VRTestSniffer: Test Smell Detector for Virtual Reality (VR) Software Projects

Faraz Gurramkonda*, Avishak Chakroborty*, Bruce Maxim*, Mohamed Wiem Mkaouer†, Foyzul Hassan*
*University of Michigan-Dearborn, Dearborn, MI, USA
†University of Michigan-Flint, Flint, MI, USA
Emails: {gfaraz, avishak, bmaxim, mmkaouer, foyzul}@umich.edu

***Abstract*—Virtual Reality (VR) is an emerging technology increasingly adopted in sectors such as gaming, education, border security, and industrial training. However, testing VR applications presents unique challenges due to factors like active user interaction, hardware dependencies, and immersive environments. Recent studies suggest that developers often write fewer test cases for VR applications, and these limited test cases frequently exhibit test smells. Current research on VR test smell detection can only identify a small subset of test smells and often lacks the necessary context for comprehensive detection. This highlights a critical gap in current testing practices for VR applications and underscores the need for approaches tailored to detecting and addressing quality issues in VR test cases.**

To address this research gap, we developed `VRTestSniffer`, a static analysis-based tool that extends test smell detection capabilities specifically for Unity-based VR applications. `VRTestSniffer` can detect 17 test smell categories, building upon those identified by the state-of-the-art tool `tsDetect`, and achieves an F1-score of 95.61%. It leverages abstract syntax trees (ASTs), control flow graphs (CFGs), and data flow graphs (DFGs) to enhance detection accuracy by capturing both control and data dependencies specific to VR testing patterns. In parallel, we conducted an empirical analysis of real-world VR projects to examine the prevalence and characteristics of these test smells. Our findings reveal that a few smelly test categories are associated with design issues such as Blob and Complex Class in functional code. We believe that `VRTestSniffer`, along with the empirical insights derived from this study, can help VR developers write more effective, reliable, and maintainable test cases. To support further research and replication, our tool, dataset, and analysis results are publicly available at [1].

***Index Terms*—Virtual Reality, Program Analysis, Test Smell**

I. Introduction

Virtual Reality (VR) has emerged as a transformative technology, revolutionizing domains such as gaming [2], education [3], [4], border security [5], and industrial training [6] by enabling immersive and interactive experiences. As VR continues to gain popularity and is adopted for increasingly complex use cases, VR applications are becoming more sophisticated and widely deployed. Ensuring their reliability and correctness is therefore critical. Software testing plays a central role in ensuring software quality and is considered one of the most essential activities in the development process [7], [8], [9]. Like traditional software systems such as web applications, VR applications require rigorous testing [10], [11], [12]. However, writing test cases is often challenging and requires the same level of care and design effort as production code [13]. Prior research [9], [14], [15] has found that developers often prioritize writing production code over creating test cases, which can result in test quality issues. In the context of VR, designing and maintaining test cases is even more complex due to the immersive, interactive, and hardware-dependent nature of the applications [16]. A recent study by Rzig et al. [11] found that 79% of the VR projects examined had no automated tests. Furthermore, in projects that did contain tests, an average of 38.43% of the test cases exhibited at least one type of test smell, with some projects containing up to 92% smelly tests.

In the context of software engineering, a test smell refers to a suboptimal test design that suffers from issues such as poor maintainability, low readability, and reduced effectiveness in detecting faults. Van Deursen et al. [17] originally introduced the concept of test smells as indicators of problems in test code that may require refactoring to improve test quality. Subsequent work by Tufano et al. [18] showed that test smells tend to have high survivability over time and that a correlation exists between smells in test code and those present in the associated production code. Several tools have been developed to detect test smells in various programming languages. For example, Peruma et al. [19] introduced `tsDetect`, a static analysis tool capable of identifying 19 types of test smells in Java projects. Similarly, Wang et al. [20] proposed `PyNose`, which detects 17 common test smells as well as one Python-specific smell in Python-based test suites. In the domain of virtual reality (VR), recent work by Rzig et al. [11] proposed a test smell detection tool capable of identifying only six test smell categories: *Assertion Roulette*, *Eager Test*, *General Fixture*, *Mystery Guest*, and *Lazy Test* in VR projects. However, their analysis was limited in scope, as it did not explore additional traditional test smell types and did not consider code context and test input while detecting test smells. Also, existing state-of-the-art tools, such as `PyNose`, and `tsDetect`, do not support VR-Frameworks such as Unity, despite being the most popular for game development [21].

In this research, we aim to address existing gaps by extending and adapting test smell detection techniques for Unity-based VR applications. We began by analyzing the test smell categories supported by the state-of-the-art tool `tsDetect`, which identifies 19 test smell types in traditional Java projects. Through our mapping analysis, we found that 17 of these smells are applicable to VR applications developed using the Unity framework and the C# programming language. While the 17 test smells are derived from existing

literature, most of them manifest differently in Unity-based VR projects; therefore, current literature tools cannot detect them. To support accurate detection of these state-of-the-art test smells, it is necessary to refine the detection rules and analysis process. Building on this observation, we developed a tool named `VRTestSniffer` to detect test smells in Unity-based VR projects. To avoid the high false positive rate of solely relying on abstract syntax trees(ASTs) [22], [23] for program analysis, `VRTestSniffer` augments ASTs with control flow graphs (CFGs) [24], and data flow graphs (DFGs) [25] to analyze VR test code. CFGs are essential for identifying control sequences which are necessary for detecting several test smell categories, while DFGs help track test inputs shared across test cases. By incorporating more contextual code information, `VRTestSniffer` achieved a 27.28% improvement in F1-score compared to the existing VR test smell detection tool developed by Rzig et al. Finally, we used `VRTestSniffer` to extract test smells from real-world VR applications and conducted an empirical analysis to examine the associations between these test smells and corresponding design issues (i.e., code smells) in production code.

We conducted our analysis of VR test smells using a dataset of 314 VR applications developed by Rzig et al. [11]. This dataset includes projects developed by independent contributors (164), organizations (67), and academic institutions (83). Through this analysis, we aim to answer the following three research questions:

- **RQ1:** What is the prevalence of test smells in VR projects?
- **RQ2:** What is the effectiveness of `VRTestSniffer` in identifying VR test smells?
- **RQ3:** Is there an association between identified test smells and code smells in VR projects?

Overall, our contributions are as follows:

- We developed `VRTestSniffer`, a test smell detection tool capable of identifying 17 test smell categories in Unity-based VR software projects. To the best of our knowledge, this is the most comprehensive test smell detection tool tailored for VR applications.
- We created a benchmark dataset for VR test smell detection, covering all 17 applicable test smell categories. This dataset provides a valuable foundation for future research and tool evaluation.
- We conducted an empirical study revealing a weak association between test smells and design issues (i.e., code smells) in the corresponding production code of VR projects. These findings can inform developers and tool builders in more effectively addressing test quality and structural design problems.
- We have made `VRTestSniffer`, the benchmark dataset, and our empirical analysis results publicly available at [1]. These resources aim to support the broader community in identifying and mitigating test smells in VR software development.

II. Background

Virtual reality (VR) has become a groundbreaking technology that offers immersive and interactive experiences in areas such as gaming, education, and industrial training. Unlike traditional desktop or mobile applications, VR applications simulate three-dimensional environments that allow users to interact in real time through gestures, motion, and gaze [26], [27]. This immersive nature necessitates not only rich visual and auditory feedback but also seamless integration with hardware devices such as head-mounted displays (HMDs), hand controllers, and motion-tracking systems.

The distinct characteristics of VR introduce several unique development challenges. First, VR environments must maintain high frame rates and low latency to prevent motion sickness and provide smooth user experiences. Second, user interaction in VR is dynamic and multidimensional, requiring spatial awareness, gesture recognition, and synchronization across multiple sensory inputs. Third, developers must address the complexity of device compatibility, real-time rendering, and interaction management in 3D space. To manage these technical challenges, developers often rely on game engines such as Unity or Unreal Engine. Unity, in particular, is one of the most widely adopted platforms for VR development [21], thanks to its extensive support for VR platforms, asset libraries, and its C# programming environment. Unity offers built-in support for key VR components, including physics, graphics, and animation systems. However, even with these capabilities, VR applications demand rigorous testing to ensure functional correctness, performance, and user safety. To evaluate and improve test quality, researchers have introduced the concept of test smells, symptoms of poor test design that can compromise test maintainability, reliability, and effectiveness. For instance, Assertion Roulette is a test smell that arises when a test method includes multiple assertions without descriptive messages or context, making it difficult to determine which assertion fails during execution. Other common test smells include Eager Test, Mystery Guest, and General Fixture, all of which affect the clarity and effectiveness of test code. While there has been significant progress in detecting and addressing test smells in traditional software domains, research on test smells in VR applications is still in its early stages.

III. Research Approach

The overview of our research approach is presented in Figure 1. Our approach consists of two main components: **(i) Development of `VRTestSniffer`, a static analysis-based tool for detecting test smells in Unity-based VR applications**; and **(ii) An empirical analysis to examine the association between test smells and code smells**. In the following subsections, we provide a detailed explanation of each component of our research approach.

A. Developing VRTestSniffer

1) Identifying VR Applicable Test Smell Categories

As discussed in Section I, the existing VR test smell detection tool proposed by Rzig et al. [11] supports only six

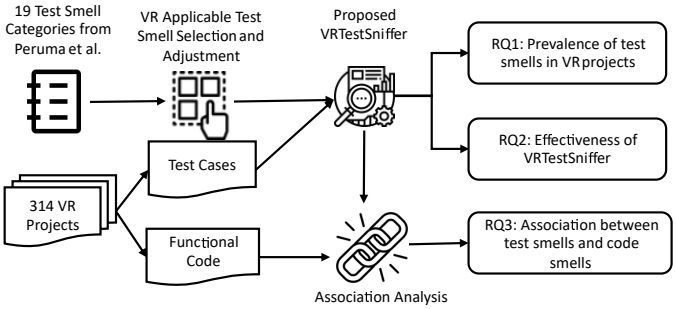

Fig. 1: Overview of the Research Approach

test smell categories. In this research, we aim to significantly extend the detection capabilities for VR applications. To this end, we conducted an applicability analysis of the 19 test smell types supported by the state-of-the-art tool `tsDetect` [28], evaluating their relevance in Unity-based VR testing scenarios. Two authors independently performed this analysis and determined that 17 of the 19 test smells are applicable to Unity-based VR projects. The remaining two—**Conditional Test Logic** and **Resource Optimism**—were excluded due to characteristics unique to VR development. **Conditional Test Logic** is typically flagged when test methods include control structures such as `if`, `for`, or `switch`, which alter test behavior. In VR contexts, however, these constructs are often necessary for simulating user interactions or environmental dynamics (e.g., physics or mechanical behaviors), and thus are not considered smelly. **Resource Optimism** is identified when tests assume the presence of external resources (e.g., files) without checking their existence and loading states in the external resources. In Unity-based VR applications, external resources are typically accessed through Unity's managed systems (e.g., `Resources.Load()`) and store animation states, making this smell largely irrelevant. Although Mystery Guest and Resource Optimism involve external resources, we retained Mystery Guest for its broader implications on test modularity and hidden dependencies, while Resource Optimism was discarded due to its narrow scope and infrequent relevance in Unity's managed resource environment.

2) Identifying VR Applicable Test Smell Rules

Although we adopted 17 test smell categories from prior literature (e.g., tsDetect), their manifestations in Unity-based VR projects differ substantially, limiting the applicability of existing detection rules. For instance, while *Sleepy Test* in conventional systems is typically identified through `Thread.sleep(x)`, in Unity-based VR projects it often arises from coroutine-based delays (e.g., `yield return new WaitForSeconds(x)` or `After(x).Seconds`). Likewise, *General Fixture* in traditional systems refers to overly generic setups that instantiate unnecessary objects or state. In Unity-based VR projects, setup is frequently performed in `Start()` or `Awake()` methods, or via Unity Test Framework's `[SetUp]`, where test classes commonly instantiate complete GameObjects, Scenes, Cameras, or VR Rigs, many of which remain unused. Listing 1 demonstrates this issue, where both Player and Enemy objects are instantiated through the Start() method, yet only the Player is exercised. Such General Fixture smells in VR projects not only hinder maintainability but also incur performance costs (e.g., slow tests, memory leaks, and frame delays). To address such VR test smell detection rules, we performed an adjustment analysis for the selected 17 smell categories and refined each detection rule to align with VR-specific testing patterns. Table I summarizes the final detection rules implemented in `VRTestSniffer`, for identifying the 17 test smell categories relevant to Unity-based VR projects.

```
1  public class BattleTests : MonoBehaviour
2  {
3      void Start()
4      {
5        var playerObj = new GameObject("Player");
6        player = playerObj.AddComponent<Player>();
7        var enemyObj = new GameObject("Enemy");
8        enemy = enemyObj.AddComponent<Enemy>();
9      }
10     [UnityTest]
11     public System.Collections.IEnumerator
       ↪ PlayerStartWithCorrectHealth()
12     {
13       yield return null; //wait for Start()
14       Assert.AreEqual(100, player.Health);
15     }
16 }
```

Listing 1: General Fixture Example

3) Developing Static Analysis-Based `VRTestSniffer`

To perform automated detection of test smells in Unity-based VR applications, we developed `VRTestSniffer`, a static analysis tool designed to identify and analyze test cases for potential test smells. The tool operates by traversing the directory structure of each VR project and examining file names to identify test files. In Unity-based VR projects, test files commonly follow naming conventions such as *Test.cs or Test*.cs. Once test files are identified, our approach parses the C# test code and generates an abstract syntax tree (AST) using srcML [29]. An Abstract Syntax Tree (AST) represents the hierarchical structure of source code by capturing its abstract syntactic elements. In our approach, we utilize `srcML` to generate the AST in XML format. We then parse this XML representation and model the AST as a graph $G_{\text{ast}} = (V_{\text{ast}}, E_{\text{ast}})$, where V_{ast} is the set of vertices, with each vertex representing a syntactic construct in the program and E_{ast} is the set of directed edges, representing the parent-child relationships between these constructs, corresponding to the structural hierarchy of the source code. The Abstract Syntax Tree (AST) is one of the most fundamental and intermediate representations of source code. It serves as the foundation for generating other representation formats, such as the Control Flow Graph (CFG). ASTs are particularly effective for identifying structural patterns and hierarchical relationships in code. However, while ASTs allow for the detection of simple syntactic constructs, they are limited in capturing deeper semantic information—such as program dependencies and data flows—which are crucial for detecting certain categories of test smells. As shown in Listing 2, a

TABLE I: Detection Rules for Unity-Based VR Test Smells

Test Smell	Standard Definition	Unity-Based VR Adaptation
Assertion Roulette	Multiple assertions without messages.	Detects assertions hidden inside Unity helper methods via interprocedural analysis.
Sensitive Equality	Equality checks on entire objects or fragile composites.	Flags equality on Unity objects (e.g., `GameObject`, `Transform`) or exact string equality of composite values.
Unknown Test	No assertions detected.	Avoids false positives when assertions occur in nested helpers or coroutines.
Redundant Print	Debug/print statements instead of validation.	Flags `Debug.Log` or `Console.WriteLine` statements without assertions, common in frame-driven tests.
Redundant Assertion	Always-true assertions.	Detects placeholders (e.g.,`Assert.IsTrue(true)`).
Sleepy Test	Explicit sleep/delay calls.	Captures Unity delay constructs (`yield return new WaitForSeconds(x)`, `After(x).Seconds`) beyond `Thread.Sleep`.
Lazy Test	Tests reuse the same logic/inputs.	Tracks parameter equivalence in coroutine tests and helpers using data-flow analysis.
General Fixture	Setup creates unused variables/objects.	Detects `[SetUp]` or `Start()` initializations—including both regular and VR objects—where created objects remain unused in the test.
Ignored Test	Tests explicitly marked ignored.	Detects Unity attributes such as `[Ignore]` and `[UnityTest, Ignore("...")]`.
Exception Test	Tests focused on exceptions.	Excludes legitimate exception-checks (`Assert.Throws`, `[ExpectedException]`).
Duplicate Assert	Identical repeated assertions.	Flags duplicates only if no state change occurs across frames/yields.
Default Test	Placeholder/template tests.	Detects Unity defaults (e.g., `Test1`) and "do-nothing" enumerators.
Mystery Guest	Hidden external dependencies.	Ignores Unity engine calls (e.g., `LoadScene`, `GetComponent`); flags unmocked I/O, DB, or network.
Eager Test	Test validates too many behaviors.	Flags coroutine tests verifying >3 subsystems (UI, scene, assets, etc.) in one method.
Empty Test	No assertions or logic.	Detects enumerators without meaningful actions or assertions.
Magic Number	Hardcoded literals.	Retained without changes.
Constructor Init.	Shared state initialized in constructors.	Flags constructors creating Unity objects/state instead of `[SetUp]`; hides fixture prep.

potential lazy test is illustrated by both test cases calling the method under test, `Operations.Max`, with seemingly similar parameters. Using only an AST to search for instances where `Operations.Max` is tested with similar inputs could incorrectly identify this as a lazy test, as the AST alone cannot differentiate between actual input values. By performing data flow analysis to determine input values, we confirm that this is not a lazy test. Similarly, VR developers often write test cases and assertions using helper methods. At the same time, Unity's component-based architecture introduces dynamic data dependencies through constructs like `GetComponent<T>()`, `SendMessage(...)`, and serialized fields. In our data flow analysis, we model `GetComponent<T>()` using symbolic aliasing by creating data dependency edges from the source GameObject to the assigned variable, allowing us to approximate inter-component value propagation. For `SendMessage(...)`, which involves dynamic dispatch via string identifiers, we conservatively add side-effect edges to all methods with matching names and compatible signatures across reachable `MonoBehaviours`. While concrete value resolution is limited by the static nature of our analysis, we annotate unresolved values with wildcards and account for common Unity-specific patterns to improve detection accuracy. Without considering such data flow analysis of test cases, our analysis could suffer from inaccuracy. Therefore, additional representations, such as control flow graphs (CFGs) and data flow graphs (DFGs), are necessary for more comprehensive analyses.

```
[Test]
public void MaxValue_With_SomeNegatives(){
 List<int> input =new List<int> {0,-8,5,17};
 int output = Operations.Max(input);
 AssertMatches(17,output)
}
[Test]
public void MaxValue_of_ListNegatives(){
 List<int> input=new List<int> {-22,-8,-5,-17};
  int output = Operations.Max(input);
  AssertMatches(-5, output);
}
private AssertMatches(int actual, int expected){
   Assert.AreEqual(actual,expected);
}
```

Listing 2: Example of potential Lazy Test

During the construction of the control flow graph (CFG), we employed a static analysis-based approach to enumerate all possible statements that can be executed. To generate test case–specific control flows, we considered both intra-procedural and inter-procedural dependencies. Specifically, IEnumerator-based coroutine methods are modeled as distinct nodes in the CFG and are linked back to their caller functions. Yield-based delays (e.g., yield return WaitForSeconds(x) or yield return null) are treated as temporal control-transfer points. For Unity's event-driven system, including lifecycle callbacks such as OnTriggerEnter, we model these explicitly as control roots and capture subsequent method invocations through static call resolution. This allowed us to capture call sequences within individual test methods as well as across methods invoked by those tests. Based on the order of statements and the associated program dependencies, we represented the CFG as a graph: $G_{\text{cfg}} = (V_{\text{cfg}}, E_{\text{cfg}})$, where V_{cfg} denotes the set of AST-derived code statements, and E_{cfg} represents the procedural dependencies (i.e., control flow edges) between these statements. Similar to the CFG, we also generated a data flow graph (DFG) on top of the AST to

capture potential changes or uses of values by statement nodes. Since DFGs are centered around variables and data, constructing them requires careful tracking of variable scopes. During DFG construction, we analyzed variable definitions and usages within each statement. We computed reaching definitions to identify where each variable is defined and subsequently used. Based on these definition-use pairs, we constructed the DFG as a graph: $G_{\text{dfg}} = (V_{\text{dfg}}, E_{\text{dfg}})$, where V_{dfg} denotes the set of AST nodes, and E_{cfg} represents data dependencies (i.e., edges between variable definitions and their corresponding uses). However, due to the static nature of our analysis, the DFG is unable to capture runtime value assignments or dynamically computed values. In such cases, we annotate the corresponding values with a wildcard symbol (*) to indicate an unknown or arbitrary value. With the AST, CFG, and DFG representations generated, we performed a unification process to construct a unified graph. During unification, for each AST node representing a statement that has associated control flow dependencies in the CFG, we incorporate these dependencies into the corresponding AST node. As illustrated in Figure 2, the AST of the test case includes a method call expression, `AssertMatches`, which has associated program dependencies in the CFG. In such cases, we link the related dependency nodes from the CFG to the method call node in the AST. Similarly, for variable assignments and usages within the AST, we consult the DFG to retrieve potential data values and attach the relevant data nodes. Finally, we overlay these three graphs into a unified intermediate representation, where each node retains its syntactic origin (AST) and is annotated with both control and data dependencies as separate edge types. This graph is traversed by our detection engine using smell-specific queries (e.g., detecting deferred assertions by following both control paths and data aliases). This augmentation of the AST tree enhances its semantic richness, enabling more accurate detection of test smells. With the unified AST tree generated by integrating the representations of AST, CFG, and DFG, we traverse the tree using a breadth-first search (BFS) algorithm to detect the test smell patterns, as defined in Table I. During the smell detection process, certain smells—such as *Empty Test*—can be identified using only AST node patterns. However, for other smell categories, such as *Lazy Test*, both AST structures and variable value information from the DFG are required to accurately detect the smells.

4) Preparing `VRTestSniffer` Evaluation Data

Our evaluation uses a curated dataset of 314 manually verified Unity VR projects, originally constructed by prior work (Rzig et al. [11]). These projects span a diverse mix of 164 independent, 67 organizational, and 83 academic VR applications, making the dataset highly representative of the broader Unity VR testing landscape. To determine the validity of our smell detector test tool in Unity-based applications, we conducted a manual analysis to establish a high-confidence benchmark dataset. This manual process was essential for assessing the accuracy of the tool and gaining deeper insight into the manifestation of the test smell. Our initial automated analysis without optimization with CFD and DFD revealed 24,021 positive and 288 negative (non-smelly) cases in 17 smell categories. To ensure our sampling is statistically sound, we incorporated a stratified random sampling with a 95% confidence level and a 5% margin of error; we selected a sample of 553 test cases for manual review, consisting of 387 smelly test cases and 166 non-smelly test cases. Two of the authors, each with experience in Unity-based test case development and software quality assurance, independently analyzed the random sample data set. Each of the test cases was evaluated on the basis of the corresponding test smell category to which it was assigned. To mitigate bias, we conducted a conflict resolution session to resolve any discrepancies and achieve concise results. The Cohen's kappa score before the conflict resolution was 0.80. This consensus-driven process resulted in a unified dataset that served as our ground truth for evaluation. The dataset was used for the evaluation of `VRTestSniffer`.

B. Manual Analysis of Test–Code Smell Relationships.

While `VRTestSniffer` can detect test smells in VR projects, it does not provide insight into the underlying causes that lead developers to write such smelly test cases. Prior work by Tufano et al. [18] suggests that test smells are often strongly associated with code smells in production code. To investigate this relationship in the context of VR applications, we aim to analyze whether test smells are more likely a consequence of code smells or broader design flaws within the associated production code. To achieve this, we mapped each of the 402 test smells (identified by `VRTestSniffer`) to their corresponding methods in the production code across a dataset of 314 Unity-based VR projects. We applied stratified random sampling with 95% confidence and a 5% margin of error to select a representative set of test cases across all smell categories. For each sampled test case, we identified the corresponding production method(s) under test by analyzing assertion targets, test inputs, and method invocation patterns. The two authors then independently assessed the associated production methods or classes for five well-established code smell types—Blob Class, Complex Class, Class Data Should Be Private, Spaghetti Code, and Functional Decomposition—using definitions and guidelines from Tufano et al. [18]. Inter-rater agreement was measured using Cohen's Kappa (K = 0.96), indicating a high level of consistency. Any disagreements were resolved through discussion and consensus during a reconciliation meeting. Finally, we applied the Odds Ratio (OR) [30] to measure the degree of association between test smells and code smells.

IV. Results

A. RQ1: Prevalence of Test Smells in VR Projects.

To investigate the prevalence of test smells, we applied `VRTestSniffer` to a dataset of 314 VR projects discussed in Section III-A4. A summary of the detected test smells across these VR projects is presented in Table II.

We analyzed the distribution of test smells across 314 VR projects. A total of 16,490 test smells were identified. After removing duplicate test cases resulting from the use

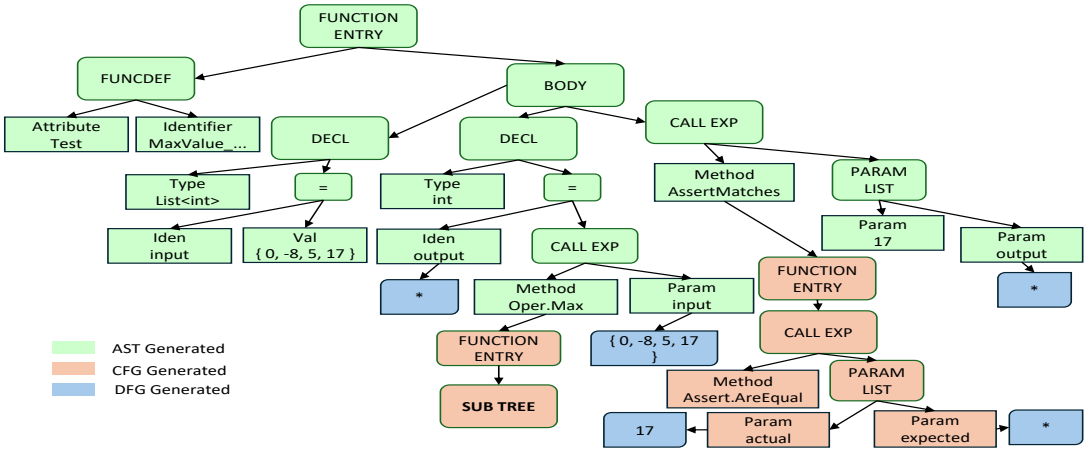

Fig. 2: Code Representation Built from AST, CFG and DFG

TABLE II: Distribution of Test Smells Across Project Classifications

Smell Type	Academic	Independent	Organization
Assertion Roulette	393	1463	3469
Sensitive Equality	31	39	5
Unknown Test	115	907	396
Redundant Print	3	5	2
Redundant Assertion	3	3	4
Sleepy Test	1	109	34
Lazy Test	162	316	380
General Fixture	104	312	595
Magic Number	172	1171	606
Ignored Test	3	9	83
Exception Throwing Test	62	45	151
Empty Test	6	16	6
Eager Test	128	587	1061
Duplicate Assert	0	0	261
Constructor Initialization	10	13	18
Default Test	2	4	2
Mystery Guest	55	270	23
Total	1250	5269	7096

of the same asset libraries, we identified 13,615 unique test case smells. Among these 13,615 test smells, organization-based projects accounted for the majority (7,096), followed by independent projects (5,269) and academic projects (1,250). The most prevalent test smell was AssertionRoulette, with 5,325 occurrences—most prominently in organization-based projects (3,469). Other commonly observed smells included Eager Test (1,776), Magic Number (1,949), Unknown Test (1,418), and General Fixture (1,011). In contrast, smells such as Redundant Print, Redundant Assertion, Default Test, and Empty Test appeared infrequently across all categories. Notably, DuplicateAssert was exclusive to organization-based projects (261 instances), potentially reflecting challenges in test modularity or code review processes in larger teams. Additionally, smells like IgnoredTest and SleepyTest were more prevalent in independent and organizational projects, pointing to gaps in test discipline or resource constraints. These results reveal the widespread and varied nature of test smells in VR software development and reinforce the value of automated detection tools like `VRTestSniffer` in maintaining high-quality test suites. On an evaluation machine (Intel Xeon W-2235 @3.80GHz, 32 GB RAM, OpenJDK 17, Ubuntu Linux), we ran `VRTestSniffer` across 314 Unity VR projects, completing the full analysis in 14 hours and 37 minutes, with an average runtime of 2 minutes and 48 seconds per project. The detailed analysis of each test smell category is presented in the following subsections.

1) Assertion Roulette (AR)

Assertion Roulette was the most frequently detected test smell in our dataset, particularly prevalent in organizational projects, which accounted for 3,469 instances. Independent projects followed with 1,463 occurrences, while academic projects exhibited 393 cases. The high incidence in organizational codebases may stem from the complexity and density of assertions within enterprise-scale test suites.

```
1 [TestMethod]
2 public void Responder_SendImage_Fills_Response_
      ↪ Correctly_For_Bmp_Images(){
3 _Responder.SendImage(_Request.Object, _Response.
      ↪ Object, _Image.Object, ImageFormat.Bmp);
4 AssertImageMatches(ImageFormat.Bmp, MimeType.
      ↪ BitmapImage);
5 }
6 private void AssertImageMatches(ImageFormat
      ↪ imageFormat, string mimeType){
7 Assert.AreEqual(HttpStatusCode.OK, _Response.
      ↪ Object.StatusCode);
8 ...
9 }
```

Listing 3: Assertion logic delegated to helper from vradarserver

Our analysis also revealed a structural pattern specific to Unity-based VR projects. In several cases, test methods did not contain any assertions directly; instead, they delegated assertion logic to helper or sub-functions. As illustrated in Listing 3, the test method `Responder_SendImage_Fills_Response_Correctly_For_Bmp_Images()` might be misclassified as a clean or unknown test if analyzed superficially, since the actual assertions reside in the `AssertImageMatches(...)` helper method. To correctly identify such instances, `VRTestSniffer` CFG to identify inter-procedure calls. This enhancement significantly improves detection accuracy while reducing false positives in both Assertion Roulette and Unknown Test categories.

2) Sensitive Equality (SE)

Sensitive equality was a rare occurrence across all categories of projects, with only a handful of cases observed. This indicates a low frequency, totaling only 39 instances, with the highest concentration found in independent projects. This scarcity may also suggest a limited reliance on exact object equality checks in Unity-based VR tests.

3) Unknown Test (UT)

Due to incomplete test documentation, an unknown test case predominantly appeared in independent projects, totaling 907 occurrences. Organizational projects have a comparatively moderate occurrence level with 396 cases, while academic projects exhibit very few cases, with only 115, indicating a better structural assignment in these settings.

4) Redundant Print (RP)

This smell was almost negligible across different classifications, with the highest occurrence being just 5 in independent projects. Its low presence suggests that test developers typically avoid writing excessive print statements, likely due to the robust debugging support provided by modern VR IDEs.

5) Redundant Assertion (RA)

Very few redundant assertions were detected, with only 4 instances found in organizational projects. This suggests that VR developers rarely include RA and often use the same assertion statement to test different functionalities.

6) Sleepy Test (ST)

Sleepy tests are relatively rare, predominantly found in independent projects (109 cases), followed by organizational projects (34 cases), and barely present in academic projects, with only a single occurrence (1 case). These involve constructs such as After.Seconds(X), and WaitForSeconds(x), as illustrated in the code snippet in Listing 4, where `X` denotes seconds. This points to potential timing issues in VR environments when testing asynchronous scenes or animations. While `Thread.sleep()` is the traditional signature of sleepy tests in conventional software applications, our exploration of the Unity-based framework revealed more domain-specific patterns leading to the same smell. In VR applications, developers frequently employ constructs like `After(x).Seconds`, `yield return new WaitForSeconds(x)`, and even domain-specific wrappers around time-based behaviors (e.g., `Wait(x)`) to simulate delays, test coroutine flows, or wait for animations to complete.

```
[UnityTest]
public IEnumerator ShouldPauseSound(){
...
yield return new WaitForSeconds(TestConstants.WAIT
    ↪ _TIME);
...
}
[Test]
public void TestToConfig(){
...
TestUtility.CreateConfigFileForTest(AvatarId, "
    ↪ Test Avatar", avatarConfig);
After(4).Seconds;
var config = new OscAvatar { Id = AvatarId }.
    ↪ ToConfig();
...
}
```

Listing 4: Test cases exhibiting Sleepy Test from iamtomhewitt/jet-dashvr and ChanyaVRC/VRCOscLib

7) Lazy Test (LT)

Lazy tests are relatively common and occurring most frequently in organizational projects (380 cases), followed by independent projects (316 cases) and academic projects (162 cases). This prevalence suggests that larger test suites in structured environments might inefficiently reuse test logic, highlighting a need for more specialized unit tests.

Identifying multiple calls to the same production method within a test suite can serve as an indicator of a lazy test. However, this heuristic alone is often insufficient, as the same production method may be invoked with different input data, representing distinct test scenarios. In VR test cases, it is common to reuse the same production object and repeat method calls across varied contexts. To address this challenge, we enhanced our approach by incorporating Data Flow Graph (DFG) analysis, which enables a more nuanced understanding of data usage and method invocation patterns. This enhancement significantly improves the detection of lazy test smells in VR applications by capturing subtle reuse behaviors that are not apparent through method-level analysis alone.

8) General Fixture (GF)

General Fixture occurrences are more frequent in organizational projects, with 595 cases, likely due to the complexity of their setup procedures. Independent projects reported 312 cases, while academic projects had 104 cases. The presence of unused setup components suggests a need for codebase refactoring to improve the test resource initialization process.

9) Magic Number (MN)

This smell was highly frequent, particularly in independent projects, with 1,171 cases, suggesting a greater use of hardcoded values in VR simulations. Organizational projects reported fewer cases, at 606, likely due to higher coding standards and a preference for configuration-driven setups. Academic projects had a moderate number of occurrences, with 172 cases, indicating some reliance on fixed values.

10) Ignored Test (IT)

Only a few ignored tests were observed in organizational projects, with 83 cases, followed by a minimal number in independent projects (9 cases) and academic projects (3 cases). These instances indicate situations where developers temporarily wrote and then disabled test cases, often because the associated features were unstable or under active development.

11) Exception Throwing Test (ETT)

Exception-throwing tests were infrequently written, with a modest presence across all project categories: 62 in academic projects, 45 in independent projects, and 151 in organizational projects. This limited use suggests a preference in VR testing for behavior verification via assertions rather than explicitly expecting and handling exceptions. It also reflects that most test scenarios are designed around visual behavior or state validation, which is more prevalent in interactive VR environments than exception-based control flow.

12) Empty Test (EMT)

Empty tests were nearly nonexistent, with only 6 instances in academic projects, 16 in independent projects, and 6 in organizational projects. This suggests that VR developers typically avoid writing placeholder tests, likely because Unity-based frameworks emphasize concrete behavior-driven validations over template-based scaffolding.

13) Eager Test (EGT)

Eager tests were prevalent in independent projects, with 587 instances, and even more so in organizational projects, with 1,061 instances. These tests often validate multiple behaviors simultaneously within a single test method. While this approach can streamline test writing, it may lead to test brittleness, especially in VR scenarios involving complex user interface flows or object transformation sequences. Such complexity can make isolating faults challenging, as failures may not clearly indicate which component is at fault.

14) Duplicate Assert (DA)

This smell was moderately frequent in organizational projects, with 261 instances detected. This implies that VR developers rarely write exact duplicate assertions.

```
[UnityTest]
public IEnumerator SwitchInactiveComponent(){
    ...
    Assert.IsTrue(objectA.activeInHierarchy);
    ...
    subject.SwitchNext();
    Assert.IsTrue(objectA.activeInHierarchy);
    ...
}
```

Listing 5: VR test case with Duplicate Assertion from ExtendRealityLtd/Zinnia.Unity

Listing 5 shows a VR test case where developers added `Assert.IsTrue(objectA.activeInHierarchy)` twice. However, these assertions are not considered duplicate assertions. Typically, they assert the initial state of an object, perform an action, and then assert again to verify the change. These are technically counted as duplication, but this assertion pattern duplication is purpose-driven, reflecting a domain-specific testing pattern commonly used to validate state transitions in VR applications.

15) Constructor Initialization (CONI)

Constructor initialization was rare, with only 41 instances detected: 18 in organizational projects, 13 in independent projects, and 10 in academic projects. Few projects opted to initialize shared resources directly through constructors. Instead, the majority preferred using the structured setup methods provided by the Unity testing framework, which offer more flexibility and clarity in test preparation.

16) Default Test (DT)

Default tests were rarely found, with only 8 instances detected in all VR projects. This suggests that test developers typically write their test methods to be more meaningful, which enhances test reporting and debugging. Meaningful tests provide clarity on the test's purpose and make it easier to identify and address issues when they arise.

17) Mystery Guest (MG)

Mystery Guest occurrences were modest, with a total of 270 instances observed, mainly in independent projects. This smell suggests that certain tests rely on external resources, such as databases or file systems, which can introduce brittleness and complicate debugging, particularly in distributed environment.

B. RQ2: Effectiveness of `VRTestSniffer`.

`VRTestSniffer` was evaluated using the dataset described in Section III-A4, which consists of 553 test cases. These include both smelly and non-smelly instances across the 17 test smell categories targeted by `VRTestSniffer`. To assess the effectiveness of our tool, we employed widely used evaluation metrics: precision, Recall, and F1-Score. These metrics are based on the classification outcomes using the following definitions:

- **True Positive (TP)**: A smelly test case correctly identified as smelly.
- **True Negative (TN)**: A non-smelly test case correctly identified as non-smelly.
- **False Positive (FP)**: A non-smelly test case incorrectly classified as smelly.
- **False Negative (FN)**: A smelly test case incorrectly classified as non-smelly.

Precision measures the proportion of correctly identified smelly test cases among all those predicted as smelly. Recall quantifies the proportion of correctly identified smelly test cases among all actual smelly cases. F1-Score provides a harmonic mean of Precision and Recall, offering a balanced evaluation of detection performance.

Table III summarizes the per-category performance results of `VRTestSniffer` in terms of these metrics. As shown, `VRTestSniffer` achieves consistently high scores across all categories, with an overall weighted average Precision of 96.18%, Recall of 95.58%, and F1-Score of 95.61%, demonstrating the robustness of our static and data flow analysis-based detection approach.

TABLE III: Evaluation Results of `VRTestSniffer`

Test Smell	Precision (%)	Recall (%)	F1 Score (%)
Assertion Roulette	96.77	90.91	93.75
Sensitive Equality	100.0	100.0	100.0
Unknown Test	92.86	78.79	85.23
Redundant Print	100.0	100.0	100.0
Redundant Assertion	100.0	66.67	80.0
Sleepy Test	100.0	100.0	100.0
Lazy Test	93.30	100.0	96.50
General Fixture	93.55	100.0	96.72
Magic Number	90.32	96.55	93.0
Ignored Test	100.0	100.0	100.0
Exception Throwing Test	100.0	100.0	100.0
Empty Test	100.0	100.0	100.0
Eager Test	84.21	100.0	91.20
Duplicate Assert	84.0	100.0	91.30
Constructor Initialization	100.0	100.0	100.0
Default Test	100.0	100.0	100.0
Mystery Guest	100.0	87.50	93.33
Average	**96.18**	**95.58**	**95.61**

To conduct a comparative evaluation, we benchmarked `VRTestSniffer` against the only existing VR-specific test smell detection tool, proposed by Rzig et al. [11]. We used the

original, unmodified artifact from Rzig et al. [11] to ensure fairness in evaluation. Their tool supports only six test smell categories: *Assertion Roulette*, *Eager Test*, *General Fixture*, *Mystery Guest*, *Lazy Test*, and *Sensitive Equality*. Since the remaining eleven categories addressed by `VRTestSniffer` are not supported by Rzig et al.'s approach, the comparison is limited to the overlapping set of smells. To ensure consistency, we executed Rzig et al.'s tool on our evaluation dataset consisting of 387 annotated test cases. Table IV presents the performance metrics—Precision, Recall, and F1-Score—for both tools across the six common smell categories. As the results indicate, `VRTestSniffer` outperforms the baseline in all evaluated categories, with particularly large improvements observed for *Lazy Test* and *Mystery Guest*, where Rzig et al.'s tool exhibits notably low recall. The superior performance of `VRTestSniffer` is primarily attributed to its integration of static pattern analysis with fine-grained control flow and data flow analysis, enabling more accurate and context-aware identification of smell instances in VR test code. While `VRTestSniffer` demonstrates comparable performance to state-of-the-art test smell detection tools such as `tsDetect` and `PyNose`, it still produces some false positives and false negatives. Our analysis revealed that the most common failure cases arise from loop-based assertions, runtime-dependent test assertions, semantic data flows across coroutine frames, and complex dependencies, such as mocking frameworks for .NET.

C. RQ3: Association Between Test Smells and Code Smell

To address this research question, we conducted a manual analysis as discussed in Section III-B to identify associations between VR test smells and code smells. For some smell categories, including Sleepy Test, Exception Test, Sensitive Equality, Ignored Test, Empty Test, Constructor Initialization, Default Test, Redundant Print, and Redundant Assert, the number of detected test smell instances was too small. Specifically, the number of test smells samples available for analysis in these categories was fewer than ten. With such small samples, calculating the Odds Ratio can be unstable, inflated, or undefined, and thus may not reliably reflect the true association. Therefore, categories with fewer than ten samples were grouped into the Other Smell category. The overall relationships observed between test smells and code smells are shown in Table V.

The associations were derived based on observable structural, behavioral, or semantic dependencies between the smelly test logic and the design flaws in the corresponding code units. For instance, test smells that reflect inadequate test isolation or poor assertion practices often correspond to code smells like long methods, large classes, or inappropriate intimacy. In the following subsections, we present a detailed breakdown of each test smell category and its most commonly associated code smells, providing insight into how poor test design may stem from or contribute to problematic code structures.

1) Assertion Roulette

Assertion Roulette was the most frequently observed test smell in our analysis. We found the highest odds ratio (OR) for Blob classes (OR=3.57), followed by complex classes (OR=1.78). In Unity, Blob classes—expansive scripts like a single VR controller handling multiple responsibilities—often require tests that cover several systems or behaviors at once. Due to unclear separation of duties, these tests include multiple assertions for various VR features (e.g., teleportation, object grabbing, UI selection) in a single method, resulting in assertion roulette. Similarly, complex classes such as MonoBehaviour manage multiple states and behaviors, leading to tests with many assertions to confirm that changes don't disrupt the VR experience. As a result, test failures make it difficult to identify the specific issue, complicating and slowing debugging, much like in traditional software.

2) Duplicate Assert

Duplicate Assert appeared in 18 instances without any associated code smell, indicating that this test smell can often arise independently of issues in the production code. However, we also observed co-occurrences with Complex Class (OR=1.47), suggesting that the presence of large, monolithic, or structurally complex classes may encourage developers to repeat assertions across similar object structures or behaviors. However, such an association is not statistically significant.

3) Lazy Test

Compared to Assertion Roulette, Lazy tests exhibited stronger associations with various code smells. Out of the 24 sampled cases, 9 were found in otherwise clean code. In contrast, the remaining cases showed significant co-occurrence with code smells such as Complex Class (4 instances), Class Data Should Be Private (2 instances), and a combination of Blob and Complex Class (3 instances). VR classes often act as large manager components that couple multiple responsibilities (e.g., input, physics, rendering) and rely heavily on Unity's lifecycle methods. At the same time, encapsulation barriers, such as private or serialized fields, limit direct access to internal states. As a result, developers frequently resort to writing shallow tests that only verify high-level behaviors and invoke the same lifecycle method. This tendency leads to a strong co-occurrence between these structural code smells and the presence of Lazy Test smells.

4) Mystery Guest

Mystery Guest tests often depend on external components such as files, shared databases, or other global states that are not clearly defined within the test. We observed strong co-occurrence with structural code smells, particularly **Blob, Complex** (6 instances, OR=31.17) and **Complex Class** (3 instances, OR=7.57). VR manager classes (e.g., SceneManager, InputManager) often depend on multiple subsystems (physics, rendering, input, assets). Tests validating such behavior often need to load scenes or initialize objects indirectly, introducing implicit dependencies outside the test's scope.

5) General Fixture

For General Fixture, we found the highest Odds Ratio for the combination of Blob and Complex Class (OR=1.93) and only Blob Class (OR=1.17). Based on our analysis, in VR projects, General Fixture often arises when Blob or Complex Classes (e.g., scene managers or controller classes)

TABLE IV: Performance Comparison of Rzig et al. [11] and VRTestSniffer

Test Smell	Rzig et al. [11]			VRTestSniffer		
	Precision (%)	Recall (%)	F1 Score (%)	Precision (%)	Recall (%)	F1 Score (%)
Assertion Roulette	96.23	100.0	98.09	96.77	90.91	93.75
Eager Test	62.5	100.0	76.12	84.21	100.0	91.2
Lazy Test	0.0	0.0	0.0	93.3	100.0	96.50
Mystery Guest	100.0	50.0	66.67	100.0	87.5	93.3
Sensitive Equality	100.0	100.0	100.0	100.0	100.0	100.0
General Fixture	100.0	50.0	66.7	93.55	100.0	96.5
Average	**76.46**	**66.67**	**67.93**	**94.64**	**94.40**	**95.21**

TABLE V: Test Smell and Code Smell Co-Occurrence Mapping

Test Smell	Sample Size	Blob	Complex Class	Functional Decomposition	Class Data Should Be Private	Blob, Complex	Blob, Functional Decomposition	Spaghetti
Assertion Roulette	148	n=8 OR=3.57	n=12 OR=1.78	n=3 OR=inf	n=2 OR=1.72	n=9 OR=1.03	n=0 OR=0	n=0 OR=inf
Eager Test	50	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=1 OR=0.29	n=1 OR=inf	n=0 OR=inf
Lazy Test	24	n=0 OR=0	n=4 OR=3.58	n=0 OR=0	n=2 OR=17.09	n=3 OR=2.42	n=0 OR=0	n=0 OR=inf
Duplicate Assert	24	n=0 OR=0	n=2 OR=1.47	n=0 OR=0	n=0 OR=0	n=1 OR=0.67	n=0 OR=0	n=0 OR=inf
Magic Number	54	n=1 OR=0.57	n=1 OR=0.26	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=inf
Mystery Guest	10	n=0 OR=0	n=3 OR=7.57	n=0 OR=0	n=0 OR=0	n=6 OR=31.16	n=0 OR=0	n=0 OR=inf
Unknown Test	39	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=0 OR=inf
General Fixture	29	n=1 OR=1.17	n=0 OR=0	n=0 OR=0	n=0 OR=0	n=3 OR=1.93	n=0 OR=0	n=0 OR=inf
Other Smells	24	n=2 OR=3.35	n=2 OR=1.47	n=0 OR=0	n=0 OR=0	n=1 OR=0.67	n=0 OR=0	n=0 OR=inf

Notes: n = frequency; OR = Odds Ratio.
Other Smells: Other Smell Includes All Smell Categories That Have Sample Size <10
inf = Odds ratio infinite (outcome in only one group).

centralize many responsibilities. Testing these classes requires initializing broad scene objects and devices, leading developers to create large, generic fixtures where only parts are used in individual tests. This mirrors the code's complexity and explains the association.

6) Minor Observations of Test and Code Smell Association

Other Smell category, which includes *Sleepy Test*, *Exception Test*, *Sensitive Equality*, *Ignored Test*, *Empty Test*, *Constructor Initialization*, *Default Test*, *Redundant Print*, and *Redundant Assert*, appeared with relatively low frequency. However, this category showed an association with *Blob* (OR=3.35) and *Complex Class* (OR=1.47). Meanwhile, **Eager Test** occasionally aligned with complex production code (1 *Blob and Complex*, 1 *Blob and Functional Decomposition*), largely due to a lack of modularity and cohesion in production classes. **Magic Number** was mostly independent of production code quality and instead reflected poor test design practices, such as the use of hardcoded literals like object location, move direction, etc. Finally, the **Unknown Test** smell category did not exhibit any association with code smell categories.

As with several other test smell categories, there were instances where no associated code smell was detected, indicating no direct relationship in those cases. However, a notable pattern emerges when examining the co-occurrence of Blob and Complex Class smells. Specifically, whenever a class is identified as a Blob in relation to any test smell, it also frequently exhibits characteristics of a Complex Class. This observation aligns with established principles of software design: Classes that grow excessively large and take on multiple responsibilities often become inherently more complex. The consistent co-occurrence of these two code smells reinforces the idea that bloated classes tend to accumulate internal complexity, making them more difficult to test and maintain. Surprisingly, we did not find any co-occurrence between test smells and spaghetti Code. Based on our observation, spaghetti code is less prevalent in VR projects because VR development environments (e.g., Unity) encourage a component-based, scene-driven, and event-driven architecture.

V. Threats to Validity

Construct Validity. The analysis in this research is based on the test smells identified by VRTestSniffer. The primary threat to construct validity lies in the reliability of VRTestSniffer. To identify test smells in VR projects, we followed the state-of-the-art tool tsDetect [19] and selected 17 of the 19 test smells discussed in their study. To further validate the correctness of our tool, we conducted a comprehensive evaluation across all smell categories. The results showed an overall weighted average F1-score of 95.61%, demonstrating the robustness of VRTestSniffer. Therefore, we believe that the analysis based on the test smells identified by VRTestSniffer reliably represents VR test smell behavior. However, since VRTestSniffer relies on static analysis and is restricted to detecting test smells documented in existing literature, it cannot capture runtime or VR-specific test smells, which constitutes a threat to construct validity. Although our manual validation confirmed our tool's accuracy (RQ2), the co-occurrence analysis (RQ3) would profit from a broader analysis. This would help collect a wider range of smells, addressing the dataset's current imbalance, and so, enhancing the statistical significance of the findings.

Internal Validity. The primary threat to the internal validity of this research is the correctness of our manual analysis on the association between test smells and code smells. To mitigate this threat, we had two independent inspectors perform the

mapping. They then participated in a reconciliation meeting to resolve any conflicts that occurred.

External Validity. The main concern regarding external validity is the representativeness of the projects studied. Our analysis is based on 314 open-source Unity-based VR projects, and therefore it does not represent other VR engines or frameworks not utilizing Unity for development. This dataset, which has been used in prior studies and is well-maintained, includes only projects with more than 100 commits. Additionally, the dataset features open-source projects backed by large organizations. Therefore, we believe that the insights derived from our research represent Unity-based VR projects. While the current version targets Unity, VRTestSniffer's architecture can be extended to support other frameworks.

VI. Implications

For VR Developers. The results from RQ1 reveal that VR test cases frequently contain a significant number of test smells, which can hinder developers' ability to effectively validate and maintain VR applications. Furthermore, RQ3 demonstrates that for a few test smell categories, there are associations between test smells and code smells, particularly with design issues such as Blob and Complex classes. These empirical findings highlight the intertwined nature of test and production code quality in VR systems. By identifying these associations and patterns, our study offers actionable insights that can help VR developers design cleaner and more maintainable test cases, ultimately improving overall software quality.

For VR Tool Builders. Results from RQ2 demonstrate the need for specialized, tailored VR testing tools. Our detection of Unity-specific patterns (e.g., WaitForSeconds(x), After(x).Seconds) indicates that conventional tools often miss context-specific smells. This observation, along with the context information gathered, can aid future tool developers in designing tools for VR test case generation, as well as more sophisticated tools for VR test smell detection and repair.

For Software Engineering Researchers: The findings of our study open up new avenues for research in VR software testing. Results from RQ1 demonstrate that test smells are highly prevalent in VR projects. These insights can motivate software engineering researchers to explore and develop advanced tools and techniques for automated test generation or test augmentation, aimed at mitigating test quality issues specific to VR applications.

VII. Related Work

Software testing is a critical component of the software development lifecycle, yet it remains time-consuming and often under-prioritized. Test code is frequently not as well-structured or maintained as production code, which can negatively impact software quality over time. Van Deursen et al. [17] first identified 11 test smell types that arise from poor design patterns. Since then, the catalog of test smells has been expanded by several researchers [31], [32], [33], [34]. While much of the early work focused on Java-based test smells, subsequent studies have investigated test smells in other languages and domains. For example, Wang et al. [20] proposed `PyNose`, which detects 17 common test smells as well as one Python-specific smells in Python-based test suites, while Bleser et al. [35], [36] explored test smells in Scala projects. Bavota et al. [37], [38] conducted comprehensive studies showing that test smells are widespread and can significantly reduce the readability and maintainability of both test suites and associated production code. Despite these advancements, prior research has limited study examining test smell in VR applications. Recent efforts have begun to explore various challenges in VR development and testing. Truelove et al. [39] performed an empirical analysis of 12,122 bug fixes across 723 updates from 30 popular VR games. Politowski et al. [40] surveyed game developers to understand testing practices in game development. Similarly, Nusrat et al. [41] analyzed VR applications to classify performance-related bugs, while several works [42], [11] reviewed the limitations of current VR testing practices and suggested directions for improvement. In terms of tooling, Wang et al. [43] proposed a testing framework that automates scene interaction in VR, and Harms [44] introduced an automated usability evaluation approach for VR environments. Qualitative studies also contribute valuable insights: Ashtari et al. [45] interviewed 21 AR/VR developers to identify testing challenges, and Vlahovic et al. [46] reviewed factors affecting VR experience quality and future directions. While prior work has addressed VR testing challenges and practices, none have solely focused on VR test smells.

VIII. Conclusion

In this work, we presented a comprehensive approach for detecting and analyzing test smells in Unity-based VR applications, a domain where automated testing practices remain underexplored. We first performed a mapping study to evaluate the applicability of 19 well-established test smell categories to VR applications and found that 17 of them are relevant. Based on this mapping, we developed `VRTestSniffer`, a static analysis tool that leverages AST, CFG, and DFG representations to detect VR-specific test smells with improved accuracy. Our empirical evaluation shows that `VRTestSniffer` outperforms existing tools, achieving a 27.28% improvement in F1-score. To further understand the impact of test smells, we conducted an empirical study analyzing their association with code smells in real-world VR projects. The findings reveal that for some test smell categories, there are associations with code smells such as Blob and Complex Classes. Overall, this work provides valuable tools, and empirical evidence for VR developers and researchers aiming to improve the testability of VR applications. In the future, we plan to develop more advanced test smell detection techniques that can identify runtime and VR-specific test smell categories while supporting a wider variety of VR frameworks by capturing event-driven, co-routine-based, and sensor interactions.

ACKNOWLEDGMENTS

The project was supported by NSF Awards #2152819, #2213763, and UofM-Dearborn Research Initiation & Development Grant.

REFERENCES

[1] VRTestSniffer, "Replication package," 2025. [Online]. Available: https://figshare.com/s/936fe001590dac84cd3a

[2] M. N. J. Dani, "Impact of virtual reality on gaming," *Virtual Reality*, vol. 6, no. 12, pp. 2033–2036, 2019.

[3] S. Kavanagh, A. Luxton-Reilly, B. Wuensche, and B. Plimmer, "A systematic review of virtual reality in education," *Themes in science and technology education*, vol. 10, no. 2, pp. 85–119, 2017.

[4] H. Ardiny and E. Khanmirza, "The role of ar and vr technologies in education developments: opportunities and challenges," in *2018 6th rsi international conference on robotics and mechatronics (icrom)*. IEEE, 2018, pp. 482–487.

[5] E. F. Foundation, "Virtual reality tour of surveillance tech," 2025. [Online]. Available: https://www.eff.org/deeplinks/2024/02/virtual-reality-tour-surveillance-tech-border-conversation-dave-maass-electronic

[6] J. E. Naranjo, D. G. Sanchez, A. Robalino-Lopez, P. Robalino-Lopez, A. Alarcon-Ortiz, and M. V. Garcia, "A scoping review on virtual reality-based industrial training," *Applied Sciences*, vol. 10, no. 22, p. 8224, 2020.

[7] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 61–70.

[8] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 103–112.

[9] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, "Developer testing in the ide: Patterns, beliefs, and behavior," *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 261–284, 2017.

[10] A. C. Correa Souza, F. L. Nunes, and M. E. Delamaro, "An automated functional testing approach for virtual reality applications," *Software Testing, Verification and Reliability*, vol. 28, no. 8, p. e1690, 2018.

[11] D. E. Rzig, N. Iqbal, I. Attisano, X. Qin, and F. Hassan, "Virtual reality (vr) automated testing in the wild: A case study on unity-based vr applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1269–1281.

[12] F. Freitas, H. Oliveira, I. Winkler, and M. Gomes, "Virtual reality on product usability testing: A systematic literature review," in *2020 22nd Symposium on Virtual and Augmented Reality (SVR)*. IEEE, 2020, pp. 67–73.

[13] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 571–579. [Online]. Available: https://doi.org/10.1145/1062455.1062556

[14] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 311–322.

[15] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Softw. Engg.*, vol. 16, no. 3, p. 325–364, Jun. 2011. [Online]. Available: https://doi.org/10.1007/s10664-010-9143-7

[16] Q. Liu, G. Alves, and J. Zhao, "Challenges and opportunities for software testing in virtual reality application development," *Graphics Interface 2023-second deadline*, 2023.

[17] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 2001, pp. 92–95.

[18] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 4–15. [Online]. Available: https://doi.org/10.1145/2970276.2970340

[19] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1650–1654.

[20] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "Pynose: a test smell detector for python," in *2021 36th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, 2021, pp. 593–605.

[21] A. P. Doucet Lars, "Game engines on steam: The definitive breakdown," Sep 2021. [Online]. Available: https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown

[22] M. Martinez, L. Duchien, and M. Monperrus, "Automatically extracting instances of code change patterns with ast analysis," in *2013 IEEE international conference on software maintenance*. IEEE, 2013, pp. 388–391.

[23] M. D. Feist, E. A. Santos, I. Watts, and A. Hindle, "Visualizing project evolution through abstract syntax tree analysis," in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2016, pp. 11–20.

[24] J. Zhang, X. Wang, H. Zhang, H. Sun, X. Liu, C. Hu, and Y. Liu, "Detecting condition-related bugs with control flow graph neural network," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1370–1382.

[25] L. Mei, W. Chan, and T. Tse, "Data flow testing of service-oriented workflow applications," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 371–380.

[26] M. Zhao, A. M. Pierce, R. Tan, T. Zhang, T. Wang, T. R. Jonker, H. Benko, and A. Gupta, "Gaze speedup: Eye gaze assisted gesture typing in virtual reality," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023, pp. 595–606.

[27] K. M. Sagayam and D. J. Hemanth, "Hand posture and gesture recognition techniques for virtual reality applications: a survey," *Virtual Reality*, vol. 21, pp. 91–107, 2017.

[28] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "Tsdetect: An open source test smells detection tool," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1650–1654. [Online]. Available: https://doi.org/10.1145/3368089.3417921

[29] "Srcml," 2022. [Online]. Available: https://www.srcml.org/

[30] H. Chen, P. Cohen, and S. Chen, "How big is a big odds ratio? interpreting the magnitudes of odds ratios in epidemiological studies," *Communications in Statistics—simulation and Computation®*, vol. 39, no. 4, pp. 860–864, 2010.

[31] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 322–331.

[32] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[33] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 391–400.

[34] M. Breugelmans and B. Van Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008.

[35] J. De Bleser, D. Di Nucci, and C. De Roover, "Assessing diffusion and perception of test smells in scala projects," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 457–467.

[36] C. D. R. Jonas De Bleser, D. Di Nucci, "Socrates: Scala radar for test smells," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, ser. Scala '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 22–26. [Online]. Available: https://doi.org/10.1145/3337932.3338815

[37] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 56–65.

[38] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Softw. Engg.*, vol. 20, no. 4, p. 1052–1094, Aug. 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9313-0

[39] A. Truelove, E. Santana de Almeida, and I. Ahmed, "We'll fix it in post: What do bug fixes in video game update notes tell us?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 736–747.

[40] C. Politowski, F. Petrillo, and Y.-G. Gu'eh'eneuc, "A survey of video game testing," *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pp. 90–99, 2021.

[41] F. Nusrat, F. Hassan, H. Zhong, and X. Wang, "How developers optimize virtual reality applications: A study of optimization commits in open source unity projects," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, Spain: IEEE, May 2021, p. 473–485. [Online]. Available: https://ieeexplore.ieee.org/document/9402052/

[42] A. Bierbaum, P. Hartling, and C. Cruz-Neira, "Automated testing of virtual reality application interfaces," in *Proceedings of the Workshop on Virtual Environments 2003*, ser. EGVE '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 107–114. [Online]. Available: https://doi.org/10.1145/769953.769966

[43] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1548–1560.

[44] P. Harms, "Automated usability evaluation of virtual reality applications," *ACM Trans. Comput.-Hum. Interact.*, vol. 26, no. 3, apr 2019. [Online]. Available: https://doi.org/10.1145/3301423

[45] N. Ashtari, A. Bunt, J. McGrenere, M. Nebeling, and P. K. Chilana, *Creating Augmented and Virtual Reality Applications: Current Practices, Challenges, and Opportunities*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3313831.3376722

[46] S. Vlahovic, M. Suznjevic, and L. Skorin-Kapov, "A survey of challenges and methods for quality of experience assessment of interactive vr applications," *Journal on Multimodal User Interfaces*, vol. 16, no. 3, p. 257–291, Sep 2022. [Online]. Available: https://doi.org/10.1007/s12193-022-00388-0