Journal of Innovations in Computer Science and Engineering
Shahid Beheshti Unversity

January 2026, Volume 3, Issue 2

Test Data Generation Using a Game with a Purpose

Nima Salami✉, **CODE ORCID: 0009-0003-1493-8681**
Faculty of Computer Science and Engineering Shahid Beheshti University of Technology Tehran, Iran, n_salami@sbu.ac.ir
Hassan Haghighi, **CODE ORCID: 0000-0002-6145-4095**
Faculty of Computer Science and Engineering Shahid Beheshti University of Technology Tehran, Iran, h_haghighi@sbu.ac.ir

Abstract
Software testing is a crucial and resource-intensive phase in the software development life cycle, where test data generation (TDG) remains a major bottleneck. While automated tools exist, human intuition is still valuable for resolving complex logic. To optimize this tedious task, we propose a 3D Game with a Purpose (GWAP) that crowdsources TDG. Unlike prior studies, our 3D environment maps program execution paths to interactive spatial puzzles and visually represents complex data structures, such as arrays and records, through customizable character attributes. This specific visual mapping enhances player interaction and addresses previous structural limitations. We evaluated the proposed approach using standard benchmark programs. Quantitative results demonstrate that players achieved 100% coverage across all 279 feasible execution paths—out of 511 total analyzed paths, 232 of which were infeasible—in an average of 2۹ minutes, whereas the baseline game failed to generate puzzles for 30.5% of these feasible paths (85 of 279) due to structural limitations. Furthermore, user evaluations show engagement among players with varying levels of expertise. The results confirm that the proposed 3D GWAP successfully handles complex input parameters while transforming a technical testing task into an effective, user-friendly environment.

CC BY

KEYWORDS
Software Testing, Game with a Purpose (GWAP), Test Data Generation, Serious game, Human-based computation game, Human-based software testing, Crowdsourcing.

1. INTRODUCTION

Software testing is a crucial phase in the software development lifecycle, playing a fundamental role in evaluating the quality of the final product. This process is typically time-consuming and resource-intensive, accounting for approximately half of total software development costs. The primary objective of software testing is to design a suite of tests capable of detecting the maximum number of potential faults within the software [1]. A critical step in this process is Test Data Generation, which directly impacts the efficiency and effectiveness of the testing phase. TDG involves defining a set of input values that satisfy specific testing criteria. While various manual and automated techniques exist for generating test data [1, 4, 5], the software industry still heavily relies on manual generation by testing specialists. This manual approach is not only costly and challenging but also makes defining all possible inputs for even medium-sized programs practically impossible. Furthermore, it often leads to a decline in motivation and productivity among testers over time. Consequently, researchers are actively seeking automated methods to optimize this process.

Automated TDG plays a key role in enhancing testing efficiency, aiming to achieve maximum code coverage and identify potential faults in the shortest possible time [1, 4, 5]. Among novel approaches, GWAP have emerged as an innovative method for TDG. By simulating real-world scenarios within a game environment and defining specific objectives, test data can be generated dynamically according to the program's requirements. This approach not only increases test coverage and facilitates the identification of rare, complex errors, but it also effectively reduces testing-related costs while improving software quality [8, 10].

Despite significant advancements in test automation, there remain open areas for improving the automation of testing activities. As mentioned, a major challenge in test design is generating data that satisfies selected coverage criteria. Existing TDG methods generally fall into three categories [1]:

1. **Random Test Data Generation:** Data is generated randomly to satisfy coverage criteria. While easy to implement, it generally achieves lower coverage compared to other techniques.
2. **Symbolic Execution:** Symbolic values are assigned to program parameters. A logical constraint is constructed for each execution path based on the path conditions. Solving these constraints yields the required test data. However, this method suffers from path explosion, heavy reliance on constraint solvers, and difficulties with functions of unknown behavior.
3. **Search-Based Software Testing (SBST):** This technique employs meta-heuristic algorithms (e.g., Genetic Algorithms, Ant Colony [18, 21] Optimization, Particle Swarm Optimization) guided by a fitness function to evaluate and score inputs based on testing requirements. A major drawback of SBST is its limitation in handling non-numeric data types and dynamic data structures (like trees and linked lists) [19, 23].

Submit Date: 2026-06-22
Revise Date: 2026-09-25
Accept Date: 2026-11-03
✉ Corresponding author

Human-generated test data benefits from human cognitive abilities, allowing for the comprehension and interpretation of complex logic that automated systems might struggle with. GWAP leverages this concept by embedding technical tasks within an entertaining environment, utilizing human computational power to solve complex problems without the players needing to understand the underlying technical operations.

Despite the potential of GWAP and crowdsourcing-which have been successfully applied in fields like cybersecurity and education [6]-existing test data generation games face notable limitations. Previous games such as Rings [2], Greenify [9], and QOTE [3] map program paths to puzzles like pipe networks, sliders, or gear mechanisms. However, these games suffer from restricted space for representing program elements, severe challenges in mapping complex data types like arrays and records, and repetitive 2D mechanics that quickly lose player engagement. Therefore, there is a need for a newly designed GWAP that overcomes these structural and engagement limitations [11].

To justify this choice, it is important to distinguish GWAP from other interactive crowdsourcing [13] approaches, such as gamification and serious games. Gamification integrates game elements into traditional workflows; however, it still requires participants to possess technical testing expertise [14]. Serious games, on the other hand, are primarily designed for training and educational purposes. In contrast, a GWAP completely abstracts the underlying technical problem. By translating complex path constraints into intuitive puzzles, a GWAP allows a non-expert crowd to solve test data generation problems purely through gameplay, without requiring any programming knowledge. This unique capability to harness human intuition for constraint solving—while hiding the software engineering complexity—makes GWAP suitable approach for this study.

Addressing the aforementioned challenges, this study aims to answer the following research questions:

1. How can complex input parameters, specifically arrays and records, be modeled within a game environment to generate valid test data?
2. Given the inclusion of arrays and records, how should execution paths and internal program conditions be mapped and visually represented in the game?
3. How should the game flow and mechanics be designed to create a more engaging environment than existing works, thereby increasing player participation?

1.1. Contributions

The primary objective of this study is to combine symbolic execution, the crowdsourcing paradigm, and GWAP to design and implement an engaging 3D game for test data generation. By transforming a tedious technical task into an enjoyable 3D experience, this approach aims to increase the diversity of the generated test data, achieve higher code coverage, and reduce the costs associated with expert-driven manual testing.

Specifically, this paper introduces a 3D game designed to resolve the limitations of prior GWAP solutions for TDG. The core contributions of this work are:

- **Support for Complex Inputs:** Unlike previous works that focused on simple primitives, the proposed game successfully models and covers complex data types, specifically arrays and records. Rather than relying on traditional abstract numbers, these inputs are modeled visually through the customizable attributes of in-game characters.
- **Enhanced Condition Mapping:** The system provides an engaging and flexible mapping for various program constraints (including standard arithmetic, logical, and relational operators). Execution paths and logical conditions extracted from the source code are effectively mapped as interactive in-game challenges.
- **3D Interactive Environment:** Transitioning from traditional 2D layouts to a 3D game boosts player engagement and allows for a more intuitive spatial visualization of execution paths. Furthermore, the system incorporates an intuitive feedback mechanism utilizing scores and visual cues to guide players toward correct solutions.

To realize these contributions, the proposed GWAP system accepts C# source code as input and follows a methodology consisting of six main steps:

1. **Preprocessing:** The target program is scanned to identify conditions, loops, and data structures.
2. **Code Mapping:** Program statements and constraints are automatically mapped to comprehensible game elements.
3. **Game Design and Implementation:** A 3D game is constructed based on the mapped elements, featuring various puzzle levels designed for both accuracy and entertainment.
4. **Crowdsourced Gameplay:** Players interact with the puzzles, indirectly generating diverse test data through their gameplay decisions.
5. **Player Evaluation:** Post-game surveys are conducted to measure player satisfaction, ease of use, and willingness to replay, aiding in further iterative design improvements.
6. **Comparative Analysis:** The efficiency (time) and effectiveness (code coverage) of the generated test data are compared against manual expert generation and existing 2D games (such as Rings).

The target programs evaluated in this study are developed in C#; however, the proposed framework can be adapted to other programming languages with minor modifications to the preprocessing phase. The testing scope is focused on the unit testing level. Therefore, standard benchmark unit programs are utilized to accurately evaluate the proposed methodology, generating test data for each unit independently rather than conducting system-level testing.

2. RELATED WORK

To address the inherent limitations of traditional Automated Test Data Generation (ATDG) techniques—such as path explosion in symbolic execution and the difficulty of resolving complex constraints [15, 16]—recent research has shifted toward Human-in-the-Loop (HITL) methodologies. Rather than detailing the foundational concepts of coverage criteria or metaheuristic search algorithms (e.g., EvoSuite [22]), this section focuses on how human intelligence has been practically integrated into software engineering through gamification, serious games, and GWAP.

2.1. Gamification and Serious Games in Software Engineering

Initial efforts to utilize human intuition in software quality assurance largely relied on gamification and educational serious games. Platforms like Halo and Code Defenders apply game mechanics to traditional testing environments. For instance, Code Defenders frames mutation testing as a competitive scenario where players either create mutants or write unit tests [7]. Similarly, INVGame gamifies the extraction of loop invariants by asking participants to identify mathematical patterns [24]. In educational contexts, serious games such as SimSE [26], Robot-on, and Code-plants [25, 27] simulate project management or help novices visualize programming logic.

While these approaches successfully improve engagement, they fundamentally rely on the players underlying programming knowledge. Gamification motivates experts to perform tasks, but it does not abstract the source code sufficiently to allow non-experts to solve algorithmic constraints.

2.2. GWAP for Test Data Generation

To remove the necessity for programming expertise, GWAP transform complex software engineering problems into abstract, interactive puzzles. By solving these visual challenges, non-expert players implicitly satisfy program constraints.

A foundational tool in this domain is Rings [2], which converted symbolic execution paths into 2D visual puzzles. While Rings proved that crowdsourcing could generate valid test inputs, it encountered structural limitations, particularly regarding visual clutter on restricted screens and the inability to effectively represent complex data structures.

Subsequent frameworks, such as QOTE [3] and Greenify [9], mapped program units and branch conditions to interactive graphical elements like sliders and locks. In these systems, players adjust values to satisfy logical conditions, turning components "green" to unlock execution paths.

2.3. Limitations of Existing Methods and the Proposed Approach

Despite the conceptual success of Rings, QOTE, and Greenify, several critical gaps remain in the current literature:

1. **Data Type Representation:** Existing GWAP are almost exclusively designed around simple primitive variables (e.g., integers and booleans). They lack the structural design required to model and test complex data structures, such as arrays and records.
2. **Dimensionality Restrictions:** Current constraint-solving games are limited to 2D interfaces. As the complexity of nested branches and constraints increases, 2D representations become cluttered, reducing both player engagement and puzzle-solving efficiency.

The proposed methodology builds upon the principles of prior GWAP but directly addresses these limitations by replacing the 2D layout with an 3D spatial environment and introducing a novel mapping technique that translates complex data types into customizable character attributes.

3. PROPOSED METHOD

The primary objective of this research is to address the persistent challenges in software test data generation. As discussed, all automated test data generation techniques face inherent limitations, keeping human developers pivotal in practical settings. To mitigate the drawbacks of human-centric approaches (e.g., lack of motivation, high costs of specialists), this section presents a crowdsourcing method utilizing Serious Games.

In the proposed game, path constraints derived from the symbolic execution of the source code are automatically transformed into 3D puzzles. By solving these puzzles, players implicitly generate the required input values. Unlike previous studies, this research introduces a systematic mapping technique for complex data structures (arrays and records) within a dynamic 3D environment.

3.1. System Architecture and Automated Mapping Rules

The proposed framework operates in four sequential stages. Initially, the target C# method is analyzed via the .NET Roslyn SDK to extract relational predicates and branch conditions as symbolic path constraints. Subsequently, these constraints are mapped to spatial and physical mechanics, such as mapping threshold conditions to pressure plates or boundary obstacles. Next, the synthesized puzzle is instantiated within the Unity environment (C# / Unity 202٢ LTS). Finally, upon successful puzzle completion, the player's avatar attributes are extracted as concrete input values; these values are then executed against the original method to verify path coverage and finalize the test case.

Figure 1 summarizes this four-stage pipeline and the responsibilities of each module.

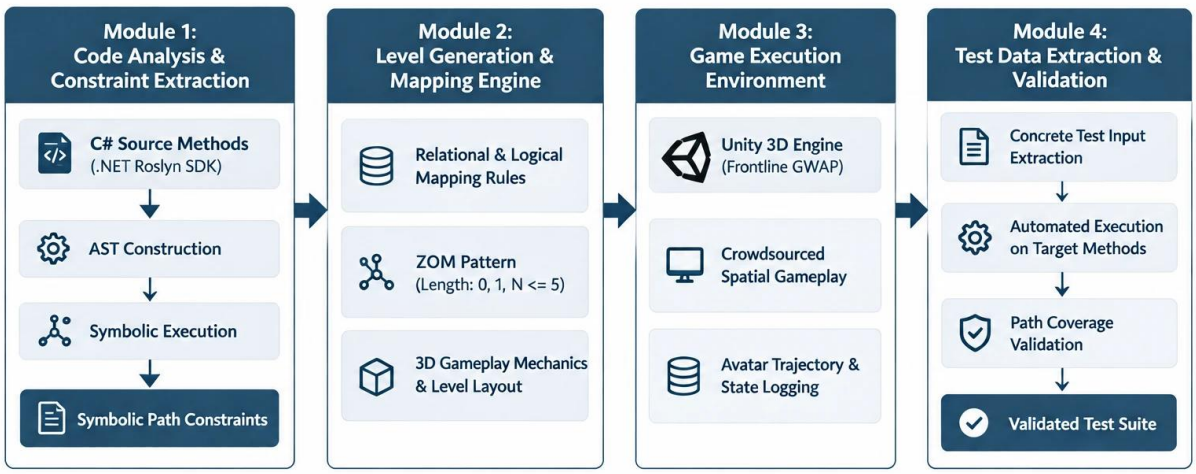


Figure 1. Frontline system architecture pipeline

The overall workflow of the Frontline framework is illustrated in Figure 1. As shown in the pipeline, the framework operates across four cohesive stages. First, in the Code Analysis & Constraint Extraction module, the target C# method is parsed via the .NET Roslyn compiler platform to construct Abstract Syntax Trees (AST) and derive symbolic path constraints. Second, the Level Generation & Mapping Engine deterministically translates these logical and relational predicates into spatial 3D mechanics and level layouts, applying the Zero-One-More (ZOM) pattern to bound array states ($N \leq 5$). Third, in the Game Execution Environment, human players interact with the automatically generated Unity 3D puzzles, navigating obstacles and pressure plates without requiring programming knowledge. Finally, the Test Data Extraction & Validation module logs spatial avatar trajectories, decodes them back into concrete programmatic inputs, and dynamically executes them against the original C# methods to achieve path coverage.

A critical aspect of the proposed methodology is the clear boundary between automated system processes and human intervention. The system operates as a fully automated pipeline up to the point of gameplay. The architecture consists of four sequential stages:

1. **Instrumentation and Symbolic Analysis (Automated):** The source code is parsed, and new instructions are inserted to track execution behavior without altering the logic. Symbolic execution is then applied to extract mathematical path constraints (e.g., $x > 5$ or $A_i == N$).
2. **Source Code to Game Element Mapping (Automated):** This is the core translation engine. Instead of manual level design, the system uses a deterministic rule set to translate logical operators and code constructs into 3D game mechanics:
    - *Relational Operators (>, >=):* Mapped to floor-based pressure plates, requiring the player to set an avatar's weight above a certain threshold.
    - *Relational Operators (<, <=):* Mapped to ceiling barriers or moving blades, restricting the avatar's height parameter.
    - *Logical AND (&&):* Mapped as sequential obstacles within a single corridor.
    - *Logical OR (||):* Mapped as parallel, branching paths where traversing any single path satisfies the condition.
    - *Loops and Arrays:* Mapped using a dynamic constraint generator based on the Zero-One-More (ZOM) pattern. ZOM bounds the search space for individual array lengths by testing three canonical states: 0 (empty), 1 (single-element), and N (multi-element, where $2 \leq N \leq 5$). However, for k independent arrays, the number of combinations scales to 3^k in the worst case. Thus, while ZOM restricts the state space of each individual array, it does not eliminate combinatorial growth across multiple arrays. In this implementation, N is capped at 5 to capture off-by-one boundary errors while keeping the 3D puzzle environment manageable for players.
3. **Game Puzzle Generation (Automated):** Based on the mapping rules, the 3D environment (obstacles, barriers, and visual indicators) is dynamically generated via the game engine without developer intervention.
4. **Gameplay and Data Generation (Human-in-the-loop):** Players (non-experts) enter the generated level, select avatar attributes (e.g., weight and height), and navigate obstacles. Successful navigation automatically writes the selected attributes to a test suite file.

3.2. Game Elements and Mechanics Design

Various elements have been designed to create an engaging and purposeful experience. The game's narrative revolves around a character named "Arman," who must overcome obstacles created by a mysterious virus by solving logical puzzles (which are equivalent to code conditions). A scoring system incentivizes players to choose their values carefully, while audio-visual feedback mechanisms guide them on their success or failure. This gamification approach transforms the tedious process of test data generation into a motivating, problem-solving activity.

3.3. Game Puzzle Design and Representative Examples

This section details the mapping from path conditions to game mechanics. We begin with a linear inequality to illustrate the baseline translation process from program constraints to spatial puzzles. While such conditions are easily handled by automated solvers, this introductory example clarifies the fundamental mechanics. We then examine more complex constraints, such as loops and arrays, where human intuition can serve as a complementary, human-in-the-loop alternative when automated constraint solving becomes computationally expensive.

3.3.1. Covering Scalar Parameters

Example 1: The source code is shown in Figure 2. The objective is to generate a test case for the execution path ABCBCBDEF, which corresponds to re-executing the while loop body and then entering the if body. The path constraint, derived from symbolic analysis, is defined as $2 < u < 4$.

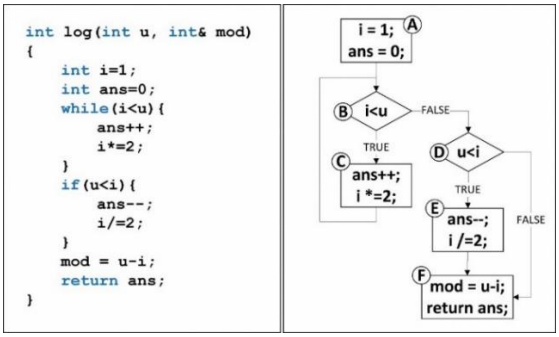


Figure 2. Source code for Example 1

To model this compound condition, two complementary mechanisms are used. The "greater than" condition ($u > 2$) is implemented with a pressure plate that activates only with a weight greater than 2. The "less than" condition ($u < 4$) is modeled as a height barrier with moving blades, requiring the player to set the avatar's height to less than 4 to avoid collision. This visual design intuitively guides the player to select values within the range (2, 4), as depicted in Figure 3.

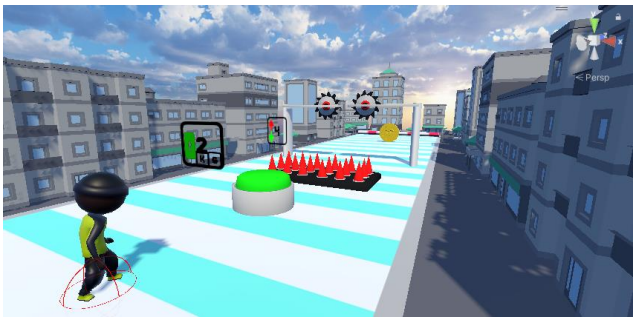


Figure 3. Image of the game environment for Example 1

3.3.2. Handling Complex Data Structures

To manage iterative structures and array-based inputs, we adopt the Zero-One-More (ZOM) pattern. ZOM abstracts the unbounded length of array inputs into three canonical sizes: empty (0), single-element (1), and multi-element (N, where $2 \leq N \leq 5$). This abstraction controls the infinite length dimension of individual arrays. However, it does not eliminate the combinatorial explosion caused by multiple arrays. In the worst case, a program with k independent arrays generates 3^k configurations. Our current evaluation targets typical unit-test scenarios with a limited number of arrays, where this growth remains manageable.

Example 2: This example addresses the challenge of path analysis when dealing with arrays (code in Figure 4). The inputs include a scalar variable and an array of fixed length five.

```
int number = 3;
int[] array = {1, 2, 3, 4, 5};
bool found = false;
foreach (int elem in array)
{
    if (elem == number)
    {
        found = true;
        break;
    }
}
Console.WriteLine(found);
```

Figure 4. Source code for Example 2

Following the ZOM pattern, three distinct puzzles are generated:

- Zero case: The array is empty, testing the system's behavior under this boundary condition.
- One case: The array contains a single element, analyzing the system's basic functionality.
- More case: The array is tested at its full capacity (five elements).

This three-pronged approach ensures that the program's behavior is thoroughly tested under boundary and typical conditions (Figure 5 and Table 1).

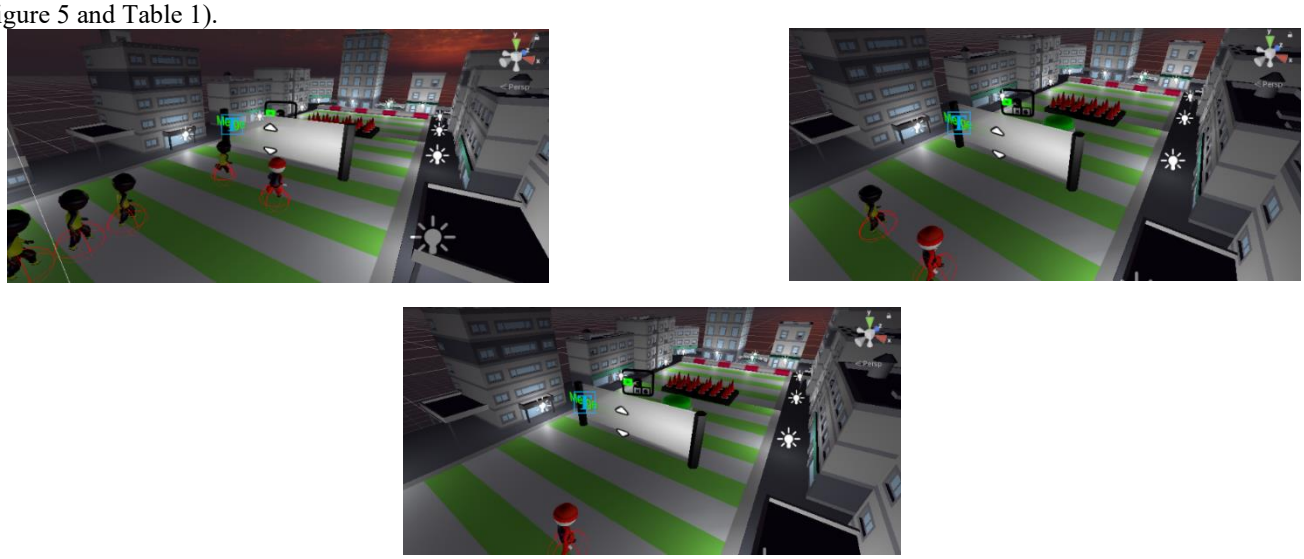


Figure 5. Image of the game environment for Example 2

Table 1. Path Constraints for Example 2

State	Symbolic Path Constraint	Zero, One, More Cases
number is found	$A_i == N$ for the first i; search terminates	Only the first match matters
Empty array	length(array) == 0 → found = false	Zero
Value in the first element	$A_0 == N$	One
Value in the last element	$A_0 \neq N \wedge A_1 \neq N \wedge \ldots \wedge A_{n-1} == N$	More

Example 3: This example examines an algorithm that checks for the equality of two arrays, disregarding the order of elements (code in Figure 6). This scenario is highly complex due to its nested loops.

```
array1 = new int[] {1, 2, 3};
array2 = new int[] {3, 2, 1};
bool sameElements = true;
foreach (int elem in array1)
{
    bool foundInArray2 = false;
    foreach (int elem2 in array2)
    {
        if (elem == elem2)
        {
            foundInArray2 = true;
            break;
        }
    }
    if (!foundInArray2)
    {
        sameElements = false;
        break;
    }
}
Console.WriteLine(sameElements);
```

Figure 6. Source code for Example 3

Since there are two input arrays, the system generates $3^2 = 9$ distinct puzzles based on the ZOM pattern. These puzzles cover all possible combinations (e.g., Zero-One, More-Zero, More-More). The most complex case (More-More) requires the player to set the weights of two independent groups of avatars (5 avatars each), as shown in Figure 7.



Figure 7. Image of the game environment for Example 3

Table 2. Path Constraints for Example 3

State	Symbolic Path Constraint	Element values are identical
Both arrays are empty	$\|A\| = \|B\| = 0$	true
At least one element from array1 is not in array2	$\exists A_i \in$ array1 $\rightarrow \forall B_j \in$ array2: $A_i \neq B_j$	false
All elements from array1 are in array2	$\forall A_i \in$ array1 $\rightarrow \exists B_j \in$ array2: $A_i == B_j$	true

Example 4: This example focuses on the record data structure (code in Figure 8). In this model, each record stores a student's information (ID and national code). Each field of the record is treated as a distinct element and is mapped to an avatar's attributes (height and weight).

```
public record Student(int StudentId, int NationalCode);
class Program
{
    static void Main()
    {
        Student student1 = new(401444411, 0025478993);
        Student student2 = new(401444411, 0025478993);

        Console.WriteLine($"student1 == student2: {student1 == student2}");

        Console.WriteLine("\nStudent1: " + student1);
        Console.WriteLine("Student2: " + student2);
    }
}
```

Figure 8. Source code for Example 4

When applying the ZOM pattern to records, the concept is adapted to evaluate the number of mismatched fields rather than the dynamic size of the structure. The mechanism for comparing records is implemented using a Splitter operator. The corresponding values of two students are subtracted from each other; if the result is zero for all fields (indicating identical values), the player passes the obstacle (Figure 9).



Figure 9. Image of the game environment for Example 4

Table 3. Path Constraints for Example 4

State	Symbolic Path Constraint	Zero, One, More Cases
Zero mismatched fields (Exact match)	$(ID_1 == ID_2) \wedge (NC_1 == NC_2)$	Zero
One mismatched field (Partial match)	$(ID_1 \neq ID_2) \wedge (NC_1 == NC_2)$	One
More mismatched fields (No match)	$(ID_1 \neq ID_2) \wedge (NC_1 \neq NC_2)$	More

4. EXPERIMENTAL RESULTS

This section evaluates the proposed serious game (referred to as Frontline) from two primary perspectives: technical efficiency in generating test data, and player engagement as a Purposeful Game (GWAP). The evaluation is conducted in two phases: comparing the proposed game-based method with traditional manual test data generation, and benchmarking it against an existing game-based approach (Rings).

٤.1. Evaluation Metrics

The performance is measured using technical and gamification metrics. Technically, we evaluate the **test suite generation time** and **path coverage**. For gamification, we utilize three standard GWAP metrics:

1. **Throughput:** The average number of problem instances solved per human hour, calculated as:

$$\text{Throughput} = \frac{\text{Average number of solved problems}}{\text{Time (hours)}} \quad (١)$$

2. **Average Lifetime Play (ALP):** The overall amount of time a player spends on the game, indicating user retention.
3. **Expected Contribution:** A composite metric representing the overall potential of the game in fostering effective participation:

$$\text{Expected Contribution} = \text{Throughput} \times \text{ALP} \quad (٢)$$

٤.2. Experimental Setup and Benchmarks

The game was developed using the Unity engine (C#), targeting cross-platform deployment. The evaluation utilized a standard benchmark suite comprising 19 C# programs (40 independent functions), selected based on previous credible studies in test data generation. These programs represent various computational complexities and include algorithms like Binary Search, Root Quadratic Equations, and Fisher statistics. Originally sourced from classic C/C++ benchmarks in previous studies, these programs were ported to C# to interface with Roslyn while preserving their original control flow structures. The benchmark suite contained a total of 511 test paths. Of these, 232 paths were identified as infeasible, leaving 279 feasible execution paths targeted for puzzle generation and coverage evaluation. The study involved two distinct participant groups:

- **Players:** 184 secondary and high school students (ages 12-18) with no formal programming expertise. To ensure experimental replicability and control for the learning curve, a standardized onboarding protocol was implemented. Prior to the recorded sessions, players received a brief verbal explanation of the game rules without any disclosure of the underlying software testing objectives. Following this, they completed a mandatory 15-minute interactive tutorial phase (comprising introductory levels). This phase allowed them to familiarize themselves with the user interface and

core mechanics (e.g., avatar movement, splitting, and merging) ensuring that the subsequent performance metrics reflected actual problem-solving efficiency rather than interface navigation learning.

- **Programmers:** 22 software developers and CS students with at least two years of professional experience, tasked with manual test data generation.

٤.3. Results and Comparison

٤.3.1. Frontline vs. Rings (Game-to-Game Comparison)

We evaluated both systems across the 279 feasible paths (excluding the 232 infeasible paths from the initial 511). Frontline successfully generated puzzles for all 279 paths. In contrast, Rings supported 194 of these feasible paths, failing to generate puzzles for the remaining 85 paths due to structural limitations (such as complex operators, arrays, and records).

Across these 194 shared puzzles, Frontline achieved significantly faster generation and processing times (28 minutes and 44 seconds compared to 95 minutes and 42 seconds for Rings). Furthermore, to ensure an unbiased comparison for time-dependent GWAP metrics (Table 4), we evaluated the subset of 99 mutually solved puzzles—representing paths successfully completed by human players in both environments. This baseline confirms Frontline's efficiency gains while maintaining strict experimental consistency.

Table 4. GWAP Metrics Comparison between Frontline and Rings

Game	Throughput (problems/hr)	ALP (hours)	Expected Contribution
Rings	62.26	0.063	3.92
Frontline (Proposed)	206.25	0.032	6.6

User questionnaires confirmed these findings, with over 50% of players finding Frontline enjoyable, easy to interact with, and preferable to play on a PC platform.

٤.3.2. Game-Based vs. Manual Generation

The comparison between the crowd-sourced game approach and manual generation by programmers yielded several key findings:

- **Path Coverage and Speed:** Players outperformed programmers in speed, spending between 14 to 28 minutes, whereas manual generation took 32 to 75 minutes. The game successfully achieved 100% coverage of all feasible paths.
- **Economic and Scalability Factors:** Player participation voluntarily without financial compensation, highlighting the cost-effectiveness of crowdsourcing compared to hiring expert programmers. Additionally, the player base is scalable and available 24/7.
- **Motivation:** Manual test data generation is a repetitive task that quickly diminishes programmer motivation. Conversely, the gamified nature of Frontline sustained participant engagement over longer periods without inducing fatigue.

5. CONCLUSION

This paper presented a novel GWAP designed to facilitate software test data generation, transforming a traditionally tedious, manual process into an engaging, crowdsourced 3D game. By systematically evaluating the framework against empirical benchmarks, this study directly addresses the core research questions:

1. **Complex Input Modeling (RQ1):** By mapping composite parameters (arrays and records) to avatar configurations and Zero-One-More (ZOM) spatial mechanics, non-expert players successfully generated valid test inputs, reaching 100% coverage across all 279 feasible execution paths without requiring programming expertise.
2. **Execution Mapping and Efficiency (RQ2):** The proposed 3D spatial mapping significantly accelerated test data generation. Frontline achieved an average generation throughput of 206.25 puzzles per hour (with an expected contribution of 6.6) and resolved the benchmark suite in 28 minutes and 44 seconds. This substantially outperformed earlier gamified tools such as *Rings* (62.26 puzzles per hour, expected contribution of 3.92, and 95 minutes and 42 seconds) and reduced completion time compared to manual generation by experienced programmers (32–75 minutes).
3. **Player Engagement and Game Mechanics (RQ3):** Post-study evaluations confirmed that over 50% of participants reported high immersion and satisfying gameplay flow, demonstrating that interactive 3D game mechanics effectively motivate and sustain non-expert crowd participation in software testing tasks.

Despite these promising results, the current implementation has certain technical constraints. As detailed in Section 3.1, to prevent visual clutter in the 3D puzzle interface, array inputs are restricted to a maximum of five elements ($N \leq 5$) with pre-defined numerical ranges. Additionally, while the ZOM pattern bounds the size of individual arrays, managing the combinatorial growth (3^k) in functions with multiple simultaneous arrays remains an open challenge; future work will investigate optimization strategies and input prioritization techniques to scale the framework to multi-array inputs without increasing player cognitive load. Regarding the mapping logic, the current mapping framework does not yet support complex pointer arithmetic or nested memory references. It is also optimized primarily for linear and polynomial constraints, meaning it may struggle with highly non-linear trigonometric functions or specific floating-point precision requirements. To address these architectural boundaries and enhance usability, future work will focus on expanding the game's logic engine to handle more complex software constructs, alongside integrating intelligent performance analysis to provide personalized, step-by-step guidance. Additionally, we plan to implement mechanisms for saving partial solutions, upgrade the tutorial system with interactive animations, and deploy the game on a web-based platform to reach a broader demographic and ensure long-term player engagement.

6. REFERENCES

[1] Ammann, P, and Offutt, J. Introduction to software testing. Cambridge University Press, 2016.
[2] Amiri-Chimeh, Saeed, Hassan Haghighi, Mojtaba Vahidi-Asl, Kamyar Setayesh-Ghajar, and Farshad Gholami-Ghavamabad. "Rings: A game with a purpose for test data generation." Interacting with Computers 30, no. 1 (2018): 1-30.
[3] Moosavi, Sharmin, Mojtaba Vahidi-Asl, and Hassan Haghighi. "Test Data Generation for Program Units Using a Game with a Purpose." Scientia Iranica (2023).
[4] Kshirasagar, K, and Tripathy, P. Software testing and quality assurance: theory and practice. John Wiley & Sons, 2011.
[5] Makoto, M, Teraguchi, M, and Inoue, K. "Effective testing and debugging methods and its supporting system with program deltas." Principles of Software Evolution, 2000. Proceedings. International Symposium on. IEEE, 2000.
[٦] J. W. Ralf Dörner, Stefan Göbel, Wolfgang Effelsberg, Ed., Serious Games, Foundations, Concepts and Practice. Cham, Switzerland: Springer, 2016. doi: https://doi.org/10.1007/978-3-319-40612-1.
[7] Sheth, S., Bell, J., and Kaiser, G. (2011) Halo (highly addictive, socially optimized) software engineering. In Proc. 1st Int. Workshop on Games and Software Engineering, pp. 29–32.ACM.
[8] Graham, D., Veenendaal van, E., Evans, I., & Black, R. (1994). Foundations of software testing; ISTQB Certification. In ACM SIGSOFT Software Engineering Notes (Vol. 19, Issue 5, pp. 128–139).
[9] Moosavi, S., Haghighi, H., Sahabi, H., Vatanzade, F., and Vahidi Asl, M. "Greenify: A Game with the Purpose of Test Data Generation for Unit Testing.", In Fundamentals of Software Engineering: 8th International Conference, FSEN 2019, Tehran, Iran, Springer International Publishing, pp. 77-92 (2019).
[10] Singh, Sanjay Kumar, and Amarjeet Singh. "Software testing". Va.ndana Publications, 2012.
[11] A. Joubert, N. Le Brun, and M. Lafourcade, Eds., Games with a Purpose (GWAPs). London: Wiley-ISTE, 2017.
[12] Tsong Yueh, and et al. "Adaptive random testing: The art of test case diversity." Journal of Systems and Software 83, no. 1, PP: 60-66, 2010.
[13] Wightman, Doug. "Crowdsourcing human-based computation." Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries. ACM, 2010.
[14] Werbach, Kevin. "(Re) defining gamification: A process approach." In Persuasive Technology: 9th International Conference, PERSUASIVE 2014, Padua, Italy, May 21-23, 2014. Proceedings 9, pp. 266-272. Springer International Publishing, 2014.
[15] Sen, K. "Concolic Testing," In Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering, Atlanta, Georgia, USA , pp. 571-572, 2007.
[16] Srivastava, P.R, Baby, K, "Automated software testing using metahurestic technique based on an ant colony optimization", in: Proceedings of 2010 International Symposium on Electronic System Design (ISED'10), pp: 235–240, 2010.
[17] Sen, K. "Cute: A concolic unit testing engine for c and java." Web, 2007.
[18] Li, H. Lam, C.P, "An ant colony optimization approach to test sequence generation for state-based software testing", in: Proceedings of the Fifth International Conference on Quality Software (QSIC'05) pp: 255–264, 2005.
[19] Bouchachia, A, Mittermeir, R, Sielecky, P, and et al, "Nature-inspired techniques for conformance testing of object-oriented software", Appl. Soft Comput. 10 (3) pp: 730–745, 2010.
[20] Tracey, N, Clark, J, Mander, K, McDermid, J, "An automated framework for structural test-data generation", In Proceedings of the Thirteenth International Conference on Automated Software Engineering (ASE'98) pp: 285–288, 1998.
[21] Mao, C, Xiao, L, Yu, X, and et al, "Adapting ant colony optimization to generate test data for software structural testing". Swarm and Evolutionary Computation, 20, pp:23-36, 2015.
[22] Harman, M, McMinn, P, "A theoretical and empirical study of search-based testin local, global, and hybrid search", IEEE Trans. Softw. Eng. 36 (2) pp: 226–247, 2010.
[23] Fraser, G, Arcuri, A, "Whole test suite generation", IEEE Trans. Softw. Eng. 39 (2), pp: 276-29, 2013.
[24] F. Taghizadeh Kafshgar Kolai, Sh. Seyfi Elahi, and M. Vahidi Asl, "Design and development of a serious game for identifying program invariants," in Proc. 7th Int. Conf. on Computer Games; Opportunities and Challenges, Isfahan, Iran, 2021.
[25] Logas, Heather, Jim Whitehead, Michael Mateas, Richard Vallejos, Lauren Scott, Daniel G. Shapiro, John T. Murray et al. "Software verification games: Designing Xylem, The Code of Plants." In FDG. 2014.
[26] E. O. Navarro and A. van der Hoek, "Simse: An interactive simulation game for software engineering education," vol.2, no.2–3, pp.1–6, 2018.
[27] J. S. B. Michael A. Miljanovic, "Robot on: A serious game for improving programming comprehension," International Workshop onGames and SoftwareEngineering, vol.2, no.2, pp.1–4, 2016.

AUTHOR BIOGRAPHY



Nima Salami is currently pursuing his Ph.D. degree in Software Engineering at the Faculty of Computer Science and Engineering, Shahid Beheshti University, Tehran, Iran. His research interests include software testing methodologies, automated testing, test optimization, software quality assurance, and serious games.
Faculty of Computer Science and Engineering Shahid Beheshti University of Technology Tehran, Iran, n_salami@sbu.ac.ir



Hassan Haghighi received his PhD degree in Computer Engineering-software from Sharif University of Technology in 2009 and is currently a Professor in the Faculty of Computer Science and Engineering at Shahid Beheshti University, Tehran, Iran. His main research interests include software testing and data management.
Faculty of Computer Science and Engineering Shahid Beheshti University of Technology
Tehran, Iran, h_haghighi@sbu.ac.ir