Are Your Requirements Covered?

Richard Mordinyi[1,2](✉)

[1] BearingPoint GmbH, Vienna, Austria
richard.mordinyi@bearingpoint.com,rmordinyi@sba-research.org
[2] SBA Research, Vienna, Austria

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 the "Requirements-Testing-Coverage" (ReTeCo) approach is described that establishes links between requirements and test cases by making use of knowledge available in software tools supporting the software engineering process and are part of the software engineering tool environment. In contrast to traditional approaches ReTeCo generates traceability links indirectly by gathering and analyzing information from version control system, ticketing system and test coverage tools. Since the approach takes into account a larger information base it is able to calculate coverage reports on a fine-grained contextual level rather than on the result of high-level artifacts.

Keywords: Requirement · Testing · Test scenario · Coverage
Issue tracking system · Version control system

1 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 [3].

Generally, requirement coverage is the number of requirements passed in proportion to the total number of requirements [27]. A requirement is considered

© Springer International Publishing AG 2018
D. Winkler et al. (Eds.): SWQD 2018, LNBIP 302, pp. 103–120, 2018.
https://doi.org/10.1007/978-3-319-71440-0_6

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 [22] 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 [42] leading to reported problems [10,22,26].

There are various approaches how traceability links between requirements and test cases may be identified (see Sect. 2 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 [14] 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 [20].

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* [41]. However, in the context of requirements coverage we need to refine its definition as the degree to which the source code of (i.e. relevant to execute/composes) a requirement is covered by tests.

This paper describes the "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: Sect. 2 summarizes related work on requirements traceability and approaches on deriving requirements and test scenario relations. Section 3 presents research questions while Sect. 4 depicts a typical use case. Section 5 describes the ReTeCo approach. The feasibility and the initial evaluation results of the prototype implementation are illustrated in Sect. 6 and discussed in Sect. 7. Finally, Sect. 8 draws conclusions and pictures future work.

2 Related Work

Traceability is the ability to follow the changes of software artifacts created during software development [28,36] and is described by the links that map related artifacts [32]. This section summarizes related work on various methods and approaches on requirements traceability, requirements coverage by means of test scenarios, and traceability between test cases and source code.

2.1 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 [24]. Similar approaches like [34] 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 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.

2.2 Information Retrieval

In the information retrieval area, there are various models which provide practical solutions for semi-automatically recovering traceability links [4]. 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 [12] 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. Some variations of the method are depicted in the following paragraphs.

The vector space model (VSM)[4] represents all artifacts as vectors which contain the word occurrences of all vocabulary terms. The cosine of the angle between two corresponding vectors is used as the similarity measure of two artifacts. In the probabilistic model (PM)[4] the similarity score is represented as the probability that a target artifact is related to a source artifact. It was shown that similar results are achieved when preliminary morphological analysis (e.g., stemming) are performed, regardless of the used model.

The Latent Semantic Indexing (LSI)[31] extends VSM by additionally capturing semantic relations of words like synonymy (words with equivalent meaning) and polysemy (word with multiple meanings) [39]. Performance evaluations show that LSI is able to compete with the VSM and PM methods, with the advantage of not being dependent on preliminary morphological analysis. This is especially useful for languages with complex grammar (e.g., German, Italian), for which stemming is not a trivial task [29].

The problem of vocabulary mismatch occurs because IR methods assume that a consistent terminology is used, which means, that concepts are described with the same words throughout the project. However, during a projects lifecycle the used terminology usually gets inconsistent and related artifacts are not recognized anymore. This leads to declining tracing accuracy. [30] evaluates the natural language semantic methods VSM with thesaurus support (VSM-T) [25], Part-of-Speech-enabled VSM (VSM-POS) [8], LSI [13], latent Dirichlet allocation (LDA) [5], explicit semantic analysis (ESA)[18] and normalized Google distance (NGD) [9]. The authors compare the methods to the basic VSM and show that explicit semantic methods (NGD, ESA, VSM-POS, VSM-T) provide better results than latent methods (LSI, LDA) in terms of precision. On the other hand, latent methods achieve higher recall values.

In [19] a way to check requirements-to-code candidate/trace links automatically is suggested by utilizing code patterns of calling relationships. It is based on that methods or functions in the source code that implement the same requirement are related as caller and callee. As a consequence, an expected trace for a method or a function can be computed by examining neighboring caller/callee methods and their associated requirements. Invalid candidate/trace links can be detected by comparing the expected trace with the actual trace.

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 [29].

2.3 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 [1] informal requirements are restructured by means of the Systems Modelling Language (SysML) [15] 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 [17] 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 [23] or [43]. The former links Use Case Diagrams and the corresponding code with the help of machine learning. Developers have to establish just about 6% trace links initially. After that the machine learning algorithm can automatically link the most of the remaining ones. The latter describes a special method for model driven development in the web engineering domain. The requirements are expressed as XML and translated by the means of XSL-transformation to an OOWS navigational model. Afterwards the links are extracted and requirement coverage is measured.

In the application of formal specifications/models requirement-, architecture- and design models are expressed for example as linear temporal logic [21], in Z-notation [40] or B-Notation [6]. The generation and/or validation of the trace links can be automated through a model checker [21], a rule based checker [40] or model based testing [6]. Further research work has been invested in techniques like annotating code, design elements or tests with traceability information [2, 14, 33], scenario-based techniques [16], graph-based techniques [7], or techniques in the context of test cases [38] relying on naming conventions, explicit fixture declarations, static test call graphs, run time traces and lexical analysis, and co-evolution logs.

3 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 [35] 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 [37] 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 Sect. 2) we have formulated the following research questions:

RQ1: how to make use of links between information units provided by engineering tools for the establishment of traceability links between requirements and test cases?
RQ2: to what extent do interlinked information in engineering tools allow more fine-grained reporting on coverage?
RQ3: to what degree is an automated process for coverage calculation achievable or still require human intervention?

To answer these research issues we designed the "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.

4 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 (i.e. issues) (Fig. 1, 2), each having a unique identifier (i.e. 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 team member is able to look up the details of the requirement in case of unclarities.

In principle, 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

[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.

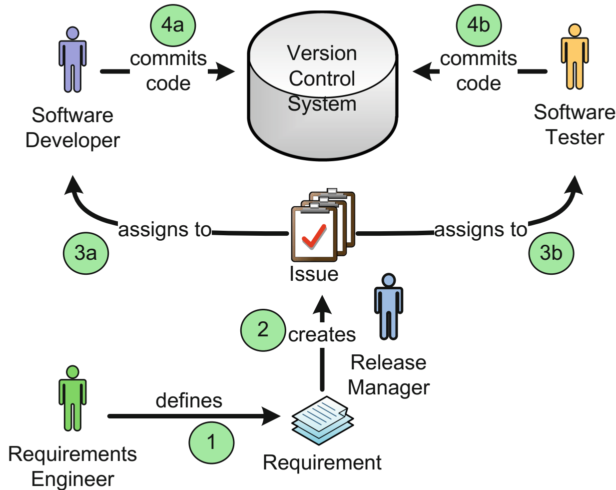

Fig. 1. Intertwining processes of various stakeholders in a software engineering environment

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?

5 Solution Approach

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

5.1 Traceability Link Model

The "Requirements-Testing-Coverage" (ReTeCo) approach calculates the coverage of a requirement as the degree to which the source code of (i.e. 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.

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

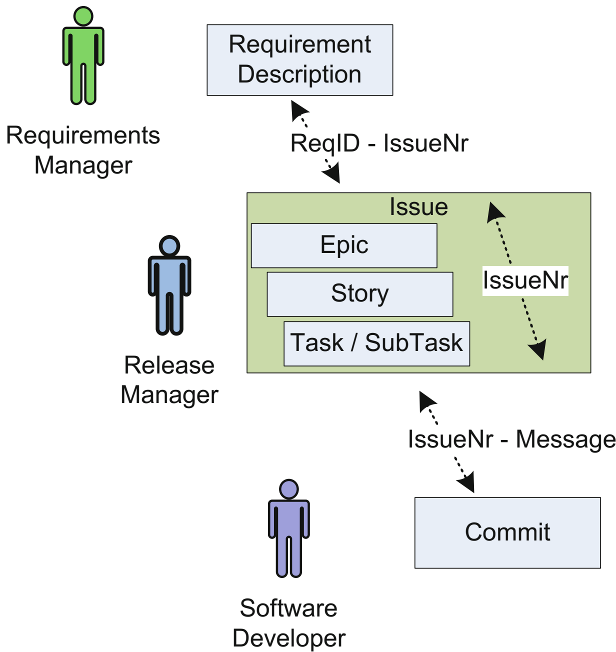

Fig. 2. Tool-supported linking of model elements define implicit traceability links

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 [11] 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.

5.2 Selection of Code Coverage Tool

The ReTeCo approach relies on the analysis of coverage reports created by coverage tools. However, there are a variety of code coverage tools which differ in their capabilities and weaknesses. For the prototype implementation it is therefore necessary to set up a set of selection criteria (see also Table 1):

SC1 - Test Relation Support: In order to be able to calculate the percentage of passed and failed tests of test covered lines, it is important to have a relation model between test scenarios and source code lines. Is a coverage tool capable of providing such information? **SC2 - Export Capability:** The ReTeCo approach aimed to automatically analyze the code coverage report and extract information from the report. Is the resulting coverage report usable for further automated processing? **SC3 - License:** For an open source prototype implementation following APLv2 the compatibility of the inspected tool's license has to be checked. **SC4 - Build Process Support:** In order to increase the flexibility of the approach, it needs to be checked to what extent the tool may be embedded into a build process systems.

Table 1. Results of the criteria investigation.

Criteria & Tools	Test relation support	Export capability	License	Build process support
JaCoCo	No	HTML, XML, CSV	EPL	Maven
EclEmma	No	HTML, XML, CSV	EPL	No
Cobertura	No	HTML, XML, CSV	Free, GNU GPL	Maven
SonarQube	No	No	LGPL v3	Maven, Ant, Gradle

As shown in Table 1, there is no tool that is able to provide a relation model between tests and source code lines. For the prototype it has been decided to go with JaCoCo as it is widely used and has a solid documentation. This also means that the calculation of the relation had to be implemented explicitly.

5.3 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

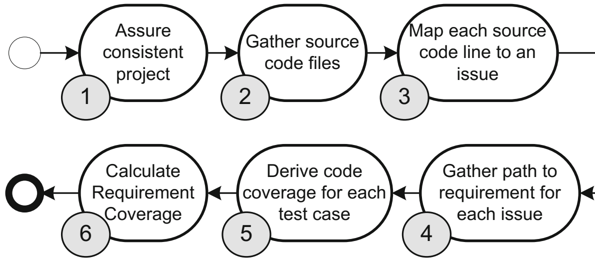

Fig. 3. Main process steps of the ReTeCo approach

that line. In the ReTeCo prototype this is done by parsing the commit (i.e. 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. Under the circumstance that there is a hierarchy of issues, 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 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

[9] https://git-scm.com/.
[10] https://eclipse.org/jgit/.
[11] https://maven.apache.org/.
[12] http://maven.apache.org/surefire/maven-surefire-plugin/.

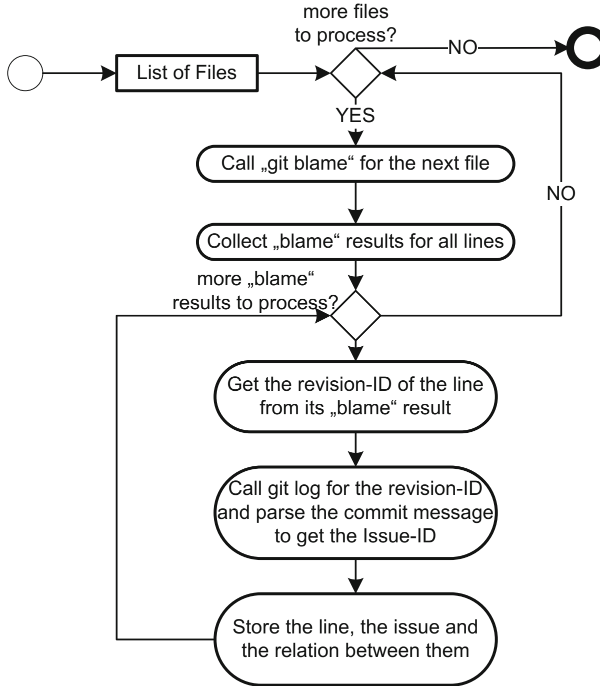

Fig. 4. Process steps for calculating issue-code relation

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 Figs. 5 and 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.

[13] http://www.eclemma.org/jacoco/.

6 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(see footnote 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 2 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 2,7 GHz and 8 GB RAM running on a Windows 7 environment. The prototype was implemented in java and compiled with Java 8. It uses JGit(see footnote 10) v4.4 to execute git commands, Maven Invoker[16] v2.2 in order to execute maven goals on target projects, JaCoCo(see footnote 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 2. Characteristics of investigated open source projects

Project	# of source code lines	# of test cases	# of issues	log size
org.ops4j.pax.web	82705	63	1229	3979
org.apache.qpid.qpid-java-build	481112	1775	1684	7768

Table 2 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, as shown in Table 3. 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 (i.e. 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.

Figure 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 (i.e. tests pass) and 1 line (16,67%) negatively (i.e. tests fail). The rest 3 lines (50%) are not covered by tests at all.

[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/.

Table 3. Evaluation results of investigated open source projects

Project	Execution time	Memory consumption
org.ops4j.pax.web	1 h 15 min	177 M
org.apache.qpid.qpid-java-build	2d 09 h 34 min	207 M

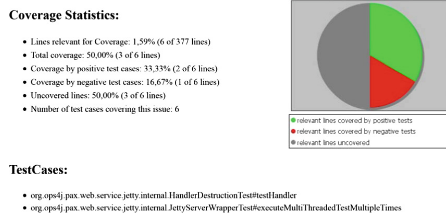

Fig. 5. Excerpt of the coverage report in the context of an issue

Figure 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.

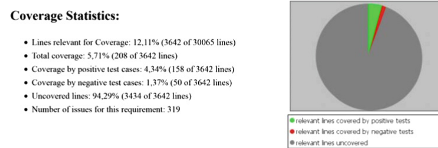

Fig. 6. Excerpt of the coverage report in the context of a requirement

7 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 (i.e. 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 Sect. 2 some of these tasks can be automated (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 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 parameterization. This implies that there is a large number of commits and issues related to the first requirement while only little for 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.

The approach assumes that changes to the code base are mainly introduced in the context of an issue which was created at some point in the software engineering life cycle and represents a part of a requirement - independent of the size of the change. Consequently, the approach "assumes" that if source code lines were removed within a commit they were removed as a result of the issue's intention. For example, the removed lines may have contributed to a misbehaviour of the system (i.e. bug) and the issue's intention was to fix it.

However, there are also changes which are committed to the code base without an Issue-ID. From the authors' experience the handling of such commits is a subject of discussions among project members. An example may be the release of the software system which requires the incremental of versions numbers. Commits without a reference to an issue will be ignored by the approach.

8 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 "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, (b) improving precision of the approach by considering dependencies between requirements, and (c) developing methods for handling cross-cutting aspects (e.g., quality requirements) that are related to many source code elements. From the prototype implementation point of view we intend to develop one that updates its indices incrementally on per-commit basis so that it can be embedded in a Continuous Integration and Testing life cycle.

References

1. Abbors, F., Truscan, D., Lilius, J.: Tracing requirements in a model-based testing approach. In: 2009 First International Conference on Advances in System Testing and Validation Lifecycle, pp. 123–128, September 2009
2. Ahn, S., Chong, K.: A feature-oriented requirements tracing method: a study of cost-benefit analysis. In: 2006 International Conference on Hybrid Information Technology, vol. 2, pp. 611–616, November 2006
3. Ali, N., Guhneuc, Y.G., Antoniol, G.: Trustrace: mining software repositories to improve the accuracy of requirement traceability links. IEEE Trans. Softw. Eng. **39**(5), 725–741 (2013)
4. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Trans. Softw. Eng. **28**(10), 970–983 (2002). https://doi.org/10.1109/TSE.2002.1041053
5. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003). http://dl.acm.org/citation.cfm?id=944919.944937
6. Bouquet, F., Jaffuel, E., Legeard, B., Peureux, F., Utting, M.: Requirements traceability in automated test generation: application to smart card software validation. SIGSOFT Softw. Eng. Notes **30**(4), 1–7 (2005). https://doi.org/10.1145/1082983.1083282
7. Burgstaller, B., Egyed, A.: Understanding where requirements are implemented. In: 2010 IEEE International Conference on Software Maintenance, pp. 1–5, September 2010
8. Capobianco, G., Lucia, A.D., Oliveto, R., Panichella, A., Panichella, S.: Improving IR-based traceability recovery via noun-based indexing of software artifacts. J. Softw. Evol. Process **25**(7), 743–762 (2013). https://doi.org/10.1002/smr.1564
9. Cilibrasi, R.L., Vitanyi, P.M.B.: The Google similarity distance. IEEE Trans. Knowl. Data Eng. **19**(3), 370–383 (2007). https://doi.org/10.1109/TKDE.2007.48
10. Cleland-Huang, J., Chang, C.K., Christensen, M.: Event-based traceability for managing evolutionary change. IEEE Trans. Softw. Eng. **29**(9), 796–810 (2003)
11. Cockburn, A.: Agile Software Development. Addison-Wesley Longman Publishing Co. Inc., Boston (2002)
12. De Lucia, A., Oliveto, R., Tortora, G.: Assessing IR-based traceability recovery tools through controlled experiments. Empir. Softw. Eng. **14**(1), 57–92 (2009). https://doi.org/10.1007/s10664-008-9090-8
13. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. **41**(6), 391–407 (1990). https://doi.org/10.1002/(SICI)1097-4571(199009)41:6⟨391::AID-ASI1⟩3.0.CO;2-9
14. Delgado, S.: Next-generation techniques for tracking design requirements coverage in automatic test software development. In: 2006 IEEE Autotestcon, pp. 806–812, September 2006
15. Delligatti, L.: SysML Distilled: A Brief Guide to the Systems Modeling Language, 1st edn. Addison-Wesley Professional, Boston (2013)
16. Egyed, A., Grunbacher, P.: Automating requirements traceability: beyond the record replay paradigm. In: Proceedings of 17th IEEE International Conference on Automated Software Engineering, pp. 163–171 (2002)
17. Fockel, M., Holtmann, J., Meyer, J.: Semi-automatic establishment and maintenance of valid traceability in automotive development processes. In: 2012 Second International Workshop on Software Engineering for Embedded Systems (SEES), pp. 37–43, June 2012

18. Gabrilovich, E., Markovitch, S.: Computing semantic relatedness using wikipedia-based explicit semantic analysis. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI 2007, pp. 1606–1611. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007). http://dl.acm.org/citation.cfm?id=1625275.1625535
19. Ghabi, A., Egyed, A.: Code patterns for automatically validating requirements-to-code traces. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 200–209, September 2012
20. Gittens, M., Romanufa, K., Godwin, D., Racicot, J.: 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, CASCON 2006. IBM Corp., Riverton, NJ, USA (2006). http://dx.doi.org/10.1145/1188966.1188981
21. Goknil, A., Kurtev, I., Van Den Berg, K.: Generation and validation of traces between requirements and architecture based on formal trace semantics. J. Syst. Softw. **88**, 112–137 (2014). https://doi.org/10.1016/j.jss.2013.10.006
22. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proceedings of IEEE International Conference on Requirements Engineering, pp. 94–101, April 1994
23. Grechanik, M., McKinley, K.S., Perry, D.E.: 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, ESEC-FSE 2007, pp. 95–104. ACM, New York (2007). http://doi.acm.org/10.1145/1287624.1287640
24. Guo, J., Monaikul, N., Cleland-Huang, J.: Trace links explained: an automated approach for generating rationales. In: 2015 IEEE 23rd International Requirements Engineering Conference (RE), pp. 202–207, August 2015
25. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans. Softw. Eng. **32**(1), 4–19 (2006). https://doi.org/10.1109/TSE.2006.3
26. Heindl, M., Biffl, S.: 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, ESEC/FSE-13, pp. 60–69. ACM, New York (2005). http://doi.acm.org/10.1145/1081706.1081717
27. Krishnamoorthi, R., Mary, S.S.A.: Factor oriented requirement coverage based system test case prioritization of new and regression test cases. Inf. Softw. Technol. **51**(4), 799–808 (2009). http://www.sciencedirect.com/science/article/pii/S0950584908001286
28. Lago, P., Muccini, H., van Vliet, H.: A scoped approach to traceability management. J. Syst. Softw. **82**(1), 168–182 (2009). https://doi.org/10.1016/j.jss.2008.08.026
29. Lucia, A.D., Fasano, F., Oliveto, R., Tortora, G.: Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. **16**(4) 2007. http://doi.acm.org/10.1145/1276933.1276934
30. Mahmoud, A., Niu, N.: On the role of semantics in automated requirements tracing. Requir. Eng. **20**(3), 281–300 (2015). https://doi.org/10.1007/s00766-013-0199-y
31. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: Proceedings of the 25th International

Conference on Software Engineering, ICSE 2003, pp. 125–135. IEEE Computer Society, Washington, DC, USA (2003). http://dl.acm.org/citation.cfm?id=776816.776832

32. Maro, S., Anjorin, A., Wohlrab, R., Steghfer, J.P.: Traceability maintenance: factors and guidelines. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 414–425, September 2016
33. Ni, D.C., Martinez, J., Eccles, J., Thomas, D., Lai, P.K.M.: Process automation with enumeration and traceability tools. In: Proceedings of the IEEE International Conference on Industrial Technology 1994, pp. 361–365, December 1994
34. Ooi, S.M., Lim, R., Lim, C.C.: An integrated system for end-to-end traceability and requirements test coverage. In: 2014 IEEE 5th International Conference on Software Engineering and Service Science, pp. 45–48 (June 2014)
35. Ortu, M., Destefanis, G., Kassab, M., Marchesi, M.: Measuring and understanding the effectiveness of JIRA developers communities. In: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics, pp. 3–10, May 2015
36. Parizi, R.M., Lee, S.P., Dabbagh, M.: Achievements and challenges in state-of-the-art software traceability between test and code artifacts. IEEE Trans. Reliab. **63**(4), 913–926 (2014)
37. Portillo-Rodrguez, J., Vizcano, A., Piattini, M., Beecham, S.: Tools used in global software engineering: a systematic mapping review. Inf. Softw. Technol. **54**(7), 663–685 (2012). http://www.sciencedirect.com/science/article/pii/S0950584912000493
38. Rompaey, B.V., Demeyer, S.: Establishing traceability links between unit test cases and units under test. In: 2009 13th European Conference on Software Maintenance and Reengineering, pp. 209–218, March 2009
39. Rosario, B.: Latent semantic indexing: an overview (2000). http://www.cse.msu.edu/cse960/Papers/LSI/LSI.pdf
40. Sengupta, S., Kanjilal, A., Bhattacharya, S.: Requirement traceability in software development process: an empirical approach. In: 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, pp. 105–111, June 2008
41. Stanbridge, C.: Retrospective requirement analysis using code coverage of GUI driven system tests. In: 2010 18th IEEE International Requirements Engineering Conference, pp. 411–412, September 2010
42. Tahat, L.H., Vaysburg, B., Korel, B., Bader, A.J.: Requirement-based automated black-box test generation. In: 25th Annual International Computer Software and Applications Conference, COMPSAC 2001, pp. 489–495 (2001)
43. Valderas, P., Pelechano, V.: Introducing requirements traceability support in model-driven development of web applications. Inf. Softw. Technol. **51**(4), 749–768 (2009). https://doi.org/10.1016/j.infsof.2008.09.008