Specification Guided Automated Debugging of CPS Models

Nikhil Kumar Singh, *Student Member, IEEE* and Indranil Saha, *Member, IEEE*

***Abstract*—Simulink/Stateflow is the *de facto* tool for developing software for safety-critical real-time cyber-physical systems (CPSs). In Simulink, the model of a CPS is captured in a block diagram based language, the model is simulated using the associated simulators, and then software code is generated automatically for the embedded controller. The presence of a bug in the Simulink model may lead to a catastrophic failure during the execution of the system developed based on the model. Unlike in application software, finding bugs in Simulink models is challenging due to the hybrid nature of the model.**

We present an automated debugging methodology of a CPS model captured in Simulink. Our methodology has two main components — bug localization and model repair. For bug localization, we capture the requirements of the system in Signal Temporal Logic (STL) and employ the run-time monitoring technique to generate a trace that violates the specification. The violating trace is used to identify the internal signals that have the potential to contribute to the violation. For precise bug localization by narrowing down the offending signals, we employ a matrix decomposition technique to find the signals contributing to the bug accurately. Our bug localization technique is precise enough to enable us to repair the model. If the bug is due to an inappropriate value for a model parameter, we employ a parameter tuning method to find a value for the parameter that repairs the model automatically. We carry out numerous case studies on Simulink models obtained from different sources and demonstrate that our automated debugging technology can fix the bugs in the Simulink models effectively.

***Index Terms*—Simulink Models, Bug Localization, Model Repair, Automated Debugging, Signal Temporal Logic, Falsification.**

I. Introduction

The development of safety-critical real-time cyber-physical systems (CPSs) poses a tremendous challenge to engineers in terms of design, implementation, and verification. The *Simulink* tool from Mathworks Inc. [1] is a software package that provides an environment for model-based development of cyber-physical systems. In the Simulink environment, a model of a cyber-physical system can be created in a block-diagram based visual language, which can be simulated using various simulators. Of late, there has been significant progress towards developing software tools for Bug localization in the Simulink models [2], [3], [4]. However, these tools require a significant amount of human intervention, which renders the debugging process tedious and time-consuming. Thus, one key challenge in the model-based development of CPSs is to develop an automated tool that provides precise information about the source of the bugs in an erroneous model without much manual effort.

There are various techniques available for automated fault localization and debugging of application software [5], [6], [7]. However, little research has been carried out in automating such efforts in case of models of cyber-physical systems. The correctness of application software is captured using test cases that provide the correct input-to-output mapping. On the other hand, the correctness of a cyber-physical system is given in the form of a set of temporal logic formulae that define a set of valid traces for the system. The bug in a C code is indicated by a test-case (inputs) that leads to a wrong output. On the other hand, a bug in a Simulink model is indicated by a system trace that violates the temporal logic specification.

In the recent past, falsification-based techniques [8], [9], [10], [11] have been employed extensively in identifying whether a model is erroneous with respect to a temporal logic specification. These techniques focus on checking whether the simulation traces satisfy the specification given in *Signal Temporal Logic* (STL) [12], [13], which is a popular logical specification language to capture real-time specifications for CPSs. In case of a falsification, we get the corresponding sequence of states that lead to the violation of the specification. Debugging, however, requires precise information about the internal structure of the model, which is not provided by the falsification techniques.

In this paper, we propose a bug localization algorithm that is based on falsification of STL specifications for Simulink models. The inputs to our algorithm are a Simulink model and an STL specification where the predicates are defined based on the signals in the model. If the specification gets falsified, we employ a matrix analysis technique to identify a small subset of the signals that contribute to the falsification. First, we perform slicing of the model based on the signals within the violated specification. Subsequently, we create a matrix that captures the value of the sliced signals at all time-stamps where the specification is falsified. This matrix provides us with key information associated with the specification violation. Now, to remove the redundancy from the matrix, we employ a matrix decomposition technique to identify the columns (signals) in the matrix corresponding to the largest singular values, indicating that they have the most impact on the bug. This analysis helps us in obtaining a small set of suspected signals which are presented to the user as the suspected cause of the falsification.

A major class of bugs in Simulink models is related to the

N. K. Singh and I. Saha are with the Department of Computer Science and Engineering, Indian Institute of Technology Kanpur e-mail: {nksingh, isaha}@cse.iitk.ac.in.

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue.

values of various parameters in the model. For example, in an automatic transmission model, there is a parameter *threshold* in the value of the vehicle speed that determines when the gear shift should take place. Automotive engineers often decide the values of such parameters based on their experiences. Though the value of the parameter chosen by their experience works most of the time, in corner cases those values may turn out to be wrong. If our bug localization algorithm detects that the cause of the failure of the model is the inappropriate value of a parameter, the parameter value can be tuned automatically to find its correct value. We provide an algorithm for repairing the model by tuning the offensive parameter automatically.

Finally, we present an algorithm that debugs a Simulink model with many independent specifications by employing the bug localization and model repair mechanism mentioned above. This algorithm enables us to fix the model even without falsifying all the specifications independently. We implement this algorithm to provide a fully automated tool to debug a Simulink model with many STL specifications. We evaluate our debugging tool on five different Simulink models with varying complexity. For each of those models, we consider a set of 1-7 STL specifications, a subset of which were falsified on the models. It turns out that all the falsifications were due to inappropriate values for some parameters. We have successfully repaired all the models based on the output generated by our bug localization technique. Our success in repairing the models demonstrates that our bug localization algorithm can pinpoint the source of the bug appropriately.

In summary, we make the following contributions in this paper.

- We present a mechanism for localization of a bug in a Simulink model. We, for the first time, employ a matrix decomposition technique to localize bug in a CPS model.
- We provide a mechanism for repairing an erroneous Simulink model by carefully tuning parameters indicated by the bug localization process.
- We develop an algorithm to debug a Simulink model with respect to a set of falsified STL specifications by appropriately combining our bug localization and debugging algorithms.
- We implement our bug localization and repair mechanism in a MATLAB based tool that helps us solve the debugging problem automatically. We apply our tool to repair five different Simulink models with 1-7 STL specifications for each of them.

II. Problem

A. Preliminaries

1) Simulink/Stateflow: Simulink/Stateflow [1] tool offers a library of *blocks* representing various discrete and continuous mathematical operations such as gain, addition, transforms, lookup tables, integration, etc. It also supports hierarchical structuring of models by grouping the related blocks into *subsystems*. Stateflow charts specify the control in the form of hierarchical finite state machines that interact with the Simulink model. A Simulink model represents the time-dependent mathematical relationship between the inputs, states, and outputs of the system.

Syntax. A Simulink model $\mathcal{M}$, syntactically, is defined as a 4-tuple $\langle V, B, C, S\rangle$:

- V refers to the variables (input, output or internal state) of the Simulink model. The input, the output and the state variables are denoted by V_I, V_O, and V_S respectively.
- B refers to the set of blocks in the Simulink model.
- C refers to the set of connections between the blocks, defined as the ordered relation $C \subseteq B \times B$.
- S refers to the set of signals in the Simulink model, defined as the mapping $S : C \rightarrow V$.

For a connection $c \in C$ in a Simulink model, we denote its source block by $\texttt{src}(c)$ and its destination block by $\texttt{dst}(c)$. Some of the Simulink blocks contain parameters (for example, the Gain block). For a block $b \in B$, its set of parameters is denoted by $\texttt{param}(b)$. For a parameter p, its value is denoted by $p.val$, and the set of possible values for the parameter is denoted by $\texttt{P}$.

Semantics. Let us denote an input to the model $\mathcal{M}$ by u. The input u is a vector capturing the values for all the variables in V_I. A Simulink model $\mathcal{M}$, semantically, is defined by a tuple $\langle X, X_0, U, T, SIM\rangle$ that consists of:

- a state-space X where $x \in X$ is a state vector capturing the valuation of all the variables in V at a given time instance.
- a set $X_0 \subseteq X$ representing initial states of the model.
- a set of input signals U.
- a time horizon $T > 0$.
- a simulator $\texttt{SIM}$: $X_0 \times U \times$ [0,T] $\rightarrow X$, $\texttt{SIM}(x_0,u,t)$ denotes the state reached at time t from initial state x_0 using input signal u.

The model $\mathcal{M}$ starts at an initial state x_0, runs on the input signal u, and generates a trace. A trace ω is defined as the sequence of states of the system evolving with discrete time-steps (from $t = 0$ to $t = T$). We denote the state of the system at time t by $x_t \in X$ and the trace ω by $\langle x_0, x_1, \ldots x_T\rangle$, where $x_t = \texttt{SIM}(x_0,u,t)$. For the Simulink model $\mathcal{M}$, we denote all the traces generated from some initial state $x_0 \in X_0$ and for some input signal $u \in U$ by $\mathcal{L}(\mathcal{M})$.

2) Signal Temporal Logic: Signal Temporal Logic (STL) [12], [13] is an extension of Metric Temporal Logic (MTL) [14] and Linear Temporal Logic [15]. It enables us to reason about real-time properties of signals (simulation traces). These specifications consist of real-time predicates over the signal variables.

The syntax of an STL specification ϕ is defined by the grammar

$$\phi := \mathsf{true} \mid \pi \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 U_I \phi_2 \qquad (1)$$

where $\pi \in \Pi$, Π is a set of atomic predicates, and $I \subseteq \mathbb{R}^+$ is an arbitrary interval. The logical operators $\neg$ and $\vee$ have their usual meaning. Here, U_I is the until operator implying that ϕ_2 becomes true at some time within the interval I and ϕ_1 must remain true until ϕ_2 becomes true. There are two other useful temporal operators, namely eventually($\Diamond_I$) and always($\Box_I$), which can be derived from the temporal and logical operators defined above. We use the temporal operators U, $\Diamond$ and $\Box$ to denote the operators U_I, $\Diamond_I$ and $\Box_I$ with the time interval I to be $[0, \infty)$.

Robust Semantics of STL: We follow the robust semantics of STL provided in [16]. We use Euclidean metric as the norm to measure the distance d between two values $v, v' \in \mathbb{R}$, i.e., $d(v, v') = \|v - v'\|$. Let $v \in \mathbb{R}$ be a value, $A \subseteq \mathbb{R}$ be a set. Then the signed distance from v to A is defined as:

$$\mathtt{Dist}(v, A) = \begin{cases} inf\{d(v, v') \mid v' \notin A\}, & \text{if } v \in A. \\ -inf\{d(v, v') \mid v' \in A\}, & \text{if } v \notin A. \end{cases} \tag{2}$$

Intuitively, $\mathtt{Dist}(v, A)$ captures how far a value v is from the violation of the inclusion in the set A. In both cases, we search for the minimum distance between v and a point on the boundary of A. Also, the case $v \notin A$ refers to a violation and thus the negative sign is used in the definition.

Let $\mathcal{O}$: $\Pi \rightarrow 2^X$ represent a mapping of a predicate (π in (1)) to a set of states. Given a trace ω and $\mathcal{O}$, we define the robust semantics of ω w.r.t. ϕ at time $t \in \mathbb{R}$, denoted by $[[\phi]](\omega, t)$, by induction as follows:

$$[[\mathtt{true}]](\omega, t) = +\infty \tag{3a}$$

$$[[\pi]](\omega, t) = \mathtt{Dist}(x_t, \mathcal{O}(\pi)) \tag{3b}$$

$$[[\neg\phi]](\omega, t) = -[[\phi]](\omega, t) \tag{3c}$$

$$[[\phi \wedge \psi]](\omega, t) = \mathtt{min}([[\phi]](\omega, t), [[\psi]](\omega, t)) \tag{3d}$$

$$[[\phi U_I \psi]](\omega, t) = \sup_{t' \in t+I} \mathtt{min}([[\psi]](\omega, t'), \inf_{t'' \in [t, t']} [[\phi]](\omega, t'')) \tag{3e}$$

The robustness metric $[[\phi]]$ maps each simulation trace ω to a real number r. If $[[\phi]](\omega, t) \neq 0$, its sign indicates the satisfaction status. Also, if ω satisfies ϕ at time t, any other trace ω' whose Euclidean distance from ω is smaller than $[[\phi]](\omega, t)$ also satisfies ϕ at time t.

We define the falsification problem as follows: For a given system $\mathcal{M}$ and a specification ϕ, find $\omega \in \mathcal{L}(\mathcal{M})$ such that $[[\phi]](\omega, t) < 0$. This is generally captured as an optimization problem:

$$\omega^* = \underset{\omega \in \mathcal{L}(\mathcal{M})}{\arg\min} \quad [[\phi]](\omega) \tag{4}$$

where, we define $[[\phi]](\omega)$ as the minimum robustness of trace ω w.r.t. ϕ.

B. Problem Definition

In this paper, we assume that a Simulink model $\mathcal{M}$ along with its STL specification ϕ is given as the input. A specification ϕ represents an acceptable behaviour of model $\mathcal{M}$, i.e., any trace $\omega \in \mathcal{L}(\mathcal{M})$ should belong to the language of ϕ, i.e., $\forall \omega \in \mathcal{L}(\mathcal{M}),\ \omega \in \mathcal{L}_\phi$. However, if there exists a trace $\omega' \in \mathcal{L}(\mathcal{M})$ that does not belong to the language of ϕ, i.e., $\omega' \notin \mathcal{L}_\phi$, then the model $\mathcal{M}$ does not satisfy the specification ϕ, and we write $\mathcal{M} \not\models \phi$. In such a situation, our goal is to find the root cause of the falsification and repair the model in such a way that the repaired model satisfies the specification.

Given a Simulink model $\mathcal{M}$ and a specification ϕ, the problems addressed in this paper are formally presented below.

Problem [Bug Localization] If $\mathcal{M} \not\models \phi$, identify the minimal set of signals $s_{spt} \subseteq S$ that can accurately explain the violation of ϕ.

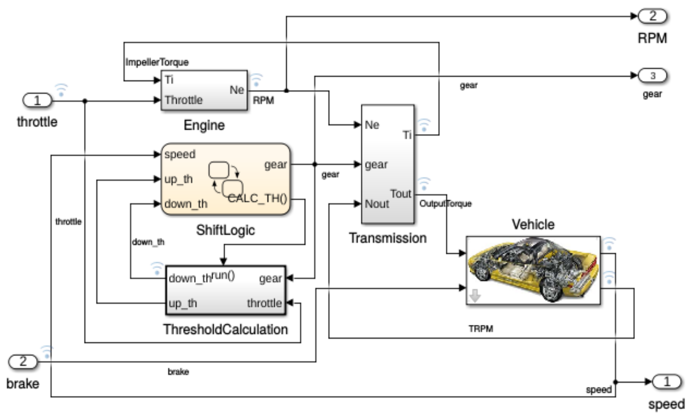


Fig. 1: Simulink model of an Automatic Transmission system (taken from [17])

Problem [Repair] If $\mathcal{M} \not\models \phi$, make minimal change to model $\mathcal{M}$ and generate a model $\mathcal{M}_r$ so that $\mathcal{M}_r \models \phi$.

In defining the problems above, we do not define the terms "accurately" and "minimal change to the model" formally. Thus, we are not looking for a solution that will provide guarantee on the optimality of the produced outputs. We rather seek for heuristic solutions the quality of which we evaluate experimentally. Also, in the above-mentioned problem definitions, we assume that the specification is correct and the falsification happens due to a fault in the model.

C. Example

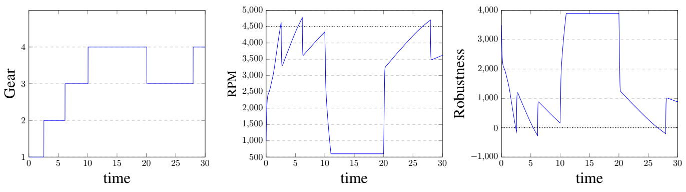
We illustrate the problem with an example Simulink model shown in Figure 1. It is a model of an Automatic Transmission [17] that exhibits both continuous and discrete behavior. The system has two inputs - `Throttle` and `Brake`. The system has continuous state variables (the speed of the engine `RPM`, the speed of the vehicle `Speed`) and discrete state variable (`Gear`). The input signals can take any value between [0, 100] and [0, 325] respectively.

Let us consider the following specification - we want to ensure that the vehicle speed v (corresponds to the signal `Speed` in the model) and the engine speed r (`RPM` in the model) are always bounded by values v_{max} and r_{max} respectively. We express this as the following STL specification:

$$\phi = \Box(v < v_{max}) \wedge \Box(r < r_{max}). \tag{5}$$

The specification is falsified, and the timestamps where violations occur are those where the robustness value is negative. The cause of violation is that the `RPM` (r) exceeds its maximum permissible value 4500 (Figure 2). This event occurs each time when there is a gear change. In the Simulink model, the gear change happens automatically when the vehicle reaches a particular speed. The underlying idea behind fixing this issue is to reduce the threshold for gear change, such that before the engine speed exceeds its maximum permissible value, the gear change takes place.

We can monitor the simulation traces of model $\mathcal{M}$ for the specifications ϕ using various tools [8], [9], [18]. In case of violation of a specification ϕ, the information provided by



(a) Gear signal (b) RPM (r) signal (c) Robustness

Fig. 2: Falsification in the Automatic Transmission model

Algorithm 1: Bug Localization Algorithm

```
1  procedure bug_localization ($\mathcal{M}$, $\phi_f$, $\omega_f$)
2      $\mathcal{M}_f \leftarrow$ flatten_model($\mathcal{M}$)
3      $err\_signal \leftarrow$ extract_signals ($\phi_f$)
4      $\gamma \leftarrow$ slice_simulink_model ($\mathcal{M}_f$, $err\_signal$)
5      $M \leftarrow [\tau]$
6      for $i = 1$ : length ($\gamma$) do
7          $y \leftarrow$ plot_sig_portrait ($\omega_f$, $\gamma[i]$)
8          $M \leftarrow [M\ y]$
9      $\rho \leftarrow$ plot_robust_sat ($\omega_f$, $\phi_f$)
10     $N \leftarrow [\,]$;  $j = 1$
11     for $i = 1$ : length ($\rho$) do
12         if $\rho[i] < 0$ then
13             $N[j] \leftarrow [\tau[i]\ \rho[i]]$;  $j \leftarrow j + 1$
14     $P \leftarrow$ join($M$, $N$)
15     $[Q, R, E] \leftarrow$ matrix_decomposition($P$)
16     $diagr \leftarrow$ abs(diag($R$))
17     $maxindex \leftarrow$ find_max_index($diagr, \kappa$)  // $\kappa$ is a parameter
18     $s_{spt} \leftarrow \gamma[E[1 : maxindex]]$
19     return $s_{spt}$
```

these tools is not sufficient for debugging. So, we need an automated procedure that can help us localize the bug in model $\mathcal{M}$. Also, we want to use this information to repair the model such that the repaired model $\mathcal{M}_r$ satisfies specification ϕ. In the next two sections, we propose algorithms for precise bug localization and model repair. We will illustrate our algorithms using this example.

III. Bug Localization

In this section, we present our bug localization algorithm (Algorithm 1). The algorithm takes as input a Simulink model $\mathcal{M}$, a falsified STL formula ϕ_f, and the falsifying trace ω_f. The algorithm outputs the set of most suspected signals s_{spt}. As the first step, we flatten the input Simulink model $\mathcal{M}$ to expand all the subsystems recursively (except the atomic subsystems) (line 2). Flattening of the input Simulink model enables us to get precise localization of the bug and hence precise fixes compared to what we could get from the *base Simulink model* $\mathcal{M}$. In line 3, we store all the signals used in the specification ϕ_f in err_signal. Next, we apply model slicing technique [19], [20] to remove the signals that are not related to the signals in err_signal using the `slice_simulink_model` function (line 4). This step gives us a set of sliced signals γ. Then, in line 5, we create a matrix M with one column containing all the time-stamps τ. Mathematically, $\tau = [0\ \delta\ 2\delta \ldots T\delta]^\mathsf{T}$, where δ is the duration of a simulation step and T is the length of the trace ω_f. In line 6-8, we add values column-wise to M_f with the columns representing the traces for signals in γ for all time-stamps. In line 9, we plot the robustness (ρ) of the trace ω_f with respect to the STL formula ϕ_f following the robust semantics of STL introduced in Section II. In line 11-13, we store all the time-stamps where specification violation occurs in matrix N. In line 14, we apply the INNER JOIN procedure [21] to M and N on the field time, remove the columns on time (τ) and robustness (ρ) and store it in matrix P. The matrix P stores those values for all the signals in slice set γ that correspond to negative robustness (falsification). The INNER JOIN procedure creates a new result table by combining column values of two tables based on a join-predicate (in this case, time τ). In case there is a match on join-predicate in both the tables, the column values for each matched pair of rows of both the tables are combined to give the result table.

In line 15-18, we remove the least significant components of P using `matrix_decomposition` function (line 15) to find the set of most suspected signals s_{spt} (line 18) that contribute to the falsification of ϕ_f.

A. Matrix decomposition for locating the bug

The matrix $P \in \mathbb{R}^{m\times n}$ contains the precise values for the signals where robustness becomes negative, indicating the time instances when the violation of the specification occurs. Each of the n columns represent the values of a signal $s \in \mathbb{R}^{m\times 1}$ for each time-point where the specification is violated. Hence, matrix P provides an effective representation for the bug (i.e. the reason for falsification). Thus, to find the most suspected signal(s) s_{spt}, we need to analyse P to find a small subset of those signals that contribute to the bug most significantly. However, the matrix contains too many signals as mere slicing of the model based on err_signal may provide many signals which are not directly related to the bug. We attempt to identify the columns in matrix P that are independent, i.e., the columns that retain the crucial information about the matrix.

For a square matrix, the rank that is equal to the number of non-zero eigenvalues of the matrix gives us the number of linearly independent columns in the matrix. The matrix P, however, is not a square matrix. For a rectangular matrix, we can use *singular values* of the matrix, which behave the same way as the eigenvalues of a square matrix. The *singular values* of a matrix A is defined as the square-root of the eigenvalues of their associated square Gram matrix ($A^\mathsf{T}A$) [22]. The number of non-zero singular values provides the rank of the rectangular matrix. The standard practice for obtaining the singular values of a rectangular matrix P is *singular value decomposition* (SVD) [22]. This decomposition represents matrix P as $P = U\Sigma V^\mathsf{T}$, where U and V are orthogonal matrices, and Σ is a diagonal matrix containing the singular values of P in non-increasing order.

Though SVD provides important information about the singular values and the rank of matrix P, the major deficiency of SVD is that we cannot relate the singular values to the columns of P. We use the *permuted version of the QR decomposition* [23], [24], [22] to get the correspondence between the columns in the original matrix P and its singular values. The permuted-QR decomposition of P expresses the matrix as the product of a real orthogonal matrix Q and an upper triangular matrix R. It produces an economy-size decomposition in which E is a permutation vector, such that

$$P(:,E) = QR \tag{6}$$

Here, $P(:,E)$ represents the matrix formed by interchanging the columns of matrix P using the permutation vector E [22]. The diagonal values of R are sorted in decreasing order. Though QR decomposition does not provide us the singular values for P directly, it has been experimentally shown that the diagonal elements of R provide a close measure of the singular values of matrix P [25].

The signals (columns in P) corresponding to the nonzero diagonal elements of R form the basis of matrix P and are expected to have the most impact on the bug. However, as the matrix decomposition is performed using numerical methods, the diagonal elements that are significantly small in comparison to the largest diagonal element can be ignored safely. We define a parameter κ that decides the maximum number of signals to be returned to the user as the result of bug localization. Let the number of nonzero elements in $diag(R)$ be κ'. Our bug localization algorithm returns $\min(\kappa, \kappa')$ signals from the set γ corresponding to the largest elements in $diag(R)$ (line 16-18). The set of signals s_{spt} has the most suspected signals because it corresponds to the largest singular values in decreasing order. If the user is interested in the most probable location of the bug, we return the first signal in s_{spt}.

B. Complexity Analysis

The number of blocks (B) and the number of connections or signals (C) in a Simulink model are generally of the same order (Table I). Also, the size of an STL formula $|\phi_f|$ is negligible with respect to the size of the model (can be taken as $|C|$). Thus, we compute the complexity with respect to $|C|$ and the length of the simulation T (= $|\tau|$, the length of the signal ω_f). The `flatten_model` function in line 2 requires to scan the Simulink model once. Thus it incurs a computation cost of $O(|C|)$. The `extract_signal` function in line 3 needs to scan the STL formula once. We ignore the complexity of this function as we assume that $|\phi_f| \ll |C|$. The `slice_simulink_model` function in line 4 incurs a time complexity of $O(|C|^2)$ as there are at most $|C|$ elements in the set err_signal, and for each element in err_signal, we need to perform a depth-first search in the graph induced by the Simulink model. The complexity of the computation in line 6-18 is governed by the complexity of the function `matrix_decomposition`, which is $O(T \cdot |C|^2)$, as the complexity of QR decomposition is $O(m \cdot n^2)$, where m and n are the number of rows and the number of columns of the matrix respectively [22]. Thus, the overall time complexity of our bug localization algorithm is $O(T \cdot |C|^2)$.

C. Example

In the example in Sec. II-C, the formula $\Box(r < r_{max})$ gets falsified. In function `bug_localization`, the `err_signal` in line 3 is {`RPM`}. The slice set γ (line 4) consists of the following set of 28 signals: {`Eii`, `ImpellerTorque`, `Nin`, `OutputTorque`, `RPM`, `TRPM`, `Brake`, `Down_th`, `Drive_ratio`, `Gear`, ..., `Lin_speed`, `Speed`, `Throttle`, ...}. This set contains signals that are in the base model as well as within the subsystems. For example, the signal `RPM` is present in the base model, while the signal `Eii` and `Lin_speed` lie within Engine subsystem and Vehicle subsystem respectively. The matrix M (line 8) is given below.

$$\begin{array}{ccccccc} \tau & Eii & ImpT & .. & RPM & .. & Lin\_s & .. \\ \left(\begin{array}{cccccccc} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 6.11 & 338.84 & 271.19 & .. & 4751.4 & .. & 6187.9 & .. \\ 6.12 & 338.23 & 270.95 & .. & 4754.8 & .. & 6193.8 & .. \\ 6.13 & 337.63 & 270.71 & .. & 4758.2 & .. & 6199.7 & .. \\ 6.14 & 337.03 & 270.47 & .. & 4761.5 & .. & 6205.6 & .. \\ 6.15 & 336.44 & 271.23 & .. & 4764.9 & .. & 6211.5 & .. \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \end{array}\right) \end{array}.$$

Here τ is the vector containing the time steps in the simulation, `Eii` refers to signal Engine Impeller Inertia, `ImpT` is Impeller Torque, `Lin_s` refers to linear speed signal (within Vehicle subsystem). The matrix N is given by

$$\begin{array}{cc} \tau & \rho \\ \left(\begin{array}{cc} \vdots & \vdots \\ 6.11 & -251.41 \\ 6.12 & -254.8 \\ 6.13 & -258.18 \\ 6.14 & -261.5 \\ 6.15 & -264.92 \\ \vdots & \vdots \end{array}\right) \end{array}.$$

Here, the second column provides the robustness values (ρ) at different time steps. The matrix obtained after the INNER JOIN operation is shown below.

$$\begin{array}{cccccccc} \tau & Eii & .. & RPM & ... & Lin\_s & ... & \rho \\ \left(\begin{array}{cccccccc} \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 6.11 & 338.84 & .. & 4751.4 & ... & 6187.9 & ... & -251.4 \\ 6.12 & 338.23 & .. & 4754.8 & ... & 6193.8 & ... & -254.8 \\ 6.13 & 337.63 & .. & 4758.2 & ... & 6199.7 & ... & -258.2 \\ 6.14 & 337.03 & .. & 4761.5 & ... & 6205.6 & ... & -261.5 \\ 6.15 & 336.44 & .. & 4764.9 & ... & 6211.5 & ... & -264.9 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array}\right) \end{array}.$$

The matrix P in line 14 is obtained by removing the first and the last column of the above matrix. Now, we perform the QR decomposition of matrix P (line 15) and obtain an upper triangular matrix R with $abs(diag(R))$ (line 16) given by [343670 126540 96160 47880 24080 6530 2150 .... 0]$^{\mathsf{T}}$. Here, $\kappa' = 7$; however, several diagonal elements have significantly smaller values compared to the first element. The matrix E is given as $E = $ [24 1 22 13 3 5 7 .... 18], which indicates that column 24 in matrix P corresponds to the largest singular value. Column 24 in matrix P corresponds to the signal `Lin_speed`. Thus, if $\kappa = 1$, we return $s_{spt} = \{$`Lin_speed`$\}$. Similarly, if $\kappa = 2$, $s_{spt} = \{$`Lin_speed`, `Eii`$\}$.

In this paper, for all experiments, we have used $\kappa = 1$, i.e., $|s_{spt}| = 1$, and we have been able to localize and fix the bugs successfully.

Algorithm 2: `ParameterTuning`

```
Input: A model $\mathcal{M}$, a signal $s \in s_{spt}$ and an STL specification $\phi$
Output: A model $\mathcal{M}_r$ that satisfies $\phi$
1  $p \leftarrow$ param (src($s$))
2  $curr\_rob \leftarrow$ min_robustness($\mathcal{M}$, $\phi$)
3  $old\_rob \leftarrow curr\_rob$
4  $default\_val \leftarrow p.val$
5  $\alpha \leftarrow$ rand$(1-\delta, 1+\delta)$
6  $\alpha_{old} \leftarrow v$, where $1-\delta \leq v \leq 1+\delta$, $v \neq \alpha$
7  while $|\alpha - \alpha_{old}| > limit$ do
8      $new\_val \leftarrow default\_val$
9      for $i=1$ to $H$ do
10         $new\_val = new\_val * \alpha$
11         $new\_rob \leftarrow$ min_robustness($\mathcal{M}(p.val = new\_val)$, $\phi$)
12         if $new\_rob > 0$ then
13             $\mathcal{M}_r \leftarrow$ save_system ($\mathcal{M} = (p.val = new\_val)$)
14             return $\mathcal{M}_r$
15         if $new\_rob > curr\_rob$ then
16             if accept($new\_val, \lambda$) then
17                 $curr\_rob \leftarrow new\_rob$
18     $\alpha_{old} \leftarrow \alpha$
19     $\alpha \leftarrow$ update($\alpha, old\_rob, curr\_rob$)
20     $old\_rob \leftarrow curr\_rob$
```

IV. Model Repair

The bug localization algorithm provides us the most suspected signal(s). Since the value of a signal depends on its source block, we focus on the parameters of the source blocks of the suspected signals and attempt to fix the model by tuning them. In our repair mechanism, we work with a *single-fault assumption*, which ensures that the value of at most one parameter is erroneous. However, the repair algorithm can be modified seamlessly to tune multiple parameters in the same block or different blocks.

The model repair problem is posed as the following non-linear optimization problem: For a system $\mathcal{M}$ and a specification ϕ, find the value of parameter $p \in \mathtt{P}$ such that $[[\phi]](p) > \epsilon$, where ϵ is the user given threshold for minimum robustness to repair the model $\mathcal{M}$. This optimization problem can be mathematically written as

$$p^* = \underset{p \in \mathtt{P} \text{ s.t. } [[\phi]](p) > \epsilon}{\arg\min} [[\phi]](p) \tag{7}$$

We solve the above non-linear optimization problem using a modified version of *Simulated annealing* [26]. It is a probabilistic technique to find an optimal value in large search spaces.

In our algorithm, we start with the default value of parameter p ($p.val$) of model $\mathcal{M}$ and a value α from a given range $[1-\delta, 1+\delta]$. The algorithm has two loops. The outer loop determines how exhaustive our search is (i.e., directions). For instance, $\alpha > 1$ means exploring parameter values greater than the default parameter value. The inner loop determines our computation budget, i.e., how far we want to go in that direction (denoted as H). For every iteration of the outer loop, we pick another α according to the improvement in robustness. We stop the outer loop if there is a convergence between two consecutive values of α. Our aim is to find an appropriate parameter value new_val for the parameter that enables us to repair the model. We denote by $\mathcal{M}(p.val = new\_val)$ the model that is obtained by setting the value of the parameter p to new_val.

In Algorithm 2, we present the model repair procedure formally. In line 1, we extract the model parameters p from the source block of signal s. In line 2, we find the minimum robustness of the model with parameter p with respect to specification ϕ. In line 5 and line 6, we assign a random value to α between $[1-\delta, 1+\delta]$. We run the outer while loop (line 7-20) until the α value converges. In line 11, we compute the minimum robustness value for the model with the new value new_val for parameter p. If the robustness value is positive, we have been able to repair the model successfully (line 12-14). Otherwise, if the robustness value of the model with respect to property ϕ is better for the new value (line 15), we accept it as the current value of parameter p with probability λ (line 16). The inner loop ensures that the minimum robustness increases or remains the same with each iteration. In line 19, we update the value of α based on whether robustness has improved or not in the inner loop. If the robustness improved, i.e., $curr\_rob > old\_rob$, then we choose the new α value close to α_{old}. Otherwise we choose α at a far distance. For example, if $\alpha_{old} > 1$ we choose $\alpha < 1$ and vice-versa. As we show in Section VI-C, the parameters involved in this algorithm can be decided experimentally.

In Algorithm 2, we have considered parameter values as points for the sake of computational efficiency. However, we could use the technique of sensitivity analysis for parameter synthesis for sets (of points), as discussed in [27].

A. Complexity Analysis

The complexity of Algorithm is $O((\mathtt{range}(\alpha)/limit) * H)$. Here $\mathtt{range}(\alpha)$ is the difference between the maximum and the minimum value of α, and H is the number of maximum possible iterations in the inner loop of the algorithm.

B. Example

In Section III, we found the signal `Lin_speed` to have the most impact on the bug. Its source block is a Gain block having parameter `2*pi*Rw` with the default value 6.28 (at `Rw` = 1). The final value of p that fixes the model with respect to the specification $\square(r < r_{max})$ is 6.90 (robustness = 45.48).

V. Automated Debugging

In the previous two sections, we have presented how to localize a fault in a Simulink model with respect to a singleton STL formula and how to debug the Simulink model with respect to the localized fault if it is related to a model parameter. In practice, a Simulink model may have many independent specifications. When we attempt to debug the Simulink model, we need to ensure that after debugging the model satisfies all its specifications. One way to achieve this is to form a new STL formula by taking conjunction of all individual STL formulas. However, once the large formula gets falsified, we need to deal with a large err_signal set, which makes the slicing ineffective, and we need to deal with a large matrix P, which is computationally expensive.

Algorithm 3: `DebugSimulinkModel`

```
Input: A Simulink model $\mathcal{M}$ and a STL property $\phi$
Output: A bug-free Simulink model $\mathcal{M}_r$
1  $\langle \phi_{min}, \omega \rangle \leftarrow$ find_min_violating_subformula ($\mathcal{M}$, $\phi$)
2  while $\phi_{min} \neq$ NULL do
3      $s_{spt} \leftarrow$ bug_localization($\mathcal{M}$, $\phi_{min}$, $\omega$)
4      $\mathcal{M}_r \leftarrow$ model_repair($\mathcal{M}$, $\phi_{min}$, $s_{spt}$)
5      if $\mathcal{M}_r =$ NULL then
6          return 'failure'
7      $\mathcal{M} \leftarrow \mathcal{M}_r$
8      $\langle \phi_{min}, \omega \rangle \leftarrow$ find_min_violating_subformula ($\mathcal{M}, \phi$)
9  return $\mathcal{M}$
10 procedure find_min_violating_subformula ($\mathcal{M}$, $\phi$)
11     $\langle res, \omega \rangle \leftarrow$ falsify($\mathcal{M}, \phi$)
12     $\Phi \leftarrow$ find_atomic_subformulas($\phi$)
13     if $res$ then
14         for $i = 1 : |\phi|$ do
15             $\Phi_i \leftarrow \{ \bigwedge_{j=1}^{i} \phi_j \mid \phi_j \in \Phi \}$
16             for $j = 1 : |\Phi_i|$ do
17                 $res \leftarrow$ check_spec($\Phi_{ij}, \omega$)
18                 if $res$ then
19                     return $\langle \Phi_{ij}, \omega \rangle$
20     else
21         return NULL
```

To deal with the inefficiency with the above-mentioned approach, we propose an incremental algorithm to debug a Simulink model by looking at the subsets of the individual specifications systematically. Our algorithm for incremental debugging of a Simulink model is presented in Algorithm 3. The inputs to our algorithm are the Simulink model $\mathcal{M}$ and the STL property ϕ, which is given in the form $\bigwedge_{i=1}^{n} \phi_i$, where ϕ_i is an atomic subformula. An *atomic subformula* is one that cannot be represented as a conjunction of other subformulas. We refer by a *subformula* or a *subspecification* a conjunction of a set of atomic subformulas. We denote by Φ the set of all atomic subformulas in ϕ. Our goal is to generate a bug-free Simulink model $\mathcal{M}_r$, which satisfies ϕ.

In line 1 of Algorithm 3, we find the minimal violating sub-formula ϕ_{min} in the specification ϕ for model $\mathcal{M}$ by invoking `find_min_violating_subformula`. The function returns a minimal length subformula that gets falsified. In the while loop (line 2- 8), we first find the most suspected signals in the Simulink model $\mathcal{M}$ (line 3). In line 4, we repair the model $\mathcal{M}$ using the signals s_{spt} if possible. If the model repair process fails, we terminate the algorithm with '`failure`'. Otherwise, in line 8, we again find the minimal violating sub-formula in the specification ϕ for the repaired model $\mathcal{M}_r$. The loop continues until we find no further falsification of ϕ.

The function `find_min_violating_subformula` employs a crude method to find a minimal *unsatisfiable core* [28] responsible for falsification. In line 11, we find the trace ω responsible for the falsification of ϕ. In line 14-19, we iteratively find the minimal violating sub-specification Φ_{ij} that is falsified w.r.t. ω. The function `check_spec`(Φ_{ij}, ω) checks if the trace ω also falsifies the sub-specification Φ_{ij}. We can employ more sophisticated techniques to find the minimal satisfiable core, for example, the one presented in [29], where the authors provide a method to find a prime implicant of a temporal logic formula.

A. Complexity Analysis

In the worst case, the debugging algorithm in Algorithm 3 may not terminate as there may be conflicting sub-specifications. Fixing the model for one sub-specification may render the model faulty for another sub-specification for which we debugged the model before. However, we analyze the complexity of the algorithm under the assumption that the sub-specifications are not conflicting, and the bugs can be fixed by the parameter tuning algorithm presented in Section IV-A. Under these assumptions, we can modify the algorithm slightly to ensure that once the model is debugged for a sub-specification, the sub-specification is removed from Φ.

On the basis of the above-mentioned assumptions and the modification in the algorithm, we can analyze the complexity of the modified algorithm as follows. In the worst case, the function `find_min_violating_subformula` will be invoked for all the sub-formulas of ϕ with successful falsification and functions for bug localization and model repair have to be invoked for all of them. Thus, the time complexity of the main algorithm `DebugSimulinkModel` is $O(2^{|\phi|} \cdot (T \cdot |C|^2 + (\texttt{range}(\alpha)/limit) \cdot H))$ (Refer to Section III-B and IV-A for the time complexity of the bug localization and the model repair functions).

B. Soundness and Completeness

The procedure `find_min_violating_subformula` relies on the method `falsify` which is sound (if an STL formula is falsified, then there indeed exists a trace that violates the STL formula) but not complete (the method may not find a trace that violates the STL formula even if there exists one). Due to this reason, even if Algorithm 3 terminates successfully, we cannot conclude with certainty that the model $\mathcal{M}$ satisfies all sub-specifications captured in ϕ. However, if we have access to an STL falsifier that is both sound and complete, our debugging algorithm is also sound despite the fact that our bug localization algorithm may not always be able to localize the bug, and our model repair is not complete. Because, if a localized bug can be repaired successfully, then the bug is indeed a legitimate bug with respect to the falsified specification. Moreover, the termination of Algorithm 3 with a valid Simulink model ensures that we have found and removed all the bugs that might cause falsification of the STL specification ϕ.

The above discussion leads to the following theorem.

Theorem 1 (Soundness)**.** *If the STL falsifier used in Algorithm 3 is both sound and complete, and Algorithm 3 terminates for a Simulink model $\mathcal{M}$ and STL specification ϕ successfully, then the model returned by Algorithm 3 indeed satisfies ϕ.*

Needless to say, Algorithm 3 is not complete due to the incompleteness of our model repair procedure.

VI. Experiments

Model	Ref.	# Blocks	# Signals	Spec
Automatic Transmission (AT)	[17]	70	61	ϕ_1
Abstract Fuel Control (AFC)	[32]	253	187	ϕ_2
Neural Network based MagLev (NNM)	[33]	103	93	ϕ_3
Anti-Lock Braking (ALB)	[34]	41	50	ϕ_4
Helicopter (Heli)	[35]	6	5	ϕ_5

TABLE I: Simulink model benchmarks

Spec	SubSpec	STL formula
ϕ_1	ϕ_{11}	$\square\ (\ v < v_{max})$
	ϕ_{12}	$\square\ (\ r < r_{max})$
	ϕ_{13}	$\square\ \neg\ ((gear == 3) \wedge (v < v_{low}\ ))$
	ϕ_{14}	$\square_{[0,25]}\ (v < v_{max})$
	ϕ_{15}	$\square_{[25,50]}\ (v > v_o)$
	ϕ_{16}	$\square\ ((gear2) \wedge \Diamond_{[.01,.02]}\ (gear1)) \implies \square_{[0,2.5]}\ \neg\ (gear2)$ where $gear1 \equiv gear == 1$, $gear2 \equiv gear == 2$
	ϕ_{17}	$\neg\ (\ \Diamond_{[0,t_1]}\ (v > v_{min}) \wedge \square\ (r < r_{min}))$
ϕ_2	ϕ_{21}	$\square_{[t_{start},t_{end}]}\ AF\_ok$ where $AF\_ok \equiv \neg(AF\_above\_ref \vee AF\_below\_ref)$ $AF\_above\_ref \equiv AF > af\_ref - tol$ $AF\_below\_ref \equiv AF < -af\_ref + tol$
	ϕ_{22}	$\square_{[t_0,t_{end}]}(AF\_above\_ref \implies \Diamond_{[0,t_{stab}]}AF\_abs\_ok)$ where $AF\_abs\_ok \equiv AF < af\_ref + tol$
	ϕ_{23}	$\square_{[t_0,t_{end}]}\ (control\_mode\_check \implies AF\_ok)$ $control\_mode\_check \equiv (controller\_mode == 1)$
	ϕ_{24}	$\square_{[t_0,t_{end}]}(AF\_will\_be\_stable)$, where $AF\_will\_be\_stable \equiv \Diamond_{[0,9]}(AF\_stable)$, $AF\_stable \equiv \square_{[0,1]}AF\_settled$, $AF\_settled \equiv abs(AF[t+dt]\text{-}\ AF[t]) < \epsilon * dt$
ϕ_3	ϕ_{31}	$\square_{[0,t_{end}-\tau]}\ ((\neg close\_ref) \implies reach\_ref\_in\_tau)$, where $reach\_ref\_in\_tau \equiv \Diamond_{[0,\tau]}\ (\square_{[0,\tau_0]}, close\_ref)$, $close\_ref \equiv (Pos - Ref) \leq c\_abs + c\_rel * abs(Ref)$
	ϕ_{32}	$\square_{[0,t_{end}]}\ (\neg\ (far\_ref))$ where $far\_ref \equiv (Pos - Ref) \geq max\_over$
ϕ_4	ϕ_{41}	$\Diamond_{[0,t_{end}]}\ (\square_{[0,3]}(\texttt{abs}\ (Vs - Ww) < thold\ ))$
	ϕ_{42}	$\square_{[0,\tau]}((slp < 1) \wedge (Ww > tol))$
ϕ_5	ϕ_5	$\Diamond_{[0,\tau]}\ (\square_{[0,3]}(\texttt{abs}\ (\dot{\theta} - \psi) < 0.01\ ))$

TABLE II: Specifications for different Simulink models

A. Implementation

Our Simulink model debugging algorithm relies on runtime monitoring of an STL specification. In our implementation, we use BREACH toolbox [9] for this purpose. However, there are other tools, such as S-Taliro [8] and AMT [18], that could also be used in implementing our algorithm. We write a wrapper MATLAB script on top of the BREACH tool to implement our algorithm. We have developed an automated mechanism to flatten a given Simulink model. We implement our own slicing algorithm in Python instead of using *Simulink Design Verifier* (SDV) [30] to achieve full automation since SDV requires manual intervention. All experiments were run on a machine with core $i7$ intel processor, 32 GB RAM, Ubuntu 16.04 OS, MATLAB version $R2017a$, and BREACH version 1.5.2. The implementation of the tool is available at https://github.com/iitkcpslab/Blars.

B. Benchmarks

We carry out our experiments on five Simulink models (ref. Table I). The STL specifications for the models, which are borrowed from [9], [31] are presented in Table II. Though the models have been taken from standard sources, they turned out not to be error-free with respect to several useful STL specifications. Here we provide a brief introduction to the models.

AT. In the Automatic Transmission (AT) model, a Stateflow performs the function of gear selection. The inputs to the model are `Throttle` and `Brake`. Based on these inputs, the speed and the rpm of the vehicle change and the Stateflow shifts the vehicle into different gears. Here the specifications captures different requirements based on the speed of the vehicle, gear and rpm signals (refer Section II for more details). The values of the parameters used in the specifications are as follows: $v_{max} = 160$, $r_{max} = 4500$, $v_o = 0$, $v_{min} = 100$, $r_{min} = 4000$, $t_1 = 10$ and $v_{low} = 30$.

AFC. In the Air-Fuel ratio Control (AFC) model, we describe a four-cylinder spark ignition engine. It contains an air-fuel controller and a model of engine dynamics. The input to the model are `Throttle` and `Speed`. Together they determine the amount of oxygen present in the exhaust gas, which determines the fuel rate (cyl_fuel). The air-mass flow rate (cyl_air) is pumped from the intake manifold. The air-fuel (`AF`) ratio is computed by dividing the air-mass flow rate by the fuel rate. Here the specifications capture various constraints on the signal air-fuel ratio. The values of the parameters used in the specifications are as follows: $af\_ref = 14.7$, $tol = 0.01$, $\epsilon = 0.01$, $dt = 0.1$, $t_{start} = 10$, $t_0 = 5$, $t_{end} = 40$.

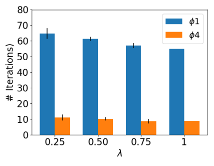
NNM. In the Neural Network based Magnetic Levitation (NNM) model, the controller transforms nonlinear system dynamics into linear dynamics by cancelling the non-linearities (feedback linearization control). The input to the model is the reference signal `Ref`. We train the neural network to represent the forward dynamics of the system. Here, the specifications demand that the plant output `Pos` is consistent with the reference signal `Ref`. The values of the parameters used in the specifications are as follows: $c\_abs = 0.01$, $c\_rel = 0.1$, $max\_over = 2$, $t_{end} = 20$, $\tau_0 = 1$, $\tau = 1$.

ALB. In the Anti-Lock Braking (ALB) model, the signals `Ww` and `Vs` refer to the wheel speed (angular) and vehicle speed, respectively. The signal `Slp` $\in [0, 1]$ is zero when wheel speed and vehicle speed are equal (i.e., no locking) and equals one when the wheel is locked. The goal here is to prevent skidding as much as possible when the brakes are applied. The values of the parameters used in the specifications are as follows: $t_{end} = 15$, $\tau = 10$, $thold = 1$, $tol = 0.1$.

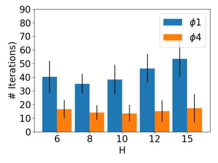
Helicopter. The helicopter model is one of the simplest models for feedback control. The input to the model is the reference angular velocity ψ. Here, we want that the angular velocity $\dot{\theta}$ (actual behavior) is consistent with the reference signal ψ (the desired behavior). The value of τ used in the specification is 5.

C. Results

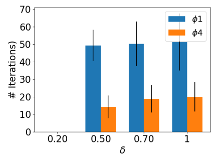
1) Deciding the parameter values for Algorithm 2: Algorithm 2 involves three parameters: λ, H, and δ. The performance of the algorithm depends on the values used for these parameters. Here the performance is measured as the number of iterations required to tune a parameter in a falsified model. As determining the optimal combination of values for the parameters in the algorithm is a complex optimization problem, we instead evaluate the performance of the algorithm for each parameter independently while keeping the values for



(a) λ



(b) H



(c) δ

Fig. 3: Number of Iterations (with error bars) for different values of λ, H, and δ for specifications ϕ_1 and ϕ_4. The default values used in our experiments is $\lambda = 1$, $H = 10$ and $\delta = 0.5$ ($\alpha \in [0.5, 1.5]$). Note that absence of data denotes that bug could not be fixed in 150 iterations.

the other parameters fixed. Figure 3 shows how the number of iterations to solve the parameter tuning problem for the specifications ϕ_1 and ϕ_4 vary with the values of λ, H and δ. Each value is an average of the values obtained from 10 executions. As we can see from the results, the number of iterations decreases monotonically with the increase in the value of λ. For H, we find that both too small values and too large values degrade the performance. For the range of α, we find that the performance of the algorithm is poor for too small range of α. It attains the best performance for $\delta = 0.5$ and remains the same (for ϕ_1) or degrades monotonically (for ϕ_4) when we increase the range. By following the results, we choose $\lambda = 1$, $H = 10$ and $\delta = 0.5$ for all our experiments described below.

2) *Results of bug localization and repair by Algorithm 3:* In Table III, we present the results of bug localization (column 5-7) and model repair (column 8-10) for the Simulink models in Table I. The $Iters$ in the third column refers to the number of iterations in Algorithm 3, i.e. the number of sub-specifications that have been falsified until no violation was found for the entire specification. The $final\_val$ refers to the value of the parameter that repairs the model. Also, $SA - Iters$ refers to the number of iterations of the inner loop in Algorithm 2. For the reference, we also present the results of debugging the Simulink models for each sub-specification independently in Table IV.

AT. In case of AT model, we need three iterations of the loop in Algorithm 3. The model $\mathcal{M}$ is first falsified w.r.t. ϕ_1 with minimal violating subformula as $\phi_{min} = \phi_{12}$. In the first iteration, the bug localization step (line 3) returns $s_{spt} = \{\texttt{Lin\_speed}\}$ and the repair step (line 4) tunes the parameter Rw (<u>R</u>adius of <u>W</u>heel) to fix the model w.r.t. ϕ_{12}. The repaired model $\mathcal{M}_r$ is still falsified w.r.t. ϕ_1 (line 8), but this time the minimal violating sub-formula is $\phi_{min} = \phi_{15}$. In the second iteration, the bug localization step returns us $s_{spt} = \{\texttt{Eii}\}$ and in the repair step, we tune the parameter Iei (<u>E</u>ngine & <u>I</u>mpeller <u>i</u>nertia) to fix the model $\mathcal{M}$ w.r.t. ϕ_{15}. The repaired model $\mathcal{M}_r$ is again falsified w.r.t. ϕ_1, with the minimal violating subformula $\phi_{min} = \phi_{16}$. This time the bug localization step returns $s_{spt} = \{\texttt{Lin\_speed}\}$ and in the repair step we tune the parameter Rw *starting with* the value set in the first iteration, to fix the model w.r.t. ϕ_{16}. The model $\mathcal{M}_r$, hence obtained, satisfies ϕ_1.

Note that we had to increase the value of the parameter Rw to fix the model for ϕ_{12}. However, to fix the model for ϕ_{16} we had to decrease the value of Rw to 3.5369, which also worked for ϕ_{12}. This is because when we fixed the model for ϕ_{12} in the first step, the parameter related to the signal `Eii` had a larger value, which caused a higher value for Rw for fixing ϕ_{12}. Once the value of `Eii` has been set to a lower value of 0.1953 while fixing the model for ϕ_{15}, ϕ_{12} got satisfied for a lower value of Rw.

Now let us explain a bug in detail to justify the accuracy of our bug localization algorithm. Consider specification ϕ_{12} (Table IV) which is violated as `RPM` exceeded its permissible limit. This increase in `RPM` can be avoided if we can enable the gear change before the `RPM` exceeds the value. In Automatic Transmission, the gear change occurs automatically based on the vehicle speed, i.e., based on whether it is greater or less than a threshold value. In this particular case, the repair algorithm increases the value of *radius of wheel* (from 6.28 to 6.908). This means that for the same rpm, we get higher vehicle speed (since $v = r * \omega$). This higher speed causes the gear change a bit earlier, i.e., at a lower `RPM` value, causing the satisfaction of ϕ_{12}.

AFC. In case of AFC model, the parameter is a Gain constant that affects the signal $s_{spt} = \texttt{L13}$ (same as *speed*). The fault here is that the Air-Fuel ratio (`AF`) violates the permissible limit. We know that at a higher speed, the intake air flow increases [32] and hence the `AF` ratio becomes higher. To reduce this ratio, we need to reduce the speed, which we do by tuning the parameter of the Gain block that is source of the signal `L13`.

NNM. In NNM model, the parameters used for repair are constants within a Gain block. Due to not having enough information about the model, we could not explain the physical interpretation of our model repair step for this model.

ALB. In the ALB model, the first sub-specification that gets falsified is ϕ_{42}, which indicates that the wheel speed `Ww` is going below a threshold. Here, the obtained suspected signal is `Brake_torque`, whose source block is a Gain block with the parameter Kf, representing the radius of the piston with respect to the wheel. The wheel speed `Ww` is given as $Ww = \int \frac{Tire\_torque - Brake\_torque}{l} dv$. Since, ϕ_{42} can be satisfied by increasing `Ww`, decreasing the gain of the source block (Kf) reduces `Brake_torque` and increases `Ww`, resulting in the satisfaction of ϕ_{42}.

Subsequently, the model now gets falsified for ϕ_{41}, which indicates that the value of wheel speed `Ww` is not sufficient for the specification to be satisfied. The bug localization algorithm indicates that signal `Filt_rate`, which is the output of a transfer function block representing the hydraulic lag associated with the brake. The parameter tuning algorithm tunes the parameter to reduce its value in such a way that the wheel speed increases sufficiently to satisfy ϕ_{41}.

It is worth noting that both ϕ_{42} and ϕ_{41} require the wheel speed `Ww` to increase but with a different degree. Once the model is fixed for ϕ_{42} by changing the value of Kf, the model got fixed for ϕ_{41} subsequently for a larger value (32.66, closer to the default value) of the constant in the transfer function block in comparison to the value (20.9715) when ϕ_{41} was considered independently (See Table IV). This is because a

$Model$	$Spec$	$Iters$	ϕ_{min}	$Bug\ Localization$			$Model\ Repair$		
				err_signal	$\mid\gamma\mid$	s_{spt}	$default\_val$	$final\_val$	$SA-Iters$
AT	$\bigwedge_{i=11}^{17}\phi_i$	3	ϕ_{12}	`RPM`	28	`Lin_speed`	6.28	6.908	12
			ϕ_{15}	`Speed`	28	`Eii`	50	0.1953	39
			ϕ_{16}	`Gear`	28	`Lin_speed`	6.908	3.5369	4
AFC	$\bigwedge_{i=21}^{24}\phi_i$	1	ϕ_{21}	`AF, AFref`	48	`L13`	9.55	1.6022	9
NN	$\bigwedge_{i=31}^{32}\phi_i$	1	ϕ_{21}	`Pos, Ref`	30	`L32`	12	2.0133	9
ALB	$\bigwedge_{i=41}^{42}\phi_i$	2	ϕ_{42}	`Slp, Ww`	16	`Brake_torque`	1	0.64	3
			ϕ_{41}	`Vs, Ww`	16	`Filt_rate`	100	32.76	6
Heli	ϕ_5	1	ϕ_5	ψ, $\dot{\theta}$	4	`Ctrl_signal`	10	14.641	15

TABLE III: Results of debugging with respect to full specifications

$\mathcal{M}$	$Spec$	$Bug\ Localization$		$Model\ Repair$		
		err_sig	s_{spt}	$parameter\ val$		SA
				$default$	$final$	$Iter$
AT	ϕ_{11}	Not Falsified				
	ϕ_{12}	`RPM`	`Lin_speed`	6.28	6.908	12
	ϕ_{13}	`Gear, Speed`	`Lin_speed`	6.28	2.0578	6
	ϕ_{14}	Not Falsified				
	ϕ_{15}	`Speed`	`Eii`	50	0.19531	39
	ϕ_{16}	`Gear`	`Eii`	50	0.0052	51
	ϕ_{17}	Not Falsified				
AFC	ϕ_{21}	`AF, AFref`	`L13`	9.55	1.6022	9
	ϕ_{22}	`AF, AFref`	`L13`	9.55	3.9117	5
	ϕ_{23}	`AF, AFref, ctrl_m`	`L13`	9.55	1.6022	9
	ϕ_{24}	`AF`	`L13`	9.55	2.5035	7
NN	ϕ_{31}	`Pos, Ref`	`L32`	12	2.0133	9
	ϕ_{32}	`Pos, Ref`	`L32`	12	4.9152	5
ALB	ϕ_{41}	`Vs, Ww`	`Filt_rate`	100	20.9715	8
	ϕ_{42}	`Slp, Ww`	`Brake_torque`	1	0.64	3
Heli	ϕ_5	ψ, $\dot{\theta}$	`Ctrl_signal`	10	14.641	15

TABLE IV: Results when the sub-specifications have been treated independently for debugging

lower value of Kf already pushed the value of wheel speed `Ww` in the desired direction.

Helicopter. In the Helicopter model, the parameter used for repair is the controller scaling factor K. Here, the suspected signal is `Ctrl_signal` whose source block is a Gain block with parameter K. The violation of ϕ_5 is attributed to the deviation of $\dot{\theta}$ from ψ. Since $\dot{\theta} = K * a * \int(\psi - \dot{\theta})dt$ (refer [35]), tuning the parameter K fixes the model.

3) Debugging for individual specification vs. for the set of all specifications: Algorithm 3 enables us to debug a Simulink model for a specification captured as a conjunction of sub-specifications without running the debugging method for all the sub-specifications independently. When we debug the model for one sub-specification, the model becomes correct w.r.t. many other sub-specifications, if they are consistent with each other. For instance, in case of AT model, we needed three iterations to fix the model (Table III), even though the specification has four atomic sub-specifications that cause falsification ($\phi_{12}, \phi_{13}, \phi_{15}, \phi_{16}$) when treated independently (Table IV). Also, as Table IV shows, in case of AFC and NNM models, we get the same s_{spt} during the independent debugging for all the sub-specifications and during the execution of Algorithm 3 with the full specification. For example, for AFC, when we debug the model for the four sub-specifications independently, in each case, we find $s_{spt} = \{\texttt{L13}\}$. When we run Algorithm 3 for the conjunction of the four sub-specifications, it terminates after one iteration by fixing the model for all sub-specifications based on the signal $s_{spt} = \{\texttt{L13}\}$.

For a model, not all its sub-specifications may get falsified when they are treated independently. For instance, for the AT model, the sub-specifications ϕ_{11}, ϕ_{14}, and ϕ_{17} are not falsified by the initial model. Nevertheless, we do not exclude them when we run Algorithm 3 on the model to ensure that fixing the model for some other sub-specification does not make it faulty with respect to the already satisfied sub-specifications. For the AT model, the inclusion of the sub-specifications ϕ_{11}, ϕ_{14}, and ϕ_{17} as part of specification ϕ_1 ensures that the final model also satisfies those specifications.

It is worth noting that if two sub-specifications are conflicting for a model (i.e., there does not exist a common value for a parameter that fixes the model for both the sub-specifications), Algorithm 3 will not terminate. Non-termination of Algorithm 3 indicates a problem in the specification, fixing which is beyond the scope of this paper.

4) Computation Time: In Table V, we present the time spent on each step in the debugging process - namely, Flattening (`Flatten`), Falsification (`Falf`), Bug Localization (`BugLoc`), Model Repair (`Repair`).

$Model$	$Computation\ Time$ (ms)				
	$Flatten$	$Falsif.$	$BugLoc$	$Repair$	$Total$
AT	2762	7783	12004	66865	628428
		12432	8463	457412	
		3051	10567	47089	
AFC	2821	18226	17169	234917	273133
NNM	3021	2038	5614	88284	98957
ALB	2261	1324	6125	21776	89808
		1674	7247	49401	
Heli	2087	730	2982	28256	34055

TABLE V: Computation Time for the debugging with respect to full specifications

D. Comparison

In this section, we compare our bug localization method with a recent work on bug localization in Simulink models [36]. In [36], the authors perform property mining from passing traces and use them to analyze the failures in the model. In a case study presented in [36], the authors introduce a bug in the AT model by changing an entry in a lookup table. We apply our bug localization tool to the erroneous Simulink model and get the most suspected signals to be $s_{spt} = \{\texttt{Nin}\}$. Ideally, s_{spt} should have contained the signal `L6` whose source block is the lookup table. However, the signal `Nin` is the product of signals `Nout` and `L6`. Thus, though our bug localization algorithm cannot point out the exact bug location, it is able to locate a block which is quite close to the source of the bug and our bug localization output should enable an engineer to identify the actual bug easily. This case study shows that our bug localization is also applicable to other bugs than bugs in model parameters.

Our bug localization algorithm is computationally much superior to the algorithm presented in [36]. Our bug localization tool requires 15.45s whereas the technique in [36] requires 239.9s to locate the bug, which is approximately 15 times larger than our computation time.

VII. Related Work

The hybrid systems research community has developed broadly two approaches to address the reliability issues of Cyber-Physical Systems. One approach is formal verification, which is based on set-based reachability analysis techniques. Some of the popular tools in this domain are SpaceEx [37], C2E2 [38] and Flow* [39]. The other approach is rigorous testing, which is often carried out based on the falsification of a specification. The prominent tools in this domain are S-TALIRO [8] and BREACH [9]. We have used the latter approach for bug localization in this paper.

Bug localization [40], [41] has always been one of the significant challenges faced by the Software Engineering community. Even when we are able to discover the presence of bugs (based on falsification or other manifestations of the bug), finding them precisely and fixing them is a tedious job [42].

Recently, various research work have been carried out to develop heuristics for fault localization in Simulink models using statistical debugging techniques iteratively [2], [43]. However, these techniques have been of limited use due to their unpredictability, due to which they need to generate additional test cases [3]. In [44], the focus is on identifying the subset of inputs that are responsible for a counterexample as a whole. The identification of essential inputs may help us in generating more test cases producing violations, which may, in turn, help in fault localization. In [4], the authors use model slicing and spectrum-based fault-localization technique [45] to find bugs in Simulink models. However, they presume the error to be in the Stateflow component of the Simulink model, thereby reducing the problem space. In this paper, we do not make any assumptions on the location of the bug. Though the above-mentioned papers deal with bug localization of Simulink models, unlike our paper, they do not provide any mechanism to use the information provided by their technique to repair the model. This way, it is hard to judge the efficacy of the proposed bug localization techniques. We apply our model-repair technique to fix bugs leading to a violation of complex real-time specifications, which has been lacking in contemporary papers.

In [46], the authors focus on parameter synthesis using reachability analysis to find the set of parameters such that $\mathcal{M} \models \phi$, which is exhaustive but computationally expensive. Other computationally complex methods like sensitivity analysis [27] and abstraction based methodology [47] have been studied for parameter synthesis in CPS models. Unlike these papers, we do not treat a model as a black-box, rather we analyze the model. Also, those computationally expensive techniques are not required for our purpose as we do not need the complete *set* of values for the parameters. We require only one value that fixes the model. In [48], the authors consider parameter tuning associated with lookup maps. They rank the parameters according to their impact on performance. This approach, however, is agnostic to specifications.

VIII. Conclusion

In this paper, we have presented a novel framework for debugging Simulink models. Our debugging framework includes a fully automated mechanism to localize bugs in the models precisely. We also provide a fully automated mechanism to repair the Simulink models using the information generated from bug localization when the bug is due to an inappropriate value for a model parameter. Our approach is based on the rigorous analysis of traces generated by Simulink models, model slicing, and, most importantly, matrix decomposition. We demonstrate the efficacy of the proposed technique by fixing bugs in five Simulink models based on the data generated by the falsification of their specification. Our success in repairing the models demonstrates that our bug localization algorithm can pinpoint the source of the bug precisely.

Despite these positive results, some caveats are worth mentioning here. The repair technique is applicable only if the set of suspected signals contains at least one signal whose source block contains tunable parameters (e.g., Gain block, Transfer Function block). Moreover, though our simplistic model repair technique has been successful in fixing a number of bugs related to parameters in models, it may not be able to fix several other bugs that may require addition, omission or replacement of some blocks and/or connections in the Simulink model in a non-trivial way. In our future work, we will explore more general model repair techniques to deal with this limitation.

Acknowledgment

The authors would like to thank the anonymous reviewers for their detailed review and insightful comments on the originally submitted manuscript. The authors thank Alexandre Donzé for his help related to using BREACH Toolbox. This research work was carried out as part of FMSAFE project supported by MHRD IMPRINT.

References

[1] MathWorks. Simulink - Simulation and model-based design. [Online]. Available: https://in.mathworks.com/products/simulink.html.
[2] B. Liu, Lucia, S. Nejati, L. C. Briand, and T. Bruckmann, "Localizing multiple faults in Simulink models," in *SANER*, 2016, pp. 146–156.

[3] B. Liu, Lucia, S. Nejati, and L. C. Briand, "Improving fault localization for Simulink models using search-based testing and prediction models," in *SANER*, 2017, pp. 359–370.

[4] E. Bartocci, T. Ferrère, N. Manjunath, and D. Ničković, "Localizing faults in Simulink/Stateflow models with STL," in *HSCC*. ACM, 2018, pp. 197–206.

[5] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *AADEBUG*, 2000. [Online]. Available: http://arxiv.org/abs/cs.SE/0012009.

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI*, 2005, pp. 15–26.

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/FSE*, 2005, pp. 286–295.

[8] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *TACAS*, 2011.

[9] A. Donzé, "Breach: A toolbox for verification and parameter synthesis of hybrid systems," in *CAV*, 2010, pp. 167–170.

[10] S. Sankaranarayanan and G. Fainekos, "Falsification of temporal properties of hybrid systems using the cross-entropy method," in *HSCC*, 2012, pp. 125–134.

[11] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, J. Kapinski, and X. Jin, "Falsification of safety properties for closed-loop control systems," in *HSCC*, 2015, pp. 299–300.

[12] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *FORMATS/FTRTFT*, 2004, pp. 152–166.

[13] ——, "Monitoring properties of analog and mixed-signal circuits," *STTT*, vol. 15, no. 3, pp. 247–268, 2013.

[14] R. Koymans, "Specifying real-time properties with Metric Temporal Logic," *Real-Time Systems*, vol. 2, no. 4, pp. 255–299, 1990.

[15] A. Pnueli, "The temporal logic of programs," in *SFCS*, 1977, pp. 46–57.

[16] A. Donzé, T. Ferrère, and O. Maler, "Efficient robust monitoring for STL," in *CAV*, 2013, pp. 264–279.

[17] MathWorks. Modeling an automatic transmission controller. [Online]. Available: https://in.mathworks.com/help/simulink/examples/modeling-an-automatic-transmission-controller.html.

[18] D. Nickovic and O. Maler, "AMT: A property-based monitoring tool for analog systems," in *FORMATS*, 2007, pp. 304–319.

[19] F. Tip, "A survey of program slicing techniques," in *Journal of Programming Languages, vol. 3*, 1995, pp. 121–189.

[20] R. Reicherdt and S. Glesner, "Slicing MATLAB Simulink models," in *ICSE*, 2012, pp. 551–561.

[21] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*, 5th ed. USA: McGraw-Hill, Inc., 2005.

[22] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Johns Hopkins, 1996.

[23] T. F. Chan, "Rank revealing QR factorizations," *Linear Algebra and its Applications*, vol. 88-89, pp. 67 – 82, 1987.

[24] Y. P. Hong and C. T. Pan, "Rank-revealing QR factorizations and the singular value decomposition," *Mathematics of Computation*, vol. 58, no. 197, pp. 213–232, Jan. 1992.

[25] MathWorks. QR decomposition. [Online]. Available: https://in.mathworks.com/help/matlab/ref/qr.html.

[26] S. Zhan, J. Lin, Z. Zhang, and Y. Zhong, "List-based simulated annealing algorithm for traveling salesman problem," *Comput. Intell. Neurosci.*, vol. 2016, pp. 1 712 630:1–1 712 630:12, 2016.

[27] A. Donzé, B. Krogh, and A. Rajhans, "Parameter synthesis for hybrid systems with an application to simulink models," in *HSCC*, 2009, pp. 165–179.

[28] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[29] T. Ferrère, O. Maler, and D. Nickovic, "Trace diagnostics using temporal implicants," in *ATVA*, 2015, pp. 241–258.

[30] MathWorks. Isolating problematic behavior with model slicer. [Online]. Available: https://www.mathworks.com/products/sldesignverifier/features.html.

[31] B. Hoxha, H. Abbas, and G. Fainekos, "Benchmarks for temporal logic requirements for automotive systems," in *ARCH*, vol. 34, 2015, pp. 25–30.

[32] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, "Powertrain control verification benchmark," in *HSCC*, 2014.

[33] MathWorks. Design NARMA-L2 neural controller in simulink. [Online]. Available: https://in.mathworks.com/help/deeplearning/ug/design-narma-l2-neural-controller-in-simulink.html.

[34] ——. Modeling an anti-lock braking system. [Online]. Available: https://in.mathworks.com/help/simulink/slref/modeling-an-anti-lock-braking-system.html.

[35] E.A.Lee and S.A.Seshia, *Introduction to Embedded Systems - A Cyber-Physical Systems Approach, Second Edition*. MIT Press, 2017.

[36] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Nickovic, "Automatic failure explanation in CPS models," in *SEFM*, ser. Lecture Notes in Computer Science, vol. 11724. Springer, 2019, pp. 69–86.

[37] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *CAV*, 2011, pp. 379–395.

[38] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2E2: A verification tool for Stateflow models," in *TACAS*, 2015, pp. 68–82.

[39] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *CAV*, 2013, pp. 258–263.

[40] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[41] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *ICSE*, 2017, pp. 609–620.

[42] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459 – 494, 1985.

[43] B. Liu, Lucia, S. Nejati, L. C. Briand, and T. Bruckmann, "Simulink fault localization: an iterative statistical debugging approach," *Softw. Test., Verif. Reliab.*, vol. 26, no. 6, pp. 431–459, 2016.

[44] R. D. Diwakaran, S. Sankaranarayanan, and A. Trivedi, "Analyzing neighborhoods of falsifying traces in cyber-physical systems," in *ICCPS*, 2017, pp. 109–119.

[45] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[46] T. Dang, T. Dreossi, and C. Piazza, "Parameter synthesis through temporal logic specifications," in *FM*, 2015, pp. 213–230.

[47] S. Bogomolov, C. Schilling, E. Bartocci, G. Batt, H. Kong, and R. Grosu, "Abstraction-based parameter synthesis for multiaffine systems," in *HVC*, 2015, pp. 19–35.

[48] J. Deshmukh, X. Jin, R. Majumdar, and V. Prabhu, "Parameter optimization in control software using statistical fault localization techniques," in *ICCPS*, 2018, pp. 220–231.

Nikhil Kumar Singh (S'20) received his B.Tech. degree in Computer Science and Engineering from the National Institute of Technology Jamshedpur (NIT Jamshedpur), India in 2014. He received his M.S. degree in Computer Science and Engineering from the Indian Institute of Technology Kanpur (IIT Kanpur), India in 2020, where he is currently pursuing his Ph.D. degree. He worked as a Member of Technical Staff at OptumSoft, Bangalore for one and half years. His research interest lies in computer-aided design and verification of cyber-physical systems.

Indranil Saha (M'12) is an Associate Professor in the Department of Computer Science and Engineering at IIT Kanpur. Prior to joining IIT Kanpur in 2015, he was a postdoctoral researcher affiliated with the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley and the Department of Computer and Information Science at the University of Pennsylvania. He obtained his B.Tech. degree in Electronics and Communication Engineering from Kalyani Government Engineering College in 2003, M.Tech. degree in Computer Science from Indian Statistical Institute, Kolkata in 2005, and Ph.D. degree in Computer Science from the University of California Los Angeles in 2013. From 2005 to 2008, he was a research scientist at Honeywell, Bangalore. His research interest lies in the application of formal methods to embedded and cyber-physical systems and robotics. He was a recipient of the Best Paper Award at the ACM SIGBED International Conference on Embedded Software (EMSOFT) in 2010. He received the ACM SIGBED Frank Anger Memorial Award in 2012 for his contribution in the intersection of embedded systems and software engineering.