On the Effectiveness of Bug Predictors with Procedural Systems: A Quantitative Study

Cristiano Werner Araújo[1](✉), Ingrid Nunes[1,2], and Daltro Nunes[1]

[1] Instituto de Informática, Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil
{cwaraujo,ingridnunes,daltro}@inf.ufrgs.br
[2] TU Dortmund, Dortmund, Germany

Abstract. 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. We present a study in which we investigated the effectiveness of existing bug prediction approaches with procedural systems. Such approaches use as input static code metrics. We evaluated to what extent they are applicable to our context, and compared their effectiveness using standard metrics, with adaptations when needed. We assessed five approaches, using eight procedural software systems, including open-source and industrial projects. We concluded that lines of code is the metric that plays the key role in our context, and approaches that use of a large set of metrics can introduce noise in the prediction model. In addition, the best results were obtained with open-source systems.

Keywords: Bug prediction · Procedural programming · Static code metrics

1 Introduction

Software testing is a crucial task to improve software quality, as it identifies software defects (or bugs) to be fixed. This task can be complemented by automated approaches that identify fault prone software components, potentially decreasing verification time, and thus improving software maintenance and reducing costs. By learning which components are fault prone, it is possible to prioritize system modules or components to be verified, thus allowing defects to be identified earlier. Given these potential benefits, many *bug prediction* approaches [7,14,16,18,22,28,34] have been proposed, investigating the use of different types of information in order to improve precision and recall when predicting defects.

Examples of inputs used by bug prediction approaches are static code metrics, change metrics, and previous defects [6]. Such approaches were individually evaluated by their authors using different software projects, or compared using

DOI: 10.1007/978-3-662-54494-5_5

the same set of projects. These evaluations involved mostly, if not only, 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. A previous study, performed by Khoshgoftaar and Allen [15], focused specifically on embedded systems. Nevertheless, they used information collected at runtime, which requires many scenarios of system execution to obtain data.

Approaches that rely on change metrics, e.g. number of changes, which often come from version control systems (VCSs), can also be adopted in the context of procedural systems, because they are paradigm- and language-independent. However, this is not the case of approaches that use static code metrics. Although there are metrics that can be measured in procedural systems, such as lines of code or cyclomatic complexity [21], most of the approaches also use OO-specific metrics, such as depth of inheritance tree or coupling between object classes [5]. Consequently, such approaches must be adapted and evaluated to be used with procedural software systems. There is evidence that change metrics can outperform static code metrics, but: (i) the former may not always be available, because of the need for the existence and access to VCSs, and (ii) information provided by static code metrics and change metrics are complementary [22].

We thus in this paper present a study conducted to evaluate approaches that rely on static code metrics [28] in the context of procedural software systems. 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. As result, we concluded that lines of code is the metric that plays the key role in our context, and approaches that use of a large set of metrics can introduce noise in the prediction model. In addition, the best results were obtained with open-source systems.

We next discuss existing bug prediction approaches. Then, we present our study settings and target systems in Sect. 3. Results and discussions are detailed in Sects. 4 and 5, respectively. Finally, we conclude in Sect. 6.

2 Related Work

As said in the introduction, bug prediction approaches make predictions based on different kinds of information. We classify such approaches into three groups. The first group of approaches [4,7,14,18,25,26,34] is based on static code metrics. They use only a snapshot of the source code, using information such as number of lines, in order to extract static code metrics. Such approaches are founded by prior studies, which concluded that there is a correlation between static code metrics and defects [4]. The second group of approaches [9,10,13,17,19,23,24,31] requires another type of information, change metrics, which is usually obtained from VCSs. Such systems allow extraction of metrics associated with changes made during the software project evolution. Examples of typical metrics of this type are number of changes and change size. Some approaches use the frequency and recency of file change [10,24], using caching concepts to indicate the most fault prone components. In addition to change metrics, Li et al. [19] used information collected from e-mails to predict a release quality. Approaches [6,16,22,33] in the last group mixed different types of metrics, and some evaluated different proposed approaches.

Given that change metrics do not depend on the language or paradigm of the analyzed project, they can be used both for OO and procedural systems. In fact, some approaches included in their evaluation procedural systems [9] or used multiple versions of a single procedural system [19]. Nevertheless, approaches based on static code metrics typically rely on a metric set that includes OO-specific metrics. Koru and Liu [18], in particular, included two procedural systems in their evaluation, ignoring the OO-specific metrics for these systems. However, none of the approaches focused solely on procedural systems or contrasted results obtained with OO and procedural systems.

In Table 1, we detail a set of recent approaches that proposed bug predictors based on static code metrics—approaches that simply 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 paper, and hereafter they are referred to as the acronyms introduced in Table 1. We included in the study approaches that *also* use change metrics [16], but only static metrics were taken into account. Moser et al. [22]'s approach extended that proposed by Zimmermann et al. [34], 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.*"

3 Study Settings

Given the lack of provision of bug predictors dedicated to procedural software systems, we performed a study to fulfill this gap. We used existing bug prediction

Table 1. Summary of investigated approaches.

Approach	Acronym	Metric suites	Learning techniques
Gyimothy et al. [7]	GY	CK [5]	Decision Trees Linear Regression Neural Networks
Jureczko and Madeyski [14]	JU	CK [5], QMOOD [2], Tang et al. [29], Martin [20], Henderson-Sellers [11]	Linear Regression
Kim et al. [16]	KI	Metrics provided by the Understand [1] tool	SVM
Koru and Liu [18]	KO	Halstead [8], McCabe [21]	Decision Tree K-Star Random Forests
Moser et al. [22]	MO	CK [5], traditional OO metrics	Decision Trees Logistic Regression Naive Bayes

approaches with procedural software systems, also making required adaptations in this process, and evaluated and compared the obtained performance. We next provide details of our study.

3.1 Goal and Research Questions

In order to design our study, we followed the GQM (*goal-question-metric*) paradigm proposed by Basili et al. [3]. 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.

3.2 Procedure

Our study procedure is composed of three main steps. We first analyzed 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.

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 [14,16,32]. 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.

Extraction of static source code metrics was performed using the Understand [1] static code analysis tool. Metrics for pure C not available in the Understand, 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++.

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 [27] was used for prediction and score calculation. The following scores were used.

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

[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/.

All presented scores are in [0, 1], were the closer to one, the better the classification. Now that we have described our procedure, we proceed to the presentation of the target procedural systems of our study.

3.3 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 2, 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 domains and have multiple sizes, as can be seen in Table 2.

Table 2. Target systems.

System	Description	LOC	#Files	#Commits	Bugs (%)
Linux	Operating system	9,434,808–9,529,552	30,058–30, 252	560,519	16–22%
Commercial system A	Telecom embedded application	407,660–509,856	1027–1148	1,027–1148	4–10%
Commercial system B	Telecom embedded application	337,203–351,923	939–949	2,211	6–12%
Commercial system C	Telecom embedded application	279,325	394	109	5%
BusyBox	Operating system applications	153,448	624	13,891	19%
Git	Version control system	153,855–157,193	500–507	41,356	16–29%
Light weight IP	Network stack for microcontrollers	18,510–32579	89–132	3,658	14–49%
CpuMiner	Bitcoin mining application	4,455–6,927	20	339	20%

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 [13,31], 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 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 analyzed 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 analyzed 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 2. In Table 2, we also present the percentage of files containing bugs for each system. In next section, we present how the investigated bug prediction approaches performed using these introduced systems.

4 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 analyzed, and we assessed how applicable they are to our context. In Table 3, 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, 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. 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 4, which shows the applicability ratios (without and with adaptations) of each approach. Note that, although

Table 3. 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

Table 3. *(Continued)*

Suite	Metric	GY	JU	KI	KO	MO	Class	Adaptation
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.

MO approach, in theory, uses 31 static code metrics from Zimmermann et al.'s [34] 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

Table 4. 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

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 Fig. 1 and Table 5. On the left hand side charts of Fig. 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 5 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 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.

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.

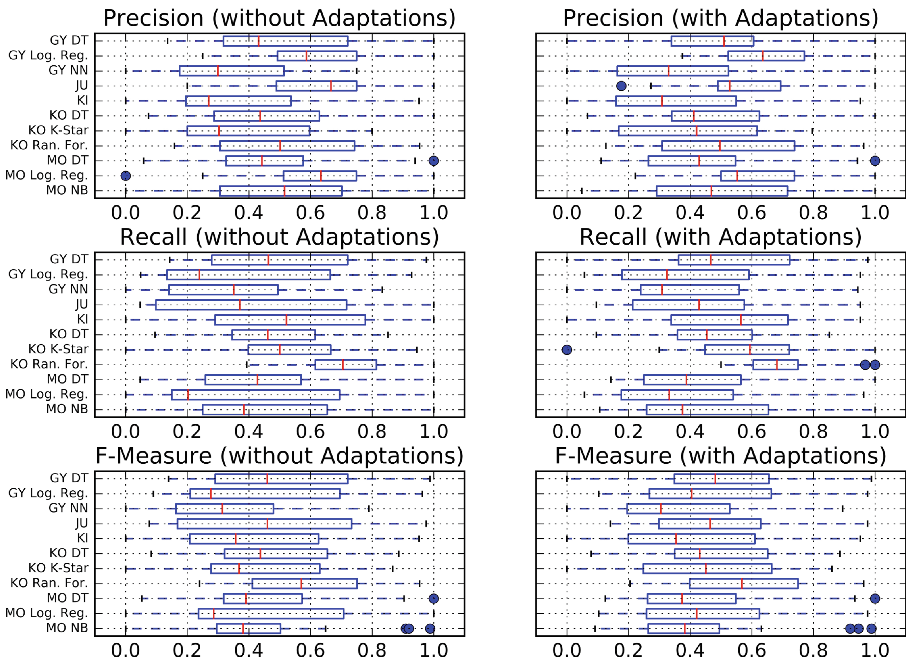


Fig. 1. Effectiveness measurements by each approach.

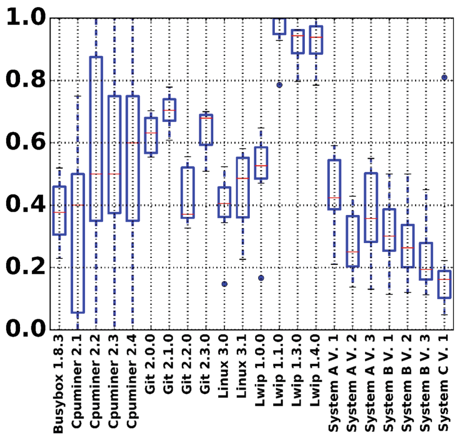
Table 5. Summary of effectiveness evaluation of each approach.

	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	

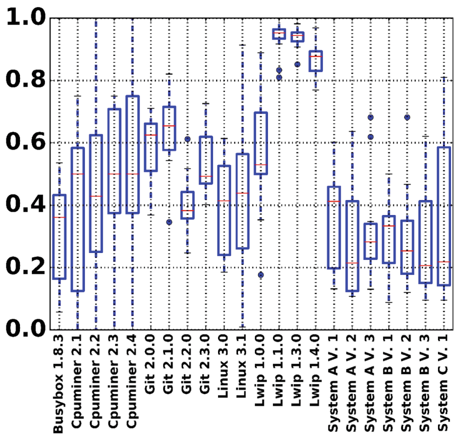
Legend: DT-Decision Trees; KS-K-Star; LR-Logistic Regression; NB-Naive Bayes; NN-Neural Networks; RF-Random Forest.

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.

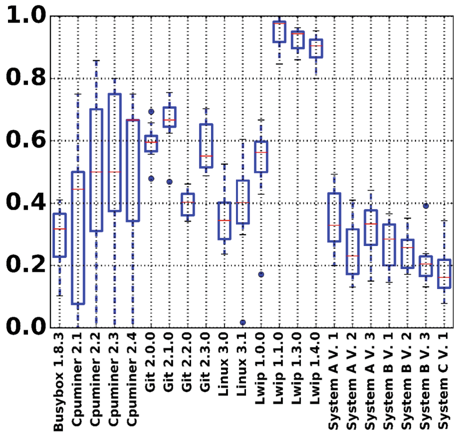
In addition to comparing results across different approaches, we also investigated how our measurements vary across different target systems, as presented in Fig. 2. We observed that commercial applications presented worse results in comparison with open source systems. This observation holds even for the Commercial System C, which has a low number of commits as described in Sect. 3.3. Analyzing results, we considered two hypotheses: (1) there are differences in the



(a) Precision



(b) Recall



(c) F-Measure

Fig. 2. Effectiveness measurements by target system.

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 possible to observe in Table 2 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 analyzed 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.

5 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 Fig. 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 5, 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 generalizable 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 [30]. 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 [12,25]. 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.

Threats to Validity. We performed an empirical evaluation of existing bug predictors, and we mitigated identified threats that could invalidate our results. An external threat is the number of projects used for evaluation. In order to address this, we selected systems from different domains, with difference sizes, and both open-source and proprietary.

A construction threat is the procedure adopted to extract defects. To mitigate this, we followed a procedure similar to that adopted by existing approaches, when issue trackers were not available. Based on the analysis of our systems, we observed that we would not be able to detect defects introduced and fixed during the development of a single release—we are not aware how or if prior work has addressed this issue, given that this was not reported. It would be inadequate to count them as bugs, because they were not present in the code from which metrics were extracted. Therefore, we added an additional step in the defect extraction, which verified if the fix commit changed code present in the code baseline. Another construction validity is that we implemented ourselves the investigated approaches. Although most approaches require only to execute learning techniques, parameters used in previous studies were not published. Consequently, we calibrated the models. For mitigating this threat, we replicated published studies using datasets that were made available by the authors, before performing our study.

6 Conclusion

In this paper, we presented a study in which we investigated how existing bug prediction approaches perform in the context of procedural software systems, using static code metrics. Although object orientation is currently the most used paradigm, procedural languages are still largely used for many fundamental applications, such as operating systems and scientific computing applications. The only investigated approach that relies solely on metrics that can be extracted from procedural systems is that proposed by Koru and Liu [18]. This approach presented one of the best results, followed by the approach proposed by Gyimothy et al. [7]. Note that such results were obtained with a subset of metrics used by these authors, given that some metrics rely on object-oriented concepts. In fact, the second best approach uses only one metric that can be extracted, namely lines of code. Therefore, we concluded that this metric plays a key role to build bug predictors in our context. We also adapted object-oriented metrics to be extracted from procedural systems. Our conclusion is that they do not improve the bug prediction for these systems.

Our results showed that bug predictors that have good performance with object-oriented systems do not necessarily are the best with procedural systems. Therefore, our future work includes the exploration of particularities of procedural systems and exploitation of metrics based on these particularities to build prediction models. Moreover, based on the analyzed systems, there is evidence that it is difficult to obtain good results with systems associated with datasets that have a low number of files with bugs. Therefore, it is important to explore techniques in the context of machine learning that deal with the issue of unbalanced classes.

Acknowledgments. The authors are very grateful to the Parks S/A (http://www.parks.com.br) Company that allowed analyzing three of their proprietary projects. Ingrid Nunes would like to thank for research grants CNPq ref. 303232/2015-3, CAPES ref. 7619-15-4, and Alexander von Humboldt, ref. BRA 1184533 HFSTCAPES-P.

References

1. Understand static code analysis tool. https://scitools.com/. Accessed 01 June 2016
2. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1), 4–17 (2002)
3. Basili, V.R., Selby, R.W., Hutchens, D.H.: Experimentation in software engineering. IEEE Trans. Softw. Eng. **12**(7), 733–743 (1986)
4. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)
5. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
6. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. MSR **2010**, 31–41 (2010)
7. Gyimothy, T., Ferenc, R., Siket, I.: Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans. Softw. Eng. **31**(10), 897–910 (2005)
8. Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York (1977)
9. Hassan, A.E.: Predicting faults using the complexity of code changes. In: ICSE 2009, pp. 78–88. IEEE Computer Society, USA (2009)
10. Hassan, A.E., Holt, R.C.: The top ten list: dynamic fault prediction. In: ICSM 2005, pp. 263–272. IEEE Computer Society, USA (2005)
11. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice-Hall Inc., Upper Saddle River (1996)
12. Jay, G., Hale, J.E., Smith, R.K., Hale, D., Kraft, N.A., Ward, C.: Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. J. Softw. Eng. Appl. **2**(3), 137–143 (2009)
13. Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 279–289, November 2013
14. Jureczko, M., Madeyski, L.: Towards identifying software project clusters with regard to defect prediction. In: PROMISE 2010, pp. 9:1–9:10. ACM, USA (2010)
15. Khoshgoftaar, T.M., Allen, E.B.: Predicting fault-prone software modules in embedded systems with classification trees. In: The 4th IEEE International Symposium on High-Assurance Systems Engineering (HASE 1999), p. 105. IEEE Computer Society, Washington, DC (1999)
16. Kim, S., Whitehead, E.J., Zhang, Y.: Classifying software changes: clean or buggy? IEEE Trans. Softw. Eng. **34**(2), 181–196 (2008)
17. Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A.: Predicting faults from cached history. In: ICSE 2008, pp. 15–16. ACM, USA (2008)
18. Koru, A.G., Liu, H.: Building effective defect prediction models in practice. IEEE Softw. **22**(6), 23–29 (2005)
19. Li, P.L., Herbsleb, J., Shaw, M.: Finding predictors of field defects for open source software systems in commonly available data sources: a case study of openBSD. In: METRICS 2005, pp. 10–32, September 2005

20. Martin, R.: OO design quality metrics: an analysis of dependencies. In: OOPSLA 1994 (1994)
21. McCabe, T.: A complexity measure. IEEE Trans. Softw. Eng. SE **2**(4), 308–320 (1976)
22. Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ICSE 2008, pp. 181–190. ACM, USA (2008)
23. Munson, J.C., Elbaum, S.G.: Code churn: a measure for estimating the impact of code change. In: ICSM 1998, pp. 24–31. IEEE Computer Society, USA (1998)
24. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: ICSE 2005, pp. 284–292. ACM, USA (2005)
25. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), pp. 452–461. ACM, New York (2006)
26. Olague, H.M., Etzkorn, L.H., Gholston, S., Quattlebaum, S.: 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 Trans. Softw. Eng. **33**(6), 402–419 (2007)
27. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: machine learning in Python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
28. Radjenović, D., Heričko, M., Torkar, R., Živkovič, A.: Software fault prediction metrics. Inf. Softw. Technol. **55**(8), 1397–1418 (2013)
29. Tang, M.H., Kao, M.H., Chen, M.H.: An empirical study on object-oriented metrics. In: METRICS 1999, p. 242. IEEE Computer Society, USA (1999)
30. Tashtoush, Y., Al-maolegi, M., Arkok, B.: The correlation among software complexity metrics with case study. Int. J. Adv. Comput. Res. **4**(2), 414–419 (2014)
31. Tian, Y., Lawall, J., Lo, D.: Identifying linux bug fixing patches. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 386–396, June 2012
32. Zhang, F., Mockus, A., Keivanloo, I., Zou, Y.: Towards building a universal defect prediction model. In: MSR 2014, pp. 182–191. ACM, USA (2014)
33. Zimmermann, T., Nagappan, N., Zeller, A.: Predicting bugs from history. In: Mens, T., Demeyer, S. (eds.) Software Evolution, pp. 69–88. Springer, Heidelberg (2008)
34. Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects for eclipse. In: PROMISE 2007, p. 9. IEEE Computer Society, USA (2007)