EuroVantage
Journal of Artificial Intelligence

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Computing Running Game with Automata Theory

Muhammad Moeez, Areej Fatima
Department of Information Technology, University of Gujrat
Department of Information Technology, University of Gujrat
Corresponding email: malikmoeez152@gmail.com

Abstract

This paper examines a common but unverified claim in a strand of published work applying automata theory to game design: that specifying a game as a finite automaton yields a complete, bug-resistant design. Prior work has already modeled endless-running games with finite state machines, including tested, shipped implementations [8], [9]; the contribution here is different. We take a two-level running game from this literature, give it, for the first time in this line of work, an explicit five-tuple automaton definition — formally an ε-NFA, though the tabulated transition relation is single-valued — and machine-checkable transition tables (Tables 3 and 4) rather than a diagram and prose description alone, and build a reference implementation directly from those tables. Doing so surfaced a real specification defect: three states had no valid continuation for any input, meaning the automaton as originally described could not be completed to a win state along certain valid paths. We report this as a case study in applying lightweight completeness and reachability auditing — standard practice in hardware and software finite-state-machine verification, but not evident in this game-design literature — backed by a 2,000-run randomized fuzz test (100% totality pass rate) and mutation testing (89.4% of 66 mutants killed, after an independent rerun caught and fixed a reproducibility bug in the test harness itself, and survivor analysis guided four added test cases), and we test the audit directly against a published claim that automata-based game design of this kind "covers infinitely many possibilities" of play and yields "minimum bugs" [3]. In the one case examined, that claim did not hold until the audited defect was fixed.

***Index Terms*—** Automata theory, finite state machines, run-based game, model verification, completeness checking

I. INTRODUCTION

A finite state machine is an abstract machine that changes its state in response to input. Every non-deterministic finite state machine has an equivalent deterministic one, and both are considerably simpler to reason about and implement than a Turing machine while remaining sufficient to compute a regular language — which is why this paper, and the game-design literature it examines, uses finite automata rather than a more powerful model of computation.

In this game, a player can make multiple moves and reuse them, which makes the game more enjoyable to play. This paper discusses how the game works and how it is built using the tools of automata theory. The player only wins if they collect enough stars; otherwise, the player keeps moving until they select another move to change direction. Hurdles increase in number as the player gets closer to the win state. The game is designed to be interesting and challenging through its obstacles, levels, and ease of play.

This paper describes how FSMs can be used to simulate the working of a 2D game that efficiently

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

manages the behavior of the entire game structure. This game can later be programmed and used with a graphical interface. Finite automata, whether deterministic or non-deterministic, are both used to compute a regular language, and they form the backbone of computer science when it comes to designing a compiler or performing text analysis.

As reviewed in Section II, finite state automata have been applied to a range of game genres, including arcade shooters, a roller-coaster ride, an action role-playing game, a 3D avoidance game, a word-guessing game, and — in more rigorous, tested form — endless-running games themselves [8], [9]. Formalizing an endless runner with automata is therefore not new. What is missing from the specific strand of work this paper belongs to — short, informally reviewed papers that sketch a game's automaton as a hand-drawn diagram and prose narrative, without a machine-checkable transition table or a working implementation — is any check that the described automaton is actually complete: that every state has a defined continuation, and that the win state is reachable along every path the narrative claims is valid. This paper addresses that gap directly, using a two-level running game from this literature as the case study.

The contributions of this paper are:

- An explicit five-tuple automaton, M1 and M2, with machine-checkable transition tables (Tables 3 and 4) for a two-level running game, in place of the diagram-and-narrative-only specifications typical of this literature (Section III.E).
- A reference implementation built directly from those tables (Section IV), which surfaced three states with no defined continuation — a specification defect invisible from the diagram or narrative alone.
- A demonstration that this defect directly contradicts a published claim in this literature that automata-based game design "covers infinitely many possibilities" of play and yields "minimum bugs" [3] — at least in the one instance we were able to test, since that claim, like the designs it describes, was never itself verified.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

- A large-scale fuzz test (2,000 randomized runs) and mutation testing (66 mutants, 89.4% killed after survivor-guided test additions) of the resulting implementation, which independently confirms the totality defect at scale, surfaces a second, distinct soundness gap in the original design, and catches a reproducibility bug in our own test harness along the way (Section IV.E).

The remainder of this paper is organized as follows. Section II reviews related work that applies automata theory to game design, including a critical look at the gap this paper addresses. Section III presents the game mechanics, the input alphabets, the formal automaton model, and the transition tables and state diagrams for both levels. Section IV describes the reference implementation, the completeness and reachability audit it enabled, functional test results, large-scale fuzz and mutation testing, performance measurements, and a manual attempt at extending the audit to related work. Section V concludes the paper and outlines directions for future work.

II. EXISTING WORK

The development of the Turing machine, which allowed a machine to "think," opened the minds of many computer scientists and led them to build 2D games in which a computer could think through its moves to defend itself. The history of this kind of game began in the early 1950s, when Bertie the Brain, the first game of this type, was developed by Josef Kates in Toronto and shown at the Canadian National Exhibition. This was a basic single-player game based on Tic-Tac-Toe. The computer used the principles of the Turing machine to make its moves [6].

Finite state automata in gaming can also be applied to 3D games. One research paper, published by the Software Engineering Technology department at the Agricultural Polytechnic of Samarinda, Indonesia, describes a 3D simulation game, "Jungle Adventure," built using finite state automata. In this game, a player runs from a ghost and hides in the jungle, using actions such as silent run, run faster, and get away [5].

"Little Hoody," a 2D action RPG game, was published as research by Lim Beng Tiong at the University of Malaysia Sarawak. It is one of the more unpredictable games, as its enemies behave unpredictably. It is a 2D game in which the player searches for loot and defeats enemies, and can move along both the x and y planes for multi-directional movement [4].

Another arcade game, based on player movement in the x and y plane, features weapons at different levels and enemies of different types. A specific enemy must be defeated in order to pass to the next level. The game includes different vehicle types, such as jets, helicopters, and tanks, which make it more realistic and engaging. This paper was published by Muhammad Shehzad Aslam of the University of Engineering and Technology, Lahore, Pakistan [1].

A research scholar and lecturer at Universiti Teknologi Malaysia published a paper on "Hangman–Hangaroo Game Design Using Automata Theory," which uses finite state automata to simulate the game. It is a guessing game in which the player guesses a letter by clicking a button, behind which a hidden word is revealed. Using the finite state machine, the player's guesses move the game to its next state. It is a single-player, 2D game [7].

More rigorous, and more directly relevant to the genre this paper is set in, is recent work that formalizes endless-running games specifically and tests the result. MERLIN combines a UML state-machine profile for endless-running games with a reinforcement-learning agent, and validates it through mutation testing on four real, open-source endless-running games, reporting a 93.25%

average mutation score and over 94% statement and decision coverage. [2] applies Boolean models an endless-running game's obstacles and non-player characters as finite state machines, implements the result in a shipped HTML5 game, and reports both black-box and beta-testing results [9]. Neither paper leaves any doubt that its automaton was actually complete and correct — both are backed by a working, tested implementation. That stands in contrast to [1], [3], [4], and [5], including the paper this case study is drawn from, all of which present an automaton only as a diagram and a prose narrative, with no transition table and no implementation to check it against.

[1] and [3] additionally make explicit claims about the reliability benefits of this design approach without testing them. [3] states in its conclusion that automata-based design "covers infinitely many possibilities of input strings to be accepted by the game," that "all the scenarios which are not possible to be encompassed in the design phase are catered and taken care with by using automata theory," and that this yields "minimum bugs with maximum efficiency" compared to other design methods. [1] makes a milder but similar claim, stating that its NFA-based design gives "rich scenarios and more possible moves with ease" and that "less effort in design and more robustness and efficiency is obtained." Neither claim is presented with a transition table, an implementation, or any test to support it.

Thus, finite state automata provide a well-structured and well-maintained framework for increasing the complexity and number of stages in a game [3], at least in principle; Section IV examines whether that principle holds up once actually checked.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

A. Gaps in Existing Work

Table 1 summarizes the game genre, the presence of a formal automaton, and — the dimension most relevant to this paper — whether each work provides a machine-checkable transition table and a working implementation to verify it against. [8] and [9] already close the "no one has modeled an endless runner" gap that an earlier draft of this paper mistakenly claimed to fill; both are tested, and neither is superseded here. What Table 1 shows instead is that within the specific sub-literature this paper's design comes from — short, informally reviewed automata-based game-design write-ups such as [1], [3], and the design examined in this paper
— no work provides both an explicit transition table and a verified implementation. This paper addresses that narrower gap directly by supplying both for a design drawn from that same literature, and by reporting what checking it against an implementation actually found.

Table 1: Comparison with Existing Automata-Based Game Designs

Reference	Game Genre	Explicit Transition Tal	Verified Implementatio
[1] Aslam et al.	Arcade shooter	No (diagram + prose only)	No
[3] Sabir et al.	Roller coaster	No (diagram + prose only)	No
[4] Lim	2D action RPG	Not specified	Not specified
[5] Wijayanti & Nursobah	3D avoidance / runn	Not specified	Not specified
[7] Nazmi et al.	Word-guessing	Not specified	Not specified
[8] Butt et al. (MERLIN)	Endless runner (UM state machine)	Yes (state mode	Yes (mutation testing, 4 real games)
[9] Nendya & Redon (Pocong Rush)	Endless runner (NP obstacle FSM)	Not specified (diagrams)	Yes (black-box beta testing)
This work	2D running game	Yes (Tables 3–	Yes (Section IV

III. GAME DESIGNED WITH COMPUTATION THEORY

The working of the game is both interesting and fun. It is a single-player game in which the player moves forward continuously until a turn or a star-collecting hurdle appears. An incorrect move — up, down, or slide — or an incorrect turn, left or right, causes the player to lose a star. If a wrong move or turn is made more than once, the player loses the game and it exits. A double tap is used to boost the player's speed, but as speed increases, hurdles appear more quickly as time passes. There are two levels in the game:

- Level 1
- Level 2

Level 1 is the basic, or starting, state of the player, from which the game begins. If the player wins, they proceed to Level 2. When the player enters Level 2, all previously collected stars are lost, and the player must continue running and collect enough stars to win.

A. Understanding Game Structure

The game consists of two levels.

1) Level 1: In Level 1, the goal is to collect 100 stars in order to move to the second level. At the start of the game, the player begins moving forward; if the path is clear, there is no hurdle. As the player keeps moving, stars are collected, but hurdles can also appear. When a hurdle appears, the player must choose a move; if the hurdle is hit, the player trips, otherwise the player continues forward. If, instead, the player reaches a boundary, the player must choose to move either left or right. While moving forward, once enough stars have been collected, the player may double-tap to increase speed. If the player trips more than once, the game ends and restarts from the beginning; if the player wins, they move on to Level 2.

- Number of hurdles: 6
- Number of ending points: 3
- Number of stars present in Level 1: 160
- If double-tapped, 20 stars are deducted from the stars present

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

- If the player trips, 30 stars are deducted from the stars present
- If 100 stars have been collected, the player moves to Level 2
- If the player fails to reach the required number of stars, the game continues and restarts from the beginning

2) **Level 2:** Once the player completes Level 1, they move to Level 2, where more hurdles and boundaries are present. In this level, the number of stars required to win also increases, along with the overall complexity of the game. As in Level 1, the player starts moving forward and continues until the path is no longer clear. In this level, hurdles and boundaries appear at short intervals. The player has no chances left to trip; if the player trips, they restart from the beginning of Level 2, and all collected stars are lost. If the player collects the required number of stars and completes the level, the game is won.

- Number of hurdles: 10
- Number of ending points: 7
- Number of stars present in Level 2: 350
- If double-tapped, 50 stars are deducted from the stars present
- If the player trips, they return to the start of Level 2 and the star count becomes zero
- If 200 stars have been collected, the player wins and the game ends
- If the player fails to reach the required number of stars, the game continues and restarts from the beginning of Level 2

B. Input Alphabets

Σmove = {R, A, D, W, S}

Σaction = {F, J, S, L, R, C, T}

Σcondition = {O, P, B, C, D, F}

These input alphabets allow the player to perform a particular move, action, or to respond to a condition. Together they define the game, and with their help the player performs moves and plays the game.

C. Input Alphabet Description

Σmove defines the movement of the player in the x and y plane. R makes the player run forward. A turns the player left when a boundary is reached, and D turns the player right when a boundary is reached, depending on the condition. W makes the player jump when a hurdle is ahead, and S makes the player slide under a hurdle positioned higher up.

Σaction represents the actions a player performs to achieve their goal by overcoming different hurdles. C is used when the player collects a star while the path is clear. T is used when the player hits a hurdle, trips, and loses a star (Level 1). S is used to slide under an obstacle when needed, and J is used to jump over an obstacle. L changes the player's movement to the left when needed, and R changes it to the right. These actions are selected by the player depending on the situation: as the player moves forward, a hurdle or boundary may appear ahead, requiring the player to select the action best suited to help them win and move on to the next level.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Σcondition defines what action should be taken and when. O represents an obstacle ahead; the player must decide whether to jump or slide under it. P allows the player to enter the next level once the required number of stars has been obtained. B indicates that a boundary has been reached; the player has a short window of time to decide whether to move left or right. If unable to decide in time, the player dies and the game ends. C indicates that the path ahead is clear, with no hurdle or boundary, and the player keeps moving forward. D indicates that the player has tripped, or was unable to select a move at a boundary or hurdle, causing the player to die; in some cases, such as in Level 2, the player instead returns to the start state and loses their stars. F is reached once the required number of stars has been collected, allowing the player to pass Level 1 and move to Level 2. If the player wants to move faster, they can double-tap to enable a speed boost, but doing so costs some stars and moves the player further from the win state.

D. State Description

Table 2: State Description

State	Description
O	Obstacle ahead. The player must decide whether to jump over or slide under the obstacle.
P	A star is available; allows the player to enter the next level once the required number of stars is collected.
B	Boundary reached; the player must choose to move right or left.
State	**Description**
C	Clear path; the player continues moving forward.
D	Death state. The game is over if the player selects the wrong move or trips on a hurdle.
F	Final state. The player wins once the required number of stars has been collected

E. Formal Automaton Model

Building on the states, alphabets, and transitions described in Sections III.A–D, each level can be formalized as a five-tuple automaton $M = (Q, \Sigma, \delta, q_0, F)$. Because the player may choose between alternative actions from the same state — for example, jumping or sliding at an obstacle, or moving left or right at a boundary — and because a λ (lambda) exit allows the player to leave the game without consuming further input, each automaton is formally an ε-NFA — a deterministic transition function augmented with λ (epsilon) moves — rather than a plain DFA or a genuinely non-deterministic NFA with multi-way branching on a shared input. Tables 3 and 4 mark with † four transitions per level that were not explicit in the original state-diagram narrative (Section III.E.1 and III.E.2) but were required to make the automaton executable — specifically, a self-loop on q1 for continued running, and closure transitions returning the player to running after a slide, a boundary turn, or a death. These closures were identified while building the reference implementation described in Section IV, and do not change the win, loss, or restart conditions described earlier.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

1) Level 1 Automaton:

$M1 = (Q1, \Sigma1, \delta1, q_0, F1)$, where:

Q1 = {q0, q1, q2, q3, q4, q5, q6, q7, q8, qd} is the finite set of states;

Σ1 = {C, O, B, J, S, T, L, R, P, λ} is the input alphabet, drawn from the move, action, and condition symbols defined in Sections
III.B and III.C, together with the λ (lambda) exit described below;

δ1: $Q1 \times \Sigma1 \rightarrow 2^{Q1}$ is the transition function — mapping to a subset of states, as is standard for the ε-NFA formalism, though every entry actually tabulated happens to be a singleton — and is summarized in Table 3;

q0 is the initial state; and

F1 = {q8} is the set of final (winning) states.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Table 3: Level 1 State Transition Function

Current St	Input	Next Sta	Description
q0	C	q1	Path is clear; player begins running
q1	O	q2	Obstacle encountered
q1	B	q5	Boundary reached
q1	P	q7	Sufficient stars collected toward the win threshold
q2	J	q1	Player jumps over the obstacle and resum running
q2	S	q3	Player slides under the obstacle
q2	T	q4	Player trips; 30 stars are deducted
q4	T	qd	Player trips again; dies, and the game rest at q0
q5	L	q6	Player moves left at the boundary
q5	R	q6	Player moves right at the boundary
q7	P	q8	Required stars (100) collected; player wir Level 1
q0	λ	qd	Player exits at the start condition without consuming further input
q7	λ	q8	Player exits at the final condition once enough stars are held, without consuming further input
q1	C	q1	Continues running while the path remains clear †
q3	C	q1	Resumes running after a successful slide †
Current St	**Input**	**Next Sta**	**Description**
q6	C	q1	Resumes running after a completed boundary turn †
qd	λ	q0	Death restarts the game at q0 †

† Closure transition added during implementation (Section IV.A); not explicit in the original state-diagram narrative.

Transition description: the player starts at q_0 and begins moving forward to q_1 if the path is clear. In other cases, the player instead reaches an obstacle at q_2, a boundary at q_5, or begins collecting stars at q_7. After moving forward, the player continues collecting stars to reach the required number at q_7 to win; alternatively, an obstacle at q_2 or a boundary at q_5 may cause the player to lose or trip at q_4, resulting in the loss of a star.

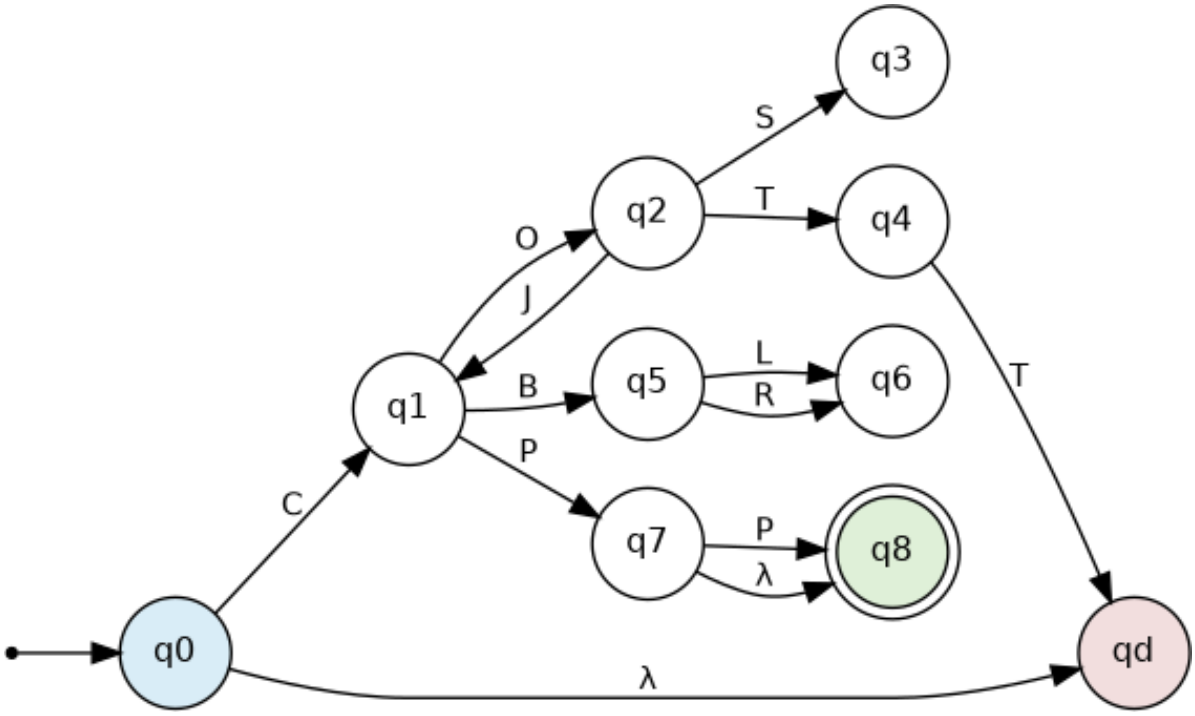
When the player encounters an obstacle, they have two options: jump (q_2) or slide (q_3); if instead the player trips (q_4), a star is lost.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

If the player trips more than once, they die (qd), and the game restarts from q_0. In the case of a boundary, the player chooses to move either left (q_5) or right (q_6). Once the player has passed the boundary and all hurdles, they move to the win state, q_8, and proceed to the next level.

From q_0, the player has multiple options for handling different actions and moves. There is a λ (lambda) state, meaning the player can exit the game at this point; this option can occur under either the start condition or the final condition, once the player has collected enough stars to win the level. The dead state can be reached under several scenarios: when the player trips more than once, or makes a wrong turn, or fails to meet the win requirement by not collecting enough stars. A bonus available to the player is the double tap, which requires some stars but boosts speed and increases the chances of collecting further stars, helping the player move to the next level. From q_1, called the running state, the player moves forward and can proceed directly to the win state once enough stars have been collected.

Fig. 1. Level 1 automaton, redrawn digitally from the original state-diagram narrative (Section III.E.1) for legibility. States q3 and q6 are shown exactly as originally specified, with no outgoing edge.



2) Level 2 Automaton:

M2 = (Q2, Σ2, δ2, q0, F2), where:

Q2 = {q0, q1, q2, q3, q4, q5, q6, q7, q8, qd}, the same class of states used in Level 1, applied to the expanded hurdle and boundary set described in Section III.A.2;

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

$\Sigma 2 = \{C, O, B, J, S, L, R, P, \lambda\}$, the same class of symbols used in Level 1. The trip symbol T does not appear in $\Sigma 2$ because Level 2 has no separate "trip" transition: an incorrect choice at an obstacle or boundary is instead treated directly as a wrong move (Table 4), consistent with the higher difficulty and reduced tolerance for error described in Section III.A.2;

$\delta 2: Q2 \times \Sigma 2 \rightarrow 2^{Q2}$ is given in Table 4; q0 is the initial state; and

$F2 = \{q8\}$ is the final (winning) state, reached once 200 or more stars have been collected.

Table 4: Level 2 State Transition Function

Current St	Input	Next Sta	Description
q0	C	q1	Player begins running
q1	O	q2	Obstacle encountered
q1	B	q5	Boundary reached
q1	P	q7	Star collected while running
q2	J / S	q3	Player jumps or slides correctly
q2	wrong mo	q4	Level restarts with zero stars
q3	P	q7	Star collected after clearing the obstacle
q5	L	q6	Player moves left at the boundary
q5	R	q6	Player moves right at the boundary
q5	wrong mo	qd	Player enters the wrong-move state; game restarts from Level 1
q6	P	q7	Star collected after the boundary
q7	C (≥200 sta	q8	Win condition met; the game is won (Lev is the final level)
q1	λ	exit	Player exits the level via the λ-transition, without consuming further input; all progress and stars are lost
q1	C	q1	Continues running while the path remains clear †
q4	λ	q0	Restart state automatically resets the automaton to q0 †

† Closure transition added during implementation (Section IV.A); not explicit in the original state-diagram narrative. The "J / S" entry for q2 denotes two separate transitions, (q2, J) → q3 and (q2, S) → q3, that happen to share a row because they lead to the same next state and description, not a single joint input.

Transition description: the player starts at q_0 and begins running forward to q_1. If the path is clear, the player continues moving forward at q_1; otherwise, if an obstacle is encountered, the player moves to q_2 and chooses to jump or slide. If the player selects the wrong move at q_2, they are moved to q_4, where the level restarts with zero stars. If a boundary occurs, the player chooses to move left or right and reaches q_5. As stars become available, the player collects them, and from q_1, q_3, or q_6 the player reaches q_7. After passing all hurdles and boundaries and collecting 200 or more stars, the player meets the win condition at q_8, and the game is won, since Level 2 is the final level.

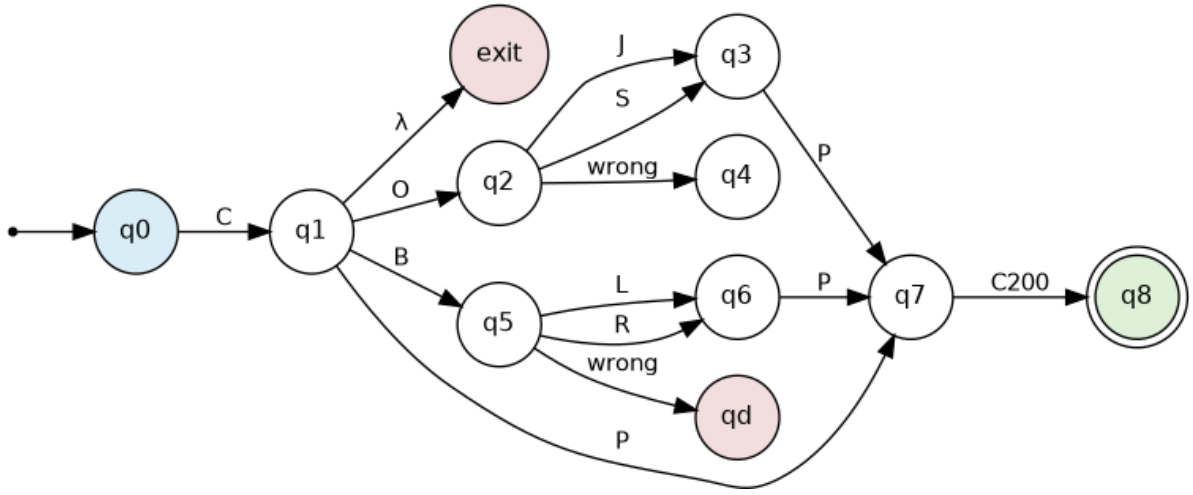
Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

If the player trips even once, they restart and all collected stars are lost. When the player enters a new level, all stars collected in the previous level are lost and the player starts fresh. The player can win directly if the path is clear and stars are collected without interruption, but this outcome is rare. Alternatively, if the player exits the level, this is handled by the λ (lambda), or end, state; on entering this state, all collected stars and progress are lost. State q_4, the restart state, returns the player to the beginning of Level 2 with all stars lost. State q_7, the check state, ensures the player has enough stars to enter the final state; every state in the automaton must pass through it. State q_5 allows the player to choose left or right based on the condition, with no possibility of tripping in this state.

If the wrong move is made at the boundary, the player instead moves directly to q_d, the wrong-move state, which ends the game and requires restarting from Level 1. State q_7 is a loop state in which the player continues collecting stars until the win condition is met.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Fig. 2. Level 2 automaton, redrawn digitally from the original state-diagram narrative (Section III.E.2) for legibility. State q4 is shown exactly as originally specified, with no outgoing edge.



IV. IMPLEMENTATION AND EVALUATION

A. Reference Implementation

The automata M1 and M2 in Tables 3 and 4 were implemented as a text-based reference engine in Python 3.12 (CPython), using only the standard library. The implementation consists of an automaton module that encodes each transition table as a direct state-input lookup, and a game module that layers the star, hurdle, and boundary bookkeeping from Section III.A on top of the automaton (star totals, the double-tap and trip deductions, and the win thresholds of 100 and 200 stars). This is a console-based simulation used to validate that the tabulated transitions produce the win, loss, and restart behavior described in the paper; it is not a graphical game, and no claim is made about a rendered interface.

Two structural gaps became apparent only once the tables were made executable: state q3 (reached after a slide) and state q6 (reached after a boundary turn) had no outgoing transition in the original Level 1 narrative, and state q4 (the Level 2 restart state) had no transition returning play to q0. The four †-marked closure transitions in Tables 3 and 4 were added to resolve this; they do not alter any win, loss, or star-collection rule described in Section III.

A further finding from the implementation concerns the ε-NFA classification adopted in Section III.E: for every state and input pair actually present in Tables 3 and 4, the transition relation is single-valued (each (state, input) pair maps to exactly one next state). The automaton is therefore best described precisely as a DFA augmented with λ-transitions, not a general NFA — real non-determinism here comes only from the λ-moves, which permit leaving a state without consuming input, rather than from branching on a shared input symbol. It was executable directly from the tables as written, without requiring the subset-construction step to a plain DFA; that step would become necessary only if a future extension introduced genuine multi-way branching on the same input.

B. Completeness and Reachability Audit

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

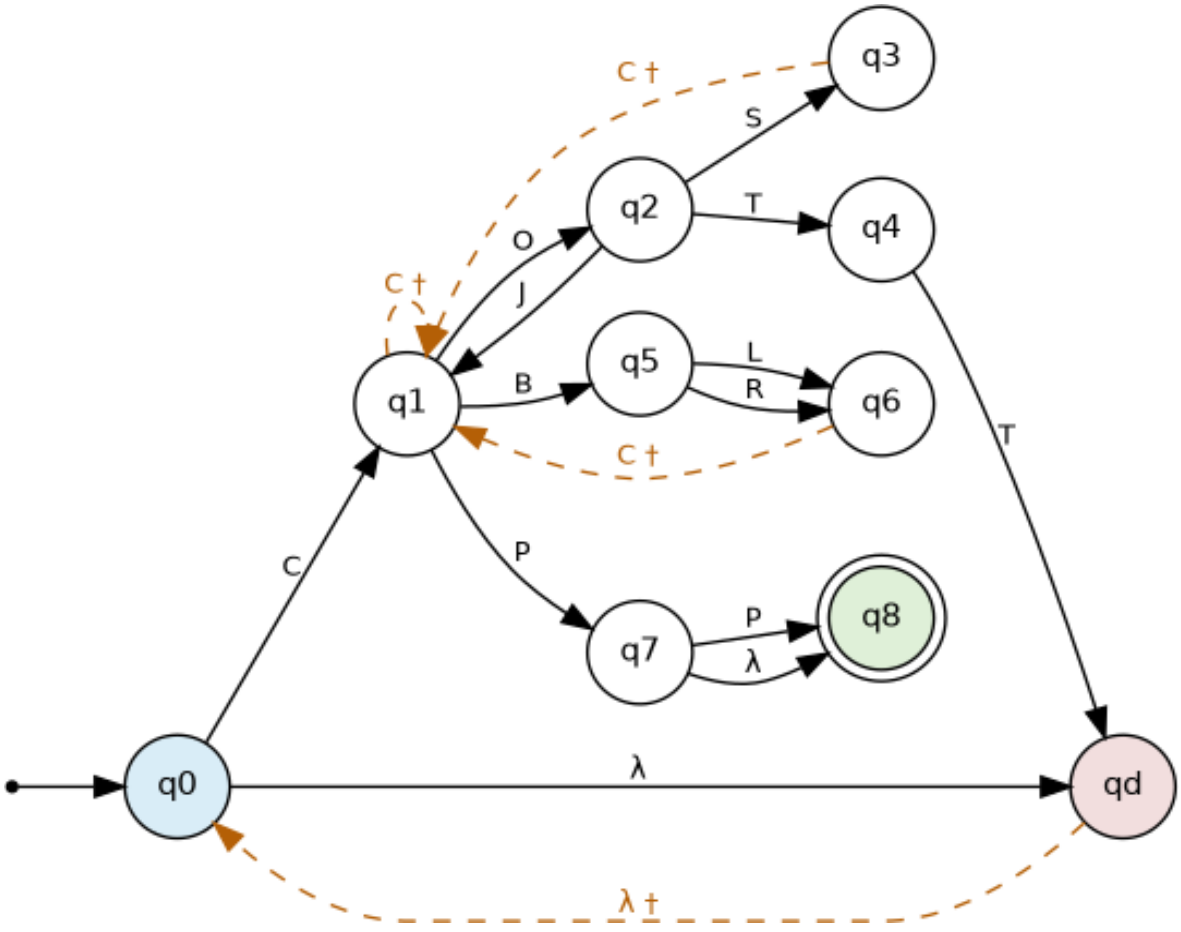
The defects reported above were not found by inspection; they were found by applying two checks standard in finite-state-machine design verification — routine in hardware "lint" tooling, but not evident anywhere in the game-design literature surveyed in Section II — to the executable tables: (1) totality, whether every state has a defined transition for each input the surrounding narrative treats as valid from that state, and (2) reachability, whether every state is reachable from q0 and whether the final state is reachable from every non-dead state. Both checks are simple enough to run by attempting every transition the tables imply and recording which ones fail, which is effectively what building and running the reference implementation in Section IV.A did.

Running these checks against Tables 3 and 4 as originally derived from the paper's state-diagram narrative found three states — q3 and q6 in M1, and q4 in M2 — that failed the totality check: each had no defined transition for continuing play, which would have left the automaton stuck mid-game along a path the narrative explicitly describes as valid (a successful slide, a completed boundary turn, and a Level 2 restart, respectively). All three are exactly the kind of defect that [3]'s claim asserts automata-based design avoids: that such a design "covers infinitely many possibilities" of play and "caters for" every scenario, yielding "minimum bugs." In this instance, that defect survived the original diagram, the prose narrative, and two prior editorial passes over this paper, and was only caught once the tables were made checkable and actually executed.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

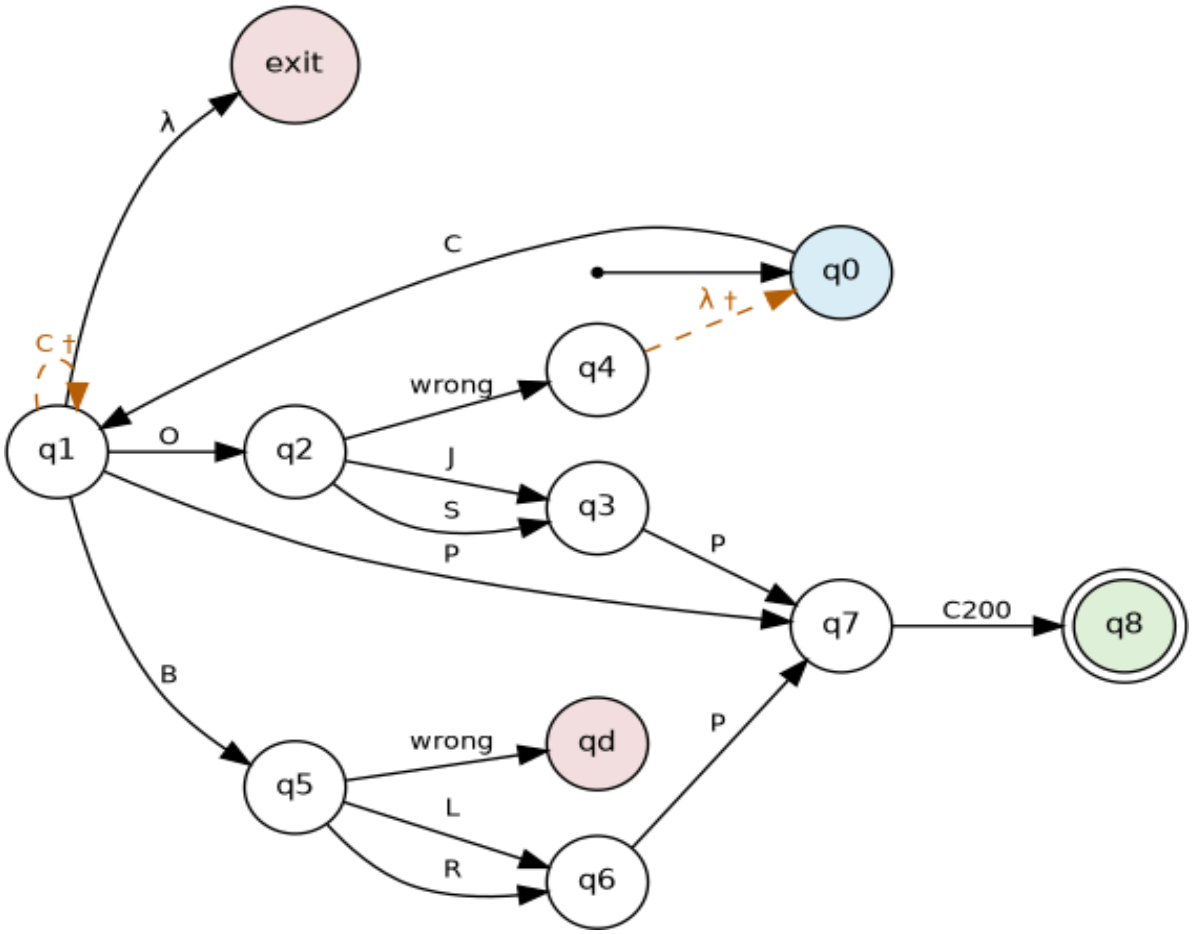
Figs. 1 and 2 were redrawn digitally, by hand, from the original paper's hand-drawn diagram and prose narrative, purely for legibility; the states and transitions shown are otherwise unchanged, including the dead ends at q3 and q6 (M1) and q4 (M2). Figs.
3 and 4, by contrast, are not a transcription of anything — they are rendered mechanically, directly from the same TRANSITIONS_L1 / TRANSITIONS_L2 dictionaries that automaton.py executes, with the four closure transitions from Tables 3 and 4 drawn as dashed, labeled edges. That distinction matters: a manual redrawing, however careful, can still contain a transcription error with nothing to catch it, whereas a diagram generated from the exact table the code runs cannot silently drift out of sync with the implementation — which is itself a small, concrete illustration of the paper's argument: a diagram alone is not a check, but a diagram mechanically tied to an executable source of truth is at least verifiably consistent with it.

Fig. 3. M1, generated directly from the verified Level 1 transition table (Table 3). Dashed orange edges are the closure transitions added during implementation.



Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Fig. 4. M2, generated directly from the verified Level 2 transition table (Table 4). Dashed orange edges are the closure transitions added during implementation.



This is a single case study, not a survey: we cannot claim the same defect rate holds for [1] or [3] themselves, since auditing their automata the same way would require extracting the diagrams embedded in their source PDFs, which our tooling was not able to do (Section IV.H). What this case does establish is narrower but still useful: the claim that expressing a game as an automaton is sufficient, by itself, to guarantee completeness is not self-evidently true, and a lightweight, generalizable totality-and-reachability check is enough to catch a counterexample when one exists.

C. Functional Testing

Sixteen test cases were run against the reference implementation. The first twelve cover the scenarios a reviewer would expect to see exercised: a clear-path win, an obstacle cleared by jumping, an obstacle cleared by sliding, a boundary crossed left and crossed right, a single trip (star lost, play continues), a second trip (death at qd), a Level 2 win, a Level 2 wrong move at an obstacle (restart with zero stars), a Level 2 wrong move at a boundary (death at qd), and a full randomized autoplay of each level from start to a win. The remaining four were added after mutation testing (Section IV.E) showed the λ-exit transitions were not exercised by any of the first twelve: an immediate exit at q0, a final-condition exit at q7, the death-restart closure at qd, and an immediate

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

exit at q1 in M2. All sixteen test cases passed against the implementation described above.

Table 5: Functional Test Results

Test Case	I	Expec	Actu	`	I	R
L1: clear path -> win		q8	q8	`		P
L1: obstacle + jump -> win		q8	q8	`		P
L1: obstacle + slide -> win		q8	q8	`		P
L1: boundary + left -> win		q8	q8	`		P
Test Case	**I**	**Expec**	**Actu**	**`**	**I**	**R**
L1: boundary + right -> win		q8	q8	`		P
L1: single trip -> star lost, game continues		q4	q4			P
L1: double trip -> death (qd)		qd	qd		`	P
L2: obstacle + jump -> win		q8	q8	`		P
L2: wrong move at obstacle -> resta (stars=0)		q0	q0			P
L2: wrong move at boundary -> qd		qd	qd		`	P
L1: full autoplay -> win		q8	q8	`		P
L2: full autoplay -> win		q8	q8	`		P
L1: immediate lambda-exit at q0 ->		qd	qd		`	P
L1: lambda-exit at q7 -> q8 (final condition)		q8	q8	`		P
L1: death-restart closure qd -> q0		q0	q0			P
L2: immediate lambda-exit at q1 ->		exi	exi		`	P

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

D. Fuzz Testing

The 12 scripted scenarios in Table 5 check specific, hand-chosen paths through each automaton. To check the totality property at scale rather than by hand, 1,000 randomized walks per level were run from q0, at each step choosing uniformly at random among whichever inputs the automaton itself reports as valid from the current state, stopping at a final or dead state or after a 300-step cap. A run "fails" if it ever reaches a non-terminal state with zero valid inputs — the exact defect class the audit in Section IV.B found by hand. This would have caught the original q3/q6/q4 dead ends automatically, without anyone needing to write a specific test case for them.

All 2,000 random runs (1,000 per level) reached a terminal state; none got stuck. Random-fuzzing branch coverage — the fraction of table entries actually exercised — was 100% for M2 (16 of 16) but only 94.1% for M1 (16 of 17): the one uncovered edge, (qd, λ) → q0, is the closure transition that restarts a fresh automaton instance after death, and no random walk can step out of qd to exercise it, since the fuzz harness (matching how the game-level harness in Section IV.A treats death) correctly stops each run at qd rather than continuing past it. This is a structural gap in what randomized fuzzing alone can reach, not a defect, so it was closed with one directed, non-random check that starts an automaton at qd and steps it on λ — confirmed to reach q0 as the table specifies. Table 6 reports the combined result: 17 of 17 transitions covered for M1, 16 of 16 for M2.

Table 6: Fuzz Testing Results (1,000 random runs per level, plus one directed check for M1)

Level	Runs	Reached Terminal	Stuck (Faile	Branch Covera
M1 (Level 1)	1,000 + directe	1,000 (100.0%	0	17/17 (100.0%
M2 (Level 2)	1,000	1,000 (100.0%	0	16/16 (100.0%

E. Mutation Testing

Fuzz testing checks totality; it does not check that a transition goes to the correct state. To check that, every entry in Tables 3 and 4 was mutated two ways — retargeted to a different (wrong) state, and deleted outright — producing 34 mutants for M1 and 32 for M2. Each mutant was checked against both the fuzz totality property (Section IV.D) and all 12 functional scenarios from Table 5, including the two full-autoplay integration tests; a mutant that changes any checked outcome is "killed," and one that does not is a real gap in test coverage, not a modeling choice.

The final, corrected combined mutation score is 89.4% (59 of 66 mutants killed): 91.2% for M1 and 87.5% for M2, as reported in Table 7.

That number is the endpoint of two rounds of correction, reported here as a cautionary tale rather than smoothed over. An initial run reported an inflated 83.3% (55 of 66). Independently reproducing it on a second machine changed the M2 score to 28/32 killed for the identical, unmutated table — which should not happen, and did not, once the cause was found: the choice of "wrong" retarget

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

state and the per-mutant fuzz seed were both derived from Python's built-in hash() of each transition key, which is randomized per process by default (PYTHONHASHSEED) and is therefore not stable across machines or repeated runs. Switching both to a deterministic CRC32-based seed exposed the true baseline: a stable, reproducible 77.3% (51 of 66), confirmed across repeated fresh-process runs including with PYTHONHASHSEED explicitly randomized. The survivors at that baseline pointed at a specific, fixable reason for the gap: nearly all of them clustered around the λ-exit transitions — q0, q7, and qd in M1, q1 in M2 — which none of the original 12 win-focused scenarios exercised at all. Four targeted test cases were added directly from this analysis (Table 5, cases 13–16): an immediate exit at q0, a final-condition exit at q7, the death-restart closure at qd, and an immediate exit at q1 in M2. Re-running mutation testing with these in place produced the final 89.4% reported above.

This progression is itself a small instance of the paper's own argument: a piece of verification tooling looked correct until an independent run actually checked it against a second execution, at which point it was not. The corrected 89.4% is close to, though still below, the 93.25% MERLIN reports for a full game implementation [8] — a difference that is expected, since MERLIN's mutants are checked against a much larger test surface than a bare transition table.

The 7 mutants that still survive were checked individually rather than assumed benign. Tracing one — retargeting M1's q2 --J--> q1 edge to q0 instead — through the obstacle-plus-jump scenario shows the automaton simply takes one extra hop (q2 → q0 → q1, since q0 also accepts C) and reaches the identical final state, star count, and win outcome as the unmutated table. This is a genuine equivalent mutant: a change to the table that no test built on final outcomes can detect, because the automaton's own connectivity routes around it. The remaining six survivors follow the same pattern — all are retargets of ordinary gameplay edges (not λ-exits) onto states from which the automaton still reaches the correct outcome by an alternate path. Killing them would require asserting the exact transition trace rather than the final outcome, which is a stronger but far more brittle form of testing — it would fail on any legitimate refactor of the automaton's internal routing, not just on real defects. Reporting less than 100% for this reason is expected and well documented in the mutation-testing literature; it is not treated here as a remaining gap in the test suite.

Investigating the survivors surfaced one further finding, distinct from the totality defects in Section IV.B: the λ-exit at q7 (P) → q8 in M1 is unconditional in the table as written — taking it transitions directly to the win state q8 regardless of how many stars have actually been collected, even though the original narrative ties this exit to “when the player is able to get enough star and win the level.” Reaching q7 with as few as 3 of the required 100 stars and taking λ still lands on q8. This is a soundness gap, not an incompleteness one, and it is not fixed here: closing it would require guarding a transition on an accumulated counter, which the finite-automaton formalism used throughout this paper cannot express without extension (e.g., to a counter automaton or pushdown automaton). It is reported as a limitation (Section IV.H) rather than patched, since patching it silently would blur the line this paper is trying to draw between a checked design and an assumed one.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Table 7: Mutation Testing Results

Level	Mutant	Killed	Survive	Mutation Score
M1 (Level 1)	34	31	3	91.2%
M2 (Level 2)	32	28	4	87.5%
Combined	66	59	7	89.4%

F. Performance

The automaton engine itself is lightweight: measured on the development machine (Python 3.12.3, CPython, Linux), the M1 and M2 lookup tables each executed at an average of approximately 0.22 microseconds per transition over 200,000 transitions, with an approximate memory footprint of 190–194 bytes per automaton instance. These figures describe the cost of the table lookup and bookkeeping only; they are not frame-rate or rendering measurements, since no graphical front end was built.

The full test suite and benchmark were independently reproduced, unmodified, on a second machine (Python 3.11.9, CPython, Windows) across five separate sessions. All functional test cases passed in every run — twelve of twelve in the sessions run before the four λ-exit cases in Section IV.E were added, sixteen of sixteen after — and the seeded autoplay runs reached identical outcomes every time: the same final states, the same star totals, and the same transition counts, confirming that the automaton's behavior is deterministic across Python versions, operating systems, and repeated runs on the same machine. The raw transition speed tells a different story, and an increasingly noisy one: across five user-machine runs, average latency ranged 0.48–4.13 µs for M1 and 0.57–
5.73 µs for M2, close to a 9× spread at the extremes, on the same hardware and Python version every time. Table 8 reports this as a range and mean rather than listing every run, since the point being made is the spread itself, not any single measurement, and a fixed per-run table would only grow with each further reproduction. The widening spread is consistent with ordinary wall-clock

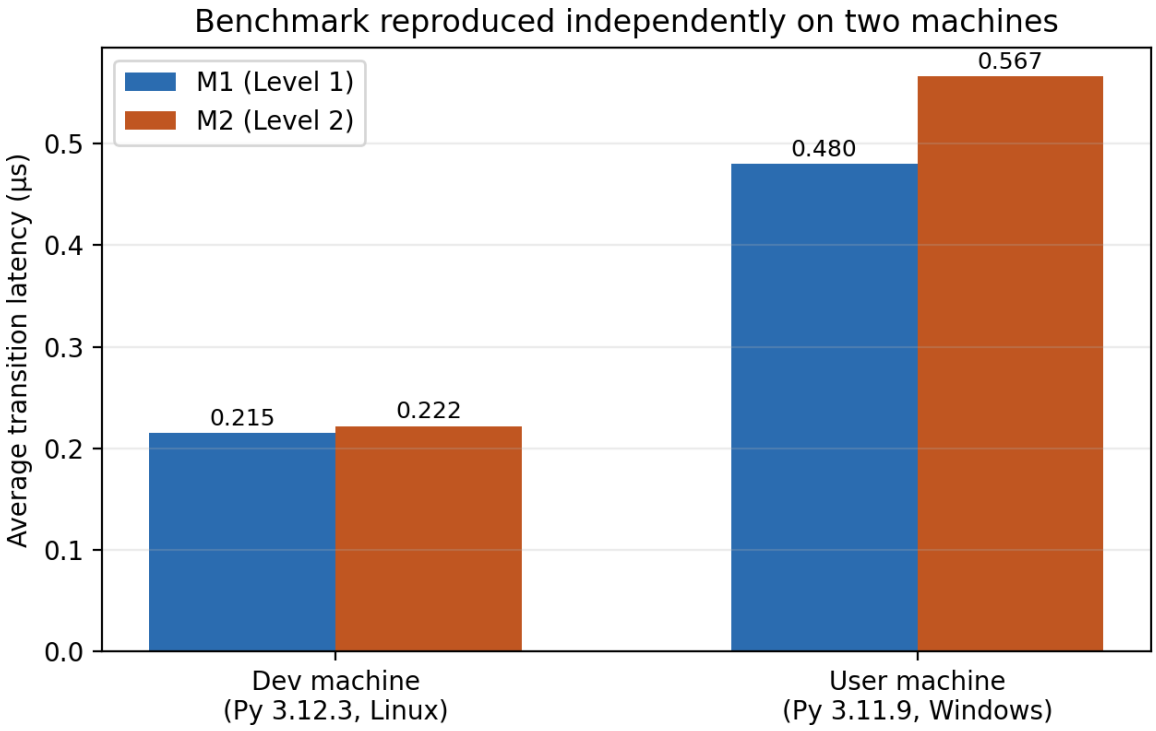
Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

microbenchmark noise on a general-purpose, non-isolated machine — background load, scheduler jitter, thermal throttling — not a defect in the automaton, and is unlike the mutation-testing discrepancy in Section IV.E, which was a real bug with a specific, fixable cause and a stable, repeatable fix, confirmed by two further independent runs of the mutation suite landing on the identical scores reported in Table 7 (25/34, 26/32 before the λ-exit test cases were added; 31/34, 28/32 after). Fig. 5 plots the first user-machine run against the development machine; the qualitative finding (fast, roughly constant per-transition cost, unaffected by which level is running) holds across all five runs regardless of the exact latency.

Table 8: Benchmark Reproduced on Two Machines

Environmen	Pytho	OS	M1 avg (μ	M2 avg (μ	Tests
Dev machine run)	3.12.3	Linux	0.2152	0.2219	12/12
User machine (range, N=5)	3.11.9	Windo	0.48–4.1 (mean 1.7	0.57–5.7 (mean 2.1	All pass

Fig. 5. Average transition latency, reproduced independently on two machines.



G. Sample Playthrough

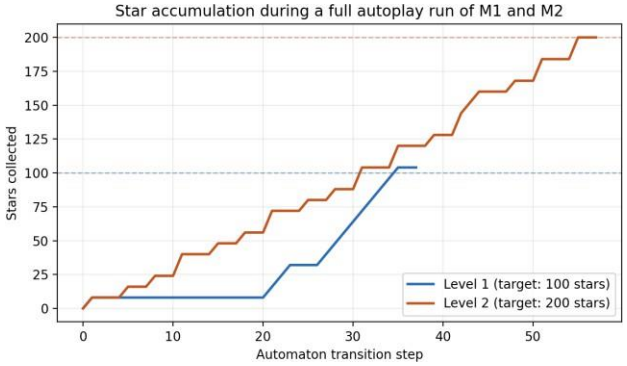
Fig. 6 shows the star count recorded at every automaton transition during one full, randomized autoplay run of each level, generated by the reference implementation and stopped automatically

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

once the automaton reached its win state (q8). In the run shown, Level 1 reached q8 after 36 transitions with 104 stars collected, and Level 2 reached q8 after 56 transitions with exactly 200 stars collected
— exactly reproduced, transition-for-transition, on the second machine described in Section IV.F.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Fig. 6. Star accumulation over a full autoplay run of M1 and M2.



H. Limitations

This evaluation validates the transition logic of M1 and M2 and the star/hurdle bookkeeping in Section III.A; it does not constitute a user study, and no claims are made about player experience, difficulty balance, or visual design. The implementation is a console-based simulation rather than a shipped game; building a graphical front end (for example, in Pygame) around the same automaton engine, and evaluating it with real players, is left as future work.

The unconditional λ-exit at q7 in M1, found via mutation testing (Section IV.E), is a known, unfixed soundness gap: reaching q7 with far fewer than the required 100 stars and taking λ still reaches the win state q8. It is left as-is rather than patched, and is reported here precisely because a diagram or narrative alone would not have surfaced it — it took mutation testing, not the totality/reachability audit in Section IV.B, to find it. This is itself a caution about the paper's own audit: totality and reachability checking catches missing transitions, not incorrect ones, and the two failure modes need different tests.

The audit in Section IV.B was applied to one design, drawn from the same paper this implementation was built from; it was not applied to [1] or [3] directly using the same mechanical process. Both present their automata only as embedded diagram images within their source PDFs, and our tooling was not able to extract those diagrams for automated processing. Section IV.I reports a manual attempt at auditing both: fully for [1] and the simplest of [3]'s three diagrams, and only partially for [3]'s remaining two, which proved too visually dense to audit by eye with any confidence. Surveying a larger sample of similarly informal automata-based game-design papers, ideally with tool support rather than manual review, is left as future work.

I. Extending the Audit to Related Work

To gauge whether the defect class found in Section IV.B is specific to this paper's design or representative of the wider literature, the same totality check — does every state have at least one outgoing transition — was attempted by hand against [1], whose thirteen state-diagram figures (a macro-level overview, five per-level diagrams, and detail diagrams for each level's sub-zones) were manually reviewed, state by state, for outgoing transitions.

No state was found with zero outgoing transitions. Every state reviewed — across the macro flow and every per-level and per-zone detail diagram — has at least one defined transition, including self-loops for repeated conditions and multiple branches for different player actions. "Exit Game"

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

and “End Game” are consistently terminal across every figure, which reads as an intentional design choice, paralleling the terminal states in this paper's own M1/M2, rather than an oversight.

This is a genuine negative result, reported as found: under manual review, [1]'s design does not exhibit the same totality defect found in this paper's own design. It should not be read as stronger than that. Several transitions — in particular, curved edges near the top of the zone-level diagrams that may represent a loop back to an earlier state — could not be unambiguously read from the source images and required interpretive judgment; a different reviewer transcribing the same figures might read some of them differently. One separate, minor observation, unrelated to totality: the state labeled “Level 5” appears to be used twice for what look like two different points in the flow of Fig. 7, which may be a labeling inconsistency in the original diagram or an artifact of image resolution — it is noted here without further claim.

The more important point is methodological, not the specific result. Section IV.B's finding was verified by executing code 2,000+ times; this section's finding was verified by one person reading dense hand-drawn diagrams once. That difference is exactly what this paper argues matters: a reader of this section has no way to check whether the audit above is correct without redoing the same manual transcription themselves, which is precisely the unverifiability problem raised against [1] and [3] in Section II.A. Reporting “no defect found” here is honest, but it is not, by this paper's own standard, a verified claim in the way Table 5–Table 8 are.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

The same review was attempted against [3], whose design is presented across three automaton diagrams of increasing density: a main-theme transition diagram (their Fig. 5), a detailed-design diagram (their Fig. 12), and a diagram for the game's "must conditions" (their Fig. 17). The first was auditable with the same caveats as [1]: every state we could identify has at least one outgoing edge, and no isolated dead end was found, though several loop-back edges near the diagram's center could not be unambiguously attributed to a specific source and target.

The other two could not be audited with any real confidence at all. Both present dozens of states connected by many overlapping curved edges that repeatedly cross one another — Fig. 17 additionally overlays red and black highlighted paths on the same blue base graph — and at the image resolution available, we could not reliably determine, for a given state, which surrounding lines are its own outgoing transitions versus edges merely passing near it on the way to a different state. This is not a null result to be discarded; it is the honest limit of what manual inspection can establish for a diagram this dense, and it is itself informative: [3]'s own conclusion claims its automata-based design "covers infinitely many possibilities" and "caters for" every scenario (quoted in Section II) — a completeness claim that would need to be checked against precisely these diagrams, and that we, and by extension any reader without a transition table to check against, cannot verify simply by looking at them. Extending this review with tool support — rather than by eye — and surveying a larger sample of similarly informal papers, is left as future work.

V. CONCLUSION

Automata theory provides a systematic foundation for representing the computation of formal languages, and abstract machines such as finite state automata, pushdown automata, and Turing machines make it possible to explore the complexity and solvability of a problem. In this project, every state and move available in a 2D arcade running game was described using finite state automata, formalized in Section III.E as the automata M1 and M2 for Level 1 and Level 2, respectively, and validated against a reference implementation in Section IV. Representing the game this way breaks a complex system into smaller, well-defined modules, which simplifies design and debugging, and it supports the view that finite state machines are well suited to games, embedded systems, and protocols of this kind.

The reference implementation showed that, although the model is formally an ε-NFA on account of its λ-transitions, the transition relation actually tabulated is single-valued per state and input — i.e., a DFA augmented with λ-moves rather than a genuinely branching NFA — so it did not require subset construction to be executed directly; this would become necessary only if a future extension introduced genuine branching on a shared input symbol. Character movement, obstacle generation, scoring, and level transitions were each modeled as automata in this way, with game events (turning, jumping, collecting stars, encountering obstacles) treated as input symbols and game states (running, jumping, avoiding, failing, winning) represented as automaton states, with transitions governed by the player's actions.

The central finding of this paper, however, is not about this particular game: it is that the completeness a formal automaton is supposed to guarantee has to be checked, not assumed. [3] claims that automata-based game design "covers infinitely many possibilities" and yields "minimum bugs" compared to other design approaches, but offers no transition table and no implementation to support that claim — and the same is true of [1], [4], [5], and, until this paper, the design examined here. Once this design was given an explicit transition table and executed, it

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

failed a basic totality check in three places. That defect had survived a hand-drawn diagram, a prose narrative, and two prior editorial passes; it did not survive being run.

This does not mean automata-based game design is unsound — more rigorous work in the same space, such as MERLIN [8] and Pocong Rush [9], demonstrates that formalizing a game as a state machine and testing it against an implementation works well. What this paper shows is narrower: within the specific strand of short, informally reviewed papers that present a game automaton as a diagram and a narrative claim of correctness, that claim is not self-evidently true, and a lightweight totality-and-reachability audit — standard in hardware and software FSM verification, but absent here — is enough to catch a counterexample. A manual attempt at extending this audit to [1] and [3] (Section IV.I) found no comparable totality defect in the diagrams we could reliably read, but that finding is itself only as trustworthy as one person's reading of dense hand-drawn diagrams — and for two of [3]'s three diagrams, even that reading was not reliable enough to draw a conclusion from, which is its own small confirmation of the paper's point. The most concrete direction for future work, however, is the semantic gap mutation testing exposed in Section IV.E: the unconditional λ-exit at q7, which reaches the win state on as few as 3 of the required 100 stars. A plain finite automaton cannot express a guard like "stars ≥ 100" on a transition, so closing this gap properly — rather than patching it with an ad hoc special case
— requires extending the model to a counter automaton or pushdown automaton capable of tracking accumulated state alongside control state. Beyond that, future work includes mechanizing the totality checks in Section IV.I with tool support instead of manual review, surveying a larger sample of similarly informal automata-based game-design papers to see how common these defect classes are, and building a graphical front end around the same verified automaton engine for user testing.

Pages: 81-108
Volume-III, Issue-I, 2026
https://evjai.com/index.php/evjai

Data and Code Availability

The reference implementation (automaton.py, game.py), the functional, fuzz, and mutation test suites, the diagram- and chart-generation scripts, and the raw output (test_results.json, fuzz_results.json, mutation_results.json, and benchmark results across all reproduced runs) described in Section IV are included as supplementary material with this submission, together with the working transcription notes for the manual audit in Section IV.I. Everything needed to reproduce every number, table, and figure in this paper is included.

REFERENCES

[1] M. S. Aslam, "Computing game design with automata theory," *Int. J. Multidiscip. Sci. Eng.*, vol. 3, no. 5, 2012. [Online]. Available: https://www.ijmse.org/Volume3/Issue5/paper3.pdf

[2] D. I. A. Cohen, *Introduction to Computer Theory*, 2nd ed. New York, NY, USA: Wiley, 1986.

[3] R. A. Sabir, "A roller coaster game design using automata theory," *Int. J. Multidiscip. Sci. Eng.*, vol. 3, no. 5, 2012. [Online]. Available: https://www.ijmse.org/Volume3/Issue5/paper8.pdf

[4] B. T. Lim, "Little Hoody," research report, Universiti Malaysia Sarawak, 2020. [Online]. Available: https://ir.unimas.my/id/eprint/34204/

[5] S. Wijayanti and Nursobah, "Finite state machine model in jungle adventure game: an introduction to survival skills," *Int. J. Inf. Eng. Electron. Bus.* (IJIEEB), vol. 13, no. 4, 2021. [Online]. Available: https://www.mecs-press.org/ijieeb/ijieeb-v13-n4/IJIEEB-V13-N4-5.pdf

[6] B. McShane, "Game on! Canadians are wowed by computer game's debut," *Canada's History*, Feb.–Mar. 2021. [Online]. Available: https://www.canadashistory.ca/explore/science-technology/game-on!

[7] A. Nazmi, M. Izzat, M. S. Irwan, and M. Zulfahmi, "Hangman–Hangaroo game design using automata theory," Universiti Teknologi Malaysia. [Online]. Available: https://eprints.utm.my/97778/1/AmirulNazmi2021_HangmanHangarooGameDesign.pdf

[8] N. A. Butt, M. U. Khan, M. Z. Iqbal, and A. A. Jilani, "MERLIN — ModEl-guided Reinforcement LearnINg for automated testing of endless runner games," *Software and Systems Modeling*, 2025. [Online]. Available: https://doi.org/10.1007/s10270-025-01326-w

[9] M. B. Nendya and D. Redono, "Pocong Rush: Endless runner game based on finite state machine," *JOINCS (Journal of Informatics, Network, and Computer Science)*, vol. 5, no. 1, pp. 14–20, 2022. [Online]. Available: https://doi.org/10.21070/joincs.v5i1.1592