Abstract—Explainability remains the holy grail in designing the next-generation pervasive artificial intelligence (AI) systems. Current neural network based AI design methods do not naturally lend themselves to reasoning for a decision making process from the input data. A primary reason for this is the overwhelming arithmetic complexity.

Built on the foundations of propositional logic and game theory, the principles of learning automata are increasingly gaining momentum for AI hardware design. The lean logic based processing has been demonstrated with significant advantages of energy efficiency and performance. The hierarchical logic underpinning can also potentially provide opportunities for by-design explainable and dependable AI hardware. In this paper, we study explainability and dependency using reachability analysis in two simulation environments. Firstly, we use a behavioral SystemC model to analyze the different state transitions. Secondly, we carry out illustrative fault injection campaigns in a low-level SystemC environment to study how reachability is affected in the presence of hardware stuck-at 1 faults. Our analysis provides the first insights into explainable decision models and demonstrates dependency advantages of learning automata driven AI hardware design.

I. INTRODUCTION

Tsetlin machine (TM) is a promising new machine learning (ML) algorithm, recently proposed by Ole-Christoffer Granmo [1]. It is built on Mikhail Tsetlin’s original learning automaton based control principles for complex systems [2] as well as contemporary linear tactics based game theory [3]. Discretization of the control states is a major simplification in TM, which allowed for using linear tactics to reinforce the states over time in parallel [4]. As such, TM can define a machine learning problem through hierarchical and powerful propositional logic expressions [1]. These have enabled the design of the first-ever hardware architecture [5], which demonstrated significantly lower energy consumption and resource frugality than state-of-the-art neural networks alike.

Figure 1 depicts a schematic diagram of different structural blocks in the TM hardware. As can be seen, a fundamental property of TM is data encoding at the input as a set of Boolean digits rather than binarized numbers with positional significance. These digits and their complements define a set of literals. The combination of these literals participating in the definition of the output class is controlled by Tsetlin automata (TAs), which are finite automata with linear tactics. Each TA constitutes a set of states that define the discrete action space. During training rewards are used to reinforce the states towards an action and penalties are used to transition the states for weakening automaton confidence in performing an action. Ensemble of TA actions define the output of a clause.
To demonstrate the number of reinforcement steps needed to fully converge to the final state as well as the corresponding action, we consider an automaton with \( 2N = 6 \) internal states and 2 actions. The state transition equations of all automaton states are given as below:

\[
G(s_n) = \begin{cases} 
\alpha_1 & \text{if } 1 \leq n \leq N \\
\alpha_2 & \text{if } (N + 1) \leq n \leq 2N
\end{cases}
\]  

To illustrate how bounded TA state transitions contribute to reachable learning formulation in the TM algorithm, we simulate a 2-input XOR using a behavioral SystemC description of the same. The inputs and their complements constitute 4 literals and as such 4 TA are used in each clause. Each automaton consists of 6 states as exemplified above. A total of 4 clauses are used in the inference circuit, of which 2 are positive clauses and 2 are negative clauses into the majority voting (i.e. classification) circuit. Figure 3 shows the internal states of 4 TA, defining one clause output only.

The state transitions in a training step correspond to 4 datapoints (which are the set of literals), but only 2 are shown. The TA start with the same initial states of \( s_3 \). After the first datapoint \( X_0 = [0, 0] \) reinforcement, the clause sees an output of 1 as all TA states suggest no inclusion of 0 literals. Overall, this results in an erroneous classification and as such 2 penalties in TA_0 and TA_2, causing them to transition.
to \( s_4 \) (Figure 3(a)). After the second datapoint (\( X_1[0, 1] \)), the clause output is 0 as the TA_2 state favors the inclusion of a 0 literal (\( X_0' \)). However, as the clause output generates a wrong classification but with a lower error, TA_1 is penalized to \( s_3 \) and TA_3 is rewarded to \( s_2 \) (Figure 3(b)). With more datapoints and their associated single-step reinforcements (Eqns.(2)-(7)), the TA continue to settle for states with higher reward probabilities, e.g., \( s_1 \) and \( s_6 \) (Figure 3(c)). This guarantees convergence during training.

The above analysis of reachability for the XOR example shows an important property of the TM, where the (integer) vector of states of TAs is effectively mapped (contracted) onto the (binary) vector of actions include/exclude. This mapping allows us to define the notion of equivalence between the states of TAs, and hence define the conditions for detecting convergence to the trained state as soon as possible, thus improving the efficiency of the system and its performance.

### III. Dependability Analysis

We continue our reachability analysis further in this section and study how dependability of the system is affected in the presence of faults. For these, we use an RTL SystemC model of a 2-input XOR with fault injection handles using [6]. Our fault injection campaign includes a stuck-at 1 fault model, applied to the reinforcement part, i.e. TA. Our future research includes comprehensive fault injection in the TM.

For demonstration purposes, we inject a stuck-at 1 fault in the least significant bit (i.e. bit position 0) of automaton 1 (i.e. TA_1) within the first clause. This is done to observe how this fault can change TA_1 state transitions (see Figure 4) when compared with the same in Figure 3. As can be seen, the automaton assumes an initial state of \( s_3 \) and does not change the state after the iteration step 1. This is equivalent to a no-action reinforcement of 4 datapoints. In the iteration step 2, the automaton state is penalized towards \( s_4 \) through an increment operation (i.e. from register value of 011 to 100). However, due to the fault the automaton transitions to \( s_5 \) (i.e. a register value of 101). After iteration step 3, the automaton is rewarded towards \( s_6 \). However, the faulty automaton state tries to transition to an unreachable state of \( s_7 \). As the state bounds are protected through a \( [\text{modulus 6 +1}] \) operation internally, the automaton changes the state to \( s_1 \). The automaton retains this state until automata in all clauses are converged (after 18 iteration steps). Note that unlike the TA_1 state in the first clause of the fault-free TM (Figure 3), the faulty automaton excludes the associated Boolean literal, \( X_0' \).

From Figure 4 it is evident that a fault in an automaton can influence its state transitions significantly. For example, the state values of TA_1 are constrained to only 3 out of 6 states: \( s_1 \), \( s_3 \) and \( s_5 \), 2 of which are inclined towards the exclude action. This affects the reinforcement as well as inference for the first clause, resulting in a maximum achievable accuracy of 75%. Indeed, defining the relationship between input datapoints and output classes can be challenging with limited state transitions if there are no other means of fault masking or mitigation.

![Figure 4](image1.png)

**Figure 4.** The impact of a stuck-at 1 fault in TA_1’s state transition and hence learning. Notice how originally included literal is now excluded because of fault in the TA.

For demonstration purposes, we inject a stuck-at 1 fault in the least significant bit (i.e. bit position 0) of automaton 1 (i.e. TA_1) within the first clause. This is done to observe how this fault can change TA_1 state transitions (see Figure 4) when compared with the same in Figure 3. As can be seen, the automaton assumes an initial state of \( s_3 \) and does not change the state after the iteration step 1. This is equivalent to a no-action reinforcement of 4 datapoints. In the iteration step 2, the automaton state is penalized towards \( s_4 \) through an increment operation (i.e. from register value of 011 to 100). However, due to the fault the automaton transitions to \( s_5 \) (i.e. a register value of 101). After iteration step 3, the automaton is rewarded towards \( s_6 \). However, the faulty automaton state tries to transition to an unreachable state of \( s_7 \). As the state bounds are protected through a \( [\text{modulus 6 +1}] \) operation internally, the automaton changes the state to \( s_1 \). The automaton retains this state until automata in all clauses are converged (after 18 iteration steps). Note that unlike the TA_1 state in the first clause of the fault-free TM (Figure 3), the faulty automaton excludes the associated Boolean literal, \( X_0' \).

From Figure 4 it is evident that a fault in an automaton can influence its state transitions significantly. For example, the state values of TA_1 are constrained to only 3 out of 6 states: \( s_1 \), \( s_3 \) and \( s_5 \), 2 of which are inclined towards the exclude action. This affects the reinforcement as well as inference for the first clause, resulting in a maximum achievable accuracy of 75%. Indeed, defining the relationship between input datapoints and output classes can be challenging with limited state transitions if there are no other means of fault masking or mitigation.

![Figure 5](image2.png)

**Figure 5.** The impact of stuck-at 1 faults in TA_1 at different bit positions in terms of accuracy and performance; number of clauses are varied to observe how clause redundancy naturally masks the faults.

Next, we will explore if TM design allows for fault masking if resource provisions are relaxed in terms of number of clauses in the inference part (Figure 1). For this, we carried out another experiment with variable number of clauses from 4 to 12, each with 6 TA states. Figure 5 presents the results in terms of the maximum training accuracy and the corresponding number of iteration steps to convergence. To observe the significance of fault positions, we injected stuck-at 1 faults in different positions of the TA_1 register: at bit positions 0, 1 and 2. As expected, when the number of clauses are increased to 8 or more, the training accuracy increases to 100% for all fault injection campaigns (Figure 5(a)). Provisioning more clauses in TM allows for further state transition variations.
More variations, in turn, provide masking of the stuck-at fault completely. This observation is akin to traditional fault-tolerant design principles [7], where redundant hardware resources together with majority voting mitigate the impact of faults. TM already features majority voting in the classification circuit and as such it allows for more clauses to independently process the different automaton states internally and yet find the team of automata that correctly define the relationship between Boolean literals and output classes.

Figure 6 shows the maximum training accuracy as well as their convergence times. As can be seen, the accuracy increases from 75% to 100% when the number of states is increased from 6 to 8, corresponding to a 1-bit increase in the automaton register size from 3 (Figure 6(a)). The increase in the register size as well as the state values allow each automaton to explore a larger state-space. In a 6-state (3-bit) automaton register, a stuck-at 1 fault in bit position 0 can result in 3 allowable states. Conversely, in an 8-state (4-bit) automaton register the same can results in 4 allowable states. Note that, with one clause unable to provide correct outcomes, the 6-state automaton converges faster than the 8-state automaton. However, as more state values are allowed in the automaton, the learning converges faster to the maximum accuracy of 100% (Figure 6(b)).

IV. SUMMARY AND CONCLUSIONS

We presented the first insights into explainability and dependability of learning automata based AI hardware design using reachability analysis. Our key findings are as follows. Firstly, with a bounded state-space, TM can start from random initial TA states and yet reach a learnt state with incremental reinforcements. As the initial training datapoints generate erroneous outcomes, the randomization-enhanced feedback mechanism continues to navigate to and strengthen an action with higher reward probabilities when it reduces errors [1]. This guarantees convergence. Secondly, with suitably chosen redundant clauses and thereby more state transition variations, stuck-at faults can be fully masked without requiring any additional fault mitigation strategy. The TM can achieve the maximum accuracy faster during training with higher number of clauses. Thirdly, by allocating more TA states and as such expanding the valid state-space, stuck-at faults can also be completely masked. Under fault scenarios, higher number of states allows for faster learning convergence. Compared with clause redundancy approach, expanding the state register sizes provides more energy-frugality. Our future work includes reachability analysis using formal checking tools and theory with comprehensive fault injection campaigns.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the inputs from Jie Lei as well as funding from EPSRC IAA project “Whisperable” and EPSRC grant STRATA (EP/N023641/1).

REFERENCES


