Karachi Institute of Engineering & Technology (KIET)

CoCIS Department

IoT Bitstream Log Validator (TM Based)

Theory of Automata Project

Course Name:	Theory of Automata
CID:	118692
Instructor:	Miss Misbah

Group Members

Name	**Student ID**
Syed Muhammad Nihal	66044
Zaid Maju	66057

Abstract

When making and testing IoT and embedded systems, we often record serial communication logs. These logs help us find problems like messed-up data or framing issues. Our project made a simple Turing Machine (TM) simulator in Python. It checks UART-like frames in these logs.

The TM reads a raw data stream and checks its structure: the start bit, data bits, parity, and stop bit. Then, it writes notes on the tape to show if each frame is good or bad. We focused on making sure you can follow every step. Every decision the TM makes—reading, writing, moving—is clear, which is perfect for showing how Theory of Automata works.

Contents

1 Introduction

1.1 Why We Did This

IoT devices commonly communicate using lightweight serial protocols, and during debuging it is normal to capture raw bitstreams for offline analysis. A practical challenge is verifying whe=ther frames in a captured log follow the expected format (e.g. correct start/stop bits and correct parity). This project uses a Turing Machine to model that verification process in a way that is explainable step -by- step, instead of taking validation as a black- box function.

1.2 Why This Matters

A finite state machine can model control flow, but a TM can additionally rewrite its tape and revisit earlier positions, that makes it a strong educational model for "mechanical" computation. In this project, the TM does not only just accept/reject; it also rewrites the tape to produce a marked output stream that looks like a tester's validation report. This makes the project meaningful both as (i) a Theory of Automata implementation and (ii) a simplified example of protocol trace validation.

1.3 What We Wanted to Do

- Make a Python simulator for a single-tape, deterministic TM
- Check UART-like 8E1 frames in a recorded data stream.
- Show the TM head moving both ways (right to read, left to mark results).
- Make a report file that shows the original log and the checked output.

2 Other Projects Like This

TThis project got its ideas from two main areas: (1) simulators for learning about automata and TMs, and (2) ways to check communication protocols.

2.1 Automata and TM Simulators

Hamada (2013) discusses simulators for computational models (including Turing machines) as a learning environment where students can experiment with machine behavior step-by-step [4]. Such work supports the idea that a TM simulator is not only a theory exercise but also a practical educational tool.

2.2 Protocol Validation

Holzmann (1987) presents an approach to automated protocol validation (Argos) based on formal models of communicating finite state machines, emphasizing systematic validation and analysis of protocol behaviors [5]. While our implementation is intentionally simplified, the overall motivation is similar: validate communication behavior using a formal model and produce interpretable traces.

2.3 UART Framing and Parity

UART framing is commonly described using a start bit, data bits, an optional parity bit, and a stop bit [2, 3]. Parity is used as a basic error detection mechanism, where even parity sets the parity bit so the total number of 1s becomes even [1].

3 How We Did It

3.1 Overall Approach

The solution is split into two layers:

- **TM Engine Layer:** Implements tape, head, states, transition function, and step execution.
- **Protocol Validator TM:** A concrete TM "program" (transition table) that validates UART-like frames in a stream.

3.2 Input Format

The input is a text log file (e.g., `iot_capture.log`), where each line contains metadata and a `raw=` field holding a bitstream. Frames are separated by the delimiter `|`. Noise/idle bits may appear between frames.

3.2.1 Frame Format (8E1 Model)

We validate a simplified UART-like frame:

$$\texttt{0}\ d_0 d_1 d_2 d_3 d_4 d_5 d_6 d_7\ p\ \texttt{1}$$

where `0` is start bit, `d0..d7` are 8 data bits, `p` is the parity bit, and `1` is the stop bit [2, 3]. Even parity is enforced so that the parity bit makes the total number of 1s in data+parity even [1].

3.3 TM Tape Design

- Tape alphabet includes: `0, 1, |, _` plus annotation symbols `s, K, E`.
- `s` is a temporary marker written on the start bit when a frame is detected.
- `K` indicates a validated (OK) frame; `E` indicates an invalid frame.

3.4 How the TM Head Moves

The validation cycle for each frame is:

1. Move **right** to locate a start bit (`0`) and mark it as `s`.
2. Move **right** through data bits and parity bit, tracking parity in the control states.
3. Read stop bit and decide OK/ERROR.
4. Move **left** back to the `s` marker and overwrite it with `K` or `E`.
5. Move **right** to the next delimiter `|` and repeat.

This explicitly demonstrates that the TM uses both directions and uses tape rewriting as part of computation.

3.5 Python Implementation Details

3.5.1 Core Components

- `Tape`: a sparse mapping from indices to symbols, supporting infinite extension.
- `TuringMachine`: executes $\delta(q, a) = (q', b, m)$ via a transition dictionary.
- `Trace Mode`: prints step-by-step state, head index, read symbol, write symbol, and movement.

3.5.2 Output

After validation, the program writes `validated_output.log` containing:

- Original selected log line
- Raw bitstream
- TM-marked bitstream
- Summary counts: number of OK frames and error frames

4 Results & Experiments

4.1 Experimental Setup

Experiments were run on a standard development environment using PyCharm. The input log file contains multiple captured lines; each test selects one entry and validates its `raw=` field.

4.2 Test Data

The dummy log simulates real capture-style entries, including timestamps, device IDs, and a raw stream with multiple frames separated by `|`. Some frames are intentionally constructed to contain parity or stop-bit errors to demonstrate detection.

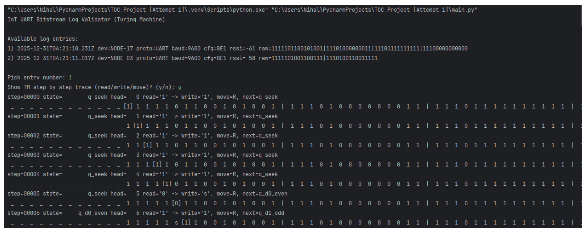
4.3 Observed Output

The main result is a transformed bitstream where each frame's start bit is overwritten with:

- `K` for valid frames
- `E` for invalid frames

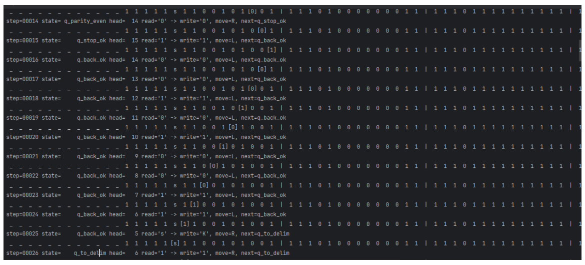
This output is saved into `validated_output.log` for traceability.

4.4 Screenshots



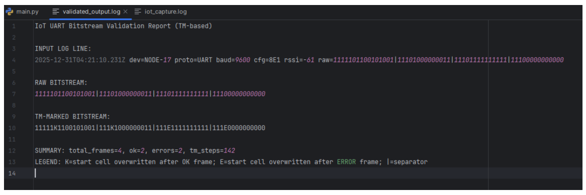
```
"C:\Users\Nihal\PycharmProjects\TOC_Project [Attempt 1]\.venv\Scripts\python.exe" "C:\Users\Nihal\PycharmProjects\TOC_Project [Attempt 1]\main.py"
IoT UART Bitstream Log Validator (Turing Machine)

Available log entries:
1) 2025-12-31T04:21:10.231Z dev=NODE-17 proto=UART baud=9600 cfg=8E1 rssi=-61 raw=1111101100101001|11101000000011|11101111111111|11100000000000
2) 2025-12-31T04:21:11.017Z dev=NODE-03 proto=UART baud=9600 cfg=8E1 rssi=-58 raw=1111101001100111|1110100110011111

Pick entry number: 1
Show TM step-by-step trace (read/write/move)? (y/n): y
step=00000 state=       q_seek head=   0 read='1' -> write='1', move=R, next=q_seek
 _ _ _ _ _ _ _ _ _ _ _ _ [1] 1 1 1 1 0 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00001 state=       q_seek head=   1 read='1' -> write='1', move=R, next=q_seek
 _ _ _ _ _ _ _ _ _ _ _ _ 1 [1] 1 1 1 0 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00002 state=       q_seek head=   2 read='1' -> write='1', move=R, next=q_seek
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 [1] 1 1 0 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00003 state=       q_seek head=   3 read='1' -> write='1', move=R, next=q_seek
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 [1] 1 0 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00004 state=       q_seek head=   4 read='1' -> write='1', move=R, next=q_seek
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 [1] 0 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00005 state=       q_seek head=   5 read='0' -> write='s', move=R, next=q_d0_even
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 [0] 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00006 state=    q_d0_even head=   6 read='1' -> write='1', move=R, next=q_d1_odd
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s [1] 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
```

(a) Console run (record selection and raw stream).



```
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 0 1 [0] 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00014 state= q_parity_even head=  14 read='0' -> write='0', move=R, next=q_stop_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 0 1 0 [0] 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00015 state=    q_stop_ok head=  15 read='1' -> write='1', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 0 1 0 0 [1] | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00016 state=    q_back_ok head=  14 read='0' -> write='0', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 0 1 0 [0] 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00017 state=    q_back_ok head=  13 read='0' -> write='0', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 0 1 [0] 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00018 state=    q_back_ok head=  12 read='1' -> write='1', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 0 [1] 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00019 state=    q_back_ok head=  11 read='0' -> write='0', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 1 [0] 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00020 state=    q_back_ok head=  10 read='1' -> write='1', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 0 [1] 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00021 state=    q_back_ok head=   9 read='0' -> write='0', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 0 [0] 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00022 state=    q_back_ok head=   8 read='0' -> write='0', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 1 [0] 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00023 state=    q_back_ok head=   7 read='1' -> write='1', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s 1 [1] 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00024 state=    q_back_ok head=   6 read='1' -> write='1', move=L, next=q_back_ok
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 s [1] 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00025 state=    q_back_ok head=   5 read='s' -> write='K', move=R, next=q_to_delim
 _ _ _ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 [s] 1 1 0 0 1 0 1 0 0 1 | 1 1 1 0 1 0 0 0 0 0 0 0 1 1 | 1 1 1 0 1 1 1 1 1 1 1 1 1 1 |
step=00026 state=   q_to_delim head=   6 read='1' -> write='1', move=R, next=q_to_delim
```

(b) Trace mode showing read/write/move steps.



```
IoT UART Bitstream Validation Report (TM-based)

INPUT LOG LINE:
2025-12-31T04:21:10.231Z dev=NODE-17 proto=UART baud=9600 cfg=8E1 rssi=-61 raw=1111101100101001|11101000000011|11101111111111|11100000000000

RAW BITSTREAM:
1111101100101001|11101000000011|11101111111111|11100000000000

TM-MARKED BITSTREAM:
11111K1100101001|111K1000000011|111E1111111111|111E0000000000

SUMMARY: total_frames=4, ok=2, errors=2, tm_steps=142
LEGEND: K=start cell overwritten after OK frame; E=start cell overwritten after ERROR frame; |=separator

```

(c) Generated validated_output.log (report file).

Figure 4.1: Execution evidence of the TM-based validator.

4.5 Discussion

In trace mode, the step- by -step output confirms that validation is achieved through TM transitions rather than direct high- level parsing . The leftward “return” phase is especially important in showing tape rewriting: the TM revisits the start of the frame to write the final decision symbol.

5 Conclusion

This project implemented a deterministic single-tape Turing Machine simulator and applied it to a realistic testing scenario: validating UART-like frames inside an IoT serial capture log. The main contribution is not only accept/reject behavior but also a rewritten tape that marks frame-level outcomes, producing an output artifact similar to a tester's report. Future work could extend the validator to more advanced framing (e.g., bit-stuffing protocols) or to compute stronger integrity checks.

Bibliography

[1] Uart: A hardware communication protocol understanding ... https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html.

[2] Stm32 uart frame formats: Start, data, parity, and stop bits. https://fastbitlab.com/stm32-uart-lecture-3-uart-frame-formats/.

[3] Universal asynchronous receiver-transmitter. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter.

[4] Mohamed Hamada. Turing machine and automata simulators. *Procedia Computer Science*, 2013. URL https://www.sciencedirect.com/science/article/pii/S1877050913004572. Available online via ScienceDirect.

[5] Gerard J. Holzmann. Automated protocol validation in argos: Assertion proving and scatter searching. *IEEE Transactions on Software Engineering*, 1987. URL https://spinroot.com/gerard/pdf/inprint/ieee87.pdf. PDF available online.