VRExplorer: A Model-based Approach for Semi-Automated Testing of Virtual Reality Scenes

Zhengyang Zhu*‡, Hong-Ning Dai†, Hanyang Guo*, Zeqin Liao*, Zibin Zheng*
*School of Software Engineering, Sun Yat-sen University, Zhuhai, China
{zhuzhy57, guohy36, liaozq8}@mail2.sysu.edu.cn, zhzibin@mail.sysu.edu.cn
†Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR, hndai@ieee.org
‡Peng Cheng Laboratory, Shenzhen, China

Abstract*—With the proliferation of Virtual Reality (VR) markets, VR applications are rapidly expanding in scale and complexity, thereby driving an urgent need for assuring VR software quality. Different from traditional mobile applications and computer software, VR testing faces unique challenges due to diverse interactions with virtual objects, complex 3D virtual environments, and intricate sequences to complete tasks. All of these emerging challenges hinder existing VR testing tools from effectively and systematically testing VR applications. In this paper, we present *VRExplorer*, a novel model-based testing tool to effectively interact with diverse virtual objects and explore complex VR scenes. Particularly, we design the Entity, Action, and Task (EAT) framework for modeling diverse VR interactions in a generic way. Built upon the EAT framework, we then present the *VRExplorer* agent, which can achieve effective scene exploration by incorporating meticulously designed path-finding algorithms into Unity's NavMesh. Moreover, the *VRExplorer* agent can also systematically execute interaction decisions on top of the Probabilistic Finite State Machine (PFSM). Experimental evaluation on 11 representative VR projects shows that *VRExplorer* consistently outperforms the state-of-the-art (SOTA) approach *VRGuide* by achieving significantly higher coverage and better efficiency. Specifically, *VRExplorer* yields up to 122.8% and 52.8% improvements over *VRGuide* in terms of executable lines of code (ELOC) coverage and method (function) coverage, respectively. Furthermore, ablation results also verify the essential contributions of each designed module. More importantly, our *VRExplorer* has successfully detected two *functional* bugs and one *non-functional* bug from real-world projects.

***Index Terms*—Software Testing, Virtual Reality, Model-based Testing, Scene Exploration**

I. Introduction

Virtual reality (VR), working together with other relevant technologies, such as Augmented Reality (AR) and Extended Reality (XR), aims to provide users with an immersive mixed reality (MR) experience [1]. Current VR applications have proliferated in diverse fields, such as medical treatment, education, audiovisual entertainment, training simulations, manufacturing, and gaming [1–6]. According to Fortune's market report [7], the global VR market is projected to grow from $44.4 billion in 2025 to $244.84 billion in 2032.

With the proliferation of VR markets, VR applications are also rapidly expanding in terms of scale and complexity. Ensuring the software quality of such complex VR applications becomes an urgent need. As a critical procedure of software development, testing can thoroughly evaluate a software to check whether both requirements and functional needs are fulfilled without defects. To cater to this growing demand in VR applications, extensive efforts have been made in VR testing, while many of them still largely rely on manual testing, which remains highly labor-intensive and inefficient [8, 9]. Most recently, several studies have aimed to test VR applications automatically. As an early attempt, *VRTest* [10] explores VR scenes by controlling the orientation of the camera to interact with virtual objects. *VRGuide* [11] has further improved *VRTest* by optimizing exploration routes to circumvent occluded objects. Besides *VRTest* and *VRGuide*, other researchers also explore using other techniques, such as computer vision (CV) and generative artificial intelligence (GenAI) to achieve scene exploration [12] or understanding [13]. However, these testing tools are still struggling to test VR applications with increased complexity.

The challenges in VR testing stem from the *unique features* of VR applications, sharply different from conventional mobile applications and computer software. First, enabled by diverse peripheral devices (e.g., VR headsets, controllers, joysticks, wands, and haptic gloves), VR systems can support a larger *diversity of interactions* (such as grabbing, pressing, touching, pulling, climbing, and shooting) than conventional mobile systems and PCs [14, 15]. Unfortunately, current VR testing tools cannot properly characterize and represent the diverse interaction behaviors. For example, the state-of-the-art (SOTA) VR testing tool, *VRGuide*, can only test virtual objects with the "*click*" interaction. Second, VR applications contain *complex 3D virtual environments* with interactions with virtual objects, thereby introducing a vast exploration state space. Take *EscapeGameVR*, a popular open-source VR gaming application, as an example, which contains 44 scenes, 8,256 GameObjects, and 1,377 C# script files. Third, VR testing often requires completing a sequence of tasks, e.g., finding a key, then using the key to open a door, next turning a handle, and finally pressing a button to escape. It is non-trivial to accomplish these complex tasks since they involve diverse interactions and trigger either events or actions in a specific order. In summary, the diverse and intricate nature of the VR interactions, coupled with large-scale virtual spaces and complex task-completion sequences, makes it difficult to comprehensively and efficiently test VR applications in a systematic and repeatable manner.

Our Approach. To address the above challenges, we present *VRExplorer* to thoroughly test Unity-based VR applications by conducting in-depth scene exploration and comprehensive interactions with virtual objects. Notably, we consider Unity-based VR applications mainly due to Unity's dominant role in VR/MR markets [16]. To tackle the first challenge mentioned above, we design the Entity, Action, and Task (EAT) framework for modeling VR interaction behaviors in a generic way. This hierarchical framework enables a reusable modeling process across VR applications developed by diverse Unity versions and interaction plugins, such as XRIT [17, 18], STEAMVR [19–21], and MRTK [22, 23], thereby enhancing cross-project *generalizability*. To address the second challenge, we present the *VRExplorer* agent built upon the EAT framework. Incorporating meticulously designed path-finding algorithms into Unity's NavMesh, the *VRExplorer* agent can achieve autonomous navigation in 3D virtual environments. Moreover, the *VRExplorer* agent can also systematically execute diverse interaction decisions on top of the Probabilistic Finite State Machine (PFSM). To address the third challenge, the proposed model-based approach transforms intricate VR task execution sequences into structured task models to enable systematic testing and automated execution.

Evaluation. To comprehensively evaluate the proposed *VRExplorer*, we conduct extensive experiments on 11 representative VR projects. The experimental results demonstrate that the proposed *VRExplorer* outperforms the SOTA approach *VRGuide* with average performance gains of 122.8% and 52.8% in terms of executable lines of code (ELOC) coverage and method (function) coverage, respectively. Moreover, the ablation study on the five ablated variants of *VRExplorer* also validates the necessity of all modules of the EAT framework. More importantly, *VRExplorer* has successfully detected two previously unknown real-world bugs (i.e., one *functional* bug and one *non-functional* bug), which can nevertheless be detected by the baseline. Further, *VRExplorer* has also successfully detected one previously-confirmed bug.

Main Contributions are summarized as follows:

- We design the EAT framework for modeling complex interaction behaviors and tasks in VR applications. To the best of our knowledge, EAT is *the first* generic three-layer abstraction framework for VR testing based on the object-oriented programming (OOP) paradigm.
- We present *VRExplorer*[1], a novel model-based testing tool to achieve effective interactions with diverse virtual objects and the exploration of complex VR scenes.
- We evaluate *VRExplorer* extensively on 11 representative VR projects and demonstrate its consistent performance superior to the SOTA approach in terms of ELOC coverage, method coverage, and interactable object coverage while preserving high efficiency. Moreover, our *VRExplorer* can also successfully detect real-world bugs in VR projects.

[1]https://github.com/TsingPig/VRExplorer_Release

II. BACKGROUND

Interactions in Unity-based VR Applications. A Unity-based VR application typically consists of multiple *Scenes*, each of which is structurally represented by a *hierarchy* of *GameObjects*. These GameObjects represent all visible and interactive elements in the 3D virtual environment and can be composed of various components, such as meshes, scripts, and colliders. Unity provides VR application developers with a rich set of tools for implementing immersive interactions. As Unity's official framework, XRIT [17] can support common VR input modalities such as ray-based selection, direct grabbing, teleportation, and gesture recognition. These interactions are typically implemented using component-based scripts attached to objects, while often relying on trigger colliders or physics events. However, the implementation logic varies significantly across projects, especially when third-party packages are used, thereby increasing the complexity of generalizing automated testing across different VR projects.

Mono Scripts in Unity-based Application Development. In Unity-based VR application development, *Mono* scripts constitute the fundamental building blocks for implementing interactive behaviors. These C# classes inherit properties from Unity's core `MonoBehaviour` [24] base class, enabling them to leverage Unity's component-based architecture. Through the Unity Inspector, developers attach these scripts to GameObjects to create diverse objects. For example, a gun can be attached with a `XRGun` class inherited from `MonoBehaviour`, with properties referencing the bullet *prefab*[2], `Shoot()`, etc.

NavMesh-Based Navigation in Unity. Unity's NavMesh system [25–27] provides a mechanism to traverse 3D environments using navigation meshes generated from static scene geometry. It supports obstacle avoidance, pathfinding, and dynamic updates, making it suitable for simulating players' movement. In VR testing, NavMesh can be leveraged to automate scene exploration by guiding a virtual agent through reachable areas. However, it only supports locomotion and requires external control logic to handle task-specific actions, such as interacting with objects or triggering events.

III. APPROACH

As shown in Fig. 1, the proposed *VRExplorer* works by first collecting and analyzing open-source VR projects in § III-A. Then, it implements the EAT framework in § III-C after model extraction in § III-B. Next, it conducts testing by *VRExplorer* in § III-D based on the EAT framework, NavMesh, and PFSM.

A. Project Collection and Analysis

To comprehensively investigate interaction behaviors in VR, we collect open-source Unity VR projects with diverse applications, interactions, and scene types (details in § V-A).

Based on this dataset, we perform a *two-pass semi-automated analysis* — static pass and dynamic pass—and then consolidate the results to construct Model Abstraction, including Object and Action abstractions (§ III-B). These

[2]Prefab is a reusable template encapsulating GameObjects and components.

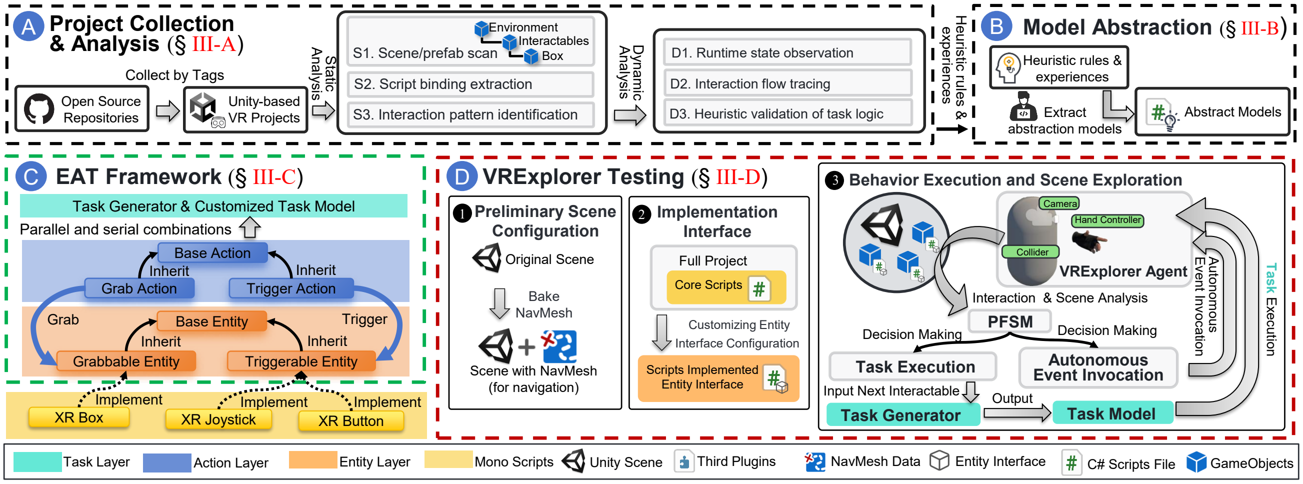


Fig. 1: Overview of *VRExplorer*

abstractions are subsequently fed into the EAT framework (§ III-C) and the PFSM-based testing workflow (§ III-D).

Static pass (S1–S3). *S1: Scene/prefab scan.* We parse scene hierarchies and prefabs to enumerate GameObjects with attached components, and prune *non-interactable* infrastructure (e.g., static walls) that are not connected to core interactions. *S2: Script binding extraction.* We inspect `MonoBehaviour`-based scripts and their serialized fields to recover object–script bindings (e.g., references to targets). *S3: Interaction pattern identification.* We analyze interaction-related APIs and callback signatures to fingerprint typical behaviors. The interaction distribution will be detailed in § IV-A.

Dynamic pass (D1–D3). Static inspection cannot observe runtime object creation, dynamic component attachment, or goal-oriented task logic. To complement this effect, we conduct a lightweight but human-guided runtime analysis guided by gameplay flows and task objectives: *D1: Runtime state observation.* At each frame t, we record snapshots of active GameObjects $\mathcal{G}_t$ and their component sets $\mathcal{C}_t$. Comparing consecutive states ($\Delta\mathcal{G}_t$, $\Delta\mathcal{C}_t$), we identify dynamically instantiated objects and runtime-added components. *D2: Interaction flow tracing.* Following execution evidence such as console outputs, function call stacks, and Unity event dispatches during gameplay, we trace when and how scripts are actually triggered (e.g., input events, collision handlers, and task-specific event chains). *D3: Heuristic validation of task logic.* Four experienced engineers heuristically reconstruct event sequences observed during gameplay to approximate these flows and validate whether each interaction contributes essentially to task progression or it is only incidental.

B. Model Abstraction

Heuristic Model Abstraction. We summarize the heuristic experiences and rules extracted from the project analysis in order to test the framework's effectiveness and get a preliminary model abstraction. Specifically, during the project analysis process, we merge static and dynamic findings into abstraction models heuristically and classify common VR interaction behaviors into abstract actions based on OOP principles, as shown in Part B in Fig. 1. When testing a real-world project, engineers can absolutely use the model we had constructed, while also experiencing core gameplay and performing the two-pass manual analysis as described in § III-A if they have customized testing demands. New interaction patterns are either matched with existing types or newly defined via new Action interfaces automatically. While execution is fully automated, abstraction still requires human-in-the-loop analysis due to the complexity of VR interactions and the limited domain-specific knowledge of existing tools. Objects are annotated with interactable features (e.g., *Grabbable*, *Triggerable*, and *Transformable*); interactions are lifted into abstract actions. This consolidated model (i) seeds the *Entity Interface Layer* in EAT by proposing interface candidates (e.g., `IGrabbable` and `ITriggerable`), (ii) provides action-level semantics to the *Action Class Layer*, and (iii) surfaces callable functions and UnityEvents to populate the task (§ III-D).

TABLE I: Example of Generalizing VR Interactable Objects

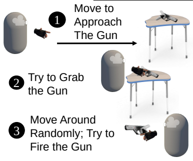
VR Interactable Objects	Interactable Features
Box, Coin, etc.	*Grabbable*
Button	*Triggerable*
Joystick	*Transformable*
Gun, Cigarette Lighter, etc.	*Grabbable* and *Triggerable*

Abstraction of Interactable Objects. We first generalize VR interactable objects into abstract objects containing interactable features. Table I gives an example of VR interactable objects with interactable features, which can determine interaction properties. For instance, objects like boxes and coins can be typically classified as *Grabbable*, meaning that they can be picked up, while objects like buttons and joysticks are considered *Triggerable* with two attributes *triggering* and *triggered*. Notably, some objects may possess multiple interactable features. For instance, a gun or a cigarette lighter possesses both *Grabbable* and *Triggerable* attributes, representing complex interactions to support not only grabbing but also acting on its function, such as shooting or turning on a button.

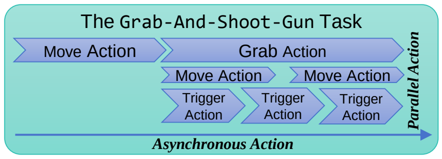
Abstraction of Actions. We then classify VR interaction behaviors into abstract actions. Table II shows an example of a VR interactive scene, in which pressing a button, closing or

TABLE II: Generalizing Interactions into Abstract Actions

VR Interaction Behaviors	**Abstract Actions**
Move Around, Jump, Fly, etc.	*Move Action*
Press, Pull, Open/Close, Turn On/Off, etc.	*Trigger Action*
Grab, Throw	*Grab Action*
Move/Scale/Rotate Objects	*Transform Action*



(a) GASG Process



(b) Action Model of GASG task

```
private List<BaseAction> GrabAndShootGunTask(
    IGrabbableEntity grabbableEntity, ITriggerableEntity triggerableEntity)
{
    List<BaseAction> task = new List<BaseAction>(){
        new MoveAction(_navMeshAgent, moveSpeed,
            grabbableEntity.transform.position),
        new GrabAction(leftHandController, grabbableEntity,
            new List<BaseAction>(){
                new ParallelAction(new List<BaseAction>(){
                    new MoveAction(_navMeshAgent, moveSpeed,
                        GetRandomTwitchTarget(transform.position)),
                    new TriggerAction(2.5f, triggerableEntity)
                })
            })
    };
    return task;
}
```

(c) C# Code Snippet of GASG task

Fig. 2: Task Instance of `Grab-And-Shoot-Gun`

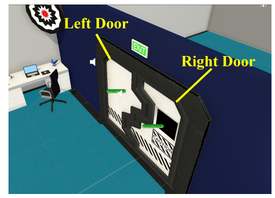
opening a door, turning on or off a lamp, and pulling a joystick, can be conceptualized as compositions of a continuous process with a following event. Specifically, the player's action, pulling a joystick, can be classified into a continuous pulling process and an event triggered after pulling is over. We classify a behavior with such characteristics into *Trigger Action*.

C. EAT Framework

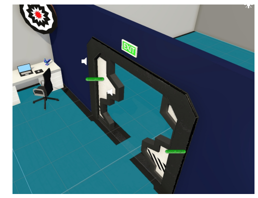
Based on the model abstraction in § III-B, we propose a three-layer framework called EAT, as shown in Part C in Fig. 1.

Entity Interface Layer. Based on the interactable objects' abstraction, we define specific interfaces to encapsulate their corresponding attributes. Particularly, we define a base interface, called `BaseEntity`, which includes the `Transform` property to represent the object's position, rotation, and scale, as well as the `Name` property. Thereafter, all other entity interfaces inherit this base interface. For example, an interface `Triggerable` includes two attributes *triggering* and *triggered* with the `Enum` property. The Entity Interface layer provides customized interfaces tailored to specific features required by the Action layer. Notably, complex interactable features can be efficiently represented through multi-inheritance by multiple interfaces, thereby allowing for higher flexibility and modularity in defining VR interactions.

Action Class Layer. We then extract behaviors with identical characteristics into the same action. We first define an abstract base class, namely `BaseAction` with a *virtual* asynchronous `Execute()` method. All other action classes inherit this base class. The Action layer can interact with the Entity Interface Layer and provide functional APIs for *VRExplorer* to simulate real players' interactive actions. For example, *Trigger Action* can be concreted into the `TriggerAction` class, which



(a) Doors Closed (NavMesh separated)



(b) Doors Opened (NavMesh connected)

Fig. 3: Dynamic doors with NavMesh Obstacle (Carve enabled)

consists of an asynchronous method `Triggering()` and a synchronous method `Triggered()`.

Task Model Layer. A composition task consists of multiple *parallel* and *sequential* actions. Parallel action executes asynchronous methods simultaneously, while sequential actions follow a strict order to complete the previous action before proceeding to the next. The Task Model layer includes a task generator and multiple predefined VR interaction tasks. The task generator creates tasks by accepting either a `MonoBehaviour` instance or an array of entities (`BaseEntity[]`). We derive the predefined tasks from commonly used task models. Fig. 2 depicts an example of the `Grab-And-Shoot-Gun` (GASG) task, which consists of three steps: approaching the table, picking up the gun, and shooting while walking in Fig. 2(a). Fig. 2(b) elaborates on the corresponding action model, where the horizontal axis represents the timeline of asynchronous actions starting with *Move Action*, followed by *Parallel Action*, within which three actions execute concurrently. The corresponding code is also shown in Fig. 2(c).

D. VRExplorer Testing

As shown in Part D in Fig. 1, *VRExplorer*'s testing process works in three sequential steps:

❶ *Preliminary Scene Configuration*, before scene exploration, we facilitate the agent's navigation capability based on Unity's NavMeshAgent [26] via NavMesh baking. We first configure terrain objects, such as floors and walls, to be static. Regarding those objects that may dynamically move, such as doors, we attach the NavMesh Obstacle [27] component with the enabled Carve option, thereby allowing them to dynamically modify the NavMesh. Fig. 3 depicts an example of opening and closing *dynamic* doors.

❷ *Implementation Interface*, on top of the EAT framework, implementing *interactable* interfaces is the core to tackle two challenges: (1) lack of generalizability caused by diverse Unity versions and the fragmentation of the VR development ecosystem; and (2) intricate scene exploration and VR interactions. Using interfaces to encapsulate implementation details can enable testing for a diversity of VR applications.

To ensure coverage and verify the reliability of interaction methods, test engineers need to select and customize the appropriate Entity layer interfaces for the Mono scripts related to the core VR interaction logic in the target project with the corresponding interface functions implemented. Take Procedure 1 as an example[3], in which a gun class `XRGun` possessing

[3]The implementation of the entity interface's code lines is marked blue.

Procedure 1 Entity Interface Layer Implementation Example.

```
class XRGun implements ITriggerable, IGrabbable:
    property TriggeringTime ← 1.5
    property Name ← "Gun"
    property Grabbable:
        if component exists then return it
        else add new Grabbable component
    method Triggerring() do nothing
    method Triggerred() calls Fire()
    method OnGrabbed() do nothing
    field projectilePrefab: GameObject
    field startPoint: Transform
    field launchSpeed: float
    method Fire():
        Instantiate projectile and apply forces
    method ApplyForce(rigidbody):
        Calculate and apply physics forces
```

both *Grabbable* and *Triggerable* features, can be implemented by inheriting `MonoBehaviour` while simultaneously implementing the `IGrabbable` and `ITriggerable` interfaces. This design allows the gun object to be both *grabbed* and *triggered* during VR interactions without additional script components. This approach allows `XRGun` to seamlessly integrate multiple interaction capabilities (i.e., *grabbable* and *triggerable*). Leveraging interface composition, we can flexibly define and extend object behaviors without modifying the base class hierarchy, thereby promoting code reusability and achieving modular design in the VR interaction system.

❸ *Scene Exploration with Behavior Execution.* after completing the scene configuration and implementing interactable interfaces, *VRExplorer* automatically performs scene exploration and VR interactions by executing the corresponding behaviors. We design two types of behaviors: *task execution* and *autonomous event invocation*. When executing tasks, *VRExplorer* perceives the scene and inputs `MonoBehaviour` instances or an array of entities (`BaseEntity[]`) into the task generator to obtain and execute the corresponding tasks. Notably, players in VR environments not only interact with objects but also invoke autonomous events, such as casting skills to gain acceleration buffs.

PFSM. To comprehensively cover such testing cases, *VRExplorer* maintains a configurable list of `UnityEvents`, allowing test engineers to easily customize the functions to be covered by configuring them in the Inspector before testing. These two types of behaviors constitute the behavior space, where each behavior can be regarded as a node. The behavior space itself forms a directed graph composed of these nodes. For behavior decision-making, *VRExplorer* employs PFSM to determine transitions between nodes, thereby defining the topology of the

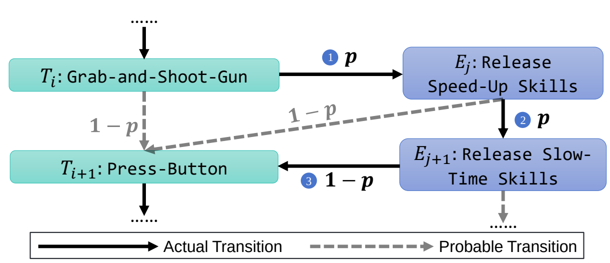


Fig. 4: Example of PFSM State Transition in a VR Scene.

TABLE III: State Transition Table of PFSM

Current State	Next State	Condition	Probability
T_i	T_{i+1}	Default	$1-p$
T_i	E_j	Default	p
E_j	E_{j+1}	Default	p
E_j	T_{i+1}	Default	$1-p$
T_i	T_{i+1}	$\mathcal{E}$ exhausted	1.0
E_j	E_{j+1}	$\mathcal{T}$ exhausted	1.0

directed graph. We use $\mathcal{T} = \{T_1, T_2, \ldots, T_N\}$ to denote the set of all task states, where T_i represents the i-th task state in the execution sequence, with $i \in \{1, 2, \ldots, N\}$. Similarly, we use $\mathcal{E} = \{E_1, E_2, \ldots, E_M\}$ to denote the set of exploration states, where E_j represents the j-th exploration state in the execution sequence, with $j \in \{1, 2, \ldots, M\}$. At each decision point in PFSM, a variable determines the transition direction. The probability of transitioning to an exploration state (E_j) is denoted by p while the probability of transitioning to a task state (T_i) is $(1-p)$. Fig. 4 depicts an example of state transition in a VR scene, where `Speed-Up Skills` node and `Slow-Time Skills` node are two example nodes of an autonomous event adding buffs to the player, while GASG node and Press-Button (PB) node are two example nodes of the task. Those four nodes are independent of each other, while the PFSM is responsible for decision-making and state transitions among them. Table III lists all the state transitions.

Path-finding for Navigation. We implement two algorithms: (i) a Greedy algorithm, which follows a local optimization strategy based on the shortest path principle, and (ii) a Backtracking algorithm with Pruning, which searches for globally optimal solutions. Since the Greedy algorithm significantly reduces time complexity, fulfilling the real-time requirements of VR testing, we mainly adopt it in our experiments (§ VI presenting a comparison of the two algorithms).

In summary, *VRExplorer* continuously obtains scene information, decides the current behavior to execute, and receives feedback after performing the behavior. This process is repeated until all interactable objects and events are covered.

IV. Evaluation of *VRExplorer*

A. Implementation

Project Collection and Analysis. We implemented a crawler to collect Unity-based VR GitHub projects with keywords like "VR", "AR", "unity", and "xr". From 971 initially filtered projects, we manually performed quality checks (e.g., compilation errors and version conflicts), consequently retaining 102 high-quality projects. We then analyze the structural and interaction characteristics of all projects in the dataset to assess their representativeness and complexity. Table IV shows varied project sizes. While most projects are relatively small (medians containing 71.5 scripts, 6,129 LOC, 620 files, and 8 scenes), a few of them with large sizes markedly raise the

TABLE IV: Statistical Metrics of Collected 102 Projects.

Metric	Mean	Variance	Min	Q1	Median	Q3	Max
Scripts	166.45	110,953.52	2	26.25	71.50	130.25	2,004
LOC	19,910.85	1,851,165,810.13	256	2,707.25	6,129.00	13,762.75	237,725
Files	902.07	902,718.22	21	301.25	620.50	1,090.00	5,303
Scenes	18.12	455.06	1	4.00	8.00	30.25	128

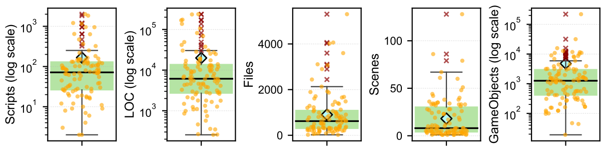


Fig. 5: Boxplots of scripts, LOC, files, and scenes.

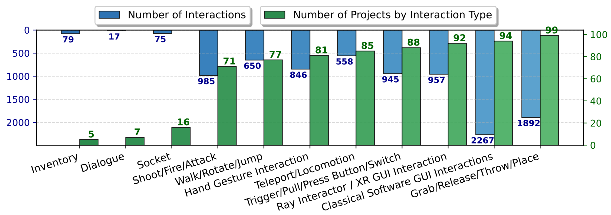


Fig. 6: Distribution of Interactions in Dataset.

medians. Fig. 5 plots their distribution. Fig. 6 indicates that the dataset includes both prevalent interactions (e.g., grabbing, GUI operations, and teleportation) and infrequent ones (e.g., inventory and dialogue). Together, these interactions cover all major interaction tasks identified in [28], thereby highlighting the constructed dataset's representativeness and diversity.

We then perform the two-pass analysis (§ III-A), summarizing heuristic rules for subsequent modeling.

Implementation of Model Abstraction. Based on the above analysis, we extract abstract models using the derived heuristics. The implementation comprises 6 scripts in the Action layer, 4 scripts in the Entity layer, and 4 pre-defined Mono scripts. Additionally, 5 pre-defined task models orchestrate these components, resulting in more than 1,600 lines of code in total. Manual exploration per project ranged from 30 to 90 minutes, depending on the engineer's proficiency.

Implementation of the EAT Framework. We derive `Grabbable`, `Transformable`, and `Triggerable` interfaces from the base interface `BaseEntity` in the Entity layer. In the Action layer, we define `GrabAction`, `MoveAction`, `ParallelAction`, `TransformAction`, and `TriggerAction` as the subclasses of the abstract base class `BaseAction`. Each subclass overrides the `Execute()` method to implement its corresponding behavior. Thereafter, we provide the implementations of common task models as the code-level compositions of actions, e.g., the PB and GASG tasks.

For each project, we analyze the interactable objects in scenes and all the C# scripts referenced by the objects, and select the codes corresponding to the core VR interaction functionality, referred to as *Core Code*. Tool libraries, third-party code, and other non-interaction scripts are excluded. This selection serves two purposes: (i) evaluation of testing performance via Assembly-Definition files, and (ii) flexible customization and configuration for different VR projects. Core Code is a *heuristic criterion* useful for any type of VR applications (even Unity-based or non-Unity-based).

We then modify classes in Core Code to implement the interfaces of the Entity Interface layer. To minimize additional workload for developers, we provide a set of predefined scripts inherited from `MonoBehaviour` for projects developed using

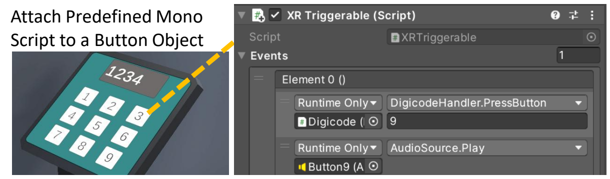


Fig. 7: Predefined Mono Script Component `XRTriggerable`

the XRIT framework. These commonly used Mono scripts already implement the corresponding interfaces. Developers can simply attach the predefined scripts to target objects and configure the methods under test via `UnityEvent` [29] in the Unity Inspector (see Fig. 7). This design allows easy extension to diverse interactions across VR applications while introducing minimal extra effort for developers, and ensures that the Core Code is fully testable and easily configurable.

Implementation of *VRExplorer*. We equip the *VRExplorer*'s GameObject with a NavMeshAgent component for navigation on NavMesh with the attached controller component (like a player's hands) to enable interactions with objects. To support scene perception, we implement the `EntityManager` class, which is responsible for registering scene information and providing APIs for scene analysis within *VRExplorer*. Then, we implement PFSM to support conditional branching for decision-making, where PFSM's inputs are derived from scene analysis. Next, we implement a `TaskGenerator` component to support task execution by translating relevant inputs into the corresponding task model.

Environment. We implement and evaluate *VRExplorer* with the comparison of other baselines (in § IV-C) on a computer with AMD Ryzen 7 5800H with Radeon Graphics CPU, 32GB RAM, and NVIDIA GeForce RTX 3060 GPU (6GB) Graphics card. This computer is installed with Windows 11 and the same Unity version as that used in all evaluated projects.

System Configuration Parameters. Since the proposed *VRExplorer* simulates a player to explore the VR scene, the *move speed* (MS) and the *turn speed* (TS) (also called *rotation speed* in [11]) greatly affect the testing performance. Regarding the move speed, excessive speed can introduce physical and interactive issues (e.g., test tools may unintentionally move out of a platform due to inertia), although a faster speed may generally improve performance. Therefore, to balance efficiency with practicality, we choose MS = 6 m/s and TS = 60 deg/s as the standard parameters for *VRGuide* and *VRExplorer* in all subsequent experiments. § VI discusses the rationale for these settings. Since these parameters are chosen to be the same for all test tools, they are not repeated in detail in the subsequent experimental results. An experiment is terminated when a test tool reaches convergence. In the PFSM model, the probability of transitioning to the next state is decided by parameter p, which is set to 0.5, indicating the same propensity for both state transitions.

B. Metrics

With reference to [11, 30–32], we consider ELOC coverage, method coverage, interactable objects coverage, and convergence time as evaluation metrics:

TABLE V: Quantitative Metrics of Selected VR Projects

	Projects	# of Scripts	LOC	# of Files	Scenes	# of GOs	Version
Group 1	unity-vr-maze	158	25,261	212	1	278	5.x
	UnityVR	150	24,858	330	3	124	2019.x
	UnityCityView	182	28,335	446	34	1,194	2019.x
Group 2	Parkinson-VR[1]	275	38,437	968	33	1,566	2019.x
	VGuns	81	10,900	848	36	1,653	2020.x
	EE-Room[2]	88	4,450	1,063	8	1,517	2020.x
	EscapeGameVR	91	6,659	1,377	44	8,256	2021.x
	VRChess	160	26,591	414	4	280	2021.x
	VR-Basics	62	2,677	724	5	2,143	2021.x
	VR-Room	65	3,660	679	2	414	2022.x
	VR-Adventure	11	260	91	2	288	2022.x

[1] Parkinson-VR stands for Parkinson-App-Virtual-Reality.
[2] EE-Room stands for Edutainment-Escape-Room.

- **ELOC Coverage (EC).** Since ELOC mainly focuses on code lines containing executable programs, EC measures the percentage of these executable lines with the exclusion of blank lines, comments, and declaration statements during testing. We adopt *Code Coverage* [33], an official tool provided by Unity, to automatically record the EC of C# scripts. Running in Unity Editor Mode, the testing tool exports the coverage data as a historical report in `.html` and a summary in `.xml`.
- **Method Coverage (MC).** Besides EC, we also consider MC to evaluate performance, quantifying the percentage of testing methods (functions) that have been invoked during testing.
- **Interactable Object Coverage (IOC).** We adopt IOC to fairly compare *VRExplorer* with SOTA baseline *VRGuide* [11]. Slightly different from [11], in which interactable objects only include objects that can receive mouse "click" events, we further extend interactable objects to all objects that support user interaction in the VR environment. These interactable objects are identified by analyzing and confirming the `MonoBehaviour` scripts associated with the scene objects.
- **Convergence Time.** We also evaluate the efficiency of the proposed approach. Particularly, we consider *convergence time* to measure how fast a tool can reach the *converged state*, which is defined as the status at which the testing tool no longer seeks additional scene exploration. The convergence time refers to the amount of time taken for the testing tool to reach the converged state from the initial state.

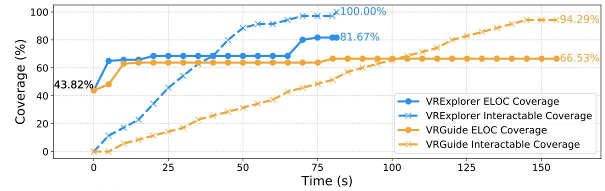
C. Baselines

Given the unique characteristics of VR applications, such as complex VR interactions and the fragmented nature of the VR development ecosystem, current automated testing tools and Android application testing frameworks are not suitable for VR application testing. To the best of our knowledge, *VRGuide* [11] is the SOTA testing approach for VR applications. Besides *VRGuide*, *VRTest* [10] is a previous version, which nevertheless has inferior performance to *VRGuide* in terms of both MC and IOC (only pointer click events received).

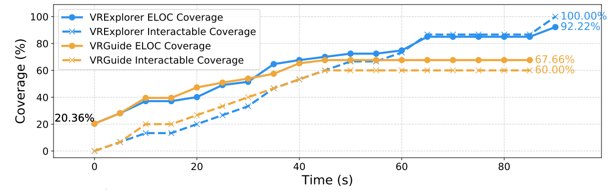
V. Experimental Results

We conduct extensive experiments to evaluate *VRExplorer* and aim to investigate three research questions (RQs):

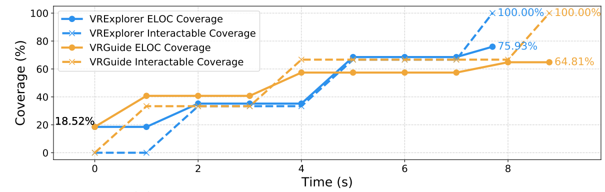
- **RQ1:** How does *VRExplorer* perform with comparison of the existing SOTA approach in diverse VR projects?
- **RQ2:** How do different modules contribute to the performance of *VRExplorer*?
- **RQ3:** Can *VRExplorer* detect real-world VR bugs?



(a) Coverage versus Time in Project `unity-vr-maze`



(b) Coverage versus Time in Project `UnityCityView`



(c) Coverage versus Time in Project `UnityVR`

Fig. 8: EC versus time during the testing process in Group 1.

A. VRExplorer Performance

Constructing VR Projects for Evaluation. To evaluate the proposed *VRExplorer*, we construct a project dataset[4], consisting of 11 representative VR projects, as summarized in Table V. This project dataset can be divided into two groups: (1) Group 1, in which we select the most complex three projects also being included in *VRGuide* [11], and (2) Group 2, into which we introduce the other eight Unity-based VR projects developed by more recent versions (e.g., after 2020.x). It is worth noting that we construct Group 1 (with relatively older Unity versions) primarily for a fair comparison with *VRGuide*, as it only supports the "click" interaction. Compared with Group 1, Group 2 contains VR projects developed by recent Unity versions, which can support more diverse interactions (while not being supported in *VRGuide*).

The 11 projects selected for experiments represent a diverse and comprehensive subset of VR applications. Firstly, the chosen projects cover all of the most commonly featured VR application types as defined in [34]: (i) Action and Shooter (`VGuns`), (ii) Simulation (`VR-Basisc`, `VR-Room`, `UnityVR`), (iii) Adventure (`unity-vr-maze`, `VR-Adventure`), (iv) Puzzle (`Edutainment-Escape-Room`, `EscapeGameVR`), (v) Medical Care (`Parkinson-VR`), and (vi) Strategy Board Game (`VRChess`). This diversity ensures that the experimental results are generalizable across different VR genres and interaction manners. Second, these projects cover a wide range of Unity versions, from older releases like 2019.4.2f1 (`UnityCityView`) to more recent versions such as 2022.3.7f1 (`VR-Adventure`), as well as the legacy version 5.4.1f1 (developed for `unity-vr-maze`). This version diversity ensures that our testing approach is evaluated across different Unity engine environments, demonstrating its robustness and compatibility.

[4]List of constructed project dataset can be found at https://github.com/TsingPig/VRExplorer/blob/main/Artifacts/Evaluated_Repo_Url.md

TABLE VI: Results on Projects in Group 1

Projects	Approaches	EC (%)	MC (%)	IOC(%)	Convergence Time Cost (s)	# of Interactable Objects
`unity-vr-maze`	*VRGuide*	66.53	70.59	94.29	145.0	35
	VRExplorer	81.67 (+22.8%)	82.35 (+16.7%)	100.00 (+6.1%)	81.4 (-43.9%)	
`UnityCityView`	*VRGuide*	67.66	78.38	60.00	45.0	15
	VRExplorer	92.22 (+36.3%)	100.00 (+27.6%)	100.00 (+66.7%)	89.3 (+98.4%)	
`UnityVR`	*VRGuide*	64.81	84.62	100.00	8.8	3
	VRExplorer	75.93 (+17.1%)	92.31 (+9.1%)	100.00	7.7 (-12.5%)	

Moreover, the selected projects exhibit a variety of scales and complexities. For instance, the number of C# scripts ranges from as few as 11 in `VR-Adventure` to over 275 in `Parkinson-VR`, with LOC spanning from around 260 to over 38,437. This selection allows us to assess the testing tools' versatility from small and medium-sized VR applications to large ones. These projects also differ in the number of scenes and the number of GameObjects (GOs), with some projects like `EscapeGameVR` having 44 scenes, reflecting complex and rich environments, while others like `VR-Room` have fewer scenes but have potentially dense and complicated interactions.

Results on Group 1. Fig. 8 plots EC versus time when testing all the projects in Group 1 where initial EC is identical across projects as initialization code (e.g., `Awake()`) runs immediately, regardless of interactions. Table VI shows the experimental results of *VRExplorer* compared with baseline *VRGuide* in Group 1. To quantitatively evaluate the performance improvement of *VRExplorer* over *VRGuide* (or other compared methods in § V-B), we define the *performance gain* of method A over method B in metric M as follows,

$$G_{AB}(M) = \frac{P_A(M) - P_B(M)}{P_B(M)} \times 100\%, \quad (1)$$

where $P_A(M)$ and $P_B(M)$ denote the performance of methods A and B with respect to metric M, respectively. For example, we evaluate the performance gain of *VRExplorer* (E) over *VRGuide* (G) in project `unity-vr-maze` in terms of EC as $G_{EG}(\mathrm{EC}) = \frac{P_E(\mathrm{EC})-P_G(\mathrm{EC})}{P_G(\mathrm{EC})} \times 100\% = \frac{81.67-66.53}{66.53} \times 100\% = 22.8\%$. Similarly, $G_{EG}(\mathrm{MC})$ and $G_{EG}(\mathrm{IOC})$ are 16.7% and 6.1%, respectively, while the convergence time cost is reduced by 43.9%. For project `UnityCityView`, we have $G_{EG}(\mathrm{EC}) = 36.3\%$, $G_{EG}(\mathrm{MC}) = 27.6\%$, and $G_{EG}(\mathrm{IOC}) = 66.7\%$. For project `UnityVR`, we have $G_{EG}(\mathrm{EC}) = 17.1\%$, $G_{EG}(\mathrm{MC}) = 9.1\%$, and a reduced convergence time cost of 12.5%. Notably, *VRExplorer* reaches 100% IOC in all three projects, while *VRGuide* achieves 100% IOC only in `UnityVR`. In summary, these results demonstrate that *VRExplorer* consistently outperforms *VRGuide* in coverage and efficiency.

Results on Projects of Group 2. Table VII reports the results of *VRExplorer* compared with *VRGuide* on the eight representative projects in Group 2 in EC and MC. Notably, we omit the IOC here mainly because *VRGuide* exhibited lower IOC than our *VRExplorer* due to its sole "clicking" interaction. Therefore, we primarily focus on the code coverage (ELOC and MC) of the core development logic of VR projects.

We observe that our *VRExplorer* consistently outperforms *VRGuide* in all eight projects. Notably, *VRExplorer* achieves $G_{EG}(\mathrm{EC}) = 567.97\%$ in `VRChess` significantly higher than *VRGuide*. This is because this project contains numerous *if-else* branches, difficult for *VRGuide* to fully cover. In contrast, *VRExplorer* can effectively explore most of these branches.

TABLE VII: Results on Projects of Group 2

Projects	Approaches	EC (%)	MC (%)
`VR-Basics`	*VRGuide*	41.38	53.22
	VRExplorer	80.17 (+93.8%)	91.93 (+72.8%)
`VR-Room`	*VRGuide*	40.97	50.63
	VRExplorer	77.61 (+89.4%)	83.54 (+65.0%)
`VGuns`	*VRGuide*	28.68	38.89
	VRExplorer	77.57 (+170.7%)	77.78 (+100.0%)
`VR-Adventure`	*VRGuide*	54.12	65.00
	VRExplorer	91.76 (+69.6%)	95.00 (+46.2%)
`EE-Room`	*VRGuide*	38.08	58.06
	VRExplorer	70.61 (+85.5%)	88.17 (+51.8%)
`EscapeGameVR`	*VRGuide*	41.77	55.26
	VRExplorer	71.08 (+70.2%)	73.68 (+33.3%)
`Parkinson-VR`	*VRGuide*	42.03	53.85
	VRExplorer	95.65 (+127.6%)	100.00 (+85.7%)
`VRChess`	*VRGuide*	10.74	50.88
	VRExplorer	71.74 (**+568.0%**)	87.72 (+72.4%)

Comprehensively considering experimental results in all 11 projects in both Group 1 and Group 2, *VRExplorer* has achieved an average EC gain of 122.8% and average MC gain of 52.8% compared to *VRGuide*. Experimental results demonstrate that *VRExplorer* outperforms *VRGuide* (in diverse performance metrics) consistently across various VR projects (even those developed with different Unity versions).

Answer to RQ1: Experimental results on all 11 VR projects demonstrate that *VRExplorer* outperforms the SOTA approach *VRGuide* in EC, MC, and IOC. In particular, *VRExplorer* achieves the average performance gains over *VRGuide* in EC and MC by 122.8% and 52.8%, respectively, across all the projects. Moreover, *VRExplorer* converges faster or comparably faster than *VRGuide* while maintaining substantially higher coverage. Performance improvements are observed across diverse VR scenarios, demonstrating *VRExplorer*'s strong generalizability in covering code and interactable VR objects during automated testing.

B. Ablation Study

To assess the contribution of different components, we perform an ablation study by selectively removing modules from the proposed *VRExplorer*. Given that *VRExplorer* is developed on top of the EAT framework, which decomposes VR interactions into reusable action units. Thus, EAT serves as the core exploration task of inherently focusing on interacting with objects. As a result, our ablation primarily targets the interaction-related components within EAT. The EAT framework constitutes the key methodological contribution of *VRExplorer* and is the only component that can be meaningfully isolated while preserving system functionality. In contrast, other modules and components serve as indispensable

TABLE VIII: Results of Ablation Study

Projects	Approaches	EC (%)	MC (%)
VR-Basics	*VRGuide*	41.38	53.22
	VRExplorer	**80.17**	**91.93**
	VRExplorer w/o *T*	68.10 (-15.0%)	77.42 (-15.9%)
	VRExplorer w/o *Tf*	59.24 (-26.1%)	70.00 (-16.2%)
VR-Room	*VRGuide*	40.97	50.63
	VRExplorer	**77.61**	**83.54**
	VRExplorer w/o *G*	58.52 (-24.6%)	69.62 (-16.4%)
	VRExplorer w/o *T*	64.12 (-17.3%)	67.00 (-19.7%)
VGuns	*VRGuide*	28.68	38.89
	VRExplorer	**77.57**	**77.78**
	VRExplorer w/o *TG*	50.37 (-35.3%)	61.11 (-16.7%)
	VRExplorer w/o *AE*	65.07 (-16.1%)	63.89 (-17.9%)

prerequisites for enabling exploration in VR scenes and are tightly integrated into the overall pipeline, rendering their removal infeasible for independent evaluation.

We then evaluate these methods by comparing them with the full-fledged *VRExplorer* framework in terms of EC and MC. Notably, we also compare them with *VRGuide* to investigate the performance improvement contributed by each module.

Ablation Experiment Configuration. As described in § III-C, interaction behaviors play a crucial role in the EAT framework. To investigate these interaction behaviors, we remove a specific interaction from a tested project's modules (as this module is used in this project). For example, we obtain VRExplorer without (w/o) *T* by removing the *Triggerable* module from the Entity layer's interface `Triggerable`, the Action layer's class `TriggerAction`, all tasks involve triggering in the Task layer, and the corresponding Mono C# scripts `XRTriggerable.cs`. Similarly, we obtain *VRExplorer*'s other ablated variants: w/o *Tf* (without *Transformable*), w/o *G* (without *Grabbable*), w/o *TG* (without both *Triggerable* and *Grabbable*), and w/o *AE* (without *autonomous event*).

Results of Ablation Study. Table VIII presents the experimental results of the ablation study in three projects: `VR-Basics`, `VR-Room`, and `VGuns`, where the best results are highlighted with underline and bold. These ablation experiments cover all the *VRExplorer*'s variants and span all the Unity versions of the projects in Group 2 (from 2020 to 2022). Specifically, we compare *VRGuide* and *VRExplorer* with its five ablated variants: *VRExplorer* w/o *G*, w/o *T*, w/o *Tf*, w/o *TG*, and w/o *AE*. To quantitatively evaluate the effect of removing each module, we also calculate the performance gain of a method over another one according to Eq. (1).

In `VR-Basics`, the full-fledged *VRExplorer* achieves the best performance in terms of 80.17% EC and 91.93% MC. We observe that removing the *Triggerable* module causes a 15.0% ($\frac{68.10-80.17}{80.17} \times 100\%$) decrease in EC and a 15.9% decrease in MC and the removal of the *Transformable* module leads to a 26.1% decrease in EC and 16.2% in MC. In `VR-Room`, we also find that the removal of the *Grabbable* module has the most pronounced effect, i.e., reducing EC by 24.6% and MC by 16.4%. Disabling the *Triggerable* module also leads to an ELOC decrease of 17.3% and a MC decrease of 19.7%. In `VGuns`, we observe that the removal of both the *Triggerable* and *Grabbable* modules causes a 35.3% decrease in EC and a 16.7% decrease in MC. Moreover, the removal of the *autonomous event* module causes a 16.1% decrease in EC and a 17.9% decrease in MC, indicating that the *autonomous event* module as described in Step ❸ (§ III-D) also contributes significantly. These results confirm the significant contribution of each interaction module to *VRExplorer*.

Overall, removing any individual module leads to a substantial drop in code coverage (in terms of EC and MC), further confirming the importance of integrating all three interaction-aware modules in the EAT framework. The full-fledged *VRExplorer* consistently yields the highest code coverage.

Answer to RQ2: Removing any module from the EAT leads to a substantial decrease in EC and MC. The degree of impact caused by the ablated module varies across diverse projects and interaction types. These results validate the necessity and effectiveness of integrating all the modules in *VRExplorer* for comprehensive interaction-aware VR testing.

C. Bug Detection in VR Projects

Since the primary goal of software testing is to detect or identify potential bugs, we also investigate whether the proposed *VRExplorer* can detect real-world bugs in VR projects. We have scanned all 11 projects through comprehensive testing conducted by *VRExplorer*. Thereafter, we have successfully detected two *functional*[5] bugs and one *non-functional* bug in total, from projects `EscapeGameVR`, `UnityCityView`, and `EscapeGameVR`. Notably, two functional bugs have been found in `EscapeGameVR` and `UnityCityView` while we have detected one non-functional bug in `EscapeGameVR`. Moreover, the bug from `UnityCityView` has been fixed by developers (after checking the commits), while two bugs from `EscapeGameVR` reported by us have not been confirmed by developers so far. Our *VRExplorer* can detect all these VR bugs, which are either functional or non-functional. Although *VRGuide* is able to detect the bug in project `UnityCityView`, it misses the other two bugs in `EscapeGameVR`.

Detected Bugs Validity Confirmation. To further confirm the validity of detected bugs, we validated all reported issues through Unity console logs, runtime exception traces, and script-trigger chains. Among them, the bug in `UnityCityView` has been confirmed and fixed by its developers (evidenced by later commits). For the two bugs in `EscapeGameVR`, we performed root-cause analyses: the functional bug is an *Unassigned Reference Exception*, consistent with CWE-395 [35] and CodeQL's rule on improper initialization [36], while the non-functional bug is caused by a missing `ArrowPrefab` resource, which is a common Unity issue documented in the official Prefab guidelines [37]. We also reported these issues to the original developers of `EscapeGameVR`, but no response was received till now.

Functional Bug Root-cause Analysis. Fig. 9 shows the testing process in Unity and Fig. 10 shows a screenshot of one functional bug detected by our *VRExplorer* in project `EscapeGameVR`. This bug occurs due to *assignment exception*, which can be explained as follows. When a bow releases an arrow, the method `ReleaseArrow()` is invoked

[5]A functional bug refers to a bug that causes the VR application not to work as expected, while a non-functional bug denotes a bug not directly related to the functionality (may be relevant to performance and usability issues).

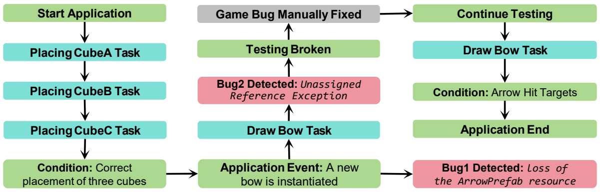


Fig. 9: Testing Process of `EscapeGameVR`

with the attempt to access the `arrowSpawnPoint` field of the `ArrowController` class. However, the developers have mistakenly failed to assign a value to this field (or the assignment was unintentionally lost), thereby resulting in an *Unassigned Reference Exception* [35, 36] bug, which is a Unity-specific runtime exception occurring when a serialized reference field (typically declared as `public` or marked with `[SerializeField]` [38]) in a `MonoBehaviour` class is accessed without being assigned a value in the Unity Inspector.

Non-functional Bugs Root-cause Analysis. Another non-functional bug in `EscapeGameVR` is caused by the loss of the `ArrowPrefab` resource. This bug can be identified by our *VRExplorer* through a simple manual inspection. The reason why these two bugs are not detected by *VRGuide* is that triggering these two bugs requires complicated actions and multiple types of interactions, while *VRGuide* cannot handle them. Differently, our *VRExplorer* can complete this difficult testing task through our model-based framework. In particular, *VRExplorer* requires the correct placement of three cubes onto platforms of the corresponding color, thereby ensuring that none of them fall outside the designated platform areas. Once this condition is satisfied, a new bow is instantiated. Consequently, triggering the bug needs the bow to be drawn and released by *VRExplorer*. This type of complex interactive task contains two or even more interactive patterns, which can not be completed by *VRGuide* (only supporting "clicking" interaction), while our approach handles this task properly.

Answer to RQ3: *VRExplorer* can successfully detect three real-world bugs in VR projects. More importantly, our *VRExplorer* is capable of detecting complex VR bugs, which can only be triggered after complex interactions.

VI. Discussions

Threats to External Validity. Although we have covered as many types of VR applications (developed by diverse Unity versions) as possible, there are still types of scenarios and interaction patterns not fully covered in our approach. Moreover, while our framework is currently tailored for VR applications developed in Unity, it can be extended to support other engines, such as Unreal Engine [39], with further adaptation, thereby enabling its broader applicability.

Threats to Internal Validity. The primary threats to internal validity arise from potential human errors during the selection of Core Code (CC) and the implementation of the interface. Specifically, during the CC selection phase, errors could be introduced due to subjective judgment. To minimize this threat, we have leveraged a majority-voting

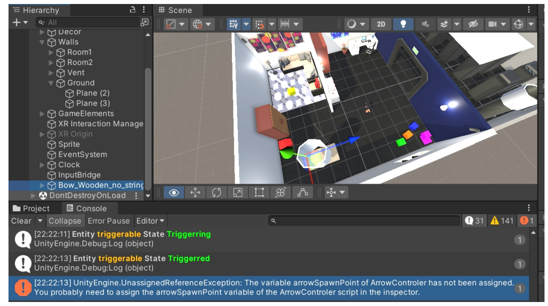


Fig. 10: A Detected Bug in `EscapeGameVR`

mechanism, where four test engineers (with VR developing experience) have independently selected code segments. The final decision has been made according to the majority vote. Moreover, during the interface implementation phase, potential inconsistencies or biases among test engineers are also present. To address this threat, we have arranged for four test engineers to collaboratively discuss the design and reach a consensus.

Discussion on Path-finding Algorithms. We also revisit the design choice of adopting the Greedy algorithm for navigation. As described in § III-D, *VRExplorer* supports both a Greedy strategy and a Backtracking-with-Pruning (B&P) algorithm. The latter solves the navigation task as a Traveling Salesman Problem (TSP) to compute globally optimal Hamiltonian paths. To quantify their differences, we performed additional experiments on randomized tasks with up to 100 interactable objects. Results show that the B&P algorithm achieves a 17.5% improvement in runtime efficiency (1233.70s vs. 1494.41s over 100 rounds) over the Greedy baseline. However, this improvement comes at the cost of significantly higher computational overhead, leading to reduced frame rates. This trade-off confirms our design decision: the Greedy algorithm remains more practical for maintaining smooth interactive performance in real-time testing than the B&P algorithm.

Impacts of Speed Parameters. To explore optimal speed parameters, we test three sets of values in `unity-vr-maze`. Specifically, we have chosen (i) MS = 3 m/s, TS = 30 deg/s, (ii) MS = 6 m/s, TS = 30 deg/s; and (iii) MS = 6 m/s, TS = 60 deg/s. Fig. 11 shows the coverage performance of *VRExplorer* compared to *VRGuide* in `unity-vr-maze` with different speed parameters. We observe the same trends for all three parameters. Notably, *VRExplorer* takes less time to reach convergence than *VRGuide* with higher EC and IOC. These results also indicate that TS has a minimal impact on efficiency, whereas MS shows a more significant effect.

Generality of VRExplorer. Our approach systematically unifies these diverse interactions by decomposing VR objects into a finite set of interactable features, which are then represented as abstract actions. Based on OOP principles, the Entity Layer and Action Layer encapsulate input modalities and target objects separately, effectively treating them as reusable and generic components. Within this feature–action hierarchy, abstract actions are further integrated into PFSM as semantic nodes and transitions, allowing frequent and domain-specific behaviors to be expressed uniformly. This design ensures

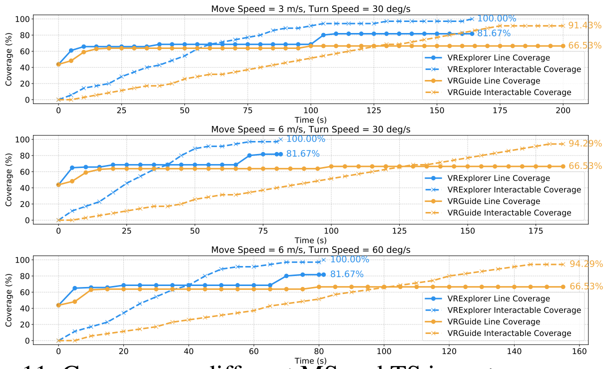


Fig. 11: Coverage on different MS and TS in `unity-vr-maze`

generality across projects while preserving extensibility to new interactions. For those not mentioned in § III, it only requires modeling a simple composition of task nodes and an additional configuration, without modifying the core logic. This OOP-based design allows the framework to scale naturally to a wide spectrum of VR interactions. For example, *GUI Interactions* (e.g., typing and pressing) can be represented by customized tasks composed of *Trigger Action* nodes, which provide event lists and function call chains for simulating different GUI interactions. Similarly, *Hand Gesture*-based interactions can be supported by treating gestures as input-trigger components in the Entity Layer, while a dedicated *Gesture Action* class in the Action Layer encapsulates concrete gestures such as swipe, pinch, and point. These gestures can then be mapped to task transitions in PFSM, e.g., "swipe-to-turn-the-page" or "pinch-to-zoom," in § III-D. In summary, the EAT framework is inherently *agnostic* to input types: each project only needs to define its input actions and associated targets, after which abstraction and task composition are handled systematically through the generic OOP framework.

Limitations and Future Work. *VRExplorer* still needs to analyze scenes manually and implement customized interfaces, inevitably introducing some extra workloads (despite being low) and a certain learning curve for test engineers. As future work, we will improve *VRExplorer*'s capability of automatically understanding VR scenes inspired by recent advances in multi-modal large models. Further, we will also automate the interface implementation and task model generation.

VII. Related Work

Automated Game Testing. As a code-based data augmentation technique, GLIB [40] can automatically detect game GUI glitches. Macklon et al. [41] present an approach to automatically detect visual bugs in `<canvas>` games. Prasetya et al. [42] leverage graph-based path-finding techniques in automated game navigation and exploration. As a Java-based multi-agent programming framework, IX4XR [43, 44] facilitates game testing by enabling test agents to interact with the game under test. As a Belief-Desire-Intention library, APLIB [32] supports developing intelligent agents capable of executing complex testing tasks. Ferdous et al. [45] propose a model-based approach by leveraging EFSM to model game behavior. However, these game testing approaches are generally tailored to specific types of games instead of VR applications.

Automated Mobile Application Testing. *Stoat* [46] is a stochastic model-based testing tool for Android apps with combined dynamic and static analysis to generate tests. Chen et al. [47] propose a model-based GUI testing approach for HarmonyOS applications with the adoption of the *arkxtest*[48] framework. AutoConsis [49] leverages a specially tailored Contrastive Language-Image Pre-training (CLIP) multi-modal model to automatically analyze Android 2D GUI pages. FASTBOT2 [50] leverages a probabilistic model that memorizes key information for testing based on a model-guided testing strategy. KEA [51] is a property-based testing tool for finding functional bugs in Android apps. However, these mobile application testing approaches can only address 2D GUI testing.

RL-Based Testing. Tufano et al. [52] employ RL to train an agent to play games in a human-like manner to identify areas that lead to FPS drops. RLBT [31] applies a curiosity-based RL approach to automate game testing by maximizing coverage. Bergdahl et al. [53] adopt a modular approach, in which RL complements classical test scripting. *Wuji* [54] leverages evolutionary algorithms, RL, and multi-objective optimization for game testing. However, RL-based approaches alone are very limited in complex VR scene exploration.

VR Application Testing. Rzig et al. [8] analyze 314 open-source VR applications, revealing that 79% of them lack automated tests. Harms [55] proposes an automated approach that extracts task trees from real VR usage recordings to detect usability smells without predefined tasks or settings. PREDART [9] predicts human ratings of virtual object placements, serving as test oracles in AR testing. *VRTest* [10] extracts information from a VR scene and controls the user's camera to explore the scene and interact with virtual objects. *VRGuide* [11] applies a computational geometry technique called Cut Extension to optimize camera routes for covering all interactable objects. Qin and Weaver [12] explore Generative AI for field of view analysis in VR testing. However, these approaches cannot address complicated VR interactions.

VIII. Conclusion

In this paper, we design the EAT framework, a generic three-layer abstraction framework based on OOP for modeling complex interaction behaviors and tasks in testing VR applications. Based on EAT, we present *VRExplorer*, a novel model-based testing tool to achieve effective interactions with diverse virtual objects and explorations of complex VR scenes. To validate the performance of our approach, we evaluate *VRExplorer* on 11 representative VR projects. The experimental results validate our approach's superior performance to the SOTA method in terms of coverage and efficiency, as well as the ability to detect complicated real-world bugs.

IX. Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB2704100, the National Natural Science Foundation of China (No. 62032025), the Seed Funding for Collaborative Research Grants of HKBU (with Grant No. RC-SFCRG/23-24/R2/SCI/06), the Major Key Project of Peng Cheng Laboratory under Grant PCL2025AS07.

REFERENCES

[1] R. Radoeva, "Overview on hardware characteristics of virtual reality systems," in *2022 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*. IEEE, 2022, p. 31.00.

[2] P. Rajeswaran, J. Varghese, P. Kumar, J. Vozenilek, and T. Kesavadas, "AirwayVR: Virtual Reality Trainer for Endotracheal Intubation," in *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2019, pp. 1345–1346.

[3] D. Hamilton, J. McKechnie, E. Edgerton *et al.*, "Immersive virtual reality as a pedagogical tool in education: a systematic literature review of quantitative learning outcomes and experimental design," *Journal of Computers in Education*, vol. 8, pp. 1–32, 2021.

[4] Y. Liu, Q. Sun, Y. Tang, Y. Li, W. Jiang, and J. Wu, "Virtual reality system for industrial training," in *2020 International Conference on Virtual Reality and Visualization (ICVRV)*, 2020, pp. 338–339.

[5] B. Wang, L. Zheng, Y. Wang, W. Fang, and L. Wang, "Towards the industry 5.0 frontier: Review and prospect of XR in product assembly," *Journal of Manufacturing Systems*, vol. 74, pp. 777–811, 2024.

[6] M. A. Muhanna, "Virtual reality and the cave: Taxonomy, interaction challenges and research directions," *Journal of King Saud University - Computer and Information Sciences*, vol. 27, no. 3, pp. 344–361, 2015.

[7] F. B. Insights, "Virtual Reality (VR) Market Size, Share & Industry Analysis," https://www.fortunebusinessinsights.com/industry-reports/virtual-reality-market-101378, 2024.

[8] 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, p. 1269–1281.

[9] T. Rafi, X. Zhang, and X. Wang, "Predart: Towards automatic oracle prediction of object placements in augmented reality testing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[10] X. Wang, "VRTest: An Extensible Framework for Automatic Testing of Virtual Reality Scenes," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 232–236.

[11] X. Wang, T. Rafi, and N. Meng, "VRGuide: Efficient Testing of Virtual Reality Scenes via Dynamic Cut Coverage," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2024, p. 951–962.

[12] X. Qin and G. Weaver, "Utilizing Generative AI for VR Exploration Testing: A Case Study," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2024, p. 228–232.

[13] S. Li, B. Li, Y. Liu, C. Gao, J. Zhang, S.-C. Cheung, and M. R. Lyu, "Grounded GUI Understanding for Vision Based Spatial Intelligent Agent: Exemplified by Virtual Reality Apps," 2024. [Online]. Available: https://arxiv.org/abs/2409.10811

[14] J. Wentzel, M. Lakier, J. Hartmann, F. Shazib, G. Casiez, and D. Vogel, "A Comparison of Virtual Reality Menu Archetypes: Raycasting, Direct Input, and Marking Menus," *IEEE Transactions on Visualization and Computer Graphics*, vol. 31, no. 9, pp. 4868–4882, 2025.

[15] T. Wan *et al.*, "Design and evaluation of controller-based raycasting methods for efficient alphanumeric and special character entry in virtual reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 30, no. 9, pp. 6493–6506, 2024.

[16] VentureBeat, "Unity financial results (q2-2024)," 2024. [Online]. Available: https://venturebeat.com/games/unity-financial-results-q2-2024/

[17] "XR Interaction Toolkit," 2024. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@3.0/manual/index.html

[18] V. Juránek, "Virtual reality toolkit for the unity game engine [online]," 2021 [cit. 2025-01-24].

[19] "SteamVR," 2024. [Online]. Available: https://github.com/ValveSoftware/steamvr_unity_plugin

[20] S. Bovet, A. Kehoe, K. Crowley, N. Curran *et al.*, "Using Traditional Keyboards in VR: SteamVR Developer Kit and Pilot Game User Study," in *2018 IEEE Games, Entertainment, Media Conference (GEM)*, 2018, pp. 1–9.

[21] V. Corporation, "Steam VR Plugin," https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647, 2024.

[22] "Mixed reality toolkit," 2022. [Online]. Available: https://github.com/microsoft/MixedRealityToolkit-Unity

[23] S. Ong and V. K. Siddaraju, *Introduction to the Mixed Reality Toolkit*. Berkeley, CA: Apress, 2021, pp. 85–110. [Online]. Available: https://doi.org/10.1007/978-1-4842-7104-9_4

[24] "Unity monobehaviour class," 2022. [Online]. Available: https://docs.unity3d.com/2022.3/Documentation/ScriptReference/MonoBehaviour.html

[25] U. Technologies, "Building a navigation mesh," 2020. [Online]. Available: https://docs.unity3d.com/2020.1/Documentation/Manual/nav-BuildingNavMesh.html

[26] Unity Technologies, "Unity - manual: Navmesh agent," 2025. [Online]. Available: https://docs.unity3d.com/Manual/class-NavMeshAgent.html

[27] U. Technologies, "Unity - manual: Nav mesh obstacle," Unity Technologies, 2025. [Online]. Available: https://docs.unity3d.com/Manual/class-NavMeshObstacle.html

[28] J. Hertel, S. Karaosmanoglu, S. Schmidt, J. Bräker, M. Semmann, and F. Steinicke, "A taxonomy of interaction techniques for immersive augmented reality based on an iterative literature review," in *2021 IEEE International*

Symposium on Mixed and Augmented Reality (ISMAR), 2021, pp. 431–440.

[29] Unity Technologies, "UnityEvent Documentation," 2025. [Online]. Available: https://docs.unity3d.com/6000.0/Documentation/ScriptReference/Events.UnityEvent.html

[30] Y. Lan, Y. Lu, M. Pan, and X. Li, "Navigating Mobile Testing Evaluation: A Comprehensive Statistical Analysis of Android GUI Testing Metrics," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 944–956.

[31] R. Ferdous, F. Kifetew, D. Prandi, and A. Susi, "Towards Agent-Based Testing of 3D Games using Reinforcement Learning," in *the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[32] I. S. W. B. Prasetya, M. Dastani, R. Prada, T. E. J. Vos, F. Dignum, and F. Kifetew, "Aplib: Tactical Agents for Testing Computer Games," in *Engineering Multi-Agent Systems*. Springer, 2020, pp. 21–41.

[33] "Code coverage," 2023. [Online]. Available: https://github.com/needle-mirror/com.unity.testtools.codecoverage

[34] M. Foxman, B. Klebig, A. Leith, D. Beyea, R. Ratan, and V. Chen, "Virtual reality genres: Comparing preferences in immersive experiences and games," in *Extended Abstracts of the 2020 Annual Symposium on Computer-Human Interaction in Play*, 11 2020.

[35] MITRE, "CWE-395: Use of NullPointerException Catch to Detect NULL Pointer Dereference," https://cwe.mitre.org/data/definitions/395.html, 2021.

[36] GitHub CodeQL, "Avoid catching nullreferenceexception," https://codeql.github.com/codeql-query-help/csharp/cs-catch-nullreferenceexception/, 2022.

[37] Unity Technologies, "Brokenprefabasset," https://docs.unity3d.com/6000.1/Documentation/ScriptReference/BrokenPrefabAsset.html, 2025.

[38] ——, "Unity Scripting API: SerializeField," https://docs.unity3d.com/6000.1/Documentation/ScriptReference/SerializeField.html, 2024.

[39] Epic Games, "Unreal engine 5," 2025. [Online]. Available: https://www.unrealengine.com/en-US/unreal-engine-5

[40] K. Chen, Y. Li, Y. Chen, C. Fan, Z. Hu, and W. Yang, "Glib: towards automated test oracle for graphically-rich applications," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, p. 1093–1104.

[41] F. Macklon *et al.*, "Automatically Detecting Visual Bugs in HTML5 Canvas Games," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[42] Prasetya *et al.*, "Navigation and exploration in 3D-game automated play testing," in *Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2020, p. 3–9.

[43] I. S. W. B. Prasetya, F. Pastor Ricós *et al.*, "An agent-based approach to automated game testing: an experience report," in *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, 2022.

[44] S. Shirzadehhajimahmood *et al.*, "Using an agent-based approach for robust automated testing of computer games," in *Proceedings of the 12th International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2021, p. 1–8.

[45] R. Ferdous, F. Kifetew, D. Prandi, I. S. W. B. Prasetya, S. Shirzadehhajimahmood, and A. Susi, "Search-based automated play testing of computer games: A model-based approach," in *Search-Based Software Engineering: 13th International Symposium*, 2021, p. 56–71.

[46] T. Su, G. Meng *et al.*, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, p. 245–256.

[47] Y. Chen, S. Wang, Y. Tao, and Y. Liu, "Model-based GUI Testing For HarmonyOS Apps," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 2411–2414.

[48] "Arkxtest," 2022. [Online]. Available: https://gitee.com/openharmony/testfwk_arkxtest

[49] Y. Hu, H. Jin, X. Wang *et al.*, "AutoConsis: Automatic GUI-driven Data Inconsistency Detection of Mobile Apps," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, 2024, p. 137–146.

[50] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[51] Y. Xiong, T. Su, J. Wang, J. Sun, G. Pu, and Z. Su, "General and practical property-based testing for android apps," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 53–64.

[52] R. Tufano, S. Scalabrino, L. Pascarella, E. Aghajani, R. Oliveto, and G. Bavota, "Using reinforcement learning for load testing of video games," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2303–2314.

[53] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén, "Augmenting automated game testing with deep reinforcement learning," *2020 IEEE Conference on Games (CoG)*, pp. 600–603, 2020.

[54] Y. Zheng, X. Xie, T. Su *et al.*, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.

[55] P. Harms, "Automated usability evaluation of virtual reality applications," *ACM Transactions on Computer-Human Interaction*, vol. 26, no. 3, Apr. 2019.