VR-DeltaDebugging: Visualization Support for Delta Debugging in Virtual Reality

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

Abstract – **Debugging is a challenging activity involved in software development and maintenance processes. Delta Debugging (DD) is an automatic debugging algorithm and methodology that applies a scientific recurrent hypothesis, trial, and result loop to systematically reduce failure-inducing inputs to a minimal set. Yet, especially for larger (structured) input sets, how DD arrived at its results and its intermediate inputs and test results may not be intuitively evident to practitioners. This paper contributes our solution concept VR-DeltaDebugging for an immersive visualization in Virtual Reality to support comprehension, analysis, and collaboration. A prototype demonstrates its feasibility, and a cased-based evaluation on execution, comprehension and analysis, and scalability provides insights into its capabilities and potential.**

Keywords – delta debugging; visualization; virtual reality; debugging; software engineering.

I. INTRODUCTION

Debugging is a costly and time-consuming activity incurred during software development and maintenance processes. A 2021 study [1] found debugging sessions (even during programming) occurred on average every eight minutes, with sessions lasting from less than a few minutes to over 100 minutes. A 2020 survey [2] of 73 developers reported that roughly a quarter of their time (26%) was spent reproducing and fixing failing tests, averaging 13 hours to fix a single bug. A study on debugging [3] found that almost half of the 303 developers (47%) spend 20-40% of their time debugging, with 26% spending even 40–60%. Over half had no formal debugging knowledge or training, and over 70% were unaware of more advanced debugging tools or approaches, which only very few applied.

Among automated software fault localization techniques and tools, Delta Debugging (DD) [4] is a method and algorithm that *simplifies and isolates failure-inducing input automatically and systematically* by testing subsets and complements of the input. This can reduce debugging effort by narrowing the relevant inputs that cause a test to fail. Debugging and testing are often performed contemporaneously, and one application area that exemplifies DD's applicability and benefit is fuzzing. Fuzzing (or fuzz testing) is an automated dynamic test technique that injects random, invalid, or unexpected inputs and observes a software's behavior (crash, memory leak, vulnerabilities, etc.). Yet fuzzing can result in a large (random) input set for a test failure. DD has been shown to be effective and efficient at isolating some input to the minimum set that still reproduces the failure [5]. DD is also applied in compiler development when dealing with program code as structure text inputs, as exemplified in [6]. As to DD's benefits, the empirical study on DD by Yu et al. [7] found that two thirds of isolated changes in the studied programs were helpful in terms of accuracy and efficiency, providing (in)direct clues in locating regression bugs; yet a third were superfluous changes or incorrect isolations. Thus, DD practitioners should have better analysis and process support tooling for insights into determining the validity of a DD result. This is a problem and underlying motivation for this paper's contribution. We seek a solution that can support DD practitioners in comprehending and analyzing the DD reduction input sets and results, and thus more readily determine valid results (or input or test case issues) and the intermediate steps that led to it. Visualization could support DD and make advanced debugging approaches more accessible to practioners. While 2D debugging tools (textual, visual, or Integrated Development Environment (IDEs)) are prevalent, there has been relatively little investigation into the potential of Virtual Reality (VR) for debugging support, in particular for DD and structured inputs.

In this paper, we propose and investigate applying immersive VR to support the DD method. In prior work, we investigated the application of VR to various other areas. A selection of our prior VR-related contributions in the Software Engineering (SE) space: VR-SDLC [8] models development lifecycles, VR-Git [9] models Git repositories, VR-DevOps [10] models Continuous Development pipelines, VR-SBOM [11] models Software Bill of Materials (SBOM) and software supply chains. HyDE [12] showed a VR-based multi-display IDE that could also be used for debugging support. This paper contributes our VR-DeltaDebugging solution concept towards immersive visualization support for Delta Debugging in VR. A prototype demonstrates its feasibility, while a case-based evaluation provides insights into its capabilities and potential for supporting comprehension, analysis, and collaboration.

This paper is structured as follows: the next section discusses related work. Section 3 describes our solution. In Section 4, our realization is presented, which is followed by our evaluation in Section 5. Finally, a conclusion is provided.

II. RELATED WORK

Regarding DD, the survey by Wong et al. on software fault localization [13] analyzed 587 papers and 68 theses, with the discussion also encompassing DD - yet there is no mention of visualization or VR. Further, all searches found no work directly involving DD visualization. Any work, tools, or

libraries are text-based or involve a Command Line Interface (CLI). As to IDE integration, DDinput [14] was an Eclipse plugin (appears to no longer appears be supported [15]). Work regarding DD tools or libraries includes Picire [16] as described in [17], and that cited in Wong et al. above [13].

As to debugging in general, VR-based work includes Mauer et al. [18] with a VR-based 3D-debugging prototype, demonstrating how VR can be used for programming comprehension and debugging. Our own prior work HyDE [12] demonstrated a VR-based multi-display IDE (Integrated Development Environment), which could also be used for direct programming and debugging support. 3D visualization work includes Code Park [19], which provides a code-centric environment for code comprehension, yet offers no debugging or editing support. Examples of 2D visualization tools supporting fault localization include Tarantula [20] and GZoltar [21], which showed that visualizations can drastically reduce debugging time.

In contrast, VR-DeltaDebugging is a VR solution directly addressing DD visualization support for (un)structured inputs.

III. SOLUTION CONCEPT

Our solution concept is grounded on prior VR research in areas related to modeling, analysis, and collaboration, some of which is highlighted here. Akpan & Shanker's systematic meta-analysis [22] in discrete event modeling found VR/3D to be advantageous for model development, analysis, and Verification and Validation (V&V). 95% of 23 papers concluded 3D was more potent and provided better analysis than 2D (e.g., evaluating model behavior or what-if analysis). Another finding was a consensus that 3D/VR can present results convincingly and understandably for decision-makers. In 74% of 19 papers, model development tasks improved significantly in 3D/VR (team support, precision, clarity).

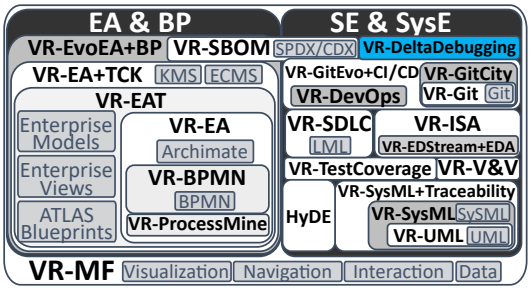

Figure 1. Conceptual map of our published VR solution concepts highlighting their differentiation (VR-DeltaDebugging highlighted in blue).

Our conceptual map of Figure 1 shows our VR-DeltaDebugging solution concept (blue) within the SE and SysE (Systems Engineering) area and in relation to our other prior VR solutions. VR-MF, our generalized VR Modeling Framework (detailed in [23]), provides the basis, providing a domain-independent hypermodeling framework addressing the VR aspects of visualization, navigation, interaction, and data integration. We have published VR-based solutions specific to the Enterprise Architecture (EA) and Business Process (BP) space (EA & BP): VR-EA [23] for mapping EA models to VR, VR-BPMN for BPMN models, VR-EAT for enterprise repository integration, VR-EA+TCK [24] for knowledge and content integration, and VR-EvoEA+BP [25] for EA evolution and business process animation, and VR-SBOM [11]. Solutions in the SE and SysE areas include: VR-Git [9], VR-GitCity, and VR-GitEvo+CI/CD for git-related solutions, VR-DevOps [10], VR-V&V (Verification and Validation), VR-TestCoverage, VR-SDLC [8], VR-ISA for Informed Software Architectures, and VR-UML and VR-SysML for software and systems modeling. HyDE [12] is our VR-based multi-display IDE, and while it can be used for debugging, hitherto none of our work focused directly on supporting debugging in VR.

With regard to structured inputs, Hierarchical Delta Debugging (HDD) [26] has been proposed as a variant to improve DD's effectiveness. However, the study by Yu et al. [27] found that HDD suprisingly did not improve accuracy nor efficiency. Thus, while our solution concept is compatible with HDD, our prototype initially focuses on DD support, incorporating HDD in future work. Since HDD is an AST-oriented reducer, our AST-based nexus can be seen as a precursor to eventual AST-based input support for HDD.

A. Visualization in VR

For text visualization (both input and test code), an interactive scrollable billboard analogy is used for the main screen, similar to terminal screens but enhanced for DD support. It offers a large interaction and viewing space for text-centric analysis. A menu is provided on the side to readily offer interaction without interfering in the analysis. The nexus view is kept synchronized and to the side of the billboard.

For structured DD text inputs, a common alternative graphical visualization form is an Abstract Syntax Tree (AST) (e.g., source code input to debug a compiler/interpreter, or any JSON/XML/HTML/YAML inputs). In VR, we visualize this AST as a *nexus* graph of nodes and edges on the surface of an invisible sphere. 3D nodes depict syntactical elements (classes, functions), while the edges (directed lines) are used to indicate semantic relationships, such as calls or class affiliations. A sphere was chosen to reduce dependency collisions while holding the entire graph spatially compact for immersive flythrough navigation. A Boundary Box (BB) is used to delimit the context of the visual model in case multiple models or model versions are loaded.

B. Navigation in VR

Dual navigation modes are supported in our solution: default gliding controls for fly-through VR, and teleporting to instantly place the camera at a selected position in space. Although teleporting can be potentially disconcerting, it may reduce the likelihood of VR sickness.

C. Interaction in VR

User-element interaction is supported through the VR controllers. A *DD Replay* capability is provided via a slider above the main screen. It is labeled with the total number of DD steps invoked. By adjusting the slider, the DD step and its result are correspondingly displayed on the main screen. Green line numbers indicate the input that passed, and red denotes inputs that failed. Since during main screen

interaction no movement is involved, DD interaction controls are offered either directly on the main screen, or via a side screen with a menu to change the context of the main screen. The VR-Tablet travels with the VR camera to support nexus interaction, in particular AST filtering by node type, and can provide detailed context-specific information for a selected element (e.g., node or relations) from the AST data.

IV. REALIZATION

The logical architecture of our prototype realization is shown in Figure 2. The VR visualization aspects of our prototype (referred to as our frontend) were realized in C# using Unity 2022.3.5f1 (LTS) with the XR Interaction Toolkit 2.3. Our backend consists of our Data Hub that contains a data repository and adapters for invocation and data transformation using Python 3.10. While the Data Hub is conceptually separated via a communication channel, in our prototype this would have created unnecessary overhead. The necessary JSON data could be readily transferred via a socket or Web API. Thus, invocation from C# of the Python adapters utilized subprocesses instead.

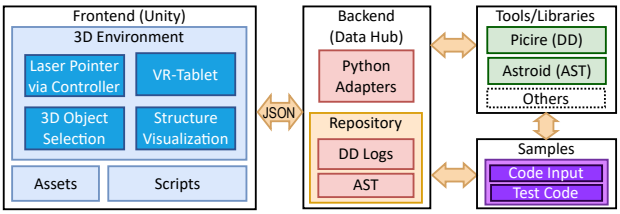

Figure 2. Logical architecture.

```
{
  "steps": [
    {
      "step": 1,
      "result": "FAIL",
      "lines": [
        "'N&+slk%hyp5'\n",
        "\"'@[3(rW*M5W]tMFPU4\\\\P@tz%[X?uo\\\\1?b4T;1bDeYtHx #UJ5\"\n",
        "'w}pMmPodJM,_%%BC~dYN6*g|Y*Ou9I<P94}7,99ivb(9`=%jJ*Y*d~OLXk!;J'\n",
        "'V\"/+!aF-(V4EOz*+s/Q,7)2@0_'\n",
        "\"!iOU8]hqg00?u(c);>:\\\\=V<ZV1=*g#UJA'No5QZ)~--[})Sdv#m*L\"\n",
        "'0iHh[~MzS.U.X}fG7aA:G<bEI\\'Ofn[\",Mx{@jfto}i3D??%V7XdtO6BjYEa#Il)~]'\n",
        "\"E`h7h)ChX0G*m,|sosJ.mu/\\\\c'EpaPi0(n{\"\"
      ],
      "line_numbers": [
        1,
        2,
        3,
        4,
        5,
        6,
        7
      ]
    },
    ...
    {
      "step": 9,
      "result": "FAIL",
      "lines": [
        "'V\"/+!aF-(V4EOz*+s/Q,7)2@0_'\n"
      ],
      "line_numbers": [
        4
      ]
    }
  ],
  "meta": {
    "message": "Reduction complete"
  }
}
```

Figure 3. Extract snippets of DD execution step log output in JSON (intermediate results removed at line 25 for brevity), showing step number, input and corresponding line numbers, and test result for that subset.

A. Backend

We utilized the Picire [16] Python DD implementation. It splits input (by characters or lines) into n chunks (we used n=2), testing these to see if any remain interesting. We created a generic DD logging proxy for testcases, which tracks separate testcase invocation sequences, storing corresponding step, input, line numbers, and result, shown in our JSON-based log output snippet in Figure 3. This retains DD execution state for subsequent playback and analysis.

Visual analysis for structured DD inputs (like source code for compilers/interpreters, JSON, markup) is supported via an AST. We exemplify feasibility by initially supporting Python. The Python Astroid module (extends the Python ast module) provides an enhanced AST with additional semantic information. We then generate a JSON-based AST data model with the following features:

- Nodes for syntactic units: classes, functions, variable assignments, imports.
- Edges between semantically-linked nodes, such as method calls or class affiliations.
- Additional data such as line numbers, code snippet, node type, and parent nodes.

B. Frontend

The nexus assists with structured inputs, exemplified with Python source code. The nexus layout is based on the Fibonacci sphere algorithm for spatial separation together with a force-directed graph algorithm, which adjusts node placement proximity based on relations, the results of which is illustrated in Figure 4. To depict directed relations between elements, rather than adding arrow heads, direction is indicated by coloring from the source (black) to the destination (white) as a gradient, as seen in Figure 5.

To support immersive interaction in the nexus sphere, a VR-Tablet offers a Nexus Stepper check box: when unchecked, the entire AST is depicted; if checked, only the corresponding portion of the input for that step is shown. It also offers a filtering capability (to ghost or make opaque in the nexus) of the visible node types using checkboxes, as shown in Figure 5. To simplify tablet interaction while keeping its size small, pagination was used instead of scrolling. The node type options depend on the loaded AST, and can include, e.g.: Module, Import, classdef, functionsdef, arguments, assign, assignname, assignattr, etc. The BB around the nexus offers a legend of the node type color assignment, and well as metrics such as the total number of nodes visible. To retain and utilize a user's spatial memory, rather than optimize spatial distance, the nexus is not relocated or its layout changed once instantiated, even if steps or filtering cause far fewer nodes to be visible.

Support for selecting a DD Replay step was implemented as a slider on top of the main screen, ranging from initial input on the left to the final result on the right. During Replay interaction in VR, the corresponding input is shown, and the line numbers are colored according to the step and test result (green for pass, red for fail). A menu screen to the right of the main screen provides the ability to load and execute a different DD context.

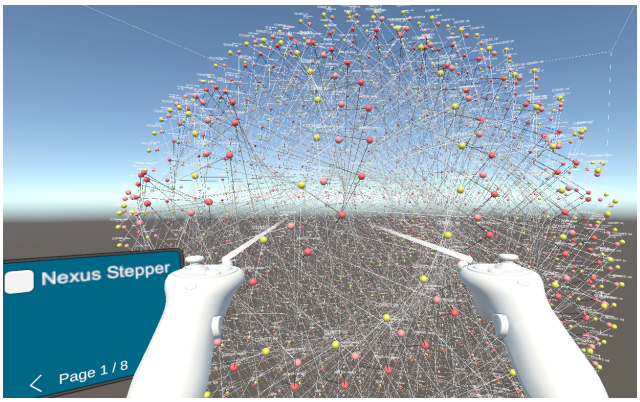

Figure 4. Input nexus of AST code graph.

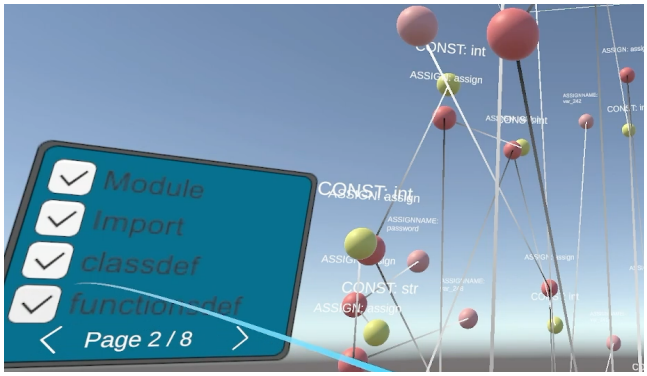

Figure 5. Nexus closeup showing directional dependencies via gradients and the node type filtering ability in the VR-Tablet.

As our evaluation did not necessitate text entry, a virtual keyboard was not included. The implementation could readily be enhanced with a virtual keyboard using laser pointer key selection, as demonstrated in our other VR solutions.

V. EVALUATION

The evaluation of our VR solution concept is based on the design science method and principles [28], in particular a viable artifact, problem relevance, and design evaluation (utility, quality, efficacy). A case study is used based on the following scenarios: DD execution support, DD comprehension and analysis support, and nexus scalability.

A. DD Execution Support in VR

To evaluate DD execution capability in VR, various tests with structured and unstructured inputs were run. The nexus only applies to structured input. To illustrate unstructured input support, input and a Python test from a DD reference site [29] were slightly adapted for our implementation, shown in Figure 6 and Figure 7 respectively.

```
'N&+slk%hyp5'
"'@[3(rW*M5W]tMFPU4\\P@tz%[X?uo\\1?b4T;1bDeYtHx #UJ5"
'w}pMmPodJM,_%%BC~dYN6*g|Y*Ou9I<P94}7,99ivb(9`=%jJj*Y*d~OLXk!;J'
'V"/+!aF-(V4EOz*+s/Q,7)2@0_'
"!iOU8]hqg00?u(c);>:\\=V<ZV1=*g#UJA'No5QZ)~--[})Sdv#m*L"
'0iHh[-MzS.U.X}fG7aA:G<bEI\'Ofn[",Mx{@jfto}i3D?7%V7XdtO6BjYEa#Il)~]'
"E`h7h)ChX0G*m,|sosJ.mu/\\c'EpaPi0(n{"
```

Figure 6. Example unstructured text input. Adapted from [29].

```
from picire.outcome import Outcome


def run_test(code_str):
    try:
        mystery(code_str)
        outcome = Outcome.PASS
    except Exception:
        outcome = Outcome.FAIL
    return outcome


def mystery(inp: str) -> None:
    x = inp.find(chr(0o17 + 0o31))
    y = inp.find(chr(0o27 + 0o22))
    if x >= 0 and y >= 0 and x < y:
        raise ValueError("Invalid input")
```

Figure 7. Example provided Python DD testcase. Adapted from [29].

After execution, the initial result (Step 1/9) is as shown in Figure 8, and moving the stepper to the end (Step 9/9) shows the final result of the line found that causes the test to fail, shown in Figure 9.

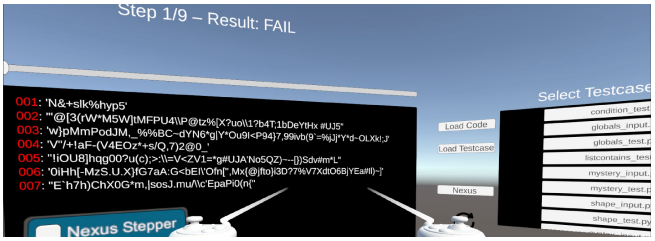

Figure 8. Unstructured input (left) and step and result status (top). Unclear as yet if the input can be further reduced to a single line (or set of characters).

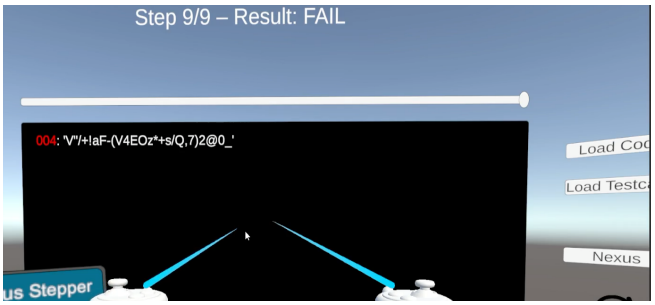

Figure 9. A single input line found to cause the DD test to fail.

B. DD Comprehension and Analysis Support in VR

DD comprehension and analysis are supported in two ways: DD Replay (via the stepper slider) and the graphical DD nexus, which provides a synchronized graphical view for structured DD input, which text-based tools do not offer.

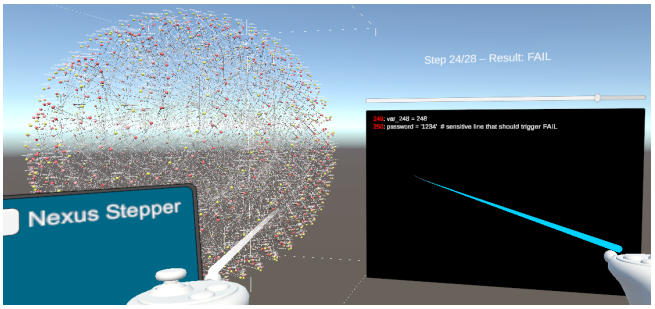

Figure 10. Complete input AST nexus (Tablet Nexus Stepper unchecked).

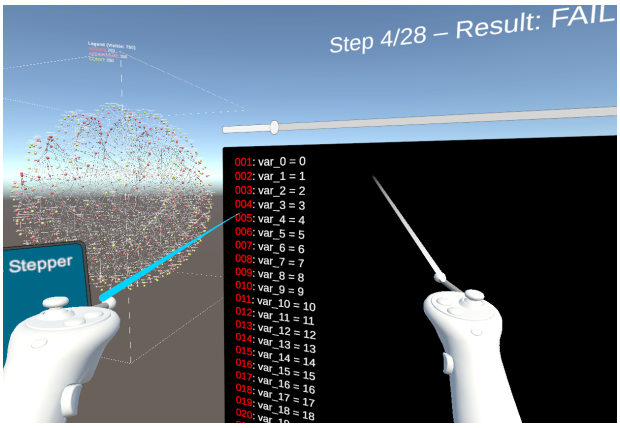

Figure 11. Replay synchronized AST nexus (left) shows reduced input set for Step 4 of 28 on screen (red line numbers indicate set in testcase failure).

To illustrate the comprehension capability, a full AST nexus (since the Nexus Stepper is unchecked on VR-Tablet) of 1500 elements, based on the complete structured text input of 500 lines of assignments in Python code, is shown in Figure 10. A faulty line was intentionally placed on line 250. The nexus view supports DD comprehension by also depicting any known structural relations of the input in a graphical and immersive form, allowing the user to better understand large structured input sets as they may relate to the DD (intermediate) results. A Python AST was used to illustrate this capability, but any structures that can be transformed to a graph-based form could use this capability.

To support analysis of DD results, with the Nexus Stepper checked, moving the slider to Step 4 shows a reduced nexus as well as a reduced textual input set on the main console, as shown in Figure 11. At Step 8, the nexus is further reduced, and the main console shows the passing input (via green line numbers), as seen in Figure 12. The Replay final result shows the failing line found, with a reduced nexus visible on the left that contains only 3 elements, shown in Figure 13.

Immersion in the nexus allows the user to perceive the relations between element types in structured input. The filtering capability by node type is illustrated in Figure 14. Here, the assign nodes were deselected in the VR-Tablet and thus ghosted, enabling the user to focus on (a) specific AST node type(s) of interest.

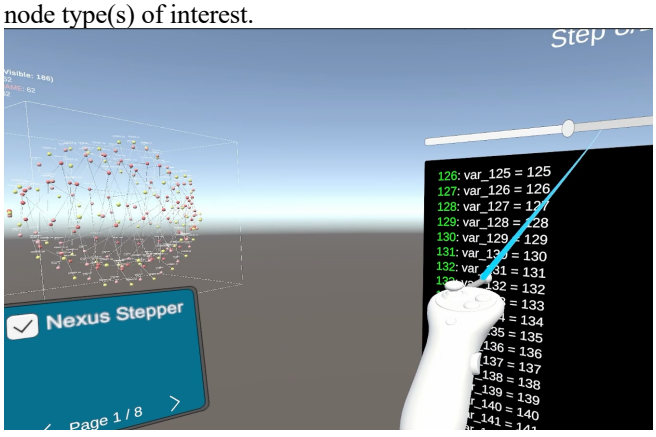

Figure 12. Replay synchronized AST nexus showing reduced input set on Step 8 (main screen green line numbers indicate input passing testcase).

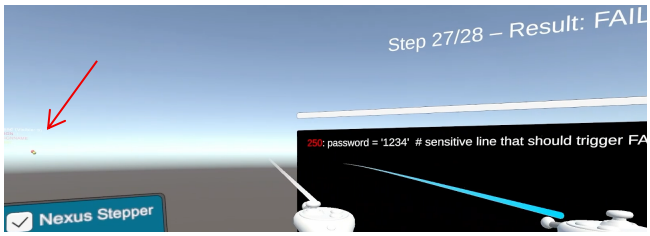

Figure 13. Replay final result showing failing line (reduced nexus on left contains only 3 elements, pointed to by red arrow annotation).

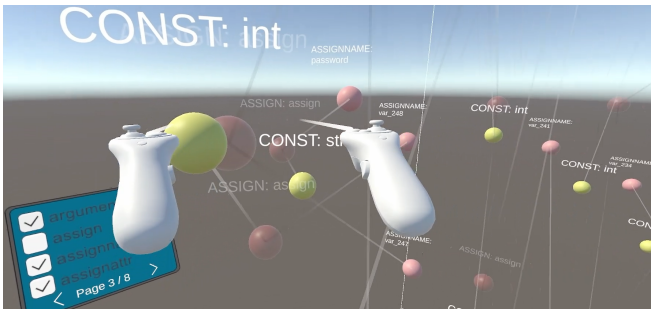

Figure 14. AST nexus filtering (assign nodes are ghosted since deselected).

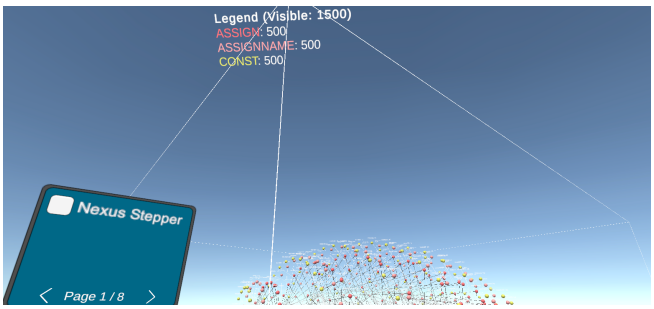

Figure 15. DD test execution input and result status.

C. *Nexus Scalability in VR*

VR has no theoretical spatial limitations. However, the number of visible elements depicted affects the frame rates, which are dependent on software and hardware capabilities. For our scalability scenario, the setup was a desktop Win 11 PC AMD Ryzen 9 7900X with 32GB RAM and NVIDIA RTX 4070 using Unity 2022.3.5f1 (LTS). A large structured input (500 lines of Python code) was depicted as an AST as shown in Figure 4. It consists of 1500 visible elements of three different types (ASSIGN, ASSIGNNAME, CONST) of 500 each and their associated dependencies, as shown on the BB in No negative usability issues were encountered, and it demonstrates the feasibility and scalability of the nexus visualization concept for structured inputs such as ASTs. Future work will evaluate larger code repositories.

VI. Conclusion and Future Work

Debugging has received relatively little visualization support, especially investigating 3D and VR enhancements opportunities and integration with automated debugging tools. This paper described our VR-DeltaDebugging solution concept that offers immersive visualization support for Delta Debugging in VR. Instead of relying on purely text-based DD

invocation, comprehension, and analysis, it offers an interactive cockpit with visual support offering intermediate DD result replay the DD steps that led to the DD result. Structured DD inputs are enhanced with an optional graphical nexus visualization that depicts elements and relations within the input, which might affect the failure, and it's depiction is synchronized with the DD Replay results.

The prototype demonstrates its feasibility. The case-based evaluation provided insights into its capabilities and potential for supporting comprehension, analysis, and scalability. VR could also offer a collaboration space regarding DD issues.

Future work includes support for Hierarchical Delta Debugging (HDD), git-bisect integration, and a comprehensive empirical study that also includes usability.

ACKNOWLEDGMENT

The author would like to thank Umut Dönmez and Jonas Kling for their assistance with the implementation, screenshots, and data preparation.

REFERENCES

[1] A. Alaboudi and T. D. LaToza, "An exploratory study of debugging episodes," arXiv preprint arXiv:2105.02162, 2021.

[2] Cambridge University Judge Business School: The business value of optimizing CI pipelines (2020). [Online]. Available from: https://info.undo.io/ci-research-report 2025.08.19

[3] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," Software Quality Journal, 25(1), 2017, pp.83-110.

[4] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why? " in Proc. Seventh European Software Eng. Conf., Seventh ACM SIGSOFT Symp. Foundations of Software Eng., (ESEC/FSE '99), vol. 1687, 1999, pp. 253–267.

[5] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," in IEEE Transactions on Software Engineering, vol. 28, no. 2, 2002, pp. 183-200.

[6] D. Stepanov, M. Akhin, and M. Belyaev, "ReduKtor: How We Stopped Worrying About Bugs in Kotlin Compiler," In: 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), IEEE, 2019, pp. 317-326.

[7] K. Yu, M. Lin, J. Chen, and X. Zhang, „Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives," Journal of Systems and Software, 85(10), 2012, pp.2305-2317.

[8] R. Oberhauser, "VR-SDLC: A Context-Enhanced Life Cycle Visualization of Software-or-Systems Development in Virtual Reality," In: Business Modeling and Software Design (BMSD 2024), LNBIP, vol 523, Springer, Cham, 2024, pp. 112-129.

[9] R. Oberhauser, "VR-Git: Git Repository Visualization and Immersion in Virtual Reality," 17th Int'l Conf. on Software Engineering Advances (ICSEA 2022), IARIA, 2022, pp. 9-14.

[10] R. Oberhauser, "VR-DevOps: Visualizing and Interacting with DevOps Pipelines in Virtual Reality," Nineteenth International Conference on Software Engineering Advances (ICSEA 2024), IARIA, 2024, pp. 43-48.

[11] R. Oberhauser, "VR-SBOM: Visualization of Software Bill of Materials and Software Supply Chains in Virtual Reality," In: Business Modeling and Software Design (BMSD 2025), LNBIP, vol 559, Springer, Cham, 2025, pp. 52-70.

[12] R. Oberhauser, A. Matic, and C. Pogolski, "HyDE: A Hyper-Display Environment in Mixed and Virtual Reality and its Application in a Software Development Case Study," International Journal on Advances in Software, 11(1 & 2), 2018, pp.195-204.

[13] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Transactions on Software Engineering, 42(8), 2016, pp.707-740.

[14] P. Bouillon, M. Burger, and A. Zeller, "Automated debugging in Eclipse: (at the touch of not even a button)," In: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, 2003, pp. 1-5.

[15] DDinput. [Online]. Available from: https://www.st.cs.uni-saarland.de/eclipse/ 2025.08.19

[16] Picire. [Online]. Available from: https://github.com/renatahodovan/picire/ 2025.08.19

[17] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm," In: International Conference on Software Engineering and Applications, Vol. 2, SciTePress, 2016, pp. 241-248.

[18] S. T. Mauer et al., „A Novel Approach for Software 3D-Debugging in Virtual Reality," In: International Conference on Human-Computer Interaction (HCII 2024), LNCS, vol 14708. Springer, Cham, 2024, pp. 235-251.

[19] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner and J. Laviola, "Code Park: A New 3D Code Visualization Tool," 2017 IEEE Working Conference on Software Visualization (VISSOFT), 2017, pp. 43-53, doi: 10.1109/VISSOFT.2017.10.

[20] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," In: Proceedings of ICSE 2001 Workshop on Software Visualization, 2001, pp. 71-75

[21] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," In: Proceedings of the First IEEE Working Conference on Software Visualization, IEEE, 2013, pp. 1–10.

[22] I. J. Akpan and M. Shanker, "The confirmed realities and myths about the benefits and costs of 3D visualization and virtual reality in discrete event modeling and simulation: A descriptive meta-analysis of evidence from research and practice," Computers & Industrial Engineering, 112, 2017, pp. 197-211

[23] R. Oberhauser and C. Pogolski, "VR-EA: Virtual Reality Visualization of Enterprise Architecture Models with ArchiMate and BPMN," In: Business Modeling and Software Design (BMSD 2019), LNBIP, vol. 356, Springer, Cham, 2019, pp. 170-187.

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

[25] R. Oberhauser, M. Baehre, and P. Sousa, "VR-EvoEA+BP: Using Virtual Reality to Visualize Enterprise Context Dynamics Related to Enterprise Evolution and Business Processes," In: Business Modeling and Software Design (BMSD 2023), LNBIP, vol 483, Springer, 2023, pp. 110-128, https://doi.org/10.1007/978-3-031-36757-1_7.

[26] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," In: International Conference on Software Engineering (ICSE 2006), ACM, 2006, pp. 142–151.

[27] K. Yu, M. Lin, J. Chen, and X. Zhang, „Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives," Journal of Systems and Software, 85(10), 2012, pp. 2305-2317.

[28] A.R. Hevner, S.T. March, J. Park, and S. Ram, "Design science in information systems research," MIS Quarterly, 28(1), 2004, pp. 75-105.

[29] A. Zeller, "Reducing Failure-Inducing Inputs." [Online]. Available from: https://www.debuggingbook.org/html/DeltaDebugger.html 2025.08.19