# Explicit State Model Checking Effects on Learning-Based Testing

Iram Jaffar[1], Muddassar Azam Sindhu[1], Shafiq-ur-Rehman[*2]
[1]Computer Science Department, Quaid-i-Azam University, Islamabad, Pakistan.
[2]Center of Excellence in Sciences and Applied Technologies (CESAT), Islamabad, Pakistan.
[*]**Correspondence:** shafiq.rehmaan@gmail.com

**Citation |** Jaffar. I, Sindhu. M. A, Rehman. S, "Explicit State Model Checking Effects on Learning-Based Testing", IJIST, Special Issue. pp 258-271, Oct 2024
**Received|** Oct 13, 2024 **Revised|** Oct 14, 2024 **Accepted|** Oct 21, 2024 **Published|** Oct 28, 2024.

Exploring the impact of integrating an explicit state model checker into the learning-based testing (LBT) framework presents an intriguing challenge. Traditionally, LBT has leveraged symbolic model checkers such as NuSMV and SAL, which use Binary Decision Diagrams (BDDs) to analyze multiple states concurrently. In contrast, explicit state model checkers evaluate one state at a time, a key distinction that suggests potential advantages for explicit state checking in the context of LBT. Thus, it is valuable to investigate how integrating an explicit state model checking algorithm might influence the performance of LBT. Model checkers explore the state space to verify conformance with user-defined correctness requirements, typically represented as Linear Temporal Logic (LTL) formulas. If a property violation is detected, it is presented as a counterexample. NuSMV and SAL employ different algorithms for generating and displaying counterexamples. This paper specifically examines the effect of SPIN-generated counterexamples on the LBT process. Evaluation metrics include Total LBT Iterations, First Bug Reporting Time (in milliseconds), Counterexample Length, Precision, and Efficiency, among others. Total Model Checking Time (in milliseconds) captures the cumulative time spent verifying the model over all iterations. SPIN consistently requires the least time for all specifications compared to other model checkers. As a result, experiments demonstrate that SPIN is more efficient when integrated with LBT, leading to faster convergence of the LBT hypothesis to the target System Under Test (SUT) in comparison to NuSMV and SAL.

**Keywords:** Software Testing; Explicit State Model Checking; Learning-Based Testing; Counterexamples.

## Introduction

Software testing aims to identify errors in a system, using a variety of techniques. White-box testing, for example, involves testing with knowledge of the system's internal structure. In contrast, black-box testing [1] focuses on evaluating the software without any insight into its internal workings. Learning-based testing (LBT) [6], a form of black-box testing, utilizes automaton learning algorithms in combination with model checkers [8]. In the realm of formal verification, model checkers are employed to verify that a system's model conforms to specified requirements [21]. When a system meets these formal specifications, its behavior is considered verified. However, if a discrepancy is found, the model checker generates a counterexample, which serves as proof of non-conformance. A counterexample represents a sequence of states starting from the initial state, where the system's behavior violates the specified property [31].

The challenge of integrating an explicit state model checker into the LBT framework offers an intriguing opportunity for further exploration. Historically, LBT has relied on symbolic model checkers such as NuSMV and SAL, which use Binary Decision Diagrams (BDDs) to analyze multiple states simultaneously. In contrast, explicit state model checkers evaluate one state at a time. This difference suggests that explicit state checking may present both distinct advantages and challenges for LBT. It is therefore important to investigate how the inclusion of an explicit state model checking algorithm influences the performance of LBT. This paper focuses specifically on the role of SPIN-generated counterexamples in the LBT process, evaluating metrics such as Total LBT Iterations, First Bug Reporting Time (in milliseconds), counterexample length, Precision, and Efficiency. Additionally, Total Model Checking Time (in milliseconds) aggregates the time taken to verify the model across all iterations. SPIN consistently outperforms NuSMV and SAL in terms of time efficiency across all specifications. Consequently, the experiments show that SPIN, when integrated with LBT, leads to faster convergence of the LBT hypothesis to the target System Under Test (SUT) compared to the other two model checkers.

## Literature Review:

The Learning-based Testing (LBT) framework [6] was previously developed and integrated with two symbolic model checkers, NuSMV and SAL [23]. In the LBT approach, the System Under Test (SUT) is treated as a black-box. LBT generates test cases by incrementally model-checking hypotheses of the SUT, which are constructed using automaton learning algorithms, against formal requirements. As the hypotheses are refined through each step of the incremental learning process, the generated test cases are also refined, thanks to the feedback loop inherent in the LBT framework. However, LBT has not previously been applied with the SPIN model checker, which is an explicit state model checker. Unlike symbolic model checkers, SPIN uses a nested depth-first search technique to explore the state space of a system model.

Model checking algorithms rely on a specific input language, and models must be encoded in this language for verification. Each model checking algorithm requires a compatible specification language to define the system's requirements. For instance, SAL intermediate language, SMV, and Promela are commonly used specification languages for reactive systems in SAL, NuSMV, and SPIN model checkers [9]. To verify the system, the properties to be checked must be formally specified, typically using property specification languages like Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) [10]. Model checkers are formal verification (FV) tools designed to ensure that systems behave according to their specifications. While counterexamples are generally considered undesirable in the FV community, they can be valuable as test cases for the testing community. Previous research has explored counterexample generation and their utility for testing purposes [23]. Next, we will

review several model checkers, highlighting the key differences in their approaches as discussed above.

**SPIN:**

The SPIN model checker is primarily used for verifying asynchronous systems [11]. It incorporates techniques such as abstraction and message passing through channels to model system behavior. SPIN is particularly effective at achieving logical consistency in its verifications. To perform requirement verification, SPIN translates the property specification, typically expressed in Linear Temporal Logic (LTL), into a Büchi automaton. A property violation is identified when the intersection of the Büchi automaton and the system model is non-empty. SPIN's model checking approach is based on the explicit representation of the entire state space, where it explores the states forward to detect potential property violations. In this context, a counterexample generated by SPIN is essentially a path from the initial state to a state where the property violation occurs [25].

**NuSMV:**

NuSMV is a symbolic model checker (SMV) that relies on Binary Decision Diagrams (BDDs) for model verification [12]. It was the first model checker to use BDDs as the foundation for its verification process. To test a given specification, NuSMV accepts a model of the System Under Test (SUT) in the SMV language and employs an Ordered Binary Decision Diagram (OBDD) to derive a transition system. The search algorithm it uses to verify whether the specification is satisfied is also based on OBDDs. If the model does not conform to the specified requirements, NuSMV generates a counterexample in the form of a trace, illustrating the path that leads to the violation.

**SAL:**

SAL [13] was developed through a collaboration between Stanford, Verimg, and Berkeley in 2013. It is built on a blackboard architecture [14], integrating a variety of specialized tools and algorithms to verify and specify the transition properties of a System Under Test (SUT). The SAL intermediate language is used to describe the system based on its transition model, and it has a unique syntax tailored to this purpose. SAL's validation tools leverage a combination of model checking, theorem proving, and other techniques to ensure the correctness of the system's behavior.

**Proposed Approach:**

This paper explores the feasibility of generating test cases for Learning-Based Testing (LBT) using the SPIN model checker by integrating two Systems Under Test (SUTs) with an incremental automaton learning algorithm [25], [26]. SPIN is a promising candidate for LBT due to its efficiency, explicit state verification capabilities, and robust counterexample generation features [21]. One of the SUTs is a cruise controller, an embedded device commonly found in modern vehicles, which has been previously tested using NuSMV and SAL within the context of LBT. The other SUT is a three-floor elevator system. The proposed approach is illustrated in Figure 1 below:

**Vehicle Cruise Controller (5 Bit):**

A vehicle cruise control system, as the name suggests, is a safety-critical system [25], [28], [30] designed to automatically regulate the speed of a vehicle. The cruise control function significantly enhances the driver's convenience by adjusting the throttle according to a user-defined setting. This function can be disabled when not needed and must be explicitly activated when automatic cruise control is desired. Such a system can be modeled as a deterministic Kripke model. A simplified version of the cruise control model uses five inputs: $\Sigma$ = {brake, decelerate, accelerate, gas, button}, and outputs a five-bit vector, where bit 1 and bit 2 represent the mode, bit 3 and bit 4 represent the speed, and bit 5 represents the button status.

- **Brake:** The brake is used by the driver to reduce the vehicle's speed.
- **Decelerate:** Deceleration represents an external factor affecting speed, such as when the vehicle is going uphill.
- **Accelerate:** Acceleration is another external factor impacting speed, typically when the vehicle is going downhill.
- **Gas:** Gas indicates the value of the throttle or accelerator pedal.
- **Button:** The button is a physical switch that toggles the cruise control feature on or off.

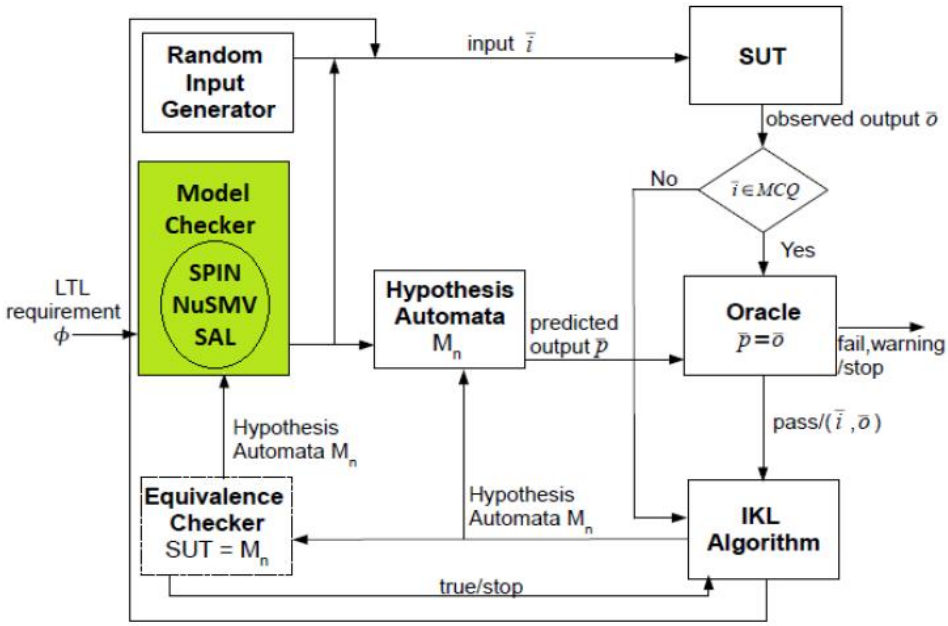


**Figure 1**. Proposed Framework of LBT

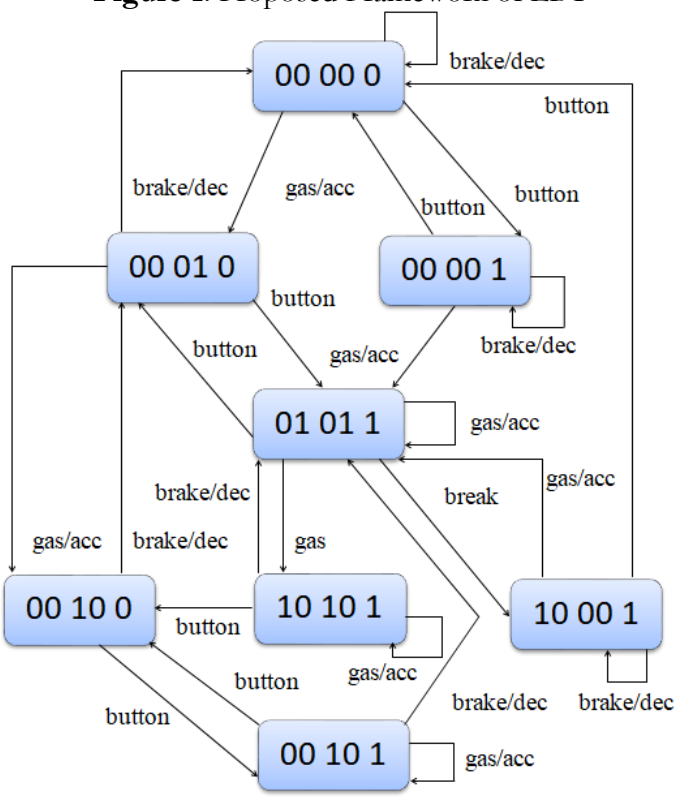


**Figure 2.** A 5-bit Cruise Controller Model

The safety requirements of the Cruise Controller, expressed as LTL (Linear Temporal Logic) formulae [10], [12], are as follows: Req 1 G (mode = cruise & speed = 1 & in = dec → X (speed = 1))

Req 2 G (mode = cruise & speed = 1 & in = acc → X (speed = 1))

Req 3 G (mode = cruise & in = brake → X (mode = disengaged))

Req 4 G (mode = cruise & in = gas → X (mode = disengaged))

Requirement 1 states that when the cruise controller is in "cruise" mode and the vehicle's speed is within the designated cruise range, the speed should remain within the cruise range even if the vehicle begins to go uphill. Requirement 2 specifies that when the cruise controller is in "cruise" mode and the speed is within the cruise range, the speed should remain within the cruise range even if the vehicle starts going downhill. In other words, the cruise control function is expected to maintain the set cruise speed regardless of the vehicle's incline. Requirement 3 states that when the vehicle is in "cruise" mode and the driver applies the brake, the cruise controller should be disengaged. Lastly, Requirement 4 specifies that when the vehicle is in "cruise" mode and the driver presses the gas pedal, the cruise controller should also be disengaged.

**A Three-Floor Elevator (8 Bit Abstracted):**

This section elaborates the detailed functionality of the three-floor elevator. The model has 38 states. Its output is an 8-bit vector. The input set comprises four inputs: {c1, c2, c3, tick}.

- **c1:** representing a call to the first floor;
- **c2:** representing a call to the second floor;
- **c3:** representing a call to the third floor;
- **Tick:** representing the clock-ticks indicating passage of time.

The output vector of the elevator is an 8-bit string: {w1, w2, w3, cl, stop, @1, @2, @3}. These bits represent different output values and are explained below:

- **w1:** represents a queued call to the first floor;
- **w2:** represents a queued call to the second floor;
- **w3:** represents a queued call to the third floor;
- **cl:** represents the fact that the elevator door is closed;
- **cl:** represents the fact that the elevator door is open;
- **stop:** represents the fact that the elevator is halted at a particular point in time;
- **stop:** represents the fact that the elevator is in motion at a particular point in time;
- **@1:** represents the fact that the elevator is on the first floor;
- **@2:** represents the fact that the elevator is on the second floor;
- **@3:** represents the fact that the elevator is at third floor.

There are three safety properties that are crucial to the functionality of the elevator. These are listed as LTL Formula:

Req 1 G (! stop → cl)

Req 2 G (stop & X (! stop) → X (! cl))

Req 3 G (stop & @1 & cl & in = c1 & X (@1) → X (! cl))

Req 4 G (! @ 1 &! @2 & !@3 → !stop)

Requirement 1 is a safety property that ensures the elevator door remains closed whenever the elevator is in motion. Requirement 2 specifies that when the elevator is stopped at a floor, its door should automatically open. Requirement 3 states that the elevator door can be opened by pressing the floor 1 button, but only when the elevator is stationary at floor 1. Finally, Requirement 4 defines that the elevator will remain in motion while traveling between floors.

**Experiment Design:**

The analysis of the two Systems Under Test (SUTs) can be broadly categorized into three groups: the Combined Category, the Learning Category, and the Model Checking Category. The first category, Combined Category, includes parameters that are relevant to both model checking and learning. The second category, Learning Category, contains measures that pertain specifically to the learning infrastructure. The third category, Model Checking Category, focuses on parameters related to model checking and the testing of Learning-Based Testing (LBT). In the following sections, we will examine each category in detail and explain the role of each measure.

**Combined Category:**

- **Total LBT Iterations:** LBT undergoes a series of iterations in each experiment before the model learning process can be considered complete. The total number of these iterations is recorded in this measure.
- **Single LBT Iteration Execution Time (Millisecond):** This parameter records the time taken by LBT to complete a single iteration, with the time measured in milliseconds (ms).
- **First Bug Reporting Time (Millisecond):** This measure records the time at which the first counterexample is encountered. After LBT generates a model, it is passed to the model checker for verification against the specified requirements. The first instance where the model fails to meet the specification is captured by this parameter.
- **Bug Discovery Duration:** This measure captures the time interval between the discovery of one bug and the next. It reflects the time lapse between these events and serves as an indicator of the effectiveness of each iteration in refining the system model. The values recorded for this measure provide insight into the progress of the learning process.
- **Length of the Counterexample:** This parameter stores the length of a counterexample returned by the model checker. Specifically, it records the length at the point when the model checker generates a counterexample during the verification process.
- **Total True Negatives:** A true negative is a counterexample correctly identified by the model checker, indicating that the model satisfies the specification. This measure records the total count of true negatives observed during a single experiment.
- **Total False Negatives:** A false negative is a counterexample identified by the model checker that is incorrect or does not actually violate the specification. This measure tracks the total number of such false negatives encountered during a single experiment.
- **Total Unique True Negatives:** Unique true negatives refer to counterexamples that are distinct and not repeated from previous iterations. In other words, each counterexample represents a new and unique path in the state space. The total count of these unique true negatives is recorded in the "Total Unique True Negatives" parameter.
- **Uniqueness in Generated True Negatives:** Uniqueness in generated true negatives is the ratio of the total unique true negatives to the total true negatives. This measure reflects the extent to which a model checker generates unique counterexamples, highlighting its ability to explore diverse paths in the state space without repetition.
- **Random Queries:** A random query is a membership query generated by a random string generator. This measure records the total number of random queries made during a single LBT experiment, providing insight into the exploratory behavior of the testing process.

- **LBT Execution Duration (Millisecond):** LBT execution duration refers to the total time required for the LBT framework to fully execute and learn a model. This value is measured in milliseconds.
- **Precision:** The precision of a model checker is determined by dividing the number of true negatives by the total number of counterexamples generated. The resulting value is recorded in this measure.
- **Efficiency:** This measure evaluates the efficiency of a model checker by dividing the time taken to report the first bug by the total LBT execution time. It reflects how quickly the model checker can detect the first bug. The resulting values are recorded in this parameter.

**Learning Category:**

- **Single Hypothesis Construction Time (Millisecond):** Single hypothesis time refers to the duration taken by LBT to generate a single hypothesis instance. This value is recorded in this measure, with the time unit being milliseconds.
- **Maximum Hypothesis Size:** Maximum hypothesis size refers to the size of the largest hypothesis generated during an LBT experiment. The resulting value is recorded in this measure.
- **Average Hypothesis Size:** Among the many hypotheses generated by LBT in a single experiment, the average hypothesis size provides a mean value for analyzing performance. This parameter stores the average size of the hypotheses produced during the experiment.
- **Internal Query:** This type of membership query requests an input string, where the condition is that the input string must be generated by the learning algorithm itself. The total count of such internal queries is recorded in this parameter.
- **External Query:** Input strings can be generated from sources other than the learning algorithm itself, such as the model checker or a random string generator. A membership query that requests an input string generated by these external sources is referred to as an external query. The total count of such external queries is recorded in this measure.

**Model Checking Category:**

- **Total MCQs (Model Checking Queries):** When LBT learns a model during an iteration, the model is subsequently sent to the model checker for verification against a specific formal property. The total number of times this verification process occurs is recorded in this measure.
- **Model Checking Time Over a Single Iteration (Milliseconds):** When the model checker receives the learned model from LBT and verifies it against the specification, the result of this verification is then returned to LBT. The amount of time taken for this verification process in a single iteration is referred to as the model checking time, and it is recorded in milliseconds.
- **Total Model Checking Time (Milliseconds):** This measure records the cumulative time spent verifying the model across all iterations. Upon completion of the learning process, it captures the total time taken for model checking throughout the entire experiment.

**Results and Discussion:**

We conducted a series of experiments with LBT to evaluate its performance using the SPIN model checker. The results obtained by integrating SPIN with LBT provide insights into how LBT performs with this new model checker, compared to its previous implementation with NuSMV and SAL. While LBT had previously been tested with both NuSMV and SAL,

our experiments with SPIN focused on the IKL model inference algorithm, one of the two algorithms used in prior studies (the other being DKL). We specifically tested IKL within the LBT framework. Figure 3 presents the results of various experiments conducted with this setup using IKL. For comparison purposes, the performance scores of NuSMV and SAL have been retained to contrast with the results achieved using the SPIN model checker.

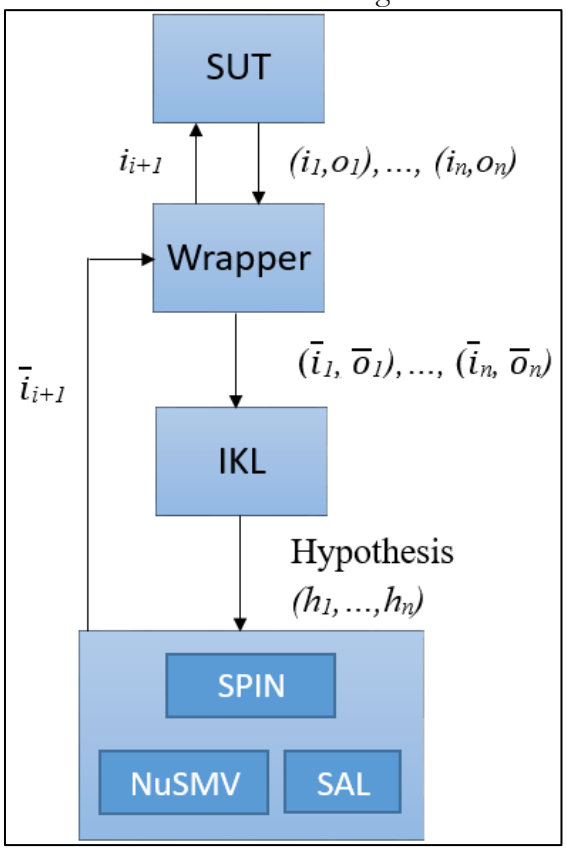


**Figure 3.** Experimental setup of Learning-based Testing with model checkers.

**Cruise Controller as SUT:**

In this section, we evaluate the results obtained from experiments using the SPIN model checker, comparing them with the earlier results from NuSMV and SAL. The learning algorithm used for these experiments is IKL. Various performance metrics were employed to assess SPIN's effectiveness, and through comparative analysis, we were able to determine which model checker performs best with IKL. Our focus is to observe how these parameters behave with the cruise control (CC) system as the SUT.

First, we examine the total LBT iterations. The results indicate that SAL requires the highest total number of iterations, followed by NuSMV, with SPIN requiring the fewest iterations. This suggests that SPIN facilitates IKL with the least number of iterations, enabling faster learning of the SUT (CC).

Next, we look at the time taken for a single LBT iteration. SAL takes significantly longer than NuSMV to perform a single iteration, and when we test SPIN on the same measure, it shows even lower values than NuSMV for three out of four specifications. This finding implies that SPIN is much more efficient than both SAL and NuSMV in terms of iteration time. Specifically, SPIN's implementation of the Vardi-Wolper framework [15] for automata-theoretic verification is more efficient than the SMT solver used by SAL and the SAT-solvers used by NuSMV for state-space exploration.

For the time to encounter the first counterexample, the pattern is similar to the single iteration time. SAL takes the longest to return a counterexample, which requires additional

time for the LBT framework to confirm whether it is a true negative. NuSMV, on the other hand, identifies the first counterexample much faster than SAL, but SPIN performs even better, providing the lowest time across all three-model checkers for this measure. In the fourth specification, SPIN's performance is almost identical to NuSMV, with a difference of just two milliseconds.

When we examine the length of the counterexample, we find that SAL's counterexamples are generally shorter and more consistent, indicating a potentially repetitive state-space exploration process. NuSMV's counterexamples are slightly longer and more variable, while SPIN's counterexamples fall between the lengths observed for SAL and NuSMV. Additionally, SPIN shows the lowest count of false negatives among the three-model checkers. This suggests that SPIN produces more correct counterexamples with fewer instances of unlearned parts of the model, especially when compared to SAL and NuSMV, where false negatives are more prevalent.

The count of unique true negatives (counterexamples) is higher for SAL than for NuSMV, indicating that SAL finds new counterexamples in each iteration rather than repeating the same ones. However, SPIN surpasses both SAL and NuSMV, producing the highest total number of unique counterexamples. This is indicative of SPIN's ability to explore the state space using multiple traversal routes, a feature facilitated by the use of active procedure types in Promela [11], which allows for the execution of all procedures with equal probability. This design forces LBT to build a new hypothesis in each iteration. Since uniqueness depends on the ratio of unique true negatives to total true negatives, SPIN outperforms both SAL and NuSMV in terms of this measure, suggesting that SPIN is more effective at generating unique counterexamples.

The total execution time of LBT is another critical measure. Here, SAL takes the longest to learn the SUT, while NuSMV takes less time, and SPIN achieves the lowest execution time across all four specifications. This is consistent with the finding that SPIN has the smallest hypothesis size, followed by NuSMV, with SAL requiring the largest hypothesis size.

Finally, we look at the precision and efficiency measures, which are computed as ratios of the above parameters. While all three-model checkers—SPIN, NuSMV, and SAL—demonstrate similar levels of efficiency in facilitating learning, SPIN performs better in terms of efficiency in the fourth specification of the SUT. In this specification, the ratio of the first bug reporting time to total LBT execution time is higher for SPIN than for the other two model checkers, indicating superior efficiency.

In conclusion, while all three-model checkers facilitate learning in the LBT framework, SPIN consistently outperforms NuSMV and SAL in several key metrics, including iteration time, bug reporting time, and counterexample uniqueness, making it a highly efficient and effective choice for LBT.

**Three-floor Elevator as SUT:**

In this sub-section, we analyze the results obtained from experiments using the three-floor elevator system as the SUT, comparing the performance of SPIN with the previously recorded results from NuSMV and SAL. By examining predefined parameters, we aim to assess the efficiency of SPIN as a model checker.

We begin by looking at the total LBT iterations. Notably, SAL consistently requires the highest number of iterations across all three specifications of the elevator system, compared to NuSMV. Higher iteration counts imply an increase in the number of MC (model checking) queries, which is evident from the results. In contrast, SPIN performs better by requiring fewer total LBT iterations. This suggests that SPIN produces more unique counterexamples, allowing LBT to explore the state space more effectively and discover previously unlearned

parts of the system with fewer iterations. Consequently, LBT reaches a complete model with SPIN in fewer iterations than with NuSMV or SAL.

Next, we analyze the time taken for a single LBT iteration. The values for NuSMV and SAL are similar, with SAL showing a slight improvement over NuSMV. However, SPIN significantly reduces the time required for each iteration. This indicates that SPIN is more efficient in terms of the time taken to perform state-space exploration, allowing LBT to complete each iteration faster than with the other two model checkers.

Moving on to the length of counterexamples, we observe that SAL generates the shortest counterexamples, NuSMV the longest, and SPIN's counterexamples fall in between. The shorter counterexamples produced by SAL suggest fewer missing loops or lasso structures, which accelerates state-space exploration. SPIN, on the other hand, generates both shorter and longer counterexamples, leading to slightly longer LBT execution times compared to SAL but shorter times than NuSMV. This is consistent with the counterexample length data, where SPIN's counterexamples exhibit a balance between the brevity of SAL and the lengthier ones from NuSMV.

For the uniqueness of counterexamples, NuSMV generates the least number of unique counterexamples, as its counterexamples tend to be reused over multiple iterations, reducing their uniqueness. SAL produces slightly more unique counterexamples, while SPIN outperforms both, generating the most unique counterexamples across all specifications. This indicates that SPIN is more effective at exploring different parts of the state space, making the LBT process more thorough.

Next, we consider the maximum and average hypothesis sizes. Both measures are lower when using SAL compared to NuSMV. With SPIN, the hypothesis sizes fall somewhere in between the two, reflecting the balance between the shorter counterexamples of SAL and the longer ones of NuSMV. Since SAL generates the smallest counterexamples, it leads to the smallest hypothesis sizes, while SPIN's performance is more balanced.

The number of model-checking queries is another key measure. SAL requires the highest number of queries, followed by NuSMV, with SPIN requiring the fewest queries. The number of queries is closely linked to the likelihood of encountering false negatives: as the number of queries increases, so does the probability of a false negative. This trend is reflected in the data, where SAL has the highest count of false negatives, followed by NuSMV, and SPIN shows the lowest count of false negatives.

In terms of precision, an analysis of the results for Specification 2 and Specification 3 reveals that SPIN achieves the best precision among the three-model checkers, followed by NuSMV and then SAL. SPIN's higher precision suggests that it more accurately identifies true negatives, resulting in fewer incorrect counterexamples.

Finally, we examine efficiency. SPIN proves to be the most efficient model checker in two out of three specifications, with NuSMV coming in second. This is particularly evident in the way SPIN achieves a balance of fast iteration times, fewer queries, and higher precision compared to SAL and NuSMV, making it the most efficient option overall for LBT.

In conclusion, SPIN consistently outperforms both NuSMV and SAL across multiple metrics, including total LBT iterations, time per iteration, counterexample uniqueness, hypothesis size, and efficiency. These findings confirm that SPIN is a highly efficient model checker for LBT, providing faster and more precise results compared to the other two model checkers.

**Findings:**

**Faster State Space Exploration by SPIN (on-the-Fly):**

NuSMV takes less time to explore the state space compared to SAL. However, SPIN outperforms both by being the fastest model checker, thanks to its use of the nested depth-first search algorithm. This approach allows SPIN to efficiently explore the state space without

relying on extensive memory consumption, which is typically required for evaluating acceptance conditions in a Büchi automaton.

**More Unique Counterexamples:**

The counterexamples generated by SPIN exhibit a higher degree of uniqueness, which enables LBT to produce more refined hypotheses of the system after each iteration. As a result, LBT can learn the black-box model of the system more quickly and efficiently.

**Faster Convergence by SPIN:**

The length of counterexamples generated by SPIN is generally consistent across all specifications and, in most cases, does not exceed the length of those produced by NuSMV. This factor contributes to a reduction in the overall LBT execution time when using SPIN.

**Conclusion**:

In this paper, we explore the feasibility of generating test cases for Learning-Based Testing (LBT) using the SPIN model checker by integrating two Systems Under Test (SUTs) with an incremental learning algorithm [25], [26]. To ensure consistency, we maintained the same experimentation platform across all tests. The results clearly demonstrate that when used in conjunction with the LBT framework, the SPIN model checker outperforms the other two model checkers, NuSMV and SAL. Specifically, SPIN enables LBT to converge faster, meaning that the architecture is able to generate intermediate hypotheses, which are refined after each iteration. This contrasts with NuSMV and SAL, which do not produce the same level of efficiency. The primary difference lies in the way these model checkers generate the state space. SPIN's state space generation is more manageable due to its use of techniques such as syntax error reports, interactive simulation, and the verifier generator [38].

Furthermore, SPIN's counterexample-generating algorithm is more efficient than those used in NuSMV and SAL. This is due to SPIN's use of the nested depth-first search algorithm, which does not require large amounts of memory for evaluating acceptance conditions from the Büchi automaton [16]. As a result, its on-the-fly procedure makes state space exploration more efficient compared to the other two model checkers. Experimental results on all four specifications for model checking total time show the following values: for SPIN, the total time is 30.7, 35.54, 42.3, and 23.4 milliseconds, respectively, compared to NuSMV's 34.12, 39.5, 43.5, and 29.8 milliseconds, and SAL's 195.73, 209.69, 236.57, and 184.14 milliseconds. These initial results indicate that SPIN converges faster on smaller case studies, although further investigation is needed to assess its performance on larger case studies.

The counterexamples produced by SPIN also exhibit a greater degree of uniqueness, allowing LBT to generate more refined hypotheses of the system after each iteration. This, in turn, enables faster learning of the black-box model of the system. Overall, the results demonstrate that SPIN is more efficient than NuSMV and SAL in the context of explicit state model checking within the LBT framework.

**Acknowledgement:** My heartfelt gratitude to my supervisor Dr. Muddassar Azam Sindhu and co-supervisor Dr Shafiq Ur Rehman, who proved to be an embodiment of excellent mentorship, empowering me by polishing my ability and skills with the huge repository of knowledge he proudly possesses.

**Author's Contribution:** Iram Jaffar analyzed the problem, implemented the proposed solution, performed experiments, and did the write-up.

Muddassar Sindhu defined the problem, supervised the research student in implementing, designing, and running experiments, and refined the write-up.

Shafiq ur Rehman defined the problem and co-supervised it during problem solution and evaluation.

**Conflict of Interest:** There exists no conflict of interest for publishing this manuscript in IJIST.

**Project Details:** This research was not conducted as a result of a project.

**References:**


[1] P. M. Jacob and M. Prasanna, "A Comparative analysis on Black box testing strategies," 2016 International Conference on Information Science (ICIS), Kochi, 2016, pp. 1-6. doi: 10.1109/INFOSCI.2016.7845290

[2] M. S. Reorda, "In-field test of safety-critical systems: is functional test a feasible solution?," 2015 16th Latin-American Test Symposium (LATS), Puerto Vallarta, 2015, pp. 1-2. doi: 10.1109/LATW.2015.7102528

[3] J. Gaur, A. Goyal, T. Choudhury, and S. Sabitha, "A walk through of software testing techniques," 2016 International Conference System Modeling & Advancement in Research Trends (SMART), Moradabad, 2016, pp. 103-108. doi: 10.1109/SYSMART.2016.7894499

[4] M. Nouman, U. Pervez, O. Hasan, and K. Saghar, "Software testing: A survey and tutorial on white and black-box testing of C/C++ programs," 2016 IEEE Region 10 Symposium (TENSYMP), Bali, 2016, pp. 225-230. doi: 10.1109/TENCONSpring.2016.7519409

[5] A. Sabbaghi and M. R. Keyvanpour, "State-based models in model-based testing: A systematic review," 2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI), Tehran, 2017, pp. 0942-0948. doi: 10.1109/KBEI.2017.8324934

[6] Meinke, Karl & Niu, Fei & Sindhu, Muddassar. (2011). Learning-Based Software Testing: A Tutorial. Communications in Computer and Information Science. 336. 10.1007/978-3-642-34781-8 16.

[7] I. Jaffar, M. Usman and A. Jolfaei, "Security Hardening of Implantable Cardioverter Defibrillators," 2019 IEEE International Conference on Industrial Technology (ICIT), 2019, pp. 1173-1178, doi: 10.1109/ICIT.2019.8755126.

[8] E. M. Clarke, Model Checking. London: The MIT Press, 2000, p. 144.

[9] N. Deb, N. Chaki and A. Ghose, "ToNuSMV: A Prototype for Enabling Model Checking of Models," 2016 IEEE 24th International Requirements Engineering Conference (RE), Beijing, 2016, pp. 397-398. doi: 10.1109/RE.2016.62

[10] Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verication. Proc. 1st Symposium on Logic in Computer Science, Cambridge, pp. 322-331, 1986

[11] Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth-first search. In: The Spin Verication System, DIMACS/32, American Mathematical Society, pp. 23-32. 1996

[12] Amir Pnueli. The temporal logic of programs. 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), pages 4657, 1977.

[13] Jamil, Abid & Arif, Muhammad & Abubakar, Normi & Ahmad, Akhlaq. (2016). Software Testing Techniques: A Literature Review. 177-182. 10.1109/ICT4M.2016.045.

[14] Ehmer, Mohd & Khan, Farmeena. (2012). A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. International Journal of Advanced Computer Science and Applications. 3. 10.14569/IJACSA.2012.030603.

[15] AP Van der Meer, R Kherrazi, and M Hamilton. Using formal specification to support model based testing asdspec: a tool combining the best of two techniques. arXiv preprint arXiv:1403.7257, 2014.

[16] Laurie Williams. A (partial) introduction to software engineering practices and methods. NCSU CSC326 Course Pack, 2009(5):3363, 2008.

[17] M. Ben-Ari. 2008. Principles of the Spin Model Checker (1st. ed.).

[18] C. Baier, & J. P. Katoen, (2008). Principles of model checking. MIT press.

[19] A. Groce, W. Visser (2003) What Went Wrong: Explaining Counterexamples. In: Ball T., Rajamani S.K. (eds) Model Checking Software. SPIN 2003. Lecture Notes in Computer Science, vol 2648. Springer, Berlin, Heidelberg

[20] G. J. Holzmann. The model checker SPIN. IEEE Transactions on Software Engineering, 23:279–295, 1997.

[21] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: A survey. Software Testing Verification and Reliability, 19(3):215261, 2009.

[22] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. Software Testing, Verication and Reliability, 22(5):297312, 2012.

[23] A. A. Shah, 2017. Learning Based Testing Using SAL (Symbolic Analysis Laboratory) Model Checker. Mphil Thesis. Quaid-i-Azam Univeristy, Islamabad, Pakistan.

[24] W. Wang, X. Bao and T. Zhao, "A research for embedded system software accident mechanism," 2017 2nd International Conference on System Reliability and Safety (ICSRS), Milan, 2017, pp. 460-464. doi: 10.1109/ICSRS.2017.8272865

[25] S. Mubeen, H. B. Lawson, J. Lundb¨ack, M. G˚alnander and K. Lundb¨ack, "Provisioning of Predictable Embedded Software in the Vehicle Industry: The Rubus Approach," 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), Buenos Aires, 2017, pp. 3-9. doi: 10.1109/SER-IP.2017..1

[26] M. Boasson, "Modeling and simulation in reactive systems," Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems, Honolulu, HI, USA, 1996, pp. 27-34. doi: 10.1109/WPDRTS.1996.557434

[27] B. Zeng and L. Tan, "Test Reactive Systems with Buchi Automata: Acceptance Condition Coverage Criteria and Performance Evaluation," 2015 IEEE International Conference on Information Reuse and Integration, San Francisco, CA, 2015, pp. 380-387. doi: 10.1109/IRI.2015.64

[28] L. E. G. Martins and T. Gorschek, "Requirements Engineering for Safety Critical Systems: Overview and Challenges," in IEEE Software, vol. 34, no. 4, pp. 49-57, 2017. doi: 10.1109/MS.2017.94

[29] H. Hwang and Y. B. Park, "Safety - Critical Software Quality Improvement Using Requirement Analysis," 2017 International Conference on Platform Technology and Service (PlatCon), Busan, 2017, pp. 1-4. doi: 10.1109/PlatCon.2017.7883725

[30] A. M. Bakr, M. M. Fouda, M. Salama, A. K. Alsammak and H. Yahia, "Hazard analysis of real-time safety critical systems using hierarchical communicating real-time state machines formal model," 2017 12th International Conference on Computer Engineering and Systems (ICCES), Cairo, 2017, pp. 628-634. doi: 10.1109/ICCES.2017.8275381

[31] R. M. Hierons, "Testing from Partial Finite State Machines without Harmonised Traces," in IEEE Transactions on Software Engineering, vol. 43, no. 11, pp. 1033-1043, 1 Nov. 2017. doi: 10.1109/TSE.2017.2652457

[32] Tan, O. Sokolsky and I. Lee, "Specification-based testing with linear temporal logic," Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004., Las Vegas, NV, 2004, pp. 493-498

[33] I. Buzhinsky and V. Vyatkin, "Testing automation systems by means of model checking," 2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Limassol, 2017, pp. 1-7. doi: 10.1109/ETFA.2017.8247579

[34] E. Clarke, J. Sumit, L. Yuan & V. Helmut. (2002), "Tree-like counterexamples in model checking," Proceedings - Symposium on Logic in Computer Science. 19- 29. 10.1109/LICS.2002.1029814.

[35] C.W. Axelrod, (2011). "Applying lessons from safety-critical systems to security-critical software," 2011 IEEE Long Island Systems, Applications and Technology Conference, 1-6.

[36] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," in IEEE Transactions on Software Engineering, vol. 50, no. 4, pp. 911-936, April 2024, doi: 10.1109/TSE.2024.3368208.

[37] P. Long, & J. Zhao, "Equivalence, identity, and unitarity checking in black-box testing of quantum programs," in Journal of Systems and Software, 211, 2024, 112000.

[38] G. J. Holzmann, "The Model Checker SPIN," in IEEE Transactions on Software Engineering, vol. 23, no. 5, pp. 279-295, May 1997.