Automated and Robust User Story Coverage

Mickael Gudin, Nicolas Herbaut

Automated and Robust User Story Coverage

Mickael Gudin[1] and Nicolas Herbaut[1][0000−0003−1540−2099]

Centre de Recherche en Informatique
Université Paris 1 Panthéon-Sorbonne
nicolas.herbaut@univ-paris1.fr

Abstract. Current practices in software testing such as Test Driven Development or Behavior Driven Development aim at linking code to expected behavior. In this context, code coverage is widely used to improve code quality, reduce bugs and ssure requirements satisfaction. Even if change tracking software allows finely analyzing code evolution, associating a particular code chunk to the requirements at the origin of the code modification is difficult for a large code base. In this preliminary work, we propose a new "user story coverage" metric that reports lacking requirement coverage quality, to help developers focus their efforts on enhancing unit and integration tests. We propose a methodology to compute this metric in a robust and automated fashion and evaluate its feasibility on open-source projects.

Keywords: requirements · code coverage · abstract-syntax-tree · software quality

1 Introduction

In the software industry, testing best practices such as Test-Driven Development(TDD) or Behavior-Driven Development(BDD) are now mainstream. TDD's goal is to have consistent test cases with the produced code whereas BDD's goal is to have consistent test cases with the expected behavior, which is the actual business needs [1]. In the unit testing phase, developers use the code coverage metric to assess how well the code base is tested, as a high coverage rate is deemed to make a software program less error-prone [2].

Another common practice is the use of software configuration management (SCM), to track code changes and facilitate collaboration. A good practice in SCMs is to have traceability between requirements and implementation [3], through commit messages. Commits often mentions the *issue ID*, which in turns contains a reference to the business needs behind the code modification, commonly formalized as User Stories (US) in agile teams. This precious traceability information, however, tends to degrade over time. The reason is that SCM-provided tools, such as blame, are line-centric: newer commits mask the previous ones, effectively breaking the traceability chain. As a consequence, there is no easy way, given a chuck of code, to backtrack to the requirement which led to its inclusion into the codebase.

This paper fills the gap in requirements to code traceability code by combining unit test coverage, SCM history and issue tracker data to compute a new metric, *User Story coverage*, which can be considered as a proxy to assess *Requirement coverage* . In the rest of the paper, we present some background, methods, evaluation and discussion before concluding.

2 Background and Related work

Requirements and User story Coverage In this paper, we make the assumption that issues contains User Stories, which is a form of requirement expressed from the perspective of an end-user goal. Agile teams often use USs as a proxy for proper requirements to facilitate developers' understanding of the desired features [4].

A semi-automated requirement coverage tool was proposed in [5], where authors demonstrated the feasibility of the concept. The main difference with our approach is that we rely on robust code differencing to prevent recent commits masking previous ones, and we also make the hypothesis that the process linking requirements and code are fully automated, through commit messages containing *issue IDs* pointing to USs.

In [6], authors present a new metric based on the Requirements Traceability Matrix (RTM) to better allocate testing efforts based on requirements coverage. We have a related goal, but we do not assume the existence of an RTM and use the existing unit tests to cover testing intents.

Robust Code differencing Code differencing is commonly performed through a text-based approach with `git diff` and `git blame` commands relying by default on the Myers algorithm [7]. Abstract syntax tree(AST)-based tools are the current state-of-the-art and bring more accurate differences, which are especially efficient at detecting refactoring and minor code modification [8].

In this paper, we decided to use vanilla GumTree for the AST-based approach and compare it with a text-based approach, leaving out considering recent enhancements in this field for future work.

3 Methods and Evaluation

In this section, we detail how we built the proof of concept[1] to compute the requirement coverage metric. To aggregate the code chunks to a given requirement, we aggregate the code of all commits that references the corresponding issues. We implemented two coverage approaches as two code chunks aggregation techniques : line-based and method-based which are in turn based on unit test coverage metrics: line coverage and method coverage.

[1] https://github.com/nh-group/dextorm

3.1 US Coverage Metric computation

Data Collection Data collection uses 3 different data sources: The Issue Manager (IM), Repository and Code Coverage (CC) from unit tests. In this section, we present the data sources and the different steps that lead to the generation of US Coverage data.

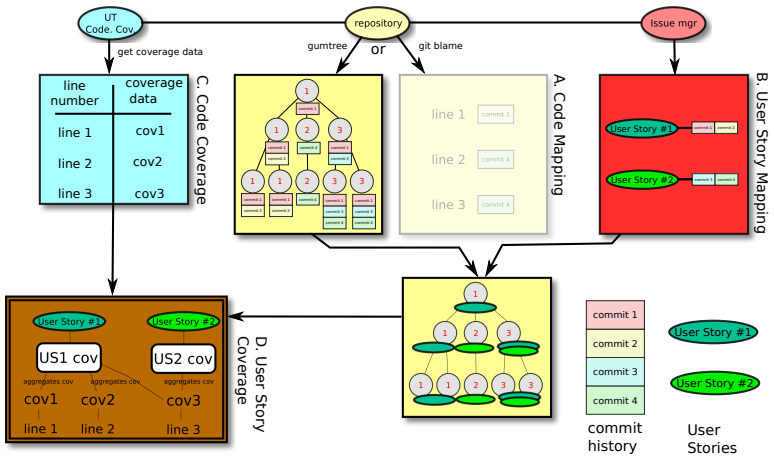

Fig. 1. High-level Architecture

Code Mapping Code mapping aims at correlating each code chunk with a commit from the repository. This operation is trivial when using the `blame` command, but offers poor precision: since (1) each line of code is associated with exactly one commit and (2) whenever any token from the line is modified by a commit, the line becomes associated with the said commit. This means that refactoring, reformatting and commenting on a line will destroy the connection between the statements and the associated commit, hence the user story.

For this paper, we developed a more robust method to compute Code Mapping, relying on GumTree. This method reads the git history for each file, and analyze each modification done on the file, producing ASTs labeled with the commit and line. More precisely, for each file F, for each commit t, we compare the two versions of the file F_{t-1} and F_t, before and after commit t. We subsequently parse the files as AST T_{t-1} and T_t, and compute the mapping M_t between T_{t-1} and T_t. If there is a mapping between nodes $N \in T_{t-1}$ and $M \in T_t$, we add all the labels of N to M. If no mapping exists, then M is labeled with commit t.

US Mapping First, we connect to the issue manager and assemble every issue corresponding to the project. The user can apply specific filters (e.g., date, la-

bel, issue status) to restrict which issues are included in the analysis. Then, we scan the whole history of the project, and gather commit messages containing references to IM issues. We finally associate each commit with the corresponding US.

Code and User Story Coverage Due to lack of space, we do not present the full description of the US coverage metric, but instead intuitions and usage examples. The *line-based User story coverage metric* is the ratio of the number of lines associated with a US, which are marked as covered by unit tests, over the total number of lines associated to the US. Likewise, *method-based User story coverage metric* follows the same principle based on method coverage.

3.2 Illustrative example

To illustrate our approach, we take the example of a US where some contact information is retrieved from a phone number, implemented in *commit1*.

```
1 public class ContactService {
2     RepositoryContact repo = new RepositoryContact();
3
4     public Contact findContactWithPhoneNumber(String
5      number) {
6         return repo.getContactWithPhoneNumber(number);
7     }
8 }
```

Listing 1.1. commit 1

```
1  +++ a/ContactService.java
2  --- b/ContactService.java
3  @@ -2,9 +2,6 @@ public class ContactService {
4       RepositoryContact repo = new RepositoryContact();
5
6       public Contact findContactWithPhoneNumber(String
7             number) {
8  +         if(number != null && number.length() == 10) {
9  +             return repo.getContactWithPhoneNumber(number
10       );
11 +         }
12 +         return null;
13 -         return repo.getContactWithPhoneNumber(number);
14      }
15  }
```

Listing 1.2. commit2 (patch format)

Assuming that the US coverage is 100% in commit1, it would stay at 100% in commit2 when computed with GumTree, since line 6 of listing 1.1 would still be associated with commit1, thanks to Gumtree being resistant to code moves.

If we were to use `git blame` however, the result would change drastically: none the news lines in listing 1.2 would be associated with commit 1 and with the US anymore: since `git blame` is line-based, the US would not be covered at all, since no line linked with commit1 would be executed in the unit tests.

3.3 Performance evaluation

We evaluated the presented US coverage computation techniques on several real-world open source projects RxJava, Shenyu and dnsjava, which use GitHub SCM to support both issues and version tracking:

We carried out computations on an Intel Xeon W-10855M CPU @ 2.80GHz with SSD and 32GB of RAM. We used the Java Microbenchmark Harness and we report the execution time with the AverageTime method over 10 executions for each combination.

We show static metrics for each project (number of classes, number of versions, number of significant line of codes and number of collected issues) and

Table 1. Performance and Runtime Comparison

Project	DA	Scope	Execution Time (s)	# Classes	# Versions	# NCLOC	# Issues	GitBlame Loss ratio	GitBlame Method Loss ratio
RxJava	GitBlame	instructions	337.54	2941	6001	368,268	3096	7.84%	23.89%
		methods	52.59						
	GumTree	instructions	5384.89						
		methods	5360.19						
apache/shenyu	GitBlame	instructions	32.28	1807	2488	100,399	1559	4.48%	20.96%
		methods	15.42						
	GumTree	instructions	846.75						
		methods	841.76						
dnsjava	GitBlame	instructions	15.91	277	2066	22,308	147	6.25%	25.62%
		methods	8.63						
	GumTree	instructions	101.17						
		methods	105.25						

the computation runtime for each combination of diff algorithms (GitBlame and GumTree) and scope (instruction-based or method-based). We also report two performance metrics: the gap between the most accurate coverage computation method (GumTree algorithm computed on instructions) and another reference method: G_i (GitBlame Loss ratio, which represents the normalized average difference between GumTree and GitBlame coverage with instruction scope) and I_i (GitBlame Method Loss Ratio, that represent the normalized average difference between instruction-based coverage for the GumTree algorithm and method-based coverage for GitBlame). We define these metrics as: $G_i = \sqrt{\sum_{i \in I} (c_i^{g,\text{inst}} - c_i^{b,\text{inst}})^2}/\#I$, and $I_i = \sqrt{\sum_{i \in I} (c_i^{g,\text{inst}} - c_i^{b,\text{meth}})^2}/\#I$ where I is the set of the issues for the project, $c_i^{g,\text{inst}}$ (resp., $c_i^{b,\text{inst}}$) is the coverage ratio reported by GumTree (resp., GitBlame) for issue i with the instruction scope and $c_i^{b,\text{meth}}$ is the coverage ratio for the method scope for the GitBlame algorithm.

4 Discussion

As we can see from Table 1, computations using the GitBlame algorithm outperform GumTree by one order of magnitude and shows a coverage precision loss of 4.48% to 7.84%. The main factor explaining this is the necessity for GumTree to compute as many diff trees as there are revisions for each file with an $\mathcal{O}(N^2)$ worst-case complexity (with N equals the number of nodes in the AST), while git blame relies on a diff algorithm [9] requiring only O(M) space (where M is the number of tokens of the file). Interestingly, using the method scope (based on comparing method signatures), does not provide a large benefit in runtime for the GumTree algorithm, since the main bottleneck is the computation of AST mappings that need to be computed anyway. For GitBlame, however, the time reduction seems significant (from 50% to 84% improvement) while increasing the loss ration from 20.96% to 25.62%. While the precision erosion is substantial, it stays limited. This suggests that we could use both approaches conjointly: using GumTree with instructions (the slowest but most precise) in an off-line setting (e.g., on a software factory while computing the other continuous integration

tasks), along with GitBlame with methods (the fastest, bit least precise) on the developer's workstation for a fast feedback loop.

5 Conclusion

In this article, we proposed a methodology to compute a proxy for requirement coverage that we called *User Story coverage*. Thanks to AST-based code differencing and data aggregation from issue manager, SCM and unit tests coverage, the metric can be automatically obtained robustly. As future work, we plan to integrate this metric in an IDE and follow a design science approach to evaluate how it can improve code quality throughout the development lifecycle. We also expect performance improvement through more advanced AST-based code differencing techniques.

Reference

[1] F. Zampetti, A. Di Sorbo, C. A. Visaggio, G. Canfora, and M. Di Penta, "Demystifying the adoption of behavior-driven development in open source projects," *Information and Software Technology*, vol. 123, p. 106311, 2020.

[2] T. Bach, A. Andrzejak, R. Pannemans, and D. Lo, "The impact of coverage on bug density in a large industrial software project," in *2017 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*, 2017, pp. 307–313.

[3] *Standard for configuration management in systems and software engineering*. IEEE Standard 828-2012, 2012.

[4] M. Cohn, *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[5] R. Mordinyi and S. Biffl, "Exploring traceability links via issues for detailed requirements coverage reports," in *2017 IEEE 25th international requirements engineering conference workshops (REW)*, 2017.

[6] C. Ziftci and I. Kruger, "Getting more from requirements traceability: Requirements testing progress," in *2013 7th international workshop on traceability in emerging forms of software engineering (TEFSE)*, 2013.

[7] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?" *Empirical Software Engineering*, vol. 25, no. 1, pp. 790–823, 2020.

[8] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on automated software engineering*, 2014, pp. 313–324.

[9] E. W. Myers, "An o(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.