Analyzing Log Entries of Version Control Systems for Detailed Requirements Coverage Reports

Richard Mordinyi*† and Stefan Biffl‡
*BearingPoint GmbH, Vienna, Austria
Email: richard.mordinyi@bearingpoint.com
†SBA Research, Vienna, Austria
Email: rmordinyi@sba-research.org
‡Institute of Software Technology and Interactive Systems
Vienna University of Technology, Vienna, Austria
Email: stefan.biffl@tuwien.ac.at

***Abstract*—The coverage of requirements is a fundamental need throughout the software life cycle. It gives project managers an indication how well the software meets expected requirements. A precondition for the process is to link requirements with project artifacts, like test cases. There are various (semi-) automated methods deriving traceable relations between requirements and test scenarios aiming to counteract time consuming and error-prone manual approaches. However, even if traceability links are correctly established coverage is calculated based on passed test scenarios without taking into account the overall code base written to realize the requirement in the first place.**

In this paper we introduce the so called "Requirements-Testing-Coverage" (ReTeCo) approach that automatically establishes links between requirements and test cases through source code lines which a) have been written in the context of an issue as part of a linked requirement, b) have been committed into a version control system, and c) produce code coverage results. Since the approach takes into account source code lines it is able to calculate coverage reports on a fine-grained contextual level rather than on the result of high-level artifacts.

We show the feasibility of the approach and initial evaluation results using the code base and test scenarios of large open source projects.

***Index Terms*—Requirement, Testing, Test Scenario, Coverage, Issue Tracking System, Version Control System**

I. Introduction

The success of a software engineering project mostly depends on the business value it is able to produce. It is therefore essential that beside achieving quality in software artifacts team members strive for fulfilling elicited requirements. The ability to ensure that the software meets the expected requirements also depends on methods being able to report that the piece of software delivered did in fact meet all the requirements. Especially in the context of maturing software that goes into several iterations of enhancements and bug fixes, it becomes more and more daunting to ensure requirement coverage in the software [1].

Generally, requirement coverage is the number of requirements passed in proportion to the total number of requirements [2]. A requirement is considered passed if it has linked test cases and all of those test cases are passed. This implies that there has to be a traceable mapping [3] between requirements and test cases. If a link between a requirement and another project artifact, e.g. a test case, exists and this link is correct, the requirement is covered by that project artifact. If however test cases are not associated with individual requirements it could be difficult for testers to determine if requirements are adequately tested [4] leading to reported problems [3], [5], [6].

There are various approaches how traceability links between requirements and test cases may be identified (see Sec. II for details). Some of them rely on certain keywords across artifacts which are manually inserted and maintained by software developers and testers and used to establish links, while other approaches aim for full automation without human guidance by e.g., information retrieval methods. Regardless of their accuracy [7] in establishing such links, the main limitation of those methods is that links taken into consideration for requirement coverage reports reflect upon requirements and test cases as high level artifacts. While methods may access the code base during the analyzing process in order to reason about potential links they do not consider that part of the source base that actually represents and forms a specific requirement or is covered by tests [8].

However, project and test managers would like to have both information on the quality of the code base as well as on how well that quality is reflected upon the requirements which are to be fulfilled and delivered. Therefore, they need detailed, fine-grained information that allows them to reason about the progressing quality of a requirement during development phases. The quality of the code base may be measured by methods like code coverage - *the degree to which the source code of a program has been tested* [9]. However, in the context of requirements coverage we need to refine its definition as the degree to which the source code of (ie. relevant to execute / composes) a requirement is covered by tests.

In this paper we introduce the so called "Requirements-Testing-Coverage" (ReTeCo) approach that automatically establishes links between requirements and test cases by identifying the source code lines that form a requirement and by identifying the test cases which cover those source code lines. If a source code line that is covered by a test case is part of the source code line relevant for a requirement, a match has been found and a link between the requirement and the test case

is created. Identification of requirement relevant source code lines is performed a) by extracting the issue number(s) of a ticketing system which are in relation to a requirement and b) by analyzing the log of the versioning system for changes on the code base introduced in the context of the issue. A match between a requirement and any of the test cases is given if code coverage analyzes shows that any of the identified source code lines is covered by at least one of the test cases. If a set of source code lines supports a single requirement, the number of source code lines (within that set) which are covered by test scenarios represent the percentage of coverage. Based on large open source projects we will show the feasibility of the approach and will discuss its advantages and limitations.

The remainder of this paper is structured as follows: Section II summarizes related work on requirements traceability and approaches on deriving requirements and test scenario relations. Section III presents research questions while section IV depicts a typical use case. Section V describes the ReTeCo approach. The feasibility and the initial evaluation results of the prototype implementation are illustrated in section VI and discussed in section VII. Finally, section VIII draws conclusions and pictures future work.

II. Related Work

Traceability is the ability to follow the changes of software artifacts created during software development [10], [11] and is described by the links that map related artifacts [12]. This section summarizes related work on methods and approaches on requirements traceability, requirements coverage by means of test scenarios, and traceability between test cases and source code.

A. Manually guided Approaches

Attempts to automate the generation of traceability links concentrated on parsing the text in the code documentation to find textual relations to requirement identifiers or to the requirement descriptions [13]. Similar approaches like [14] improved accuracy by introducing specific types of comments. When text written in these comments follow some rules, the tool can trace it accurately to its requirement. The advantage of this this approach is that it separates the comments written for the trace from the documentation itself. Despite their accuracy, the text parser must be very accurate and must interpret the meaning of the textual documentation to find a relation to the requirements. Even if the parser is accurate, there is no guarantee that the documentation of both the requirements and the code is up-to-date. Poor maintenance lead to wrong results and thus to higher risks undesirably increasing efforts required from developers.

B. Information Retrieval

In the information retrieval area, there are various models which provide practical solutions for semi-automatically recovering traceability links [15]. At first, links are generated by comparing source artifacts (e.g., requirements, use cases) with target artifacts (e.g., code, test cases) and ranking pairs of them by their similarity, which is calculated depending on the used model. A threshold defines the minimum similarity score for a link to be considered as a candidate link. These candidate links are then evaluated manually, where false positives are filtered out. The remaining correct links are called traceability links. Evaluations [16] show that this process of traceability links recovery with the aid of an information retrieval tool is significantly faster than manual methods and tracing accuracy is positively affected. Nevertheless, human analysts are still needed for final decisions regarding the validity of candidate links. Variations of the method base on the vector space model (VSM) and the probabilistic model (PM) [15], Latent Semantic Indexing (LSI) [17], the VSM with thesaurus support (VSM-T) [18], Part-of-Speech-enabled VSM (VSM-POS) [19], latent Dirichlet allocation (LDA) [20], explicit semantic analysis (ESA) [21], or the normalized Google distance (NGD) [22].

However, there are limitations when using IR-based traceability link recovery methods that cannot be completely solved by improvements of IR methods either. Namely, it is not possible to identify all correct trace links without manually eliminating a large number of false positives. Lowering the similarity threshold to find more correct links, will in fact lead to a strongly increasing amount of incorrect links that have to be removed manually [23].

C. Model-based Techniques

Model-based requirement traceability techniques try to translate requirements to e.g., UML, XML or formal specifications. This is necessary to semi-automatically generate trace links and/or check them to consequently establish a certain degree of automation. In [24] informal requirements are restructured by means of the Systems Modelling Language (SysML) [25] into a requirement-model. These elements are manually linked with various elements of different design models, which are used for automatically deriving test cases relying on different coverage criteria and the corresponding links. In [26] trace links are generated during model transformations, which are applied to produce a design model and in further consequence discipline-specific models from a requirement model. The resulting correspondence model between source and target represents the trace links. Methods relying on UML models are for example described in [27] or [28].

In the application of formal specifications/models requirement-, architecture- and design models are expressed for example as linear temporal logic [29], in Z-notation [30] or B-Notation [31]. The generation and/or validation of the trace links can be automatized through a model checker [29], a rule based checker [30] or model based testing [31]. Further research work has been invested in techniques like annotating code, design elements or tests with traceability information [32], [7], [33], scenario-based techniques [34], graph-based techniques [35], or techniques in the context of test cases [36] relying on naming conventions, explicit fixture declarations, static test call graphs, run time traces and lexical analysis, and co-evolution logs.

III. Research Issues

The quality of software development tools and environments in supporting development has improved significantly during the last decade. While at first the effective, quality-assured support of development was one of the main concerns, nowadays tools tend to focus on better interconnecting the engineer with other information sources. They tend to interlink information [37] in any of the used tools in the project's software engineering environment as much as possible. Anyhow, effective software engineering projects cannot afford to dispense using at least a requirement modeling tool, issue tracking system, or a version control system [38] in their environment.

Since the main aim of an engineering project is to meet requirements as expected by the customer, it is essential for project managers and for engineers to know the degree of requirement coverage. Given the limitations of current requirements traceability approaches (see Sec. II) we have formulated the following research questions:

- how to make use of links between information units provided by engineering tools for the establishment of traceability links between requirements and test cases?
- to what extent do interlinked information in engineering tools allow more fine-grained reporting on coverage?
- to what degree is a process for coverage calculation automatable or still require human intervention?

To answer these research issues we designed the so called "Requirements-Testing-Coverage" (ReTeCo) approach and implemented a prototype [1]. We then performed initial evaluations using the code base, requirements and issue sets of large and popular open source projects.

IV. Use Case

Figure 1 depicts a typical software engineering process describing how requirements are "transformed" into source code. The requirements engineer is responsible for eliciting and clearly specifying the project's requirements (Fig. 1, 1). Usually, such artifacts are managed by a requirements management tool, like Polarion[2], Rational[3], or RMsis[4]. In cooperation with a release manager and usually with a member of the development team the requirements are divided into multiple working tasks (ie. issues) (Fig. 1, 2), each having a unique identifier (ie. issue number/id). This requires high understanding of the client specifications and the business goals alongside with high understanding of the technical abilities of the development teams. Tools for managing working tasks are for example Bugzilla[5], HP Quality Center[6], or Atlassian Jira[7]. At this point the release manager inserts a (web) link into the working task pointing to the requirement, so that any other

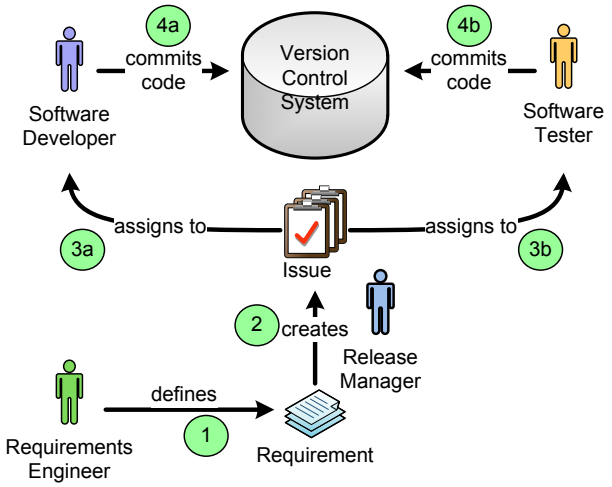

Fig. 1. Intertwining processes of various stakeholders in a software engineering environment

team member is able to look up the details of the requirement in case of unclarities.

In principal, a working tasks may describe any pensum for the assignee of the task. In this context it either describes the details to be implemented by the developer (Fig. 1, 3a), or it documents the test cases to be implemented by a tester (Fig. 1, 3b).

In both cases development will be done using a version control system. Once the working task is finished the changes in the repository are committed and pushed (Fig. 1, 4a and 4b). The changes made to the code base reflect the required behaviour as described in each working task. The commit itself requires the committer to provide a commit message. Beside describing what changes were added to the code base, the committer also adds the ID of the working task to the message[8] indicating the context in which the development was done. Depending on the size of the working task or the way a developer works several commits may have been done in the context of one working task.

The release manager needs to estimate and evaluate the state of development at different phases of the project life cycle so that he/she can decide upon delivery of the software. Once all working tasks have been done, he/she asks - among other things - the following questions to ensure high-quality delivery: a) which test scenarios check intended functionality of a requirement, b) how many of those tests are positive, c) how many of those tests fail, and d) could have any test cases been overlooked?

V. Solution Approach

The following section explains the traceability link model and the process of how to make use of it for detailed requirement coverage reports.

[1] download available at https://github.com/mindpixel/requirementsCoverage
[2] http://polarion.siemens.com
[3] http://www-03.ibm.com/software/products/en/ratidoor
[4] https://products.optimizory.com/rmsis
[5] https://www.bugzilla.org/
[6] https://saas.hpe.com/en-us/software/Quality-Center
[7] https://www.atlassian.com/software/jira

[8] an example on message structure: https://confluence.atlassian.com/fisheye/using-smart-commits-298976812.html

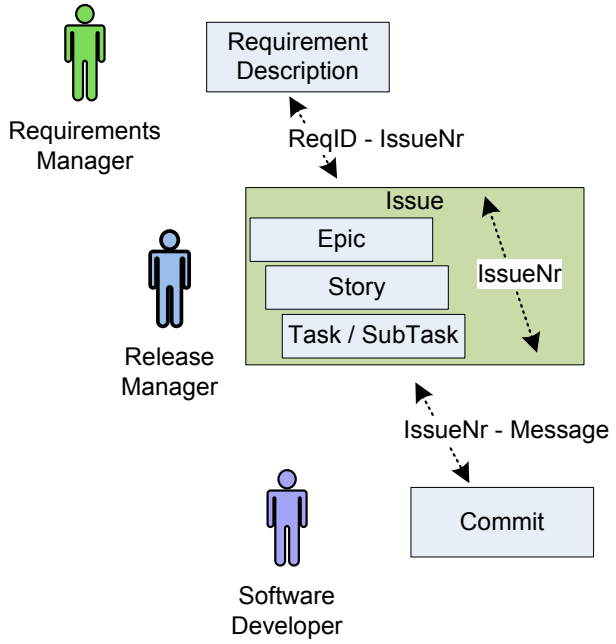

Fig. 2. Tool-supported Linking of Model Elements define implicit Traceability Links

A. Traceability Link Model

The "Requirements-Testing-Coverage" (ReTeCo) approach calculates the coverage of a requirement as the degree to which the source code of (ie. relevant to execute) a requirement is covered by tests. In order to to do it needs to rely on a traceable link between a requirement and a source code line. Figure 2 presents the relation of engineering artifacts that form the TLM for ReTeCo.

The central element of the TLM is the *Issue*. An issue is in relation to a requirement as well as to the code base. The relation between requirement and an issue is set up when the requirement is organized as a set of issues reflecting the intend of the requirement. The relation is defined within an issue containing a reference to the requirement. This means that there is a 1:n relation between the two model elements.

The relation between source code lines and an issue is set up by the developer when the developer commits the changes into the version control system introduced into the code base due to the task description in the issue. The relation is defined within the commit message by providing the issue number in that message. Since a source code line may have been altered several times, there is an n:m relation between source code and issue.

Issues may also be organized in an hierarchy. In the context of agile software development [39] it is common to distinguish between Epics, Stories, Tasks (and Sub-Tasks). An epic is a large body of work that can be broken down into a number of smaller stories. A story or user story is the smallest unit of work in an agile framework. It is a software system requirement that is expressed in a few short sentences, ideally using non-technical language. The goal of a user story is to deliver a particular value back to the customer. A task is a concrete implementation requirement for the development team. Relations between the various issues types are created (semi-) automatically whenever a sub-issue is created.

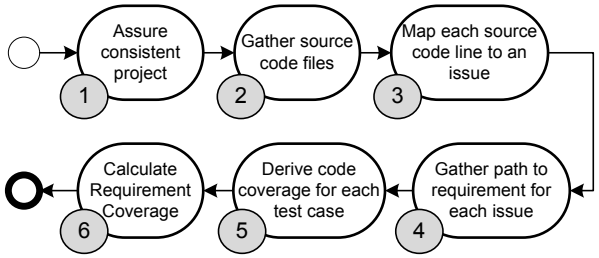

Fig. 3. Main Process Steps of the ReTeCo Approach

B. Requirement Coverage Calculation Process

This section explains the main process step (see also Fig. 3) of the ReTeCo approach as well as implementation details of the prototype:

1. Check Consistency of the Project: In the first step the process has to ensure that project under investigation is correct and the source code can be compiled.

2. Build Source Index: In the second step the process searches all directories recursively for source code files and memorizes their locations.

3. Build Commit Index: In the third step (see also Fig. 4) the process traces each line of the source code to the Issue-ID that initially created or changed that line. In the ReTeCo prototype this is done by parsing the commit (ie. log) messages of the version management system. The prototype parses a git repository[9] of the target project by using the JGit framework[10]. It calls a series of git commands on the repository for each source code file to find out which issue is related to which source code line. At first, the prototype calls the command *git blame* for the inspected file. The result of this command contains the revision number for each line of code. Then the prototype calls the command *git log* for the each revision number. The result of running this command contains the commit message of the commit in which the line of code was modified. By parsing the commit message (e.g., using regular expression) the Issue-ID is extracted from the commit message. After repeating this process step for all lines of source code in all source files, the final outcome of this step is a set of traceable relations between source code lines and Issue-IDs.

4. Build Requirement Index: In the fourth step, each IssueID is traced back to a requirement. For each issue the corresponding parent issue is requested from the issue tracking system until the "root" issue (e.g., Epic) has been found. As explained in the previous section, the "root" issue contains the reference to the requirement it is reflecting. At this point the

[9]https://git-scm.com/
[10]https://eclipse.org/jgit/

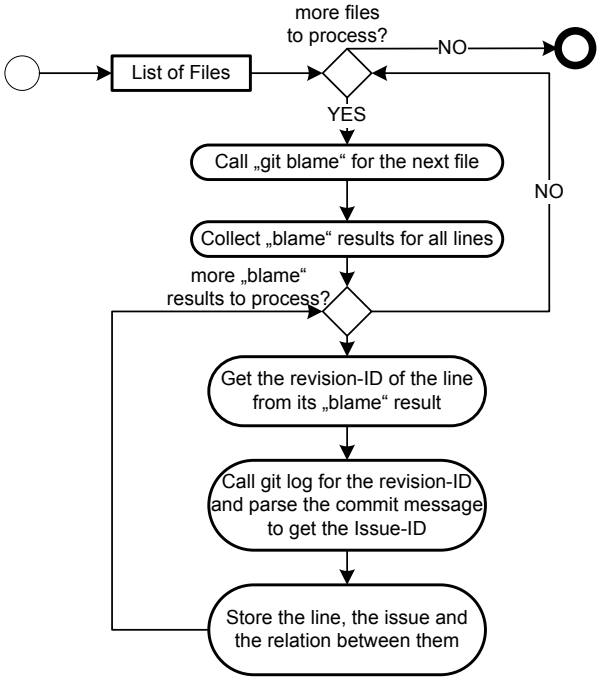

Fig. 4. Process Steps for Calculating Issue-Code Relation

traceability links between source code lines and requirements have been established.

5. Perform Test Coverage: In the next step the code coverage information of the project is gathered. In this step for each line of source code it needs to be retrieved by which test case that line is covered. The outcome of this process step is a set of relations between source code lines and covering test cases. The prototype divides this task into two steps. In the first step all test cases are executed (e.g., by calling the maven[11] test goal). During the process the name of each test cases and its status (success or failure) is collected. The prototype retrieves this information by parsing the maven surefire report[12]. In the second step a code coverage measurement tool is run (e.g., JaCoCo[13]) to measure the test coverage information of the project. However, JaCoCo is not able to provide direct traces between a specific test case and a source code line. It only indicates if the code is traversed by any of the test cases. Therefore, code coverage reports are created for each test case separately in order to be able to relate each test case and its resulting report to a specific issue and thus to a requirement. The prototype parses the resulting JaCoCo report and extracts which lines of code are relevant to the coverage measurement.

6. Calculate Requirement Coverage: In the final step the requirement coverage report is calculated (see Fig. 5 and Fig. 6 as examples). At this point it is known a) which source code line is related to which Issue-IDs (and therefore to which requirement) and b) which source code line is covered by which test case. This allows the process to start calculating the coverage for each requirement. Calculation is done by analyzing and aggregating the results of each test coverage report in the context of the corresponding issues and source code lines. The final outcome of this process step contains the coverage information of the entire project - for each requirement, and for each issue of each requirement.

[11]https://maven.apache.org/
[12]http://maven.apache.org/surefire/maven-surefire-plugin/
[13]http://www.eclemma.org/jacoco/

VI. Initial Evaluation

In the following we demonstrate the feasibility of our approach by illustrating initial evaluation results. In order to investigate and evaluate the approach, we have implemented a prototype[1] and analyzed its performance in sense of execution time and memory consumption. We have evaluated the ReTeCo approach in the context of two open source projects ops4j paxweb[14] and Apache qpid[15]. Table I depicts the key characteristics of each of the projects: size of the code base, number of test cases, number of issues, and number of commits.

The evaluations were conducted on a Intel Core i7-2620M with 2,7GHz and 8GB RAM running on a Windows 7 environment. The prototype was implemented in java and compiled with Java 8. It uses JGit[10] v4.4 to execute git commands, Maven Invoker[16] v2.2 in order to execute maven goals on target projects, JaCoCo[13] v0.7.9 as the code coverage measurement tool to create the code coverage reports of the target project, and JFreeChart[17] v1.0.14 to create pie-charts showing the final requirements coverage reports.

Table I shows that the qpid project has 5,8 times more source code lines (even half the number of commits) and 28 times more test cases than the paxweb project. Although qpid is a larger project the execution of the prototype requires only relatively little more amount of RAM. However, the process execution time is in case of qpid significantly greater than in case of paxweb. Anyhow, the reason for the large execution time is given due to the limitation of JaCoCo. JaCoCo is not capable of directly (ie on the first run) reporting traces between test cases and a source code lines. Each test case had to be run separately, leading to high execution times.

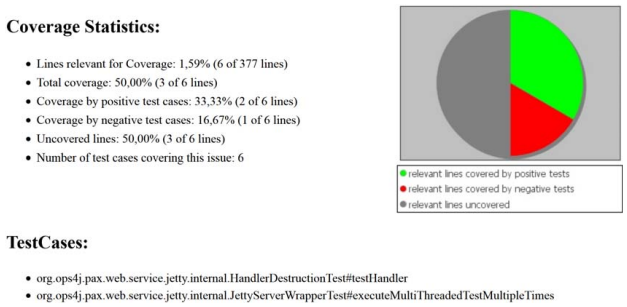

Fig. 5. Excerpt of the Coverage Report in the Context of an Issue

[14]https://github.com/ops4j/org.ops4j.pax.web
[15]https://qpid.apache.org/
[16]https://maven.apache.org/plugins/maven-invoker-plugin/
[17]http://www.jfree.org/jfreechart/

project	# of source code lines	# of test cases	# of issues	log size	execution time	memory consumption
org.ops4j.pax.web	82705	63	1229	3979	1h 15m	177M
org.apache.qpid.qpid-java-build	481112	1775	1684	7768	2d 09h 34m	207M

TABLE I
PERFORMANCE CHARACTERISTICS AND EVALUATION RESULTS OF INVESTIGATED OPEN SOURCE PROJECTS

Fig. 5 shows collected information about the coverage of a single issue which is in relation to 6 test cases. It shows that while 377 lines of code were written or updated in the context of that issue, only 6 of them are relevant for coverage analysis. Left out of consideration are lines such as comments, javadoc, import statements, or configuration files. 50% of those lines are covered by tests - 2 lines (33,33%) positively and 1 line (16,67%) negatively. The rest 3 lines (50%) are not covered by tests at all.

Fig. 6 shows aggregated information related to the details of a requirement. It shows that 30065 lines of code were contributed to the requirement in 319 issues. Out of them 3642 lines of code are considered relevant. Out of the set of relevant lines 208 lines of code or 5,71% are covered by tests - 158 lines (4,34%) positively and 50 lines (1,37%) negatively. The rest of the lines 94,29% are not covered by tests.

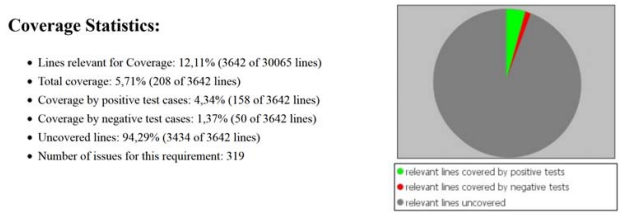

Fig. 6. Excerpt of the Coverage Report in the Context of a Requirement

VII. DISCUSSION

The Requirements-Testing-Coverage (ReTeCo) approach aims to provide a requirements coverage report on the basis of aggregated test coverage results of each issue contributing to the composition of the requirement.

Although the approach is automatically capable of calculating the requirements coverage on a source code line bases, it depends on some preconditions. It requires from various members of a software engineering project to properly handle working task identifiers (ie. Issue-IDs of an issue tracking system). The approach needs from the release manager to insert a reference to the requirement into the issue. In case of an issue hierarchy it needs to have ensured that the hierarchy allows unambiguous traceability from leaf issues to the root issue. Finally, it needs from developers to have the Issue-ID inserted into the commit message.

However, these tasks are not performed by a single role, but may be distributed among project members reducing the overall complexity and responsibility for each of the roles. Additionally, as described in Sec. II some of these tasks can be automatized (e.g., by using textual comparison). Furthermore, test managers or requirement managers do not need to make any directives for developers regarding the correct nomenclature and usage of source code elements (like naming of classes, javadoc structure, or use of specific keywords) helping developers concentrate on the task described in their issue. While the correct interlinking of issues may be outsourced to the deployed issue tracking system, quality checks may be put in place which ensure: a) root issue has a reference to a requirement (e.g., if a certain reference-type is instantiated can be queried in an issue tracking system) and b) the commit message follows a certain regular expression pattern (e.g., check if Issue-ID is followed by message content can be executed in so called pre-commit git hooks).

In general uncovered requirements refer to requirements with no linked test cases. It helps project members know about test cases that should be created to cover such requirements as well. The ReTeCo approach is also capable of pointing out such requirements. However, the approach considers uncovered requirements as ones where no source code line is covered by any test case in the context of the issues composing that requirement.

The percentage value calculated by the presented approach may be misleading and has to be read with caution. There are at least two scenarios to consider:

Scenario 1: Assuming there is a group of requirements but from development point of view only with small differences between them. Usually, a developer invests a lot of time and code into realizing the first requirement while implementing the others through parametrization. This implies that there is a large number of commits and issues related to the first requirement while only little for the the others. Since the changed code base is smaller for those requirements, it is therefore easier to reach higher coverage.

Scenario 2: If there is requirement under development it might be the case that the approach temporarily calculates 100% coverage. This however, may only state that the source code lines composing the requirement up to that point in time are completely covered by tests. The approach is not able to indicate when development of a requirement has finished.

VIII. CONCLUSION AND FUTURE WORK

It is the project members responsibility to develop and deliver requirements as expected by the customer. It is also their responsibility to achieve quality in software artifacts, especially in the ones needed to fulfill the requirement. Requirements coverage indicates the number of requirements passed in proportion to the total number of requirements. A requirement is considered passed if it has linked test cases and all of those test cases are passed. This requires traceable links between requirements and test cases. While there are various approaches describing how to establish mappings

(semi-) automatically, they focus on test cases as high level artifacts without taking into consideration the code base that is under test or compose a requirement.

This paper presented the so called "Requirements-Testing-Coverage" (ReTeCo) approach which creates traceability links between requirements and test cases through source code lines which a) have been written in the context of an issue, b) have been committed into a version control system, and c) produce code coverage results. Although the approach requires manual human assistance to properly set Issue-IDs, it does not require explicit linking of requirements and test cases as it recreates traceability links implicitly through analyzing references created by various team members throughout the software development life cycle between requirements, issues, and commit message. Since the approach takes into account source code lines it is able to calculate coverage reports on a fine-grained contextual level. The paper therefore calculates requirement coverage not by passed test cases linked to that requirement but indirectly, by analyzing the lines of tested source code composing a requirement.

Initial evaluation results in the context of two open source projects showed the feasibility of the proposed approach. However, the ReTeCo approach is not able to identify relations between requirements and test cases if tests do not cover any source code line realizing a requirement.

Future work will focus on a) combining the ReTeCo approach with approaches from related work in order to avoid manual linking of requirements with issues and issues with commits (e.g., based on textual comparison) whenever they are created or committed, and b) improving precision of the approach by considering dependencies between requirements.

ACKNOWLEDGMENT

The authors thank Moaz Baghdadi, Betim Bryma, Andreas Riegler, and Klaus Walla for their contributions.

REFERENCES

[1] N. Ali, Y. G. Guhneuc, and G. Antoniol, "Trustrace: Mining software repositories to improve the accuracy of requirement traceability links," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 725–741, May 2013.

[2] R. Krishnamoorthi and S. S. A. Mary, "Factor oriented requirement coverage based system test case prioritization of new and regression test cases," *Information and Software Technology*, vol. 51, no. 4, pp. 799 – 808, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584908001286

[3] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *Proceedings of IEEE International Conference on Requirements Engineering*, Apr 1994, pp. 94–101.

[4] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based automated black-box test generation," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, 2001, pp. 489–495.

[5] J. Cleland-Huang, C. K. Chang, and M. Christensen, "Event-based traceability for managing evolutionary change," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 796–810, Sept 2003.

[6] M. Heindl and S. Biffl, "A case study on value-based requirements tracing," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 60–69. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081717

[7] S. Delgado, "Next-generation techniques for tracking design requirements coverage in automatic test software development," in *2006 IEEE Autotestcon*, Sept 2006, pp. 806–812.

[8] M. Gittens, K. Romanufa, D. Godwin, and J. Racicot, "All code coverage is not created equal: A case study in prioritized code coverage," in *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '06. Riverton, NJ, USA: IBM Corp., 2006. [Online]. Available: http://dx.doi.org/10.1145/1188966.1188981

[9] C. Stanbridge, "Retrospective requirement analysis using code coverage of gui driven system tests," in *2010 18th IEEE International Requirements Engineering Conference*, Sept 2010, pp. 411–412.

[10] R. M. Parizi, S. P. Lee, and M. Dabbagh, "Achievements and challenges in state-of-the-art software traceability between test and code artifacts," *IEEE Transactions on Reliability*, vol. 63, no. 4, pp. 913–926, Dec 2014.

[11] P. Lago, H. Muccini, and H. van Vliet, "A scoped approach to traceability management," *J. Syst. Softw.*, vol. 82, no. 1, pp. 168–182, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2008.08.026

[12] S. Maro, A. Anjorin, R. Wohlrab, and J. P. Steghfer, "Traceability maintenance: Factors and guidelines," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sept 2016, pp. 414–425.

[13] J. Guo, N. Monaikul, and J. Cleland-Huang, "Trace links explained: An automated approach for generating rationales," in *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, Aug 2015, pp. 202–207.

[14] S. M. Ooi, R. Lim, and C. C. Lim, "An integrated system for end-to-end traceability and requirements test coverage," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*, June 2014, pp. 45–48.

[15] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1041053

[16] A. De Lucia, R. Oliveto, and G. Tortora, "Assessing ir-based traceability recovery tools through controlled experiments," *Empirical Softw. Engg.*, vol. 14, no. 1, pp. 57–92, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9090-8

[17] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776832

[18] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 4–19, Jan. 2006. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.3

[19] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, vol. 25, no. 7, pp. 743–762, 2013. [Online]. Available: http://dx.doi.org/10.1002/smr.1564

[20] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[21] E. Gabrilovich and S. Markovitch, "Computing semantic relatedness using wikipedia-based explicit semantic analysis," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, ser. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 1606–1611. [Online]. Available: http://dl.acm.org/citation.cfm?id=1625275.1625535

[22] R. L. Cilibrasi and P. M. B. Vitanyi, "The google similarity distance," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, no. 3, pp. 370–383, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2007.48

[23] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, Sep. 2007. [Online]. Available: http://doi.acm.org/10.1145/1276933.1276934

[24] F. Abbors, D. Truscan, and J. Lilius, "Tracing requirements in a model-based testing approach," in *2009 First International Conference on Advances in System Testing and Validation Lifecycle*, Sept 2009, pp. 123–128.

[25] L. Delligatti, *SysML Distilled: A Brief Guide to the Systems Modeling Language*, 1st ed. Addison-Wesley Professional, 2013.

[26] M. Fockel, J. Holtmann, and J. Meyer, "Semi-automatic establishment and maintenance of valid traceability in automotive development processes," in *2012 Second International Workshop on Software Engineering for Embedded Systems (SEES)*, June 2012, pp. 37–43.

[27] M. Grechanik, K. S. McKinley, and D. E. Perry, "Recovering and using use-case-diagram-to-source-code traceability links," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287640

[28] P. Valderas and V. Pelechano, "Introducing requirements traceability support in model-driven development of web applications," *Inf. Softw. Technol.*, vol. 51, no. 4, pp. 749–768, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.09.008

[29] A. Goknil, I. Kurtev, and K. Van Den Berg, "Generation and validation of traces between requirements and architecture based on formal trace semantics," *J. Syst. Softw.*, vol. 88, pp. 112–137, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.10.006

[30] S. Sengupta, A. Kanjilal, and S. Bhattacharya, "Requirement traceability in software development process: An empirical approach," in *2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping*, June 2008, pp. 105–111.

[31] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting, "Requirements traceability in automated test generation: Application to smart card software validation," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083282

[32] D. C. Ni, J. Martinez, J. Eccles, D. Thomas, and P. K. M. Lai, "Process automation with enumeration and traceability tools," in *Industrial Technology, 1994., Proceedings of the IEEE International Conference on*, Dec 1994, pp. 361–365.

[33] S. Ahn and K. Chong, "A feature-oriented requirements tracing method: A study of cost-benefit analysis," in *2006 International Conference on Hybrid Information Technology*, vol. 2, Nov 2006, pp. 611–616.

[34] A. Egyed and P. Grunbacher, "Automating requirements traceability: Beyond the record replay paradigm," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, 2002, pp. 163–171.

[35] B. Burgstaller and A. Egyed, "Understanding where requirements are implemented," in *2010 IEEE International Conference on Software Maintenance*, Sept 2010, pp. 1–5.

[36] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *2009 13th European Conference on Software Maintenance and Reengineering*, March 2009, pp. 209–218.

[37] M. Ortu, G. Destefanis, M. Kassab, and M. Marchesi, "Measuring and understanding the effectiveness of jira developers communities," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, May 2015, pp. 3–10.

[38] J. Portillo-Rodrguez, A. Vizcano, M. Piattini, and S. Beecham, "Tools used in global software engineering: A systematic mapping review," *Information and Software Technology*, vol. 54, no. 7, pp. 663 – 685, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584912000493

[39] A. Cockburn, *Agile Software Development*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.