NUML International Journal of Engineering and Computing
Volume: 2 Issue: 1
https://numl.edu.pk/journals/nijec
Print ISSN: 2788-9629
E-ISSN: 2791-3465
DOI: 10.52015/nijec.v2i1.32

Learning-Based Testing Using SAL (Symbolic Analysis Laboratory) Model Checker

Anjum Ara Shah[a,*], Muddassar Azam Sindhu[a]

[a]Department of Computer Science Quaid-I-Azam University Islamabad, Pakistan (anjumara@cs.qau.edu.pk, masindhu@qau.edu.pk)

Submitted	Revised	Published
29-May-2023	09-July-2023	26-July-2023

Abstract

This paper studies learning-based testing (LBT) for reactive systems with different learning algorithms and model checkers. LBT is a technique that requires a learning algorithm to learn the models to generate test cases automatically. We have used the generic methodology of LBT to test reactive systems with two different model inference algorithms (i.e., IKL, DKL) and two different model checking tools (i.e., NuSMV, SAL). To investigate the feasibility of LBT, we integrated our SUTs with these algorithms in LBT and tested if LBT optimizes test generation with these algorithms. We tested our SUTs with Boolean data types to check the difference in the working of model inference and model checking algorithms which we analyzed experimentally. The results show that LBT works better with DKL and SAL. DKL is a recently proposed model inference algorithm, and SAL is the latest model checker on which the research is being carried out. DKL and SAL algorithms explore errors in reactive SUTs with the h LBT framework more quickly and efficiently.

Keywords: LBT, IKL, DKL, NuSMV, SAL, SUT

1. Introduction

Verification and testing are two important steps to guarantee the quality of software. Verification ensures that the system we have designed is performing the desired tasks. Testing is to ensure that the tasks our system has to perform are done correctly. Model-based software testing is a technique in which an abstract model of the software system generates test cases automatically. It was first proposed in 1996 [1]. The model-based testing encompasses

* Corresponding Author: Anjum Ara Shah (anjumara@cs.qau.edu.pk)

the following activities, (a) building a model, (b) devising test selection criteria, (c) generating operational test specifications ion from the test selection criteria, (d) generating tests, and (e) executing these tests on the SUT [2]. A Model is an abstraction of a system with which we can analyze the expected behavior of that system. The research focuses mainly on these research questions and will work on these questions to unveil the interesting possibilities of the LBT framework.

Q1: How will the test case generation process be affected if the hypotheses generated by the IKL algorithm or another algorithm are model checked with some other model-checking tool?

Q2: What will be the effect on the performance of the BT framework if we use another multi-bit automaton learning algorithm instead of the IKL automaton learning algorithm?

Experimenting with LBT reveals some interesting results in the applicability of LBT to different types of SUTs. The paper is organized as follows: Sections I and II introduce model-checking methods, how they have been applied for LBT or other testing methodologies, and related work. Section III covers LBT architecture, Section IV deals with grammatical inference methods, and Section V covers model checkers used to model check LBT. LTL and Counterexample are provided in sections VI and VII. Section VIII describes the experimental setup, section IX describes case studies, and Section X describes the outcomes of our research and their analysis. Section XI of the article concludes with contributions and future work.

2. Related Work

Learning-based testing (LBT) is a new methodology for automating iterative black-box software requirements testing. Model inference and model-checking techniques are combined in the process. However, several model inference optimizations are required to provide scalable testing for big systems. The IKL learning algorithm, which is an active incremental learning approach for deterministic Kripke structures, is described in this study. They explicitly demonstrate the correctness of IKL and explore the optimizations it contains to achieve testing scalability. It also analyses a black box heuristic for test termination based on IKL learning convergence [3]

The LBT is a specification-based testing incorporating the Black-box technique with model checking to verify and test procedural and reactive systems [4-6]. The fundamental idea behind the LBT architecture is to produce a wide-ranging set of test cases by deploying an algorithm for model checking with the help of an incremental model learning algorithm such as IKL and DKL. The verification of systems by using learning and model checking is known as counterexample-guided abstraction refinement (CEGAR) [7]. The LBT works by integrating three components, (a) target black-box SUT and (b) a formal requirement specification to be checked against SUT a (c) a learned model of SUT. Specification-based testing [8] involves the first two components, the target SUT and t formal requirement. The third component is the learned model, the learning aspect in LBT, which uses the heuristic method to generate test cases automatically. It learns the black box SUT by using those test cases as queries. The Basic Steps of LBT are [6]:

1) The test case is executed on the SUT first, and then the output is created. The input/output pair is sent into the learning algorithm, which generates a model.

2) the Model is validated against the requirement to obtain the Counterexample.

3) This is sent into the LBT and utilized as the next test case. The paper discusses two algorithms, IKL [6] and the DKL [9].

3. LBT Architecture

We will define LBT architecture by using Figure 1. The Mn is passed as input ip to the model checker and a requirement as a φ, where φ is a temporal formula. The φ will remain the same in a certain testing experiment. The model checker checks the model against the requirement specification and identifies a counterexample as an input sequence ip. We can check two types of φ on Mn ii: e.a safety formula and a liveness formula. If the φ is a safety property, the input ip will be a finite sequence ip = ip1,. ..., ipk. In the case of the liveness formula, the input ip may be a finite sequence or an infinite sequence. Infinite counterexamples can be represented as a sequence of abw where a is the finite pi. e. i.e., handle, and bw is the infinite part, i.e., the loop. In this case, the LBT architecture considers the handle and truncates the loop and considers only a single execution, i.e. $ip = ab$ or $ip = abb$. LBT uses a random input generator to generate an input sequence ip by carefully not repeating any previously generated input. The input sequence is constructed by the model checker when it encounters a counterexample, but when it does not find any counterexamples, the formula gets satisfied and returns true. We can get input ip from these sources and a new input sequence ip = ip1, ..., ipk . Figure 1 shows that if ip is constructed from the model checker (either SAL or NuS whichever ever we are using) then it is applied to the current model Mn to get the predicted output po = po1, ..., pok for our SUT which will be passed to oracle component to get a verdict. It is not possible to practice this step in the case of a random input generator as we do not know whether the input sequence is either a counterexample to the φ or not. In all these occurrences the input ip is delivered to the SUT to get the observed output sequence or actual output op = op1, ..., opk. The final step of LBT architecture is the oracle step. We can see that if we have got the $ifrom\ the$ m model checker and have a predicted output po then both actual op and predicted output po are passed to oracle. The oracle implements the Boolean equality test $op = po$. It can give three types of verdicts on this Boolean equality test. The verdicts are categorized as failing, warning, and passing. If the test returns true and ip was a finite test case, the oracle will council's use it because op is by construction a counterexample against φ. If the test proclaims to be true and the test case was pruned from an infinite test case, ip then it will be a weakened verdict leading to a warning. We can conclude on the fact that we have not seen any difference between observed behavior op and an incorrect behaviour po. It will be considered a potential error by the system tester. If $op \neq po$ or if no observed output op exists, then it becomes difficult for tor oracle to issue an immediate verdict. Both cases can be possible, it may be possible that op is a counterexample to the correctness of the formula φ or the syntactic structure of the formula φ is not bound to ip and op due to its simple structure. But since Mn+1 is automatically constructed in response to the output behaviour op. The model checking step will later confirm this op as an error in this case.

4. Learning Algorithms in LBT

We will consider two different learning algorithms proposed by different researchers to test with two other model checkers with incremental learning-based testing approaches and evaluate the results.

4.1 IKL

IKL [6], is a k-bit extension of Angluin's ID algorithm. It extends the DFA learning to learn DKS with k-bit output using the bit-slicing and lazy partitioning refinement technique. Multi-bit output is necessary for the practical testing of reactive systems because these systems are not bound to one-bit output [10].

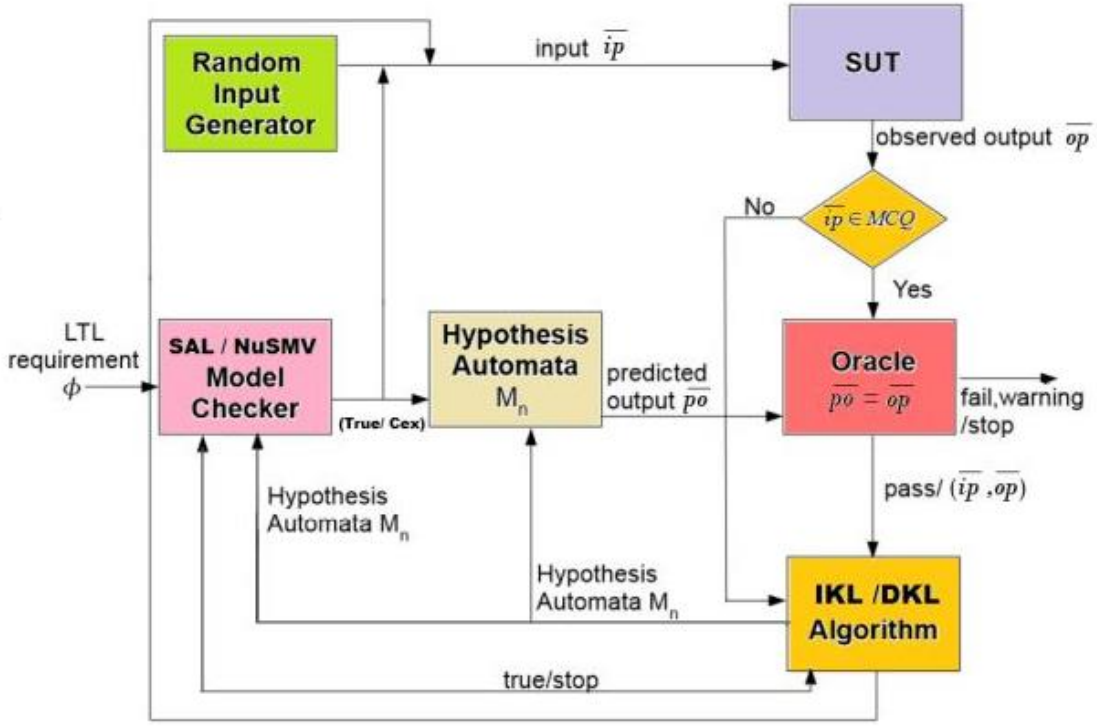


Figure 1 Learning-Based Testing

It starts from a k-bit DFA, slices it into a family of k-bit structures, and unlearns all the algorithms jointly by learning cooperatively. This cooperation between these structures is known as lazy learning, which supports frequent model checking during testing. Using lazy learning in IKL is to teach the 1-bit DFA family to produce a new k-bit hypothesis Kripke with maximum frequency. After inferring the k 1-bit automata, the sub-direct product is applied to assemble these into a single k-bit automaton. The minimization is conducted if the assembled Kripke is not in its optimal state space, and there seems to be a need to minimize the states to reduce the large state space, as the state space of assembled automaton could be very large. To further reduce the resultant Kripke, t applies L applies the generalized version (i.e., for Kripke structures) of a DFA minimization algorithm by Jhon Hopcroft proposed in 1971 [11].

4.2 DKL

The DKL [9] algorithm is an appendage of Kearns [12] algorithm with some technical changes that is an incremental learning algorithm that aims to learn deterministic Kripke structures [13]. The algorithm is designed to avoid/ prevent unnecessary state space blow-up for intermediate hypotheses constructed by IKL due to the sub-direct product. Another dimension of DKL is that it does not require a minimization procedure as IKL requires

for reducing the state space of the hypothesis. Even after executing a minimization procedure over the intermediate hypotheses generated by IKL, some unnecessary states are sometimes left in the hypotheses (i.e., the states that are not pr the target automaton). We have integrated both algorithms in asynchronous architecture, shown in Figure l Checking Algorithms in LBT.

In this section, we will study two widely used different model checkers. Model checkers are different by various properties, i.e., analysis techniques, modeling languages, and counterexample algorithms. All these techniques make us select the best-required model checker according to the system's needs under test.

4.3 NuSMV

NuSMV [14] is an SMV (symbolic model verifier) based model checker based on BDD (Binary Decision Diagram) [15]. The SMV only provided BBD-based symbolic model checking functionality, but the NuSMV provided us with the extended functionalities inherited from the previous version in many directions. The main functionality introduced in NuSMV was integrating the propositional satisfiability-based model-checking technique SAT [16]. Nowadays, SAT-based-based techniques are providing great success in the industry and have opened new research venues for researchers in model checking. The BDD and SAT have solved several problems, and due to this, both can be considered complementary model-checking techniques [17].

4.4 SAL

The SAL [18] is the latest model checker with blackboard architecture [19]. The SAL uses an intermediate language [19] which is the heart of this framework and serves as a target for translators to extract system descriptions for languages such as Java, Esteral, Verilog, etc., from the transition system. The main components that provide basic functions that can analyze a property in SA include validation tools based on model checking, theorem proving, invariant generation, counterexample generation, and abstraction with slicing [21–23]. The explicit state algorithms cannot manage larger state systems compared to new model-checking tools that follow symbolic state verification algorithms. The symbolic model checking gives the advantage of expressing a larger state space.

4.5 Linear Time Properties

The linear time properties define the trace of a Ts it possesses [8]. The notion of linear and safety properties was first used by Lamport in 1977 [24]. Properties fall into three categories, safety properties, liveness properties, and invariant. Safety property follows the idea that something bad will not occur during the execution. On the other hand, liveness property hopes that, eventually, something good will occur during the execution of Ts. A property satisfied liveness if it satisfies the idea that something good finally happens.

5. Counterexample

The process of model checking provides us the facility to achieve the correctness of complex systems by getting rid of generated counterexamples and finding errors because of the counterexample trace [25]. Analyzing the trace makes correcting the model or specification necessary to get the verification process done successfully. A counterexample may occur due to incorrect system modeling, which can be corrected by tracing the error and remodeling the system [26]. Another error trace may result due to faulty specification, which can be a false negative. This type of error can be corrected by analyzing the Counterexample and providing specifications against the model to be verified.

6. Experimental Setup

We conducted the experiments by running the whole LBT framework on a Linux machine with Core-i7-7700 with 16 GB of RAM. We evaluated two case studies, a cruise controller (CC) model and an elevator model on LBT with incremental learning algorithms IKL and DKL programmed Java. The Specifications were evaluated against some errors injected into the learning model. The errors were injected by changing the correct output bits or by mutating the transitions to make the model erroneous. Each type of experiment was executed ten times for that specification. (Since there is an element of randomness in LBT due to the random string generator (see [6]), each type of experiment was repeated ten times). The results are averaged to obtain precise values of the result parameters. The counterexamples found are used as test cases in the next iteration.

Case Studies (Reactive Systems)

The reactive systems are interactive, and nature drives control (or event). These systems need continuous interaction with the environment in which they are operating. Some examples of these systems are avionics systems, ticket and resource reservation systems, nuclear reactor systems, complex and lifesaving diagnostic systems, etc. By using formalism, these systems become convenient and admissible to automatic verification, which can be done by using model checkers or can be tested with automated testing. We will consider two examples of reactive systems as our case studies. The case studies are Vehicle cruise controller (5-Bit) and Three-Floor Elevator.

i. A Vehicle cruise controller (5-Bit): A cruise control (CC) is an automatic vehicle speed control system used to restrain the speed of a vehicle. By embedding cruise control, the driver gets the privilege of controlling a vehicle's throttle by automatic operation. It works effectively in steady traffic conditions and improves ease for the driver. A cruise control can be turned on/off explicitly and automatically (i.e., by pressing the brakes or the cruise on/off button). A simplified cruise controller can be modeled as a deterministic Kripke structure with inputs σ = brake, decelerate, accelerate, button, and an output vector of 5 bits. A model of such a cruise control system is presented in Figure 2.

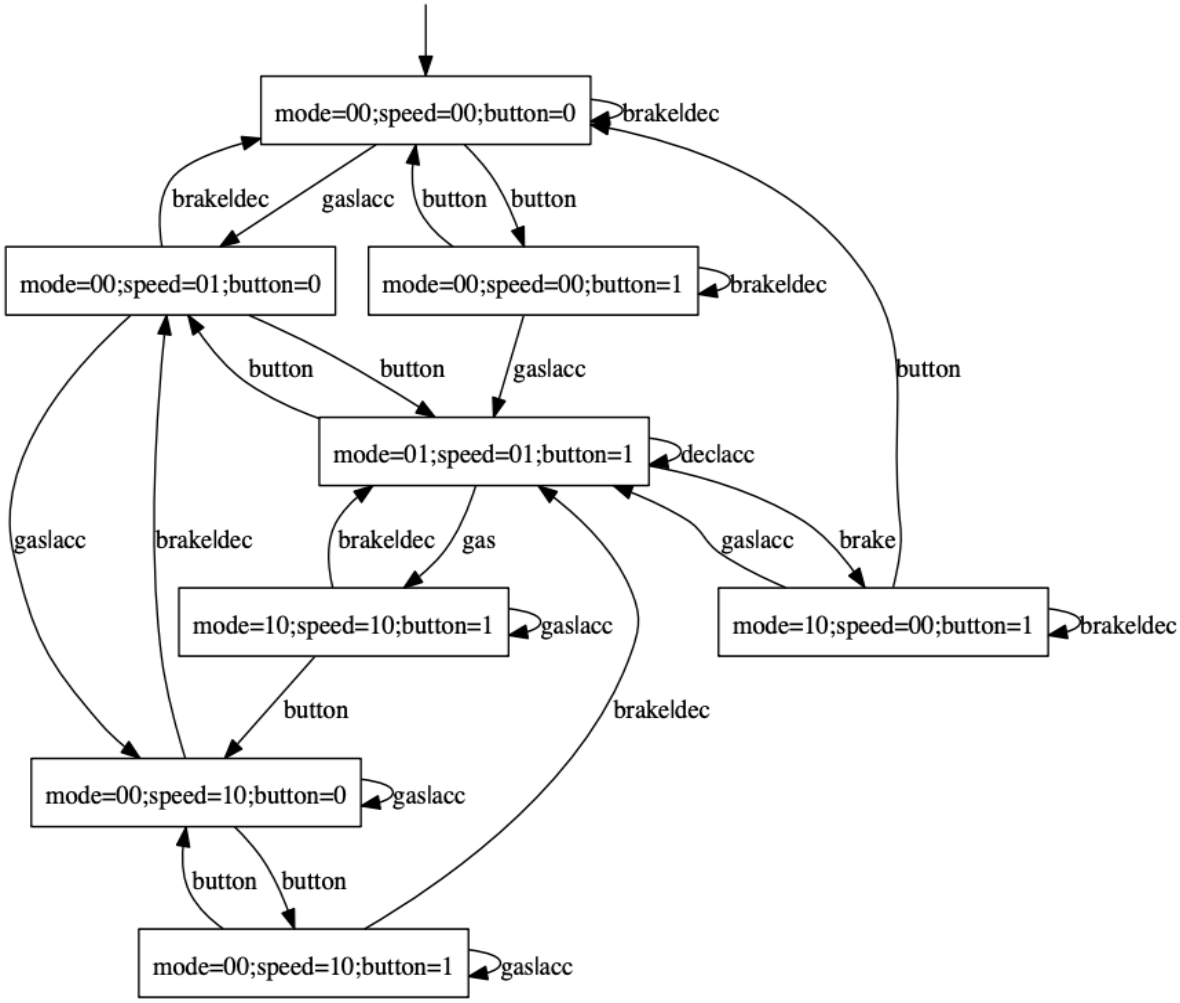
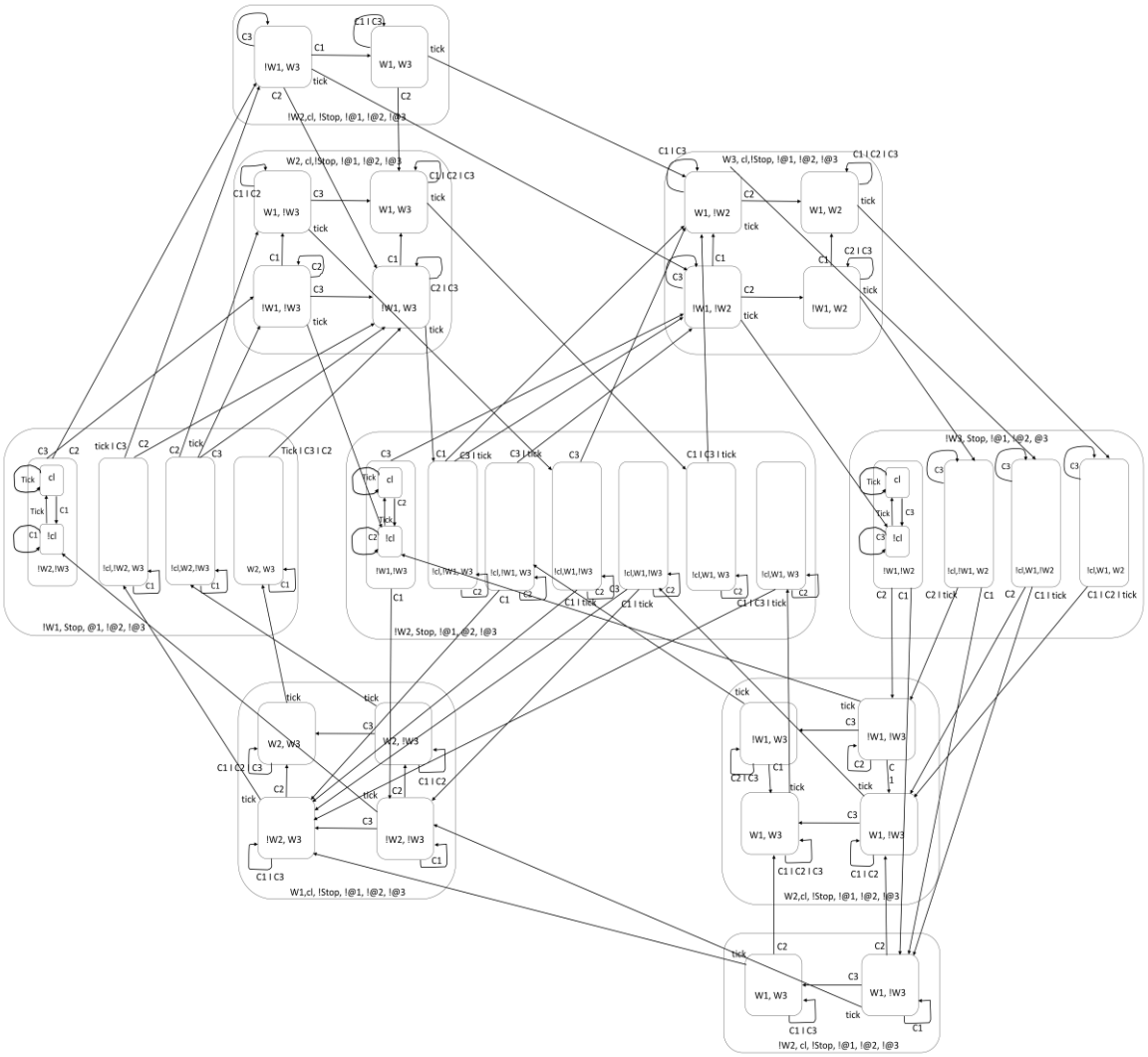


Figure 2 A 5-bit Cruise Controller Model

Table 1 Cruise Controller LTL Requirements Specification

Specification	LTL Formula
S1	G (mode = cruise & speed = 1 & in = dec → X(speed = 1))
S2	G (mode = cruise & speed = 1 & in = acc → X(speed = 1))
S3	G (mode = cruise & in = brake → X (mode = disengaged))
S4	G (mode = cruise & in = gas → X (mode = disengaged))

ii. A Three-Floor Elevator (8-Bit): The Three-Floor elevator model is another embedded safety-critical system. A model of such a system is presented in Figure 3, represented as a hierarchical states chart. The model has 38 states, an 8-bit output vector, and a set of 4 inputs: c1, c2, c3, and tick.



7.

Figure 3 3-Floor Elevator

Table 2 Elevator LTL Requirements Specification

Specification	LTL Formula
S1	G (! stop → cl)
S2	G (! stop & X(stop) → X (! cl))
S3	G (stop & @1 & cl & in=c1 → X (@1 & !cl))

10. Results and Evaluation

This section of the chapter deals with the findings we observed in our experiments by testing the working of two inference algorithms with two different model checkers. The overall results of the experiments are given below. By experimenting with two model checkers with different state exploration platforms, we found that NuSMV takes much less time in state space conversion to OBDD and explores it in less time. But in the case of SAL, we observed that the algorithm SAL used to convert the model takes a lot of time to transform the state

space in bit vectors and to explore them. It means the time measure is not considered a priority for SAL algorithms which deteriorates the performance of SAL. We observed that SAL works better in the context of counterexample generation. The counterexample generation algorithm of SAL mostly yields smaller counterexamples without lasso, which are easy to interpret and execute on practical systems for testing and helps to find the bug in the system. SAL worked well with our cruise controller and elevator model with IKL and DKL. The results of the case studies observed at SAL works well in the context of time and model-checking algorithm with highly abstracted models.

Table 3 Cruise Controller Results S1

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	1.31	1.55	6.23	7.3
|Hyp| max	8	8	51.9	59.5
|Hyp| avg	5.1	5.08	16.92	18.27
BQs	1808.4	1778.2	18775	22747
RQs	153649	4.7	1E+06	16131
LBT Iterations	8	8	30.4	29.2
MCQs	8	8	30.4	29.2
CEs Uniqueness	0.28	0.45	0.12	0.15
|CE|	7.02	5	7.03	5
MCtime iter (ms)	52.76	626.38	34.12	195.74
MCtime total (ms)	422.1	5011	1003.8	5636.1
CEtime first (ms)	99.9	918.4	213.8	2060.2
CEtime avg (ms)	162.69	1211.6	92.55	412.75
LBT time iter (ms)	97.26	843.18	45.09	203.36
LBT time total (ms)	778.1	6745.4	1532.2	5873.9
Precision	0.795	0.94	0.767	0.799

Table 4 Cruise Controller Results S2

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	1.59	1.14	8.63	15.06
|Hyp| max	8	8	71.2	77.7
|Hyp| avg	5.06	5.05	19.68	20.63
BQs	1985.3	1929.7	28301.5	30744
RQs	18991.1	390674.9	373779.9	1005817
LBT Iterations	8	8	32.6	37.4
MCQs	8	8	32.6	37.4

CEs Uniqueness	0.23	0.23	0.14	0.13
|CE|	7	5	7	5
MCtime iter (ms)	55.04	636.53	39.51	209.69
MCtime total (ms)	440.3	5092.2	1248.4	7606.8
CEtime first (ms)	57.3	878.7	340	1491.8
CEtime avg (ms)	103.55	1174.8	147.33	352.52
LBT time iter (ms)	65.93	831.74	50.46	228.41
LBT time total (ms)	527.4	6653.9	1760.4	8602.7
Precision	0.963	1	0.603	0.879

We also observed that NuSMV tends to work considerably well in the context of time and model-checking algorithms with Boolean and integer data types. SAL uses an SMT solver, Yices 2.0, which was presented in the SMT competition in 2009 and was the winner of that competition due to its type-checking and bit vector-solving strategy from the experiments benchmarked the performance of NuSMV, which uses an SAT solver and found that SAL does not provide us reliable performance in the context of time. We observed that SAL is in its evolution period compared to NuSMV, which has matured with time. There are many venues still not explored by the SAL community.

Table 5 Cruise Controller Results S3

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	1.04	0.76	8.39	25.26
|Hyp| max	7	7	86.3	99.7
|Hyp| avg	4.5	4.51	21.12	25.26
BQs	1379.2	1399	34671	28623
RQs	11534	227670	1038.3	347339.4
LBT Iterations	7	7	25.6	29.9
MCQs	7	7	25.6	29.9
CEs Uniqueness	0.65	0.43	0.39	0.29
|CE|	6.52	5	5.87	5
MCtime iter (ms)	47.34	490.26	43.48	236.57
MCtime total (ms)	331.4	3431.8	1145.3	6816.1
CEtime first (ms)	61.1	724.9	286.6	1399.4
CEtime avg (ms)	92.9	1981.6	118.65	598.28
LBT time iter (ms)	56.49	613.17	51.97	246.01
LBT time total (ms)	395.4	4292.2	1376.4	7180.7
Precision	1	0.96	0.614	0.666

They are working on it to make SAL better, but it needs better conversion and state space exploration algorithms. The NuSMV does not consider the model checking of a single instance of state in its set of states in the model, which forces us to make some amendments to our inference algorithm. We do not have to provide a different solution for this case with SAL. It simply model checks the state space of the single-state model. We observed that the inference algorithms differ in learning strategy, leading us to think about the algorithms we use for the LBT framework.

The DKL provides equal iterations in the case of both the model checkers equal to several states of the model to generate the counterexamples by which the biases in model checking queries we faced with IKL vanishes. The inference algorithm IKL learns the model in components by taking the sub-direct product of the model and using tables as a data structure to save the intermediate units in memory which takes a lot of time and demands huge memory space. Due to this, the state explosion problem often occurs in the n case of large and com Kripke structures used by the learning algorithm. The DKL does not learn the model in components and learns an optimal number of times. The hypothesis it constructs also does not exceed the state size. It learns the model in a fixed number of iterations that are equal ta o number of states of the model. The DKL uses an es tree data structure to save the model in memory. Due to this, the state space explosion problem does not occur in learning for even large and complex Kripke structures. Due to a smaller number of iterations and the data structure it follows, DKL takes less time to learn than IKL.

Table 6 Cruise Controller Results S4

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	1	1.186	5.762	5.894
|Hyp| max	7	7	33.1	47.9
|Hyp| avg	4.414	4.429	14.277	16.308
BQs	1216.7	1202.1	21324	21145
RQs	90770	126236	1285.7	70.7
LBT Iterations	7	7	17.5	20.9
MCQs	7	7	17.5	20.9
CEs Uniqueness	0.806	0.545	0.778	0.319
|CE|	5.783	5	5.978	5
MCtime iter (ms)	44.386	486.171	29.810	184.142
MCtime total (ms)	310.7	3403.2	521.5	3881.5
CEtime first (ms)	132.5	845.1	473.2	2711.6
CEtime avg (ms)	165.5	934.45	325,936	1085.91
LBT time iter (ms)	94.057	556.071	35.689	190.090
LBT time total (ms)	658.4	3892.5	624.4	4014.9
Precision	0.748	0.637	0.261	0.488

Table 7 Elevator Results S1

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	8.74	6.78	1178	453.81
|Hyp| max	38	38	846.5	854.1
|Hyp| avg	20.18	20.17	102.97	114.95
BQs	34860.6	34171.1	448861	287200
RQs	3080494	4334029	5169.4	1955.6
LBT Iterations	38	38	159.3	106.6
MCQs	38	38	159.3	106.6
CEs Uniqueness	0.109	0.032	0.023	0.019
|CE|	4.57	2	3.74	2.01
MCtime iter (ms)	256.63	2085.23	81.42	282.43
MCtime total (ms)	9752	79238.8	11172.8	26934.3
CEtime first (ms)	42.7	176.6	16	175.1
CEtime avg (ms)	1890.4	4230.5	1262.59	744.91
LBT time iter (ms)	1837.05	4121.49	1259.48	736.29
LBT time total (ms)	69808	156616.5	309991	102253
Precision	0.992	1	0.997	0.989

Table 8 Elevator Results S2

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	8.06	7.75	517.44	259.25
|Hyp| max	38	38	813.5	856.6
|Hyp| avg	20.25	20.24	99.3	147.38
BQs	33412	34015	344776	229303
RQs	2E+06	3E+06	2520.1	2537.8
LBT Iterations	38	38	129	86.4
MCQs	38	38	129	86.4
CEs Uniqueness	0.368	0.085	0.09	0.049
|CE|	10.15	3	6.95	3
MCtime iter (ms)	242.36	2221	79.12	422.9
MCtime total (ms)	9209.8	84399	9430.3	34114
CEtime first (ms)	18.9	188.1	18.2	96
CEtime avg (ms)	1025.3	3678.8	599.62	691.69
LBT time iter (ms)	956.08	3476.1	596.59	682.22

LBT time total (ms)	36331	132090	100919	61532
Precision	0.932	0.951	0.994	0.986

240

Table 1 Elevator Results S3

Parameters	DKL		IKL	
	NuSMV	SAL	NuSMV	SAL
HCtime iter (ms)	0.34	0.26	14.87	7.62
|Hyp| max	7	7	9.1	9.7
|Hyp| avg	4.57	4.57	5.2	6.03
BQs	518.2	513.8	10044	6982.8
RQs	700466	93711	145.3	8.2
LBT Iterations	7	7	13.7	11.5
MCQs	7	7	13.7	11.5
CEs Uniqueness	0.167	0.167	0.086	0.12
|CE|	5	4	4	4
MCtime iter (ms)	35.97	383.11	10.26	101.83
MCtime total (ms)	251.8	2681.8	137.9	1170.4
CEtime first (ms)	21.1	188.9	27.3	205.6
CEtime avg (ms)	478.18	489.53	29.62	140.66
LBT time iter (ms)	411.77	432.93	25.22	109.48
LBT time total (ms)	2882.4	3030.5	367	1260.1
Precision	1	1	0.847	0.800

11. Conclusions and Future Work

245 The analysis showed that SAL is good at generating the Counterexample, so its precision is higher than NuSMV. As in learning-based testing, we are more concerned about the quality of counterexample generation; we consider the algorithm of SAL better than NuSMV because it finds a different counterexample more often and it also possesses the distinction that it generates the Counterexample without loop. By getting a counterexample without a loop we do not have to truncate the loop part of the Counterexample. We know that the inference 250 algorithm IKL learns the model in components while DKL learns the model an optimal number of times. DKL takes less time to learn than IKL due to a smaller number of iterations and the data structure it follows. If we integrate SAL as a model-checking tool and DKL as a model inference algorithm, we can get the best of the LBT framework. IKL learns the model by taking the sub-direct product of the model and uses tables as a data structure to save the intermediate units in memory which takes a lot of time and demands huge memory space. Due to this, 255 the state explosion problem occurs in the case of large and complex systems.

On the other hand, DKL does not learn the model in components and learns an optimal number of times. The hypothesis it constructs also does not exceed the state size. It learns the model in a fixed number of iterations equal to several states of the model and uses a tree data structure to save the model in memory. Due to this, the state space explosion problem does not occur for even large and complex systems. The analysis showed that SAL is good at generating the Counterexample, so its precision is higher than NuSMV. As in learning-based testing, we are more concerned about the quality of counterexample generation; we consider the algorithm of SAL better than NuSMV because it finds a different counterexample more often and it also possesses the distinction that it generates the Counterexample without loop. By getting a counterexample without a loop we do not have to truncate the loop part of the Counterexample. The only drawback of SAL is that the algorithm of SAL needs more effort in the context of time. The research can be extended to writing translators of one model-checking language to other model-checking languages to ease the task of conversion to different model checkers. It can be made possible to produce the state chart from the model and present the Counterexample in the state chart so that the tester can understand where the bug resides. The task could be to convert state charts to models and vice versa. In incremental learning LBT, whenever a learned hypothesis is generated and the model checked it becomes difficult to trace the error transitions in the model according to the Counterexample. It can be done possible to make a framework to identify the error transitions from the learned model and generate a visualization to make the image clear.

References

[1] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking, 1996.

[2] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. Software Testing, Verification, and Reliability, 22(5):297–312, 2012.

[3] Sindhu, Muddassar A. "An Efficient Model Inference Algorithm for Learning-based Testing of Reactive Systems." arXiv preprint arXiv:2008.06268 (2020).

[4] Karl Meinke. Automated black-box testing of functional correctness using function approximation. ACM SIGSOFT Software Engineering Notes, 29(4):143–153, 2004.

[5] Karl Meinke and Fei Niu. A learning-based approach to unit testing of numerical software. In ICTSS, pages 221–235. Springer, 2010.

[6] Karl Meinke and Muddassar A Sindhu. Incremental learning-based testing for reactive systems. In International Conference on Tests and Proofs, pages 134–151. Springer, 2011.

[7] Edmund Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. Springer, 2003.

[8] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. Principles of model checking. MIT Press, 2008.

[9] Rabia Mazhar and Muddassar Azam Sindhu. Dkl: an efficient algorithm for learning deterministic Kripke structures. Acta Informatica, 58(6):611–651, 2021.

[10] Muddassar Sindhu. Algorithms and Tools for Learning-based Testing of Reactive Systems. Ph.D. thesis, KTH Royal Institute of Technology, 2013.

[11] J Hartmanis and J E Hopcroft. An overview of the theory of computational complexity. Journal of the Association for Computing Machinery, 18(3):444–475, 1971.

[12] Michael J Kearns and Umesh Vazirani. An introduction to computational learning theory. MIT Press, 1994.

[13] Paul Ammann and Rupak Jeff Offutt, Majumdar. Introduction to Software Testing. Cambridge University Press (2008). ISBN: 978-0-521- 88038-1.£ 32.99. 322 pp. Hardcover. Br Computer Soc, 2010.

[14] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In International conference on computer aided verification, pages 495–499. Springer, 1999.

[15] Sheldon B. Akers. Binary decision diagrams. IEEE Transactions on Computers, (6):509–516, 1978.

[16] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P Kurshan, and Kenneth L McMillan. An analysis of sat-based model checking techniques in an industrial environment. In Advanced Research Working Conference on Correct Hardware Design and Verification Methods, pages 254–268. Springer, 2005.

[17] Mukul R Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. International Journal on Software Tools for Technology Transfer, 7(2):156–173, 2005.

[18] Dr. John Rushby. SymboAnalysisysis Laboratory (sal), 2013.

[19] D Rudenko and A Borisov. An overview of blackboard architecture application for real tasks. In Scientific Proceedings Of Riga Technical University, Ser, volume 5, pages 50–57, 2007.

[20] Leonardo De Moura, Sam Owre, and Natarajan Shankar. The Sal language manual. Computer Science Laboratory, SRI International, Menlo Park, Tech. Rep. CSL-01-01, 2003.

[21] Gregoire Hamon, Leonardo De Moura, and John Rushby. Generating ´ efficient test sets with a model checker. In Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on, pages 261–270. IEEE, 2004.

[22] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Munoz, Sam ´ Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saıdi, Natarajan Shankar, et al. An overview of Sal. In Proceedings of the 5th NASA Langley Formal Methods Workshop. Williamsburg, VA, 2000.

[23] Jin Song Dong and Huibiao Zhu. Formal Methods and Software Engineering: 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010, Proceedings, volume 6447. Springer, 2010.

[24] Leslie Lamport. Proving the correctness of multiprocess programs. IEEE Transactions on software engineering, (2):125–143, 1977.

[25] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. SPIN Workshop on Model Checking of Software, pages 121–136, 2003.

[26] Bernhard Steffen. An abstract framework for counterexample analysis in active automata learning. JMLR: Workshop and Conference Proceedings, (1993):79–93, 2014.