Simulating and Determining Autonomous Agentic Sentry Behavior using Deterministic Finite Automata

By

Abdul Basit Khatri

Dec 2025

ABSTRACT

This project explores the integration of formal language theory with autonomous robotics by applying a Deterministic Finite Automaton (DFA) to the behavioral architecture of a sentry agent. While autonomous systems often rely on scripted logic that can become prone to race conditions or undefined behaviors, this project utilizes the rigid structure of a 5-tuple mathematical model to ensure absolute predictability in real-time simulations.

By discretizing complex environmental stimuli into a formal alphabet, the system maps sensor inputs, such as intruder detection or range alerts, directly to state transitions. The implementation, written in Python, demonstrates that a finite state machine can effectively govern complex agent interactions, ensuring the system remains stable by never entering an undefined state. A primary focus of this design is the *Catch* state, a terminal trap state that provides a definitive and safe conclusion to the agent logic loop upon objective completion. This documentation provides the complete methodology, a transition table analysis, and a formal proof of determinism for the resulting sentry behavior.

1. INTRODUCTION

1.1 Background of Automata in AI

The Theory of Automata provides the mathematical foundation for computer science. Within this theory, the Deterministic Finite Automaton (DFA) is the most fundamental model of computation. While modern AI often leans toward probabilistic models, the DFA remains relevant for systems requiring high reliability and predictable outcomes.

1.2 Objectives of the Study

The primary objective of this project is to move beyond the theoretical "blackboard" definitions of DFAs and apply them to a functional game simulation. By creating a Sentry that reacts to a player, we demonstrate:

1. The conversion of spatial distances into formal symbols.
2. The execution of state-specific behaviors (Patrol vs. Chase).
3. The mathematical necessity of the "Trap State" in ending a computational process.

2. FORMAL DEFINITION OF THE SENTRY'S DFA

A DFA is formally defined as a 5-tuple $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$. My simulation strictly adheres to this mathematical structure.

2.1 The State Set ($\boldsymbol{Q}$)

We define three distinct states that represent the full behavioral range of the Sentry:

- $\boldsymbol{q_{Patrol}}$: The initial state where the Sentry follows a circular path.
- $\boldsymbol{q_{Chase}}$: The active state where the Sentry pursues the player.
- $\boldsymbol{q_{Catch}}$: The final/accepting state where the player is caught.

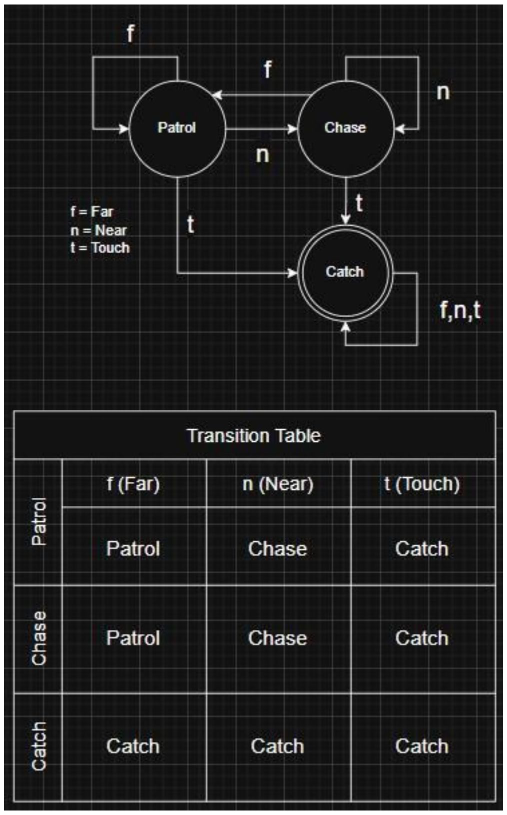
2.2 The Input Alphabet ($\boldsymbol{\Sigma}$)

The alphabet consists of symbols derived from the interaction between the Player and the Sentry:

- $\boldsymbol{f}$ **(Far)**: Distance is greater than the detection threshold.
- $\boldsymbol{n}$ **(Near)**: Player enters the detection radius.
- $\boldsymbol{t}$ **(Touch)**: Player and Sentry collide.

2.3 The Transition Function (δ)

The transition function $\delta: Q \times \Sigma \rightarrow Q$ defines how the Sentry moves between states. To remain a **Pure DFA**, the machine must be "complete," meaning every state must have exactly one transition for every symbol in Σ.



Transition Table

	f (Far)	n (Near)	t (Touch)
Patrol	Patrol	Chase	Catch
Chase	Patrol	Chase	Catch
Catch	Catch	Catch	Catch

3. SYSTEM DESIGN AND METHODOLOGY

3.1 Lexical Analysis (Symbol Generation)

The core challenge in this simulation is the "Lexical Analyzer", the part of the code that converts physical coordinates into formal symbols. The system calculates the Euclidean distance d

$$d = \sqrt{(x_s - x_p)^2 + (y_s - y_p)^2}$$

The mapping follows these rules:

1. If Collision is detected symbol = **'t'**
2. If d < 200 symbol = **'n'**
3. Otherwise, symbol = **'f'**

3.2 Implementation Strategy

The project uses a **table-driven approach**. Rather than using complex nested if statements, the state transitions are stored in a dictionary. This mirrors the mathematical transition table exactly, ensuring that the logic is separated from the execution.

4. TECHNICAL IMPLEMENTATION AND DEMO

4.1 Code

```
import pygame
import math
import random

# ===================== INIT =====================
pygame.init()
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption(“DFA Sentinel”)

clock = pygame.time.Clock()
font = pygame.font.SysFont(None, 28)

# ===================== COLORS =====================
WHITE = (255, 255, 255)
BLUE  = (50, 120, 255)    # Player
RED   = (255, 70, 70)     # Sentry
GOLD  = (255, 200, 0)     # Treasure
BLACK = (0, 0, 0)
GREEN = (60, 200, 60)

# ===================== DFA =====================
DFA = {
    “Patrol”: {“f”: “Patrol”, “n”: “Chase”, “t”: “Catch”},
    “Chase”:  {“f”: “Patrol”, “n”: “Chase”, “t”: “Catch”},
    “Catch”:  {“f”: “Catch”,  ”n”: “Catch”, “t”: “Catch”}
}

state = “Patrol”

# ===================== ENTITIES =====================
player = pygame.Rect(100, 100, 30, 30)
sentry = pygame.Rect(400, 300, 40, 40)
treasure = pygame.Rect(600, 350, 30, 30)

PLAYER_SPEED = 5
SENTRY_SPEED = 4
```

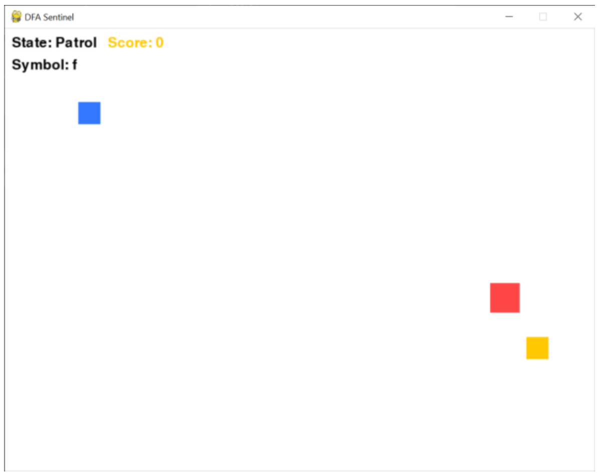
```
NEAR_DISTANCE = 200
PATROL_RADIUS = 80
patrol_angle = 0.0
returning_to_patrol = False
score = 0
game_over = False
player_won = False

# ==================== FUNCTIONS ====================
def distance(a, b):
    return math.hypot(a.centerx - b.centerx, a.centery - b.centery)

def get_symbol():
    if player.colliderect(sentry):
        return “t”
    elif distance(player, sentry) <= NEAR_DISTANCE:
        return “n”
    else:
        return “f”

def move_towards(rect, target_rect, speed):
    dx = target_rect.centerx - rect.centerx
    dy = target_rect.centery - rect.centery
    dist = math.hypot(dx, dy)
    if dist > 0:
        rect.x += int(speed * dx / dist)
        rect.y += int(speed * dy / dist)

def patrol_around_treasure():
    global patrol_angle, returning_to_patrol

    target_x = treasure.centerx + PATROL_RADIUS * math.cos(patrol_angle)
    target_y = treasure.centery + PATROL_RADIUS * math.sin(patrol_angle)
    target = pygame.Rect(target_x, target_y, 1, 1)

    # Smooth return before orbiting
    if returning_to_patrol:
        move_towards(sentry, target, SENTRY_SPEED)
        if distance(sentry, target) < 5:
            returning_to_patrol = False
    else:
        patrol_angle += 0.03
        sentry.centerx = int(target_x)
        sentry.centery = int(target_y)
```

```
def sentry_behavior():
    if state == “Patrol”:
        patrol_around_treasure()
    elif state == “Chase”:
        move_towards(sentry, player, SENTRY_SPEED)

def reset_game():
    global state, game_over, player_won, returning_to_patrol, patrol_angle, score

    # Reset DFA
    state = “Patrol”

    # Reset flags
    game_over = False
    player_won = False
    returning_to_patrol = False
    patrol_angle = 0.0
    score = 0
    # Reset positions
    player.topleft = (100, 100)
    randomize_treasure()
    sentry.center = (treasure.centerx + PATROL_RADIUS, treasure.centery)

def randomize_treasure():
    margin = 80
    treasure.x = random.randint(margin, WIDTH - margin)
    treasure.y = random.randint(margin, HEIGHT - margin)

randomize_treasure()

# ==================== GAME LOOP ====================
running = True
while running:
    clock.tick(60)
    screen.fill(WHITE)

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        # Restart game on R key
        if event.type == pygame.KEYDOWN and game_over:
            if event.key == pygame.K_r:
                reset_game()
```

```
if not game_over:
    # Player movement
    keys = pygame.key.get_pressed()
    if keys[pygame.K_w]: player.y -= PLAYER_SPEED
    if keys[pygame.K_s]: player.y += PLAYER_SPEED
    if keys[pygame.K_a]: player.x -= PLAYER_SPEED
    if keys[pygame.K_d]: player.x += PLAYER_SPEED

    # DFA Update
    symbol = get_symbol()
    next_state = DFA[state][symbol]

    # Detect Chase -> Patrol transition
    if state == “Chase” and next_state == “Patrol”:
        returning_to_patrol = True

    state = next_state

    # Sentry movement
    if state != “Catch”:
        sentry_behavior()

    # End conditions
    if state == “Catch”:
        game_over = True

    if player.colliderect(treasure):
        score += 1
        randomize_treasure()

# ===================== DRAW =====================
pygame.draw.rect(screen, GOLD, treasure)
pygame.draw.rect(screen, BLUE if state != “Catch” else BLACK, player)
pygame.draw.rect(screen, RED, sentry)

screen.blit(font.render(f”State: {state}”, True, BLACK), (10, 10))
screen.blit(font.render(f”Symbol: {get_symbol()}”, True, BLACK), (10, 40))
screen.blit(font.render(f”Score: {score}”, True, GOLD), (140, 10))

if game_over:
    if player_won:
        msg = “YOU STOLE THE TREASURE!”
        color = GREEN
        screen.blit(font.render(msg, True, color),((WIDTH / 3.5), HEIGHT // 2))
    else:
            msg = “PLAYER CAUGHT (ACCEPTING STATE)”
            color = RED
            screen.blit(font.render(msg, True, color),((WIDTH // 4), HEIGHT // 2))

        screen.blit(font.render(“Press R to Restart”, True, BLACK),(WIDTH // 2 - 120, HEIGHT // 2 + 40))

    pygame.display.flip()

pygame.quit()
```

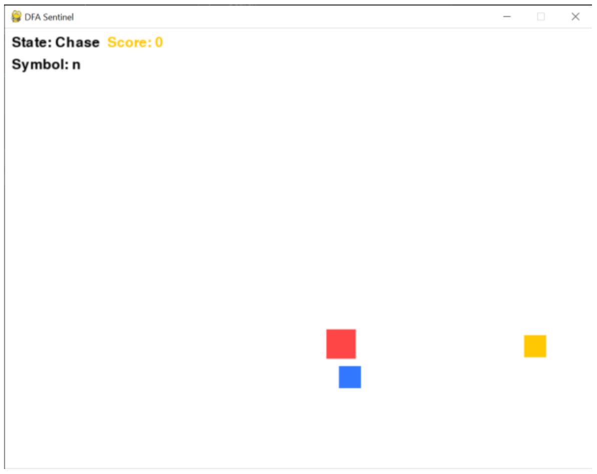
4.2 Demo Screenshots

4.2.1 Patrol State



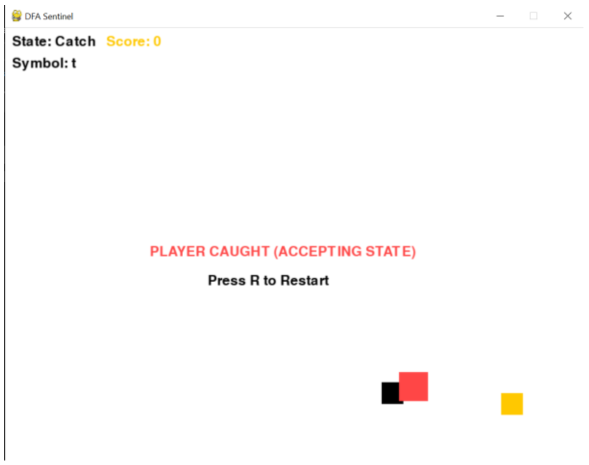
State: Patrol Score: 0
Symbol: f

4.2.2 Chase State



State: Chase Score: 0
Symbol: n

4.2.3 Catch State



State: Catch Score: 0
Symbol: t

PLAYER CAUGHT (ACCEPTING STATE)
Press R to Restart

5. MATHEMATICAL ANALYSIS OF THE MODEL

5.1 Determinism and One-to-One Mapping

A “Pure DFA” must have no ambiguity. In my code:

$$next\_state \;=\; DFA[state][symbol]$$

Because this dictionary lookup only returns a single value, it proves the system is deterministic. There are no “epsilon transitions” (ϵ), and no state has two different paths for the same symbol.

5.2 The Absorbing (Trap) State

The state Catch is a classic example of an absorbing state in Automata Theory. Once M enters q_{Catch}, it is impossible to leave. This is represented by the transitions:

$$\delta(Catch, f) = Catch$$

$$\delta(Catch, n) = Catch$$

$$\delta(Catch, t) = Catch$$

In the context of the game, this symbolizes a “Terminal Condition.”

5.3 Complexity Analysis

The time complexity for a state transition in this DFA is $O(1)$. Regardless of the number of states added to the machine, the transition lookup remains constant. This makes DFAs the most efficient models for real-time agent control.

6. COMPARATIVE DISCUSSION

6.1 Why a DFA and not an NFA?

A Nondeterministic Finite Automaton (NFA) would allow the Sentry to be in multiple states at once (e.g., patrolling and chasing simultaneously). While NFAs are equivalent to DFAs in power (Kleene’s Theorem), they are harder to implement in a real-time game loop. A Pure DFA provides the “Single Source of Truth” needed for clear AI behavior.

6.2 Why not Mealy or Moore Machines?

While Mealy and Moore machines produce output based on transitions or states, my project focuses on the **State Recognition** aspect of Automata. We treat the agent’s behavior as a manifestation of the state itself, rather than an external output string. This keeps the focus strictly on the DFA’s ability to categorize environmental “strings” into behaviors.

7. CONCLUSION

This project demonstrates the strength of Finite Automata in software design. By modeling a Sentry as a Pure DFA, we created a system that is mathematically verifiable and logically sound. The simulation proves that environmental inputs can be successfully treated as formal language, and transitions can be managed through a rigorous 5-tuple definition.

8. REFERENCES

1. **R. Ghzouli, T. Berger, E. B. Johnsen, A. Wasowski and S. Dragule (2023),** *"Behavior Trees and State Machines in Robotics Applications," in IEEE Transactions on Software Engineering, vol. 49, no. 9, pp. 4243-4267, Sept. 2023, doi: 10.1109/TSE.2023.3269081.*
2. **Millington, I. (2019).** *AI for Games (Third Edition).*
3. **A. Ganapathi, P. Sivakumar, A. Elango, H. Gupta and N. Panda,** *"Exploring NPC Behaviors in Games through Finite Automata," 2024 5th IEEE Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2024, pp. 1-8, doi: 10.1109/GCAT62922.2024.10923923.*
4. **Ostapenko, I. Y., and E. D. Kosenko. (2017) "**Using The Method Of Finite Automaton For Artificial Intelligence Modeling In Computer Games." *ІНФОРМАТИКА, ІНФОРМАЦІЙНІ СИСТЕМИ ТА ТЕХНОЛОГІЇ*: 84.
5. **Ali, S., Kalyan, A., & Jamil, N. (2019).** *Design and implementation of Ludo game using automata theory.* (It used automata for game behavioral design)
6. **Uludagli, Cagkan & Oguz, Kaya. (2023).** *Non-player character decision-making in computer games. Artificial Intelligence Review. 56. 1-33. 10.1007/s10462-023-10491-7.*
7. **Yusoff, Yusliza & Nazmi, Amirul & Izzat, Mohd & Irwan, Mohd & Zulfahmi, Muhd & Sallehuddin, Roselina. (2021).** *Hangman–Hangaroo Game Design Using Automata Theory. International Journal of Innovative Computing. 11. 7-11. 10.11113/ijic.v11n1.275.*
8. **Nambiar, A.S., Likhita, K., Sri Pujya, K.V.S. and Supriya, M., (2022).** *Design of Super Mario Game Using Finite State Machines. In Computer Networks and Inventive Communication Technologies: Proceedings of Fifth ICCNCT 2022 (pp. 739 - 752).*
9. **Song, Boyao. (2022).** *Game Development with Python Using Pygame. 10.4108/eai.17-6-2022.2322877.*
10. **Someone. (2024).** *Mastering Pygame: A Theoretical Approach to Building a "Chase The Enemy" Game in Python* [The pen name is "someone"]