be implemented in the C language. This is mainly due to two reasons: (i) it simplifies the process of extracting metrics; and (ii) C is the most popular and used procedural language. The second requirement is that information regarding bug fixes should be available, either through commit messages or an issue tracker. Selected applications are from different

[1] <https://github.com/dborowiec/commentedCodeDetector>

[2] Available at <http://www.gnu.org/software/cflow/>, <http://ctags.sourceforge.net/>, and <http://cccc.sourceforge.net/>, respectively.

[3] <http://www.inf.ufrgs.br/prosoft/resources/bug-prediction-procedural/>

Table 3.1: Target Systems.

System	Description	LOC	#Files	#Commits	Bugs (%)
Linux [4]	Operating System	9,434,808 9,529,552	30,058 30,252	560,519	16 22
Commercial System A [5]	Telecom Embedded Application	407,660 509,856	1027 1148	1,027 1148	4 10
Commercial System B [5]	Telecom Embedded Application	337,203 351,923	939 949	2,211	6 12
Commercial System C [5]	Telecom Embedded Application	279,325	394	109	5
Busybox [6]	Operating System Applications	153,448	624	13,891	19
Git [7]	Version Control System	153,855 157,193	500	41,356	16 29
Light Weight IP [8]	Network Stack for Microcontrollers	18,510 32579	89 132	3,658	14 49
CpuMiner [9]	Bitcoin Mining Application	4,455 6,927	20	339	20

domains and have multiple sizes, as can be seen in Table 3.1.

Our target systems include Linux, which is an established operating system and a well documented project. Much work has been developed specifically on bug prediction for Linux (JIANG; TAN; KIM, 2013; TIAN; LAWALL; LO, 2012), but all used change metrics. It is the largest project used in our study. Busybox, in turn, provides operating system tools for embedded systems, being associated with Linux. Git is a widely used multi-platform VCS, with a consolidated development process, while Light Weight IP is a bare-metal network stack, thus being a low-level microcontroller environment, with restricted resources. CpuMiner is the smallest investigated system, but with a complex domain. It consists of a Bitcoin calculator, performing cryptographic calculations. Finally, the commercial applications included in our study consist of logic controllers for network

[4] <https://www.linux.org/>
[5] <http://parks.com.br/>
[6] <https://busybox.net/>
[7] <https://git-scm.com/>
[8] <https://savannah.nongnu.org/projects/lwip/>
[9] <https://github.com/pooler/cpuminer>

devices containing hardware configuration, network protocols, configuration management and user interface.

To investigate a larger dataset, we used more than one system version when possible—some versions were not available and we excluded versions that diverged from the master branch. For each system, we analysed bug fixes of a release i, extracting metrics from the source code of this release and bugs using commits made before the release $i + 1$. Therefore, if we had N releases available, we managed to evaluate $N - 1$ releases. Consequently, for applications with just one analysed release, e.g. Commercial System C, we had, in fact, two available releases. Releases were determined using VCS tags for all systems. We investigated only two Linux versions due to the computational time needed for metric extraction. Moreover, only one version was investigated from the Commercial System C, because it was mostly developed by a third-party company, and the company that gave us access to it is responsible only for evolving it. Therefore, we had no access to the source code repository used during this initial application development. This explains the low number of commits presented in Table 3.1. In this table we also present the percentage of files containing bugs for each system. Because defect criticality is not present in the commit message, and not all evaluated approaches used this information, no information regarding defect criticality was extracted or used to compose the dataset. In next section, we present how the investigated bug prediction approaches performed using these introduced systems.

3.2 Results and Analysis

In this section, we report obtained results, after performing the procedure described above. Results are presented and discussed according to our research questions.

RQ1: How bug prediction approaches based on static code metrics can be applied to procedural software systems?

Each of the five investigated approaches was analysed, and we assessed how applicable they are to our context. In Table 3.2, we list all static code metrics used by the selected approaches. We grouped some sets of metrics, due to space restrictions. The number in parenthesis indicate the number of metrics in each group. Based on Table 3.2, it is possible to observe that all but one of the approaches use metrics that rely on OO concepts. Therefore, we adapted such metrics in order to extract them from procedural systems to build bug predictors—they are described in the last column of Table 3.2.

Adaptations follow the overall mapping rule described in our study procedure.

Considering this information, we classified metrics used by each approach in three classes (column *Class*): (i) those that *can* be extracted from procedural systems, labeled with Y; (ii) those that *cannot* be extracted from procedural systems, labeled with N; and (iii) those that can be extracted from procedural systems only *with adaptations*, labeled with A. Based on this classification, we verified how much applicable each approach is, using the measurements described in the previous section. We present results in Table 3.3, which shows the applicability ratios (without and with adaptations) of each approach. Note that, although MO approach, in theory, uses 31 static code metrics from Zimmermann et al.'s (ZIMMERMANN; PREMRAJ; ZELLER, 2007) dataset, its provided dataset contains only 17 metrics extractable from source code. Other metrics in the dataset are target metrics, e.g. *TrivialBugs*, or rely on CVS information, e.g. *CvsEntropy*, which is not our focus.

Results indicate that the GY, JU, and MO approaches largely rely on OO metrics, while KO uses only metrics that do not rely on OO concepts. With our adaptations, it is possible to use at least 67% (KI has the minimum A-Score$_{WA}$) of proposed metrics of each approach. Given this analysis, we proceed to the evaluation of the effectiveness of each approach.

RQ2: What is the effectiveness of bug prediction approaches based on static code metrics, possibly adapted, with procedural software systems?

We executed each investigated approach, considering different learning techniques, with all target systems (and their different versions). As result, we obtained the precision, recall and F-measure values presented in Figure 3.1 and Table 3.4. On the left hand side charts of Figure 3.1, we show results using only the metrics that can be extracted from procedural systems, while those on the right hand side also include adapted metrics. Table 3.4 reports the mean and standard deviation of the values obtained with our different target systems. For a comparison, we show in the baseline row the results reported by each approach's authors, if they were provided.

Comparing results obtained with and without adaptations, we observed that they are similar to each other—all measurements vary ± 0.05. The differences are so small that they could be due to the randomness of the 10-fold cross validation. This can be seen in the KO approach (A-Score$_{NA}$ = 1.00), which uses no OO metrics, thus both evaluations use the same set of metrics. Consequently, there is evidence that the OO-inspired metrics bring little information associated with defect presence in procedural software systems

Table 3.2: Static Code Metrics used by Bug Prediction Approaches.

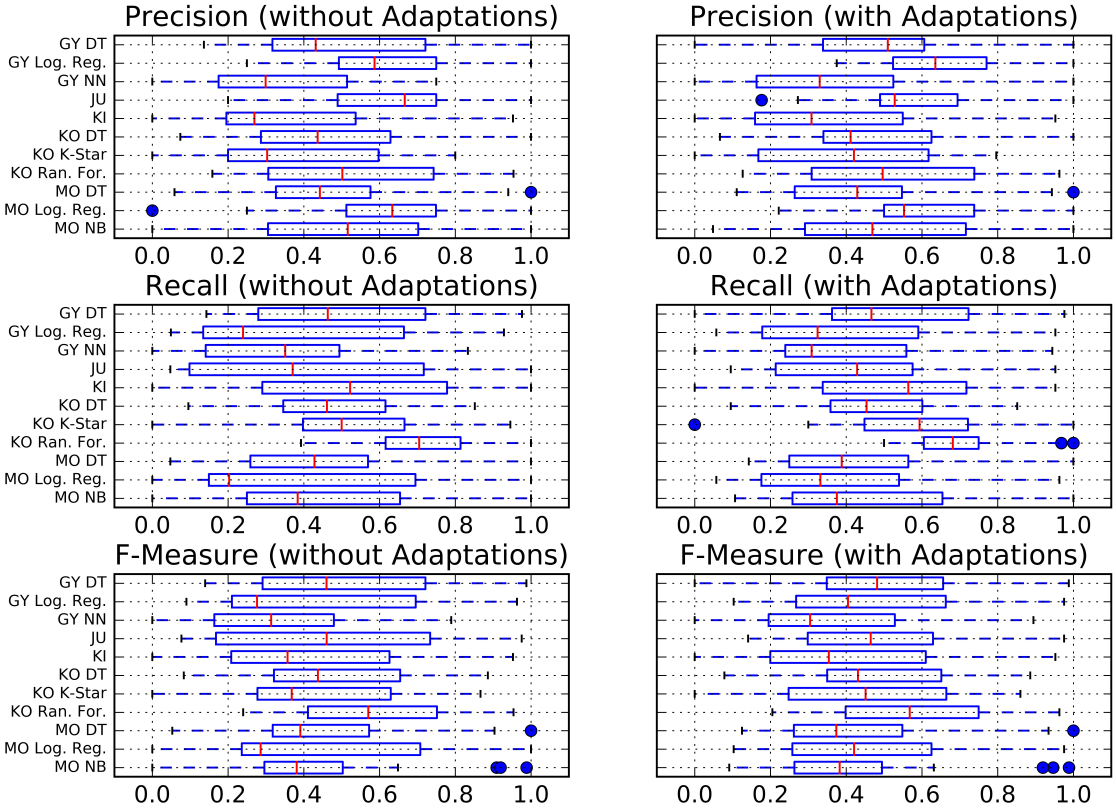
Suite	Metric	GY	JU	KI	KO	MO	Class	Adaptation
	Lines of Code (LOC)	✓	✓	✓	✓	✓	Y	
	Line Count			✓	✓		Y	
	Lines of Comment			✓	✓		Y	
	Lines of Code with Comments				✓		Y	
	Blank Lines			✓	✓		Y	
	Fan-in/Fan-out (2)					✓	Y	
	Branch Count				✓		Y	
McCabe	Cyclomatic Complexity (avg)		✓	✓			Y	
McCabe	Cyclomatic Complexity (max)		✓	✓	✓		Y	
McCabe	Essential Complexity			✓	✓		Y	
McCabe	Design Complexity			✓	✓		Y	
Halstead	Standard and Derived Metrics (12)				✓		Y	
	Understand Metrics (29)			✓			Y	
	Understand Metrics - OO (18)						N	
OO	Number of Inherited Attributes					✓	N	
OO	Number of Inherited Methods					✓	N	
OO	Number of Attributes					✓	A	Number of global variables
OO	Number of Methods					✓	A	Number of functions
OO	Number of Private Attributes					✓	A	Number of global variables not declared in the header file
OO	Number of Public Attributes					✓	A	Number of global variables declared in the header file
OO	Number of Private Methods					✓	A	Number of functions not declared in the header file
QMOOD	Number of Public Methods (NPM)		✓			✓	A	Number of functions declared in the header file
QMOOD	Data Access Metrics (DAM)		✓				Y	
QMOOD	Measure of Aggregation (MOA)		✓				Y	
QMOOD	Measure of Functional Abstraction (MFA)		✓				N	
QMOOD	Cohesion among Methods of Class (CAM)		✓				A	Use of types of function parameters instead of method parameters
CK	Depth of Inheritance Tree (DIT)	✓	✓	✓		✓	N	
CK	Number of Children (NOC)	✓	✓	✓		✓	N	
CK	Coupling between Object Classes (CBO)	✓	✓	✓		✓	A	Functions or global variables from other files used in a target file
CK	Response for a Class (RFC)	✓	✓	✓		✓	A	Number of distinct functions from other files called by a target file
CK	Weighted Methods per Class (WMC)	✓	✓	✓		✓	A	Weighted functions per file
CK	Lack of Cohesion in Methods (LCOM)	✓	✓	✓		✓	A	Global variables count as attributes and functions count as methods
HS	Lack of Cohesion in Methods (LOCM3)		✓				Y	Same as LCOM
	Lack of Cohesion on Methods allowing Negative value (LCOMN)	✓					Y	Same as LCOM
Tang et al.	Inheritance Coupling (IC)		✓				Y	
Tang et al.	Coupling between Methods (CBM)		✓				N	
Tang et al.	Average Method Complexity (AMC)		✓				A	Average complexity of functions in file
Martin	Afferent Couplings (Ca)		✓				A	Number of files that use a pair of header and source file
Martin	Efferent Couplings (Ce)		✓				A	Number of referenced header files

Legend: Y-Yes; N-No; A-Adaptations Required.

Table 3.3: Approach Applicability to Procedural Software Systems.

	GY	**JU**	**KI**	**KO**	**MO**
Extractable Metrics	1	5	37	21	3
Metrics Extractable with Adaptations	5	11	4	0	10
Not Extractable Metrics	2	4	20	0	4
Total	8	20	61	21	17
A-Score$_{NA}$	0.12	0.25	0.60	1.00	0.17
A-Score$_{WA}$	0.75	0.80	0.67	1.00	0.76

Figure 3.1: Effectiveness Measurements by Each Approach.



and increase model complexity. Therefore, they can be discarded. Note that, for some approaches, the number of adapted metrics is not small, as discussed in the previous research question.

The best results were obtained with KO RF (which is based only on no OO metrics), considering F-measure, which combines precision and recall. Two approaches presented the worst results. The first, KI, relies on a large set of metrics. The second, GY, presented worse results only with NN, but results obtained with the other algorithms (DT and LR) are much better, providing evidence of the importance of the selected algorithm. Considering precision and recall individually, it is possible to observe that two other approaches (GY LR and MO LR) have higher precision than KO, at the cost of compromising recall.

Table 3.4: Summary of Effectiveness Evaluation of Each Approach.

	GY DT	**GY LR**	**GY NN**	**JU**	**KI**	**KO DT**	**KO KS**	**KO RF**	**MO DT**	**MO LR**	**MO NB**
Precision											
Without Adaptations	0.52 (0.27)	0.64 (0.21)	0.34 (0.21)	0.62 (0.22)	0.36 (0.30)	0.49 (0.28)	0.39 (0.26)	0.52 (0.26)	0.48 (0.25)	0.64 (0.25)	0.53 (0.30)
With Adaptations	0.52 (0.28)	0.66 (0.20)	0.38 (0.27)	0.59 (0.21)	0.38 (0.30)	0.49 (0.28)	0.40 (0.26)	0.53 (0.26)	0.46 (0.25)	0.61 (0.21)	0.53 (0.30)
Baseline	0.68		0.68	0.82	0.72	0.62				0.65	
Recall											
Without Adaptations	0.51 (0.25)	0.37 (0.30)	0.35 (0.25)	0.42 (0.33)	0.52 (0.35)	0.47 (0.23)	0.55 (0.25)	0.72 (0.18)	0.45 (0.25)	0.39 (0.34)	0.44 (0.29)
With Adaptations	0.51 (0.26)	0.40 (0.29)	0.40 (0.25)	0.44 (0.27)	0.50 (0.30)	0.47 (0.22)	0.57 (0.28)	0.71 (0.15)	0.44 (0.24)	0.41 (0.29)	0.47 (0.28)
Baseline	0.67		0.64	0.89	0.68	0.68			0.42	0.33	0.40
F-Measure											
Without Adaptations	0.51 (0.26)	0.44 (0.29)	0.33 (0.22)	0.47 (0.31)	0.40 (0.31)	0.48 (0.24)	0.44 (0.26)	0.59 (0.23)	0.46 (0.25)	0.45 (0.32)	0.42 (0.26)
With Adaptations	0.51 (0.26)	0.48 (0.28)	0.37 (0.25)	0.49 (0.25)	0.40 (0.30)	0.47 (0.24)	0.45 (0.27)	0.58 (0.23)	0.45 (0.24)	0.46 (0.27)	0.43 (0.25)
Baseline	0.67		0.65	0.85	0.69	0.65				0.36	

Legend: DT-Decision Trees; KS-K-Star; LR-Logistic Regression; NB-Naive Bayes; NN-Neural Networks; RF-Random Forest.

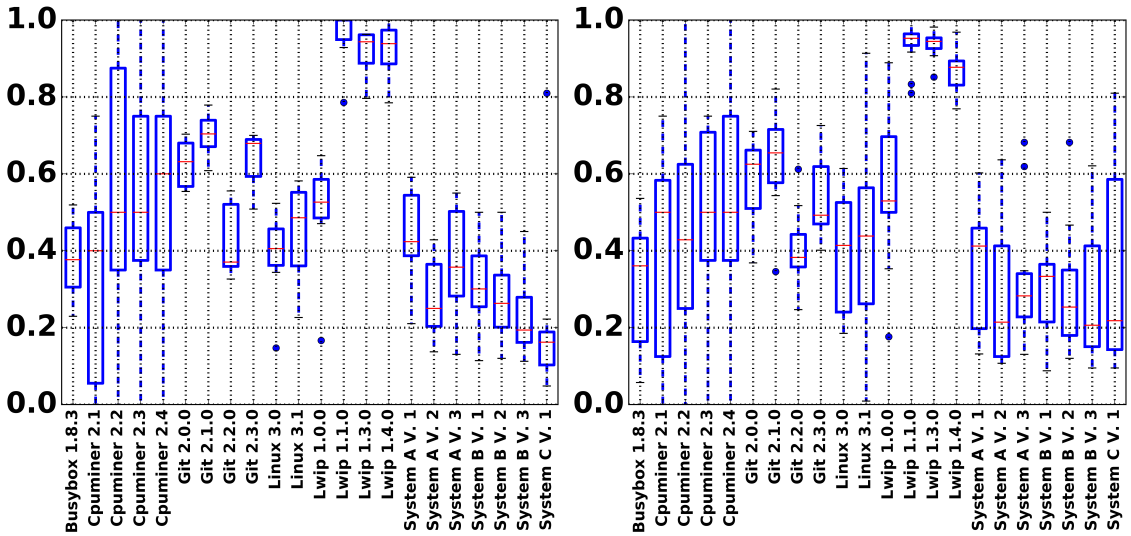
With respect to the GY approach, the approach that with DT obtained the second best results, it is interesting to highlight that it has only one metric used without adaptations: LOC. Other approaches with best results also use this metric. However, the other metrics used by GY slightly improved both precision and recall for LR and NN but for DT, which obtained the best results for GY, they remained the same. Therefore, there is evidence that LOC plays a key role in our context. Although KI also uses LOC, the other used metrics might have introduced noise in the model used for prediction.

In addition to comparing results across different approaches, we also investigated how our measurements vary across different target systems, as presented in Figure 3.2. We observed that commercial applications presented worse results in comparison with open source systems. This observation holds even for the Commercial System C, which has a low number of commits as described in Section 3.1.3. Analysing results, we considered two hypotheses: (1) there are differences in the system datasets that has impact in the construction of the prediction model; and (2) coding standards and practices adopted by developers of our commercial applications are less suitable for bug prediction. The number of investigated systems is not large enough to allow us to reach a conclusion regarding this and, therefore, further studies could help clarify this issue. However, it is

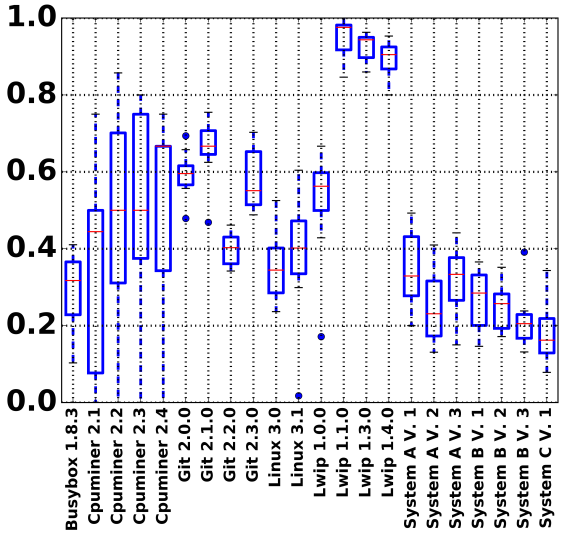
Figure 3.2: Effectiveness Measurements by Target System.

(a) Precision

(b) Recall



(c) F-Measure



possible to observe in Table 3.1 that the percentage of files with bugs is much lower in commercial applications. Consequently, the highly unbalanced classes in these datasets make the prediction model construction more difficult. Moreover, although our proprietary applications are maintained by the same company, they were originally developed outside this company (not by the same provider). Consequently, hypothesis (2) is less likely to be true.

With LwIP, an open source system, results obtained were impressively high, for three of its four analysed versions. Based on an analysis of LwIP's commits and release information, our hypothesis is that, again, the balance between the dataset classes is the reason for these results. In the version in which LwIP performed significantly well, the

number of files with bugs are similar to that of files with no bugs. Therefore, this facilitates the machine learning process.

3.3 Discussion

We now discuss relevant issues that emerged from the analysis of our results. These issues are related to the differences of results obtained using different sets of metrics or systems.

Use of Adapted Object-oriented Metrics. Based on our results, we observed that all metrics adapted from OO metrics were not helpful to predict defects in procedural systems. On the one hand, this was expected given that programming practices are different in procedural and OO systems. Moreover, metrics that are associated with inheritance could not be adapted, because this concept does not exist in procedural systems, and such metrics may be relevant to be used in combination with other OO metrics to build predictors. On the other hand, some of the metrics, such as CBO, capture coupling and cohesion in classes, while our adapted metrics capture them in source files. Therefore, they could have been helpful. Although coupling is useful in predictors for OO systems, we could not observe this in our study. Possibly, this metric alone may be not enough for the predictor, and it should be combined with other metrics that cannot be adapted, e.g. those related with inheritance, so that a proper correlation with bugs is found.

Open-source vs. Proprietary Systems. As discussed before, the results obtained with open-source and proprietary systems are different. This can be seen in Figure 3.2. As discussed before, a potential explanation is that these differences are due to the unbalanced classes (i.e. number of files with bugs and with no bugs) in the proprietary systems' datasets. Because of the low number of instances of files with bugs, it is difficult to the learning technique to build a model that distinguishes these two classes. This is actually a general problem of bug prediction because, typically, the number of files with bugs is relatively small. Moreover, datasets usually contain noise, because the bugs are not those that exist, but those that were identified. Therefore, techniques that address these issues are essential and should be explored in the context of bug prediction.

Another possible explanation for the differences between results is the development process adopted in open-source and proprietary systems. In the former, developers have their own agenda (most of them are volunteers or employers of different companies), while in the latter changes can be limited to a set of files in each software release, because

it may be focused on a particular system feature.

Effectiveness with Object-oriented vs. Procedural Systems. In Table 3.4, we presented previously reported results for us to have as a baseline. Note that the results reported by the KI approach include change metrics, and KO's evaluation included procedural and OO systems, made available by NASA. As can be seen, for all approaches but MO, our results are worse. The only approach that presented results similar to ours is KO but with a different learning technique (our results with RF is similar to the baseline performance, which used DT). All approaches performed better with the original set, indicating that obtained results may not generalise to systems other than those obtained with the dataset used for evaluation. Moreover, the differences between results can also be explained using the arguments we presented above, when we compared results using adapted OO metrics—use of a subset of metrics and different meanings of the relationships between the metrics and defects.

In addition to these issues that might explain difference between results, the typical application domains of procedural systems may also be an issue. Such domains often involve low level details or complex calculations. Consequently, complexity metrics may be more correlated with defects than metrics associated with aspects more relevant to OO systems, such as response for a class or number of children. In fact, previous work indicates that there is a correlation between code complexity and defects (TASHTOUSH; AL-MAOLEGI; ARKOK, 2014). Moreover, variability is often present in such application domains, which results in the inclusion of macro definitions from the C language. This may compromise code legibility and make it more fault-prone.

A relevant observation from the results of the GY approach is the importance of the lines of code (LOC) metric for building a bug predictor for procedural systems. Using only LOC for identifying fault-prone files is almost as good as using other metrics, confirming the correlation between LOC and defects (JAY et al., 2009; NAGAPPAN; BALL; ZELLER, 2006). This may be an indication that approaches are overfitting their models with large amounts of metrics, which do not bring useful information. Therefore, studies that identify which metrics are in fact responsible for good prediction results, both for OO and procedural systems, are needed. This also helps reduce the cost of metric extraction.

3.4 Final Remarks

In this chapter, we presented a comparison of existing bug prediction approaches using static source code metrics adapting these approaches, i.e. the source code metrics if needed, to procedural software systems. We found no performance increase by adapting existing static source code metrics. Moreover, the best performing approach uses only metrics directly applicable to procedural software systems. Based on that, we next propose a set of attributes which can be extracted from procedural source code and increase bug prediction performance in this context.

4 QUALITY FEATURES FOR BUG PREDICTION IN PROCEDURAL SOFTWARE SYSTEMS

Based on our findings in Chapter 3, we concluded that the adaptation of OO static source code metrics does not improve bug predictors performance. In this chapter, we propose a set of features to be used for bug prediction that can be extracted from the source code of procedural software systems. The selected element granularity is that used in Chapter 3, i.e. file, because: (i) it is more specific than a system module, which provides less useful information to make decisions regarding the system; and (ii) more information can be extracted from files than from single procedures, thus helping build a prediction model.

We selected four features to be investigated in our work based on two main criteria. First, we selected features to which we can find a rationale that explains the relationship between them and faults, i.e., there are arguments that justify the investigation of those features, considering our target context. Second, we aimed at exploring features that can be extracted from tools typically adopted in the existing development processes, so that the impact of obtaining them would be reduced in existing projects.

We next detail each of our selected features, detailing the reason for including them in the study and also how we made them operational but, before, we introduce an example in Figure 4.1 that is used to explain our features. In this example, there are functions and a struct to represent Cartesian points developed to calculate the distance between two points. A possible debug feature can be enabled.

4.1 Compiler Warnings

A tool used in every software project is a compiler. Compilers significantly changed since they were conceived, evolving from simple transformers from a higher-level to a lower-level code to tools that not only do so, but also provides warnings of points in the code that can potentially lead to faults. An example is casting a 64-bit floating point to 16-bit signed integer value, which can lead to fatal failures if done without precaution, as it was the case with Ariane 5 (BEN-ARI, 2001).

Therefore, *compiler warnings* is the first feature we describe to be incorporated

Figure 4.1: Running Example.

(a) main.c

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <stdlib.h>
4  #include "point.h"
5
6  int display(char *str) {
7  #ifdef DEBUG
8   debug(__FUNCTION__);
9  #endif
10
11  print(str);
12 }
13
14 void debug(char *s) {
15  print(s);
16 }
17
18 int distance(struct point *p1, struct point *p2) {
19 #ifdef DEBUG
20  debug(__FUNCTION__);
21 #endif
22  return sqrt(pow((p1->x - p2->x),2) +
23  pow((p1->y - p2->y),2));
24 }
25
26 #define ZERO 0
27 #define ALLOC(size) malloc(size)
28
29 int main(int argc, char *argv[]) {
30  struct point* p1, *p2;
31  int d;
32  char s[100];
33  p1->x = ZERO;
34  p2->x = ZERO;
35  p2 = ALLOC(2);
36  p2->x = atoi(argv[1]);
37  p2->y = atoi(argv[2]);
38  d = distance(p1, p2);
39  sprintf(s,"Distance: %d",d);
40  display(s);
41 }
```

(b) io.c

```
1  #include <stdio.h>
2  int print(char *s) {
3   printf(s);
4  }
```

(c) io.h

```
1  int print(char *s);
```

(d) point.h

```
1  struct point {
2    int x;
3    int y;
4  };
```

to bug predictors. Besides being intuitively associated with possible faults, Moser et al. (MOSER; RUSSO; SUCCI, 2007) provided evidence that there is a positive correlation between software defects and compiler warnings. Moreover, these warnings are

information that can be easily extracted given that it is a mandatory tool used in software development.

To practically use compiler warnings as a feature of bug predictors, we count the number of warnings present in each source file. In the particular case of the C language, the GCC[1] compiler is widely adopted. Therefore, this is the compiler investigated in our further presented evaluation, although we are aware that there are alternative options, such as the Clang/LLVM suite[2]. GCC provides many levels of warnings: (i) standard; (ii) medium (with the option `Wall` enabled); and (iii) all warnings (with the option `Wextra` enabled). There are occurrences of all these different levels of warnings in the file `main.c` of our running example. In lines 33 and 34, there is a warning displayed in all levels (*request for member 'type' in something not a structure or union*). In lines 12 and 41, there is a warning displayed only with either the `Wall` or `Wextra` option enabled (*control reaches end of non-void function*). Finally, the warning shown in line 29 (*unused parameter*) is displayed only if the `Wextra` option enabled.

Although promising, this feature has a limitation. In many integrated development environments (IDEs), compiler warnings are automatically displayed to developers. Hence, developers may have already processed them, and left in the code only the warnings that are associated with safe commands.

4.2 Static Code Analysers

Static code analysers are helpful tools that analyse source code without the need for executing it, and typically report issues that are considered poor programming practices and can thus lead to faults. Such reported issues range from allocation problems to non-compliance with standards. Developers may introduce these issues due to, e.g. inexperience, time pressure or even intentionally. In the last case, there is a justification for introducing issues and those reported are ignored.

However, unintentionally introduced issues can be associated with faults, if they are not fixed. So we use the output of static code analysers as a feature to be investigated in bug predictors. Although there is evidence (RAHMAN et al., 2014) that the use of static analysis does not improve bug predictors based on machine learning, such evidence was obtained in the context of object-oriented systems. As our investigated context is pro-

[1] <https://gcc.gnu.org/>
[2] <http://llvm.org/>

cedural software systems, this might not be the case, given that there is a range of coding issues reported only in this context, mainly for the C language, which provides developers with freedom, for example, with type casting. Nevertheless, if a project incorporates the use of static code analysers as part of the development process, so that developers are always aware of coding issues, the use of such tools is expected to be not helpful for bug prediction.

From the many static code analysers available, we focused on exploring those that do not require compilation or customisation's. If so, this would make their adoption more difficult and human-dependent, which could prevent their practical adoption. Given this criterion, we selected two open-source static code analysers, namely CppCheck[3] and Uno[4].

Examples of coding issues present in Figure 4.1a reported by such tools in our running example are in line 29 (*local variable never used*) and in lines 33, 34 and 38 (*uninitialised variable*).

4.3 Duplicated Code

Duplicated code (DUP) is a widely known poor programming practice, being reported as code smell by Fowler (FOWLER; BECK, 1999). Code smells are symptoms present in the source code, which are usually associated with larger code problems. They can be used as hints of points in the code that should be refactored. Moreover, there is evidence that duplicated code is correlated to faults (JUERGENS et al., 2009; KAMEI et al., 2011; MONDEN et al., 2002). Therefore, the presence of duplicated code in portions of a given system is a potential candidate of feature to be included in bug predictors.

Evaluating duplicated code can be done at different levels of granularity, ranging from the analysis of sequences of characters to file content as a whole. The finer the selected granularity, the higher the computational cost to evaluate duplication. Using a fine-grained granularity can compromise the technical feasibility of calculating duplicated code in a large-scale system. Within a file, it is less likely that there are large portions of duplicated code, because it does not make sense to copy a procedure to the same file. However, this may occur across different files. Consequently, we evaluate duplicated code from two perspectives: internal and external, explained as follows together with how

[3] <http://cppcheck.sourceforge.net/>
[4] <http://spinroot.com/uno/>

we measure them. We propose a particular measurement of duplicated code, instead of using clone detection tools, because the tools available require (i) a significant amount of memory and processing power, making the computation practically unfeasible for large systems (e.g. Linux, which is used in our evaluation); and (ii) a significant amount of effort to setup, requiring changes in the build process of each project.

Internal duplicated code (IDC) is a metric that refers to the number of portions of code that are equal within a file, considering the character level of granularity. In order to detect duplicated portions of code, the minimum size of a sequence of characters must be specified, so that it is considered a case of code duplication. This minimum size is a parameter of the IDC metric. The value of this metric is then the number of cases of duplicated code, which are above the specified threshold. In Figure 4.1, there are two cases of duplicated code, if the threshold were 14: (1) lines 18 and 30, (2) lines 14 and 26; and (3) lines 28 and 42. If the threshold were 20, only (1) would be a case of duplicated code. The internal duplicated code metric only counts each pair of duplicated code once, and sub-sequences of characters are also not counted. Moreover, spaces and indentation are not taken into account.

For analysing *external duplicated code* (EDP), we use a courser-grained granularity, namely lines of code. This metric corresponds to sum of the amount of code duplicated in other files. For every pair of files, we calculate the amount of shared lines of code. Therefore, the external duplicated code metric is defined as follows.

$$EDP(f_i) = \sum_{f_j \in F, f_i \neq f_j} \frac{shared\_loc(f_i, f_j)}{min(loc(f_i), loc(f_j))} \tag{4.1}$$

where f_i and f_j are files in the file set F, $shared\_loc(f_i, f_j)$ is the number of shared lines of code between f_i and f_j, and $loc(f)$ is the number of lines of code of a file f. In order to support the extraction of the EDP metric, we use the `comm` utility, available in Unix-like operating systems. We used it instead of clone detection tools because the tools available were (i) not easy to setup – requiring changes in the build process of each project, (ii) have an exponential complexity (pair-wise comparison of each file), hence requiring too much CPU/memory, turning the computation for big systems, e.g. Linux, unfeasible.

Table 4.1: Types of Preprocessor Directives.

Directive Type	Example	Description
Include	`#include <stdio.h>`	Includes the text of the referred file
Constant	`#define SIZE 100`	Creates a symbolic name for a constant number
Macro	`#define CtF(C)` `C * 1.8 + 32`	Creates a symbolic name for an expression (example converts Celsius to Fahrenheit)

Key	`#ifdef __mobile__`
	`[ ... ]`
	`#endif`
		Removes the code in between the directives, if the given key is disabled

4.4 Preprocessor Usage

Preprocessors, in C, processes the written source code before it is compiled, to replace written text, i.e. preprocessor *directives*, for the text that will be actually compiled. This step, which occurs before the compilation, provides developers with the ability to include header files, macro definitions and expansions, and conditional compilation. In Table 4.1, we detail the most common types of directives. Even though constants are also considered macros, we distinguish them because macros are more complex than a simple constant declaration. Keys are used for conditional compilation.

Although this gives pre-compilation flexibility for developers, it can decrease code legibility, leading to maintainability problems and consequently faults (SPENCER; COLLYER, 1992). However, it is not reasonable to expect that this would reduce the use of directives, because it is a powerful tool to address configurable software (MEDEIROS et al., 2015). Given that there is evidence that software variability and defects are related (NIE; ZHANG, 2011)—measuring variability in terms of options (keys) used in the compilation, which are associated with directives used for conditional compilation—analysing directives can give us information regarding fault-proneness. We thus count the number of directives used in each file, according to their type, and used them as metrics for bug prediction. There are other types of directives, e.g. to specify user-defined compilation errors (`#error`), but we limit ourselves to focus on those that are most commonly used.

The four directives presented in Figure 4.1 have different impact on the source code. The use of of includes accounts for the coupling of the file, i.e. the higher the number of includes, the higher the number of functions the developers use or can use within

that file. Macros decrease legibility, because often some functions, casts or loops are hidden within a chain of macros. Constants, despite helping legibility, have better forms of implementation than using preprocessor [5], thus are analysed for correlation with defects. The constant and macro attributes are counted by occurrence recursively in headers within the same project. Hence, values of a header file are added to the source file including it. The number of `#include` and keys are also counted by occurrence, but using only those present in the file, except that the `#undef` is not counted given that it always co-occur another key.

In our example, there are the following preprocessor directives in the `main.c` file: (i) four includes (lines 1–4); (ii) one constant (line 26); and (iii) one macro (line 27); and (iv) one compilation key (line 19).

4.5 Final Remarks

We presented a set of software quality attributes in this chapter. These attributes are extractable from procedural software systems and provide descriptions of aspects that are not covered entirely by traditional metrics such McCabe or Halstead. In next chapter, we describe our evaluation of these attributes combined with the traditional metrics in a subset of the systems used in Chapter 3 analysing their impact on bug prediction performance individually and combined in the context of procedural software systems.

[5] <http://stackoverflow.com/questions/1674032/static-const-vs-define-vs-enum>

5 EVALUATION

The features proposed in Chapter 4 to increase bug predictor performance in the procedural software system context are based on an intuitive reasoning considering our practical experience and empirical investigations. Therefore, there is a need for evaluating their effectiveness to obtain concrete evidence of their contribution to bug prediction. In this section, we detail an empirical study performed to do so.

We next present our study settings in Section 5.1. In Section 5.2 we detail the procedure followed. The targeted systems are presented in Section 5.3. Then, we present and analyse obtained results in Section 5.4. Finally, we point out threats to the validity of our study in Section 5.5.

5.1 Goal and Research Question

The main goal of bug predictors is to identify fault-prone elements. To assess how our proposed features support this task, as in our study in Chapter 3, we designed our study using the Goal-Question-Metrics (GQM) (BASILI; SELBY; HUTCHENS, 1986) paradigm. Therefore, we first define our study goal that, following the GQM template, is *to validate the relevance of our proposed features to bug prediction, evaluate their effectiveness in identifying fault-prone files from the perspective of the researcher in the context of five procedural software systems*. Based on this goal, we derived two research questions:

RQ-1: *How effective are our proposed features to identify fault-prone elements compared to previously investigates features?*

RQ-2: *What is the best set of features for predicting bugs in the context of procedural software systems?*

In order to answer both research questions, we rely on the metrics typically used to measure effectiveness in binary classification problems, which is our case given that we classify each project file as with or without bugs. Such metrics are: (i) *precision*, which presents the relation between predicted bugs (PB) that are correctly predicted considering the known bugs (KB) ($Precision = |PB \cap KB|/|PB|$); (ii) *recall*, which calculates the portion of known bugs that are correctly predicted ($Recall = |PB \cap KB|/|KB|$);

and *f-measure*, which combines them, calculated according formula presented next.

$$\text{f-measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{5.1}$$

Our study uses as a baseline the set of features used in Koru and Liu's (KORU; LIU, 2005) work. Such set of features (detailed in next section) was chosen due the best performing approach found in Chapter 3, which compared the effectiveness of different existing bug prediction approaches that rely on static code metrics in the context of procedural software systems. Koru and Liu's (KORU; LIU, 2005) approach performed best, together with the use of the random forests (BREIMAN, 2001) machine learning algorithm. This selected set of features includes only metrics that can be extracted from procedural systems, as opposed to object-oriented metrics, e.g. coupling between objects (CBO).

Next, we describe the procedure of our study, detailing how we use the selected metrics to answer our research questions.

5.2 Procedure

Our study procedure consists of three key steps: (i) construction of the dataset (which includes feature and bug extraction); (ii) execution of the machine learning algorithm and machine learning scores collection; and (iii) result analysis. We next detail each of them.

5.2.1 Dataset Preparation

The dataset preparation requires us to extract the selected features (proposed and those used as a baseline) from a given software project, associated with each file of a procedural software system, and classify such files as a file with or without bugs. This is the information that is required to build and evaluate a prediction model. In order to obtain features, we extracted a given version of a project's source code from a version control system (VCS), and calculated such features, which are all quantitative. Most of the features are obtained with the Understand[1] static code analysis tool. We also use the

[1] <https://scitools.com/>

Table 5.1: Analysed Features.

Feature	Metric	Suite	Description
DOr_H	DOr_H	Halstead	Distinct Operators
OpC_H	OpC_H	Halstead	Operands Count
DOp_H	DOp_H	Halstead	Distinct Operands
LEN_H	LEN_H	Halstead	Program Length
VOC_H	VOC_H	Halstead	Program Vocabulary
VOL_H	VOL_H	Halstead	Volume
DIF_H	DIF_H	Halstead	Difficulty
EFF_H	EFF_H	Halstead	Effort
TIM_H	TIM_H	Halstead	Time
BUG_H	BUG_H	Halstead	Bug
LOC	LOC		Lines of Code
CyC_{Mc}	CyC_{Mc}	McCabe	Cyclomatic Complexity (sum)
EsC_{Mc}	EsC_{Mc}	McCabe	Essential Complexity (sum)
CPL	CPS		Compiler Warnings Standard
	CPW		Compiler Warnings Wall
	CPX		Compiler Warnings Wextra
SAL	SAU		Static Analysers Uno
	SAC		Static Analysers CppCheck
DUP	IDC		Internal Duplicated Code
	EDC		External Duplicated Code
PRE	PRC		Preprocessor Constants
	PRM		Preprocessor Macros
	PRK		Preprocessor Keys
	PRI		Preprocessor Includes

tools reported in the previous section to calculate our proposed features.

We detail in Table 5.1 all features included in our dataset, which consists of Koru and Liu's (KORU; LIU, 2005) features and those proposed in 4. The former is composed of the Halstead's metrics suite (HALSTEAD, 1977), Lines of Code (LOC), and two McCabe's complexity metrics (MCCABE, 1976). As can be seen in Table 5.1, we distinguish the term feature and metrics, and this distinction is relevant only for our proposed features. We use the term *feature* to refer to our general properties examined in the code, such as duplicated code, while *metric* is used describe the concrete value extracted from the code. This differs from the terminology from the machine learning community, which would refer to our metrics as features. This distinction is important in the next step of our procedure. In addition to the feature and metric names, we also provide in Table 5.1 the suite to which they belong and a short description.

The dataset preparation also includes classifying each file as with or without bugs. This is done by using information either available in an issue tracker system, such as

Bugzilla, or a VCS. If the former is used, we retrieve from the issue tracker closed issues classified as bug fixes, and classify files related to the commit associated with the issue as *with bug*. If the latter is used, we analyse VCS commit messages, such as in previous work (KIM et al., 2007). More specifically, we search, in commit messages, for terms that indicate that a commit is associated with a bug fix. For example, in one of our target systems, Linux, commits marked with "fix" are related to solving bugs. We preferably use issue trackers to identify files with bugs but, when they are unavailable, we use VCS. We provide the complete dataset and the methodology used to generate it online.[2]

5.2.2 Execution Details of the Classification Algorithm

With a built dataset, we are able to run a machine learning algorithm. As aforementioned, among the many classification algorithms available, we adopted the random forests algorithm, as presented in Chapter 3. In particular, we used the Random Forest implementation of the Scikit-Learn Framework (PEDREGOSA et al., 2011), executed with the default parameters. This implementation also provides the precision, recall, and f-measure metrics. To obtain such measurements, we used 3-fold cross-validation. We did not use 10-fold cross-validation due to the size of some of our target systems, thus demanding significant computational cost.

In order to evaluate the effectiveness of our proposed metrics and identify the best set to build the prediction model, we used a brute-force feature selection (WITTEN; FRANK, 2005) process. This means that we executed the classification algorithm using different subsets of features, and measured the effectiveness of each. By using a brute-force approach, we investigated all possible subsets of features, i.e. all subsets with one feature, two features, and so on. This allows us to obtain optimal results, rather than executing the algorithm with only some subsets selected using a heuristic, for example.

However, a brute-force approach comes with a computational cost, because the number of subsets grows exponentially in terms of the number of investigated features. Given the high computational costs of the feature selection process and that we are interested in evaluating our proposed features in a general way, the different metrics associated with each of our proposed features are treated as a group. Therefore, in the feature selection process, we either include all metrics associated with a given feature, or none of them is used. Consequently, our feature selection process investigates subsets of 17 features in

[2] <http://www.inf.ufrgs.br/prosoft/resources/bug-prediction-procedural/>

total, which results in $2^{17} - 1 = 131,071$ subsets.

5.2.3 Result Analysis Method

The execution of the feature selection process, with all possible feature subsets, leads us to a large amount of data to analyse. Therefore, we used a systematic process to analyse them. For RQ1, we look at subsets that have only one feature and compared them. We not only compare them in terms of the selected measurements, but also their position in a ranking ordered according to each of these measurements. The analysis of results to answer RQ2 is more complicated than RQ1 because each target project in our study is associated with different measurements. Therefore, different feature groups may achieve the best measurements for different projects.

For selecting the best feature subset, alternative perspectives may be adopted. It is possible to take the average of the value obtained for each project, but this means that a high value obtained with one project can compensate a low value obtained with another. Therefore, we selected a collection of best subsets, using alternative criteria adopted in the context of social choice (MASTHOFF, 2011). These criteria are described below, which are used to select the best subsets considering each of our measurements (precision, recall, and f-measure).

Average. The Average criterion selects the feature subset that has the best average value.

Borda Count. The Borda Count criterion builds a general ranking of feature subsets based on rankings made for each target project. For a given project, a ranking is built and the worst set of features gets a value 0, the second worst gets 1, and so on until the best set of features gets the maximum value (i.e. the number of subsets in the ranking). A final value is given to each feature subset as the sum of the values in each project ranking. The general ranking is built based on such final values, in descending order. The best subset is the first in the general ranking.

Copeland Rule. The Copeland Rule criterion calculates how many times a feature subset is better than another using a majority vote (wins), and how many times it is worse (losses). Then, the best subset is the one that has the maximum difference between wins and losses.

Least Misery. The Least Misery criterion associates with each feature subset the worst

obtained value, considering all projects. Then, the best subset is that with the best value (from these worst values).

5.3 Target Projects

We selected five systems to be evaluated in our study, from which two are proprietary and three are open source systems. These systems were selected based on two requirements: (i) the systems must have at least two versions released, so that the earlier version allows extraction of features and the later version is used to extract bugs; and (ii) there must be a way to identify bug fixes in commit messages, when an issue tracker is not available or do not trace a bug fix to changed files. We present details of the selected systems in Table 5.2. The largest system selected to be part of our study is Linux, which is widely used and has a well-structured development process. The smallest system is Busybox, which creates binaries of operating system infrastructure. We also selected Light Weight IP (or LwIP), which is a networking stack used in environments without operating systems (i.e. bare metal). The remaining systems are proprietary systems (referred to as System A and System B, due to a confidentiality agreement), which are core applications of telecommunication equipment, responsible for handling network protocols, as well as configuring and monitoring equipment. These two proprietary systems are maintained by the same company, but their first version(s) was developed by third-parties. The used systems are a subset of the systems used in the study described in Chapter 3. In this study we used a smaller number of systems because the complexity for executing the feature selection and evaluating its results in more systems would not be done in a timely fashion, either to computing power, either to data examination. Moreover, we only used one version of each system, in contrast to the two to four versions used previously.

5.4 Results

We next present and analyse our experiment results, with the aim of answering our research questions. We detail collected data that provide foundations for our observations, and the complete obtained results are available in Appendix B. Results presented in this section refer to the average of the values obtained for individual target projects, for each measurement.

Table 5.2: Target Projects.

Project	Description	LOC	Files	Commits	Extracted Defects	Version
Linux	Operating System	10,290,337	20,962	560,519	16%	3.0
System A	Telecom Embedded Application	437,530	394	1,060	5%	*Confidential*
System B	Telecom Embedded Application	336,354	497	2,211	7%	*Confidential*
Busybox	Operating System Applications	156,314	627	13,891	19%	1.8.3
LwIP	Network Stack for Microcontrollers	53,596	95	3,658	14%	1.0.0

5.4.1 Individual Feature Effectiveness (RQ1)

To investigate the effectiveness of each individual feature, including those proposed, in the identification of fault-prone files, we analysed results obtained with each feature subset containing a single feature. Table 5.3 details results obtained with all individual feature subsets ordered in a descending order according to each measurement. When results are the same, we use first the standard deviation (the lowest, the better) and then the average ranking position obtained across different projects (the highest, the better) to determine the position in the ranking. This analysis helps evaluate the contribution of each feature for bug prediction in our context, i.e. procedural software systems.

As expected, results are low, because a single feature is likely not enough to correctly predict bugs. Interestingly, this is more evident in recall (0.04–0.26) than in precision (0.05–0.50). This may occur because high values of certain features—e.g. due to (very) long (high LOC) or complex (high EsC_{Mc}) files, or files in which there is a high number of preprocessors (high PRE)—may lead to the introduction of defects. However, not so high values depend on other factors to increase the probability of defect introduction. Therefore, a classification algorithm may be unable to classify such files as fault-prone based solely on the information of a single feature.

The feature that obtained best results, both for precision (0.50) and recall (0.26) and consequently for f-measure (0.33), is LOC. In fact, LOC has been acknowledged to have a high correlation with software faults in previous work (JAY et al., 2009; NAGAP-

Table 5.3: Individual Feature Analysis.
M—Mean, SD—Standard Deviation

(a) Precision

Feature	M	SD
LOC	0.50	0.16
EsC_{Mc}	0.50	0.29
CyC_{Mc}	0.44	0.28
PRE	0.41	0.11
DUP	0.41	0.20
OpC_H	0.39	0.33
CPL	0.35	0.32
SAL	0.31	0.20
DOp_H	0.27	0.37
BUG_H	0.20	0.24
VOL_H	0.20	0.28
VOC_H	0.12	0.10
LEN_H	0.07	0.07
TIM_H	0.06	0.07
EFF_H	0.06	0.07
DIF_H	0.06	0.08
DOr_H	0.05	0.07

(b) Recall

Feature	M	SD
LOC	0.26	0.08
EsC_{Mc}	0.20	0.17
DUP	0.17	0.10
PRE	0.16	0.07
SAL	0.16	0.11
CPL	0.16	0.23
CyC_{Mc}	0.10	0.15
BUG_H	0.09	0.11
VOC_H	0.08	0.08
VOL_H	0.08	0.10
DOp_H	0.08	0.11
DIF_H	0.08	0.12
OpC_H	0.06	0.06
TIM_H	0.06	0.12
DOr_H	0.05	0.06
EFF_H	0.04	0.07
LEN_H	0.04	0.07

(c) F-Measure

Feature	M	SD
LOC	0.33	0.07
EsC_{Mc}	0.25	0.20
PRE	0.22	0.08
DUP	0.22	0.10
SAL	0.19	0.13
CPL	0.18	0.27
CyC_{Mc}	0.12	0.14
BUG_H	0.10	0.10
VOL_H	0.10	0.12
OpC_H	0.09	0.05
VOC_H	0.09	0.08
DOp_H	0.07	0.08
DIF_H	0.07	0.09
DOr_H	0.05	0.07
TIM_H	0.05	0.09
LEN_H	0.04	0.07
EFF_H	0.04	0.07

PAN; BALL; ZELLER, 2006) and in the findings described in Chapter 3. This indicates that this feature should be included in most of the bug predictors, unless they are system-tailored. The feature that consistently obtained second best results is EsC_{Mc}, which has a similar precision to LOC (0.50), but lower recall (0.20). However, the precision obtained with EsC_{Mc} is mainly due to the precision obtained with some target projects. It achieved precisions of 0.87, 0.80, 0.38, 0.37, and 0.10 to LwIP, System B, Busybox, Linux, and System A, respectively. This precision variation can be observed in its high standard deviation (0.29). Consequently, the relationship between EsC_{Mc} and faults is highly system-dependent.

Regarding our code quality features (highlighted in Table 5.3), it can be seen that they obtained high results in comparison with many of the features, being always ranked in the first eight positions (first half of the positions). Considering f-measure, which balances precision and recall, our proposed features obtained results that are only not better than the two best features (LOC and EsC_{Mc}), but are better than all other eleven features. *This gives evidence that our proposed features can indeed be useful to predict bugs in procedural software systems*. For recall, all features presented similar results, varying only in 0.01. The difference among precision is larger, ranging from 0.31 to 0.41. PRE and DUP obtained the best results among our proposed features. We observed

that mainly CPL, but also DUP and SAL, have high standard deviation. This may be explained by project-specific practices. For example, if compiler warnings are taken into account by developers of a certain project, this feature will likely be not very helpful to identify fault-prone files in that project.

Finally, we observed that features associated with the Halstead metric suite are the least helpful. All of them obtained values lower than or equal to 0.10 for all measurements, except OpC_H, DOp_H, BUG_H, and VOL_H for precision. The last three, although achieved higher precision, it is still low (0.20–0.27). OpC_H has a high precision (0.39) in comparison with other features, but also high standard deviation (0.33). Surprisingly, this feature obtained 1.00 of precision with System A, but this same measurement is 0.06 for System B.

These presented results and observations indicate that our proposed code quality features can be potentially useful in bug prediction in our investigated context, mainly when compared to metrics of the Halstead suite. Nevertheless, as discussed, individual features are insufficient to obtain good results if used alone. We thus next investigate all possible feature subsets, focusing on those that obtained best results.

5.4.2 Best Feature Subsets (RQ2)

As explained in our study procedure, the number of all possible feature subsets considering 17 features is extremely large. Therefore, we selected representative subsets to be analysed, presented in Table 5.4, split into three groups. The first is composed of complete feature sets: (i) the set with all baseline features (*baseline*); (ii) the set with all proposed code quality features (*quality features*); and (iii) the set that includes both (*full set*). The second refers to the best subsets selected according to the four criteria adopted in social choice, introduced in Section 5.2. The third presents the subsets that have the best value for a particular project, referred to as *P Best*, where *P* is the project name. Selected feature subsets vary according to the measurement used in the selection. For each feature, we present not only the average precision, recall, and f-measure obtained across the different projects, but also the features present in the respective subset.

Considering the subset with our proposed features, it is possible to observe that it achieved superior results in comparison with our baseline, considering precision (0.50 *vs.* 0.39) and f-measure (0.27 *vs.* 0.24). Recall is the same for both subsets. However, we highlight that the variance in recall across target projects using our code quality features

is smaller. This can be seen in Figure 5.1, which shows the box blot of obtained results for each measurement, showing the variance across different projects. Subsets presented in this figure are those selected based on f-measure only, due to space restrictions; but we remind the reader that the first three subsets are independent of the measurement used for selection.

Given that the *quality features* set achieved better results, a naive approach would be to believe that a set combining the baseline and quality features would achieve better results. However, the results in Table 5.4 show that the improvement is minimal. Obtained precision is the same (0.50), while recall is slightly better (0.19 *vs.* 0.20). In fact, none of these initially investigated sets achieved results better than those obtained with LOC only, as discussed in the previous section. Therefore, this emphasises the need for feature selection.

Large improvements can be seen in the selected best subsets (considering the social choice criteria), with respect to the subsets discussed above and LOC. For all best subsets, which are twelve in total as there are four selected subsets for each of the three measurements, results are improved not only considering the measurement that was used for the subset selection, but also the other measurements. The only exception is the subset using the least misery criterion considering recall, which has a lower precision (0.46) than the full set (0.50). However, obtained results can be perceived in each of the measurements—gains are at least 0.12 of precision, 0.09 of recall, and 0.08 of f-measure, with respect to the full set. Therefore, depending on the goal of using a bug predictor, a subset that would better satisfy the goal should be adopted.

With respect to the lower result aforementioned associated with the least misery criterion, we clarify that this is expected because social choice criteria, other than average, do not aim at maximising average results. Maximising average results have the disadvantage that one (very) high value may compensate a low value. Consequently, there may be projects with poor results. Borda count and Copeland rule address this by not taking into account absolute values, but the relative position among subsets, considering a ranking based on a certain criterion. Least misery, in turn, prevents projects with (very) low values, as can be seen in Figure 5.1.

Now we analyse the features present in each of the best subsets. First, we observed that the number of features used to improve precision is higher than the number to improve recall. The best subsets have from 11 to 15 features. All features are present in at least two subsets, except TIM_H that is present only in the least misery subset. Recall, in turn,

has on average 7 features, not taking into account the Copeland rule subset that, different from the other subsets, has a high number of feature (13). The prioritisation of f-measure leads also to small feature subsets, being Copeland rule also an exception. This means that more information is needed to identify true positives, but the cost is to increase the number of false negatives as well.

Given the good results that LOC obtains individually, it is expected that it is always present in best subsets. This actually occurred, as can be seen in Table 5.4. The second best individual feature, EsC_{Mc}, is also present in all best subsets considering precision. Although it has results similar to those obtained with the baseline and full sets for both precision and recall, interestingly it is present in none of the best subsets considering recall, and only one considering f-measure. OpC_H is also present in the best subsets according to precision—its ranking position was relatively high considering this measurement. Although it achieved low recall when considered individually, it is present in two best subsets considering recall.

Regarding our code quality features, they are largely present in subsets selected using precision (only DUP is not present in the average subset), and at least two of them are present in each of the subsets considering recall and f-measure, except the least misery subset considering recall that included only DUP. This gives evidence that regardless of the criteria to select a subset, code quality features are present in the selected subset, indicating their value to our investigated problem.

Finally, we focus on the project-specific best subsets, shown in the last part of Table 5.4. They, on average, present lower results than best subsets selected according to the social choice criteria, because they have a larger variance in the results as shown in Figure 5.1. However, due to a high value obtained with at least one of the projects (often outliers), the average is higher than the full set in many cases.

With respect to the feature presence in these subsets, we observed that they, in general, include a small number of features, with some exceptions (mainly Busybox), being this number as low as one feature in three cases for precision. Moreover, there is little intersection among these subsets. This further indicates that project-specific characteristics highly impact on the adequate set of features to bug prediction. This indicates that it is possibly better to tailor bug predictors to individual projects than aiming to search for the best off-the-shelf bug predictor. However, it is fundamental to identify a set of features that can potentially contribute to the identification of fault-prone software elements, in order to serve as a starting point for making customisations in a predictor. Moreover, it

is important to specify bug predictors that have the best performance in general, because they can be used when there is no information available to make such customisations.

5.5 Threats to validity

We now report threats to the validity of our study, and describe how they were mitigated. A construction threat is the method used to extract defects to build our dataset, which (i) may be not completely accurate, mainly when commit messages from VCSs are used; and (ii) extracts only known defects, rather than those that actually exist. The latter problem is a general known problem in bug prediction (HERZIG; JUST; ZELLER, 2013) and, given that all existing defects are unknown in real software, existing work in this context limit themselves to do the best considering the information available. Regarding the former problem, we used whenever possible issue trackers, which are less dependent on the consistent use of message patterns. Nevertheless, when it was not available, our defect extraction method is in accordance with those adopted in previous work (ZHANG et al., 2014; KIM; WHITEHEAD JR.; ZHANG, 2008; BIRD et al., 2009). Another problem is that the applicability of the approaches is compromised without the ability to predict the critically of the predicted bug. Nevertheless, despite the presence of the information of the critically in some of the examined projects, the approaches examined in Chapter 3 did not use this information either. Hence, for a fair comparison, we used only the binary presence of bugs in this work.

We also identified four threats to the external validity. The first is that we used only one single classification algorithm, namely random forests. Consequently, results are not generalisable to other algorithms. However, we emphasise that the exploration of alternative algorithms is not in the scope of this study, and we selected the algorithm that performed best with procedural systems in the study performed on Chapter 3. The second external threat is the limited number of target projects. In order to mitigate this threat, we selected systems varying in size, domain and development environment (open source *vs.* proprietary). The third external threat is that different executions of k-fold cross validation can present different results when random folds are used. We used 3-fold cross validation (as opposed to a higher number, such as 10-fold), due to computational restrictions. To prevent random extremely positive or negative results, we checked for outliers in the three executions. We identified no outlier in the results. The forth and last threat derives from the number of projects, which is not large. The number of projects is

sufficient to support our conclusions. However, it is not a number large enough to perform statistical tests and obtain results that are valid (that is, a statistical test can be performed, but results would not be meaningful). This was also the case of many of previous similar studies. Finally, we highlight that our results are valid in the context of our investigated scenario, i.e. procedural systems written in the C language. Consequently, further studies should be conducted to verify whether our results hold in other contexts.

5.6 Final Remarks

In this chapter, we presented a study which evaluated the software quality attributes proposed in Chapter 4. We found that these attributes, together with long-standing software metrics, e.g. lines of code, enhance the bug predictor performance in the procedural software systems. Next, we present a summary of our contributions and issues that shall be addressed in future work.

Table 5.4: Analysis of Best Feature Subsets.

(a) Selection based on Precision

Feature Subset	DOr_H	OpC_H	DOp_H	LEN_H	VOC_H	VOL_H	DIF_H	EFF_H	TIM_H	BUG_H	LOC	CyC_{Mc}	EsC_{Mc}	CPL	SAL	DUP	PRE	**Precision**	**Recall**	**F-Measure**
Baseline	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					0.39	0.19	0.24
Full Set	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.50	0.20	0.28
Quality Features														✓	✓	✓	✓	0.50	0.19	0.27
Average		✓	✓		✓		✓	✓		✓	✓		✓	✓	✓		✓	0.71	0.20	0.29
Borda Count	✓	✓	✓			✓					✓	✓	✓	✓	✓	✓	✓	0.69	0.22	0.33
Copeland Rule		✓		✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	0.64	0.20	0.29
Least Misery	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	0.62	0.22	0.32
System A Best		✓																0.39	0.07	0.09
System B Best				✓									✓					0.55	0.19	0.24
Busybox Best	✓	✓	✓		✓			✓	✓		✓	✓			✓	✓	✓	0.57	0.24	0.30
LwIP Best												✓						0.44	0.11	0.13
Linux Best														✓				0.36	0.16	0.19

(b) Selection based on Recall

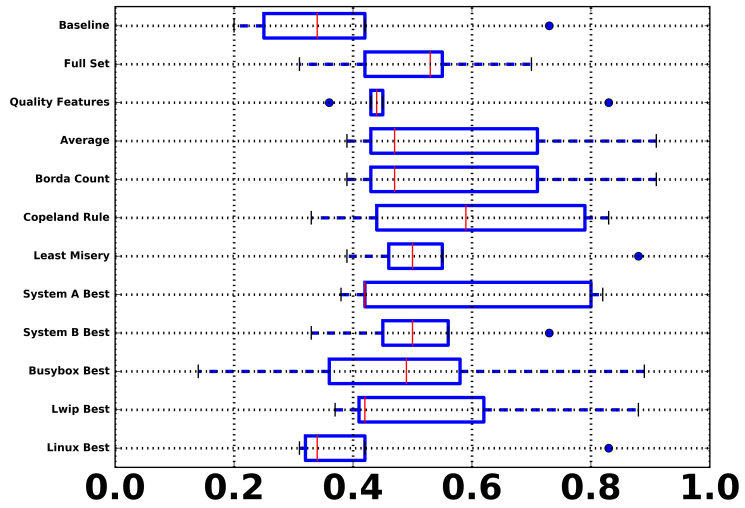
Feature Subset	DOr_H	OpC_H	DOp_H	LEN_H	VOC_H	VOL_H	DIF_H	EFF_H	TIM_H	BUG_H	LOC	CyC_{Mc}	EsC_{Mc}	CPL	SAL	DUP	PRE	**Precision**	**Recall**	**F-Measure**
Baseline	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					0.39	0.19	0.24
Full Set	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.50	0.20	0.28
Quality Features														✓	✓	✓	✓	0.50	0.19	0.27
Average	✓		✓				✓			✓	✓			✓			✓	0.58	0.35	0.40
Borda Count		✓				✓			✓		✓				✓	✓		0.58	0.32	0.40
Copeland Rule	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓			✓	✓	✓		0.56	0.29	0.37
Least Misery			✓		✓	✓			✓	✓	✓	✓				✓		0.46	0.30	0.36
System A Best	✓	✓	✓	✓	✓		✓				✓			✓		✓	✓	0.53	0.26	0.34
System B Best			✓								✓			✓				0.51	0.28	0.34
Busybox Best			✓				✓	✓		✓	✓		✓		✓			0.49	0.26	0.33
LwIP Best											✓		✓	✓				0.39	0.29	0.28
Linux Best									✓				✓	✓				0.44	0.21	0.24

(c) Selection based on F-Measure

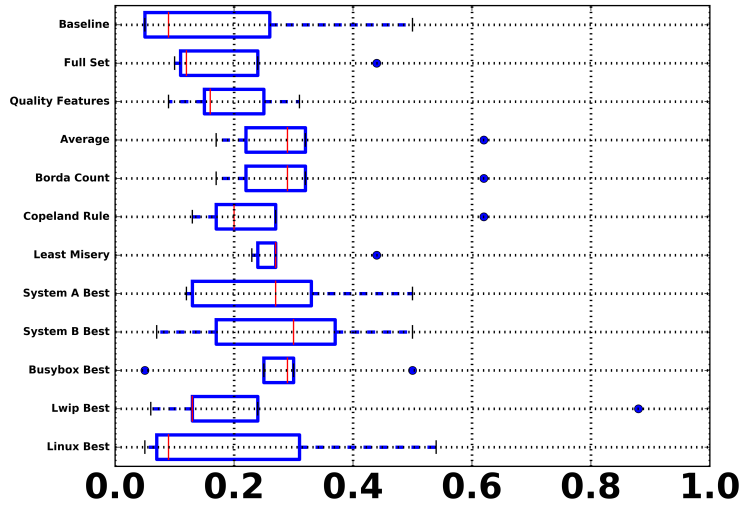
Feature Subset	DOr_H	OpC_H	DOp_H	LEN_H	VOC_H	VOL_H	DIF_H	EFF_H	TIM_H	BUG_H	LOC	CyC_{Mc}	EsC_{Mc}	CPL	SAL	DUP	PRE	**Precision**	**Recall**	**F-Measure**
Baseline	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					0.39	0.19	0.24
Full Set	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	0.50	0.20	0.28
Quality Features														✓	✓	✓	✓	0.50	0.19	0.27
Average		✓				✓			✓		✓				✓	✓		0.58	0.32	0.40
Borda Count		✓				✓			✓		✓				✓	✓		0.58	0.32	0.40
Copeland Rule		✓	✓	✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓	0.60	0.28	0.36
Least Misery			✓								✓			✓	✓			0.56	0.29	0.38
System A Best			✓	✓	✓		✓	✓			✓						✓	0.57	0.27	0.36
System B Best			✓								✓			✓				0.51	0.28	0.34
Busybox Best	✓	✓									✓		✓	✓		✓		0.49	0.28	0.35
LwIP Best	✓	✓			✓		✓				✓	✓	✓	✓	✓	✓	✓	0.54	0.29	0.34
Linux Best									✓				✓	✓				0.44	0.21	0.24

Figure 5.1: Measurement variance analysis across target projects: best feature subsets selected based on f-measure.

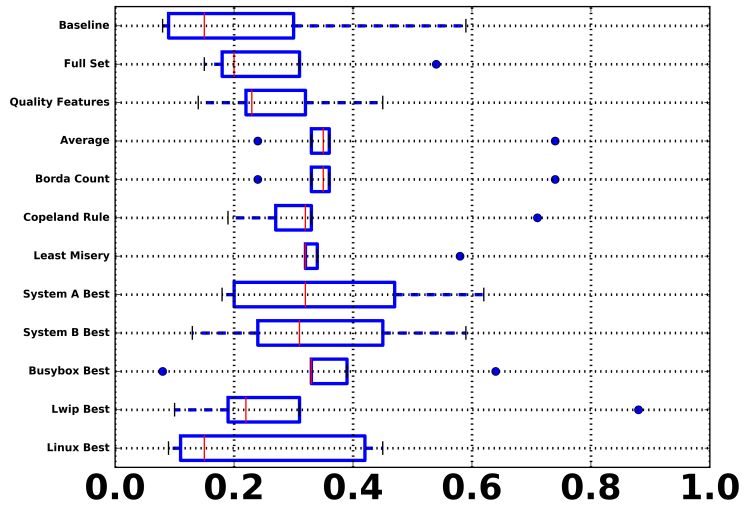
(a) Precision



(b) Recall



(c) F-Measure



6 CONCLUSION

Bug prediction is an important means of improving the verification of software systems. Its use within development processes is still a work in progress, and further improvement is necessary for the practical use of bug predictors. Static source code metrics play an important role in bug prediction because it has two important characteristics which other types of features—e.g. source code change metrics—do not have: (i) it does not depend on source code history and (ii) it provide means for the developer to remove an entity from the bug-prone entities list by refactoring the code and thus fixing static source code metrics. Hence, improvement of techniques using static source code metrics is one of the objectives that should be sought to improve bug prediction.

In this work, we evaluated bug prediction approaches in the context of procedural software systems. We found that the adaption of OO static source code metrics does not improve the prediction performance. Given the best existing approach we found, we extended it with quality features based on software engineering good practices. With these new features and already used ones, we evaluated it and concluded that our quality features performed, on average, as better as the approach found. We also found that feature selection remains as an important step towards a better bug prediction in the procedural software systems context.

6.1 Contributions

Given the results presented in this dissertation, we list below our main contributions.

Comparison of bug prediction in the context of procedural software systems. In Chapter 3 we presented a comparison of bug prediction approaches in the context of procedural software systems. Based on our findings we determined which source code static metrics perform best together with the associated machine learning algorithm. We also concluded that adapting OO static source code metrics does not improve bug prediction models for the evaluated context. A difference was also found between proprietary systems and open source software as described in Chapter 3.

Use code quality features as bug prediction features. We also proposed to use source code quality features to enhance bug prediction models for procedural software

systems. The reasoning and details of these features are detailed in Chapter 4. A study, described in Chapter 5, was performed and we concluded that the proposed features (i) perform better for prediction than the baseline approach when used alone, and (ii) enhanced the bug prediction performance for procedural software systems when used combined.

Importance of feature selection for procedural software systems bug prediction. Based on the study presented in Chapter 5, we found improvement of more than twenty percent prediction performance when tailoring the features set for a system. Nevertheless, this set may perform much worse for other systems. Hence for a specific system with a established dataset, performing a feature selection will enhance the prediction results for procedural software system, corroborating the literature that found this same results for the OO context (SHIVAJI et al., 2013; GAO et al., 2011; CATAL; DIRI, 2009).

6.2 Future Work

This work contribution improves the applicability of bug prediction in the evaluated context. Nevertheless, our work has limitations that can be explored in future work. We discuss them below.

Reproduction with a higher number of systems. Our work is limited due to the number of systems used. For a higher confidence a much higher number of software systems should be evaluated. Such evaluation has many issues to be addressed first like (i) the correct finding of defective files and(ii), the finding of meaningful software systems, i.e. with a development cycle long enough to be studied.

Explore other source code quality features. Static code analysis used in this study were turn-key tools. The use of more comprehensive tools (e.g. Clang static analyser[1]) is a possible path to improving the performance of bug prediction. An alternative not explored in our code quality features in Chapter 5 is the Splint static analyser. [2] It was not used because it required build system integration, i.e. makefile tweaking. Moreover, other methods such as graph measurements of the abstract syntax tree provided by compilers or amount of unreachable code were also not taken into

[1] <https://clang-analyzer.llvm.org/>
[2] <http://splint.org/>

account due to the same reason. We believe that the use of this other information will improve more bug prediction performance in the procedural software systems context as the extra features we used improved the model we evaluated.

Integration of bug prediction in the software life-cycle. Another remaining question we did not address in our work is *How to integrate bug prediction to the existing development software development process*. To the best of our knowledge, Lewis et al. (LEWIS et al., 2013) is the single report of work in this context. Defining in which step of the development process and how the information should be listed is specific to each process. We believe that some contexts are worth studying: (i) listing the most bug-prone entities before testing, (ii) listing the entities for peer-review, and (iii) determining the stability of a version, providing a threshold for the release of it.

In summary, we investigated the use of bug prediction in the context of procedural software systems. We evaluated existing approaches, finding the best for this context. Based on it, we proposed new features and combined with the best approach found to enhance the performance in the procedural software systems context. Now, our objective is to implement the use of bug prediction within a development process, determining the best step to do it, easing the adoption of such promising technique by developers.

REFERENCES

ABDI, H.; WILLIAMS, L. J. **Principal component analysis**. [s.n.], 2010. 433–470 p. Disponível em: <http://doi.org/10.1002/wics.101>.

ABREU, F. B.; CARAPUçA, R. Object-Oriented Software Engineering : Measuring and Controlling the Development Process. **4th. International Conference of Software Quality**, v. 4, n. October, p. 3–5, 1994.

ALPAYDIN, E. **Introduction to machine learning (OIP).** [S.l.: s.n.], 2004.

ALPAYDIN, E. **Introduction to Machine Learning**. 2nd. ed. [S.l.]: The MIT Press, 2010.

BANSIYA, J.; DAVIS, C. G. A hierarchical model for object-oriented design quality assessment. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 28, n. 1, p. 4–17, jan. 2002.

BASILI, V. R.; BRIAND, L. C.; MELO, W. L. A validation of object-oriented design metrics as quality indicators. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 22, n. 10, p. 751–761, Oct 1996. Disponível em: <http://doi.org/10.1109/32.544352>.

BASILI, V. R.; SELBY, R. W.; HUTCHENS, D. H. Experimentation in software engineering. **IEEE Transactions on Software Engineering**, SE-12, n. 7, p. 733–743, July 1986. Disponível em: <http://doi.org/10.1109/TSE.1986.6312975>.

BEN-ARI, M. The bug that destroyed a rocket. **ACM SIGCSE Bulletin**, ACM, New York, NY, USA, v. 33, n. 2, p. 58–59, jun. 2001. Disponível em: <http://doi.org/10.1145/571922.571958>.

BINKLEY, A. B.; SCHACH, S. R. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: **Proceedings of the 20th International Conference on Software Engineering**. [S.l.: s.n.], 1998. p. 452–455.

BIRD, C. et al. Fair and balanced?: Bias in bug-fix datasets. In: **Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering**. New York, NY, USA: ACM, 2009. (ESEC/FSE '09), p. 121–130. Disponível em: <http://doi.acm.org/10.1145/1595696.1595716>.

BLACK, A. **Critical Testing Process: Plan, Prepare, Perform, Perfect**. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2003.

BREIMAN, L. Random forests. **Machine Learning**, v. 45, n. 1, p. 5–32, 2001. Disponível em: <http://doi.org/10.1023/A:1010933404324>.

CATAL, C.; DIRI, B. Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. **Information Sciences**, v. 179, n. 8, p. 1040 – 1058, 2009. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020025508005173>.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 20, n. 6, p. 476–493, Jun 1994. Disponível em: <http://doi.org/10.1109/32.295895>.

CORTES, C.; VAPNIK, V. Support-vector networks. **Machine Learning**, v. 20, n. 3, p. 273–297, 1995.

D'AMBROS, M.; LANZA, M.; ROBBES, R. An extensive comparison of bug prediction approaches. In: **2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)**. [s.n.], 2010. p. 31–41. Disponível em: <http://doi.org/10.1109/MSR.2010.5463279>.

DRAPER, N. R.; SMITH, H. **Applied Regression Analysis**. [s.n.], 2014. 704 p. Disponível em: <https://books.google.ca/books?id=uSReBAAAQBAJ>.

FERENC, R. et al. Columbus - reverse engineering tool and schema for C++. **International Conference on Software Maintenance, 2002. Proceedings.**, 2002.

FORGY, E. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. **Biometrics**, v. 21, n. 1, p. 768, 1965.

FOWLER, M.; BECK, K. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 1999. (Component software series). Disponível em: <https://books.google.com.br/books?id=1MsETFPD3I0C>.

GAO, K. et al. Choosing software metrics for defect prediction: An investigation on feature selection techniques. **Softw. Pract. Exper.**, John Wiley & Sons, Inc., New York, NY, USA, v. 41, n. 5, p. 579–606, abr. 2011. Disponível em: <http://dx.doi.org/10.1002/spe.1043>.

GYIMOTHY, T.; FERENC, R.; SIKET, I. Empirical validation of object-oriented metrics on open source software for fault prediction. **IEEE Transactions on Software Engineering**, v. 31, n. 10, p. 897–910, Oct 2005. Disponível em: <http://doi.org/10.1109/TSE.2005.112>.

HALL, M. et al. The weka data mining software: An update. **SIGKDD Explorations**, v. 12, 2009.

HALL, T. et al. A systematic literature review on fault prediction performance in software engineering. **IEEE Transactions on Software Engineering**, v. 38, n. 6, p. 1276–1304, Nov 2012.

HALSTEAD, M. H. **Elements of Software Science (Operating and Programming Systems Series)**. New York, NY, USA: Elsevier Science Inc., 1977.

HASSAN, A. E.; HOLT, R. C. The top ten list: dynamic fault prediction. In: **21st IEEE International Conference on Software Maintenance (ICSM'05)**. [s.n.], 2005. p. 263–272. Disponível em: <http://doi.org/10.1109/ICSM.2005.91>.

HENDERSON-SELLERS, B. **Object-oriented Metrics: Measures of Complexity**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

HERZIG, K.; JUST, S.; ZELLER, A. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: **Proceedings of the 2013 International Conference on Software Engineering**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 392–401. Disponível em: <http://doi.org/10.1109/ICSE.2013.6606585>.

JAECHANG, N. **Survey on Software Defect Prediction**. 2017. Disponível em: <http://lifove.github.io/files/PQE_Survey_JC.pdf>.

JAY, G. et al. Cyclomatic Complexity and Lines of Code : Empirical Evidence of a Stable Linear Relationship. **Journal of Software Engineering and Applications**, v. 2, n. 3, p. 137–143, 2009. Disponível em: <http://doi.org/10.4236/jsea.2009.23020>.

JIANG, T.; TAN, L.; KIM, S. Personalized defect prediction. In: **2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [s.n.], 2013. p. 279–289. Disponível em: <http://doi.org/10.1109/ASE.2013.6693087>.

JUERGENS, E. et al. Do code clones matter? In: **Proceedings of the 31st International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. (ICSE '09), p. 485–495. Disponível em: <http://doi.org/10.1109/ICSE.2009.5070547>.

JURECZKO, M.; DIOMIDIS, S. Using Object-Oriented Design Metrics to Predict Software Defects. **Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej**, p. 69–81, 2010.

JURECZKO, M.; MADEYSKI, L. Towards identifying software project clusters with regard to defect prediction. In: **Proceedings of the 6th International Conference on Predictive Models in Software Engineering**. New York, NY, USA: ACM, 2010. (PROMISE '10), p. 9:1–9:10. Disponível em: <https://doi.org/10.1145/1868328.1868342>.

KAMEI, Y. et al. An empirical study of fault prediction with code clone metrics. In: **2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement**. [S.l.: s.n.], 2011. p. 55–61.

KIM, S.; WHITEHEAD JR., E. J.; ZHANG, Y. Classifying software changes: Clean or buggy? **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, USA, v. 34, n. 2, p. 181–196, mar. 2008. Disponível em: <http://doi.org/10.1109/TSE.2007.70773>.

KIM, S. et al. Predicting faults from cached history. In: **Proceedings of the 29th International Conference on Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. (ICSE '07), p. 489–498. Disponível em: <http://doi.org/10.1109/ICSE.2007.66>.

KO, A. J.; MYERS, B. a. A framework and methodology for studying the causes of software errors in programming systems. **Journal of Visual Languages and Computing**, v. 16, n. 1-2 SPEC. ISS., p. 41–84, 2005.

KOHONEN, T. Self-organized formation of topologically correct feature maps. **Biological Cybernetics**, Springer-Verlag, v. 43, n. 1, p. 59–69, 1982.

KORU, A. G.; LIU, H. Building effective defect-prediction models in practice. **IEEE Software**, v. 22, n. 6, p. 23–29, Nov 2005. Disponível em: <http://doi.org/10.1109/MS.2005.149>.

LEWIS, C. et al. Does bug prediction support human developers? findings from a google case study. In: **2013 35th International Conference on Software Engineering (ICSE)**. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 372–381. Disponível em: <http://doi.org/10.1109/ICSE.2013.6606583>.

MARTIN, R. Oo design quality metrics: An analysis of dependencies. In: **OOPSLA'94**. [S.l.: s.n.], 1994.

MASTHOFF, J. Group recommender systems: Combining individual models. In: ____. **Recommender Systems Handbook**. Boston, MA: Springer US, 2011. p. 677–702. Disponível em: <http://doi.org/10.1007/978-0-387-85820-3_21>.

MCCABE, T. J. A complexity measure. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 308–320, Dec 1976. Disponível em: <http://doi.org/10.1109/TSE.1976.233837>.

MEDEIROS, F. et al. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In: BOYLAND, J. T. (Ed.). **29th European Conference on Object-Oriented Programming (ECOOP 2015)**. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. (Leibniz International Proceedings in Informatics (LIPIcs), v. 37), p. 495–518. Disponível em: <http://doi.org/10.4230/LIPIcs.ECOOP.2015.495>.

MONDEN, A. et al. Software quality analysis by code clones in industrial legacy software. In: **Proceedings Eighth IEEE Symposium on Software Metrics**. [S.l.: s.n.], 2002. p. 87–94.

MOSER, R.; PEDRYCZ, W.; SUCCI, G. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: **Proceedings of the 30th International Conference on Software Engineering**. New York, NY, USA: ACM, 2008. (ICSE '08), p. 181–190. Disponível em: <http://doi.org/10.1145/1368088.1368114>.

MOSER, R.; RUSSO, B.; SUCCI, G. Empirical analysis on the correlation between gcc compiler warnings and revision numbers of source files in five industrial software projects. **Empirical Software Engineering**, v. 12, n. 3, p. 295–310, 2007. Disponível em: <http://doi.org/10.1007/s10664-006-9029-x>.

MUNSON, J. C.; ELBAUM, S. G. Code churn: A measure for estimating the impact of code change. In: **Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)**. Washington, DC, USA: IEEE Computer Society, 1998. (ICSM '98), p. 24–31. Disponível em: <http://doi.org/10.1109/ICSM.1998.738486>.

NAGAPPAN, N.; BALL, T. Use of relative code churn measures to predict system defect density. In: **Proceedings of the 27th International Conference on Software Engineering**. New York, NY, USA: ACM, 2005. (ICSE '05), p. 284–292. Disponível em: <http://doi.org/10.1145/1062455.1062514>.

NAGAPPAN, N.; BALL, T.; ZELLER, A. Mining metrics to predict component failures. In: **Proceedings of the 28th International Conference on Software Engineering**. New York, NY, USA: ACM, 2006. (ICSE '06), p. 452–461. Disponível em: <http://doi.org/10.1145/1134285.1134349>.

NIE, K.; ZHANG, L. On the relationship between preprocessor-based software variability and software defects. In: **High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on**. [s.n.], 2011. p. 178–179. Disponível em: <http://doi.org/10.1109/HASE.2011.44>.

OLAGUE, H. M. et al. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. **IEEE Transactions on Software Engineering**, v. 33, n. 6, p. 402–419, June 2007. Disponível em: <http://doi.org/10.1109/TSE.2007.1015>.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011.

RADJENOVIć, D. et al. Software fault prediction metrics. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, v. 55, n. 8, p. 1397–1418, ago. 2013. Disponível em: <http://dx.doi.org/10.1016/j.infsof.2013.02.009>.

RAHMAN, F. et al. Comparing static bug finders and statistical prediction. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: ACM, 2014. (ICSE 2014), p. 424–434. Disponível em: <http://doi.org/10.1145/2568225.2568269>.

SHIRABAD, J. S.; MENZIES, T. **The PROMISE Repository of Software Engineering Databases.** 2005. School of Information Technology and Engineering, University of Ottawa, Canada. Disponível em: <http://promise.site.uottawa.ca/SERepository>.

SHIVAJI, S. et al. Reducing features to improve code change-based bug prediction. **IEEE Transactions on Software Engineering**, v. 39, n. 4, p. 552–569, April 2013.

SOMMERVILLE, I. **Software Engineering**. [S.l.: s.n.], 2010. 56–81 p.

SPENCER, H.; COLLYER, G. `#ifdef` considered harmful, or portability experience with c news. In: **USENIX Annual Technical Conference**. [S.l.: s.n.], 1992.

TANG, M.-H.; KAO, M.-H.; CHEN, M.-H. An empirical study on object-oriented metrics. In: **Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)**. [S.l.: s.n.], 1999. p. 242–249.

TASHTOUSH, Y.; AL-MAOLEGI, M.; ARKOK, B. The Correlation among Software Complexity Metrics with Case Study. **International Journal of Advanced Computer Research**, v. 4, n. 2, p. 414–419, 2014.

TIAN, Y.; LAWALL, J.; LO, D. Identifying linux bug fixing patches. In: **2012 34th International Conference on Software Engineering (ICSE)**. Piscataway, NJ, USA: IEEE Press, 2012. p. 386–396. Disponível em: <http://doi.org/10.1109/ICSE.2012.6227176>.

TIOBE. **TIOBE Index for July 2015**. 2015. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

TOOLWORKS, I. S. **Understand**. 2017. "<https://scitools.com/features/>. [Online: accessed 29-April-2017].

WITTEN, I. H.; FRANK, E. **Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

YULE, G. U. An introduction to the theory of statistics /. 1968.

ZHANG, F. et al. Towards building a universal defect prediction model. In: **Proceedings of the 11th Working Conference on Mining Software Repositories**. New York, NY, USA: ACM, 2014. (MSR 2014), p. 182–191. Disponível em: <http://doi.acm.org/10.1145/2597073.2597078>.

ZIMMERMANN, T.; NAGAPPAN, N.; ZELLER, A. Predicting bugs from history. In: ____. **Software Evolution**. [S.l.]: Springer, 2008. cap. 4, p. 69–88.

ZIMMERMANN, T.; PREMRAJ, R.; ZELLER, A. Predicting defects for eclipse. In: **Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on**. [s.n.], 2007. p. 9–15. Disponível em: <http://doi.org/10.1109/PROMISE.2007.10>.

APPENDIX A — RESUMO EXTENDIDO

No processo de desenvolvimento de software, uma parte significativa do esforço e do tempo é dedicada à identificação de defeitos — ou *bugs*. Esse processo normalmente consiste em integrar módulos menores em porções mais sofisticadas. A descoberta de um defeito na etapa de integração e teste de um módulo de alto nível possui, portanto, um custo mais alto pois todas integrações anteriores devem ser validadas. Para endereçar esses problema várias técnicas são aplicadas: testes, análise estática, verificação formal e semi-formal, padrões de projeto e padrões de código (SOMMERVILLE, 2010).

Um método (semi)automático para priorizar recursos de todas ferramentas descritas acima são os preditores de *bugs* (HASSAN; HOLT, 2005) (ZIMMERMANN; NAGAPPAN; ZELLER, 2008) (KIM et al., 2007).Estes consistem em ferramentas que usam informações coletadas de controle de versões, controle de tarefa e métricas de código fonte para construir um modelo preditivo de quais seriam os módulos mais propícios a apresentar erros no sistema. Essas abordagens se dividem naquelas que: (i) utilizam métricas do código fonte, (ii) utilizam métricas de mudança do código fonte e (iii) utilizam métricas do processo. As abordagens (i) e (ii) podem ser usadas para prever módulos a nível de arquivo, enquanto a (iii) geralmente é mais útil como indicador de estabilidade do software como um todo. No caso de abordagens de mudança de código fonte, a linguagem é indiferente, ao passo que abordagens que usam métricas de código fonte dependem da linguagem e do paradigma utilizado.

Um nicho cujas abordagens existentes usando métricas de código para predição de *bugs* não abordam com profundidade é o de sistemas procedurais. Esses sistemas, normalmente escritos em C, ainda são responsáveis por uma porcentagem considerável do desenvolvimento de software (TIOBE, 2015), sendo geralmente utilizados na implementação de sistemas críticos: sistemas operacionais, bibliotecas chaves, computação de alto desempenho, etc. Esses sistemas não são contemplados pela maioria das abordagens pois as mesmas usam métricas aplicáveis a sistemas orientados a objetos, reduzindo o número de métricas úteis para predição nesse contexto. Com base nisso nós chegamos à seguinte questão de pesquisa:

Como prever bugs (semi)-automaticamente em sistemas procedurais?

No nosso primeiro estudo comparamos cinco abordagens existentes (GYIMOTHY; FERENC; SIKET, 2005; JURECZKO; MADEYSKI, 2010; KIM; WHITEHEAD JR.; ZHANG, 2008; MOSER; PEDRYCZ; SUCCI, 2008; KORU; LIU, 2005) no contexto

de sistemas procedurais, usando nove sistemas de diversos domínios. Nós encontramos que Koru e Liu (KORU; LIU, 2005) obteve a melhor performance. Baseados nisso, propusemos quatro novos atributos para serem utilizados para predição de *bugs* nesse contexto: código duplicado, avisos de compilador, analisadores estáticos e contagens de pré-processador. Em um segundo estudo realizamos uma seleção de atributos completa em cinco sistemas, usando os atributos propostos e os utilizados por Koru e Liu, assim como o mesmo algoritmo de aprendizado de máquina usado por eles: *Random Forests*. Nesse estudo encontramos que linhas de código e a complexidade de McCabe obteve uma melhor performance sobre todos os sistemas, seguido pelos atributos propostos. Também encontramos que a seleção para cada sistema supera o modelo com a melhor média para qualquer uma das métricas utilizadas: precisão, *recall* e *f-measure*. Abaixo iremos repassar as abordagens existentes de bugs, descrever em detalhes o primeiro estudo, os atributos propostos, o segundo estudo, nossas principais contribuições e trabalho futuro.

Como já introduzido acima, preditores de *bugs* podem ser divididos em várias categorias. Aqui apresentamos as seguintes divisões: métricas de código fonte, métricas de mudança de código fonte, comparações a abordagens híbridas, e por fim abordagens cujo objetivo é estudar a migração de modelos entre sistemas. Além dessa categorização outras muito mais complexas já foram realizadas (HALL et al., 2012), levando em consideração quais as métricas utilizadas, qual o algoritmo de aprendizado de máquina, etc. A seguir mostramos apenas as divisões pertinentes ao escopo desse trabalho.

Métricas Estáticas de Código Fonte Métricas de código fonte medem, de diversas formas, a complexidade do código fonte. Entre essas métricas estão: McCabe (MCCABE, 1976), Halstead (HALSTEAD, 1977), CK (CHIDAMBER; KEMERER, 1994), Abreu e Carapuça (ABREU; CARAPUçA, 1994) e Bansiya e Davis (BANSIYA; DAVIS, 2002). Basili et al. (BASILI; BRIAND; MELO, 1996) e Binkley e Schach (BINKLEY; SCHACH, 1998) são precursores do uso de métricas de código fonte para a predição de *bugs*, o primeiro realizou o primeiro estudo registrado, enquanto que o segundo comparou esse modelo à métricas de tempo de execução. Olague et al. (OLAGUE et al., 2007), Koru e Liu (KO; MYERS, 2005), Gyimothy et al. (GYIMOTHY; FERENC; SIKET, 2005), Zimmermann et al. (ZIMMERMANN; PREMRAJ; ZELLER, 2007) utilizaram métricas de código fonte para predição de *bugs* em sistemas mais realistas que os anteriores, obtendo bons resultados. Outra métrica usada é o de clone de código. Kamei et al. (KAMEI et al., 2011) e Monden et al. (MONDEN et al., 2002) encontraram pouca corre-

lação com predição de defeitos, porém encontraram correlações com confiabilidade de código.

Métricas de Mudança de Código Fonte Outra metodologia é extrair métricas de mudança, baseado na premissa de que o defeito será introduzido na porção do sistema que foi alterado. Munson e Elbaum (MUNSON; ELBAUM, 1998) usaram a variação do código fonte para previsão de um sistema embarcado. Nagappan e Ball (NAGAPPAN; BALL, 2005) estenderam o conceito à variação e à remoção, aplicando a abordagem nos sistemas da Microsoft®. Hassan e Holt (HASSAN; HOLT, 2005) usaram seis sistemas de código aberto para sua avaliação, a proposta era proporcionar uma lista dos dez módulos mais propensos a defeitos, usando uma abordagem baseada em *caches*. Kim et al. (KIM et al., 2007)a usou o conceito de localidade temporal e especial para predição, além de usar o conceito de mudança conjunta.

Comparações e Abordagens Híbridas de Métricas Estáticas e de Métricas Mudança Outras abordagens comparam e/ou combinam tanto métricas estáticas quanto métricas de mudança, e em alguns casos usam métricas de processo. Zimmermann et al. (ZIMMERMANN; NAGAPPAN; ZELLER, 2008) analisa a evolução dos defeitos em contraste com o processo de desenvolvimento, comparando sua própria implementação com Basili et al. (BASILI; BRIAND; MELO, 1996) e Nagappan e Ball (NAGAPPAN; BALL, 2005) usando, como esses componentes da Microsoft®. Moser et al. (MOSER; PEDRYCZ; SUCCI, 2008) comparou métricas estáticas e de mudança no projeto Eclipse. Outros executaram uma seleção fina de atributos usando heurísticas (SHIVAJI et al., 2013; GAO et al., 2011).

Uso de Modelos em Múltiplos Sistemas Um outro ponto endereçado por outros trabalhos é a migração de modelos entre diferentes sistemas. Jureczko e Madeyski (JURECZKO; MADEYSKI, 2010) e Zhang et al. (ZHANG et al., 2014) agruparam diversos projetos e construíram preditores específicos para cada sistema e também genéricos. Encontraram que apesar preditores específicos possuírem uma melhor performance, preditores genéricos possuem uma performance adequada, sendo a melhor alternativa para projetos cujo histórico de defeitos não permita a construção de um preditor próprio. Nesse contexto Lewis et al. (LEWIS et al., 2013) incorporou um preditor de *bugs* à todos projetos do Google®, encontrando que para um melhor uso por parte dos usuários são necessárias ações que possam ser disparadas

pelo usuário para resolver a indicação de *bugs*.

Como não existem avaliações das abordagens existentes sobre predição de *bugs* no contexto de sistemas procedurais, nós efetuamos um estudo aplicando essas abordagens à sistemas procedurais, adaptando métricas de código fonte quando necessário. Abaixo listamos o *GQM* conforme proposto por Basili et al. (BASILI; SELBY; HUTCHENS, 1986). *Verificar a eficácia de abordagens de predição de bugs no contexto de sistemas de software procedurais, avaliando abordagens existentes baseadas em métricas de código estáticas da perspectiva do pesquisador no contexto de oito sistemas de código aberto e projetos de código proprietário.*

Os sistemas utilizados para essa avaliação são: (i) Linux, sistema operacional altamente difundido com mais de 30.000 linhas de código; (ii) Sistemas Comerciais A,B e C, todos no contexto de equipamentos de telecomunicação; (iii) BusyBox, sistema com 624 arquivos para geração de utilitários de sistema para sistemas embarcados; (iv) LwIP, protocolos de rede para microcontroladores sem sistema operacional; (v) CpuMiner aplicação com 20 arquivos para computação de *BitCoin*. Todos os sistemas escolhidos são escritos em C.

No estudo, as cinco abordagens existentes foram comparadas, a nível de aplicabilidade e performance em sistemas procedurais (GYIMOTHY; FERENC; SIKET, 2005; JURECZKO; DIOMIDIS, 2010; KIM; WHITEHEAD JR.; ZHANG, 2008; KORU; LIU, 2005; MOSER; PEDRYCZ; SUCCI, 2008). Métricas de código fonte cujo foco são sistemas orientados a objetos foram adaptadas usando-se a premissa que um arquivo fonte e seu respectivo cabeçalho são uma classe, métricas de herança não foram adaptadas. Com base nessa adaptação a adaptabilidade das abordagens foi extraída. Com base nas métricas extraídas, foi executada uma predição para cada sistema usando o(s) algoritmos de aprendizado de máquina propostos por cada abordagem. Com base nos resultados da predição usando validação cruzada em 10 estratos, foram medidas as performances, tanto para cada sistema quanto para cada abordagem. Como resultado, foi encontrado que a abordagem de Koru e Liu (KORU; LIU, 2005) possui o mais alto grau de adaptabilidade (100%). Além disso, essa mesma abordagem apresentou, em média, a melhor performance quando usando o algoritmo *Random Forests*, com *F-Measure* de 0.59 e desvio padrão de 0.23. A abordagem de Gyimothy et al. (GYIMOTHY; FERENC; SIKET, 2005) sem métricas adaptadas continha apenas linhas de código, e mesmo assim obteve uma performance razoável, mostrando que uma parcela significativa do poder preditivo dessas abordagens está presente nesse atributo. Baseado nos resultados obtidos, os três pontos a

seguir são apresentados.

Uso de métricas orientadas a objetos adaptadas O uso de métricas adaptadas de sistemas orientados a objetos para sistemas procedurais não mostrou nenhuma melhor de performance. Tal diferença pode ser pelo fato de que o paradigma de desenvolvimento usado difere, ou pela informação contida nas métricas de herança.

Contraste entre sistemas abertos e proprietários Sistemas de código aberto possuíram, em média, uma melhor performance do que sistemas proprietários. Isso pode ser devido ao desbalanceamento da classe alvo ou pela diferença no desenvolvimento dos sistemas procedurais cujos objetivos em uma determinada versão geralmente são apontados por um número menor de *stakeholders*.

Comparação da eficácia em sistemas orientados a objetos e procedurais As abordagens obtiveram para sistemas procedurais desempenho inferior ao obtido para sistemas orientados a objetos. Isso pode ser consequência do fato descrito acima, da restrição das métricas, ou uma característica do sistemas em si.

Com base nas restrições encontradas e no desempenho inferior dos preditores para sistemas procedurais, nós propomos o uso de alguns atributos baseados em boas práticas de engenharia de software. O raciocínio utilizado se baseia em duas premissas: (i) onde há defeitos simples, provavelmente haverá defeitos complexos; (ii) código com pouca legibilidade ou informações ocultas induz o desenvolvedor a cometer erros. Abaixo são listados os quatro atributos utilizados.

Código Duplicado Como existe evidência de que código duplicado é correlacionado aos defeitos (JUERGENS et al., 2009; KAMEI et al., 2011; MONDEN et al., 2002), e intuitivamente o código duplicado pode induzir a um concerto parcial, o mesmo foi usado. Para implementação foram usadas duas granularidades, código duplicado dentro do arquivo fonte e entre arquivos fontes.

***Warnings* de Compilador** *Warnings* de compilador são locais no código onde uma falha latente está presente. Em alguns casos a falha não se manifesta; porém ao deixa-los o desenvolvedor pode estar apressado, desleixado ou ser inexperiente, em qualquer dos casos a presença de outros defeitos nesse arquivo é mais provável. Moser et al. (MOSER; RUSSO; SUCCI, 2007) também encontrou correlação de *Warnings* com defeitos. Nesse trabalho foi usado o GCC e seus três respetivos níveis de *Warnings*: normal, Wall e Wextra.

Erros de Analisadores Estáticos Analisadores estáticos procuram padrões de código com defeitos conhecidos, como alocação e a não desalocação de recursos. O raciocínio para sua inclusão é o mesmo dos *Warnings* de compilador: se o desenvolvedor se permite deixar erros de análise estática, outros defeitos mais graves podem estar presentes. Foram usados analisadores estáticos que não requerem integração com o processo de compilação e que são abertos. Os selecionados foram o CppCheck e o Uno.

Contagens de Pré-Processador O pré-processador é uma forma de adicionar código condicional em C. Ele pode gerar macros que atuam como funções, porém que são expandidas em código fonte diretamente antes da compilação, também pode ser usado para definir constantes, incluir outros arquivos e também habilitar/desabilitar partes do código para a compilação. Essas funcionalidades podem causar problemas por aumentar a variabilidade do código, introduzindo uma maior complexidade ao desenvolvimento. Para cada arquivo foram contadas o número dessas diretivas e usadas como atributos.

Para avaliar os atributos propostos acima, um estudo foi executado combinando os atributos propostos com a abordagem de melhor performance: Koru e Liu (KORU; LIU, 2005). O propósito do estudo é avaliar a relevância dos atributos propostos no contexto de predição de *bugs* e encontrar o melhor conjunto de atributos para predição de *bugs* em sistemas de software procedurais. Nesse segundo estudo, cinco sistemas foram avaliados, sendo estes um subconjunto dos sistemas de primeiro estudo: Linux, Sistemas Comerciais A e B, BusyBox e LwIP. Somente uma versão foi avaliada por sistema, pois a avaliação de múltiplas versões gera uma complexidade computacional que torna o estudo impraticável. No estudo foram seguidos os seguintes passos: (i) preparação do conjunto de dados usando-se as métricas de Koru e Liu (KO; MYERS, 2005), combinadas com os atributos propostos; (ii) execução de uma seleção de atributos usando-se todas as combinações de atributos(*brute force feature selection*);(iii) análise dos resultados utilizando-se métodos de escolha(média, Copeland, menor miséria e contagem de borda(*Borda Count*). Foram comparados os melhores conjuntos para cada métrica de aprendizado de máquina: *f-measure*, precisão e *recall*. Também foram comparados a execução de cada atributo separadamente. Foi encontrado que, para cada sistema individualmente um conjunto pequeno de atributo alcança uma alta performance; porém para melhores conjuntos para todos os sistemas continham mais métricas. Foi encontrado que os artefatos de qualidades propostos como atributos aumentaram a precisão sem interferir nas demais métricas. Eles tam-

bém estiveram no topo da performance média considerando-se apenas um atributo para o modelo. As métricas de complexidade de McCabe, juntamente com linhas de código apresentaram uma performance melhor ainda, entretanto os melhores conjuntos para precisão contêm tanto um artefato de qualidade como uma das métricas de McCabe, no caso de *recall* e *f-measure* todos melhores conjuntos para todos sistemas contêm pelo menos um artefato de qualidade, e apenas 3 contêm métricas de McCabe. Linhas de código estive presente em todos os conjuntos para todos os sistemas e em parte considerável dos conjuntos para sistemas individuais. Abaixo listamos as principais conclusões derivadas dos resultados obtidos.

Linhas de código e complexidade de McCabe são os atributos mais gerais para predição de *bugs* Como métricas de complexidade de McCabe e linhas de código estiveram em praticamente todos os melhores conjuntos, apresentaram boa performance como único atributos, sugerimos que os mesmos sejam sempre considerados como atributos para a construção de preditor de *bugs* no contexto de sistemas procedurais.

Artefatos de qualidade são bons atributos de predição de *bugs* Os artefatos de qualidade apresentaram, apesar de não em todos os sistemas, uma boa capacidade preditiva, sendo assim, também devem ser considerados ao aplicar-se predição de *bugs* para sistemas procedurais.

Cada sistema possui um subconjunto de atributos ótimo Outra conclusão é a diferença de atributos no conjunto ótimo de cada sistema, mostrando que a seleção de atributos para cada sistema, nesse contexto, pode aumentar a performance consideravelmente.

Seguem abaixo as principais contribuições realizadas nesse trabalho.

Comparação de abordagens de predição de *bugs* no contexto de sistemas procedurais Foram comparados, no contexto de sistemas procedurais de software, cinco abordagens, mostrando que a adaptação de métricas orientadas à objetos não melhora o desempenho dos mesmos, também foi encontrado que a melhor abordagem utiliza métricas de Halstead, McCabe e linhas de código, juntamente com o algoritmo *Random Forests*.

Uso de artefatos de qualidade como atributos de predição de bugs Após determinar a melhor abordagem, foi proposto o uso de artefatos de qualidade de software como

atributos adicionais na predição de *bugs* no contexto de sistemas procedurais. Constatou-se que os atributos propostos aumentam a precisão dos modelos de predição sem prejudicar o *recall* e a *f-measure*. Foi encontrado também que a seleção de atributos é importante para um modelo com melhor desempenho também nesse contexto.

A partir desse trabalho duas questões podem ser estudadas. O uso de analisadores estáticos cuja integração com o processo de compilação é necessário, descobrindo mais defeitos (latentes ou não) e portanto aumentado a informação presente nos modelos. E a integração de modelos desse tipo no processo de desenvolvimento, seja utilizando-os no ordenamento de testes ou revisões, seja recomendando alterações ou refatorações para o desenvolvedor, seja atribuindo um grau de estabilidade para a versão a ser liberada.

APPENDIX B — COMPLETE RESULTS

In this appendix, we present the complete results obtained in the study described in the Chapter 5. We present the obtained results for the three machine learning metrics evaluated: precision in Table B.1, recall in Table B.2 and f-measure in Table B.3. The results for each best set are presented for sets selection by the three metrics also: precision in Table B.4, recall in Table B.5 and F-measure in Table B.6.We present the same results presented in Figure 5.1, but for precision in Figure B.1 and for recall in Figure B.2.

Table B.1: Position Ordered By Precision

System A			System B			Busybox			LWIP			Linux			Mean		
Order	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**
DOp_H	640	1.00	EsC_{Mc}	424	0.80	EsC_{Mc}	39337	0.38	CyC_{Mc}	8060	1.00	CPL	0	0.66	EsC_{Mc}	57754	0.50
OpC_H	683	1.00	PRE	11553	0.60	LOC	73368	0.32	EsC_{Mc}	34447	0.87	CyC_{Mc}	42040	0.40	PRE	72704	0.41
LOC	6616	0.75	DUP	34868	0.52	PRE	98958	0.28	CPL	49445	0.83	PRE	54705	0.39	LOC	72787	0.50
PRE	67324	0.45	LOC	50181	0.48	SAL	114971	0.23	VOL_H	90581	0.75	EsC_{Mc}	84869	0.36	CyC_{Mc}	80218	0.44
DUP	99229	0.33	CyC_{Mc}	113739	0.33	CyC_{Mc}	118452	0.22	DUP	90584	0.75	LOC	111218	0.33	CPL	87060	0.35
CyC_{Mc}	118800	0.25	SAL	124337	0.25	OpC_H	121288	0.21	SAL	101635	0.71	DUP	120261	0.30	DUP	94438	0.41
VOC_H	120804	0.22	TIM_H	129175	0.08	VOC_H	126881	0.17	BUG_H	114162	0.66	BUG_H	128412	0.20	OpC_H	102227	0.39
SAL	121049	0.22	OpC_H	129298	0.06	DOp_H	127093	0.17	LOC	122552	0.62	VOL_H	128490	0.20	DOp_H	103855	0.27
CPL	128414	0.13	LEN_H	129340	0.05	DUP	127249	0.16	OpC_H	129913	0.50	VOC_H	128762	0.20	SAL	118611	0.31
EsC_{Mc}	129696	0.10	VOL_H	129465	0.00	CPL	127825	0.16	PRE	130982	0.33	TIM_H	129349	0.20	VOL_H	122035	0.20
EFF_H	130748	0.00	DIF_H	129498	0.00	DIF_H	128940	0.14	DOp_H	131040	0.00	LEN_H	129546	0.19	BUG_H	126437	0.20
TIM_H	130751	0.00	DOp_H	129562	0.00	DOr_H	128941	0.14	LEN_H	131043	0.00	OpC_H	129953	0.19	VOC_H	127412	0.12
BUG_H	130755	0.00	VOC_H	129573	0.00	BUG_H	129105	0.14	VOC_H	131044	0.00	DIF_H	130728	0.18	DIF_H	130196	0.06
VOL_H	130760	0.00	CPL	129616	0.00	EFF_H	129872	0.12	TIM_H	131045	0.00	EFF_H	130781	0.18	LEN_H	130217	0.07
DIF_H	130768	0.00	EFF_H	129710	0.00	LEN_H	130385	0.10	DIF_H	131048	0.00	DOp_H	130943	0.17	TIM_H	130247	0.06
LEN_H	130774	0.00	BUG_H	129753	0.00	VOL_H	130880	0.05	EFF_H	131049	0.00	SAL	131066	0.15	EFF_H	130432	0.06
DOr_H	131070	0.00	DOr_H	131070	0.00	TIM_H	130919	0.05	DOr_H	131070	0.00	DOr_H	131069	0.14	DOr_H	130644	0.05

Table B.2: Position Ordered By Recall

System A			System B			Busybox			LWIP			Linux			Mean		
Order	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**
LOC	7595	0.21	LOC	3887	0.25	SAL	7080	0.16	CPL	5714	0.62	EsC_{Mc}	142	0.41	LOC	26686	0.26
DUP	44751	0.14	PRE	17230	0.20	CPL	10342	0.15	EsC_{Mc}	83437	0.43	CyC_{Mc}	182	0.41	DUP	71151	0.17
PRE	91336	0.09	DUP	49601	0.16	LOC	10349	0.15	SAL	111049	0.31	LOC	470	0.38	EsC_{Mc}	75113	0.20
VOC_H	91341	0.09	EsC_{Mc}	116825	0.05	DOr_H	25460	0.12	LOC	111133	0.31	DUP	2493	0.35	CPL	79690	0.16
CyC_{Mc}	121849	0.03	SAL	123270	0.04	DIF_H	28280	0.12	PRE	124826	0.25	TIM_H	11382	0.31	SAL	83753	0.16
EsC_{Mc}	121853	0.03	TIM_H	126784	0.02	EsC_{Mc}	53312	0.10	DUP	128923	0.18	DIF_H	12399	0.30	DIF_H	86390	0.08
SAL	121859	0.03	OpC_H	128961	0.01	VOC_H	53448	0.10	VOL_H	128957	0.18	BUG_H	22565	0.29	PRE	86913	0.16
CPL	121863	0.03	CyC_{Mc}	129181	0.01	DOp_H	53469	0.10	BUG_H	130599	0.12	DOp_H	22820	0.29	DOp_H	93134	0.08
DOp_H	128844	0.01	LEN_H	129344	0.01	OpC_H	97098	0.06	CyC_{Mc}	130978	0.06	SAL	55508	0.26	CyC_{Mc}	102434	0.10
OpC_H	129684	0.01	CPL	129462	0.00	PRE	106195	0.05	OpC_H	131036	0.06	VOL_H	92811	0.24	VOC_H	104372	0.08
DIF_H	130729	0.00	EFF_H	129465	0.00	BUG_H	123380	0.03	TIM_H	131040	0.00	PRE	94979	0.23	TIM_H	106119	0.06
VOL_H	130730	0.00	BUG_H	129481	0.00	CyC_{Mc}	129984	0.01	EFF_H	131041	0.00	VOC_H	116544	0.22	BUG_H	107355	0.09
LEN_H	130734	0.00	VOC_H	129486	0.00	DUP	129987	0.01	DIF_H	131042	0.00	EFF_H	127110	0.20	DOr_H	109903	0.05
EFF_H	130747	0.00	VOL_H	129489	0.00	LEN_H	130632	0.00	VOC_H	131043	0.00	LEN_H	129635	0.18	VOL_H	122528	0.08
TIM_H	130748	0.00	DOp_H	129492	0.00	TIM_H	130644	0.00	LEN_H	131044	0.00	OpC_H	129824	0.18	OpC_H	123320	0.06
BUG_H	130752	0.00	DIF_H	129503	0.00	VOL_H	130654	0.00	DOp_H	131046	0.00	DOr_H	130847	0.14	EFF_H	129804	0.04
DOr_H	131070	0.00	DOr_H	131070	0.00	EFF_H	130657	0.00	DOr_H	131070	0.00	CPL	131070	0.00	LEN_H	130277	0.04

Table B.3: Position Ordered By F-measure

System A			System B			Busybox			LWIP			Linux			Mean		
Order	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**	**Order**	**Position**	**Value**
LOC	2153	0.33	LOC	3281	0.33	LOC	16525	0.20	CPL	3325	0.71	CyC$_{Mc}$3		0.40	LOC	28937	0.33
DUP	52795	0.20	PRE	10569	0.30	SAL	21053	0.19	EsC$_{Mc}$	56522	0.58	EsC$_{Mc}$39		0.39	EsC$_{Mc}$	69374	0.25
PRE	87385	0.15	DUP	39905	0.24	EsC$_{Mc}$	45905	0.15	SAL	115899	0.43	LOC	1639	0.35	DUP	76619	0.22
VOC$_H$	97514	0.12	EsC$_{Mc}$	116811	0.10	CPL	48875	0.15	LOC	121091	0.41	DUP	31180	0.32	PRE	83745	0.22
CyC$_{Mc}$	124698	0.06	SAL	124787	0.06	DOr$_H$	68302	0.13	DUP	129273	0.30	PRE	86427	0.29	CPL	88054	0.18
SAL	125688	0.06	TIM$_H$	128060	0.04	DIF$_H$	68303	0.13	VOL$_H$	129322	0.30	TIM$_H$	124613	0.24	CyC$_{Mc}$	102739	0.12
CPL	127080	0.05	CyC$_{Mc}$	128247	0.02	VOC$_H$	75349	0.12	PRE	130264	0.28	BUG$_H$	124633	0.24	SAL	103516	0.19
EsC$_{Mc}$	127595	0.05	OpC$_H$	129327	0.02	DOp$_H$	76132	0.12	BUG$_H$	130723	0.21	DIF$_H$	126330	0.23	VOC$_H$	112525	0.09
DOp$_H$	127850	0.03	LEN$_H$	129362	0.02	OpC$_H$	99441	0.10	CyC$_{Mc}$	130953	0.11	VOL$_H$	127617	0.22	DIF$_H$	117270	0.07
OpC$_H$	127960	0.03	DOp$_H$	129914	0.00	PRE	104080	0.09	OpC$_H$	131014	0.11	DOp$_H$	127758	0.22	DOr$_H$	118509	0.05
BUG$_H$	130746	0.00	VOC$_H$	129915	0.00	BUG$_H$	127219	0.05	EFF$_H$	131040	0.00	VOC$_H$	128808	0.21	DOp$_H$	118539	0.07
TIM$_H$	130749	0.00	VOL$_H$	129916	0.00	CyC$_{Mc}$	129794	0.03	DOp$_H$	131041	0.00	SAL	130154	0.19	OpC$_H$	123647	0.09
EFF$_H$	130751	0.00	DIF$_H$	129917	0.00	DUP	129942	0.03	LEN$_H$	131042	0.00	EFF$_H$	130367	0.19	BUG$_H$	128648	0.10
DIF$_H$	130756	0.00	EFF$_H$	129918	0.00	EFF$_H$	130665	0.01	VOC$_H$	131043	0.00	LEN$_H$	130388	0.19	TIM$_H$	129075	0.05
VOL$_H$	130760	0.00	BUG$_H$	129919	0.00	LEN$_H$	130686	0.01	DIF$_H$	131045	0.00	OpC$_H$	130493	0.18	VOL$_H$	129697	0.10
LEN$_H$	130762	0.00	CPL	129920	0.00	VOL$_H$	130871	0.01	TIM$_H$	131046	0.00	DOr$_H$	131036	0.14	LEN$_H$	130448	0.04
DOr$_H$	131070	0.00	DOr$_H$	131070	0.00	TIM$_H$	130908	0.01	DOr$_H$	131070	0.00	CPL	131070	0.00	EFF$_H$	130548	0.04

Table B.4: Performance for Selected Sets Using Precision.

	System A			System B			Busybox			LWIP			Linux			Mean		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Baseline	0.42	0.09	0.15	0.25	0.05	0.09	0.20	0.05	0.08	0.73	0.50	0.59	0.34	0.26	0.30	0.39	0.19	0.24
Full Set	0.55	0.11	0.18	0.53	0.12	0.20	0.31	0.10	0.15	0.70	0.44	0.54	0.42	0.24	0.31	0.50	0.20	0.28
Quality Features	0.36	0.09	0.14	0.44	0.15	0.22	0.45	0.16	0.23	0.83	0.31	0.45	0.43	0.25	0.32	0.50	0.19	0.27
Average	1.00	0.13	0.23	0.71	0.13	0.22	0.54	0.12	0.19	0.86	0.38	0.52	0.45	0.23	0.30	0.71	0.20	0.29
Borda Count	0.77	0.18	0.29	0.72	0.17	0.28	0.50	0.11	0.18	1.00	0.44	0.61	0.46	0.22	0.30	0.69	0.22	0.33
Copeland	0.75	0.11	0.19	0.67	0.16	0.26	0.44	0.10	0.16	0.86	0.38	0.52	0.46	0.25	0.33	0.64	0.20	0.29
Least Misery	0.71	0.18	0.29	0.68	0.17	0.28	0.50	0.09	0.15	0.70	0.44	0.54	0.49	0.24	0.32	0.62	0.22	0.32
System A V. 2 Best	1.00	0.02	0.04	0.06	0.01	0.02	0.21	0.07	0.10	0.50	0.06	0.11	0.20	0.18	0.19	0.39	0.07	0.09
System B V. 2 Best	0.08	0.02	0.03	1.00	0.05	0.10	0.50	0.12	0.20	0.86	0.38	0.52	0.33	0.40	0.36	0.55	0.19	0.24
Busybox 1.8.3 Best	0.35	0.15	0.21	0.48	0.15	0.22	0.88	0.06	0.11	0.71	0.62	0.67	0.43	0.23	0.30	0.57	0.24	0.30
Lwip 1.0.0 Best	0.25	0.04	0.06	0.33	0.01	0.03	0.22	0.02	0.03	1.00	0.06	0.12	0.41	0.41	0.41	0.44	0.11	0.13
Linux 3.0 Best	0.13	0.04	0.06	0.00	0.00	0.00	0.16	0.15	0.16	0.83	0.62	0.71	0.67	0.00	0.00	0.36	0.16	0.19
Baseline Improvement	0.58	0.09	0.14	0.75	0.12	0.19	0.68	0.11	0.15	0.27	0.12	0.12	0.33	0.15	0.11	0.32	0.05	0.09

P - Precision **R** - Recall **F** - F-Measure

Table B.5: Performance for Selected Sets Using Recall.

	System A			System B			Busybox			LWIP			Linux			Mean		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Baseline	0.42	0.09	0.15	0.25	0.05	0.09	0.20	0.05	0.08	0.73	0.50	0.59	0.34	0.26	0.30	0.39	0.19	0.24
Full Set	0.55	0.11	0.18	0.53	0.12	0.20	0.31	0.10	0.15	0.70	0.44	0.54	0.42	0.24	0.31	0.50	0.20	0.28
Quality Features	0.36	0.09	0.14	0.44	0.15	0.22	0.45	0.16	0.23	0.83	0.31	0.45	0.43	0.25	0.32	0.50	0.19	0.27
Average	0.67	0.18	0.29	0.56	0.19	0.28	0.49	0.21	0.29	0.78	0.88	0.82	0.41	0.27	0.32	0.58	0.35	0.40
Borda Count	0.71	0.22	0.33	0.47	0.29	0.36	0.43	0.17	0.24	0.91	0.62	0.74	0.39	0.32	0.35	0.58	0.32	0.40
Copeland	0.56	0.18	0.27	0.50	0.25	0.34	0.50	0.15	0.23	0.90	0.56	0.69	0.35	0.30	0.32	0.56	0.29	0.37
Least Misery	0.29	0.25	0.27	0.45	0.25	0.32	0.40	0.24	0.30	0.80	0.50	0.62	0.35	0.26	0.30	0.46	0.30	0.36
System A V. 2 Best	0.54	0.36	0.43	0.53	0.21	0.30	0.32	0.07	0.12	0.86	0.38	0.52	0.42	0.27	0.33	0.53	0.26	0.34
System B V. 2 Best	0.50	0.07	0.13	0.56	0.37	0.45	0.45	0.17	0.24	0.73	0.50	0.59	0.33	0.30	0.31	0.51	0.28	0.34
Busybox 1.8.3 Best	0.43	0.18	0.26	0.38	0.13	0.20	0.42	0.35	0.38	0.86	0.38	0.52	0.35	0.26	0.30	0.49	0.26	0.33
Lwip 1.0.0 Best	0.17	0.04	0.06	0.42	0.07	0.11	0.39	0.10	0.16	0.74	0.88	0.80	0.24	0.35	0.28	0.39	0.29	0.28
Linux 3.0 Best	0.42	0.09	0.15	0.31	0.05	0.09	0.32	0.07	0.11	0.83	0.31	0.45	0.34	0.54	0.42	0.44	0.21	0.24
Baseline Improvement	0.29	0.27	0.28	0.31	0.32	0.36	0.30	0.30	0.30	0.18	0.38	0.23	0.09	0.28	0.12	0.19	0.16	0.16

P - Precision **R** - Recall **F** - F-Measure

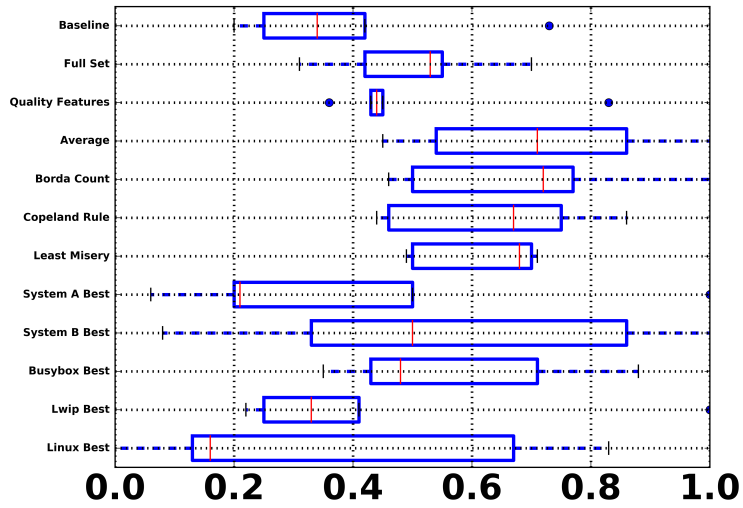
Table B.6: Performance for Selected Sets Using F-Measure.

	System A			**System B**			**Busybox**			**LWIP**			**Linux**			**Mean**		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
Baseline	0.42	0.09	0.15	0.25	0.05	0.09	0.20	0.05	0.08	0.73	0.50	0.59	0.34	0.26	0.30	0.39	0.19	0.24
Full Set	0.55	0.11	0.18	0.53	0.12	0.20	0.31	0.10	0.15	0.70	0.44	0.54	0.42	0.24	0.31	0.50	0.20	0.28
Quality Features	0.36	0.09	0.14	0.44	0.15	0.22	0.45	0.16	0.23	0.83	0.31	0.45	0.43	0.25	0.32	0.50	0.19	0.27
Average	0.71	0.22	0.33	0.47	0.29	0.36	0.43	0.17	0.24	0.91	0.62	0.74	0.39	0.32	0.35	0.58	0.32	0.40
Borda Count	0.71	0.22	0.33	0.47	0.29	0.36	0.43	0.17	0.24	0.91	0.62	0.74	0.39	0.32	0.35	0.58	0.32	0.40
Copeland	0.79	0.20	0.32	0.59	0.17	0.27	0.33	0.13	0.19	0.83	0.62	0.71	0.44	0.27	0.33	0.60	0.28	0.36
Least Misery	0.50	0.24	0.32	0.55	0.23	0.32	0.46	0.27	0.34	0.88	0.44	0.58	0.39	0.27	0.32	0.56	0.29	0.38
System A V. 2 Best	0.82	0.33	0.47	0.42	0.13	0.20	0.42	0.12	0.18	0.80	0.50	0.62	0.38	0.27	0.32	0.57	0.27	0.36
System B V. 2 Best	0.50	0.07	0.13	0.56	0.37	0.45	0.45	0.17	0.24	0.73	0.50	0.59	0.33	0.30	0.31	0.51	0.28	0.34
Busybox 1.8.3 Best	0.14	0.05	0.08	0.49	0.25	0.33	0.58	0.29	0.39	0.89	0.50	0.64	0.36	0.30	0.33	0.49	0.28	0.35
Lwip 1.0.0 Best	0.37	0.13	0.19	0.62	0.13	0.22	0.41	0.06	0.10	0.88	0.88	0.88	0.42	0.24	0.31	0.54	0.29	0.34
Linux 3.0 Best	0.42	0.09	0.15	0.31	0.05	0.09	0.32	0.07	0.11	0.83	0.31	0.45	0.34	0.54	0.42	0.44	0.21	0.24
Baseline Improvement	0.40	0.24	0.32	0.37	0.32	0.36	0.38	0.24	0.31	0.18	0.38	0.29	0.10	0.28	0.12	0.21	0.13	0.16

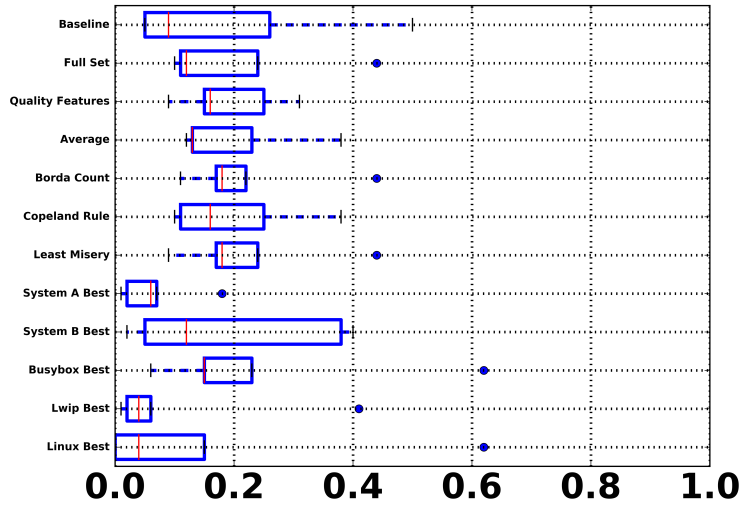
P - Precision **R** - Recall **F** - F-Measure

Figure B.1: Measurement variance analysis across target projects: best feature subsets selected based on precision.

(a) Precision



(b) Recall



(c) F-Measure

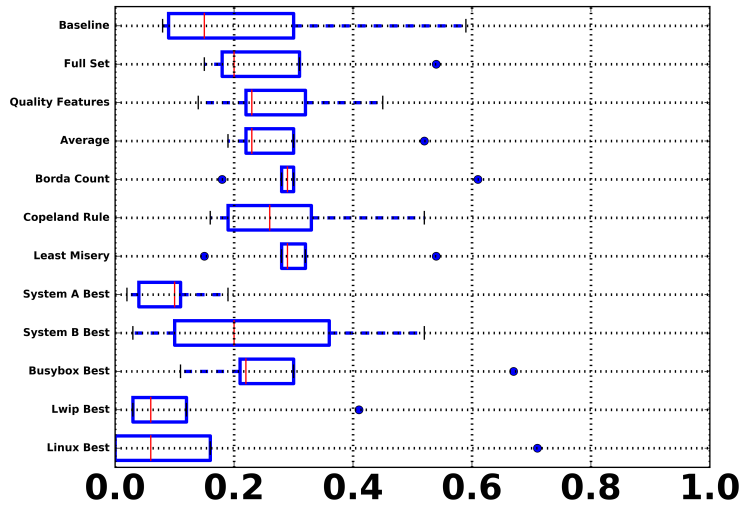
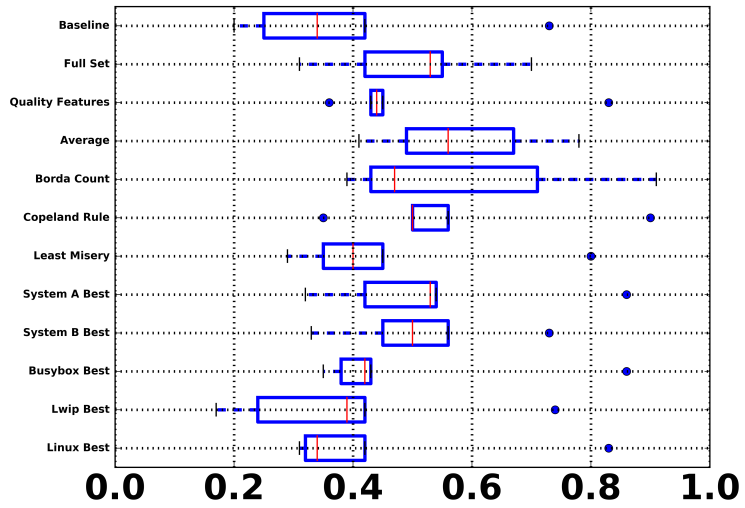
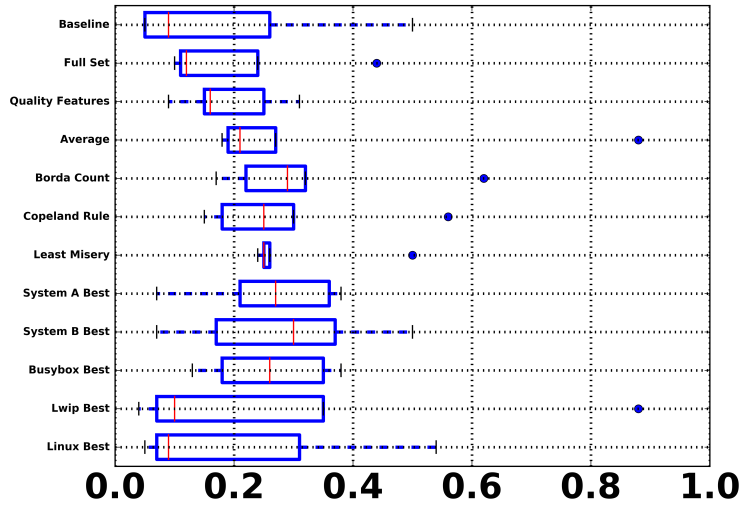


Figure B.2: Measurement variance analysis across target projects: best feature subsets selected based on recall.

(a) Precision



(b) Recall



(c) F-Measure