Received 11 July 2023, accepted 8 September 2023, date of publication 12 September 2023, date of current version 19 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3314572

RESEARCH ARTICLE

Mining Task-Specific Lines of Code Counters

MIROSLAW OCHODEK[1], (Member, IEEE), KRZYSZTOF DURCZAK[1], JERZY NAWROCKI[1], AND MIROSLAW STARON[2]

[1]Faculty of Computing and Telecommunications, Poznan University of Technology, 60-965 Poznan, Poland
[2]Department of Computer Science and Engineering, University of Gothenburg | Chalmers, 41756 Gothenburg, Sweden

Corresponding author: Miroslaw Ochodek (miroslaw.ochodek@put.poznan.pl)

The work of Miroslaw Ochodek was supported by the National Science Centre, Poland, within the research project "Source-Code-Representations for Machine-Learning-Based Identification of Defective Code Fragments" (OPUS 21), under Grant 2021/41/B/ST6/02510. The work of Jerzy Nawrocki was supported by the Poznan University of Technology under Grant 0311/SBAD/0738. The work of Miroslaw Staron was supported by the Software Center (www.software-center.se), University of Gothenburg | Chalmers.

ABSTRACT Context: Lines of code (LOC) is a fundamental software code measure that is widely used as a proxy for software development effort or as a normalization factor in many other software-related measures (e.g., defect density). Unfortunately, the problem is that it is not clear which lines of code should be counted: all of them or some specific ones depending on the project context and task in mind? Objective: To design a generator of task-specific LOC measures and their counters mined directly from data that optimize the correlation between the LOC measures and variables they proxy for (e.g., code-review duration). Method: We use Design Science Research as our research methodology to build and validate a generator of task-specific LOC measures and their counters. The generated LOC counters have a form of binary decision trees inferred from historical data using Genetic Programming. The proposed tool was validated based on three tasks, i.e., mining LOC measures to proxy for code readability, number of assertions in unit tests, and code-review duration. Results: Task-specific LOC measures showed a "strong" to "very strong" negative correlation with code-readability score (Kendall's τ ranging from -0.83 to -0.76) compared to "weak" to "strong" negative correlation for the best among the standard LOC measures (τ ranging from -0.36 to -0.13). For the problem of proxying for the number of assertions in unit tests, correlation coefficients were also higher for task-specific LOC measures by ca. 11% to 21% (τ ranged from 0.31 to 0.34). Finally, task-specific LOC measures showed a stronger correlation with code-review duration than the best among the standard LOC measures ($\tau = 0.31$, 0.36, and 0.37 compared to 0.11, 0.08, 0.16, respectively). Conclusions: Our study shows that it is possible to mine task-specific LOC counters from historical datasets using Genetic Programming. Task-specific LOC measures obtained that way show stronger correlations with the variables they proxy for than the standard LOC measures.

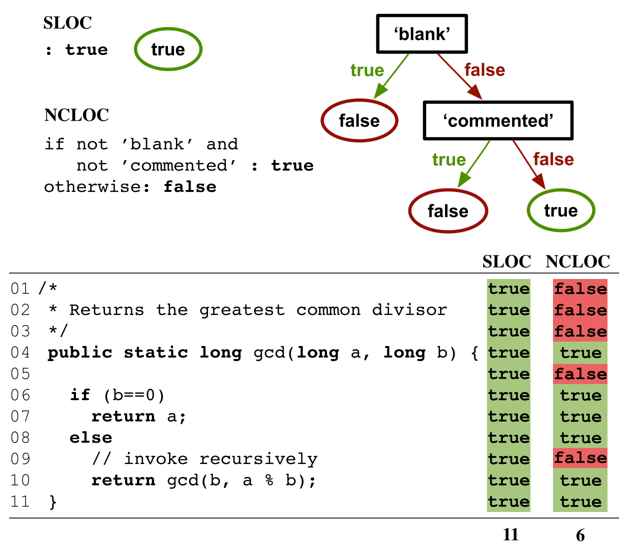
INDEX TERMS Software measurement, software size, lines of code, LOC.

I. INTRODUCTION

Source code measures are perceived as a good approximation of higher-level measures, e.g., software complexity or defect proneness. In particular, lines of code (LOC) is one of the oldest and most recognized software size measures, both as a measure of software size and a proxy for other measures, e.g., software development effort [1], [2], developers' productivity [3], defect density [4], program faults [5], testing effort [6], or pull-request latency [7]. Depending on a study, the strength of the dependencies between the LOC and the proxy differ [8], and recent studies on software size have found the possibilities of using LOC measures as the baseline for validating other measures, thus implicitly giving the lines of code measure a primary role in software measurement [9].

Typically, a procedure for counting lines of code, and thus the measurement instruments implementing it, consists of two steps. In the first step, each line has to be classified as either to be counted (`true`) or not (`false`), and in the

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



SLOC

```
: true   true
```

NCLOC

```
if not 'blank' and
   not 'commented' : true
otherwise: false
```

```
                                                   SLOC  NCLOC
/*                                                 true  false
 * Returns the greatest common divisor             true  false
 */                                                true  false
public static long gcd(long a, long b) {           true  true
                                                   true  false
  if (b==0)                                        true  true
    return a;                                      true  true
  else                                             true  true
    // invoke recursively                          true  false
    return gcd(b, a % b);                          true  true
}                                                  true  true
                                                   11    6
```

FIGURE 1. Defining SLOC and NCLOC with the use of decision rules and decision trees, and an example of applying the definitions to measure the size of a method.

second step, we count the lines for which the response was `true`. Depending on what is included in the counted lines, the definition of the measure changes. The simplest LOC definition is to count all the lines in the source files (SLOC). A more complicated, but also frequently used LOC variant, is to count only the non-commented and non-blank lines of code (NCLOC/ELOC). Alternatively, one could count only the commented lines (CLOC) or blank lines (BLOC). In Figure 1, we present formal definitions of SLOC and NLOC as decision rules and decision trees.

Although the standard LOC measures are widely used, they are also often criticized as being sensitive to programming style, programming language, or the presence of generated code [10], [11], [12]. The community has also observed problems with inconsistency and lack of transparency in the counting algorithms (and tools) [11], [13], [14]. Finally, the study by Barb et al. [15] questioned the ability of standard LOC measures to proxy for complexity, size, productivity, and effort. Hence, it seems that the rules of counting LOC could be improved if we:

- adapt them to fit a type of task/context (perhaps not the same lines of code should be counted when considering code review, testing, or maintenance effort);
- mine them from a dataset representing the specificity of a given software development environment (e.g., a company or programmers involved in an open-source project and their programming styles).

We propose a generator of task-specific LOC definitions and their counters that meets the above requirements. The core of the generated LOC counter is a binary decision tree (see Fig. 1). Such trees are mined from a given dataset using Genetic Programming (GP).

We perform several studies focusing on three types of software-related tasks: code readability, unit testing, and code reviews to 1) optimize the choice of GP-related parameters of the tool from the perspective of its performance, and 2) validate the tool by assessing the effectiveness of task-specific LOC measures while proxying for the code-readability score, number of assertions in unit tests, and code-review duration. Since our goal is to improve the ability of standard LOC measures to proxy for these measures in particular contexts, we use the standard LOC measures as a baseline in our study.

The outline of the remaining part of this paper is as follows. Section II presents the related work in the areas of LOC-based software measurement. Section IV presents the Genetic Programming algorithm used in this study. Section V describes the method for quantifying lines, i.e., feature extraction. Sections VI–VIII present the results of applying our method to finding which lines should be counted when considering code readability, assertions in test cases, and code-review duration. Section IX presents the answers to our research questions and their interpretation. Finally, Section X presents the conclusions.

II. RELATED WORK

The main use cases of LOC measures are (1) to use them as predictors of development or maintenance effort, (2) as features describing source code for Machine-Learning-based models for code analysis, (3) as a factor "normalizing" other measures, or (4) as a standard against which other measures can be evaluated [9], [16].

Although LOC measures have been in use for many decades, there are only a few variants that have been widely studied. As we have already stated, the standard definitions of LOC measures include counting all source lines of code (SLOC), counting only non-commented and non-blank lines of code (NCLOC or ELOC) [4], [5], [17], [18], counting commented lines of code (CLOC) or blank lines of code (BLOC). Alternatively, one can count the number of executable statements (ES) in the code [19]. It has been shown that the standard LOC measures correlate with other commonly used software measures [9], [18]. However, at the same time, they have been widely criticized [10], [11], [12], e.g., as being unable to measure code written in multiple programming languages or being sensitive to programming styles.

Interestingly, despite the critique of the standard LOC measures as being too generic in some aspects, the studies proposing their variants tailored to specific tasks or contexts are scarce. For instance, Dundas [20] proposed a custom LOC measure that could be used to evaluate code quality by counting so-called simple lines, condition-based lines, looped-based lines, and exception-handling lines. Nguyen et al. [13] discussed how logical and physical lines should be counted for different programming languages. Finally, Jones [21] proposed a list of guidelines for counting LOC for different purposes (e.g., counting reusable code, scaffold code, etc.).

When it comes to the methods and tools allowing to define custom LOC measures, probably the most recognized is the framework proposed by SEI [22], which helps to precisely define the rules for counting lines of code. Recently, Ochodek et al. [23] proposed a tool called Flexible LOC Counter/Classifier (CCFlex) that employs machine-learning algorithms to automatically implement LOC measurement instruments based on the labeled examples of code snippets. This study is the most similar to ours. However, CCFlex solves the problem of inferring the counting rules from the examples of counts being made while we propose a method that automatically derives a LOC-measure definition from the pairs consisting of code fragments and values of some external measure that we want to proxy for with lines of code. Consequently, instead of answering the question of how the lines were counted, our tool answers the question of which lines are worth being counted.

Although standard LOC measures correlate with many code metrics [9], most of the time, one has to collect multiple measures to evaluate the code from the perspective of a single quality characteristic. As a result, it is a common practice to rely on code-metric suites, e.g., REBOOT [24], QMOOD [25], CK metrics [26], or even multiple variants of metrics measuring the same aspects, e.g., code cohesion [27]. Unfortunately, collecting multiple metrics increases the costs of running a measurement program. Therefore, methods like Goal Question Metric (GQM) advise limiting the number of measures to the most relevant ones [28]. Our method for mining task-specific LOC measures could be perceived as a trade-off between the measurement costs (we collect only one measure) and the ability to capture multiple aspects of code (which could be potentially done more accurately with a suite of dedicated measures).

Since we formulate the problem of finding a task-specific LOC measure as a search-based optimization problem, we can classify our research as belonging to Search-Based Software Engineering (SBSE). According to Harman et al. [29], SBSE is "an approach to Software Engineering (SE) in which Search-Based Optimization (SBO) algorithms are used to address problems in SE." They also give numerous examples of SE-related problems addressed by the use of Genetic Programming (GP) in the areas such as software testing, design, requirements, project management, and refactoring. According to Afzal and Torkar [30], GP has been also extensively applied to software engineering predictive modeling, including software fault proneness prediction (e.g., [31]), software cost/effort/size estimation (e.g., [32]), and software fault prediction/software reliability growth modeling (e.g., [33]). However, neither of these studies proposed to use GP to construct a tool for mining task-specific LOC measures.

III. RESEARCH METHOD

We organized our study according to the Design Science Research (DSR) methodology [34], [35]. DSR is a problem-solving research paradigm that focuses on creating and evaluating artifacts as treatments for practical problems. The scope of our study includes two first steps of the DSR-engineering-cycle—designing and validating the treatment, which require answering the following research questions:

- RQ1: Can binary decision trees mined from datasets using Genetic Programming provide better indicators of code readability, unit-testing effort, and code-review duration than the standard LOC measures (i.e. the number of all source lines of code, number of non-commented or non-blank lines of code, etc.)?
- RQ2: How to use Genetic Programming (GP) to generate good quality specialized LOC counters in a reasonable time?
- RQ3: What code-line features are discovered by GP as important attributes of code in the context of the three software-related tasks, i.e. code understanding, testing, and code review?

The first question (RQ1) regards the validation of the treatment. Since we want to find task-specific LOC measures, we assume that such measures would have to be more accurate proxies for task-related dependent variables than the standard LOC measures, i.e., SLOC, NCLOC, BLOC, CLOC. The second research question (RQ2) is a general, design-related question covering all issues related to designing and tuning the treatment—a GP-based tool for searching the space of LOC measure definitions based on the provided data sample. Finally, by answering the last research question (RQ3), we want to investigate whether the choice of source-code line features in the generated task-specific LOC measures is coincidental or could be justified by referring to the existing theories in Software Engineering.

The replication package for this study is publicly available.[1] It includes the datasets, code, and instructions allowing to replicate analyses presented in this paper.

A. TASKS UNDER STUDY

We selected three tasks to validate the proposed treatment, i.e., mining tasks-specific LOC measures to proxy for code-readability score, (2) the number of assertions in unit test cases, and (3) code-review duration.

Each of the studies contributes to answering the research questions RQ1 and RQ3. In addition, we use the study on code readability to investigate how to balance the accuracy and computation costs depending on the GP-related parameters of our tool (RQ2) because the study is based on three independent and relatively small datasets allowing us to experiment with different algorithm settings.

Finally, we decided to focus on a single programming language (Java) to be able to use the same set of features describing lines of code in all the studies and make their results comparable.

B. EVALUATION CRITERIA

We evaluate a LOC measure definition by investigating how strongly it correlates with the external variable it proxies for. In particular, we use Kendall's τ since it can capture

[1] Replication package — https://zenodo.org/record/8331775

non-linear relationships between variables and is robust to outlying observations. It is also preferred [36], [37] over Spearman's ρ, which is another widely use non-parametric correlation coefficient. In order to interpret the strength of the relationship, we estimate Pearson's r based on Kendall's τ using the formula $r = sin(0.5\pi\rho)$ [38] and use the guidelines for interpreting Pearson's r [37] (in the absence of such guidelines for Kendall's τ).

IV. GENCOME

We developed a tool called GENetic COunt-based Measure findEr (GENCOME)[2] that uses Genetic Programming (GP) to search the space of possible count-based measures definitions expressed in the form of binary decision trees. The tool is implemented in Python with the use of the DEAP computational framework [39].

The problem of finding task-specific LOC measures could be defined as a search problem where we search the space of all possible decision trees or rules that could be derived from a set of features used to describe lines of code to find the measure definition that maximizes or minimizes the correlation between the number of lines of code and some variable of choice (e.g., development effort, testing effort). Unfortunately, with the increase in the complexity of decision trees or rules (e.g., the increase of decision-tree height) searching the entire space would quickly become practically infeasible. For instance, for binary trees, the maximum number of nodes n is equal to $2^{(height+1)} - 1$ and when the objects are characterized by k features, we can generate up to k^n complete trees (as permutations with repetitions of n elements taken from the set of k elements).[3] Consequently, even for small trees with a height of 3 and 10 features describing counted objects, there would be up to 10^{15} complete trees to search through (not to mention the incomplete ones).

GENCOME finds definitions of task-specific LOC measures based on a dataset—X and y (see Figure 2). X consists of lines of code characterized by a set of features. Each line has a reference to a unique identifier of the entity it belongs to (e.g., the name of the source file). Entries in y are the pairs of entity identifiers and values of the output variable we want to proxy for (e.g., the number of assertions in test cases, code-readability score, or code-review duration). The tool uses GP to find a definition of LOC measure that maximizes (or minimizes) the correlation coefficient (in the GP terminology called fitness function) for the number of lines of code and the external variable. It is possible to provide multiple pairs of Xs and ys. The algorithm will search for a LOC measure definition that optimizes a multi-objective fitness function being the mean of the correlation coefficients calculated for each of the X, y pairs.

[2]GENCOME — https://github.com/mochodek/gencome.

[3]Please note that this is an upper boundary, and in practice, the number of valid decision trees to investigate would be lower than that (e.g., valid decision trees have only two types of terminal nodes—'true' and 'false'; we could exclude isomorphic trees or trees having nonsense combinations of nodes—e.g., 'true' and 'true' pair of children nodes, etc.).

GENCOME represents LOC definitions as binary decision trees (see Figure 1) with two types of terminal nodes `true` and `false`—a final decision whether to count the line or not. Other types of nodes correspond to the presence or absence of a given feature in the line of code. For instance, a node `'bracket'` is evaluated by checking whether a line contains at least one opening or closing round bracket. A tree-based definition can be transformed into a rule-based definition by joining the nodes (or their negations) on each of the paths from the root node to one of the `true` terminal nodes using the `and` logical operator and adding one more rule `otherwise: false`.

The search process starts by generating an initial population of μ decision trees (called individuals). The tool controls the process of generating the trees by ensuring that all pairs of terminal nodes in the trees are either `true` and `false` or `false` and `true` and that the height of each tree is between the minimum and maximum allowed height (h) requested by the user. In the next step, each individual is evaluated using the fitness function. The following steps are performed g times, and in each iteration, a new generation of individuals is created and evaluated:

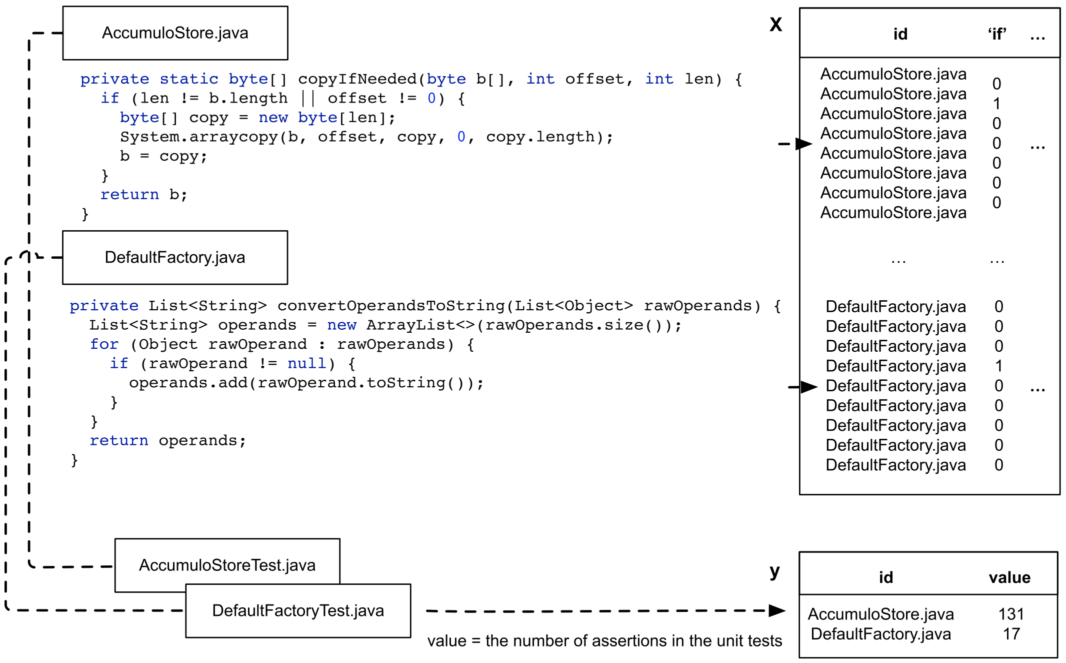
- Selection — the highest rated individuals in the population are selected for reproduction using a tournament algorithm (in each tournament, ts individuals are randomly sampled and the best-fitted one is selected for reproduction).
- Reproduction — the selected individuals produce offspring with the use of crossover and mutation operators.
- Evaluation — the quality of offspring is evaluated by calculating the fitness function for each of the individuals.
- Replacement — individuals from the old population are replaced by the new ones.

The process is controlled by two parameters defining the probability of crossover (*crossProb*) and mutation (*mutProb*) (the user can also independently choose the probability for each of the mutation operators). The crossover operator (see Figure 3 A) allows generating new individuals that inherit parts of the trees from their parents, while mutation is a technique that helps to get out from local optima (see Figure 3 B–E). During each iteration, GENCOME controls the height of the trees so they remain in the range expected by the user.

The choice of parameters μ, g, ts, *crossProb*, and *mutProb* determines how fast the fitness function will converge to the optimum and how likely it will get out of local optima. Although the choice of the parameters is determined by multiple factors [40], a commonly accepted rule of thumb is to set *crossProb* close to 1.0 while keeping *mutProb* close to zero to prevent the search procedure from becoming random. Also, usually, the bigger the size of the initial population (n), the more likely it is to find the optimal solution in a fewer number of generations (g).

V. EXTRACTING LINE FEATURES

GENCOME operates independently of the mechanism used to extract features from lines of code. In this study, we use



```java
private static byte[] copyIfNeeded(byte b[], int offset, int len) {
  if (len != b.length || offset != 0) {
    byte[] copy = new byte[len];
    System.arraycopy(b, offset, copy, 0, copy.length);
    b = copy;
  }
  return b;
}
```

```java
private List<String> convertOperandsToString(List<Object> rawOperands) {
  List<String> operands = new ArrayList<>(rawOperands.size());
  for (Object rawOperand : rawOperands) {
    if (rawOperand != null) {
      operands.add(rawOperand.toString());
    }
  }
  return operands;
}
```

id	'if'	...
AccumuloStore.java	0	
AccumuloStore.java	1	
AccumuloStore.java	0	
AccumuloStore.java	0	...
AccumuloStore.java	0	
AccumuloStore.java	0	
AccumuloStore.java	0	
AccumuloStore.java		
...	...	
DefaultFactory.java	0	
DefaultFactory.java	0	
DefaultFactory.java	0	
DefaultFactory.java	1	
DefaultFactory.java	0	...
DefaultFactory.java	0	
DefaultFactory.java	0	
DefaultFactory.java	0	
DefaultFactory.java	0	

id	value
AccumuloStore.java	131
DefaultFactory.java	17

FIGURE 2. An example of the dataset structure—*X* and *y* files.

an external tool, CCFlex [23], to extract over 100 features[4] from lines of code written in Java. Most of them represent the number of occurrences of a given token or substring in a line (e.g., Java keywords, operators, primitive types, numbers, camel-case names, etc.). We also count the number of characters and tokens and transform them to a set of binary features (e.g., 'up_to_20_chars', 'more_than_80_chars'). We also add some features grouping other token-based features (e.g., 'access_keyword', 'keyword', 'any_bracket', 'bracket', 'loop_header'). For instance, a feature called 'access_keyword' counts the occurrences of private, public, and protected keywords in a line.

VI. TASK #1: CODE READABILITY

We study the task of using LOC as the proxy for the code-readability score. According to [41] "code readability measures the effort of the developer to access the information contained in the code." Poor code readability increases the difficulty of software maintenance tasks [41], [42], [43]. Therefore, code-readability score (the higher the score, the more readable the code) should be negatively correlated with the maintenance effort.

A. DATA COLLECTION

We based this study on three publicly available datasets[5] collected by Buse and Weimer [43] ($D_{b\&w}$), Dorn [44] (D_{dorn}), and Scalabrino et al. [41] (D_{scal}). The datasets include code snippets written in Java for which readability was manually assessed by human annotators. $D_{b\&w}$ is the oldest dataset among them. It consists of 100 Java code snippets (768 SLOC) evaluated by 120 student annotators. In contrast to the remaining two datasets, the code snippets in this dataset are small and range from 4 to 13 lines of code (mean = 7.7, SD = 2.5). The D_{dorn} dataset includes 121 Java code snippets (3,617 SLOC) ranging from 9 to 50 lines of code (mean = 29.9, SD = 16.3). The readability of the snippets was assessed by 5,468 annotators, including 1,800 industrial developers. Finally, D_{scal} is the newest dataset among the three datasets. It consists of 200 Java methods (5,337 SLOC) ranging from 10 to 42 lines of code (mean = 26.7, SD = 10.2). The readability of the code snippets was evaluated by 30 students. In all the datasets, the final code-readability score for each snippet was calculated as the mean score given by the annotators.

B. CODE READABILITY AND STANDARD LOC MEASURES

The correlations between the standard LOC measures and code-readability score for the datasets $D_{b\&w}$, D_{dorn}, and D_{scal} are presented in Table 1. We can see that the code-readability score is negatively correlated with the number of lines of code (SLOC) for D_{dorn} and D_{scal} (τ equal to −0.24 and −0.28, respectively). Even stronger negative correlations ("moderate" to "strong") could be observed for NCLOC ($\tau = -0.36$). Surprisingly, we can observe a "weak" positive correlation between SLOC and code-readability score for the $D_{b\&w}$ dataset, ($\tau = 0.16$) and a "weak" negative correlation for NCLOC ($\tau = -0.13$). We suspect this might be caused by the fact that the code snippets in $D_{b\&w}$ are too small to have a visible impact on the difficulty of reading the code. At the same time, we can observe that the

[4]The complete list of features could be found in the replication package.

[5]The datasets $D_{b\&w}$, D_{scal} and D_{dorn} are available at https://dibt.unimol.it/report/readability.

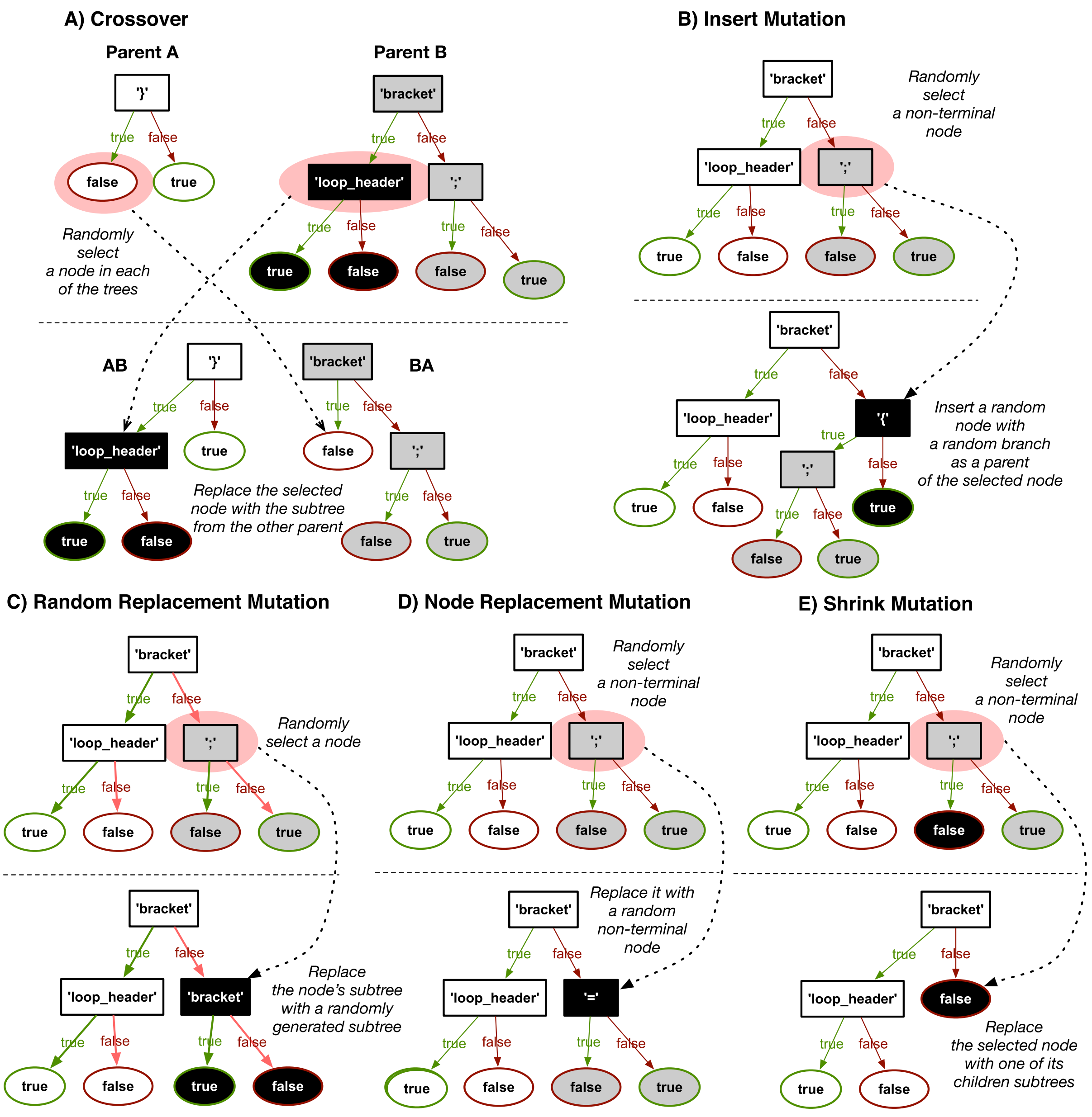

A) Crossover
Parent A
Parent B
Randomly select a node in each of the trees
AB
BA
Replace the selected node with the subtree from the other parent
B) Insert Mutation
Randomly select a non-terminal node
Insert a random node with a random branch as a parent of the selected node
C) Random Replacement Mutation
Randomly select a node
Replace the node's subtree with a randomly generated subtree
D) Node Replacement Mutation
Randomly select a non-terminal node
Replace it with a random non-terminal node
E) Shrink Mutation
Randomly select a non-terminal node
Replace the selected node with one of its children subtrees
'bracket'
'loop_header'
';'
'}'
'{'
'='
true
false

FIGURE 3. Examples of applying the genetic operators available in GENCOME to generate new LOC measure definitions (the outgoing arcs `true`/`false` determine whether the feature represented by the node is greater than zero or not; e.g., 'bracket' $\xrightarrow{true}$ … means that either '(' or ')' appears at least once in the line).

number of commented lines (CLOC) is positively correlated with the readability score for all the datasets (τ equal to 0.25, 0.39, and 0.16). Finally, the number of blank lines of code (BLOC) is negatively correlated (a "weak" correlation) with the code readability score for D_{dorn} and D_{scal} (τ equal to -0.14 and -0.08) and positively correlated with code-readability score ("weak" to "moderate") for $D_{b\&w}$ ($\tau = 0.23$).

C. CODE READABILITY AND TASK-SPECIFIC LOC MEASURES

We ran GENCOME to find LOC-measure definitions for each dataset (an intra-dataset search) and then cross-applied the measures between the datasets (an inter-dataset search). We also searched for the measure definitions using all three

TABLE 1. Correlations between the standard LOC measures and code-readability score.

	$D_{b\&w}$		D_{dorn}		D_{scal}	
Measure	τ	ES	τ	ES	τ	ES
SLOC	0.16*	W	-0.24***	W-M	-0.28***	M-S
NCLOC	-0.13	W	-0.36***	M-S	-0.36***	M-S
CLOC	0.25**	W-M	0.39***	M-S	0.16**	W
BLOC	0.23**	W-M	-0.14*	W	-0.08	N-W

p-value $\leq$ 0.05*, 0.01**, 0.001***.
ES (effect size) — N–negligible, W–weak, M–moderate, S–strong, VS–very strong, P–perfect.

datasets at once and a pair of D_{dorn} and D_{scal} datasets by using multi-objective fitness functions (a multi-dataset search).

Also, as we stated in Section III-A, we used the study on code readability to investigate the impact of GP-related parameters, i.e., the population size (μ), the maximum allowed height of trees (h), and the number of generations (g) on the fitness of the measure definitions found by GENCOME as these parameters can have the biggest impact on processing time. Unfortunately, simultaneously searching the space of all considered GP-related attributes (including *crossProb* and *mutProb*) is not a feasible option. Therefore, we decided to perform a systematic search starting from the typical settings found in the literature, narrowing ranges and fixing the values of some parameters. In particular, we divided the search process into three stages. The aim of the first stage was to extensively search the parameters' space with high crossover probability (*crossProb*) and low mutation probability (*mutProb*), which is a widely used configuration [40]. In the second stage, we narrowed the search space and swapped the values of *mutProb* and *crossProb*. It was motivated by the fact that high mutation probability can help increase the diversity in small populations [40]. In the last stage, we significantly increased the population size (by two orders of magnitude) to investigate its impact on the fitness function. During the study, we controlled the maximum height of generated decision trees (h) by setting it to 1, 2, 3, 4, 5, 6, 7, 8, and 16 to balance the generalizability of the task-specific LOC measures and their ability to proxy for the output variable. Finally, we set the tournament size (ts) to 4 for all simulations.

In the first stage, we performed the analyses for μ = 100, 200, 300, 400 and set *crossProb* and *mutProb* to 0.8 and 0.2, respectively However, since we roll back all mutation operations that produce individuals exceeding the maximum allowed height, the probability of successful mutation is visibly lower than that. We ran the algorithm for each of the configurations for 1000 generations. The correlations between the LOC counted using the best task-specific LOC measures found by GENCOME and code-readability scores are presented in Table 2. For the intra-dataset search, the correlations were "strong" to "very strong" ($\tau = -0.83$ for $D_{b\&w}$, -0.78 for D_{dorn}, and -0.76 for D_{scal}). Also, the observed standard deviation calculated for τ between the runs ranged between 0.01 and 0.03. Therefore, the process of finding LOC measures seem stable. Even for the decision trees of height one (e.g., '`brackets`': `true`, `otherwise: false`), the calculated τ was lower than for the standard LOC measures ($\tau = -0.37$ for $D_{b\&w}$, -0.40 for D_{dorn}, and -0.37 for D_{scal}). For all the inter-dataset searches, the observed correlations could be interpreted as "moderate" to "strong" / "very strong" (τ ranged from -0.56 to -0.39). Finally, the multi-dataset search seems to be a compromise between the intra- and inter-dataset searches.

By analyzing the plot in Figure 4, we can see that increasing the population size allows finding better measure definitions, as long as the processing is performed for a sufficient number of generations. Also, we can see that this effect gets stronger with the increase of the maximum allowed height of decision trees. For instance, for the trees with $h = 4$, the difference in τ between the smallest and biggest populations was up to ca. 0.04 while for $h = 16$, it increased to ca. 0.15. Based on these results, we conclude that setting $g = h \times 50$ could be used as a rule of thumb while configuring GENCOME since it allows to reduce the processing time with little effect on the results.

Figure 5 shows an example of how the choice of *crossProb* and *mutProb* (0.8, 0.2 or 0.2, 0.8) can affect the fitness function τ (comparing the results of stage 1 and stage 2 of the search procedure). When h was very small (1, 2, or 4), there was nearly no difference in τ between both settings (none of the strategies was consistently better). However, for the larger trees of h equal to 8 and 16, the best results were obtained for *crossProb* = 0.2 and *mutProb* = 0.8. One of the explanations for this phenomenon is that when generated trees reach their maximum allowed height, the crossover operation might no longer allow to introduce structural changes into their offspring. Conversely, mutation operators can easily modify the already grown trees (e.g., by shrinking them) which helps to avoid stalling at local optima. Therefore, we recommend using higher *mutProb* for larger tree sizes ($h \geq 8$), while for the smaller trees, we perceive both of the strategies as equally effective.

The results for applying the strategy of using large populations in stage 3 while reducing the number of generations are presented in Figure 6 (μ = 10,000, and g = 50). Although the number of generated trees was approximately the same as for $\mu = 500$ and $g = 1000$, the observed correlations (*tau*) were visibly weaker for almost all the cases. Consequently, we decided to abandon that strategy as being computationally ineffective.

We analyzed the structure of the decision trees and most frequently appearing line features to investigate what characterizes lines that are good proxies for code readability (see Table 3). The feature's frequency of appearing is expressed as the percentage of positive rules (that evaluate to `true`) in which the feature appears.

It seems that the line features that are the most important from the perspective of proxying for code readability are the presence of brackets ('`bracket`'—round brackets). The second important feature is the line length ('`up_to_80_chars`', '`more_than_80_chars`', '`up_to_40_chars`', and '`up_to_60_chars`'). We could also see that equality operators and assignment operators are consistently being used as nodes in the decision trees. According to Buse and Weimer [43] the number of brackets, and line length are very strong predictors of code readability.

VII. TASK #2: ASSERTIONS IN UNIT TESTS

The second task that we study is to use LOC as a proxy for the number of assertions in unit-level test cases (and indirectly as a proxy for testing effort [6], [45], [46]).

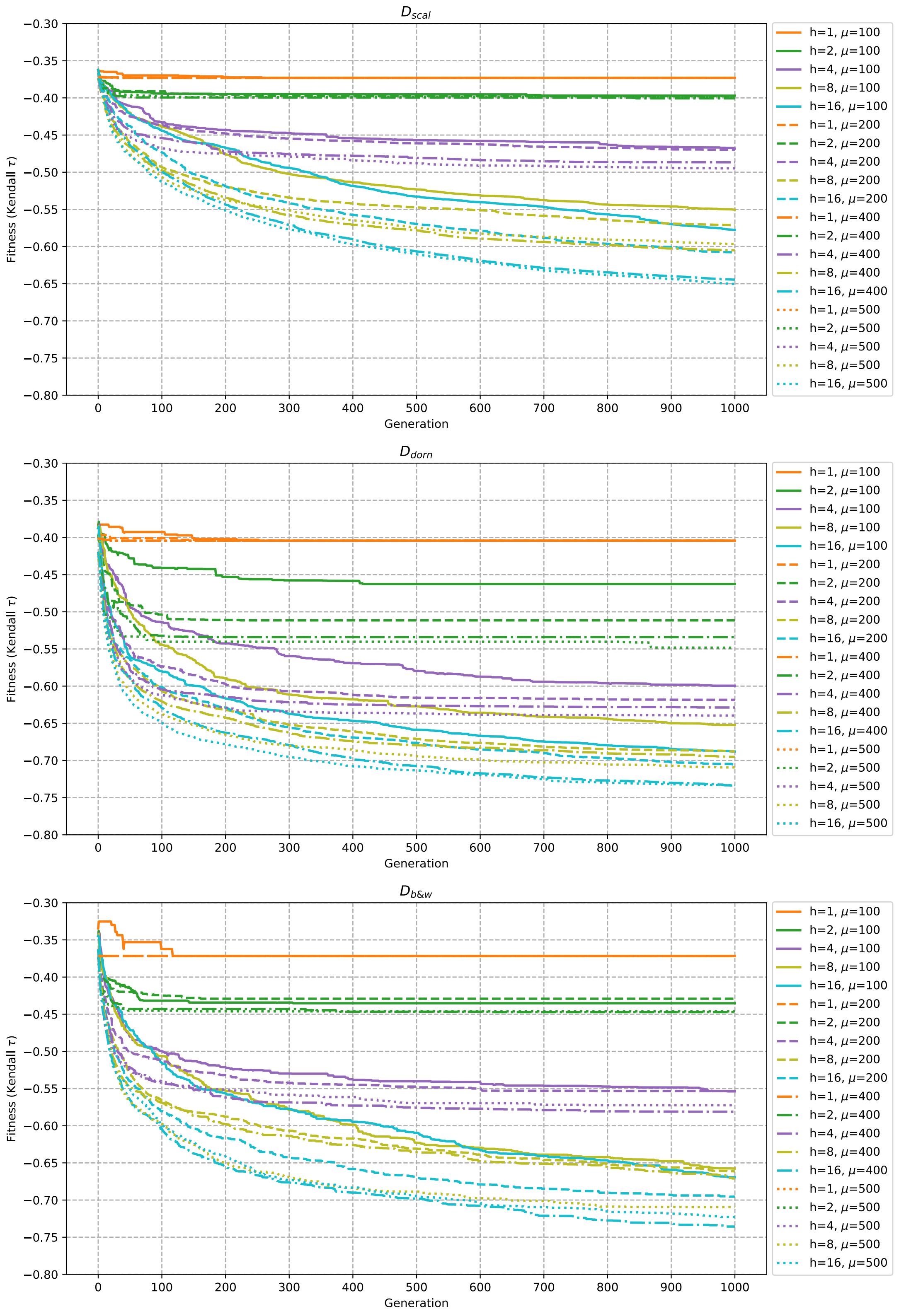


FIGURE 4. The average Fitness (τ) depending on population size (μ), maximum allowed height of decision trees (*h*), and number of generations (*g*) for *crossProb* = 0.8 and *mutProb* = 0.2. To ensure the clarity of presentation, the plots include the subset of the tested parameters μ and *h* (the full plots are available in the replication package).

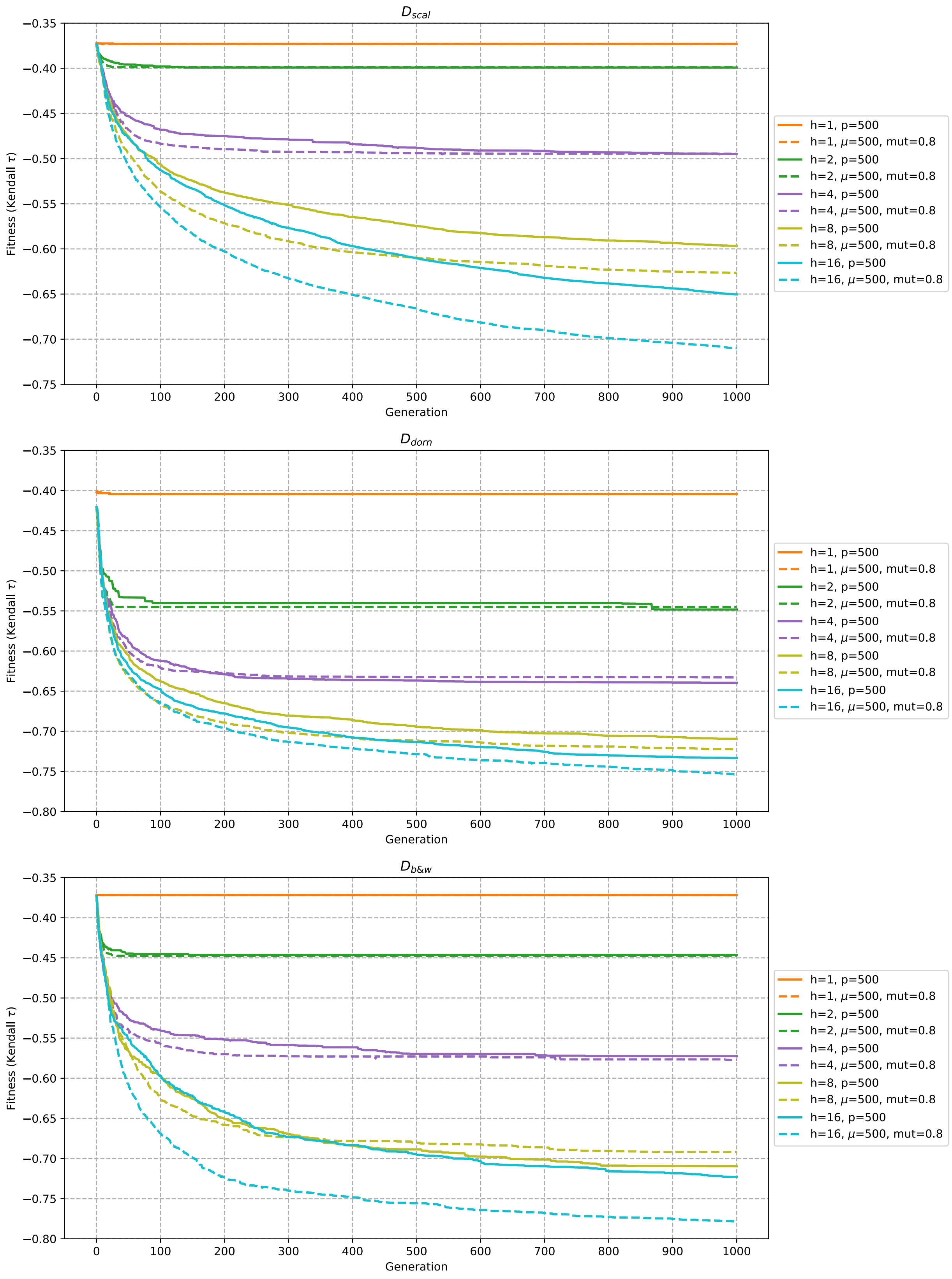


FIGURE 5. Comparison between the average Fitness (τ) for *crossProb* = 0.8, *mutProb* = 0.2 and *crossProb* = 0.2, *mutProb* = 0.8.

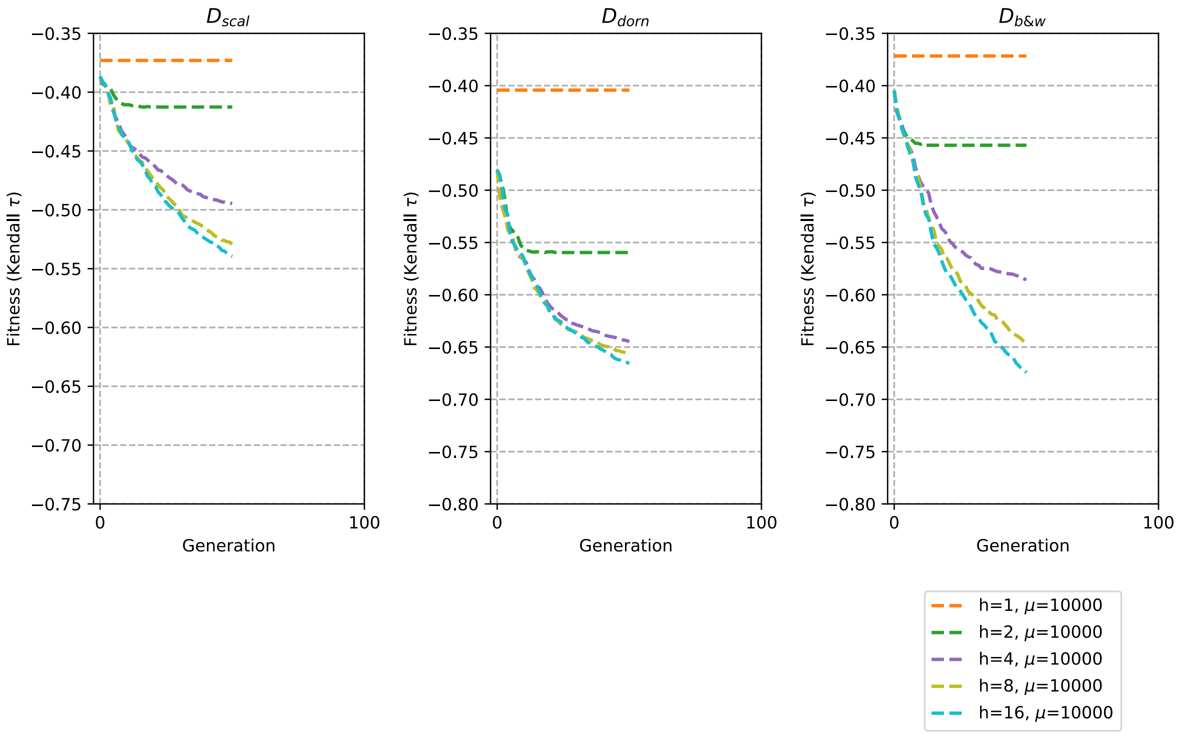


FIGURE 6. The average Fitness (τ) for the strategy of using large populations while reducing the number of generations (μ = 10,000, and g = 50).

TABLE 2. Correlations between the best task-specific LOC measures and code-readability score.

Applied to →	$D_{b\&w}$			D_{dorn}			D_{scal}		
Derived from ↓	τ_{min}/τ_{avg}	ES	h	τ_{min}/τ_{avg}	ES	h	τ_{min}/τ_{avg}	ES	h
$D_{b\&w}$	-0.83***	S-VS	16	-0.56***	M-VS	2	-0.39***	M-S	16
	-0.78 (std=0.03)			-0.49 (std=0.05)			-0.37 (std=0.01)		
D_{dorn}	-0.45***	M-S	4	-0.78***	S-VS	16	-0.39***	M-S	16
	-0.43 (std=0.02)			-0.76 (std=0.01)			-0.37 (std=0.01)		
D_{scal}	-0.39***	M-S	7	-0.45***	M-S	8	-0.76***	S-VS	16
	-0.36 (std=0.02)			-0.40 (std=0.02)			-0.71 (std=0.03)		
D_{scal}, D_{dorn},	-0.79***	S-VS	16	-0.67***	S-VS	16	-0.50***	M-VS	16
$D_{b\&w}$	-0.77 (std=0.02)			-0.61 (std=0.05)			-0.42 (std=0.03)		
D_{scal}, D_{dorn}	-0.46***	M-S	3	-0.79***	S-VS	16	-0.52***	M-VS	16
	-0.43 (std=0.02)			-0.75 (std=0.02)			-0.48 (std=0.03)		

p-value ≤ 0.05*, 0.01**, 0.001***.

ES (effect size) — N–negligible, W–weak, M–moderate, S–strong, VS–very strong, P–perfect.

A. DATA COLLECTION

We collected code and unit tests from a large sample of repositories hosted on GitHub (D_{unit}). First, We took the list of 1.85M repository URLs collected by Munaiah et al. [47]. Then, we used the Reaper [47] and PHANTOM [48] tools to curate the list for engineered software projects (we took only those repositories for which both tools agreed) and selected repositories labeled as containing projects implemented in Java (36K GitHub URLs). In the following step, we downloaded the code from the repositories and extracted the pairs of .java files containing the code and unit tests for that code. When performing this step, we limited the search to the projects following the Maven convention for structuring Java projects and using JUnit 4 or 5 to implement unit tests (9,390 GitHub repositories). We matched the code and tests files by assuming that the tests for a class implemented in src/main/java/A.java are available in src/test/java/ATest.java or src/test/java/ATests.java. We assumed that a testing class should consist of at least two test cases (e.g., at least a pair of positive and negative test cases), therefore, we rejected the repositories for which the mean number of testing methods per testing class was lower than two. As a result, we obtained data from 6,991 GitHub repositories (more than 143K pairs of code and test files). Since such a big dataset would make the evaluation of individuals run too long, we downsampled the dataset by randomly selecting 2000 file pairs (1.2M SLOC of code in the source files).

B. UNIT TESTS AND STANDARD LOC MEASURES

The correlation coefficients calculated for the standard LOC measures and the number of assertions in unit tests are presented in Table 4. We observed ''moderate'' to ''strong''

TABLE 3. Top 10 most frequently appearing line features in the positive rules (`...: true`) of the best task-specific LOC-measure definitions for code readability.

Dataset	Features
All	'bracket' (43%), ')' (35%), 'up_to_80_chars' (28%), 'more_than_80_chars' (28%), 'String' (24%), '>' (24%), '(' (23%), 'up_to_40_chars' (22%), '=' (21%), 'up_to_60_chars' (20%).
$D_{b\&w}$	'up_to_80_chars' (40%), '>' (36%), ',' (30%), 'bracket' (27%), ')' (27%), 'up_to_60_chars' (25%), 'class' (21%), '?' (20%), ';' (19%), 'up_to_40_chars' (18%).
D_{dorn}	'bracket' (56%), ')' (45%), '(' (34%), 'more_than_80_chars' (29%), 'String' (28%), '.' (26%), 'up_to_80_chars' (25%), 'block_code' (22%), 'type_def_keyword' (19%), '>' (18%).
D_{scal}	'bracket' (40%), 'more_than_80_chars' (33%), ')' (30%), '=' (28%), 'up_to_40_chars' (27%), '"' (26%), 'up_to_80_chars' (25%), 'String' (24%), '>' (22%), '?' (21%).

TABLE 4. Correlations between the standard / task specific LOC measures and number of assertions in unit tests.

Measure	τ	ES
SLOC	0.28***	M-S
NCLOC	0.27***	M-S
CLOC	0.22***	W-M
BLOC	0.27***	M-S
h=1, μ=200, g=50	0.31*** (τ_{avg}=0.31, std=0.00)	M-S
h=2, μ=200, g=100	0.32*** (τ_{avg}=0.33, std=0.00)	M-S
h=3, μ=200, g=150	0.33*** (τ_{avg}=0.33, std=0.00)	M-S
h=4, μ=200, g=200	0.34*** (τ_{avg}=0.33, std=0.00)	M-S
h=5, μ=200, g=250	0.34*** (τ_{avg}=0.33, std=0.00)	M-S

p-value $\leq$ 0.05*, 0.01**, 0.001***.
ES (effect size) — N–negligible, W–weak, M–moderate, S–strong, VS–very strong, P–perfect.

correlations for SLOC, NCLOC, and BLOC ($\tau = 0.28$, 0.27, and 0.27, respectively). The correlation between CLOC and the number of assertions was slightly weaker ("weak" to "moderate") than for the remaining standard LOC measures ($\tau = 0.22$).

C. UNIT TESTS AND TASK-SPECIFIC LOC MEASURES

The dataset D_{unit} is bigger by two orders of magnitude than all the datasets we used for the study on code readability, therefore, we decided to set the population size (μ) to 200 and performed the analyses for h between 1 and 5. Each time, the number of generations (g) was set to $h \times 50$.

The correlation between the task-specific LOC measures and the number of assertions in unit tests are presented in Table 4. Similarly to the standard LOC measures, we observed "moderate" to "strong" correlations, however, the calculated correlations coefficients were higher than the ones calculated for the standard LOC measures by ca. 11% to 21% (τ ranged from 0.31 to 0.34).

When analyzing the structure of the decision trees, we observed that the definitions of the task-specific measures were similar to the code complexity measures that are known to be used to determine the number of test cases

TABLE 5. Top 10 most frequently appearing line features in the positive rules (`...: true`) of the best task-specific LOC-measure definitions for the number of assertions in unit tests.

Features
'return' (60%), 'relational_operator' (27%), 'condition_check' (25%), 'case' (18%), 'if' (17%), '"' (11%), ']' (9%), 'assignment_operator' (8%), '.' (8%), 'up_to_20_chars' (7%), '{' (7%), '>' (7%), '(' (7%), 'whole_line_comment' (6%), 'else' (6%), '*' (6%), 'up_to_60_chars' (6%), 'camel_case' (6%), '#' (6%), 'square_bracket' (6%).

needed to achieve a given branch or path test coverage, such as McCabe's Cyclomatic Complexity (CC) [49] or SonarQube's Complexity.[6] For instance, one of such LOC definitions stated that a line is counted if it is not commented (`'comment'`) and contains a `'relational_operator'` or if it contains a `'return'` keyword. As it is presented in Table 5, the list of most frequently appearing line features contains other features that are related to paths in the code, i.e., `'condition_check'` (grouping `if`, `case`, `else`, `switch`), `'curly_brackets'` (determines the boundaries of code blocks), and `'.'` or `'camel_case'` (a reference to a field or a method call).

VIII. TASK #3: CODE-REVIEW DURATION

We study the possibility of using LOC as a proxy for code-review duration. In particular, we focus on counting modified lines of code (code churn) since they are a better predictor of code-review duration than the total size of the code being reviewed [7], [50], [51].

A. DATA COLLECTION

We collected data on code reviews from three mature Eclipse projects, i.e., Eclipse JDT (D_{jdt}), Platform (D_{plat}), and Papyrus (D_{pap}). We fetched the data concerning code reviews and the source code under review from the Gerrit and Git repositories owned by the Eclipse Foundation.[7] We collected data concerning 2,375 code reviews conducted between 2014-06-19 and 2020-08-05. In the next step, We filtered out the reviews that included files different than Java source files to preserve consistency between the studies and prevent deriving LOC measure definitions mixing features of different types of files. Then, we ran CCFlex to extract line features from the source files but once the features were extracted, we selected only the lines that were modified. The resulting datasets included data from 890 reviews (320 reviews and 125.6K SLOC in D_{jdt}; 163 reviews and 69.3K SLOC in D_{plat}; and 407 reviews and 268.7K SLOC in D_{pap}).

B. CODE REVIEWS AND STANDARD LOC MEASURES

The correlations between the standard LOC measures and code-review duration are summarized in Table 6. None of these correlations could be characterized as stronger than "weak" (τ ranged from −0.14 to 0.16). We also observed

[6]https://docs.sonarqube.org/latest/user-guide/metric-definitions
[7]Gerrit—https://git.eclipse.org/r, Git—https://git.eclipse.org/c/.

TABLE 6. Correlation between the standard LOC measures and code-review duration.

Measure	D_{jdt}		D_{plat}		D_{pap}	
	τ	ES	τ	ES	τ	ES
SLOC	0.16***	W	0.04	None	0.12***	N-W
NCLOC	0.16***	W	0.08	N-W	0.11**	N-W
CLOC	0.07	N-W	-0.14*	W	0.11***	N-W
BLOC	0.10*	N-W	-0.05	None	0.10**	N-W

p-value $\leq$ 0.05*, 0.01**, 0.001***.
ES (effect size) — N–negligible, W–weak, M–moderate, S–strong, VS–very strong, P–perfect

that the approach to reviews might differ between the considered projects since we observed a "weak" negative correlation between NCLOC and code-review duration for one of the projects (Eclipse JDT) which suggests that in this project reviews last shorter if the code is commented.

C. CODE REVIEWS AND TASK-SPECIFIC LOC MEASURES

Since the size of D_{jdt}, D_{plat}, and D_{pap} is bigger than the datasets regarding code readability by one order of magnitude, we decided to reduce the number of GENCOME runs by fixing the population size (μ) to 200, selecting h between 1 and 6, and setting the number of generations (g) to $h \times 50$.

We performed intra- and inter-dataset searches for task-specific LOC measure definitions. As it can be seen in Table 7, the correlations for the measures found in the intra-dataset searches were visibly stronger than for the standard LOC measures (τ between 0.31 and 0.37). However, for the inter-dataset searchers, the correlations were similar to those observed for the standard LOC measures.

The lists of most frequently appearing features presented in Table 8 includes many line features that were important for more than one project, e.g, features regarding the length of lines (e.g., 'up_to_80_chars', 'more_than_16_words'), code blocks ('{' or 'block_code'), comments ('block_comment'), comparison operators ('<' or ':'), or the presence of string literal in the line ('"'). We can justify the importance of these features by referring to the studies on the types of defects discovered in code reviews. According to Mäntylä and Lassenius [52], the most frequently found "Visual Representation Defects" regard the usage of brackets or line length, also the second most frequent issue related "Documentation–Textual Defects" concerns problems with comments, and finally, the most frequent among "Logic Defects" are issues related to comparison operations. Therefore, it seems that the features used in the LOC definitions derived by GENCOME characterize lines of code that might be defective, and consequently, could increase the duration of code reviews [7], [51].

IX. DISCUSSION

A. SUMMARY OF FINDINGS

RQ1: Can binary decision trees mined from datasets using Genetic Programming provide better indicators of code readability, unit-testing effort, and code-review duration than the standard LOC measures? The described generator of LOC counters was able to find task-specific LOC measures that correlated stronger with the considered external variables than any of the standard LOC measures for each of the considered tasks. We even found the superiority of the simplest definitions of task-specific LOC measure derived from binary decision trees of height equal to one.

At the same time, we observed that there is a trade-off between the portability and specificity of the task-specific LOC measures. We even found the tipping point of the depth of the decision trees—at the height of 4 or above, the task-specific measures performed better for the same project, but worse across different projects. This is rather natural as task-specific measures are expected to be task-specific.

Therefore, we recommend finding new measures using GENCOME rather than searching for universal measures for all kinds of projects. In the case of the reuse of measures, we recommend inspecting the definitions found by GENCOME to verify whether the counting rules they propose could be justified by experience or theories in Software Engineering.

RQ2: How to use Genetic Programming to generate good quality specialized LOC counters in a reasonable time? We designed and implemented a GP-based tool called GENCOME that is capable of finding definitions of count-based measures based on the provided data sample by optimizing the correlation between the measure and an external variable of choice.

We used the study on code readability to explore the performance of GENCOME and the quality of its results. We found that the most important parameter is the maximum allowed height of decision trees (h). The height of the tree impacts the computational cost of evaluating the individuals and the maximum complexity of the rules they can encode. The choice of h should be made depending on the representativeness of the population sample we use to derive the measures and the granularity of the features. For the less representative samples, smaller trees would be preferred. Also, using a grouping of elementary features can help achieve good results for smaller decision trees (e.g., 'access_keyword' = 'private' or 'public' or 'protected').

Two other parameters that are important from the perspective of computation costs are the population size (μ) and the number of generations (g). We observed that the population size should not be smaller than the number of features used to describe the lines of code because if a feature is not present in the initial population then the only way it could be introduced to the trees in the population is by the result of the mutation operation. Based on the results of all the studies, we recommend starting with the population size of 200 individuals as a sensible value balancing the computational costs and quality of the results. For the number of generations, we propose to set it depending on the maximum allowed tree height ($g = h \times 50$). We used two strategies for setting the crossover probability (*crossProb*) and mutation probability (*mutProb*) (0.8, 0.2 and

TABLE 7. Correlation between the best task-specific LOC measures and code-review duration.

Applied to →	D_{jdt}			D_{plat}			D_{pap}		
Derived from ↓	τ_{max}/τ_{avg}	ES	h	τ_{max}/τ_{avg}	ES	h	τ_{max}/τ_{avg}	ES	h
D_{jdt}	0.37*** 0.33 (std=0.02)	M-S	6	0.12* 0.08 (std=0.02)	N-W	5	0.13*** 0.11 (std=0.01)	W	4
D_{plat}	0.20*** 0.14 (std=0.03)	W-M	6	0.36*** 0.30 (std=0.03)	M-S	6	0.12** 0.07 (std=0.03)	N-W	3
D_{pap}	0.18*** 0.15 (std=0.01)	W	6	0.11 0.05 (std=0.03)	N-W	4	0.31*** 0.28 (std=0.02)	M-S	6

p-value ≤ 0.05*, 0.01**, 0.001***.

ES (effect size) — N–negligible, W–weak, M–moderate, S–strong, VS–very strong, P–perfect.

TABLE 8. Top 10 most frequently appearing line features in the positive rules (`...: true`) of the best task-specific LOC-measure definitions for code-review duration.

Dataset	Features
All	’"’ (25%), ’<’ (20%), ’{’ (18%), ’more_than_16_words’ (15%), ’block_comment’ (13%), ’continue’ (13%), ’up_to_8_words’ (13%), ’up_to_80_chars’ (12%), ’up_to_12_words’ (12%), ’:’ (10%).
D_{jdt}	’"’ (31%), ’{’ (24%), ’continue’ (17%), ’curly_bracket’ (16%), ’more_than_16_words’ (16%), ’<’ (16%), ’block_comment’ (12%), ’]’ (12%), ’*’ (11%), ’/’ (11%).
D_{plat}	’"’ (18%), ’{’ (16%), ’<’ (16%), ’up_to_8_words’ (15%), ’continue’ (14%), ’for’ (13%), ’:’ (13%), ’up_to_80_chars’ (12%), ’up_to_60_chars’ (12%), ’block_comment’ (12%).
D_{pap}	’<’ (28%), ’"’ (27%), ’more_than_16_words’ (19%), ’up_to_12_words’ (18%), ’{’ (16%), ’up_to_8_words’ (15%), ’block_comment’ (15%), ’up_to_80_chars’ (14%), ’:’ (14%), ’block_code’ (11%).

0.2, 0.8), None of them seemed superior for the smaller trees (*h* up to 4), however, the strategy for choosing higher *mutProb* seems to be a better choice for the larger trees ($h \geq 5$). Finally, we can recommend setting the tournament size (*ts*) to 4, however, this recommendation is not supported by a systematic simulation study on optimizing the choice of this parameter.

The computational costs also depend on the size of the dataset (X, y) since each individual is used to classify all the lines in the X file to calculate the fitness.

In this study, we mined the measures using commodity hardware. For small datasets and low values of the aforementioned parameters, the execution time was up to a few minutes, while for the largest datasets and the most extreme values of the parameters, it raised up to a few days at maximum. Thus, performance-wise, usage of GENCOME in industrial settings seems feasible.

RQ3: What code-line features are discovered by GP as important attributes of code in the context of the three software-related tasks, i.e. code understanding, testing, and code review? The quality of the output of GENCOME was compared to the most frequently used line features for all three considered tasks found in the body of knowledge. As a result, we learned that it is worth counting the lines containing brackets or being too long as they correlate negatively with the code readability [43]. For the problem of using LOC as a proxy for the number of assertions in unit tests, the rules derived by GENCOME could be considered as empirical approximations of measures such as McCabe's Cyclomatic Complexity as they focus on counting the lines containing conditional expressions, method calls, or return statements. Finally, for using lines of code as an indicator of code-review duration, the most frequently used features could be mapped to some of the most commonly raised issues in code reviews [52].

B. OTHER GENCOME USE CASES

LOC measures are one of the most frequently used source code measures with numerous use cases [53]. Their reported use cases include software fault and defect prediction (e.g., [54], [55]), software quality estimation (e.g., [56]), code smells detection (e.g., [57]), dead code prediction (e.g., [58]), code churn estimation (e.g., [59]), software testing (e.g., [60]), software refactoring (e.g., [61]), software maintainability (e.g., [62]), predicting software build outcomes (e.g., [63]), software reliability (e.g., [64]), software security (e.g., [65]). In nearly all of these use cases, LOC measures are used as predictors or proxies for some other variables. Numerous studies on prediction in Software Engineering reported that prediction models trained on context-specific datasets outperform those trained on generic datasets [66], [67]. Therefore, since GENCOME can find task/context-specific variants of LOC measures that show stronger correlations with external variables than their standard counterparts, they could be treated as replacements for the state-of-the-art LOC measures for all these applications, as long as historical data are available.

C. THREATS TO VALIDITY

We discuss the threats to validity for this study based on the general guidelines provided by Wohlin et al. [68] and the framework dedicated to Search-Based Software Engineering (SBSE) proposed by Barros and Neto [69].

1) CONCLUSION VALIDITY

Genetic Programming depends on stochastic random-number generators. To mitigate this threat, we performed 10 runs of each study and observed consistent results between the runs.

2) CONSTRUCT VALIDITY

We used the correlation coefficient as a fitness function since our goal was to find a LOC-measure definition that will be the best proxy for a given external variable. Therefore, this choice seems to be justified, however, we cannot prove that this is the best function to be used for this purpose. Also, there are no single rules to evaluate the effect size of correlations. We mitigate this problem by using the guidelines for interpreting correlation coefficients coming from three different research areas [37].

We used an external tool called CCFlex to extract features from lines of code and used them to count the standard LOC measures. To mitigate the threat related to miscounting the lines, we verified the way the tool counts the standard LOC measures on a random sample of code.

Also, in the study on unit tests, we matched the pairs of source code and testing code based on the names and locations of the .java files. Although this approach should not lead to false positives because we rejected files that did not contain tests, we could falsely reject some of the projects that did not follow this naming convention. Nevertheless, we downsampled the dataset to 2000 file pairs, therefore, even if we missed some pairs in the full dataset the impact of this threat on the results of our study would be negligible. Similarly, we measured the duration of code reviews based on the data reported in the Gerrit instances. The measurement could be biased by factors such as time-zone differences since the open-source community working on Eclipse is globally distributed. Nevertheless, the impact of this threat is also negligible since we aim at discovering context-specific LOC measures.

Finally, although GENCOME does not focus on extracting features describing lines of code, the availability of the features relevant to a given task is extremely important. We based the set of features on the syntax of Java programming language and the physical features of lines (e.g., line length). We treated most of them as binary variables (e.g., the presence of a keyword in the line), however, we could use different strategies for discretizing the features, which could yield better results.

3) INTERNAL VALIDITY

According to Barros and Neto [69], the main SBSE-related threats in this category regard the problems with the reproducibility of the research or lack of real problem instances. Since we discussed the choice of the parameters and made the source code publicly available the former threat should not materialize in our study. Also, since we based our study on the data from real software development projects, also the latter threat seems not relevant to our study.

4) EXTERNAL VALIDITY

We applied GENCOME to derive task-specific LOC measures for three effort-related tasks. We used the datasets containing from 768 SLOC to 1.2M SLOC. Therefore, the variability in the size of the datasets seems sufficiently large to mitigate the threats related to the ''lack of evaluations for instances of growing size and complexity'' [69]. There is also a threat related to the representativeness of the source code in the datasets. Apart from the datasets related to code readability, all other datasets contain code directly from software projects. In particular, the dataset on unit tests was composed by mining 36K GitHub repositories. However, since we studied only three different tasks in the context of open-source projects, we have to accept the fact that the proposed treatment might not give comparable results when applied to other tasks or when it is used in other contexts.

X. CONCLUSION

In this paper, we introduced GENCOME—a tool that generates task-specific LOC-measure definitions and counters by mining historical data with Genetic Programming. The generated LOC counters have a form of binary decision trees with maximum height (h) controlled by the user. Based on the conducted studies, we proposed to set the default value of the population-size parameter (g) to $h \times 50$ to balance the trade-off between the performance and effectiveness of the tool.

We performed three studies to validate the tool and the effectiveness of the generated task-specific LOC measures in proxying for the code-readability score, number of assertions in unit tests, and code-review duration. The inferred task-specific LOC measures showed a ''strong'' to ''very strong'' negative correlation with code-readability score (Kendall's τ ranging from -0.83 to -0.76) compared to ''weak'' to ''strong'' negative correlation for the best among the standard LOC measures (τ ranging from -0.36 to -0.13). For the problem of proxying for the number of assertions in unit tests, correlation coefficients were also higher for task-specific LOC measures by ca. 11% to 21% (τ ranged from 0.31 to 0.34). Finally, task-specific LOC measures showed a stronger correlation with code-review duration than the best among the standard LOC measures ($\tau = 0.31$, 0.36, and 0.37 compared to 0.11, 0.08, 0.16, respectively). In all of these studies, task-specific LOC measures showed stronger correlations with the proxied variables than the state-of-the-art LOC measures (SLOC, NCLOC, CLOC, and BLOC).

Also, we observed that the choice of code-line features in the generated decision trees can be justified by referring to the theories in a given research area. For example, task-specific LOC measures proxying for the number of assertions in unit tests seemed to be empirical approximations of McCabe's Cyclomatic Complexity.

In all of these studies, task-specific LOC measures showed stronger correlations with the proxied variables than the state-of-the-art LOC measures (SLOC, NCLOC, CLOC, and BLOC). Therefore, they could be considered as their replacements. The definitions of task-specific LOC measures are inferred directly from the context-specific historical data which mitigates some of the well-known weaknesses of the standard LOC measures. Also, the GENCOME-generated definitions of LOC measures are executable and can be instantly used as measurement instruments. Unfortunately, there are some limitations to the proposed approach. We observed that there is a trade-off between the portability and specificity of the task-specific LOC measures. We even found the tipping point of the depth of the decision trees—at the height of 4 or above, the task-specific measures performed better for the same project, but worse across different projects. This is rather natural as task-specific measures are expected to be task-specific.

As future research, one could consider using GENCOME to search for new count-based measures in other areas. For instance, a good candidate metric would be to count the number of specific types of tasks or defects in projects that would strongly correlate with development effort.

REFERENCES

[1] J. Wen, S. Li, Z. Lin, Y. Hu, and C. Huang, "Systematic literature review of machine learning based software development effort estimation models," *Inf. Softw. Technol.*, vol. 54, no. 1, pp. 41–59, Jan. 2012.

[2] A. Kaur and K. Kaur, "Systematic literature review of mobile application development and testing effort estimation," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 34, no. 2, pp. 1–15, Feb. 2022.

[3] M. S. Unluturk and K. Kurtel, "Quantifying productivity of individual software programmers: Practical approach," *Comput. Informat.*, vol. 34, no. 4, pp. 959–972, 2016.

[4] 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.

[5] T. M. Khoshgoftaar and J. C. Munson, "The lines of code metric as a predictor of program faults: A critical analysis," in *Proc. 14th Annu. Int. Comput. Softw. Appl. Conf.*, 1990, pp. 408–409.

[6] M. Badri and F. Toure, "Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes," *J. Softw. Eng. Appl.*, vol. 5, no. 7, pp. 513–526, 2012.

[7] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on GitHub," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 367–371.

[8] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions," *J. Softw., Evol. Process*, vol. 28, no. 7, pp. 589–618, Jul. 2016.

[9] Y. Gil and G. Lalouche, "On the correlation between size and metric validity," *Empirical Softw. Eng.*, vol. 22, no. 5, pp. 2585–2611, Oct. 2017.

[10] C. Jones, "Software metrics: Good, bad and missing," *Computer*, vol. 27, no. 9, pp. 98–100, Sep. 1994.

[11] J. Schofield, "The statistically unreliable nature of lines of code," *CrossTalk*, vol. 18, no. 4, pp. 29–33, 2005.

[12] P. P. Texel, "Measuring software development status: Do we really know where we are?" in *Proc. SoutheastCon*, Apr. 2015, pp. 1–6.

[13] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm, "A SLOC counting standard," in *Proc. COCOMO II Forum*, 2007 pp. 1–16.

[14] M. Staron, D. Durisic, and R. Rana, "Improving measurement certainty by using calibration to find systematic measurement error—A case of lines-of-code measure," in *Software Engineering: Challenges and Solutions*. Cham, Switzerland: Springer, 2017, pp. 119–132.

[15] A. S. Barb, C. J. Neill, R. S. Sangwan, and M. J. Piovoso, "A statistical study of the relevance of lines of code measures in software projects," *Innov. Syst. Softw. Eng.*, vol. 10, no. 4, pp. 243–260, Dec. 2014.

[16] J. Rosenberg, "Some misconceptions about lines of code," in *Proc. 4th Int. Softw. Metrics Symp.*, 1997, pp. 137–142.

[17] E. Morozoff, "Using a line of code metric to understand software rework," *IEEE Softw.*, vol. 27, no. 1, pp. 72–77, Jan. 2010.

[18] M. A. A. Mamun, C. Berger, and J. Hansson, "Effects of measurements on correlations of software code metrics," *Empirical Softw. Eng.*, vol. 24, no. 4, pp. 2764–2818, Aug. 2019.

[19] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*. Boca Raton, FL, USA: CRC Press, 2014.

[20] J. B. Dundas, "Understanding code patterns analysis, interpretation and measurement," in *Proc. Int. Conf. Comput. Electr. Eng.*, Dec. 2008, pp. 150–154.

[21] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*. New York, NY, USA: McGraw-Hill, 2008.

[22] R. E. Park, "Software size measurement: A framework for counting source statements," Carnegie-Mellon Univ. Softw. Eng. Inst., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-92-TR-020, 1992.

[23] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron, "Recognizing lines of code violating company-specific coding guidelines using machine learning," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 220–265, Jan. 2020.

[24] G. Sindre, R. Conradi, and E.-A. Karlsson, "The REBOOT approach to software reuse," *J. Syst. Softw.*, vol. 30, no. 3, pp. 201–212, Sep. 1995.

[25] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[26] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Mar. 1994.

[27] E. N. H. Kırğıl and T. E. Ayyıldız, "Predicting software cohesion metrics with machine learning techniques," *Appl. Sci.*, vol. 13, no. 6, p. 3722, Mar. 2023.

[28] P. Berander and P. Jönsson, "A goal question metric based approach for efficient measurement framework definition," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, Sep. 2006, pp. 316–325.

[29] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–61, 2012.

[30] W. Afzal and R. Torkar, "On the application of genetic programming for software engineering predictive modeling: A systematic review," *Expert Syst. Appl.*, vol. 38, no. 9, pp. 11984–11997, Sep. 2011.

[31] Y. Liu and T. Khoshgoftaar, "Reducing overfitting in genetic programming models for software quality classification," in *Proc. 8th IEEE Int. Symp. High Assurance Syst. Eng.*, Feb. 2004, pp. 56–65.

[32] E. N. Regolin, G. A. de Souza, A. R. T. Pozo, and S. R. Vergilio, "Exploring machine learning techniques for software size estimation," in *Proc. 23rd Int. Conf. Chilean Comput. Sci. Soc.*, 2003, pp. 130–136.

[33] E. O. Costa, G. A. de Souza, A. T. R. Pozo, and S. R. Vergilio, "Exploring genetic programming and boosting techniques to model software reliability," *IEEE Trans. Rel.*, vol. 56, no. 3, pp. 422–434, Sep. 2007.

[34] R. H. Von Alan, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quart.*, vol. 28, no. 1, pp. 75–105, 2004.

[35] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Cham, Switzerland: Springer, 2014.

[36] D. C. Howell, *Statistical Methods for Psychology*. 8th ed. Wadsworth, OH, USA: Cengage Learning, 2012.

[37] H. Akoglu, "User's guide to correlation coefficients," *Turkish J. Emergency Med.*, vol. 18, no. 3, pp. 91–93, Sep. 2018.

[38] D. A. Walker, "JMASM9: Converting Kendall's tau for correlational or meta-analytic analyses," *J. Modern Appl. Stat. Methods*, vol. 2, no. 2, p. 26, 2003.

[39] F.-A. Fortin, F.-M. De Rainville, M.-A. G. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Lang. Res.*, vol. 13, pp. 2171–2175, Jul. 2012.

[40] A. Hassanat, K. Almohammadi, E. Alkafaween, E. Abunawas, A. Hammouri, and V. B. S. Prasath, "Choosing mutation and crossover ratios for genetic algorithms—A review with a new dynamic approach," *Information*, vol. 10, no. 12, p. 390, Dec. 2019.

[41] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk, "A comprehensive model for code readability," *J. Softw., Evol. Process*, vol. 30, no. 6, p. e1958, Jun. 2018.

[42] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, "An integrated measure of software maintainability," in *Proc. Annu. Rel. Maintainability Symp.*, 2002, pp. 235–241.

[43] R. P. L. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 546–558, Jul. 2010.

[44] J. Dorn, "A general software readability model," M.S. thesis, Dept. Comput. Sci., Univ. Virginia, Charlottesville, VA, USA, 2012. [Online]. Available: https://web.eecs.umich.edu/~weimerw/students/dorn-mcs-paper.pdf

[45] F. Toure, M. Badri, and L. Lamontagne, "Predicting different levels of the unit testing effort of classes using source code metrics: A multiple case study on open-source software," *Innov. Syst. Softw. Eng.*, vol. 14, no. 1, pp. 15–46, Mar. 2018.

[46] M. Badri, F. Toure, and L. Lamontagne, "Predicting unit testing effort levels of classes: An exploratory study based on multinomial logistic regression modeling," *Proc. Comput. Sci.*, vol. 62, pp. 529–538, 2015.

[47] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Softw. Eng.*, vol. 22, no. 6, pp. 3219–3253, Dec. 2017.

[48] P. Pickerill, H. J. Jungen, M. Ochodek, M. Maćkowiak, and M. Staron, "PHANTOM: Curating GitHub for engineered software projects using time-series clustering," *Empirical Softw. Eng.*, vol. 25, no. 4, pp. 2897–2929, Jul. 2020.

[49] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[50] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proc. 11th Work. Conf. Mining Softw. Repositories*, May 2014, pp. 202–211.

[51] M. Staron, W. Meding, O. Söder, and M. Bäck, "Measurement and impact factors of speed of reviews and integration in continuous software engineering," *Found. Comput. Decis. Sci.*, vol. 43, no. 4, pp. 281–303, Dec. 2018.

[52] M. V. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 430–448, May 2009.

[53] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *J. Syst. Softw.*, vol. 128, pp. 164–197, Jun. 2017.

[54] A. Abuasad and I. M. Alsmadi, "Evaluating the correlation between software defect and design coupling metrics," in *Proc. Int. Conf. Comput., Inf. Telecommun. Syst. (CITS)*, May 2012, pp. 1–5.

[55] J. Al Dallal, "Transitive-based object-oriented lack-of-cohesion metric," *Proc. Comput. Sci.*, vol. 3, pp. 1581–1587, Jan. 2011.

[56] A. Abuasad and I. M. Alsmadi, "The correlation between source code analysis change recommendations and software metrics," in *Proc. 3rd Int. Conf. Inf. Commun. Syst.*, Apr. 2012, pp. 1–5.

[57] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proc. 11th Int. Conf. Predictive Models Data Anal. Softw. Eng.*, Oct. 2015, pp. 1–10.

[58] R. Francese, S. Murad, I. Passero, and G. Tortora, "Metric pictures: The approach and applications," in *Proc. Int. Conf. Comput. Eng. Syst.*, Nov. 2010, pp. 320–325.

[59] S. Karus and M. Dumas, "Code churn estimation using organisational and code metrics: An experimental comparison," *Inf. Softw. Technol.*, vol. 54, no. 2, pp. 203–211, Feb. 2012.

[60] S. Almugrin and A. Melton, "Estimation of responsibility metrics to determine package maintainability and testability," in *Proc. 2nd Int. Conf. Trustworthy Syst. Their Appl.*, Jul. 2015, pp. 100–109.

[61] J. A. Dallal, "Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics," *Inf. Softw. Technol.*, vol. 54, no. 10, pp. 1125–1141, Oct. 2012.

[62] H. A. Al-Jamimi and M. Ahmed, "Prediction of software maintainability using fuzzy logic," in *Proc. IEEE Int. Conf. Comput. Sci. Autom. Eng.*, Jun. 2012, pp. 702–705.

[63] J. Finlay, A. Connor, and R. Pears, "Mining software metrics from jazz," in *Proc. 9th Int. Conf. Softw. Eng. Res., Manage. Appl.*, Aug. 2011, pp. 39–45.

[64] S. K. Dubey and A. Mishra, "Fuzzy qualitative evaluation of reliability of object oriented software system," in *Proc. Int. Conf. Adv. Eng. Technol. Res.*, Aug. 2014, pp. 1–6.

[65] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Comput. Fraud Secur.*, vol. 2013, no. 5, pp. 8–17, May 2013.

[66] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, "Local vs. global models for effort estimation and defect prediction," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2011, pp. 343–351.

[67] N. Bettenburg, M. Nagappan, and A. E. Hassan, "Think locally, act globally: Improving defect and effort prediction models," in *Proc. 9th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, Jun. 2012, pp. 60–69.

[68] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Cham, Switzerland: Springer, 2012.

[69] M. de Oliveira Barros and A. C. Dias-Neto, "Threats to validity in search-based software engineering empirical studies," *Relatórios Técnicos DIA/UNIRIO*, vol. 5, no. 1, pp. 1–10, 2011.

MIROSLAW OCHODEK (Member, IEEE) received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the Poznan University of Technology, Poland, in 2004, 2006, and 2011, respectively, and the Dr.habil. degree in information and communication technology, in Poland, in 2021.

Since 2010, he has been an Assistant Professor with the Faculty of Computing and Telecommunications, Poznan University of Technology. In 2018, he was also employed as a Senior Lecturer at the Gothenburg University of Technology. His research interests include measurement and predictions, requirements engineering, and empirical software engineering.

KRZYSZTOF DURCZAK received the B.S. degree in computing from the Faculty of Computing, Poznan University of Technology, Poland, in 2019, and the M.S. degree in computing from the Poznan University of Technology, Poland, in 2020.

Since 2019, he has been employed as a Mobile Software Developer at the Polish company JakDojade, specializing in developing software for the Android operating system.

JERZY NAWROCKI received the M.Sc., Ph.D., and Habilitation degrees in computer science from the Poznan University of Technology, Poznan, Poland, in 1980, 1984, and 1994, respectively.

Since 1980, he has been with the Poznan University of Technology. In 1986/1987, he spent a year as a Visiting Researcher at the University of Nijmegen, Nijmegen, The Netherlands. In the past, he served, amongst others, as the Dean of the Faculty of Computing Science, Poznan University of Technology, where he is currently a Professor and the Vice Director of the Institute of Computing Science.

Prof. Nawrocki was an active member of various national and international committees. In 2021, he was conferred the title of Professor by the President of Poland. He was a recipient of the IFIP Outstanding Service Award. Currently, he is the Vice Chair of the Committee of Informatics of the Polish Academy of Sciences, a Councilor of the International Federation for Information Processing (IFIP), and a member of IFIP Working Group 2.4 on Software Implementation Technology.

MIROSLAW STARON is currently a Professor with the Department of Computer Science and Engineering, University of Gothenburg, Sweden. He has published extensively on software metrics, model-driven software development, and empirical software engineering, and cooperates with Ericsson, Volvo, and other telecom companies and car manufacturers. He has written two books about automotive software development and about measurement.