IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 7 May 2024, accepted 25 May 2024, date of publication 29 May 2024, date of current version 6 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3406929

RESEARCH ARTICLE

The Effectiveness of Hidden Dependence Metrics in Bug Prediction

JUDIT JÁSZ
Department of Software Engineering, University of Szeged, 6720 Szeged, Hungary
e-mail: jasy@inf.u-szeged.hu

This work was supported in part by European Union Project within the Framework of the Artificial Intelligence National Laboratory under Grant RRF-2.3.1-21-2022-00004; in part by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA Funding Scheme under Project TKP2021-NVA-09; and in part by the University of Szeged Open Access Fund under Grant 6842.

ABSTRACT Finding and fixing bugs in programs is perhaps one of the most difficult, yet most important, tasks in software maintenance. This is why in the last decades, a lot of work has been done on this topic, most of which is based on machine learning methods. Studies on bug prediction can be found for almost all programming languages. The solutions presented generally try to predict bugs based on information that can be easily extracted from the source code, rather than more expensive solutions that require a deeper understanding of the program. Another feature of these solutions is that they usually try to predict faults at a high level (module/file/class), which is useful, but locating the bug itself is still a difficult task. This work presents a solution that attempts to predict bugs at the method level, while also tracking the dependencies in the program using an efficient algorithm, resulting in an approach that can predict bugs more accurately. The practical measurements show that the defined approach really outperforms predictions based on traditional metrics in most cases, and with proper filtering, the best-performing RandomForest algorithm according to the F-measure can even achieve an improvement of up to 11%. Finally, it is proven that the introduced metrics are even suitable for predicting bugs that will appear later in a given project if sufficient learning data is available.

INDEX TERMS Bug prediction, method level, hidden dependencies, metrics.

I. INTRODUCTION

During software development, one of the biggest challenges is to ensure the quality of the software, to discover and correct the bugs/errors in it, and to avoid newly introduced bugs during the possible development of the software. There are many tools to detect easily findable bugs in software, even during development. However, noticing errors that require a deeper understanding of the program is much harder.

Bug prediction, or defect prediction is therefore a very important area of software development, with extensive literature and solutions. These methods can be used to estimate which program elements (file, class, method) are most prone to misbehaving without executing the program. This is of great importance in program testing, since the limited testing capacity can be concentrated on checking these program elements.

Most of the existing methods, as the one presented, are product-based solutions that predict bugs by analysing the source code of the program. The most common way of this is to build an AST (abstract syntax tree) for the program and to compute the appropriate metrics on it. The size and complexity of the program elements can be well defined by this method, and it is also possible to specify, for example, how many other methods call a given method, which is important for the computation of the coupling metrics [3]. However, these solutions only estimate the presence of errors from data that can be read directly from the AST, without the more complex analyses that require an understanding of the program, which could point out errors that may be hidden in this way.

The associate editor coordinating the review of this manuscript and approving it for publication was Khursheed Aurangzeb.

Solutions that take into consideration the control and data dependencies in the program, including hidden dependencies, where there are not any direct dependencies among the dependent elements, are difficult to implement, much more expensive, and often cannot be adapted to larger programs. However, a major bug is often introduced into the program because it is difficult to see and understand the dependencies within the system that ultimately resulted in the bug.

To understand how a program works or to determine the source of a bug in a program, programmers often turn to methods that try to follow human thinking. One such method is program slicing [35], which, starting from an arbitrary statement in the program, determines the statements that are affected by that statement or that affect the statement, thereby narrowing the search space where the source of the error may be. Surprisingly, there are only a few publications using program slice metrics in error prediction models [4], [36]. This may be because computing program slices for larger programs can be very time-consuming, and their practical application is therefore questionable.

In most error prediction solutions, the program is handled in a larger unit, such as the module, file or, in the case of object-oriented languages, the class. According to the examined granularity, the method specifies how reliable a given program unit is or whether it may contain errors. Of course, locating a potential error is much more difficult if it is only known that the file contains an error instead of this information being more specific, for example at the method or statement level.

There are solutions, similar to the method introduced in this paper, that try to efficiently predict errors by approximating the slicing, but these solutions also tend to provide only a file-level solution [1]. These methods, or the metrics defined by these methods, can be used to predict errors as traditional software metrics, but because they take into account not only the structure of the program but also its behaviour, they can be even better [27].

The presented method of this work approximates the program slicing method and proposes metrics that can be used to predict software bugs more effectively than traditional software metrics. The goal is to provide an approach that is computationally efficient at the method level for large systems, as opposed to slice-based metrics, and thus suitable for method-level bug prediction. The method-level prediction helps to ensure the accuracy of the bug localization.

The introduced metrics are based on the previously introduced Static Execute After (SEA) relation and the graph used to define the relation [16]. The essence of the method is that it tries to estimate possible program executions based on the call graph and infer dependencies between program elements based on this. An earlier work has shown that the dependency sets computed by the SEA method approximate program slicing sets well at the method level [15]. At the same time, the computational requirements of SEA sets are significantly smaller than those of slicing sets, so they can be efficiently implemented for large programs.

In this paper, the accuracy with which models built from metrics derived from SEA sets can estimate erroneous methods is investigated. The models were built and evaluated on the BugHunter dataset [9], which is widely used in the field. The results showed that bug prediction models built from metrics derived from SEA sets using different learning algorithms are generally more accurate than models computed from AST metrics. The best performing Random Forest models show an 11% increase in F-measure.

The results are summarised by answering the following research questions:

- **RQ1:** How suitable are the SEA-based metrics, which also take into account the behaviour of the software, for predicting bugs at the method level? Are these metrics better than traditional metrics for predicting bugs?
- **RQ2:** How much can prediction results be improved by properly filtering the dataset created with the SEA-based metrics?
- **RQ3:** Can the introduced metrics help error prediction by themselves when trained and used on a particular project? What are the limitations of such a learning approach?

The rest of the paper is organized as follows. Section II describes the most common solutions and datasets in the area of bug prediction. Section III deals with the Static Execute After relation, which is a method-level approximation of program slicing dependencies, and defines the SEA-related metrics that are used for bug prediction. Section IV briefly introduces the dataset on which the practical evaluation is implemented. Research questions are answered in Section V by presenting and evaluating practical measurements, and then summarising the results in Section VI.

II. RELATED WORKS

Thanks to the prominence of the topic, numerous methods and datasets are available for almost every programming language in the field of bug prediction [25]. Although the method of this work is language-independent, in this paper it is evaluated on Java programs, so the literature review focuses on solutions and evaluation databases that analyse error prediction solutions for programs written in Java.

In most solutions, some kind of metric provides the basis for bug prediction. These can be some kind of product or process-based metrics [21], [22], but can also be based on traditional source code metrics or object oriented metrics [2], [5], [29]. Some of the solutions examine the errors of previous versions of a given software and their characteristics for error prediction [19], [24]. All of these methods work with easily definable features that focus on the structure of the program. Techniques that take into account the behaviour of the program may include anti-pattern and code-smell solutions [18], [26], [30], but they have the disadvantage of not being able to detect all kinds of bugs. However, when examining the effects of changes to the program that may lead to a bug, it is not enough to examine only the dependencies

that can be directly read from the code, or which occur based on a given pattern. So-called hidden dependencies should also be investigated, which can be considered as special data flows [32], [37]. At the class level, this hidden dependency can occur between classes that are apparently unrelated, but which are nevertheless related through a mediator class and influence each other's functioning. At the method level, it may involve methods that are not in a direct or indirect call relation, but affect each other by virtue of their execution one after the other [32].

Methods that look at the behaviour of the program are most often based on the program slicing method [1], [27], which is capable of detecting these dependencies at the statement level, but because of its cost, only a class-level approximation is used in prediction. Although prediction may be more efficient for approaches that predict errors at a higher granularity, identifying the error itself in a larger code snippet is more difficult. In addition, practical measurements have shown in [20] that on average only 17% of the methods in a file that is said to be faulty are responsible for the error, i.e. actually finding the error requires scanning a lot of code that is not even related to the error. Although the proposed method in this paper is metric-based, these metrics are defined on the program representation that aims to approximate program slices, and thus all kinds of direct and hidden dependencies, at the method level. Based on the understanding of the program, it has the property that the metrics of code fragments that are harder to understand and process are different from the metrics of code fragments of similar complexity that are easier to understand and process because of their role in the program. Although these are not the metrics that can be determined by slicing, the underlying method for determining the metrics has been shown to be efficient in approximating slicing at the method level, while still being efficient for large programs [15].

In order to evaluate the presented method, it is essential to select an appropriate bug database. To provide a bug prediction solution, which is ready for industry use, it is essential that it is evaluated on a properly labelled and trained database, possibly compared with the results of other methods [6]. The extensively used Java bug prediction datasets are Promise [17], the Bug Prediction dataset [7], Eclipse Bug Dataset [38], Bugcatchers Bug Dataset [12], and the GitHub Bug Dataset [31], which are used by the Unified Bug dataset [10] that unifies a larger dataset to make different prediction methods more widely evaluable. These datasets contain several small and large projects, each with several versions, and in these versions, the program elements are labelled buggy or not buggy based on the entries and tags of the version tracking systems. Although the Unified Bug Dataset itself contains a large number of entries (47,618 at the class level and 43,744 at the file level), which makes these databases suitable for testing various machine learning or deep learning methods with a sufficient amount of data, the disadvantage of these databases is that in most cases only high-level bug assignments are available, with few assignments at the method level. This may be because extracting method information from existing version tracking systems is a more difficult problem [13].

The method-level bug prediction of Giger et al. [11] and Hata et al. [13] has already demonstrated practically that the smaller the area of the bug, the more efficiently it can be predicted. The former has built its own dataset to prove its method, but it is not publicly available. The first major dataset to examine Java programs with method-level data is the ELFF dataset [28]. This dataset contains 69 versions of 23 projects. This dataset already contains significantly more method-level data than the previous ones, but the number of buggy and non-buggy methods is very unbalanced. Mo et al. [20] have made a similarly large method-level bug database available for Java programs. This classifies 45,682 methods of 18 projects into bug-prone and non-bug-prone classes (13,571 methods are bug-prone, while 32,071 are not bug-prone). A weakness of their database may be that they classify methods that they have not seen modified in bug-fix commits as not buggy.

The BugHunter dataset [9] using the experience gained from the creation of the databases available up to that time, provides perhaps the largest method-level bug prediction dataset for Java. Although it only examines 15 projects, these projects have more than 13,000 versions, and these versions have 150,000 methods that are classified as buggy or not buggy based on the version tracking system of the project. The advantage of this dataset is that in each case it only classifies those methods that can be uniquely classified by the bug report or fix associated with that version. With this method, it may be able to reduce the number of false negatives substantially compared to previous databases. Since the method presented in this paper is evaluated on the BugHunter dataset, it will be described in more detail in Section IV.

III. STATIC EXECUTE AFTER RELATION

Most often, statically definable metrics are used to predict bugs in software. These metrics describe the structure of the methods/classes and their relationship to other methods/classes. The intuition behind the use of these metrics is that a complex method/class with many connections likely to be responsible for errors in the software than those that are small and simple with few connections. Quite simply because they are more difficult to understand, to see through and, if necessary, to modify.

These metrics often do not take into account the effects of the hidden dependencies of the software because these dependencies arise between program components that appear to be completely unrelated [37]. Moreover, in most cases, the relationships are determined based on only statically available information among the source code elements. However, the number of connections can be much larger, for example through polymorphism and inheritance in object-oriented programs.

As a motivating example, consider a snippet of the JUnit program, one of the analysed programs of the evaluation,

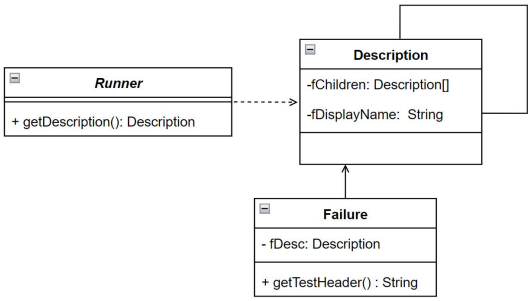

FIGURE 1. Motivation example.

from which the part corresponding to the class diagram in Figure 1 could be highlighted. The description of the TestSuits' tests is provided by the `Description` class according to the Composite design pattern, which is generated by various concrete implementations of the abstract `Runner` class by executing the `getDescription()` method. When a `Description` object is created, the value of the `fDisplayName` final data member is set. When the tests are run, there will be tests that fail, in which case `Failure` objects are created and initialized with the `Description` object associated with the failing test. Any display that then wants to list the failure will call the `getTestHeader()` method of the `Failure` object, which effectively returns the `fDisplayName` of the description for that failure object, as set by the `Runner` class. It can be seen that there is no concrete relationship between the `Runner` and `Failure` classes (class level), and no call relationship between the `getDescription()` and `getTestHeader()` methods (method level), however, the name that the former sets determines what will be returned in the event of a failure. That is, there is a hidden dependency between these classes or methods. The aim of this given work is to give some metrics that are also able to take into account such types of dependencies.

A. DEFINING THE SEA RELATIONS

The Static Execute After relations of the methods, presented in [15], are intended to also reveal these hidden dependencies in the program. The idea for the relation was given by the Execute After relation of Orso et al. [23], which is based on the analysis of execution traces of a program. The essence of this relation is that it relates program elements that run one after the other during the execution of the program.

The Static Execute After (SEA) relation, which can be derived from static information available from the source code analysis of the program, was defined based on this dynamic Execute After relation, which in turn can be determined during the execution of the analysed program.

Formally the SEA relation for methods f and g is defined as follows:

$$SEA = CALL \cup RET \cup SEQ[\cup ID],$$

where

$$\begin{aligned}
(f, g) \in CALL &\iff f \text{ (transitively) calls } g,\\
(f, g) \in RET &\iff f \text{ (transitively) returns into } g,\\
(f, g) \in SEQ &\iff \exists h : h \text{ (transitively) calls } f \text{ first,}\\
&\quad \text{then } h \text{ (transitively) calls } g, \text{ and}\\
&\quad \text{the second call site is flow-reachable}\\
&\quad \text{from the first one.}\\
(f, g) \in ID &\iff f = g
\end{aligned}$$

It should be pointed out, however, that the determination of these relations, should take into account all possible destinations of an actual call location from a late binding. That is, for example, in a C language, calls implemented with function pointers must be handled separately, processing each possible destination the call might point to. Similarly, for object-oriented languages, a call may have multiple destinations due to polymorphism, all of which must be taken into account when analyzing the program. Even a pointer analysis, or just a simpler heuristic algorithm, can be used to connect not only the call targets that can be determined from static information, but all potential targets in the call graph. Based on the call graph the above relations, which form the basis of the SEA relation, are easily computable. Of course, the determination of SEA relations may give relations that will not be realized during program executions, so it gives only an upper estimation compared to the original Appiwatanapong definition. At the same time, the SEA relation can be easily computed without dynamic executions of the program.

B. DETERMINING THE SEA RELATIONS

This chapter briefly introduces the concepts that are necessary for a simple determination of SEA relations among program methods. The program representation presented here forms the basis of the simple metrics defined in Section III-C, which will form the basis of the practical evaluation in Section V.

First, the call graph of the program must be defined. However, it is not enough to determine a call graph based only on static type information, it is important to map all possible call relations that could occur during any program execution. This can still be specified just by parsing the source code. However, care should be taken to ensure that the call graph defined in this way contains all the call relations that can be realised by the different executions of the program. The evaluation of this paper uses the so-called Class Hierarchy Analysis approach [8], but of course, a more accurate but time-consuming method could also be usable.

By itself, a call graph has much less information than is needed to define the SEA, since the potential execution order of the call sites within a given method is not known, and without it, the SEQ relations that are part of the SEA are not determinable. Therefore, as a second step, the so-called intra- and interprocedural Component ControlFlow

Graph (CCFG and ICCFG, respectively) are introduced. Each CCFG represents a single method. The representation is much poorer than a control flow graph representation, but sufficient enough to isolate the call site locations within a method and to provide information about the order in which control can be placed on each call site. Each CCFG graph consists of an entry node and a number of component nodes connected by control edges. Technically, the CCFG is derived from the CFG, but by keeping only the call sites and the paths between them, and by also taking the strongly connected components of this reduced CFG graph, these components become the vertices of the CCFG graph. The explanation for this is that if multiple call sites are connected through a loop, then the destinations of those call sites will be mutually SEA-related, which can be expressed by a single node with a reflexive control flow edge. Connecting the CCFGs of the methods gives the analysed program's ICCFG, which can be used with its appropriate traversal to determine the SEA relations between methods.

C. SEA-BASED SOFTWARE METRICS

In this section, the SEA-based metrics used in the evaluation are presented. These metrics are very simple to compute using the ICCFG graph. Of course, based on the ICCFG, many other, even better metrics could be also definable, but the purpose of this paper is only to present that even these very quickly computable metrics have in most cases better bug prediction abilities than metrics that only look at statically determinable data and direct relationships. Since the definition of these metrics is based on a relation or program representation that can be used to approximate not only the AST-based, but also the hidden dependencies in the program, the metrics introduced are considered hidden dependency metrics, referred to in the rest of this work as SEA-based metrics.

In the evaluation, the following metrics are assigned to each method analysed:

- **Number of caller edges**: the number of components from which the method is directly reachable with a call edge. This number is an upper bound of the number of methods from which the method can be called directly (or to which the current method can return directly), since there may be situations when a method calls another method several times, from different components.
- **Number of call blocks**: the number of components in the method's CCFG graph that contain at least one call edge to a method which is a part of the analysed system. This metric value is higher for methods that are more complex and harder to understand because of the large number of method calls. However, for a method that is otherwise considered complex and contains few or no calling edges, this value will be small, since even though it takes more time to understand the method, everything is locally given.

TABLE 1. The characteristics of the selected projects.

Project name	Number of analysed version	investigated methods	buggy methods	not buggy methods
Android Universal Image Loader[a]	61	251	88	163
ANTLR v4[b]	116	427	132	295
Elasticsearch[c]	4 060	49 681	19 060	30 621
jUnit[d]	58	512	112	400
MapDB[e]	194	1 778	638	1140
mcMMO[f]	403	1 340	476	864
Neo4j[g]	695	9 686	3 170	6 516
Netty[h]	1 344	13 508	3 601	9 907
OrientDB[i]	959	13 160	4 429	8 731
Oryx 2[j]	87	842	100	742
Titan[k]	110	1 009	285	724
Eclipse plugin for Ceylon[l]	385	2 357	652	1 705
Hazelcast[m]	3 817	56 218	23 694	32 524
Broadleaf Commerce[n]	781	5 225	1 315	3 910
TOTAL	13 070	155 994	57 752	98 242

[a] https://github.com/nostra13/Android-Universal-Image-Loader
[b] https://github.com/antlr/antlr4
[c] https://github.com/elastic/elasticsearch
[d] https://github.com/junit-team/junit4
[e] https://github.com/jankotek/MapDB
[f] https://github.com/mcMMO-Dev/mcMMO
[g] https://github.com/neo4j/neo4j
[h] https://github.com/netty/netty
[i] https://github.com/orientechnologies/orientdb
[j] https://github.com/OryxProject/oryx
[k] https://github.com/thinkaurelius/titan
[l] https://github.com/eclipse/ceylon-ide-eclipse
[m] https://github.com/hazelcast/hazelcast
[n] https://github.com/BroadleafCommerce/BroadleafCommerce

- **Number of callee edges**: the number of call edges originating from the component nodes of the method's CCFG. This number estimates from above the number of methods in a CALL relationship with the given method.
- **Number of SEA relations**: the number of the SEA relation of the method, that is, the number of methods that are SEA-related to the given method. A method that runs after another method can potentially depend on that method, even if that dependency is not easily visible.
- **Number of CALL relations**: the number of the method CALL relations, i.e. the number of methods that can be accessed directly or indirectly from the given method through call edges.
- **Number of RET relations**: the number of methods to which the method can return directly or indirectly.
- **Number of SEQ relations**: the number of methods with which the method has a SEQ relation.
- **CALL∪RET relations ratio**: ratio, which compares the number of methods in call and ret relation to the number of methods in SEA relation with the method.
- **"Strong" relations ratio**: ratio of the number of methods among the SEA connections with which the given method has both SEQ and CALL or RET connections.
- **Only SEQ relations ratio**: ratio of the number of methods among the SEA connections with which the given method has only SEQ connections.

- **Has loop**: attribute with two classes, its value is 1 if at least one of the components of the given method has a reflexive edge, i.e. the method contains a cycle that contains method calls. Otherwise its value is 0.

IV. BUGHUNTER DATASET

In this paper, the presented method is evaluated on the BugHunter dataset introduced in [9]. In this study, the main parts of the evaluation of the paper [9] are repeated in order to compare the prediction ability of the introduced metrics and the traditional software metrics they used. In the evaluation, the introduced metrics are computed by the Open Static Analyser tool.[1]

A. DATA SOURCE

The BugHunter dataset is based on a few GitHub projects. GitHub is one of today's most popular source code hosting services, which includes features such as built-in bug and issue tracking. This feature made it possible to classify commits by bug fixes and to classify certain methods as buggy or not buggy in a given revision. In contrast to other bug databases, the authors of the BugHunter dataset examine thousands of revisions of different Java projects, and in them, they characterize only the methods that they know have had a bug fixed in that revision by their modification, and thus the bug has been eliminated, or they specify methods that are supposed to have a bug in the revision under examination, but have been fixed in the next revision.

When creating this bug database, the authors focused on GitHub projects that were written in Java and were larger in size, as these systems more often contain issues tagged as bugs that are subsequently fixed at some point, and this fact is reflected in the log messages. In addition, their selection criteria included that the chosen projects should be active and popular, ranked by the number of stars given by logged-in users.

The authors analyzed several revisions of the BugHunter dataset projects, and determined the methods associated with each revision that were classified as buggy or non-buggy. In some cases, a method was affected by multiple errors, so instead of simply assigning a binary classification to the methods, a metric was assigned depending on how many errors the method was affected by. The authors observed that in some cases the same metric values were associated with different bug numbers. In order to reduce the noise in the BugHunter dataset and to eliminate these redundancies, different filtering solutions were tried when designing the database. Of these, the best results were obtained with what the authors call "subtract" and "removal" filtering. Analysing these filterings separately, the authors' final conclusion was that subtract filtering was the most appropriate, so we used this filtering as the starting point of our practical evaluation.

[1] https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

TABLE 2. The method level metrics of the BugHunter dataset.

Abbreviation	Full name
CLOC	Comment Lines of Code
LOC	Lines of code
LLOC	Logical Lines of Code
NL	Nesting Level
NLE	Nesting Level Else-If
NII	Number of Incoming Invocations
NOI	Number of Outgoing Invocations
CD	Comment Density
DLOC	Documentation Lines of Code
TCD	Total Comment Density
TCLOC	Total Comment Lines of Code
NOS	Number of Statements
TLOC	Total Lines of Code
TLLOC	Total Logical Lines of Code
TNOS	Total Number of statements
McCC	McCabe's Cyclomatic Complexity
HCPL	Halstead Calculated Program Length
HDIF	Halstead Difficulty
HEFF	Halstead Effort
HNDB	Halstead Number of Delivered Bus
HPL	Halstead Program Length
HPV	Halstead Program Vocabulary
HTRP	Halstead Time Required to Program
HVOL	Halstead Volume
MIMS	Maintainability Index (Microsoft version)
MI	Maintainability Index (Original version)
MISEI	Maintainability Index (SEI version)
MISM	Maintainability Index (SourceMeter version)
NUMPAR	Number of Parameters

B. DATA CORRECTION

Unfortunately, the selection strategy of the projects was not entirely appropriate, because one of the selected projects, NASA's Mission Control Technologies, has since become unavailable at the link provided, so this project was omitted from this current analysis, and thus also its classified methods.

In addition, there were some method names in the database that were not possible to identify with the analyser used. Since the dataset identified the methods only by their JNI names, and in some cases not by their exact names due to some type resolution issue, these methods could not be identified unambiguously. Similarly, due to the different numbering of anonymous methods, there were some not clearly identifiable methods.

Since the number of these cases did not vary much with the size of the database, these methods were also omitted from the database and the evaluation, rather than possibly mismatching other methods by a heuristic. In order to make the existing dataset usable for everyone in the future, independent of the generation of the JNI name of the methods, or the numbering of the anonymous methods, in the online appendix of this paper, the line information alongside the method names are provided, which will make their identification clear.

Overall, this omission resulted in a slightly smaller number of methods than in the original "subtract" dataset. Table 1 summarises the characteristics of each of the projects on which the evaluation was carried out in Section V. In the case of two projects, significantly more methods were analysed than in the others. This is important in the present work, since looking for the answer to how much each project in itself, as an independent error prediction model, can help the further development of the given software is also of key interest.

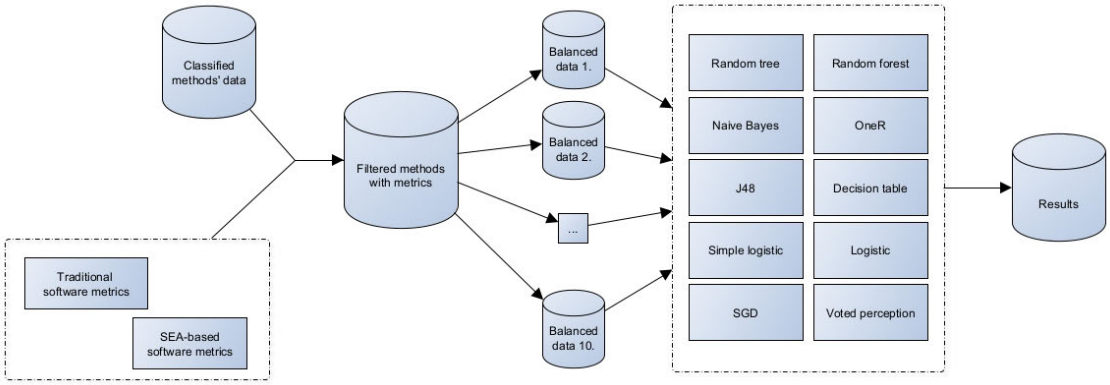

FIGURE 2. The main steps of the evaluation.

C. USED METRICS

The BugHunter dataset gives traditional statically computable metric values for each analysed method at method, class and file level. Using these metrics in learning at different levels, the authors of the dataset realized that the metrics defined at the method level proved to be the best predictors in terms of bug prediction. Since the current evaluation deals with metrics that can be defined at the method level, so only the method-level metrics of the BugHunter dataset are used in the current evaluation for comparison. The evaluation of this paper repeats the measurements presented in [9] with the metrics of Table 2 and with the set of methods that were slightly reduced according to the subsection IV-B, and then compares these results with the predictive ability of the metrics defined in subsection III-C.

V. EVALUATION

The evaluation of the paper summarizes the achieved results. The measurements implemented in [9] were partly repeated to show that using SEA-based metrics instead of traditional software metrics can even give more accurate results. Since the initial dataset was slightly modified, it is important to be able to repeat the original learnings and compare the recomputed data with the learning results of SEA-based metrics. This comparison forms the basis of the first RQ.

The practical evaluation followed the process shown in Figure 2 for answering each of the questions. The main strength of the BugHunter dataset is that it classifies methods of many versions of many projects. These methods are labelled as buggy or not buggy. The BugHunter dataset also provides the traditional metric values for these methods, which are highlighted in Section IV-C. In addition to the traditional metrics, SEA-based metrics were also calculated for labelled methods. To compare which set of metrics has better predictive ability, all measurements were performed with the same settings. Measurements were carried out by the Weka tool [34], which, in each case, took the learning algorithms with their own default settings.

The metric values assigned to methods can be redundant in the sense that the exact same metrics can be associated with multiple methods. It is also possible that, for the same metrics, the classification of methods is different. That is, in some cases, the method is labelled as buggy, and in some cases not buggy. Cases like these make the database noisy, so filtering them in different ways to get a more accurate result before the actual evaluation. The effects of different filterings are investigated by the second RQ.

Since the number of buggy and non-buggy methods in the dataset is unbalanced, random undersampling[2] [14], [33] was applied to the data, also similar to [9], to ensure that the number of elements in the two categories is equal, adjusted to the number of elements of the smaller category. This was repeated 10 times using different seeds, and the learning was performed on 10 balanced datasets, after which the results was averaged. During the training, 10-fold cross-validation was used to measure the accuracy of the models with different machine learning algorithms. The purpose of filtering (i.e. reducing noise), the repeated preparation of balanced data sets, and the application of 10-fold cross-validation are to reduce overfitting. While answering the last RQ, this process was modified so that the balanced datasets always consisted of only the methods of a single project, i.e. not the entire dataset was examined.

To compare the different models, the precision, recall, and F-measure metrics were used in the usual way:

$$precision = \frac{TP}{TP + FP},$$
$$recall = \frac{TP}{TP + FN},$$
$$F\text{-}measure = 2 \cdot \frac{precision \cdot recall}{precision + recall},$$

where TP is the number of methods correctly marked as buggy, FP is the number of methods marked as buggy which were not actually buggy, and FN is the number of methods

[2]weka.filters.supervised.instance.SpreadSubsample Weka's filtering.

TABLE 3. The ranking list of the learning algorithms based on the difference of the SEA-based and traditional metric based F-measure values. The best algorithm is bolded.

Algorithm	Precision		Recall		F-measure	
	traditional	SEA	traditional	SEA	traditional	SEA
NaiveBayes	0.5441	0.5851	0.523	0.5809	0.4594	0.5751
OneR	0.5983	0.705	0.5983	0.7015	0.5982	0.7005
DecisionTable	0.6181	0.666	0.6082	0.6611	0.5998	0.6583
RandomForest	**0.6604**	**0.7161**	**0.6594**	**0.7159**	**0.6587**	**0.7158**
RandomTree	0.6518	0.6884	0.6517	0.6881	0.6517	0.6881
J48	0.6452	0.6619	0.638	0.6615	0.6333	0.6614
VotedPerceptron	0.5605	0.5854	0.5455	0.5688	0.5212	0.5472
SimpleLogistic	0.589	0.5929	0.589	0.5929	0.589	0.5929
SGD	0.5927	0.5899	0.5914	0.5899	0.5902	0.5898
Logistic	0.5951	0.5901	0.5946	0.5901	0.594	0.5901

marked as not buggy that should have been classified as buggy.

A. EVALUATION ON THE BUGHUNTER DATASET

The BugHunter dataset classifies thousands of methods and labels them as buggy or not buggy. However, it also provides each method with its traditional software metrics, which can be used to test any learning algorithm with any setting. The first RQ seeks to answer the question of how well the SEA-based metrics can improve error prediction for the most frequently used machine learning algorithms.

> **RQ1:** How suitable are the SEA-based metrics, which also take into account the behaviour of the software, for predicting bugs at the method level? Are these metrics better than traditional metrics for predicting bugs?

Performing the same learnings on both the traditional and SEA-based metrics, can be found that although the number of SEA-based metrics is much fewer than the number of traditional metrics, their bug prediction ability is better in most cases. Table 3 compares the precision, recall, and F-measure values obtained. For easier transparency, the rows in the table are sorted by the difference of the F-measure metrics when using SEA-based metrics as opposed to the traditional metrics. As can be seen, the newly introduced metrics could improve the F-measure values of the NaiveBayes algorithm by more than 11% and even OneR shows an improvement of over 10%. There are only two cases where the F-measure values are worse, but in these cases, the differences are almost negligible. It is important to highlight that the best-performing RandomForest algorithm for traditional metrics also improved by 6% in terms of F-measure using SEA-based metrics.

As pointed out earlier, for a given set of metrics, noise in the database is conceivable in the sense that a method will be labelled as both faulty and not faulty in cases with the same metric values. That is why it is worth filtering the dataset before learning to manage noise. In this measurement, the filtering was selected in a way to gave the best results for traditional metrics. However, by replacing the metrics with a completely different set of metrics, it is possible that this filtering is inappropriate and results in noise, which reduces the predictive ability of the new set of metrics.

> **Answering RQ1:** It can be concluded that SEA-based metrics performed better than traditional metrics for most learning algorithms, even when the dataset was filtered by the latter metrics, and SEA-based metrics may have introduced noise.

B. FILTERING

Since the filtering strategy of the BugHunter dataset is based on the similarity of the metrics they provide, the question arises whether the same filtering using on completely different metrics than the original will be better or not. That is why, the measurements of this section repeated the filtering itself, but this time on SEA-based metrics, and compared the results.

> **RQ2:** How much can prediction results be improved by properly filtering the dataset created with the SEA-based metrics?

The BugHunter dataset distinguishes between 4 basic filtering states and one no filtering state. These filterings sort out methods that are identical in their metric values, differing only at least in their classification label, i.e. whether the method is buggy or not. In the unfiltered version, which is denoted as "ALL", none of the methods will be filtered out. Filtered datasets are obtained by filtering this initial **ALL** dataset. For each of these methods, the possible sets of metrics associated with them are taken. For each occurring set, it is determined how many times the set is buggy and how many times it is not buggy in the **ALL** dataset. After that, the filtered datasets are defined in a similar way to BugHunter's dataset filtering.

- **REMOVAL**: it keeps a given set of metrics with the same number of elements as the majority class.

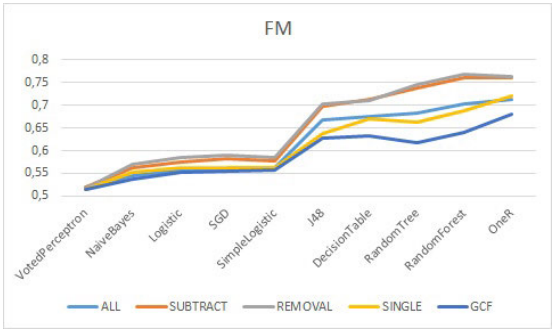

FIGURE 3. F-measure values using the SEA-based metrics and based on different filtering methods.

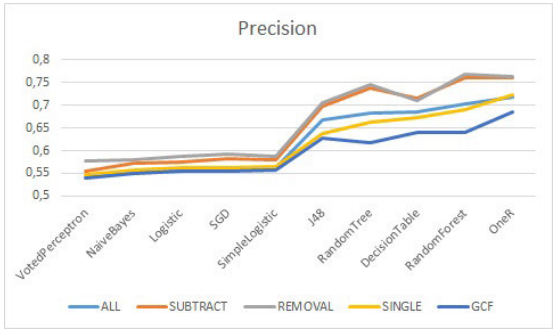

FIGURE 4. Precision values using the SEA based metrics and based on different filtering methods.

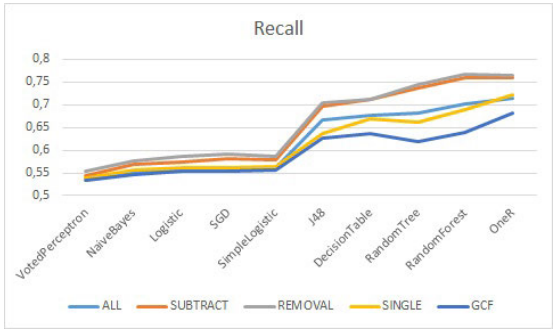

FIGURE 5. F-measure, precision, and recall values using the SEA-based metrics and based on different filtering methods.

- **SUBTRACT**: it keeps a given set of metrics as many times as the difference between the number of elements of the majority and minority classes.
- **SINGLE**: it keeps a single instance of a given set of metrics with the label of the majority class. If the number of the two classes is the same, then it does not keep a copy of this metric set.
- **GCF**: it keeps the labels of both classes of a given set of metrics, but instead of the original multiplicity of the classes, it divides the original multiplicities by the greatest common divisor of the multiplicities of the two classes.

Based on Figures 3, 4, and 5, by using and filtering the SEA-based metrics, similar results were obtained to those found when filtering the BugHunter dataset with the traditional metrics.

That is, overall, it can be said that the best learning results can be achieved by filtering the entire dataset with the "removal" and "subtract" filtering. These two are very close overall. At the same time, the "single" and "GCF" filters also produced results below the baseline in the current measurements.

Although it can be seen from the figures, and also from the online appendix, it can be quantified that the efficiency of the RandomForest algorithm, which is the most efficient for traditional metrics, is further improved for SEA-based metrics, with an F-measure value of 76% for the "subtract" filtering and 76.7% for the "removal" filtering.

Answering RQ2: It can be stated that by filtering the database according to SEA-based metrics and then performing the learning on these filtered datasets, learning outcomes continue to improve, up to 10.8%, for the most successful learner. However, it can be also concluded that different filterings of a fixed set of metrics had a similar effect on the results as it was seen in the results of [9]. That is, this measurement came to a similar conclusion as in [9], namely that the "subtract" and "remove" filtering techniques were the most effective for prediction.

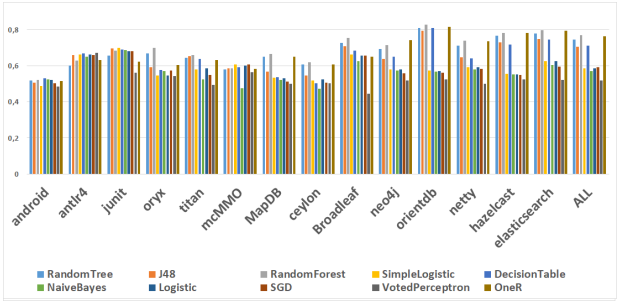

FIGURE 6. Average F-measure values per projects with "removal" filtering.

C. PROJECT-BASED BUG PREDICTION

In the measurements so far, the learning algorithms were evaluated on a model based on several projects. However, the question arises as to how good error predictors the given metrics can be for a separate but independent project. To test this, the purpose of the following experiment was to see how well the bugs can be predicted in a project by creating the learning model based only on the given project. The last RQ is formulated accordingly.

RQ3: Can the introduced metrics help error prediction by themselves when trained and used on a particular project? What are the limitations of such a learning approach?

The measurements followed the same steps as the evaluation of the full database. Multiple occurrences of the same set

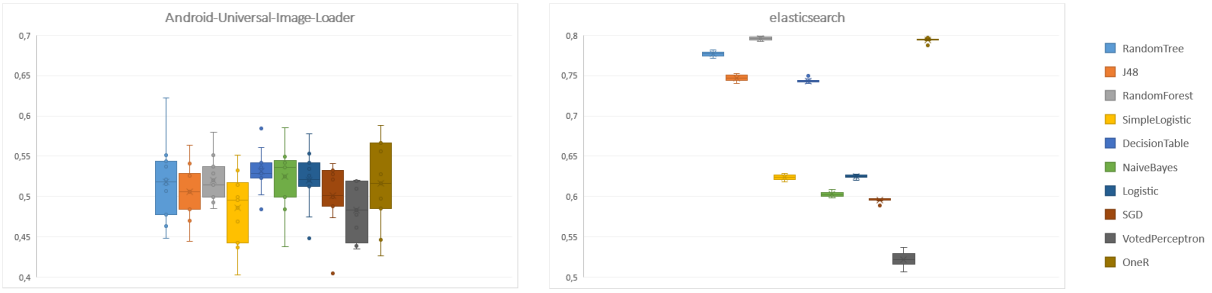

FIGURE 7. The deviation of F-Measure values of the smallest and the largest model.

of metrics can cause noise here too, so the filtering techniques described in the previous sections were applied. Since the two best filterings were the ''removal'' and ''subtract'' filterings, these were the only ones measured.

The filtered methods of the given project were then further balanced so that the number of methods with and without bugs was the same. The algorithm used was again Weka's SpreadSubsample algorithm. After that, the models were trained and tested using 10-fold cross-validation. All of these were repeated 10 times using different seeds for the first filtering step and the results were averaged.

The basic hypothesis was that this kind of learning would only be successful if a large amount of training data was available, but it was realized that this was not necessarily true. For each project filtered both with filtering ''removal'' and ''subtract'' the resulting F-measure, precision, and recall values were compared with the values obtained on the full dataset. Figure 6 shows the F-measure values with ''removal'' filtering. The recall and precision values, as well as the values calculated for ''subtract'' filtering, show similar trends. These results can all be found in the online appendix. In the figure, the projects are sorted by ''size'', i.e. their order reflects the number of methods analysed in each project.

The conclusion is that only for the Android-Universal-Image-Loader, ceylon-ide-eclipse, and MapDB projects there were no learning algorithms that on average could perform better on the individual project than on all data. Their total number of analysed methods is 251, 2357, and 1778, respectively, i.e. they were the smaller projects. However, there were learning algorithms that performed better in the similarly small titan, junit, and oryx, with 1009, 512, and 842 analysed methods, respectively. However, the hazelcast and elasticsearch projects with significantly more methods analysed still gave much higher F-measure values.

Looking at the learning algorithms, it is clear that for all projects, the RandomTree, RandomForest, ONeR, DecesionTable, and J48 algorithms were the most successful in using SEA-based metrics both at the project level and for the whole dataset.

There is, however, another important conclusion from these measurements. To ensure that for each learning algorithm, the dataset is based on the same number of buggy and non-buggy methods, random undersampling was applied 10 times. The final values of precision, recall, and F-measure were obtained by averaging the results of the learning on the resulting datasets. Observed at during the measurements is how much the variance of these values depends on the size of the project. Without claiming to be complete, Figure 7 highlights the smallest and largest projects, as well as their separate F-measure values on each learnings, and their standard deviation. (Similar results of the precision and recall values are available in the online appendix.)

It can be found at that the average F-measure values of the small programs are not necessarily bad in each case, but the standard deviation is much larger for all measurements. These smaller projects often have outliers, too. For the larger programs, where more methods (at least 5000) were classified according to whether or not they were buggy, not only were the average F-measure values better, but the values of the different measurements were also closer to each other.

> **Answering RQ3:** Overall, it can be concluded that on a project in which the error habits are known, the SEA metrics are suitable for predicting the project's errors. However, it is important to point out that for unbalanced datasets, the result obtained for smaller projects is highly dependent on the balancing, and the more data it has, the more reliable the prediction becomes. The size of the training set is very important when building a prediction model.

D. ONLINE APPENDIX

The online database is available at: https://www.inf.u-szeged.hu/~jasy/research/hiddenMetricsInBugPrediction or https://dx.doi.org/10.21227/t74t-vj82. The biggest problem with using the BugHunter database was that the methods it contained were not always clearly identifiable in the source code. One technical reason for this is the generation of JNI names for the method names, which is not always perfect if the types are not resolved correctly by the parser. The other reason is simply that the naming of anonymous classes is not deterministic, so different analysers may give different names to the same anonymous class.

In order to make it easier to identify the methods in the database, the online appendix completes the original dataset with the physical location of each method (i.e. in which file, in which rows, the method is located) in the hope that this will facilitate similar evaluations by many in the future.

The online appendix of course also contains details of all the presented results. For each method, the online database also contains the calculated metric values and all the data from the evaluation of each research question.

VI. CONCLUSION AND THREATS TO VALIDITY

This paper presented a new approach for method-level error prediction. It showed that instead of using traditional software metrics, metrics that take into account the potential implementations of the software under study, and hence the dependencies that are realized, often allow for more efficient learning and prediction. The results were validated on an existing bug database where the prediction ability of the newly introduced SEA-based metrics was compared with the prediction ability of traditional software metrics. All of the results are available in the online appendix of the paper. The online appendix completes the original database with additional elements to help others to more easily and accurately identify the database's methods in the source code. Through the newly performed experiments, it is confirmed that the filtering strategy added to the BugHunter database has similar properties for both traditional and newly introduced metrics. Finally, the paper has shown that SEA-based metrics can be used to predict future project failures, even independently, using the known failures of a project.

This work is not extensive in the sense that it did not try to determine the best settings for each learning method, but only applied their default parameter settings. Nevertheless, it can be stated that the metrics presented predict errors in the programs at the method level with good efficiency. These results are better despite the fact that the evaluation used a relatively imprecise heuristic to detect call relations when calculating the introduced metrics. In future works, the author considers it important to examine how the metrics calculated on more expensive but more accurate call graphs improve the error prediction ability of the method. Furthermore, it is also planned to not only evaluate the introduced metrics using traditional machine learning methods, but also to demonstrate their predictive ability using deep-learning models. Planned future work also includes the evaluation of the suitability of the presented metrics for predicting bugs on other bug databases, possibly on bug databases of other programming languages.

REFERENCES

[1] B. S. Alqadi and J. I. Maletic, "Slice-based cognitive complexity metrics for defect prediction," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2020, pp. 411–422.

[2] V. Basili, L. Briand, and W. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[3] A. Beszedes, R. Ferenc, and T. Gyimothy, "Columbus: A reverse engineering approach," in *Proc. 13th Workshop Softw. Technol. Eng. Pract.*, Budapest, Hungary, Sep. 2005, pp. 93–96. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.80.6933

[4] S. Black, S. Counsell, T. Hall, and P. Wernick, "Using program slicing to identify faults in software," in *Beyond Program Slicing* (Dagstuhl Seminar Proceedings), vol. 5451, D. W. binkley, M. harman, and J. krinke, Eds. Dagstuhl, Germany: Schloss Dagstuhl-Leibniz-Zentrum Für Informatik, 2006. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/DagSemProc.05451.11

[5] L. Briand, J. Daly, and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 25, no. 1, pp. 91–121, Jan./Feb. 1999.

[6] S. Chowdhury, G. Uddin, H. Hemmati, and R. Holmes, "Method-level bug prediction: Problems and promises," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, pp. 1–31, 2024, doi: 10.1145/3640331.

[7] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, May 2010, pp. 31–41.

[8] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. European Conf. Object-Oriented Program.* Aarhus, Denmark: Springer, 1995, pp. 77–101.

[9] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, "An automatically created novel bug dataset and its validation in bug prediction," *J. Syst. Softw.*, vol. 169, Nov. 2020, Art. no. 110691. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301436

[10] R. Ferenc, Z. Toth, G. Ladanyi, I. Siket, and T. Gyimothy, "A public unified bug dataset for Java and its assessment regarding metrics and bug prediction," *Softw. Quality J.*, vol. 28, pp. 1447–1506, Dec. 2020, doi: 10.1007/s11219-020-09515-0.

[11] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Sep. 2012, pp. 171–180.

[12] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some code smells have a significant but small effect on faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 1–39, Sep. 2014, doi: 10.1145/2629648.

[13] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 200–210.

[14] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.

[15] J. Jász, "Static execute after algorithms as alternatives for impact analysis," *Periodica Polytechnica Elec. Eng.*, vol. 52, nos. 3–4, pp. 163–176, 2008.

[16] J. Jász, A. Beszédes, T. Gyimothy, and V. Rajlich, "Static execute after/before as a replacement of traditional software dependencies," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Oct. 2008, pp. 137–146.

[17] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.* New York, NY, USA: Association for Computing Machinery, 2010, pp. 1–10, doi: 10.1145/1868328.1868342.

[18] F. Khomh, M. Di Penta, Y.-G. Guhneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Eng.*, vol. 17, pp. 243–275, Jun. 2012.

[19] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.

[20] R. Mo, S. Wei, Q. Feng, and Z. Li, "An exploratory study of bug prediction at the method level," *Inf. Softw. Technol.*, vol. 144, Apr. 2022, Art. no. 106794. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921002330

[21] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, May 2008, pp. 181–190.

[22] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.

[23] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proc. 11th ACM SIGSOFT Symp. Found. Softw. Eng. Held Jointly, 9th Eur. Softw. Eng. Conf.*, Sep. 2003, pp. 128–137.

[24] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, Apr. 2005.
[25] J. Pachouly, S. Ahirrao, K. Kotecha, G. Selvachandran, and A. Abraham, "A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools," *Eng. Appl. Artif. Intell.*, vol. 111, May 2022, Art. no. 104773. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622000616
[26] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 244–255.
[27] K. Pan, S. Kim, and E. J. Whitehead Jr., "Bug classification using program slicing metrics," in *Proc. 6th IEEE Int. Workshop Source Code Anal. Manipulation*, Sep. 2006, pp. 31–42.
[28] T. Shippey, T. Hall, S. Counsell, and D. Bowes, "So you need more method level datasets for your software defect prediction? Voilà!" in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.* New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–6, doi: 10.1145/2961111.2962620.
[29] R. Subramanyam and M. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, Apr. 2003.
[30] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting bugs using antipatterns," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2013, pp. 270–279.
[31] Z. Töth, P. Gyimesi, and R. Ferenc, "A public bug database of GitHub projects and its application in bug prediction," in *Proc. 16th Int. Conf. Comput. Sci. Appl.* Beijing, China: Springer, Jul. 2016, pp. 625–638. [Online]. Available: https://link.springer.com/chapter/10.1007%2F978-3-319-42089-944
[32] R. Vanciu and V. Rajlich, "Hidden dependencies in software systems," in *Proc. 26th IEEE Int. Conf. Softw. Maintenance*, R. Marinescu, M. Lanza, and A. Marcus, Eds., Timisoara, Romania, Sep. 2010, pp. 1–10, doi: 10.1109/ICSM.2010.5609657.
[33] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans. Rel.*, vol. 62, no. 2, pp. 434–443, Jun. 2013.
[34] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann, 2016.
[35] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 2, pp. 1–36, Mar. 2005, doi: 10.1145/1050849.1050865.
[36] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, and Z. Zhang, "Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? An empirical study," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 331–357, Apr. 2015.
[37] Z. Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Proc. 9th Int. Workshop Program Comprehension*, Feb. 2001, pp. 293–299.
[38] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.* New York, NY, USA: Association for Computing Machinery, 2009, pp. 91–100, doi: 10.1145/1595696.1595713.

JUDIT JÁSZ received the Ph.D. degree in computer science from the University of Szeged, Hungary, in 2010. She is currently an Assistant Professor with the Department of Software Engineering, University of Szeged. Her main research interests include static program analysis and bug prediction. In addition to research, she is actively involved in the department's teaching activities and research development projects.