UltraInstinctVR: A Model-Based Tool for Automated System Testing of VR Applications

Gerry Longfils✉
University of Namur
NADI
Namur, Belgium
gerry.longfils@unamur.be

Maxime Cauz
University of Namur
NADI
Namur, Belgium
maxime.cauz@unamur.be

Arnaud Blouin
INSA Rennes
Diverse Team
Rennes, France
arnaud.blouin@irisa.fr

Xavier Devroey
University of Namur
NADI
Namur, Belgium
xavier.devroey@unamur.be

Abstract

Virtual Reality (VR) applications are increasingly used in domains such as surgical training and industrial marketing, yet their long-term adoption remains limited by the lack of effective, systematic, and reproducible testing approaches tailored to VR. To address this gap, we present UltraInstinctVR, a novel tool that automates the generation and execution of concrete system-level tests for VR applications, helping developers efficiently validate interactions and behaviors. We evaluated UltraInstinctVR against state-of-the-art automated VR testing tools across 10 open-source VR applications, where it outperformed existing approaches in detecting unique failures and uncovering real-world bugs. In this paper, we further extended this evaluation through an empirical user study with ten VR developers. Results highlight strong dependability and efficiency while identifying opportunities for improvement in other user experience dimensions. Overall, these findings position UltraInstinctVR as a promising step toward more practical, effective, and scalable VR testing. The tool is publicly available at https://github.com/snail-unamur/UltraInstinctVR, with a demonstration video available at https://youtu.be/U8DWTmXmeh4.

CCS Concepts

• **Software and its engineering → Software testing and debugging**;

Keywords

Virtual Reality, Interaction Testing, Model-based Testing, System Testing

ACM Reference Format:
Gerry Longfils, Maxime Cauz, Arnaud Blouin, and Xavier Devroey. 2026. UltraInstinctVR: A Model-Based Tool for Automated System Testing of VR Applications. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834642

ASE '26, Munich, Germany

1 Introduction

Virtual reality (VR) is an emerging technology that enables users to engage with immersive [1], computer-generated environments, typically through head-mounted displays. Nowadays, these types of applications are increasingly adopted in the market, with a wide variety of use cases emerging (*e.g.,* surgical training, gaming, data visualization tool). However, software engineering practices and tools supporting the development of such applications (*e.g.,* standard frameworks, integrated development environments, dependency management, and automated tools) are still evolving and remain relatively immature, as highlighted by Andrade et al. [2].

Currently, the testing of VR applications largely relies on manual processes [2], as no widely adopted tools exist to automate the testing workflow. To address this limitation, we propose UltraInstinctVR, a novel Model-Based Testing (MBT) approach that automates the generation and execution of system-level tests for VR applications. The generated tests target failures related to common VR user interactions, including selection, locomotion, and object manipulation. The tool is distributed as an open-source Unity package that can be directly imported into VR projects and is publicly available at https://github.com/snail-unamur/UltraInstinctVR

Our approach builds on recent VR development practices that leverage multiple models throughout the development process, enabling their reuse by both researchers and industry practitioners. In particular, it leverages VR scenario models [4], which describe sequences of interactions within virtual environments and can be used to automate user actions such as navigation and object manipulation. These models are commonly employed in applications such as generating demonstration scenarios or automating behaviors in 3D environments.

2 Background and Motivation

Andrade et al. [2] highlight the lack of systematic software testing practices in VR, showing that testing is less consistently integrated into the development lifecycle than in non-VR systems, thereby increasing the likelihood of defects. In a follow-up study, they adopt

a stakeholder perspective [3] and propose a taxonomy of VR faults (*e.g.*, performance, networking, stability), identifying those most impactful for users and developers.

Model-Based Testing (MBT) automates test generation from models but is limited by the cost of model creation [13]. GUI testing mitigates this by extracting runtime models (GUI ripping), which infers interaction models and reduces manual effort. In VR, scenario-based approaches such as #SEVEN [4] use formal models (*e.g.*, Petri nets) to define interaction sequences that can be reused for both test generation and oracle definition. Xareus, a Unity plug-in, extends this idea to more complex and collaborative VR interactions, though with increased modeling effort [6].

Automating the testing process, VRGreed [14] and VRGuide [15] simulate navigation and interaction while improving exploration efficiency via dynamic cut coverage. However, their fault detection capabilities remain largely limited to runtime exceptions. PLUME [8] supports interaction recording and replay but requires manual analysis and is limited to single-user scenarios. The iv4XR framework [10] provides an agent-based approach for testing interactive systems, though it is not yet fully adapted to VR. XRIntTest [7] automates VR interaction testing by generating interaction sequences, improving coverage of 3D interactions. Finally, VRExplorer [16] enables systematic exploration using the Entity–Action–Task model and demonstrates improved coverage and broader fault detection.

Motivation. Although existing automated testing tools focus on interaction generation, their fault detection is largely limited to crash and non-functional failures. To address functional failures, we propose UltraInstinctVR, a model-based testing approach structured around three components: automated interaction generation via agents, interaction detection and correctness verification using *Sensors* and *Effector* from *Xareus* [6].

3 Approach & Architecture

In this section, we present the workflow of a testing campaign using UltraInstinctVR, focusing on configuration, execution, and oracle verification through a simple teleportation scenario. As shown in Figure 1, the tester first defines tests agents (A), each assigned a task that triggers specific VR interactions through interaction with scene objects. The tester then specifies corresponding oracles as Petri nets (B) to validate agent behavior. Finally, UltraInstinctVR executes each agent (C) according to its defined behavior and evaluates the associated oracle (D).

Regarding implementation, we use *Unity6* [12] due to its widespread adoption in the VR community and *Xareus* [6], a Unity plug-in for oracle specification, as it uniquely supports complex VR behavior modeling using a Petri net abstraction. UltraInstinctVR is distributed as a Unity package that automatically imports all required scripts and components; the tester then adds the parent *GameObject* to the scene to enable sequential and independent execution of test agents.Documentation and installation details are available in our GitHub repository, while Xareus licensing information is available on its official website https://xareus.insa-rennes.fr/?tabs=air We also provide a template test agent that testers can reuse to reduce the effort required to design and implement new tests.

```
1
2  public override void SafeReset()
3  {
4      // Initialize the last position to the user's initial
           position
5      lastPosition = GetPlayerPosition();
6  }
7  public override Result UnityStepSensorCheck() {
8    Vector3 currentPosition = GetPlayerPosition();
9    // Interaction detection
10   if (Vector3.Distance(currentPosition, lastPosition) >
         teleportDistanceThreshold){
11     // Assign a teleportation key for retrieving the
           interaction later
12     string teleportKey = $"{TELEPORT_KEY}_{Guid.NewGuid()}";
13     // Add the context  to Xareus
14     eventContext.Add(teleportKey, currentPosition.ToString());
15     lastPosition = currentPosition;
16     return new Result(true, eventContext);
17   }
18   return new Result(false, eventContext);
19 }
```

Listing 1: Sensor implementation

```
1  public override void SafeEffectorUpdate(){
2    Vector3 currentPosition = GetPlayerPosition();
3    Debug.Log("TestGenerated - Teleportation");
4    // Represent the ASSERT in the effector
5    if (Vector3.Distance(currentPosition, lastPosition) >
         teleportDistanceThreshold){
6      Debug.Log("ORACLE CanTeleport - TestPassed [...]" );
7      lastPosition = currentPosition;
8    }
9    else{
10     Debug.LogError("ORACLE CanTeleport - Failed - [...]");
11   }
12 }
```

Listing 2: Effector implementation

Defining a test agent (A). The tester defines a test agent by creating it as a `GameObject` in Unity and configuring a wide range of parameters, including collider size, number of locomotion attempts, collision handling, locomotion behavior, and other interaction mechanisms. In total, more than 100 parameters can be adjusted to customize the agent's behavior in the VR environment. Exploring how combinatorial interaction testing can be applied to those parameters to amplify the tests is part of our future work.

In our running example, the tester creates a `GameObject` representing the agent and imports all required components (*e.g.*, `XR Interaction Manager`) as well as the basic components needed to track the position and state of relevant `GameObjects`. The tester must also implement interaction scripts in C#, which define how automated interactions are executed and specify the behavior of the test agent within the environment. For example, in our running example, the tester configures the test agent's position to reach the desired target position.

In the current implementation, readily usable test agent behaviors are organized into two main categories: locomotion-related use cases (*e.g.*, valid and invalid teleportation scenarios) and interaction tasks (*e.g.*, object selection, grabbing, movement, and collision). Together, these predefined behaviors enable systematic exploration of both navigation and interaction scenarios within VR applications.

Defining the oracle (B). To define an oracle, and as shown in Listing 1, the tester first implements a `Sensor` *i.e.,* a function executed at each frame update that monitors the virtual environment (*e.g.*, the VR space, `GameObjects`, and test agents) using the *Xareus* plug-in, When the sensor condition is satisfied, it triggers a transition in

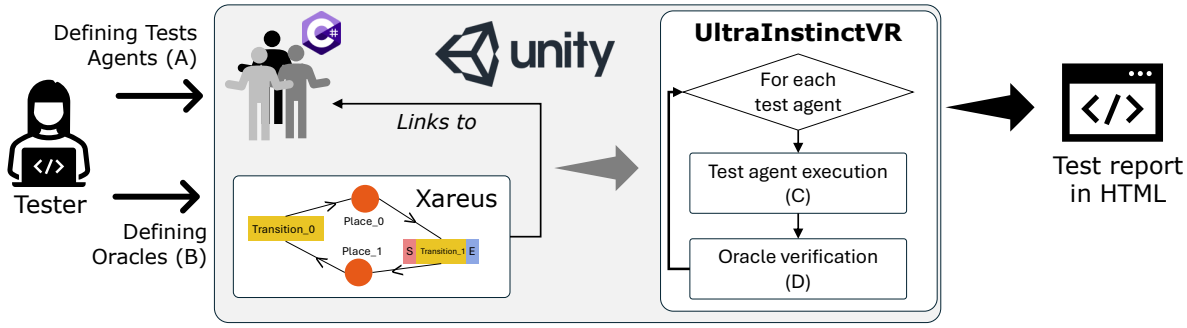

Figure 1: Flow of a testing campaign using UltraInstinctVR

the underlying Petri net, which activates an associated `Effector` in *Xareus*. As illustrated in Listing 2, the `Effector` implements the oracle mechanism by validating the correctness of detected interactions. This process is modeled using a cyclic Petri net created by the tester using the plug-in *Xareus*, where each transition is associated with a specific `Sensor` and `Effector`.

In practice, for each transition, the tester implements two C# files: one for the `Sensor` and one for the `Effector`. Using templates provided by *Xareus* [6], the tester defines in the `SafeReset()` method which variables are monitored, and in the `UnityStepSensorCheck()` method how these variables are evaluated to trigger the corresponding `Effector`. For the `Effector`, the tester specifies in the `SafeEffectorUpdate()` method the conditions under which the oracle passes or fails. Petri nets may be simple, as in our running example, or more complex, depending on the testing requirements and the system under test.

Test agent execution (C). Once the tester has defined the test agents and their associated oracles, the testing campaign can be initiated by executing the Unity application. During runtime, agents are executed sequentially, each independently performing its behavior within the VR environment. Technically, each test agent is implemented as a sub-`GameObject` of a parent `GameObject`. The parent `GameObject` coordinates the execution of its children, activating each agent in turn after the previous one completes. This mechanism is implemented using a thread pool, where each child is added to the pool and executed sequentially and independently by dedicated threads.

Oracle verification (D). Once triggered, an `Effector` evaluates a conditional assertion to determine whether the test passes or fails. In our example, we verify that the test agent can teleport in the VR environment.

4 Evaluation

In our previous study [9], we empirically compared UltraInstinctVR with other state-of-the-art VR testing tools across ten open-source projects in terms of line coverage, interaction coverage, and unique failure detection, and we also conducted a root cause analysis of failures detected by UltraInstinctVR in two projects. Results show that UltraInstinctVR achieves line coverage ranging from 12% to 25%, which is comparable to state-of-the-art approaches such as VRGuide (between 15% and 23%), VRGreed (between 12% and 23%), VRExplorer (between 10% and 12%), and XRIntTest (between 8% and 15%). In terms of interaction coverage, UltraInstinctVR reaches between 27% and 72%, while competing tools range from 0% to 75%. Overall, these findings show that UltraInstinctVR delivers competitive coverage performance across key metrics while demonstrating stronger effectiveness in several specific scenarios. In failure detection, UltraInstinctVR identified 30 unique failures, outperforming *VRGreed* (17), *VRGuide* (16), *XRIntTest* (13), and *VRExplorer* (8), while sharing substantial overlaps with all tools. Although it missed a few *XRIntTest* specific failures, it was overall the most effective due to its advanced oracle design. Root cause analysis identified three representative failures: a gaze interaction defect fixed by disabling the *XRGazeInteractor*, an interaction fault where agents passed through *GameObjects* resolved with a *RigidBody* based collision prevention script, and a *NullReferenceException* caused by misconfigured rigid bodies and missing components, fixed by disabling the faulty component. According to Andrade's fault model, the first and third defects were classified as stability faults.

In this paper, we further extended our quantitative evaluation with a developer study to assess effectiveness and user experience by using the standard *User Experience Questionnaire* (UEQ) [11]. Participants were tasked with identifying and fixing defects based on failures detected by UltraInstinctVR in a VR application we developed. A total of 10 participants aged 20–58 years took part in the study, including student interns, developers, researchers, and academic professors, all working within a computer science research laboratory.

Developer evaluation setup. We used `BlocEmpiler`, which is available in our replication package at https://zenodo.org/records/20054137. Developing our own application gave us full control over included interactions and defects, such as interaction-related bugs targeted by UltraInstinctVR, unrelated issues (*e.g.*, `NullPointerReferences`), missing components, and runtime warnings. The application contains 7,110 lines of code and 282 game objects.

Participants were first introduced to the software under test. They then received a comprehensive presentation on UltraInstinctVR, along with a detailed `README` file that provided documentation to support their use throughout the session. They first had to run UltraInstinctVR, and after that, they were given 40 minutes to identify and fix as many defects as possible. After the session, they

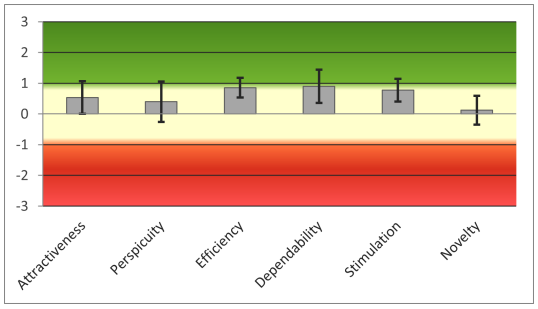

Figure 2: UEQ scales (Mean and Error bar)

completed the UEQ questionnaire to assess user experience across six dimensions: attractiveness, perspicuity (ease of use), efficiency, dependability (control over interactions), stimulation (motivation), and novelty of the tool.

During the evaluation sessions, researchers passively observed participants' interactions with the system, recording the number of defects they identified and successfully resolved. Several participants also voluntarily provided qualitative feedback on their experience using UltraInstinctVR.

Evaluation results. As summarized in Figure 2, UltraInstinctVR received an overall positive assessment. According to the UEQ handbook [11], mean values between −0.8 and 0.8 indicate neutral user experience, while values above this range reflect positive evaluations. In our results, **efficiency** (mean = 0.850, variance = 0.27) and **dependability** (mean = 0.900, variance = 0.77) were positively rated, whereas **perspicuity** (0.400, 1.13), **stimulation** (0.775, 0.35), **novelty** (0.125, 0.57), and **attractiveness** (0.553, 0.73) remained neutral. These results show that our tool received positive evaluations for **efficiency** and **dependability**, while the remaining metrics were assessed as neutral. Regarding defect detection and correction, participants identified 62.85% of the ten injected defects, in addition to the default defect present in the application, and successfully fixed 47.14% of these defects, along with the default defect.

5 Discussion

Participants feedback. Participants reported positive impressions of UltraInstinctVR in terms of efficiency and dependability, while attractiveness, stimulation, perspicuity, and novelty were rated as neutral. To improve perspicuity, participants suggested reducing console log verbosity and enriching test reports with contextual information, such as the `TestAgent` name and the component associated with detected defects. The `TestAgent` name can be easily included since the executed agent is always known, whereas identifying the faulty component is more complex: Unity logs may sometimes expose it directly, but in other cases this information is absent, particularly for oracle failures, where a dedicated defect localization mechanism would be required. Improving attractiveness and stimulation remains challenging due to the task-oriented nature of testing tools, although these aspects could be valuable for educational use cases, where inspiration could be drawn from platforms such as *Inginious* [5].

Development challenges. Several practical challenges emerged during the development of UltraInstinctVR. A key technical challenge was to ensure that test agents could run independently and sequentially without interfering with one another. This was addressed by encapsulating test agent execution within a thread pool, enabling isolated, one-by-one execution and preventing chaotic interactions during testing campaigns. Another challenge involved repurposing *Xareus*, which was originally designed to support VR application development rather than software testing tasks. This required understanding how to effectively leverage its `Sensor` and `Effector` mechanisms to define testing oracles.

Tester challenges. Finally, implementing automated VR interactions in C# is particularly challenging due to the complexity of immersive user behaviors. Creating complete automated interaction sequences, such as grabbing multiple objects, rotating them, or coordinating complex object manipulations, can be both time-consuming and inefficient for testers. For now, UltraInstinctVR comes with a set of predefined behaviors and their corresponding oracles. In our future work, we will explore how a domain-specific language can help to streamline the specification and automation of complex VR interactions, alleviating the tester from writing complex C# code.

6 Conclusion

Our research introduces UltraInstinctVR, an automated framework for testing interaction-based behaviors in VR applications by generating and executing system-level tests. User evaluations show that it effectively supports defect identification, although assistance for defect remediation remains limited. UEQ results highlight strengths in efficiency and dependability, with room for improvement in stimulation, attractiveness, and clarity. Future work will address user feedback by adding a progress indicator, encapsulating UltraInstinctVR as a dedicated test runner, and developing an embedded domain-specific language (eDSL) to support more complex interactions, including voice-based and multimodal scenarios. In addition, this eDSL helps testers to better manage large sets of parameters and facilitates more efficient parameter tuning. This improved usability may contribute to higher scores in the UEQ dimensions of perspicuity, stimulation, novelty, and attractiveness.

Data availability statement

The replication package is publicly available and contains the application used in the human experience *BlocEmpiler*, as well as the UEQ evaluation results provided in the file *UEQDataAnalysis-ToolVersion12.xlsx*. All associated data and materials can be accessed through the Zenodo record at https://zenodo.org/records/20054137, which is also available via the following DOI: https://doi.org/10.5281/zenodo.20054137.

Acknowledgments

This research was partially funded by the CyberExcellence by DigitalWallonia project (No. 2110186) funded by the Public Service of Wallonia (SPW Recherche).

References

[1] Sarvesh Agrawal, Adèle Simon, Søren Bech, Klaus Bærentsen, and Søren Forchhammer. 2019. Defining immersion: Literature review and implications for research on immersive audiovisual experiences. *AES* 68, 6 (2019), 404–417. doi:10.17743/aesconv.2019.978-1-942220-31-2

[2] Stevão A Andrade, Fatima LS Nunes, and Marcio E Delamaro. 2019. Towards the systematic testing of virtual reality programs. In *SVR 2019*. IEEE, 196–205. doi:10.1109/SVR.2019.00044

[3] Stevão A Andrade, Alvaro Joffre U Quevedo, Fatima LS Nunes, and Márcio E Delamaro. 2020. Understanding VR software testing needs from stakeholders' points of view. In *SVR 2020*. IEEE, 57–66. doi:10.1109/SVR51698.2020.00024

[4] Guillaume Claude, Valérie Gouranton, Rozenn Bouville Berthelot, and Bruno Arnaldi. 2014. Short paper:# seven, a sensor effector based scenarios model for driving collaborative virtual environment. (2014), 4 pages. https://hal.science/hal-01086237.

[5] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. 2015. Automatic grading of programming exercises in a MOOC using the INGInious platform. *European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS'15)* (2015), 86–91. doi:2078.5/229236

[6] Lysa Gramoli, Florian Nouviale, Adrien Reuzeau, Alexandre Audinot, Mathieu Risy, Tangui Marchand-Guerniou, Maé Mavromatis, Bruno Arnaldi, and Valérie Gouranton. 2025. Xareus: a Framework to Create Interactive Applications without Coding. In *VRW 2025*. IEEE, 1658–1659. doi:10.1109/VRW66409.2025.00474

[7] Ruizhen Gu, José Miguel Rojas, and Donghwan Shin. 2025. XRintTest: An automated framework for user interaction testing in extended reality applications. In *ASE 2025*. IEEE, 4013–4016. doi:10.1109/ASE63991.2025.00363

[8] Charles Javerliat, Sophie Villenave, Pierre Raimbaud, and Guillaume Lavoué. 2024. PLUME: Record, Replay, Analyze and Share User Behavior in 6DoF XR Experiences. *IEEE Transactions on Visualization and Computer Graphics* 30, 5 (2024), 2087–2097. doi:10.1109/TVCG.2024.3372107

[9] Gerry Longfils, Maxime Cauz, Arnaud Blouin, and Xavier Devroey. 2026. System Test Generation for Virtual Reality Applications using Scenario Models. *arXiv preprint arXiv:2605.07534* (2026). https://arxiv.org/abs/2605.07534

[10] ISWB Prasetya, Samira Shirzadehhajimahmood, Saba Gholizadeh Ansari, Pedro Fernandes, and Rui Prada. 2021. An agent-based architecture for ai-enhanced automated testing for xr systems, a short paper. In *ICSTW 2021*. IEEE, 213–217. doi:10.1109/ICSTW52544.2021.00044

[11] UEQ team. 2025. A fast and reliable questionnaire to measure the User Experience of interactive products. https://www.ueq-online.org/.

[12] Unity. 2025. Unity Technologies. https://learn.unity.com/.

[13] Mark Utting and Bruno Legeard. 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann. doi:10.1016/B978-0-12-372501-1.X5000-5

[14] Xiaoyin Wang. 2022. Vrtest: An extensible framework for automatic testing of virtual reality scenes. In *ICST 2022*. 232–236. doi:10.1145/3510454.3516870

[15] Xiaoyin Wang, Tahmid Rafi, and Na Meng. 2023. VRGuide: Efficient Testing of Virtual Reality Scenes via Dynamic Cut Coverage. In *ASE 2023*. IEEE, 951–962. doi:10.1109/ASE56229.2023.00197

[16] Zhengyang Zhu, Hong-Ning Dai, Hanyang Guo, Zeqin Liao, and Zibin Zheng. 2025. VRExplorer: A Model-based Approach for Semi-Automated Testing of Virtual Reality Scenes. In *ASE 2025*. IEEE, 482–494. doi:10.1109/ASE63991.2025.00047