Inspecting Code Churns to Prioritize Test Cases

Francesco Altiero, Anna Corazza, Sergio Di Martino(✉), Adriano Peron, and Luigi Libero Lucio Starace

University of Naples Federico II, Naples, Italy
{francesco.altiero,anna.corazza,sergio.dimartino,adriano.peron2,
luigiliberolucio.starace}@unina.it

Abstract. Within the context of software evolution, due to time-to-market pressure, it is not uncommon that a company has not enough time and/or resources to re-execute the whole test suite on the new software version, to check for non-regression. To face this issue, many Regression Test Prioritization techniques have been proposed, aimed at ranking test cases in a way that tests more likely to expose faults have higher priority. Some of these techniques exploit code churn metrics, i.e. some quantification of code changes between two subsequent versions of a software artifact, which have been proven to be effective indicators of defect-prone components. In this paper, we first present three new Regression Test Prioritization strategies, based on a novel code churn metric, that we empirically assessed on an open source software system. Results highlighted that the proposal is promising, but that it might be further improved by a more detailed analysis on the nature of the changes introduced between two subsequent code versions. To this aim, in this paper we also sketch a more refined approach we are currently investigating, that quantifies changes in a code base at a finer grained level. Intuitively, we seek to prioritize tests that stress more fault-prone changes (e.g., structural changes in the control flow), w.r.t. those that are less likely to introduce errors (e.g., the renaming of a variable). To do so, we propose the exploitation of the Abstract Syntax Tree (AST) representation of source code, and to quantify differences between ASTs by means of specifically designed Tree Kernel functions, a type of similarity measure for tree-based data structures, which have shown to be very effective in other domains, thanks to their customizability.

Keywords: Regression testing · Test prioritization · Code churn.

1 Introduction

Within the software maintenance, changes in the source code can introduce bugs and faults in the software, not only in the new features but also in already validated functionalities. In literature, this phenomenon is called *Software Regression*. *Regression Testing* is a set of activities aimed at providing confidence that

Published by Springer Nature Switzerland AG 2020
V. Casola et al. (Eds.): ICTSS 2020, LNCS 12543, pp. 272–285, 2020.
https://doi.org/10.1007/978-3-030-64881-7_17

(I) the changed parts of the software behave as intended, and (II) the unchanged parts have not been adversely affected by modifications [1,13]. Although extensively used in industry, regression testing is challenging from both a process management and a resource management perspective, being extremely time consuming. This is especially true with novel software development methodologies, such as *Continuous Integration/Continuous Delivery*, where the rate of generation of new releases is very high [10]. As a consequence, many research efforts have been dedicated to propose different approaches for regression testing, with the general goal to reduce the amount of required testing efforts, while at the same time keeping a high confidence on the quality of the software, as reported in a recent survey on these techniques [19]. In the literature, the key strategies to reduce regression testing costs are:

1. Regression test selection - selecting subset of existing test cases to run on the modified software (e.g., [14,24]);
2. Regression test suite minimization - reducing the test suite size to a minimal subset, to maintain the same level of coverage as the original test suite;
3. Regression test suite prioritization - finding an ideal order of test cases according to some criteria, such that test cases with higher priority are executed earlier than ones with lower priority [18].

In this paper, we focus on the third scenario, by proposing three test suite prioritization strategies, able to meaningfully combine traditional code coverage metrics with the concept of *code churn*, a metric used in software engineering to measure the amount of code changes taking place within a software unit over the time [23]. We investigate the use of code churns for testing, since many studies have found that parts of software with higher churn exhibit also higher defect density (e.g.: [23,32]), but, to the best of our knowledge, this metric has never been significantly exploited to prioritize regression tests. We then evaluated the effectiveness of the proposed test prioritization strategies against a widely adopted strategy as a baseline, i.e. the total coverage prioritization, on an open source software system, showing promising results.

Moreover, we also sketch a novel approach which analyses the nature of the code churn, that we believe could further improve the performance of the proposed strategies. In detail, in presence of code modifications, we aim at giving higher priority to tests that stress the more changes being potentially more fault-prone (e.g., structural changes in the control flow), w.r.t. those that are less likely to introduce errors (e.g., the renaming of a variable). To do so, we propose the exploitation of the Abstract Syntax Tree (AST) representation of source code, and to quantify differences between ASTs by means of specifically designed *Tree Kernel* functions, a type of similarity measure for tree-based data structures, which turned out to be very effective in other ICT/software engineering domains (e.g. [3]), thanks to their customizability.

The paper is organized as follows. In Sect. 2 we present an overview of the state-of-the-art in test prioritization, whereas in Sect. 3 we formalize the proposed prioritization strategies. In Sect. 4, we report on the empirical study we conducted to assess the effectiveness of the proposed strategies, and discuss the

results of this experiment. In Sect. 5, we sketch the new churn quantification approach we are currently investigating. At last, in Sect. 6, we draw our conclusions.

2 Related Works

With recent software development processes, like Continuous Deployment, the software codebase is updated very often, and these changes should be readily deployed to the customers. In this scenario, regression testing is a critical issue, as the re-execution of the whole test suite, for each new revision, may be too costly/time consuming. A common solution to this problem involves the use of *regression test prioritization* approaches [15,28], which aim at permuting the test suite of a software system, with the goal to give higher priority to tests with higher chance to find faults. In this scenario, in presence of time constraints, a project manager can choose to re-execute only the n most relevant test cases. A lot of research efforts have been spent in this direction (e.g., see survey [35]). Research efforts have been also devoted to defining metrics to quantify and compare the rates of fault detection of test suites [7,8]. Another class of related papers deals with prioritization techniques that are driven by requirements with higher priority, or operate in the presence of time constraints [20,33]. The application of Genetic Algorithms to determine the most effective test order has also been leveraged in different research studies, as in [16,31].

Coverage-based test case prioritization techniques are among the most widely studied approaches for regression test prioritization, as stated in [19]. These techniques aim to rank test cases according to the amount of coverage on source code they provide, considering either block coverage, decision (branch) coverage or statement coverage. A comparison of search algorithms for coverage-based regression test prioritization has been performed in [21]. A case study of several coverage-based regression test prioritization techniques on a real-world complex industrial system which includes real regression faults has been presented in [5].

Two main strategies are employed in coverage-based regression test prioritization, namely the *total* strategy and *additional* strategy [11]. *Total* strategy ranks test cases according to how much they contribute to increment the overall coverage, while *additional* strategy considers the increment of coverage supplied by a test case only on source code which was not covered by the execution of any prior test case. [12] presented a NP-hard *optimal* strategy to maximize the *average percentage of branch covered* metric (see [21]) as intermediate goal, showing that it performs worse than *additional* strategy to the ultimate goal of detecting faults.

There were previous works which exploited the combination of code coverage analysis along with change impact analysis. For example, [17] experimentally applied a procedure-level coverage regression test based on change-based test selection methods to *WebKit*,[1] an open source web browser engine project.

[1] https://webkit.org/.

Approaches for particular types of applications (such as for software product lines [30]) or testing strategies (e.g., model-based testing [25]) have also been introduced, as well as the use of techniques from different application domains (e.g., information retrieval ones [29]). These approaches have been employed, for example, to address coverage profiling overhead (in terms of time and space) and potential problems associated with the imprecision of static program analysis.

Several studies investigated methods to improve regression testing in Continuous Integration (CI) development environments, as analyzed in [26], which detected history-based regression test prioritization techniques as the mainly adopted approaches in CI environments. In particular, the work in [10] has introduced two regression testing techniques (for testing selection and prioritization, respectively) which use readily available test suite execution history data to determine what tests are worth executing and executing with higher priority.

3 The Proposed Prioritization Strategies

In this section, we introduce the necessary notation and concepts, and then formalize the churn-based prioritization strategy we propose.

3.1 Preliminary Definitions

In the test case prioritization problem, given a test suite $\mathcal{TS}$ over a current software version V, the goal is to find an ordering of the tests in $\mathcal{TS}$ that maximizes the regression fault-revealing capability over time of the tests on the next version of the software V' [12]. In the prioritization framework we formalize in this section, we assume that a reference *structural code unit* (e.g., statements, branches, methods), with respect to which coverage and churn metrics are evaluated, has been selected.

Given two subsequent versions V and V', the code churn between them captures the information about the amount of structural code units that were altered between the two versions. More formally, we encode the code churn between V and V' using two functions: $changed_V$, and $deleted_V$. These functions assign to each structured code unit in V a boolean value with the following semantics. For each structured code unit s in V, $changed_V(s)$ (resp., $deleted_V(s)$) is true iff s is changed (resp., deleted) in the next version V', and false otherwise.

Moreover, given a test case $t \in \mathcal{TS}$, we define $CovTest_V(t)$ as the set of structural code units that are covered by the execution of t.

To represent the coverage contribution of a test case t in a way that also takes into account churn information, we introduce the concept of *churn coverage* as follows.

For a test case t, the corresponding churn coverage $ChurnCov_V(t)$ is the triple $\langle c, d, u \rangle$, where:

- c is the number of structural code units covered by the test case t which are changed in the next version. More formally,

$$c = |\{s \in CovTest_V(t) \mid changed_V(s)\}|;$$

– d counts the number of structural code units, covered by the test case t, which have been deleted in the next version. According to our formalization,

$$d = |\{s \in CovTest_V(V) \mid deleted_V(s)\}|;$$

– u is the number of structural code units that are covered by t and remain unchanged in the next version, i.e. $u = |CovTest_V(V)| - (c + d)$.

3.2 The Proposed Ranking Strategies

By introducing suitable ordering criteria for tests in a way that takes into account churn coverage information, it is possible to define different churn-based prioritization strategies.

In what follows, we start by re-defining the total coverage prioritization strategy, which we will use as a baseline for our experiments, and then we propose three definitions of the $\preceq$ ordering relation leveraging churn coverage increments information, to instantiate different prioritization strategies.

Baseline Strategy: Total Coverage
As for the baseline strategy, we consider the *total coverage prioritization*, which is based on the definition of *total structural code unit coverage prioritization*, provided by [27]. This strategy takes into account the total coverage provided by a test case and ranks tests decreasingly according to this measure. Thus, it relies only on coverage information and does not consider churns at all.

The ordering $\preceq_{Tot}$ which realizes this strategy is defined as follows. Given two tests t and t', with $ChurnCov_V(t) = \langle c, d, u\rangle$ and $ChurnCov(t') = \langle c', d', u'\rangle$, it holds that

$$t \preceq_{Tot} t' \text{ iff } c + d + u \leq c' + d' + u'.$$

Strategy 1: Prioritize Churn
The *prioritize churn* strategy prioritizes tests based on their coverage of changed and deleted structural code units. This strategy assigns a higher priority to test cases which cover little outside of the code units which have been altered (i.e. changed or deleted), and is derived from the *specific strategy* defined in [17]. The ordering $\preceq_{Churn}$ which implements the *prioritize changed* strategy is defined as follows. Given two tests t and t', with $ChurnCov(t) = \langle c, d, u\rangle$ and $ChurnCov(t') = \langle c', d', u'\rangle$, it holds that

$$t \preceq_{Churn} t' \text{ iff } \frac{d + c}{d + c + u} \leq \frac{d' + c'}{d' + c' + u'}.$$

Strategy 2: Prioritize Unchanged
In a symmetric manner w.r.t. the *prioritize churn* strategy, the *prioritize unchanged* strategy aims at prioritizing tests covering more unchanged structural code units. This strategy is inspired by the *General strategy* defined in [17]. Intuitively, this ordering ranks test cases according to the ratio of unchanged statements a test case covers on the total number of its covered statements.

Using churn coverage information to formalize the principles of this strategy, we define the ordering $\preceq_{Unch}$ as follows. Given two tests t and t', with $ChurnCov_V(t) = \langle c, d, u \rangle$ and $ChurnCov_V(t) = \langle c', d', u' \rangle$, it holds that

$$t \preceq_{Unch} t' \text{ iff } \frac{u}{c+d+u} \leq \frac{u'}{c'+d'+u'}.$$

Strategy 3: Combined Approach
In the third approach, we propose a *combined* strategy, which selects first test cases covering more changed parts of the product, and then test cases guaranteeing the highest coverage of the unchanged parts.

Intuitively, this strategy first considers tests covering at least one changed structural code unit, ranking them according to the number of covered changed units, and then focuses on the remaining tests, ranking them according to their overall coverage.

More formally, given two tests t and t', with $ChurnCov_V(t) = \langle c, d, u \rangle$ and $ChurnCov_V(t') = \langle c', d', u' \rangle$ the ordering $\preceq_{Comb}$ realizing this strategy is defined as follows:

$$t \preceq_{Comb} t' \text{ iff one of the following is satisfied:}$$

1. $(c' + d') - (c + d) > 0$;
2. $\mid (c + d) - (c' + d') \mid = 0$ and $u \leq u'$.

4 Empirical Evaluation of the Proposed Strategies

In this section we present the design of the empirical study we performed to assess the effectiveness of the proposed strategies to prioritize test cases, and then discuss the results of our evaluation.

4.1 Experimental Protocol

The goal of our investigation is to assess the effectiveness of the proposed prioritization strategies (see Sect. 3), using the standard total code coverage approach as a baseline. To this aim, we first realized a toolchain implementing the three proposed strategies (plus the baseline), evaluating churns at method level. An high-level architectural representation of this solution is shown in Fig. 1.

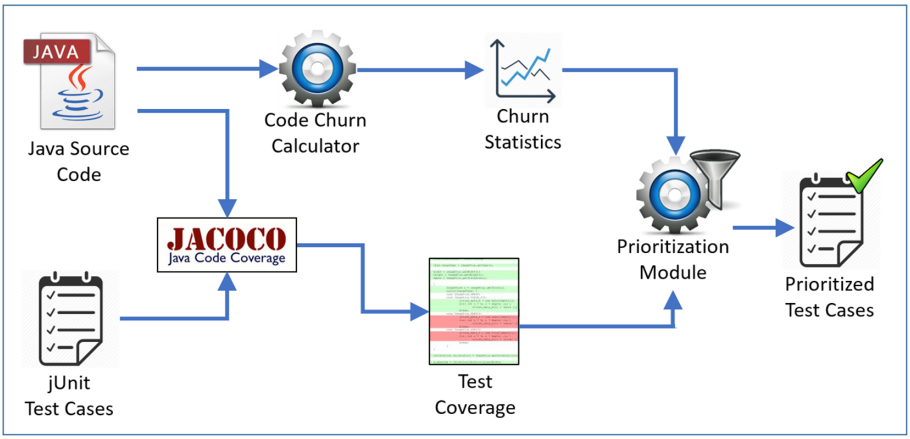

Fig. 1. Architecture of the developed solution to prioritize test cases.

The solution computes, for the various versions of a software product, statistics on code coverage of the test suites and code churn. In particular, test coverage information is obtained by JaCoCo, a widely used open source library for measuring and reporting Java code coverage.[2] Churn statistics at method level are computed by a tool we specifically developed, based on the metrics calculated by the SonarQube tool.[3] Then, a Prioritization Module takes in input the metrics obtained by Jacoco and the Code Churn Calculator, to rank test cases. We developed multiple versions of the Prioritization module, in order to implement the various strategies described in the previous section.

As *Object* of the experiments, we used a Java project often employed in software engineering empirical studies, namely *Siena* (Scalable Internet Event Notification Architecture). It is a scalable publish/subscribe event notification middleware for distributed applications [2], and is also available within the SIR repository [6], which contains software-related artifacts meant to support rigorous controlled software engineering experiments. We considered eight subsequent versions of *Siena*, the latest of which included 26 classes corresponding to 6035 lines-of-code, and 567 test cases. In Table 1, we detail, for each of the considered versions, the total number of lines-of-code, and the coverage percentage achieved by the whole test suite. Moreover, in Table 2 we report code churn information between the subsequent versions we considered.

As for the *experimental protocol*, we used our solution to prioritize test cases in the Siena test suite, for each pair of consecutive versions, and measured the relative *coverage profit* for each of the test cases. More formally, given a prioritized test suite $\langle t_1, \ldots, t_n \rangle$, and an index $i \in [1, \ldots, n]$, we define the relative coverage profit up to the i-th test in the prioritized test suite as follows:

[2] The JaCoCo tool can be obtained freely at http://www.eclemma.org/jacoco/.
[3] The SonarQube tool can be obtained freely at https://www.sonarqube.org/.

Table 1. Size and coverage statistics for the considered versions of Siena.

Version	Total lines of code	Covered lines of code
V_0	11384	46%
V_1	11343	30%
V_2	11349	29%
V_3	11423	29%
V_4	11471	46%
V_5	11471	47%
V_6	11426	46%

Table 2. Churn metrics for Siena.

Versions	Changed methods	Deleted methods	Unchanged methods
$V_0 \rightarrow V_1$	2	9	185
$V_1 \rightarrow V_2$	1	0	186
$V_2 \rightarrow V_3$	3	0	240
$V_3 \rightarrow V_4$	1	0	252
$V_4 \rightarrow V_5$	3	0	251
$V_5 \rightarrow V_6$	1	1	252
$V_6 \rightarrow V_7$	9	4	241

$$CovProfit(i) = \frac{\left|\bigcup_{j=1}^{i} CovTest(t_j)\right|}{\left|\bigcup_{j=1}^{n} CovTest(t_j)\right|},$$

where $CovTest(t)$ represents the set of bytecode-level instructions covered by the execution of the test case t.

4.2 Results and Discussion

The results of our experiment show that the combination of code coverage and churn metrics to prioritize test cases is leading to interesting results, especially when used between two software versions with consistent churns. In particular, for versions V_6 and V_7 of Siena, which exhibit the higher churn, with 9 changed methods and 4 deleted ones (see Table 2), the churn-based strategies remarkably outperform the baseline, as indicated by the decisively more rapidly-growing relative profits curves we report in Fig. 2. In particular, the two strategies that emphasize churn coverage, namely *prioritize churn* and *combined* perform sensibly better than the others. For software versions with smaller churns, on the other hand, the differences were less remarkable, with the churn-based strategies only slightly outperforming the baseline.

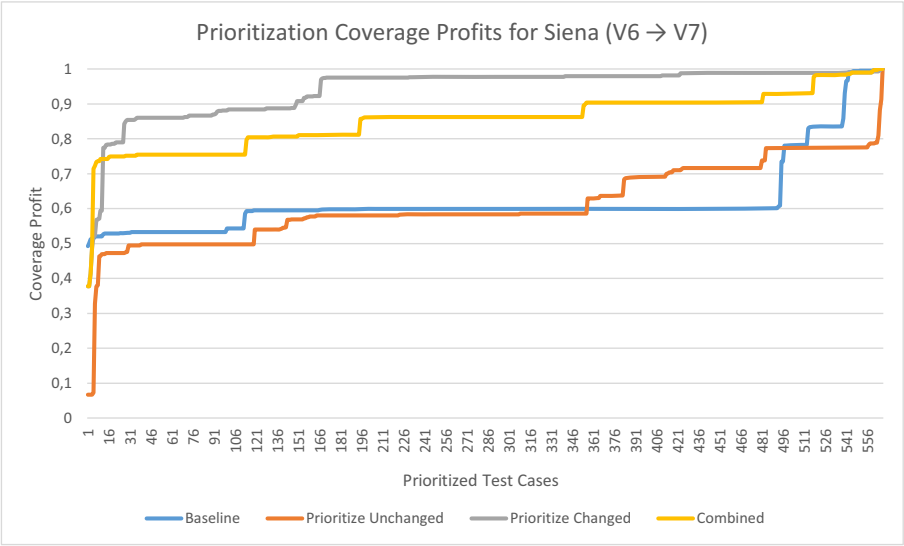

Fig. 2. Coverage profits of the considered strategies for Siena versions V_6 to V_7.

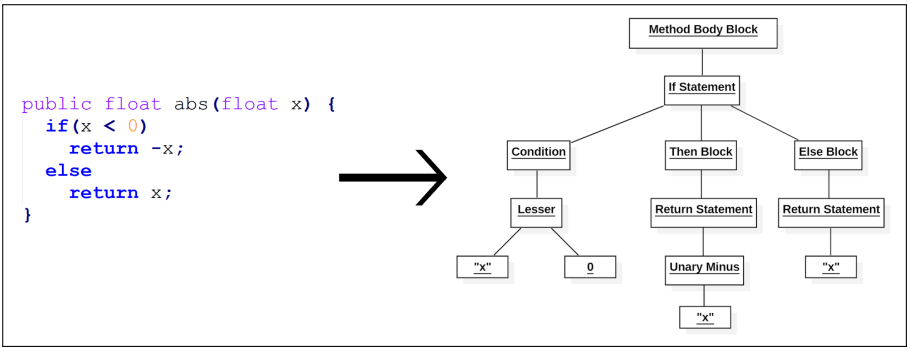

Fig. 3. A Java method and an example of the *Abstract Syntax Tree* of its body block.

Even though our churn-based strategies have proved themselves to be promising, we believe that they could be further improved by taking into account not only the fact that a given structural unit of code changed, but also the nature of said change. Indeed, it is reasonable to assume that not all the changes in source code have the same probability of introducing new faults. For example, one could argue that a refactoring operation, like a simple renaming of a local variable in a method, is less likely to introduce new faults, whereas significant control flow changes such as the update of a stopping condition in an iteration construct are more likely to introduce faults. To capture this intuition, in the next section we sketch a more refined approach we are currently investigating, that quantifies changes in a code-base at a finer grained level, discriminating

between more and less significant changes. Intuitively, this new approach could allow us to prioritize tests that stress more fault-prone changes, w.r.t. those that are less likely to introduce errors.

5 A Novel Strategy to Measure Code Churns

The code churn definition we proposed in Sect. 3 quantifies the amount of changed, unchanged and deleted code between versions, at method granularity level. A more refined strategy for regression test prioritization could also take into account the *nature* of code changes, ranking test cases according to their coverage on altered code which is more likely to introduce faults.

To quantify and weight the amount of changes between two versions of a software project, we propose the evaluation of similarity between the *Abstract Syntax Trees* (*AST*s) representation of their source code, using *Tree Kernel* functions.

*AST*s are a structured representation of source code, widely employed in many scenarios, like for example in compilers. They have also been widely and profitably used in software engineering, e.g., for *code clone detection* [3,34]. An *AST* is a tree-based representation of source code, whose inner nodes represent constructs of the programming language and leaf nodes are related to *tokens* (i.e., variable and method names, literals) appearing in a source code fragment. The topological relations between nodes establish the context of statements and expressions. A *label* is assigned to each node: inner nodes are labeled with information about the particular type of construct they model, while leaf nodes are labeled with the sequence of characters of their token. In Fig. 3, an example of an *AST* of a source code fragment in the Java language is depicted.

Tree Kernel functions are a particular family of *kernel* functions which specifically evaluate similarity between two tree objects. They have been extensively studied in tasks of *Machine Learning* and *Natural Language Processing* [22]. *Tree Kernels* assess the similarity of two tree structures taking into account both the topological information and the labels of their respective nodes, and are typically highly customizable according to a set of specific parameters, allowing them to be tailored to meet different needs in the application domain. A prior approach to evaluate similarity of source code using *AST* and *Tree Kernels* can be found in [3], where a similar technique was applied to *code clone detection*, with profitable results.

Similarity between *AST*s of source code can be used to obtain information about the amount and magnitude of changes in two subsequent versions of a software. These information can be included into code churn in order to support the ranking of test cases for regression test prioritization. To this purpose, we plan to model source code using a suitably defined *AST* representation, and to evaluate similarity between two structured code units of subsequent software versions by comparing the corresponding pair of *AST*s, by means of a specifically-designed *Tree Kernel* function, which can be normalized to produce a similarity score ξ in the range $[0, 1]$. Diversity could be evaluated as well by subtracting this similarity score from 1, i.e., $1 - \xi$.

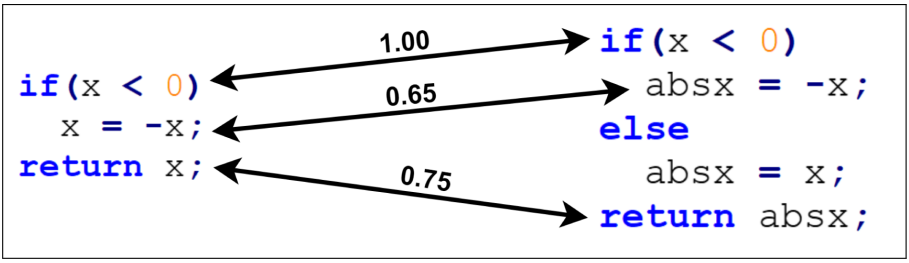

Fig. 4. An example of similarity scores evaluated by a *Tree Kernel* function at statement granularity level.

Figure 4 shows an example of normalized scores provided by a *Tree Kernel* for corresponding statements in two versions of a source code fragment.

To include similarity measures in the code churn, we extend the notation defined in Sect. 3. In particular, we characterize a code churn w.r.t. two subsequent versions V and V' not only by means of the $changed_V$ and the $deleted_V$ functions, but also with a new diversity function $diversity_V$. This function assigns to each structured code unit in V a diversity score in the range $[0, 1]$. In particular, deleted units are evaluated to 1, since there is no corresponding unit in the next version to which they can be compared. Similarly, unchanged units are evaluated to 0, as the diversity clearly is minimal in this case. In the other cases, i.e., when s changes in the next version, the score reflects the magnitude of the change.

With this new function in place, it is possible to re-define the churn coverage of a given test t, namely $ChurnCov_V(t) = \langle c, d, u \rangle$, in a way that takes into account the diversity score information. In the new churn coverage object, d and u are computed as described in Sect. 3, while c can be taken as the sum of the diversity scores in the code units covered by the test case. More formally,

$$c = \sum_{s \in Ch_V(t)} diversity_V(s),$$

with $Ch(t)$ being the set of structured code units covered by t which are changed in the next version.

6 Conclusions and Future Works

In this work, we proposed three prioritization strategies leveraging not only test coverage, but also the notion of code churn, i.e., information about which structural code units changed between two subsequent versions of a software. Intuitively, the parts of code that changed between two software versions are those that require to be tested with higher priority, w.r.t. unchanged parts which have already been tested. Indeed, code churns have been proven to be an effective indicator of defect-prone components.

We assessed the effectiveness of the proposed prioritization strategies by conducting an empirical study on a well-known open source software system, namely *Siena*. To do so, we implemented a prioritization solution consisting in both open source software and tools we specifically developed, and used this solution to prioritize the tests in the Siena test suite for 7 pairs of subsequent versions. As a baseline for our evaluation, we considered the well-known total coverage prioritization approach, which has been used in several other studies and does not take into account churn information. The promising results of our evaluation showed that the proposed strategies that prioritize the coverage of changed parts significantly outperform the baseline strategy in the version pairs in which there is a significant amount of changed parts. For prioritization tasks in which there is only a small amount of changed parts between versions, the results were inconclusive, and the churn-based strategies performed only slightly better than the baseline.

Moreover, we sketched a more refined approach to the evaluation of code churns, employing Abstract Syntax Trees to model the considered structured code units, and suitably-designed tree kernel functions to evaluate the degree of similarity between subsequent versions of a given unit. This approach is able to capture not only the fact that a given structured code unit changed or not, but also the nature of said change. Intuitively, we believe that not all changes have the same likelihood of introducing new faults, and thus this novel approach we are currently investigating could further improve the effectiveness of the proposed strategies.

In future works, we plan to extend our empirical evaluation by considering more software versions and additional coverage metrics, such as APSC [21]. Moreover, we plan to implement the novel churn quantification approach we sketched in this paper, and to conduct new empirical evaluations involving a greater number of software systems, considering evaluation metrics which measure fault-detection rate, such as the widely-used *APFD* metric [9]. Furthermore, we will explore the possibility of using our tree kernel-based approach to evaluate similarity between different graphical user interfaces (GUIs), which can also be represented with a tree-like structure (e.g.: xml layout, html documents). This could lead to the development of more advanced automatic GUI testing tools, which we could then evaluate as in [4].

References

1. Baresi, L., Pezzè, M.: An introduction to software testing. Electron. Notes Theor. Comput. Sci. **148**, 89–111 (2006). Elsevier
2. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Achieving scalability and expressiveness in an internet-scale event notification service. In: Proceedings of ACM Symposium on Principles of Distributed Computing, PODC 2000, pp. 219–227. ACM, New York (2000)
3. Corazza, A., Di Martino, S., Maggio, V., Scanniello, G.: A tree kernel based approach for clone detection. In: 2010 IEEE International Conference on Software Maintenance, pp. 1–5. IEEE (2010)

4. Di Martino, S., Fasolino, A.R., Starace, L.L.L., Tramontana, P.: Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing. Softw. Test. Verif. Reliab. (2020). https://doi.org/10.1002/stvr.1754
5. Di Nardo, D., Alshahwan, N., Briand, L., Labiche, Y.: Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. Softw. Test. Verif. Reliab. **25**, 371–396 (2015). https://doi.org/10.1002/stvr.1572. John Wiley and Sons Ltd
6. Do, H., Elbaum, S.G., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir. Softw. Eng.: Int. J. **10**(4), 405–435 (2005)
7. Elbaum, S., Malishevsky, A.G., Rothermel, G.: Prioritizing test cases for regression testing. In: Proceedings of International Symposium on Software Testing and Analysis, ISSTA 2000, pp. 102–112. ACM (2000)
8. Elbaum, S.G., Malishevsky, A.G., Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of ICSE, pp. 329–338. IEEE Computer Society (2001)
9. Elbaum, S.G., Malishevsky, A.G., Rothermel, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)
10. Elbaum, S.G., Rothermel, G., Penix, J.: Techniques for improving regression testing in continuous integration development environments. In: Proceedings of FSE, pp. 235–245. ACM (2014)
11. Hao, D., Zhang, L., Zhang, L., Rothermel, G., Mei, H.: A unified test case prioritization approach. ACM Trans. Softw. Eng. Methodol. **24**(2), 10:1–10:31 (2014)
12. Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., Xie, T.: To be optimal or not in test-case prioritization. IEEE Trans. Softw. Eng. **42**(5) (2016). https://doi.org/10.1109/TSE.2015.2496939
13. Harrold, M.J., et al.: Regression test selection for Java software. In: Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2001, pp. 312–326. ACM (2001)
14. Harrold, M.J., Rosenblum, D.S., Rothermel, G., Weyuker, E.J.: Empirical studies of a prediction model for regression test selection. IEEE Trans. Softw. Eng. **27**(3), 248–263 (2001)
15. Hemmati, H.: Advances in techniques for test prioritization. Adv. Comput. **112**, 185–221 (2019). https://doi.org/10.1016/bs.adcom.2017.12.004
16. Huang, Y.C., Peng, K.L., Huang, C.Y.: A history-based cost-cognizant test case prioritization technique in regression testing. J. Syst. Softw. **85**(3), 626–637 (2012)
17. Jasz, J., Lango, L., Gyimothy, T., Gergely, T., Beszedes, A., Schrettner, L.: Code coverage-based regression test selection and prioritization in WebKit. In: Proceedings of International Conference on Software Maintenance, ICSM 2012, pp. 46–55. IEEE Computer Society (2012)
18. Kaushik, N., Salehie, M., Tahvildari, L., Li, S., Moore, M.: Dynamic prioritization in regression testing. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 135–138 (2011)
19. Khatibsyarbini, M., Isa, M.A., Jawawi, D.N., Tumeng, R.: Test case prioritization approaches in regression testing: a systematic literature review. Inf. Softw. Technol. **93**, 74–93 (2018)
20. Kim, J.M., Porter, A.: A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of ICSE, pp. 119–129. ACM (2002)

21. Li, Z., Harman, M., Hierons, R.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. **33**(4), 225–237 (2007)
22. Moschitti, A.: Efficient convolution kernels for dependency and constituent syntactic trees. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 318–329. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_32
23. Nagappan, N., Ball, T.: Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th International Conference on Software Engineering, 2005. ICSE 2005, pp. 284–292. IEEE (2005)
24. Nanda, A., Mani, S., Sinha, S., Harrold, M., Orso, A.: Regression testing in the presence of non-code changes. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), pp. 21–30 (2011)
25. Ouriques, J., Cartaxo, E., Machado, P.: On the influence of model structure and test case profile on the prioritization of test cases in the context of model-based testing. In: 2013 27th Brazilian Symposium on Software Engineering (SBES), pp. 119–128 (2013)
26. Prado Lima, J.A., Vergilio, S.R.: Test case prioritization in continuous integration environments: a systematic mapping study. Inf. Softw. Technol. **121**, 106–268 (2020). https://doi.org/10.1016/j.infsof.2020.106268
27. Rothermel, G., Untch, R., Chu, C., Harrold, M.: Test case prioritization: an empirical study. In: Proceedings of the International Conference on Software Maintenance, pp. 179–188 (1999)
28. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)
29. Saha, R.K., Zhang, L., Khurshid, S., Perry, D.E.: An information retrieval approach for regression test prioritization based on program changes. In: ICSE (2015)
30. Sánchez, A.B., Segura, S., Cortés, A.R.: A comparison of test case prioritization criteria for software product lines. In: ICST, pp. 41–50. IEEE Computer Society (2014)
31. Sarro, F., Di Martino, S., Ferrucci, F., Gravino, C.: A further analysis on the use of genetic algorithm to configure support vector machines for inter-release fault prediction. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 1215–1220. ACM (2012)
32. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. **37**(6), 772–787 (2011)
33. Srikanth, H., Banerjee, S., Williams, L., Osborne, J.A.: Towards the prioritization of system test cases. Softw. Test. Verif. Reliab. **24**(4), 320–337 (2014)
34. Ul Ain, Q., Haider Butt, W., Anwar, M.W., Azam, F., Maqbool, B.: A systematic review on code clone detection. IEEE Access **7**, 86121–86144 (2019). https://doi.org/10.1109/ACCESS.2019.2918202
35. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**(2), 67–120 (2012)