Active Learning of Abstract System Models from Traces using Model Checking

1st Natasha Yogananda Jeppu
University of Oxford
Oxford, UK
natasha.yogananda.jeppu@cs.ox.ac.uk

2nd Tom Melham
University of Oxford
Oxford, UK
tom.melham@cs.ox.ac.uk

3rd Daniel Kroening
Amazon, Inc
London, UK
daniel.kroening@magd.ox.ac.uk

***Abstract*—We present a new active model-learning approach to generating abstractions of a system implementation, as finite state automata (FSAs), from execution traces. Given an implementation and a set of observable system variables, the generated automata admit all system behaviours over the given variables and provide useful insight in the form of invariants that hold on the implementation. To achieve this, the proposed approach uses a pluggable model learning component that can generate an FSA from a given set of traces. Conditions that encode a completeness hypothesis are then extracted from the FSA under construction and used to evaluate its degree of completeness by checking their truth value against the system using software model checking. This generates new traces that express any missing behaviours. The new trace data is used to iteratively refine the abstraction, until all system behaviours are admitted by the learned abstraction. To evaluate the approach, we reverse-engineer a set of publicly available Simulink Stateflow models from their C implementations.**

***Index Terms*—active model learning, execution traces, system abstraction, software model checking**

I. Introduction

Automaton inference from trace data is an established method for automated generation of system abstractions. Modern passive learning algorithms also infer guards and operations on system variables from the trace data, yielding symbolic models [1]–[4]. But the behaviours admitted by these models are, of course, limited to only those manifest in the traces. On the other hand, active learning algorithms can, in principle, generate exact system models [5], [6]. But when used in practice to learn symbolic abstractions, these algorithms suffer from high query complexity and can only learn models with transitions labelled by simple predicates, such as equality/inequality relations [7]–[9]. A survey of the related work is provided in [10].

We present a new active learning approach that combines a black-box analysis, in the form of model learning from traces, with a white-box analysis, in the form of software model checking [11]. The model learning component can be any algorithm that can generate an automaton that accepts a given set of system execution traces. Model checking is used to evaluate the degree of completeness of the learned automaton and identify any missing behaviours. This evaluation procedure operates at the level of the abstraction and not individual system traces, unlike query-based active learning algorithms, and therefore can be easily implemented using existing model checkers. Exact details of the procedure are discussed in Section III.

The procedure to evaluate degree of completeness of the learned model yields a set of new traces that exemplify system behaviours identified to be missing from the model. New traces are used to augment the input trace set for model learning and iteratively generate an extended abstraction that covers missing behaviours. Given a model learning algorithm that can infer symbolic abstractions from trace data, such as [4], the approach can learn models that are more expressive than the abstractions learned using existing active learning algorithms.

II. Background

The system for which we wish to generate an abstraction is represented as a tuple $\mathcal{S} = (X, X', R, \mathit{Init})$. $X = \{x_1, \ldots, x_m\}$ is a set of observable system variables over some domain D. We simplify the presentation by assuming all variables have the same domain. The set $X' = \{x'_1, \ldots, x'_m\}$ contains corresponding primed variables, which represent an update to the unprimed variable after a discrete time step. The transition relation $R(X, X')$ describes the relationship between x_i and x'_i for $1 \leq i \leq m$ and is represented using a characteristic function, i.e., a Boolean-valued expression over $(X \cup X')$. The set of initial system states is represented using its characteristic function $\mathit{Init}(X)$.

A *valuation* $v : X \rightarrow D$ maps the variables in X to values in D. An *observation* at time step t is a valuation of the variables at that time, and is denoted by v_t. A *trace* is a sequence of observations over time; we write a trace σ with n observations as a sequence of valuations $\sigma = v_1, \ldots, v_n$. We define an execution trace or *positive* trace for $\mathcal{S}$ as a trace $\sigma = v_1, \ldots, v_n$ that corresponds to a system execution path, i.e., $(v_t, v_{t+1}) \models R$ for $1 \leq t < n$ and there exists a valuation $v' \models \mathit{Init}$ such that $(v', v_1) \models R$. A *negative* trace is a trace that does not correspond to any system execution path. We represent the set of execution traces by $\mathit{Traces}_X(\mathcal{S})$.

The active learning algorithm learns a system model as a finite state automaton (FSA). Our abstractions are represented symbolically and feature predicates on the transition edges, such as the abstraction in Fig. 1, and therefore extend FSAs to operate over infinite alphabets. We represent the learned abstraction as a non-deterministic finite automaton (NFA) $\mathcal{M} = (\mathcal{Q}, \mathcal{Q}^0, \Sigma, F, \delta)$ over an infinite alphabet Σ, where $\mathcal{Q}$ is a finite set of states, $\mathcal{Q}^0 \subseteq \mathcal{Q}$ are the initial states, $F \subseteq \mathcal{Q}$ is the

This research was funded in part by the Semiconductor Research Corporation, Task 2707.001, and Balliol College, Jason Hu scholarship.

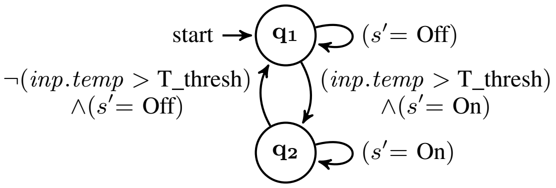

Fig. 1: Generated abstraction modeling operation mode switches for a Home Climate-Control Cooler system.

set of accepting states, and $\delta : \mathcal{Q} \times \Sigma \to \mathcal{P}(\mathcal{Q})$ is the transition function. The alphabet Σ corresponds to the set of valuations for variables in X, i.e., $\Sigma = (X \longrightarrow D)$.

The NFA admits a trace $\sigma = v_1, \ldots, v_n$ if there exists a sequence of automaton states $q_1, \ldots, q_{n+1}$ such that $q_1 \in \mathcal{Q}^0$ and $q_{i+1} \in \delta(q_i, v_i)$ for $1 \leq i \leq n$. Any finite prefix of a system execution trace σ is also an execution trace. Thus, if the generated NFA admits σ, it must also admit all finite prefixes of σ. In other words, the language of the automaton, $L(\mathcal{M})$, must be *prefix-closed*. All states of our automaton are accepting, i.e., the NFA rejects traces by running into a 'dead end'.

III. Active Learning of Abstract System Models

The approach uses a pluggable model learning component to generate models from traces. Our requirement for this component is simple: given a set of execution traces $\mathcal{T}$, the component returns an NFA $\mathcal{M}$ that accepts (at least) all traces in $\mathcal{T}$.

To evaluate the degree of completeness of the set of traces, we use the structure of the NFA $\mathcal{M}$ to extract conditions that can be checked against the system implementation. The conditions collectively encode the following completeness hypothesis: for any transition available in the system defined by the transition relation R, there is a corresponding transition in $\mathcal{M}$. The hypothesis is formulated based on defining a *simulation* relation between the system $\mathcal{S}$ and abstraction $\mathcal{M}$.

Definition 1: If $\mathcal{Q}'$ represents the set of system states for $\mathcal{S}$ and $q'_{v_t} \in \mathcal{Q}'$ represent the system state characterised by valuation v_t of variables in X, then we define a binary relation $\mathcal{R}' \subseteq \mathcal{Q}' \times \mathcal{Q}$ to be a simulation if $(q'_{v_t}, q_i) \in \mathcal{R}'$ implies that $\forall (v_t, v_{t+1}) \models R$ i.e, $q'_{v_t} \to q'_{v_{t+1}}$, $\exists q_{i+1} \in \mathcal{Q}$ such that $q_{i+1} \in \delta(q_i, v_{t+1})$ and $(q'_{v_{t+1}}, q_{i+1}) \in \mathcal{R}'$.

A. Completeness Conditions for a Candidate Abstraction

Given a candidate abstraction $\mathcal{M}$ for a system $\mathcal{S}$, we extract the following conditions encoding the completeness hypothesis:

$$v_t \models \mathit{Init} \wedge (v_t, v_{t+1}) \models R \implies v_{t+1} \models \bigvee_{p_o \in P_{(0,out)}} p_o \quad (1)$$

where $P_{(0,out)}$ is the set of predicates for all outgoing transitions from an automaton state $q_0 \in \mathcal{Q}^0$, and for all $p_i \in P_{(j,in)}$

$$v_t \models p_i \wedge (v_t, v_{t+1}) \models R \implies v_{t+1} \models \bigvee_{p_o \in P_{(j,out)}} p_o \quad (2)$$

where $P_{(j,in)}$ is the set of predicates on the incoming transitions to state $q_j \in \mathcal{Q}$ and $P_{(j,out)}$ is the set of predicates on outgoing transitions from q_j. Condition (2) is extracted for all states in $\mathcal{Q}$.

We compute the fraction of conditions that hold on the system, denoted by α, as a quantitative measure of the degree of completeness of the learned model. If all extracted conditions hold, i.e., $\alpha = 1$, then the generated model admits all system behaviours. A violation indicates missing behaviour in $\mathcal{M}$. A proof is provided in [10].

B. Verifying Extracted Conditions Against the System

To enable the application of existing software model checkers, we construct source code for functions that encode conditions (1) and (2) of the form $v_t \models r \wedge (v_t, v_{t+1}) \models R \Longrightarrow v_{t+1} \models s$ as assume/assert pairs, as illustrated in Fig. 2a.

To check if the system satisfies a condition, we run model checking on the constructed code. When all assume/assert pairs are proved valid, this implies that the extracted conditions are always satisfied and therefore can be used as system invariants.

In case of a failure, the checker returns a sequence of valuations $\sigma'' = v_t, v_{t+1}$ as the counterexample, such that $v_t \models r \wedge (v_t, v_{t+1}) \models R \wedge v_{t+1} \not\models s$. This can be used to construct a set of new traces as follows. For each trace $\sigma \in \mathcal{T}$ we find the smallest prefix $\sigma' = v_1, v_2, \ldots, v_j$ such that $v_j \models r$. We then construct a new trace $\sigma_{CE} = v_1, v_2, \ldots, v_{j-1}, v_t, v_{t+1}$ for each prefix σ'. Note that since $v_t \models r$, the new trace σ_{CE} does not change the system behaviour represented by σ' but merely augments it to include the missing behaviour. The set of new traces $\mathcal{T}_{CE}$ thus generated is used as an additional input to the model learning component, which in turn generates a new abstraction that admits the missing behaviour.

For a violation of condition (1), the checker returns a counterexample $\sigma'' = v_0, v_1$ such that $v_0 \models \mathit{Init}$ and $(v_0, v_1) \models R$. σ'' is therefore a valid counterexample. However, the counterexample for a violation of condition (2) could be spurious. Let $\sigma'' = v_t, v_{t+1}$ be the corresponding counterexample generated by the model checker. Here, it is not guaranteed that the system state characterised by v_t is reachable from an initial system state. Therefore, the counterexample may not actually correspond to missing system behaviour.

C. Identifying Spurious Violations

To check if a counterexample $\sigma'' = v_t, v_{t+1}$ is spurious, the valuation v_t is encoded as the following Boolean formula:

$$s' := \bigwedge_{x_i \in X} (x_i = v_t(x_i))$$

and the negation, $\neg s'$, is used to assert that s' never holds at any point in the execution of $\mathcal{S}$ starting from an initial state, as shown in Fig. 2b. The system $\mathcal{S}$ is modelled as multiple

```
assume(r ∧ R)
assert(s)
```

(a)

```
assume(Init)
while true do
    assume(R)
    assert(¬s')
end while
```

(b)

Fig. 2: Constructed source code for (a) Condition check (b) Counterexample validity check.

TABLE I: Results of experimental evaluation of the active learning algorithm.

Benchmark			$\lvert X \rvert$	k	Our Algorithm						Random Sampling		
					i	d	N	α	$T(s)$	$\%T_m$	N	α	$T(s)$
AutomaticTransmissionUsingDurationOperator			4	125	6	**1**	5	**1**	3678.3	2.7	4	0.2	38.2
BangBangControl UsingTemporalLogic	Heater		5	62	4	**1**	4	**1**	11845.5	0.1	3	0.6	64
	On				4	**1**	5	**1**	11078	0.2	5	0.7	89.5
CountEvents			3	20	2	**1**	3	**1**	10.8	41.7	4	0.8	56.5
FrameSyncController			3	530	1	0	1	0	timeout		2	0.7	31
HomeClimateControlUsingTheTruthtableBlock			7	10	1	**1**	2	**1**	5	18.2	2	**1**	72.3
KarplusStrongAlgorithm UsingStateflow	DelayLine		5	100	2	**1**	3	**1**	430.9	0.8	3	**1**	33.9
	MovingAverage				3	**1**	3	**1**	1441.1	0.4	3	**1**	35.2
LadderLogicScheduler			3	10	9	**1**	4	**1**	157	63.9	3	0	52.9
MealyVendingMachine			2	10	1	**1**	4	**1**	8.9	49.1	4	**1**	67
ModelingACdPlayerradio UsingEnumeratedDataType	CdPlayer BehaviourModel	DiscPresent	13	205	4	0.1	4	0.2	timeout		12	0.2	953.7
		Overall			4	0.8	6	0.6	timeout		7	0.6	282.8
	CdPlayer ModeManager	ModeManager			1	**1**	4	**1**	10.9	32.2	4	**1**	416.1
		InOn			1	**1**	5	0.7	timeout		5	0.8	876.9
		Overall			1	**1**	2	**1**	4.7	17.5	2	**1**	138
ModelingALaunchAbortSystem	Abort	InabortLogic	6	22	2	**1**	6	**1**	518.9	3.5	seg fault		
		Overall			1	**1**	4	**1**	7.6	39.1	4	**1**	70.6
	ModeLogic				4	**1**	5	**1**	52.2	30.8	5	0.4	107.8
ModelingAnIntersectionOf Two1wayStreetsUsingStateflow	InRed		11	60	1	0.8	8	0.4	timeout		8	0.4	105.6
	Overall				1	**1**	6	0.6	timeout		6	0.6	81.8
ModelingARedundantSensorPairUsingAtomicSubchart			6	20	4	**1**	4	**1**	1007.6	1.8	5	0.5	72.4
ModelingASecuritySystem	InAlarm	InOn	16	100	16	**1**	4	**1**	1599	18.5	seg fault		
		Overall			1	**1**	3	**1**	7.4	20.7	seg fault		
	InDoor				1	**1**	3	**1**	7.1	16.7	seg fault		
	InMotion	InActive			9	**1**	4	**1**	1017.1	8.9	seg fault		
		Overall			1	**1**	3	**1**	7.5	15	seg fault		
	InWin				1	**1**	3	**1**	8.1	17.8	seg fault		
MonitorTestPointsInStateflowChart			2	20	1	**1**	2	**1**	2.9	30.2	2	**1**	29.6
MooreTrafficLight			3	40	3	**1**	7	**1**	89.3	38.4	9	0.7	124
ReuseStatesByUsingAtomicSubcharts			2	10	1	**1**	3	**1**	5.8	27.3	3	**1**	52.8
SchedulingSimulinkAlgorithmsUsingStateflow			3	127	5	**1**	3	**1**	54.7	17.7	4	0.8	35.9
SequenceRecognitionUsingMealyAndMooreChart			2	30	1	**1**	5	**1**	9.8	54.4	5	**1**	88.2
ServerQueueingSystem			4	40	2	**1**	3	**1**	13.8	31	4	0.6	99.3
StatesWhenEnabling			2	30	1	**1**	4	**1**	4.3	35.8	4	**1**	32.1
StateTransitionMatrixViewForStateTransitionTable			3	25	4	**1**	5	**1**	53.9	52.8	7	0.8	89.1
Superstep	With Super Step		1	10	1	**1**	1	**1**	139.7	0.4	1	**1**	21.8
	Without Super Step				1	**1**	3	**1**	141.4	0.8	3	**1**	25.5
TemporalLogicScheduler			2	202	6	**1**	4	**1**	270.8	4.4	4	**1**	36
UsingSimulinkFunctionsToDesignSwitchingControllers			3	10	1	**1**	4	**1**	7.2	27.1	4	**1**	41.5
VarSize	SizeBasedProcessing		4	35	2	**1**	3	**1**	115.6	3.4	4	0.6	38.9
	VarSizeSignalSource				2	**1**	5	**1**	157	5.3	6	0.7	47.4
ViewDifferencesBetweenMessagesEventsAndData			2	10	2	**1**	4	**1**	9	46.9	4	**1**	34.8
YoYoControlOfSatellite	InActive	InReelMoving	8	10	2	**1**	4	**1**	18.5	47.3	4	**1**	60.1
		Overall			2	**1**	4	**1**	19.5	46.7	4	**1**	71.9
	Overall				1	**1**	3	**1**	3.7	28.7	3	**1**	43

unwindings of the transition relation R (lines 2-5 in Fig. 2b). We verify this using model checking with k-induction [12], [13]. If both the base case and step case for k-induction hold, it is guaranteed that the counterexample is spurious, in which case we strengthen the assumption in Fig. 2a to $(r \land \neg s' \land R)$ and repeat the condition check. In case of a violation only in the step case, there is no conclusive evidence for validity. Since we are not interested in generating an exact system model but rather an over-approximation that provides useful insight into the system, we treat such a counterexample as valid.

For the bound k, a value greater than or equal to the diameter of the system guarantees completeness [12]. We discuss ways to approximate this value in [10]. Note that a poor choice for the bound k results in more spurious behaviours being added to the model, resulting in low accuracy. But, the learned models are guaranteed to admit all system traces defined over X, irrespective of the value for k.

IV. Evaluation and Results

A. Evaluation Setup

For our experiments we use Trace2Model (T2M) [4], [14] as the model learning component. To evaluate the degree of completeness we use the C Bounded Model Checker (CBMC v5.35) [15]. We implement Python modules for the following: constructing the source code to check each condition, processing the CBMC output, and translating CBMC counterexamples into a set of trace inputs for model learning. Note that any model checker can be used in place of CBMC.

To evaluate our algorithm, we reverse-engineer a set of FSAs from their respective C implementations. We use a dataset of Simulink Stateflow models [16] as our benchmark set. For each benchmark, we use Embedded Coder [17] to automatically generate a C implementation. The generated C implementation is used as the system $\mathcal{S}$. Further details are provided in [10].

The implementation and benchmarks are available online [18].

B. Experiments and Results

For each benchmark, we generate an initial set of 50 traces, each of length 50, by executing the system with randomly sampled inputs. Some of the Stateflow models are implemented as multiple parallel and hierarchical FSAs. For a given implementation $\mathcal{S}$ and a set of observables X, we attempt to reproduce each state machine separately using traces defined over all variables in X. We therefore generate an abstraction with state transitions at a system level for each FSA in $\mathcal{S}$.

The results are summarised in Table I. We quantitatively assess the quality of the final generated model for each FSA by assigning a score d, computed as the fraction of state transitions in the Stateflow model that match corresponding transitions in the abstraction. For hierarchical Stateflow models, we flatten the FSAs and compare the learned abstraction with the flattened FSA. We record the number of model learning iterations i, the number of states N and degree of completeness α for the final model, the total runtime T and the percentage of total runtime attributed to model learning, denoted by $\%T_m$. We set a timeout of 10 h for our experiments. For benchmarks that time out, we present the results for the model generated right before timeout.

1) Runtime: The active learning algorithm is able to generate abstractions in under 1 h for the majority of the benchmarks. For the benchmarks that time out, the model checker tends to go through a large number of invalid counterexamples before arriving at a valid counterexample for a condition violation. This is because, depending on the size of the domain D, there can be a large number of possible valuations that violate an extracted condition, of which very few may correspond to a valid system state. In such cases, runtime can be improved by strengthening the assumption $r \wedge R$ in Fig. 2a with domain knowledge to guide the model checker towards valid counterexamples. For *FrameSyncController*, CBMC takes a long time to check each condition. This is because the implementation features several operations, such as memory access and array operations, that especially increase proof complexity and proof runtime.

2) Generated Model Accuracy: The algorithm is guaranteed to generate an abstraction that admits all system behaviours, as is confirmed by α in Table I. We also see that $d = 1$ for these benchmarks. For two benchmarks, although the Simulink model matched the generated abstraction ($d = 1$), the algorithm timed out before it could eliminate all spurious violations ($\alpha < 1$).

3) Number of Learning Iterations: In each learning iteration j, $|L(\mathcal{M}_j)| > |L(\mathcal{M}_{j-1})|$ as $L(\mathcal{M}_j) \supseteq L(\mathcal{M}_{j-1}) \cup \mathcal{T}_{CE_j}$ and $\mathcal{T}_{CE_j} \cap L(\mathcal{M}_{j-1}) = \emptyset$. Here, $\mathcal{M}_j$ and $\mathcal{T}_{CE_j}$ are the generated abstraction and the set of new traces collected in iteration j respectively. The algorithm terminates when $L(\mathcal{M}_j) \supseteq \mathit{Traces}_X(S)$. The number of learning iterations therefore depends on $|\mathit{Traces}_X(S) \cap L(\mathcal{M}_0)|$, where $\mathcal{M}_0$ is the abstraction generated from the initial trace set.

C. Comparison with Random Sampling

We performed a set of experiments to check if random sampling is sufficient to learn abstractions that admit all behaviours. A million randomly sampled inputs are used to execute each benchmark. Generated traces are fed to T2M to passively learn a model. T2M fails to generate a model for 7 benchmarks, as its predicate synthesis procedure returns 'segmentation fault'. For 50% of the remaining benchmarks, random sampling fails to produce a model admitting all system behaviours ($\alpha < 1$).

D. Threats to Validity

The key threat to the validity of our experimental claim is benchmark bias. We have attempted to limit this bias by using a set of benchmarks that was curated by others. Further, we use C implementations of Simulink Stateflow models that are auto-generated using a specific code generator. Although there is diversity among these benchmarks, our algorithm may not generalise to software that is not generated from Simulink models, or software generated using a different code generator.

References

[1] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 501–510.

[2] N. Walkinshaw and M. Hall, "Inferring computational state machine models from program executions," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 122–132.

[3] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, Jun 2016.

[4] N. Y. Jeppu, T. Melham, D. Kroening, and J. O'Leary, "Learning concise models from long execution traces," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, July 2020, pp. 1–6.

[5] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[6] G. Argyros and L. D'Antoni, "The learnability of symbolic automata," in *CAV*. Springer, 2018, pp. 427–445.

[7] B. Garhewal, F. Vaandrager, F. Howar, T. Schrijvers, T. Lenaerts, and R. Smits, "Grey-box learning of register automata," in *Integrated Formal Methods*. Springer, 2020, pp. 22–40.

[8] F. Howar and B. Steffen, "Active automata learning in practice – an annotated bibliography of the years 2011 to 2016," in *Machine Learning for Dynamic Software Analysis*, 2018.

[9] F. Howar, B. Jonsson, and F. Vaandrager, "Combining black-box and white-box techniques for learning register automata," in *Computing and Software Science: State of the Art and Perspectives*. Springer, 2019, pp. 563–588.

[10] N. Y. Jeppu, T. Melham, and D. Kroening, "Active learning of abstract system models from traces using model checking [Extended]," *arXiv:2112.05990*, 2021.

[11] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*, 2nd ed. MIT Press, 2018.

[12] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*. Springer, 2000, pp. 127–144.

[13] A. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," in *Static Analysis Symposium (SAS)*, ser. LNCS, vol. 6887. Springer, 2011, pp. 351–368.

[14] N. Y. Jeppu, *Trace2Model*, 2020. [Online]. Available: https://github.com/natasha-jeppu/Trace2Model

[15] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 2988. Springer, 2004, pp. 168–176.

[16] Simulink, "Stateflow examples," 2021. [Online]. Available: https://uk.mathworks.com/help/stateflow/examples.html?s_tid=CRUX_topnav

[17] Simulink, "Embedded coder," 2021. [Online]. Available: https://uk.mathworks.com/products/embedded-coder.html

[18] N. Y. Jeppu, *ActiveLearning*, 2021. [Online]. Available: https://github.com/natasha-jeppu/ActiveLearning