VR-TestCoverage: Test Coverage Visualization and Immersion in Virtual Reality

Roy Oberhauser[0000-0002-7606-8226]
Computer Science Dept.
Aalen University
Aalen, Germany
e-mail: roy.oberhauser@hs-aalen.de

***Abstract*—With the increasing pressure to deliver additional software functionality, software engineers and developers are often confronted with the dilemma of sufficient software testing. One aspect to avoid is test redundancy, and measuring test (or code or statement) coverage can help focus test development on those areas that are not yet sufficiently tested. As software projects grow, it can be difficult to visualize both the software product and the software testing area and their dependencies. This paper contributes VR-TestCoverage, a Virtual Reality (VR) solution concept for visualizing and interacting with test coverage, test results, and test dependency data in VR. Our VR implementation shows its feasibility. The evaluation results based on a case study show its support for three testing-related scenarios.**

Keywords – Software test coverage; code coverage; virtual reality; visualization; software testing.

I. INTRODUCTION

Source code portfolios can grow and become very large for both open-source projects, government organizations, and companies, as exemplified with the over 2 billion Lines of Code (LOC) accessed by 25k developers at Google [1]. There are estimated to be over 25m professional software developers worldwide [2] who continue to add source code to private and public repositories. One quality aspect to consider is how well this code is tested, and if any changes have been covered by tests. With large code bases, visualization of test coverage can provide insights.

Software testing is one important Knowledge Area (KA) within the Software Engineering Body Of Knowledge (SWEBOK) [3]. Both the SWEBOK and the international software testing standard ISO/IEC/IEEE 29119 [4] include test coverage measures within their test technique descriptions. Test effectiveness is always a challenging factor to measure. While test coverage (a.k.a. code coverage, in this paper we assume statement coverage) as a single factor may not be strongly correlated with test effectiveness [5], it nevertheless is still low to moderately correlated, and this can be helpful and supportive data for the test effort.

Considering the adoption rate of test coverage by software developers, for an insight into the industrial popularity of test coverage, of 512 developers randomly surveyed at Google in a 2019 survey [6], 45% indicated they use it (very) often when authoring a changelist and 25% sometimes. When reviewing a changelist, 40% use coverage (very) often and 28% sometimes. Only 10% of respondents never use coverage, which conversely means 90% do. So overall, a substantial number of developers apply code coverage regularly and find value in it. Voluntary adoption at the project level went from 20% in 2015 to over 90% by 2019. Yet, these relatively high reported rates in professional private companies may not correspondingly be found in smaller, less-professional companies or in voluntary development work, e.g., on open source projects. For instance, a survey of 102 open source Android app developers [7] reported that 64% did not use or did not consider code coverage useful for measuring test case quality. Some of the reasons mentioned include usability and the learning curve of available tools as well as the lack of knowledge of the tools and techniques. While testing has never typically been the forte of software developers, one challenge is how to motivate developers to test and measure test coverage, to leverage the utility and intuitive accessibility of testing data, to enhance the usability of testing tools and methods (especially for newcomers).

Virtual Reality (VR) is a mediated visual environment which is created and then experienced as telepresence by the perceiver. VR provides an unlimited immersive space for visualizing and analyzing in a 3D spatial structure viewable from different perspectives. Via its unique visualization and immersive capability for digital data, VR can play a part as a motivational factor for software testing and for depicting and utilizing test coverage data. By supporting tool-independent access to coverage data, usability is enhanced, and accessibility for all stakeholders is supported (including unfamiliar newcomers). In our view, an immersive VR experience can be beneficial for software analysis tasks such as testing coverage analysis. Müller et al. [8] compared VR vs. 2D for a software analysis task, finding that VR does not significantly decrease comprehension and analysis time nor significantly improve correctness (although fewer errors were made). While interaction time was less efficient, VR improved the user experience, was more motivating, less demanding, more inventive/innovative, and more clearly structured.

As software projects grow in size and complexity, an immersive digital environment can provide an additional visualization capability to comprehend and analyze both the software production code (i.e., test target) and the software test suite and how they relate, as well as determine areas where the code coverage achieved by the test suite is below expectations.

As to our prior work with VR for software engineering, VR-UML [9] provides VR-based visualization of Unified

Modeling Language (UML) and VR-SysML [10] for System Modeling Language (SysML) diagrams. VR-Git [11] provides VR-based visualization for Git repositories. This paper contributes VR-TestCoverage, a solution concept for visualizing and interacting with test coverage data in VR. Our prototype realization shows its feasibility, and a case-based evaluation provides insights into its capabilities.

The remainder of this paper is structured as follows: Section 2 discusses related work. In Section 3, the solution concept is described. Section 4 provides details about the realization. The evaluation is described in Section 5 and is followed by a conclusion.

II. Related Work

Our search found no other VR work directly addressing test coverage (or code coverage). VR-related work regarding software analysis includes VR City [12], which applies a 3D city metaphor. While it briefly mentions that its work might be used for test coverage, it shows no actual results in this regard and in this regard only a trace mode visualization is depicted.

Non-VR work on code coverage includes Dreef et al. [13], which applies a global overview test-matrix visualization. Rahmani et al. [14] incorporates JaCoCo to process coverage metrics and TRGeneration to visualize a control flow graph and assist the tester in determining the test input requirements to increase coverage. VIRTuM [15] is an IntelliJ JetBrains plugin that provides static and dynamic test-related metrics. Alemerien and Magel [16] list various coverage tools they assess in their study, determining that there is a wide range of differences in how the metrics are calculated. Open Code Coverage Framework (OCCF) [17] proposes a framework to unify code coverage across many programming languages.

In contrast, our solution is VR-based and not Integrated Development Environment (IDE)-specific, thus it can be flexibly used independently of any IDEs and tools (as long as any tool-generated coverage report is converted into the required import format). Rather than focusing on source code and control flow details, it provides an overall high-level coverage view of production code to help focus testing efforts on areas that are insufficiently tested.

III. Solution Concept

In Figure 1, the VR-TestCoverage solution concept is shown relative to our other VR solutions in the software engineering area. VR-TestCoverage utilizes our generalized VR Modeling Framework (VR-MF) (detailed in [18]). VR-MF provides a VR-based domain-independent hypermodeling framework addressing four aspects requiring special attention when modeling in VR: visualization, navigation, interaction, and data retrieval. Our VR-based solutions specific to Software Engineering (SE) include VR-TestCoverage (the focus of this paper) and the aforementioned VR-Git [11], VR-UML [9] and VR-SysML [10]. Since Enterprise Architecture (EA) can encompass SE models, development, and test aspects and thus be applicable for collaboration in VR, our other VR modeling solutions in the EA area include VR-EA [18], which visualizes EA ArchiMate models in VR; VR-ProcessMine [19] supports process mining and analysis in VR; and VR-BPMN [20] visualizes Business Process Modeling Notation (BPMN) models in VR. VR-EAT [1] integrates the EA Tool (EAT) Atlas to provide dynamically-generated EA diagrams in VR, while VR-EA+TCK [1] integrates Knowledge Management Systems (KMS) and/or Enterprise Content Management Systems (ECMS).

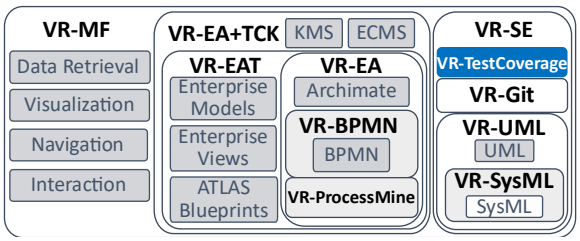

Figure 1. Conceptual map of our various VR solution concepts.

A. Visualization in VR

A plane is used to group the production code (test suite target) as well as the test suite. A tree map using a step pyramid paradigm (or mountain range) is used to stack containers (i.e., groups, collections, folders, directories, packages) in the third dimension (height) on the plane.

One visualization challenge we faced was that we initially thought we could depict the test target code by simply overlaying a layer on the production code and indicating which test "covered" what production code. However, once we completed the dependency analysis of large projects, we found an n-m relation between tests and the test targets, while one test may have a focus, it nevertheless may indirectly invoke many other dependent portions of the target. Thus, we chose to keep the visual depiction of the test suite separated from the test target (since it can have its own hierarchical organization), yet to use the same visualization paradigm to depict "containers" or collections as packages or folders. However, to retain the intuitive paradigm of "coverage," we elected to place the test suite visualization directly above the test target. That way, dependencies can be followed from top to bottom, and the test target should not depend on any test code. Since the most concrete tests are typically the smallest (greatest depth, leaves rather than containers), the test suite uses the opposite of height, rather depth, to bring these closer to the target. Dependencies are then shown as lines between the test and test target, analogous to puppet strings.

B. Navigation in VR

The space that can be traversed in VR can become quite large, whereas the physical space of the VR user may be constrained, e.g., to a desk. Thus, the left controller was used for controlling flight (moving the VR camera), while the right controller was used for interaction.

C. Interaction in VR

Since interaction in VR is not yet standardized, in our concept, user-element interaction is supported primarily through VR controllers and a *VR-Tablet*. The VR-Tablet is used to provide context-specific detailed element information. It includes a *virtual keyboard* for text entry via laser pointer key selection.

IV. Realization

To avoid redundancy, only realization aspects not explicitly mentioned in the evaluation section are described in this section. While the VR-TestCoverage solution concept is generic, for the realization of a prototype we focused on the .NET platform. The logical architecture for our VR-TestCoverage prototype realization is shown in Figure 2. Basic visualization, navigation, and interaction functionality in our VR prototype is implemented with Unity 2021.1 and the OpenVR XR Plugin 1.1.4, shown in the Unity block (top left, blue). The JSONUtility library is used for JSON processing.

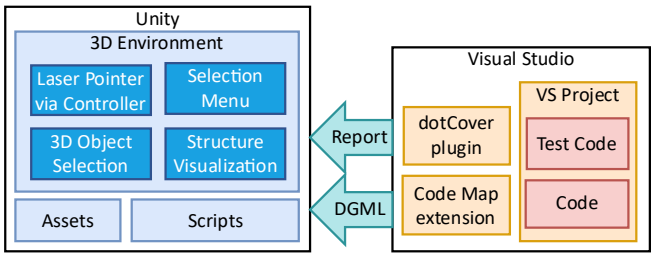

Figure 2. VR-TestCoverage logical architecture.


```
{
  "Children" : [
    {
      "DotCoverVersion": "2021.2.2",
      "Kind": "SolutionRoot",
      "CoveredStatements": 688,
      "TotalStatements": 4869,
      "CoveragePercent": 14,
      "Children": [
        {
          "Kind": "SolutionFolder",
          "Name": "APIs",
          "CoveredStatements": 202,
          "TotalStatements": 3728,
          "CoveragePercent": 5,
          "Children": [
            {
              "Kind": "Project",
              "Name": "Geocoding.Google",
              "CoveredStatements": 190,
              "TotalStatements": 794,
              "CoveragePercent": 24,
              "Children": [
                {
                  "Kind": "Assembly",
                  "Name": "(4.0.0.0, .NETStandard,Version=v1.3)",
                  "CoveredStatements": 190,
                  "TotalStatements": 397,
                  "CoveragePercent": 48,
                  "Children": [
```


Figure 3. DotCover coverage report snippet for the Geocoding.net project.

As a test coverage tool, we utilized JetBrains dotCover. This Microsoft Visual Studio plugin is a .NET Unit test runner and code coverage tool that can generate a statement coverage report in JSON, XML, etc. (see Figure 3). While it is a static analysis tool, it can also import coverage reports. A challenge we faced is that among the coverage tools we considered, they only report on dependencies between test targets, and do not explicitly indicate or name direct dependencies to the invoking test.

For determining C# code dependencies, Visual Studio 2022 Enterprise Edition (EE) provides a Code Map that is stored as a Directed Graph Markup Language (DGML) file. We then convert its XML-like format to JSON (see Figure 4). The dependency report is then partitioned into a node report and a link report. Only direct dependencies between test and test target are considered, otherwise the dependency structure could readily become very complex with large sets of intermediate nodes and their interdependencies.


```
{
  "DirectedGraph": {
    "-DataVirtualized": "True",
    "-Layout": "Sugiyama",
    "-ZoomLevel": "-1",
    "-xmlns": "http://schemas.microsoft.com/vs/2009/dgml",
    "Nodes": {
      "Node": [
        {
          "-Id": "(@1 @21 @107 Member=get_ShortName)",
          "-Category": "CodeSchema_Method",
          "-Bounds": "-3238.24687463366,-1648.42218743455,118.52,25.96",
          "-CodeSchemaProperty_IsCompilerGenerated": "True",
          "-CodeSchemaProperty_IsPublic": "True",
          "-DelayedCrossGroupLinksState": "Fetched",
          "-Label": "get_ShortName",
          "-self-closing": "true"
        },
        {
          "-Id": "(@1 @21 @109 @1033)",
          "-Category": "CodeSchema_Field",
          "-Bounds": "-2998.79012658679,-1526.46051751268,115.19,25.96",
          "-CodeSchemaProperty_IsPublic": "True",
          "-CodeSchemaProperty_IsStatic": "True",
          "-DelayedCrossGroupLinksState": "Fetched",
          "-Label": "Neighborhood",
          "-self-closing": "true"
        },
        {
          "-Id": "(@1 @21 @109 @1055)",
          "-Category": "CodeSchema_Field",
          "-Bounds": "-2998.79004032377,-1414.62055169237,96.2833333333333,25.96",
          "-CodeSchemaProperty_IsPublic": "True",
          "-CodeSchemaProperty_IsStatic": "True",
          "-DelayedCrossGroupLinksState": "Fetched",
          "-Label": "PostalCode",
          "-self-closing": "true"
        },
```


Figure 4. Code Map snippet (in JSON) for the Geocoding.net project for determining dependencies.

Tests in the test suite (and their containers) are colored based on the test result status: green for successful, red for failed, and yellow for other (such as ignored). Coverage of the test targets is shown as a bar on all four sides and on the elevation, with the blue area visually indicating the percentage of coverage, and black used for the rest. The coverage percentage is also shown numerically.

V. Evaluation

To evaluate our prototype realization of our solution concept, we use a case study based on three scenarios: test coverage, test results, and test dependencies. Geocoding.net was used as an example C# project for demonstration purposes. However, any C# project could be used by the prototype, and currently any coverage tool could be used by mapping and transforming the report format to the DotCover JSON format.

A. Test Coverage Scenario

Testers focused on test coverage are typically concerned about the overall coverage (e.g., to compare its level against some high-level test goal), while also concerned about

assessing details and risks as to which areas were not covered by tests.

Visualization of the System Under Test (SUT) or test target is shown on a plane, with the coverage percentages for a container (folder, directory, package) shown on each side (see Figure 5). A top-level container is used to represent the overall project. The test suite is projected above this onto a separate plane and upside-down.

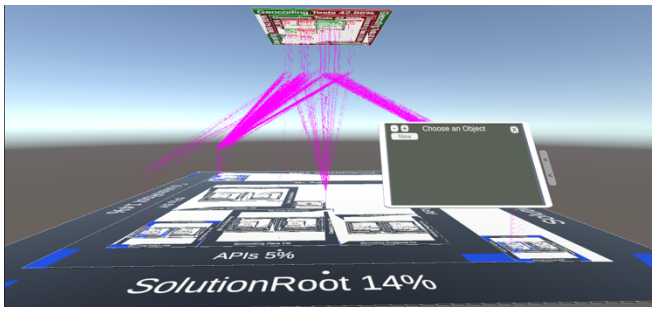

Figure 5. VR-TestCoverage: test target and code coverage on bottom, test suite and test results visible on top, the VR-Tablet on the right, and dependencies drawn as magenta lines.

The test coverage of the test targets is indicated via a bar on all four sides so that from any perspective the coverage is visually indicated (see Figure 6). A bar graph is used on all sides, with blue visually indicating the percentage of coverage and black used for the rest (the exact coverage percentage is also shown numerically). A stepped pyramid paradigm is used to portray the granularity, with the highest cubes having the finest granularity or depth, and the lowest being the least granular. For instance, a user can quickly hone in on overall areas with little to no blue, meaning that coverage there was scarce, and one can quickly find and focus on details (without losing the overview) by focusing on the higher elevations.

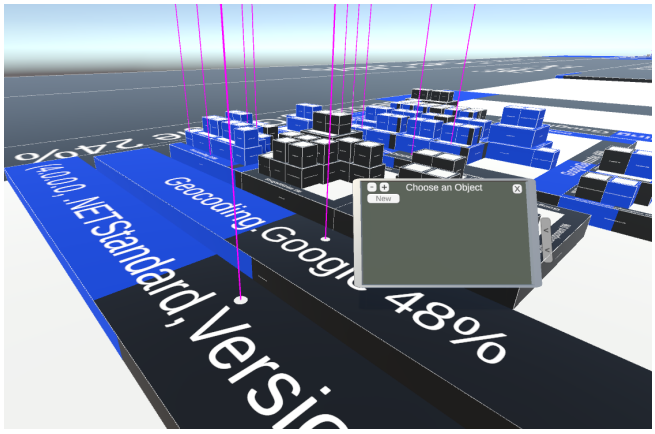

Figure 6. VR-TestCoverage showing stepped pyramid with highest points being finest granularity.

Selecting a test target element causes all other target elements and unassociated dependency links to become transparent, while details from the coverage report can be inspected in the VR-Tablet.

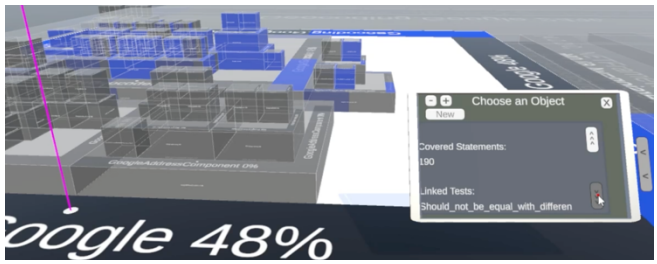

Figure 7. VR-Tablet showing report details for the selected element (non-selected elements become transparent).

B. *Test Results Scenario*

If tests have been run, besides coverage, a tester is also typically interested in the test results and overall pass (or success) rate.

We visualize the test suite as a tree map of all tests using a step pyramid for the third dimension to indicate granularity via depth. Analogously to how coverage was shown as a colored bar on all four sides of a container, on the test suite green is proportionally shown for success rate and red for failure (yellow for other), with its numerical value also given (see Figure 8).

Figure 8. Test suite overview; bar indicates pass percentage for a collection (green for pass, red for failed).

Figure 9 provides a closeup, showing how test case and unit test information is provided, showing the test cases (lowest and closest to the test target), the test unit (showing name and percentage), and a test container (folder or directory). The VR-Tablet permits one to inspect the test results for a selected test.

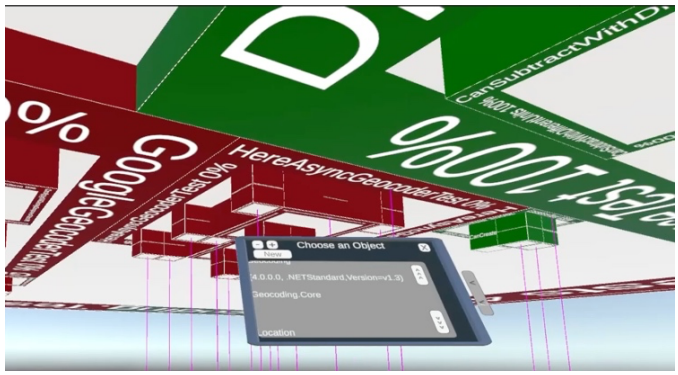

Figure 9. Test suite success shown by test case, test unit, and test container (folder or directory).

C. *Test Dependency Scenario*

One scenario that is often unavailable for testers is a dependency view, with which they can view which tests are directly invoking or reaching which target code. Typically, by convention tests are named in such a way to express the test target, yet the dependencies could nevertheless differ from what one might expect. This is especially true if the test suite consists not only of unit tests but also integration or system tests. By eliminating the guess work, dependencies could be used to determine which tests are primarily reaching a target, and then focus on extending that test in order to increase the target coverage. One challenge is that there is not necessarily a 1-1 match of a test to its test target, thus dependency links provide a way to visualize these hitherto hidden dependencies.

As was shown in Figure 7, VR-TestCoverage depicts the test dependencies of a selected target as a magenta line. When followed, the associated test cases can be seen in the test suite (Figure 10) and can be followed to the most granular level of the test case (Figure 11). Unassociated tests are then not opaque.

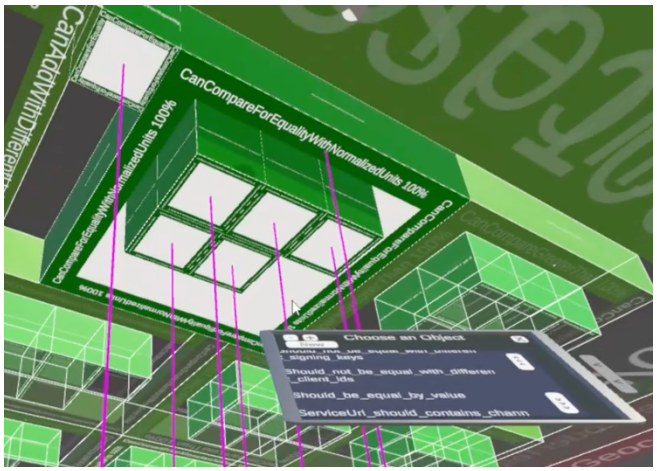

Figure 10. Links followed to dependent test cases.

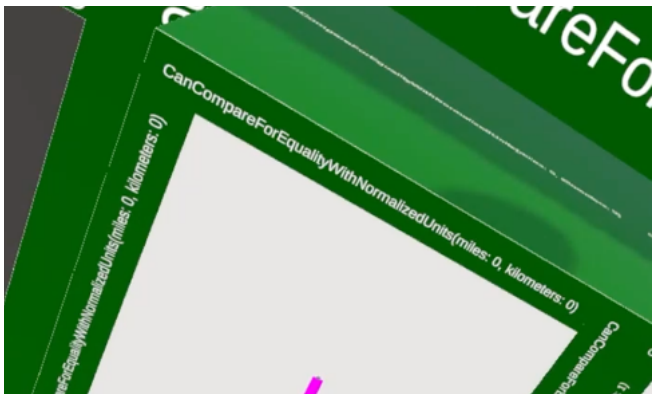

Figure 11. Bottom view showing dependent test cases and pass rate.

As shown in Figure 12, the VR-Tablet can be used to inspect test report details about a selected test object, here showing the test method (CanCompareForEqualityWithNormalizedUnits), test data input values (miles: 1, kilometers: 1.609344), and test status (success).

Figure 12. VR-Tablet showing test case details.

These links can be followed to the test target plane to determine what a selected test is directly reaching (Figure 13).

Figure 13. Dependent target areas remain opaque when a test element is selected.

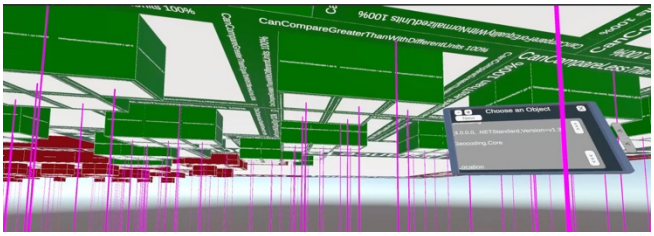

Figure 14. All test dependencies shown (by toggling selection).

By unselecting a test element, all dependencies are shown again with all test elements opaque, as can be seen in Figure 14. Thus, one can follow overall groups or determine that certain tests are perhaps prepared and not (as yet) linked or related to the test target, since no dependencies to the target are shown. This could occur if tests are written before the production code has been implemented, which can be expected, for instance when applying test-driven techniques. Alternatively, this could be an indicator of a test suite and test target mismatch, perhaps if the production code was significantly changed without making associated changes to the test suite.

VI. Conclusion

VR-TestCoverage contributes an immersive test coverage experience for visually depicting and navigating both tests and test targets in VR. Our solution concept visualizes the system under test (production code or test target) on a plane using a tree map with a step pyramid paradigm. The test suite is analogously depicted above it and the dependency links shown. Although the solution concept is generalized, to demonstrate its feasibility, we implemented the VR prototype with .NET technologies. The solution concept was then evaluated using our prototype based on a case study involving three scenarios: test coverage, test results, and test dependencies. The evaluation results showed that all three scenarios are supported by our solution concept and its realization. Immersion provides a different experience for the user in how to experience such coverage metrics and reports, and can enhance and provide a motivational aspect to the testing process in general. For the dilemma of sufficient testing, the insights from VR-TestCoverage can help developers determine areas that have been neglected in testing, or at least to be aware of those areas if they are intentionally out of scope for testing at the timepoint.

Future work includes evaluation with various code projects, supporting various snapshots and coverage difference / variance analysis, supporting a generic coverage report format, support for directly invoking and changing tests within VR, supporting additional programming languages and tool reports, including additional visual constructs, integrating additional metric and tooling capabilities, and conducting a comprehensive empirical study.

Acknowledgment

The authors would like to thank Lukas Tobias Westhäußer for his assistance with the design, implementation, and evaluation.

References

[1] C. Metz, “Google Is 2 Billion Lines of Code—And It’s All in One Place,” 2015. [Online]. Available from: http://www.wired.com/2015/09/ google-2-billion-lines-codeand-one-place/ 2022.07.25

[2] Evans Data Corporation. [Online]. Available from: https://evansdata.com/press/viewRelease.php?pressID=293 2022.07.25

[3] “Software Engineering — Guide to the software engineering body of knowledge (SWEBOK),” ISO/IEC TR 19759:2015, 2015.

[4] "ISO/IEC/IEEE International Standard - Software and systems engineering--Software testing--Part 4: Test techniques," ISO/IEC/IEEE 29119-4:2015, 2015, doi: 10.1109/IEEESTD.2015.7346375

[5] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” Proc. 36th Int’l Conf. on Software Eng. (ICSE 2014), ACM, 2014, pp. 435-445.

[6] M. Ivanković, G. Petrović, R. Just, and G. Fraser, “Code coverage at Google,” Proc. 2019 27th ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Eng., ACM, 2019, pp. 955-963.

[7] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, “How do Developers Test Android Applications?,” 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2017, pp. 613-622, doi: 10.1109/ICSME.2017.47

[8] R. Müller, P. Kovacs, J. Schilbach, and D. Zeckzer, “How to master challenges in experimental evaluation of 2D versus 3D software visualizations,” In: 2014 IEEE VIS International Workshop on 3Dvis (3Dvis), IEEE, 2014, pp. 33-36.

[9] R. Oberhauser, “VR-UML: The unified modeling language in virtual reality – an immersive modeling experience,” International Symposium on Business Modeling and Software Design, Springer, Cham, 2021, pp. 40-58.

[10] R. Oberhauser, “VR-SysML: SysML Model Visualization and Immersion in Virtual Reality,” International Conference of Modern Systems Engineering Solutions (MODERN SYSTEMS 2022), IARIA, 2022, pp. 59-64.

[11] R. Oberhauser, “VR-Git: Git Repository Visualization and Immersion in Virtual Reality,” The Seventeenth International Conference on Software Engineering Advances (ICSEA 2022), IARIA, 2022. To be published.

[12] J. Vincur, P. Navrat, and I. Polasek, “VR City: Software analysis in virtual reality environment,” In 2017 IEEE international conference on software quality, reliability and security companion (QRS-C), IEEE, 2017, pp. 509-516.

[13] K. Dreef, V. K. Palepu, and J. A. Jones, “Global Overviews of Granular Test Coverage with Matrix Visualizations,” 2021 Working Conference on Software Visualization (VISSOFT), 2021, pp. 44-54, doi: 10.1109/VISSOFT52517.2021.00014

[14] A. Rahmani, J.L. Min, and A. Maspupah, “An evaluation of code coverage adequacy in automatic testing using control flow graph visualization,” In 2020 IEEE 10th Symposium on Computer Applications & Industrial Electronics (ISCAIE), IEEE, 2020, pp. 239-244.

[15] F. Pecorelli, G. Di Lillo, F. Palomba, and A. De Lucia, “VITRuM: A plug-in for the visualization of test-related metrics,” Proc. Int’l Conf. on Adv. Visual Interfaces (AVI ’20), ACM, 2020, pp. 1-3, doi: 10.1145/3399715.3399954

[16] K. Alemerien and K. Magel, “Examining the effectiveness of testing coverage tools: An empirical study,” Int’l J of Software Engineering and its Applications, 8(5), 2014, pp.139-162.

[17] K. Sakamoto, K. Shimojo, R. Takasawa, H. Washizaki, and Y. Fukazawa, "OCCF: A Framework for Developing Test Coverage Measurement Tools Supporting Multiple Programming Languages," 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 422-430, doi: 10.1109/ICST.2013.59

[18] R. Oberhauser and C. Pogolski, "VR-EA: Virtual Reality Visualization of Enterprise Architecture Models with ArchiMate and BPMN," In: Shishkov, B. (ed.) BMSD 2019. LNBIP, vol. 356, Springer, Cham, 2019, pp. 170–187.

[19] R. Oberhauser, “VR-ProcessMine: Immersive Process Mining Visualization and Analysis in Virtual Reality,” International Conference on Information, Process, and Knowledge Management (eKNOW 2022), IARIA, 2022, pp. 29-36.

[20] R. Oberhauser, C. Pogolski, and A. Matic, "VR-BPMN: Visualizing BPMN models in Virtual Reality," In: Shishkov, B. (ed.) BMSD 2018. LNBIP, vol. 319, Springer, Cham, 2018, pp. 83–97. https://doi.org/10.1007/978-3-319-94214-8_6

[21] R. Oberhauser, P. Sousa, and F. Michel, "VR-EAT: Visualization of Enterprise Architecture Tool Diagrams in Virtual Reality," In: Shishkov B. (eds) Business Modeling and Software Design. BMSD 2020. LNBIP, vol 391, Springer, Cham, 2020, pp. 221-239. doi: 10.1007/978-3-030-52306-0_14

[22] R. Oberhauser, M. Baehre, and P. Sousa, “VR-EA+TCK: Visualizing Enterprise Architecture, Content, and Knowledge in Virtual Reality,” In: Shishkov, B. (eds) Business Modeling and Software Design. BMSD 2022. LNBIP, vol 453, pp. 122-140. Springer, Cham. doi:10.1007/978-3-031-11510-3_8