Towards Visualization of Unit Test and Source Code Relations

By Nadera Aljawabrah

A Thesis:

Submitted To The Phd School In Computer Science Of The University Of Szeged In Partial Fulfilment Of The Requirements For The Degree Of
Doctor of Philosophy

Supervisor: Dr.Tamás Gergely

Szeged,2020

1 Introduction

Source code evolves very often. Nevertheless, test cases that examine it are not updated; maintaining consistency and traceability information between unit tests and source code is costly and time-consuming. As well as testing is frequently neglected due to the pressure of time to move on to the next change of a software or due to market. Furthermore, system developers or people who maintained it may no longer available due to turnover or outsourcing. Test-to-code traceability is the ability to relate the test unit and source code artifacts created during the software development life cycle (SDLC).

Traceability of test and code relations is fundamental to support various activities of software development such as program comprehension, verification and validation, impact analysis, reuse, maintenance, and software evolution. Notwithstanding its importance, many significant challenges are still associated with traceability. One of these challenges is is how to support the comprehension and maintenance of these links efficiently and effectively? Visualization is an important aspect of traceability as it aids developers to understand test-to-code relations, verify the quality of trace links, identify the disagreement between traceability links inferred from different sources, understand which code modules are tested by which unit tests, and lessen bugs while updating the existing features of a piece of software or adding new features to it.

According to [16], in practice, the development of test-to-code traceability links is not sufficiently managed in literature. Recent research on test-to-code traceability links is directed on how to identify the links between test and code, as well as only a few specific approaches that have been suggested and used to recover test-to-code traceability links. Most of these approaches have come from the outstanding work [23]. In recent years, research into combining test-to-code traceability links recovery approaches has become very popular [18], [12], [2]. It helps to improve the quality and accuracy of the retrieved link. However, it is is complex, time-consuming, and susceptible to error task to manually retrieve the traceability links. This effort can be notably diminished by automatically establish and retrieve the links between unit test and unit under test, as well as, adopting visualization techniques to present these links in a simple and intuitive way [20].

Several publications have appeared [8], [15], [9] documenting the visualization of traceability links among different software artifacts (e.g. requirements, source codes, documents, etc.). Furthermore, many visualization tools have been designed to represent

traces between software artifacts in different views [9],[11]. However, as yet, no visualization method focused on test-code traceability links, neither tools were implemented with this focus., which in turn, lays the foundation of our work.
In this thesis, we focus on the specific problem of automatically recovering and visualizing the traceability links between test cases and the related production classes. We provide an innovative visualization test-to-code traceability approach that combines various test-to-code recovery approaches and support efficient visualization technique. We developed a tool, called TCTracVis, that supports the automated recovery approaches and the simultaneous visualizations of test and code relations. The main idea of using the traceability recover approaches is to help software engineers to trace the relationships between unit test and source code, and automatically extract traceability links at low cost and time. The goal of using visualization is to identify the disagreement between traceability links inferred from different sources [3]. This might point out places where something is wrong with the tests and/or the code (at least their relationship) in a specific system.
The key points discussed in this work are as follows.

- A comprehensive overview investigating existence research on the traceability between tests and code.
- A visualization method that visually presenting the test-to-code traceability links alongside the three traceability recovery methods to retrieve these links. the visualization method is implemented using TCTracVis visualization tool.
- Evaluation of the presented approach and tool support in terms of usability and efficiency throughout an empirical study.

2 Traceability links in Literature

The IEEE standard glossary of software engineering of terminology [1] defines traceability as "The degree to which a relationship can be established between two or more products of the development process". There are two types of traceability as mentioned in [21]: 1) traceability between software artifacts at the same level of software life-cycle which is known as vertical traceability (e.g. traceability between requirement components). 2) and traceability between software artifacts at different levels of software life-

cycle which is known as horizontal traceability (e.g. traceability between source code and test cases).

One of the most common terms in software traceability is requirements traceability. In requirements engineering, the term traceability is explicitly related to requirements. Requirements traceability is defined by [11] as "The ability to describe and follow the life of a requirement in both a forwards and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases)". This definition is first and most widely accepted and used in the context of requirements traceability. It discusses the artifacts refinements and iterations. However, it is explicitly oriented to requirements and to follow a requirement's life, but nothing is mentioned about the use of traceability.

In [4], we proposed a model that defines the requirements elicitation process. The model focused on the improvement of the requirements quality by applying the requirements tracking and refinement. The aim of tracking requirements is to allocate each requirement to a stakeholder or a user who requests it. Each phase is useful in ensuring the satisfaction of the users and will fulfill the requirements as needed.

The author in [14] defined system traceability as "The ability to relate uniquely identifiable system engineering artifacts created and evolved during the development of a system, maintain these relationships throughout the development life cycle and use them to facilitate system development activities". The artifacts in system development in this case include all artifacts that are related to the system. Based on this definition, we can deduce that three aspects of traceability should be taken into account during system development: identify the artifacts involved, maintain the links between them, and use these links to facilitate the activities in the development process.

2.1 Test-to-Code Traceability Links

Test-to-code relations can be treated as traceability links that display how test cases and the code under test can be connected. Test suites are usually used to evaluate software systems and detect the program faults. The larger the programs are, the larger the test cases executed, thus these links emphasize the consistency between unit test and tested code (e.g. when a test case fails, the links show which part of the code is related to this failure). Test cases and tested code can be connected by different types of relations, for

example,

- **Direct tests**. When developers produce test classes that only test their counterparts in the production classes [22].
- **Indirect tests**. When developers produce test classes contain methods that actually execute tests on other [22].

Current research on test-to-code traceability links is focused on how to retrieve the links between test and code [18],[23], [17]. Rompaey and Demeyer[23] have compared six traceability recovery strategies in terms of the applicability and the accuracy of each approach. The comparison covers only those approaches relating to requirement traceability and test-to-code traceability. The strategies have been evaluated based on three open-source Java programs. In these approaches, units under test are identified by matching test cases and production code's names, examining method invocation in test cases, looking at the last calls right before assert statements, and capturing changes on the test cases and the production code in the version control change log. The results show that last call before assert, lexical analysis and co-evolution have high applicability; however, they have low accuracy. While naming convention and fixture element types showed high precision and recall. The best results are provided by combining the high-applicability strategies with the high-accuracy ones. However, none of the proposed approaches provides tool support to facilitate traceability links, as well as, none of these approaches support visualization of traceability links [16] . Moreover, there is no single technique that is superior to all others.

3 Visualization of Traceability Links

In recent years, research on visualization of traceability links has become very popular. A great effort has devoted to the use of visualization techniques to help users to understand and analyze traceability information. Visualization techniques depict links between software artifacts due to the context to accomplish a task. Visualization techniques and tools have been developed depending on the type of traceability information being visualized and the objectives of visualization. For example, to understand the dependencies and relationships between software artifacts, how they interact with each other, and help document links between several kinds of software artifacts (e.g. requirements, tests) [13]. In

Table 1, a set of traceability approaches and tools are listed. Each approach provides one or more visualization techniques which may display links in different ways depending on the information task context. In the domain of test and code relations, Visualization is an

Table 1: Traceability Links Visualization Techniques

Approach	Visualization technique	Traceability information	Tool Support
[8]	Graph	Links between software artifacts	ADAMS
[6]	Hierarchical graphical structure	Requirements information	
[11]	Graph	Requirements relationships	ChainGraph
[15]	Sunburst and Netmap	Elements of requirements knowledge	
[19]	Sunburst, matrix, tree, graph	Links between software artifacts	Multi-Viso
[10]	Sunburst, tree, matrix, list, table, bar, gauge, radial view	Links between software artifacts	D3TraceView
[24]	Colored squares	Links between software artifacts	TraceVis
[7]	Text	Links between software artifacts	Poirot
[5]	TreeMap and hierarchical tree	Links between source code and documentation	DCTracVis

approach that effectively supports understanding test-to-code relations and helps various tasks in the software development life cycle (SDLC). It can provide a detailed description of how tests and tested code are connected. It can improve the maintenance of test and code links by reducing the submitted effort in understanding software . However, the lack of visualization support, one of the main challenges in the current of test-to-code traceability recovery approaches and tools.

4 Visualization of Trace Approach

As there are many sources from where the traceability relations can be inferred, one of the most important questions is to decide which source or combination of sources is the best

to determine the test-to-code links. It is obvious that, if these sources disagree, this will make it harder to understand what is going on, what was the goal of the developer, how the components are really related, and change impact analysis can yield in false results. Fortunately, visualization can aid this task.

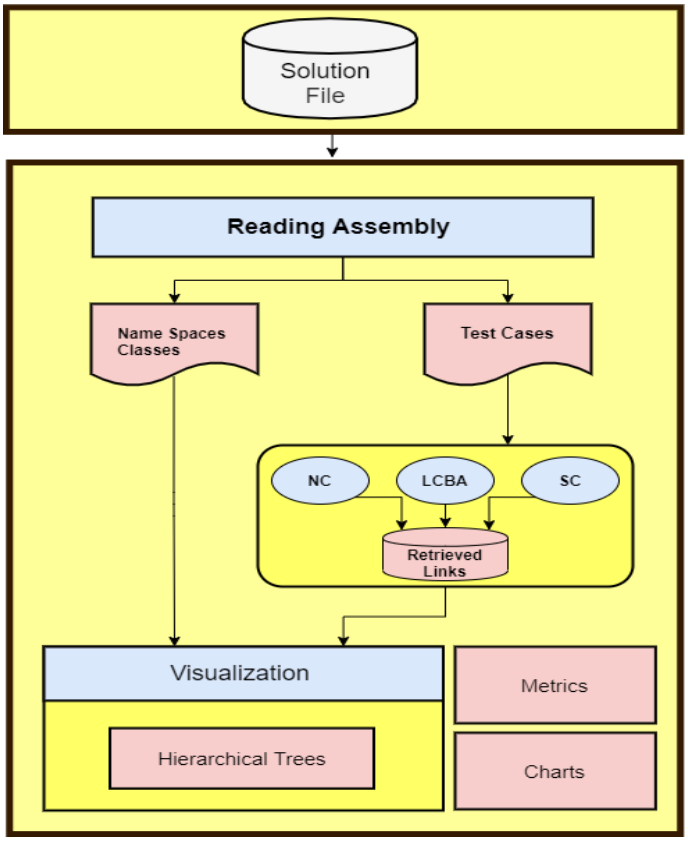

Figure 1: Architecture of the TCTracVis

Our approach consists of three parts. The first part consists of artifacts from two different areas, source code and units test.The second part presents different sources for capturing the links between code and test. The third part includes the visualization method used to visually present the traceability links inferred from traceability links sources in two levels, class-level and method-level. The trace visualization approach is implemented as a trace visualization tool, called TCTracVis. Figure 1 illustrates the traceability visualization process of our approach.

In our implementation tool, we combine three traceability recovery techniques to retrieve links between unit's test and tested code and display the captured links using a hierar-

chy tree graph visualization technique. These techniques are: Naming convention, Last Call Before Assert (LCBA), and Static Call Graph (SCG). Test-to-code traceability links are recovered automatically in TCtracVis using these approaches according to which approach the user selects for recovery.

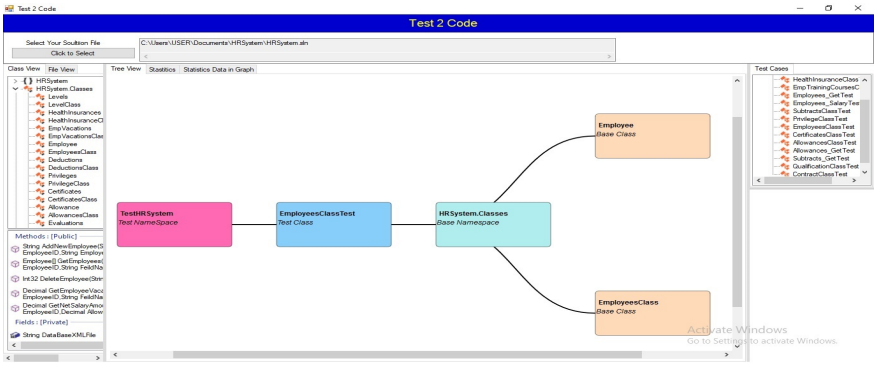

Figure 2: Traceability Links of *EmployeeClassTest* Test Class

In the following, we provide examples of the visualization of test-to-code traceability links for *issue-registerTest* test case from UnitTestExample solution using NC, LCBA, and SCG respectively.

Figures 2 and 3 illustrate recovering test and code links using NC method. A test class is connected to a base class by matching their names. This approach supports recovering links at class-level. In this level, the traceability links have the advantages of being bidirectional. Thus, links can be visualized in two directions:

- **TC-to-CUT**. The test class is selected, then all related tested classes, which a selected test class was written to evaluate, are displayed. Figure 2 shows an example of TC-to-CUT visualization. It can be seen when *EmployeeClassTest* test class is selected, all the classes under the test *EmployeeClass* and *Employee* are shown. The visualization can also show the namespaces where the TC and CUT belong to.

- **CUT-to-TC**. In this view, the tested class is selected, then all test classes which evaluate CUT are displayed with namespaces that belong to. In Figure 3, *EmployeeClass* is a tested class which is evaluated by three test classes, *Employee-GetTest*, *Employee-SalaryTest*, and *EmployeeClassTest*.

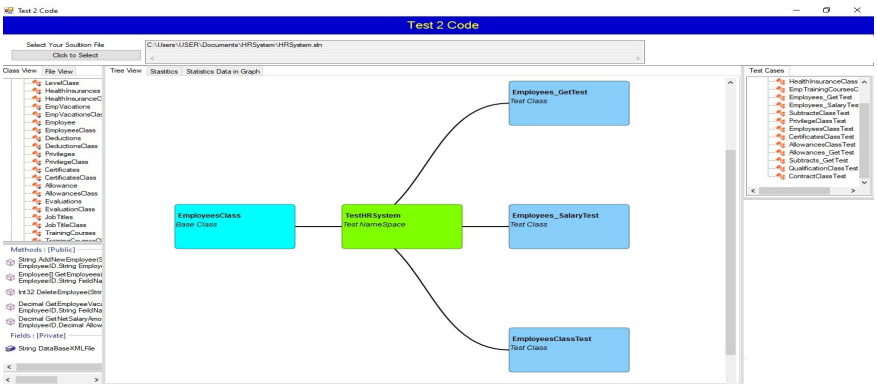

Figure 3: Traceability Links of *EmployeeClass*(CUT) Class

Figure 4 provides visualizing the traceability links of *issue-registerTest* test class using last call before assert strategy. The figure shows a set of tested classes which are called by *issue-registerTest* test class in the statements performed right before assert statements in *issuebookTest* test method. Finally, static call graph strategy is used in Figure 5 to establish the links of *issue-registerTest* test class. The visualization shows the production classes that are invoked most in the implementation of test class and the number of times they are called in each test method.

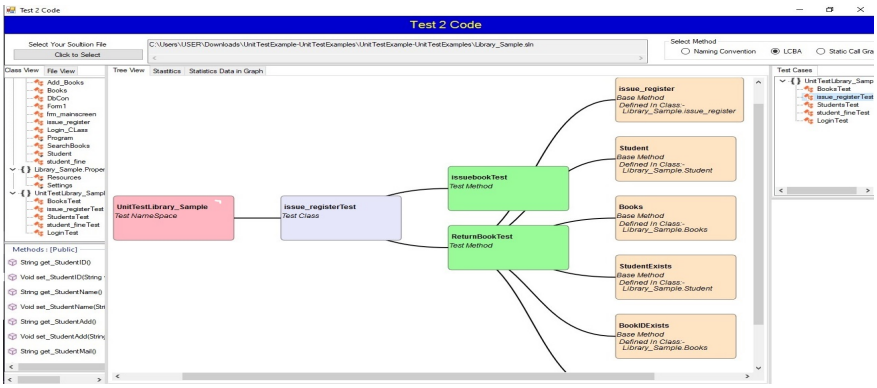

Figure 4: Traceability links of issue-registerTest using LCBA

As shown in the previous figures, we can see that test-to-code traceability links from different sources are displayed which, in turn, provide a clearer picture of what is taking place within these tests. Furthermore, a hierarchical tree view presents a detailed overview of traceability links at method-level specifically with SCG and LCBA approaches.

Further features in TCTracVis involve some metrics about the traced solution (e.g. no. of base classes, no. of test classes, no. of classes not tested), these metrics can be visually displayed using several bar charts. The metrics provide a quick overview of artifacts of the traced solution, which can help better to extract valuable information with less effort.

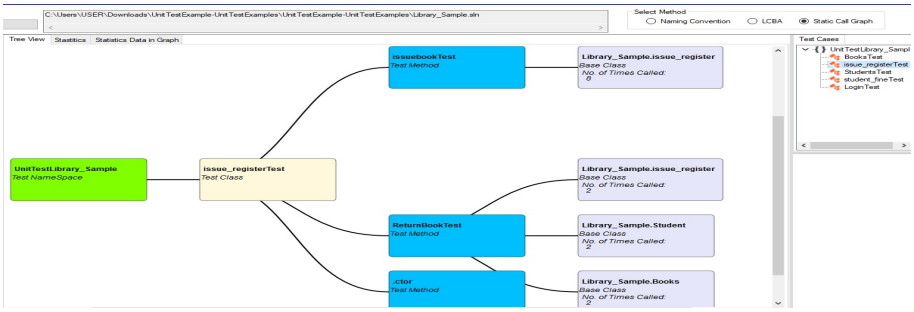

Figure 5: Traceability links of issue-registerTest using SCG

5 Evaluation Of Visualization Approach

We conducted a usability study to develop an understanding of the effectiveness and usability of our visualization traceability approach to justify the effort and time spent in the design of the TCTracVis tool,as well as, to testify to the TCTracvis visualization tool. We used in the study UnitTestExample [1] solution, which is a c# open source windows forms application with main module functionality that is served by several small classes which are used in unit testing.

It is worth mentioning that our tool is robust to support large projects, however we selected the UnitTestExample as its small size makes the manual evaluation much easier for the participants. The usability study is undertaken to answer the following questions:

- Is the use of multiple-source links visualization better, for software engineers, developers, and testers to find solutions to their problems, than using a single-source-visualization?

- Does the use of TCTracVis tool help to enhance the overall browsing, comprehension, and maintenance of test-to-code traceability links of a system?.

[1] https://github.com/situ-pati/UnitTestExample

To answer the questions above, we defined a set of tasks to be performed using our visualization traceability tool. These tasks have been also performed manually to measure TCTracVis's added value to traditional software engineering processes in manual tracing. A group of 24 subjects with varying levels of expertise in software development and unit testing were assigned for the evaluation of our tool and for performing the tasks. Among the subjects were 17 students, (3) from industry, and (4) from academics. We divided the subjects into two groups: a control group and an experimental group. The former group is assigned to perform the tasks manually, while the latter group is assigned to perform the tasks using the TCTracVis tool. In the beginning, we provided the subjects with a brief introduction to help them to get familiar with our approach and tasks. After the tasks completion, a set of questions on our tool have been answered by them.

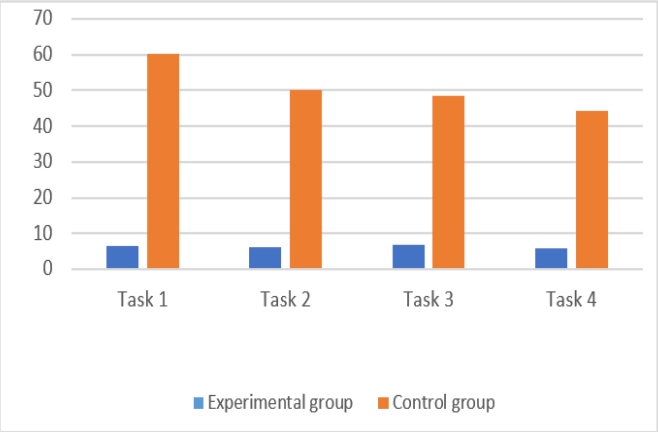

Figure 6: Average Time to Complete Tasks

As illustrated in Figure 6, the time taken to complete all tasks using our tool varying from (5) minutes to (10) minutes. While the same tasks completed manually with times varying from (45) minutes to (60) minutes. The time varies depending on the subjects' experience in software development and, for the experimental group, whether they often use traceability tools or not. In figure 7, it can be seen that the number of steps needed to perform the tasks by the control group is much more than the number of steps needed to perform the tasks by the experimental group. During the manual evaluation, subjects often switched between source code files and test case files to read, perform the task and write down the notes about the artifacts, the links, the time, and the number of steps to perform each task. Our tool can effectively provide all the required information in a single

view.

After the evaluation, the analysis of the main outcomes performed was on a number of questions the subjects answered based on their experience of using our tool. The main purpose of the questions is to assess the tasks performed by the subjects.

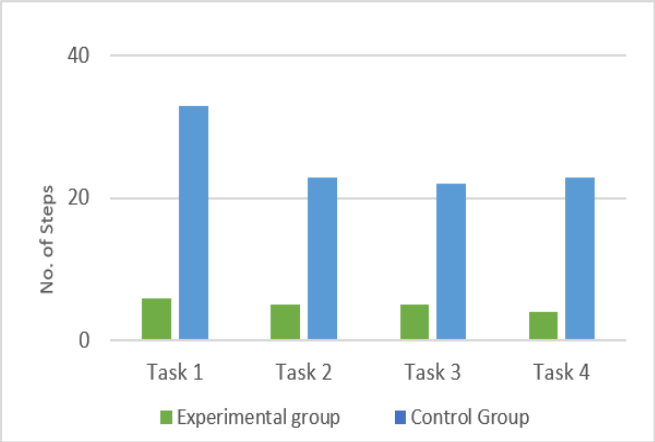

Figure 7: Number of Steps to Perform the Tasks

Overall, the results revealed that the participants strongly agreed that the visualization of traceability link inferred from different sources is more efficient and helpful than using a single source, and that, the results showed that the visualization tool can efficiently support understanding, browsing, and maintaining of a test to code traceability links in a software system.

6 Conclusion and Future Orientation

In this thesis, we focus on the specific problem of recovering and visualizing the traceability links between test cases and the related production classes. We provide an innovative approach for automatically capturing the traceability links between unit test and classes from multiple sources of links, and visualizing these captured links in order to help the testers and developers to get a bigger picture about what is going on with the tests and understand the relationships between test cases and the corresponding units under test. Our thesis consists of four main parts, namely background introduced in Chapter 2, state of the art in Chapter 3, the visualization approach in Chapter 4, and the evaluation of

visualization approach in Chapter 5.
Based on the results presented in this thesis, there are potential areas of future work as follows:

- Implement visualization traceability approach on other programming language.
- Support other types of traceability recovery approaches.
- Support an overall overview visualization of project.
- A more thorough analysis of the use of traceability links during development.

This thesis is based on the following publications:

1. Aljawabrah, Nadera, and Tamás Gergely. "Visualization of test-to-code relations to detect problems of unit tests." The 11th Conference of Phd Students in Computer Science. 2018.
2. Aljawabrah, Nadera, Támas Gergely, and Mohammad Kharabsheh. "Understanding Test-to-Code Traceability Links: The Need for a Better Visualizing Model." International Conference on Computational Science and Its Applications. Springer, Cham, 2019.
3. Aljawabrah, N., and Qusef, A. (2019, December). TCTracVis: test-to-code traceability links visualization tool. In Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems (pp. 1-4).
4. Nadera Aljawabrah, AbdAllah Qusef, Tamás Gergely, and Adhyatmananda Pati, Visualizing Multilevel Test-to-Code Relations. In 3rd International Conference on Information and Communication Technology and Applications. Springer (CCIS), 2020.
5. Nadera Aljawabrah, Tamás Gergely, Sanjay Misra, and Luis Fernandez-Sanz, Automated Recovery and Visualization of Test-to-Code (TCT) Links: An Evaluation, in the submission to IEEE Access.

References

[1] *Institute of Electrical and Electronics Engineers*. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[2] Jens Krinke [31] White, Robert and Raymond Tan. *"Establishing Multilevel Test-to-Code Teaceability Links*. In 42nd International Conference on Software Engineering (ICSE'20). ACM, 2020, 2020.

[3] Gergely T. Aljawabrah, N. and M. Kharabsheh. *Understanding Test-to-Code Traceability Links: The Need for a Better Visualizing Model*. In Inter-national Conference on Computational Science and Its Applicationspp (428-441). Springer, Cham., 2019.

[4] H. Bani-Salameh and N. Al jawabreh. *Towards a comprehensive survey of the requirements elicitation process improvements*. In Proceedings ofthe International Conference on Intelligent Information Processing, Securityand Advanced Communication. pp (1-6), 2015.

[5] Hosking J. Chen, X. and J. Grundy. *Visualizing trace-ability links between source code and documentation*. In 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). (pp.119-126). IEEE, 2012.

[6] Berenbach B. Clark S. Settimi R. Cleland-Huang, J. and E. Romanova. *Best practices for automated traceability*. Computer, 40(6), pp (27-35), 2007.

[7] J. Cleland-Huang and R. Habrat. *Visual support in automated tracing*. In Second International Workshop on Requirements Engineering Visual-ization (REV 2007). (pp. 4-4). IEEE., 2007.

[8] Fasano F. Oliveto R. De Lucia, A. and G. Tortora. *Adams re-trace: A traceability recovery tool*. In Ninth European Conference on SoftwareMaintenance and Reengineering. pp (32-41). IEEE., 2005.

[9] Malimpensa G. de Oliveira T. R. Olivatto G. andFabbri S. C. Di Thommazo, A. *Requirements traceability matrix: Automatic generationand visualization*. In 2012 26th Brazilian Symposium on Software Engineering.pp (101-110). IEEE, 2012.

[10] A. D. A. Gilberto Filho and A Zisman. *D3TraceView: A Traceability Visu-alization Tool*.

[11] Lohmann S. Lauenroth K. Heim, P. and J Ziegler. *Graph-based visualization of requirements relationships*. In 2008 Requirements Engineering Visualization.pp (51-55). IEEE, 2008.

[12] László Vidács Viktor Csuvik Ferenc Horváth Arpád Beszédes Kicsi, András and Ferenc Kocsis. *Supporting product line adoption by combining syntactic and textual feature extraction*. International Conference on Software Reuse, pp. 148-163. Springer, Cham, 2018.

[13] Muccini H. Lago, P. and H. Van Vliet. *A scoped approach to traceability management*. Journal of Systems and Software, 82(1), pp (168-182), 2009.

[14] S. Maro. *Addressing Traceability Challenges in the Development ofEmbedded Systems*. 2017.

[15] Jüppner D. Merten, T. and A. Delater. *Improved representationof traceability links in requirements engineering knowledge using Sunburstand Netmap visualizations*. In 2011 4th International Workshop on ManagingRequirements Knowledge. pp (17-21). IEEE., 2011.

[16] Sai Peck Lee Parizi, Reza Meimandi and Mohammad Dabbagh. *Achievements and challenges in state-of-the-art software traceability betweentest and code artifacts*. IEEE Transactions on Reliability63, no. 4, pp 913-926, 2014.

[17] A. Qusef. *Recovering test-to-code traceability via slicing and conceptual coupling*. In 2011 18thWorking Conference on Reverse Engineering (pp. 417-420). IEEE, 2011.

[18] Oliveto R. Qusef, A. and A. De Lucia. *Recovering traceability linksbetween unit tests and classes under test: An improved method*. In 2010 IEEEInternational Conference on Software Maintenance. pp (1-10). IEEE, 2010.

[19] Lencastre M. Rodrigues, A. and A. D. A. Gilberto Filho. *Multi-VisioTrace: traceability visualization tool*. In 2016 10th InternationalConference on the Quality of Information and Communications Technology(QUATIC). (pp. 61-66). IEEE, 2016.

[20] G. C. Roman and K. C. Cox. *Program visualization: The art of map-ping programs to pictures*. In Proceedings of the 14th international conferenceon Software engineering. pp (412-420), 1992.

[21] Pourya Nikfard S. B. I. Mohammad Hossein Abolghasem Zadeh . Mohammad Nazri Kama. *Software Changes*. Int. Conf. Adv. Comput. Netw -ACN 2013, 2013.

[22] Moonen L. Van Den Bergh A. Van Deursen, A. and G. Kok. *Refactoring test code*. In Proceedings of the 2nd international conference on extremeprogramming and flexible processes in software engineering (XP). pp (92-95), 2001.

[23] Van Rompaey, B. and Demeyer, S. *Establishing traceability links be-tween unit test cases and units under test*. In 2009 13th European Conferenceon Software Maintenance and Reengineering.pp 209-218. IEEE, 2009.

[24] Xie X. WMarcus, A. and D. Poshyvanyk. *When and how to visualize traceability links?* In Proceedings of the 3rd internationalworkshop on Traceability in emerging forms of software engineering. (pp. 56-61), 2005.