Specification-Guided Automated Debugging of CPS Models[1]

A thesis submitted in fulfilment of the requirements

*for the degree of **MS-Research***

by

Nikhil Kumar Singh

Roll No: 17211403

under the guidance of

Dr. Indranil Saha

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

July 2021

[1]This work was supported by the FMSAFE project funded by MHRD IMPRINT.

Certificate

It is certified that the work contained in this thesis entitled "Specification-Guided Automated Debugging of CPS Models" by "Nikhil Kumar Singh" has been carried out under my supervision and that it has not been submitted elsewhere for a degree.

Dr. Indranil Saha

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Kanpur

December 2019

Declaration

This is to certify that the thesis titled "Specification-Guided Automated Debugging of CPS Models" has been authored by me. It presents the research conducted by me under the supervision of "Dr. Indranil Saha". To the best of my knowledge, it is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, in part or in full, for a degree. Further, due credit has been attributed to the relevant state-of-the-art and collaborations (if any) with appropriate citations and acknowledgements, in line with established norms and practices.

26-12-19

Nikhil Kumar Singh

December 2019

MS-Research
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur

Abstract

Simulink/Stateflow is the *de facto* tool for developing software for safety-critical real-time cyber-physical systems. 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 catastrophic effect during the execution of the system developed based on the model. Unlike the 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 runtime 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 model slicing technique and a matrix decomposition technique for finding independent signals. Our bug localization technique is precise enough to enable us to repair the model. If the bug is due to the inappropriate value for a model parameter, we employ an automated 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 localize bugs in the Simulink models effectively. All the bugs we find are due to having erroneous values for some parameters. Automated parameter tuning enables us to find the appropriate values for the parameters, which leads to the successful repair of the model.

Acknowledgements

I am deeply grateful to my advisor Dr. Indranil Saha for his guidance and support during the course of this work. His thoughts and ideas has helped me immensely to grow as a researcher. His deep interest in this area has been a source of inspiration to me.

I would like to express my gratitude for my parents who supported me throughout this journey.

I would like to thank CSE Dept, IIT Kanpur for providing me a conducive environment for research.

Nikhil Kumar Singh

Contents

Publication

Nikhil Kumar Singh and Indranil Saha. *Specification-Guided Automated Debugging of CPS Models*. Accepted at ACM SIGBED International Conference on Embedded Software (EMSOFT 2020) September 20 –25, 2020, Hamburg, Germany.

List of Figures

List of Tables

Symbols

$\mathcal{M}$	model
$\mathcal{M}_f$	flattened model
$\mathcal{M}_r$	repaired model
σ	trace
ϕ	specification
γ	sliced set of signals
$\mathcal{P}$	matrix of system states values
τ	timestamps
p	parameter
$p.val$	value of parameter
λ	controlling direction of exploration for new parameter values
δ	amount by which parameter value changes in each iteration of exploration
Γ	sub-specification
ψ	elements within a sub-specification

Dedicated to my Parents.

Chapter 1

Introduction

Development of safety-critical real-time cyber-physical systems pose a tremendous challenge to the engineers in terms of design, implementation, and verification. The *Simulink* tool from Mathworks Inc. [43] is a software package that provides an environment for model-based development of cyber-physical systems (CPSs). 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 a significant progress towards developing software tools for Bug localization in the Simulink models [34, 35, 33, 6]. 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 erroneous model without much manual effort.

There is a variety of techniques available for automated fault localization and debugging of application software [10, 49, 29, 32, 36]. However, little research has been carried out in automating such efforts in case of models of cyber-physical systems. In the recent past, falsification-based techniques [3, 15, 50, 60] have been employed extensively in identifying whether a model is erroneous. These techniques focus on checking whether the simulation traces satisfy a specification given in the form of *Signal Temporal Logic* (STL) [37, 38], 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 thesis, we propose a bug localization algorithm that is based on STL-based falsification of 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 follow four major steps to identify a minimal subset of the signals that contribute to the falsification. First, with respect to the trace that violates the given specification, we recursively split the specification and check each of its sub-specifications that is responsible for violation. In the next step, we perform slicing of the model based on the signals within the minimal violated sub-specification. We proposed our own slicing algorithm instead of using *Simulink Design Verifier* (SDV) [40] to achieve full automation, since SDV requires manual intervention. In the next step, we create a matrix of system data where each column represents a signal and each row represents a time-stamp. This matrix basically captures the state of the systems at all time-stamps where the specification is falsified. This provides us with all the data associated with the specification violation. Now, we remove the redundancy from the matrix using Matrix decomposition techniques. These techniques help us to break a large matrix into simpler matrices that are easier for analysis, and finally help us obtain a minimal set of mutually independent signals. These signals are presented to the user as the root cause of the falsification.

A major class of bugs in Simulink models is related to the values of various parameters in the model. For example, in an automatic transmission model, there is a parameter *threshold* in the value of engine rpm 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 for most of the time, in corner cases, those values may turn out to be wrong. If our bug localization algorithm detects that the root 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.

We evaluate our bug localization and model repair mechanism on five different Simulink models with varying complexity. For each of those models, we consider 1-4 STL specifications 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 precisely.

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

- We present a mechanism for precise localization of a bug in a Simulink model. Our bug localization mechanism relies on the data generated in the process of falsification of an STL specification, and employs techniques for model slicing and finding linearly independent signals.
- We provide a mechanism for repairing an erroneous Simulink model based on the output generated by the bug localization process. Our repair process eliminates the root cause of violation of an STL specification by carefully tuning Simulink model parameters.
- 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 1-4 STL specifications for each of them.

The rest of the thesis is organized as follows. In Chapter 2, we provide the formal problem definition with the required preliminaries and illustrate the problem with a motivating example. In Chapter 3, we present our bug localization algorithm and the model repair mechanism. In Chapter 4, we provide the details of our experiments on applying our algorithms to five different Simulink models. In Chapter 5, we compare and contrast our work with the existing literature and conclude the thesis with an account of future research directions.

Chapter 2

Problem

2.1 Preliminaries

2.1.1 Simulink/Stateflow

Simulink/Stateflow [43] tool consists of a library of *blocks* representing various discrete and continuous mathematical operations such as gain, addition, transforms, lookup tables, and integration. 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 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 connection between blocks, defined as the ordered relation $C \subseteq B \times B$.
- S refers the 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 $\texttt{dst}(c)$. A number of Simulink blocks contain parameters (for example, the Gain block). For a block $b \in B$, its set of parameters is denoted by `param` (b). For a parameter p, its value is denoted by $p.val$.

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 $\langle X, SIM, X_0, U, T\rangle$ that consists of:

- a state-space X where $x \in X$ is a (possibly infinite dimensional) 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 inputs U.
- a time horizon $T > 0$.
- a simulator SIM: $X_0 \times U \times$ $[0,T] \rightarrow X$, $\texttt{SIM}(x_0,u(t'),t)$ denotes the state reached at time t from initial state x_0 using input $u(t')$, $0 \leq t' < t$.

The model $\mathcal{M}$ starts at a state x_0, runs on the input $u(t)$, and generates a trace. A trace σ is defined as the sequence of the 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 σ by $\langle x_0, x_1, \ldots x_T\rangle$ where $x_t = \texttt{SIM}(x_0,u(t'),t)$, $0 \leq t' < t$. 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 $u(t) \in U$ by $\mathcal{L}(\mathcal{M})$.

2.1.2 Signal Temporal Logic

Signal Temporal Logic (STL) [37, 38] is an extension of Metric Temporal Logic (MTL) [30] and Linear Temporal Logic [47]. It enables us to reason about real-time properties of signals (simulation traces). These specifications consists of real-time predicates over the signal variables.

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

$$\phi = \mathsf{true} \mid \pi \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 U_I \phi_2 \tag{2.1}$$

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 sometime in the time 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. The formula $\Diamond_I \phi$ means that the formula ϕ will be `true` sometime in the time interval I. The formula $\Box_I \phi$ means that the formula ϕ will be always `true` in the time interval I. 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 $[0, \infty]$.

We define the distance d between two states as follows:

$$d(x, x') = \begin{cases} \| \, x|_{V_{\mathbb{R}}} - x'|_{V_{\mathbb{R}}} \, \|, & \text{if } \forall v \in V_{\mathbb{D}},\ eval(x, v) = eval(x', v). \\ \infty, & \text{otherwise.} \end{cases} \tag{2.2}$$

Here, $\| \, x - x' \, \|$ is the Euclidean metric that measures the distance between two states x ans x'. Equation 2.2 captures that if all the discrete state variables hold equal values in state x and x', then the distance between the two states is given by the Euclidean distance between the states restricted to only the real state variables. However, if any of the discrete state variables assumes different values in the two states x and x' then the distance between them is ∞.

Robust Semantics of STL: Given a trace $\sigma \in \mathcal{L}(\mathcal{M})$ and $\mathcal{O}{:}\Pi \to 2^X$, we define the robust semantics[16] of ϕ w.r.t. σ at time $t \in N$ by induction as follows:

$$[[\texttt{true}]](\sigma, t) \;=\; +\infty \tag{2.3a}$$

$$[[\pi]](\sigma, t) \;=\; \texttt{Dist}(x_t, \mathcal{O}(\pi)) \tag{2.3b}$$

$$[[\neg\phi]](\sigma, t) \;=\; -[[\phi]](\sigma, t) \tag{2.3c}$$

$$[[\phi \wedge \psi]](\sigma, t) = \texttt{min}\{[[\phi]](\sigma, t), [[\psi]](\sigma, t)\} \tag{2.3d}$$

$$[[\phi U_I \psi]](\sigma, t) = \sup_{t' \in t+I} \texttt{min}\{[[\psi]](\sigma, t'), \inf_{t'' \in [t,t']} [[\phi]](\sigma, t'')\} \tag{2.3e}$$

If $[[\phi]](\sigma, t) \neq 0$, its sign indicates the satisfaction status. Also, if σ satisfies ϕ at time t, any other trace σ' whose euclidean distance from σ is smaller than $[[\phi]](\sigma, t)$ also satisfies ϕ at time t. The robustness metric $[[\phi]]$ maps each simulation trace σ to a real number r. Intuitively, robustness of a trace $\sigma \in \mathcal{L}(\mathcal{M})$ with respect to an STL formula ϕ is the radius of the largest ball centered at trace σ that we can fit within $\mathcal{L}_\phi$, where $\mathcal{L}_\phi$ is the set of all signals that satisfy ϕ.

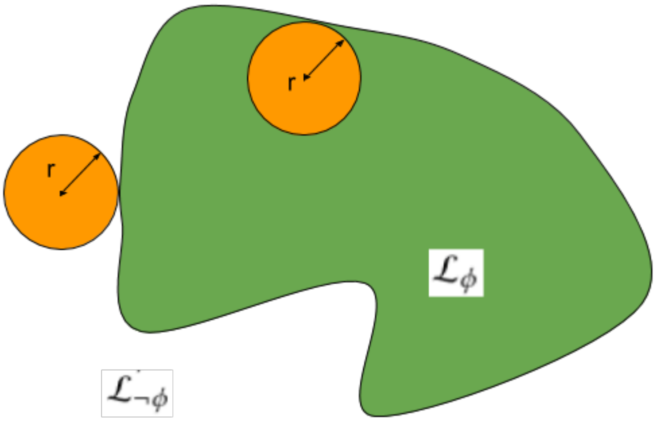


FIGURE 2.1: Robust satisfaction of formulae ϕ

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

$$\sigma^* = \texttt{arg} \min_{\sigma \in \mathcal{L}(\mathcal{M})} [[\phi]](\sigma) \tag{2.4}$$

2.1.3 Linear Independence of vectors

The signals of the system are the basic entity of our analysis in this thesis. Here, we consider each signal as a vector. Thus, all signals together form a vector space [51]. A set of vectors (in a vector space) is said to be *linearly dependent* [56] if at least one of the vectors in the set can be defined as a linear combination of the others. Otherwise, the vectors are said to be *linearly independent*. The concept of linear dependence enables us to determine the *basis* for a vector space. The vectors in a subset $\{v_1, v_2, ..., v_n\}$ of a vector space $\mathcal{V}$ are said to be linearly dependent, if

$$a_1.v_1 + a_2.v_2 + ... + a_n.v_n = 0 \tag{2.5}$$

and atleast one a_i is not equal to 0.

2.2 Problem Definition

In this thesis, 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 the model $\mathcal{M}$

i.e. any trace $w \in \mathcal{L}(\mathcal{M})$ should belong to the language of ϕ, i.e., $\forall \sigma \in \mathcal{L}(\mathcal{M})$, $\sigma \in \mathcal{L}_\phi$. However, if there exists a trace $w' \in \mathcal{L}(\mathcal{M})$ that does not belong to the language of ϕ, i.e., $\sigma' \notin \mathcal{L}_\phi$, 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 thesis are formally presented below.

Problem 1 (Bug Localization). If $\mathcal{M} \not\models \phi$, identify the minimal set of signals $S_{min} \subseteq S$ that can accurately explain the violation of ϕ.

Problem 2 (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 definition, we assume that the specification is correct and the falsification happens due to a fault in the model.

2.3 Motivating Example

We illustrate the problem with an example Simulink model shown in Figure 2.2. It is a model of an Automatic Transmission [42] 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 variables (`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 ω (`RPM` in the model) are always bounded by values v_{max} and ω_{max} respectively. We express this as the following STL specification:

$$\Box(v < v_{max}) \wedge \Box(\omega < \omega_{max}) \tag{2.6}$$

In case the specification is falsified as shown in Figure 2.3(a), the timestamps where specification violation occurs are those where the robustness value is negative. The cause of

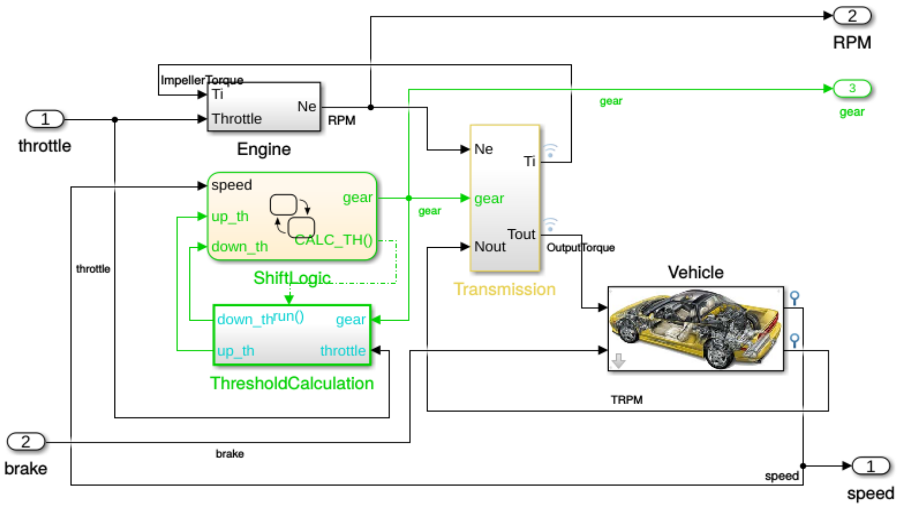
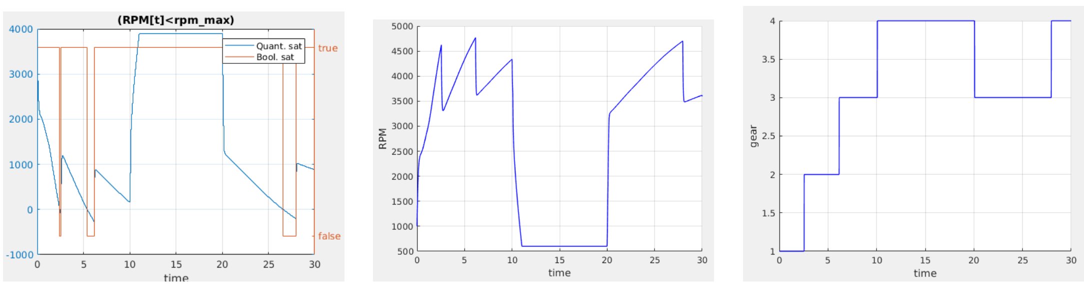


FIGURE 2.2: Simulink model of an Automatic Transmission system

violation is that the RPM (ω) exceeds its maximum permissible value 4500 (Figure 2.3(b)). This event occurs each time when there is a gear change (Figure 2.3(c)). In Automatic Transmission, the gear change happens automatically when the vehicle speed v reaches a threshold. 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.



(a) Trace violating $\omega < \omega_{max}$ (b) RPM signal during violation (c) Gear signal during violation

FIGURE 2.3: Traces for Automatic Transmission Simulink model

We can monitor the simulation traces of model $\mathcal{M}$ for the specifications ϕ using various tools [15, 3, 44]. In case of violation of a specification ϕ, the information provided by these tools is not sufficient for debugging. So, we need an automated procedure that can help us localize the bug in the 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 chapter, we propose algorithms for precise bug localization and model repair. We will illustrate our algorithms using our example.

Chapter 3

Algorithms

In this section, we present an algorithm for bug localization and a mechanism for repair of models based on the information produced by the bug localization algorithm.

3.1 Bug Localization

In this section, we present our bug localization algorithm. The complete listing of the algorithm is provided in Algorithm 1. The main function takes as input a Simulink model $\mathcal{M}$ and a specification ϕ and outputs a set of linearly independent signals s_{ind}. As the first step, we flatten the input Simulink model $\mathcal{M}$ to expand all the subsystems recursively (except atomic subsystems). Flattening of the input Simulink model enables us to get much more precise localization of the bug and hence precise fixes compared to what we would have in the base Simulink model. This may not be evident in simpler models with a very few subsystems but for complex models, having a large number of subsystems, it improves the accuracy of bug localization. The flattened model for Automatic transmission model shown in Figure 3.1.

The bug localization relies on four main operations. First, the algorithm `FindErrorSignal` uses the falsification technique to identify the minimal sub-specification ϕ_{min} that also gets falsified, and collect all the signals in ϕ_{min} in *err_signal* (line 2). Next, we perform model slicing technique to remove the signals that are not related to the signals in *err_signal* using the `SliceSimulinkModel` algorithm (line 3). Then, the algorithm `FindStateAtViolation` (line 4) creates a matrix $\mathcal{P}$ and stores data for all the signals in slice set γ that correspond to negative robustness (falsification). However, the matrix

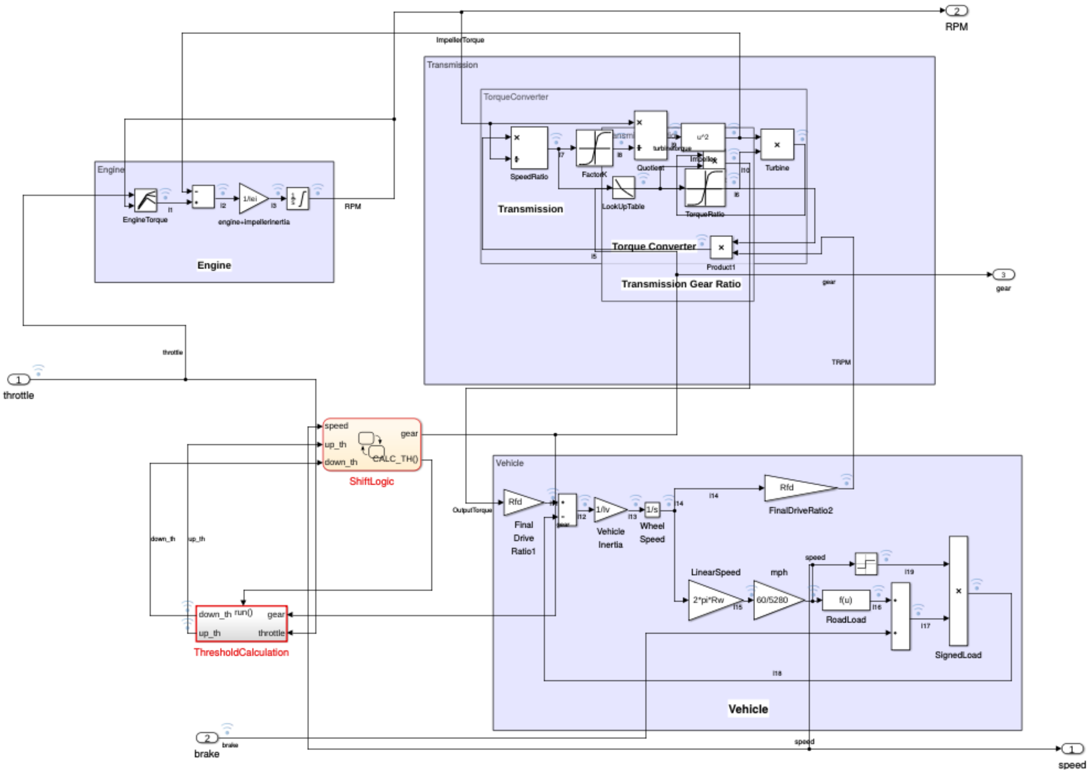


FIGURE 3.1: Flattened Automatic Transmission model

Algorithm 1: `BugLocalization`

Input: A model $\mathcal{M}$ and an STL specification ϕ

Output: The set of independent signals s_{ind} and the set of error signals err_signal

1 $\mathcal{M}_f \leftarrow$ `FlattenModel`$(\mathcal{M})$

2 $\langle \phi_{min}, err\_signal \rangle \leftarrow$ `FindErrorSignal` $(\mathcal{M}_f, \phi)$

3 $\gamma \leftarrow$ `SliceSimulinkModel` $(\mathcal{M}_f, err\_signal)$

4 $\mathcal{P} \leftarrow$ `FindStateAtViolation` $(\mathcal{M}_f, \gamma, \phi_{min})$

5 $s_{ind} \leftarrow$ `FindIndependentSignals` $(\mathcal{P}, \gamma, tol)$

6 **return** $\langle s_{ind}, err\_signal \rangle$

$\mathcal{P}$ still contains many signals which are dependent on each other and further refined by `findIndependentSignals` algorithm (line 5) based on matrix decomposition to find the smallest set of independent signals s_{ind} that contribute to the falsification of ϕ. We now present these four functions in detail.

3.1.1 Flattening

The flattening of Automatic Transmission model is shown in Figure 3.1. The algorithm for flattening a given simulink model $\mathcal{M}$ is given by Algorithm 2. For flattening a given

Algorithm 2: `FlatteningSimulinkModel`

Input: A simulink model $\mathcal{M}$

Output: A flattened model $\mathcal{M}_f$

```
1  all ← listAllSubsystems(M)
2  atomic ← listAtomicSubsystems(M)
3  masked ← listMaskedSubsystems(M)
4  all ← all \ (atomic ∪ masked)
5  for level = 1 to maxdepth do
6      for i = 1 to length(all) do
7          suffix ← get_suffix(all(i))
8          expand_subsystem(suffix)
9      for i = 1 to length(masked) do
10         suffix ← get_suffix(masked(i))
11         maskObj ← get_mask(suffix)
12         save_workspace_variables(maskObj)
13         remove_mask(maskObj)
14         expand_subsystem(suffix)
15 M_f ← save_model_workspace(M)
16 blocks ← list_blocks(M_f)
17 for i = 1 to length(blocks) do
18     ph ← get_outport_handles(blocks(i))
19     for j = 1 to length(ph) do
20         enable_data_logging(phj)
21 return ⟨ M_f ⟩
```

Simulink model, we first retrieve the list of all subsystems, atomic subsystems and masked subsystems by parsing the Simulink XML file (line 1-4). Then, in the first iteration, we flatten all subsystems except atomic and masked subsystems (line 6-8). In the next iteration, we flatten the masked subsystems but after saving the workspace variables and removing the mask (line 9-14). Then we enable data logging for the signals in the flattened Simulink Model (line 17-20). All the steps involved in flattening are fully automated. One requirement for the flattening procedure is that we need all the signals to be annotated. We may choose to skip annotations for some of the signals (in case we already know they aren't suspected), then also our tool works well. The only difference is that it will restrict the search space for suspected signals to the set of annotated signals.

3.1.2 Minimal violating sub-specification and Error Signals

We assume that a specification ϕ is given in the form $\bigwedge_{i=1}^{n} \phi_i$ where ϕ_i is a combination of atomic sub-formulas. An *atomic* formula is one which can't be represented as conjunction

Algorithm 3: `FindErrorSignals`

Input: A Simulink model $\mathcal{M}_f$ and a STL property ϕ

Output: The minimal violating subformulae and a set of faulty signals

for $i = 1 : |\phi|$ **do**
  $\Phi_i \leftarrow$ *set of all subformulas of length i*
  for $j = 1 : |\Phi_i|$ **do**
    $res \leftarrow$ `falsify`(Φ_{ij})
    if res **then**
      $err\_signals \leftarrow$ `stl_extract_signals` (Φ_{ij})
      return $\langle \Phi_{ij},\ err\_signals \rangle$

return `NULL`

of any sub-formula. The above is a reasonable assumption since similar analogy can be found in the area of proposition logic like CNFs.

The Algorithm 3 is similar to finding the unsat core responsible for falsification. We consider the list of all sub-formulas of the original formula of specific lengths one-by-one. Once we find one of the sub-formula or a combination of them that was responsible for falsification, we return it. Thus, for a given falsification (trace), the algorithm will always return the unique sub-formulae or a combination of them. So, in the case of "Or" and "Implication" `FindErrorSignals`, it simply returns the whole formula. The logic is that in ORs, falsification is only if all sub-formulas are falsified (and similarly for implications).

The complexity of this algorithm is $\mathcal{O}(|\phi|.2^{|\phi|})$.

Example. Using the algorithm `FindErrorSignals`, we get

$$\square\ (\omega < \omega_{max})$$

as our minimal violating sub-specification and ω (RPM) as our error signal for the problem defined in Section 2.3. In this case, we have considered a simple specification (eq. 2.6) for the sake of illustration. However, the importance of `FindErrorSignals` function becomes more evident as we move towards complex specifications.

3.1.3 Model Slicing

Model Slicing [48, 55, 52] is a technique which, for a given signal, finds the other signals dependent on it. The given signal is termed as *slicing criterion*. Our goal is to use model

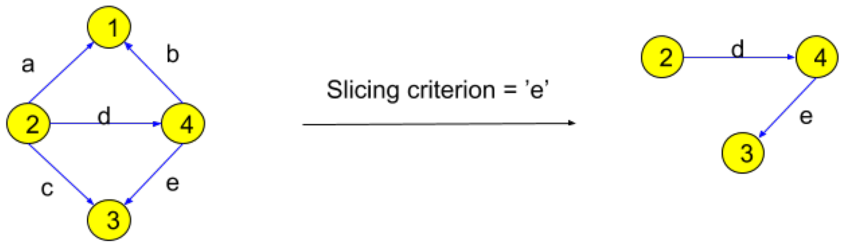


FIGURE 3.2: Model Slicing Technique

Algorithm 4: `SliceSimulinkModel`

Input: A model $\mathcal{M}_f$ and the set *err_signal* as a slicing criterion
Output: The slice set γ

1 $G \leftarrow$ `create_graph` $(\mathcal{M}_f)$
2 $H \leftarrow$ `reverse_graph` (G)
3 $\gamma \leftarrow \emptyset$
4 **for** $e \in err\_signals$ **do**
5     $p \leftarrow$ `dest` (err_Signal)
6     $\gamma \leftarrow \gamma \cup$ `find_reachable_edges` (H, p)
7 **return** γ

slicing to prune all the signals from our model that do not affect the signals in *err_signals*.

Before presenting the algorithm `SliceSimulinkModel`, let us describe the slicing mechanism mathematically. We first convert our Simulink Model $\mathcal{M}$ into a program dependency graph G. In G, the blocks (B) are represented by nodes $(v \in V)$ and signals/connections (C) are represented by edges $(e \in E)$. We define $\texttt{dep}(v)$ as the set of nodes on which the node v is dependent i.e. there exists a path from such a node v' to v. Mathematically, $\texttt{dep}(v) = \{v' \mid v' \in V.v' \rightsquigarrow v\}$, where $v' \rightsquigarrow v$ denotes the path from v' to v in G. We define the *slice* of signal s as the set of all edges (signals) that are present on path $v \in V$ to $\texttt{dst}(s)$. Mathematically, we can define

$$\texttt{slice}(s) = \{e \mid \exists\, u, v \in \texttt{dep}(\texttt{dst}(s)) \text{ and } e \in E \text{ such that } u \xrightarrow{e} v\}.$$

For example, in Figure 3.2, $\texttt{slice}(e)$ is $\{d,e\}$.

In algorithm `SliceSimulinkModel`, we start by creating a graph G for the model $\mathcal{M}$ where the nodes in the graph G correspond to the blocks in B and the edges correspond to the connections in C (line 1) (refer Figure 3.3). In line 2, we reverse the graph by reversing direction of the edges of the graph G. In line 4-6 we find signals upstream of any signal

$e \in err\_signals$ using a graph reachability algorithm (like DFS) and store them in γ. In line 7, we return the set γ.

The complexity of this algorithm is $\mathcal{O}(|B| * |C| + |C|^2)$, since $err\_signal \subseteq C$. Here $|B|$ is the number of blocks and $|C|$ is the number of connections in model $\mathcal{M}$.

Example. From algorithm `FindErrorSignals`, we found our `err_signal` to be `RPM` (denoted as ω). The slice set γ (line 7) consists of the following set {`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 `drive_ratio` lie within Engine subsystem and vehicle subsystem respectively.

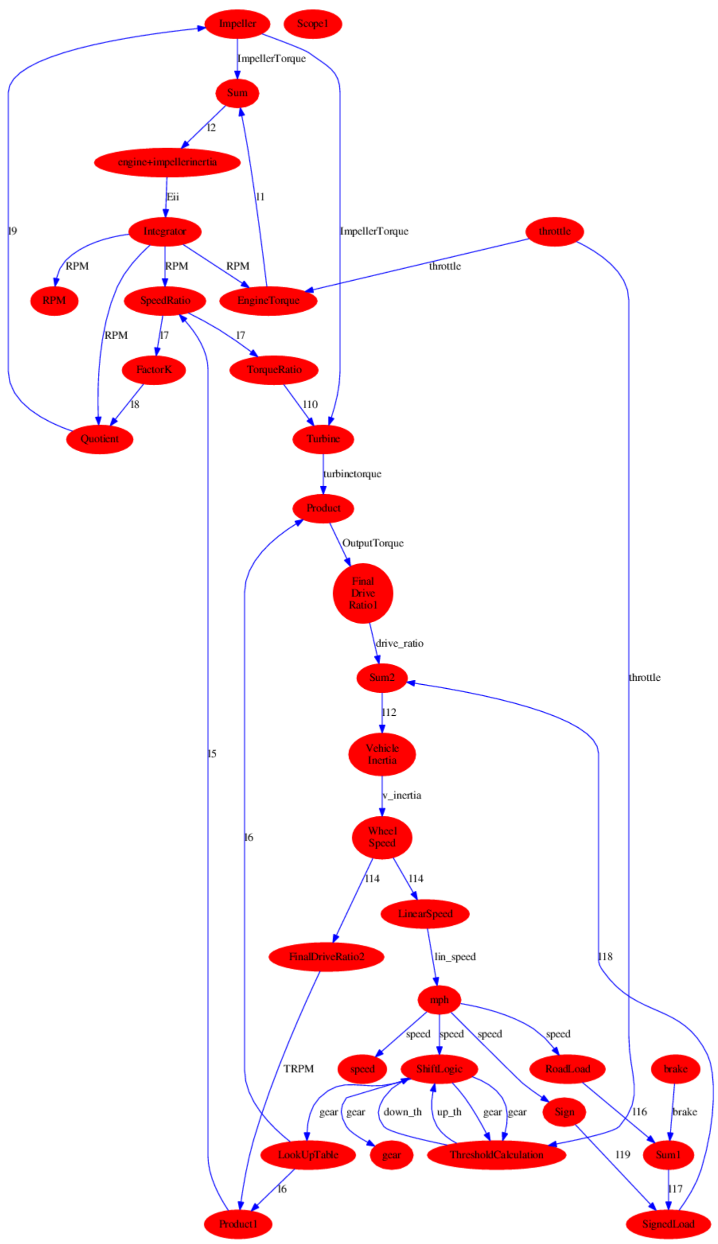


FIGURE 3.3: Graph generation for flattened Automatic Transmission Simulink model for model slicing

3.1.4 Finding system states at violation

In algorithm `FindStateAtViolation`, we find the values of system state at each time-stamp where specification violation occurs i.e. the part of trace (epochs) where robustness of ϕ is negative. In line 1, we create a matrix M with one column containing all the time-stamps τ. In line 2-4, we add values column-wise to M with each column representing the

Algorithm 5: `FindStateAtViolation`

Input: A model $\mathcal{M}_f$, slice set γ and minimal violating formulae ϕ_{min}

Output: A matrix $\mathcal{P}$ consisting of states of sliced model $\mathcal{M}_f$ at timestamps where specification violation occurs

1 $M \leftarrow [\tau]$
2 **for** $i = 1 :$ `length` (γ) **do**
3     $y \leftarrow$ `plot_sig_portrait` $(\gamma[i])$
4     $M \leftarrow [M\ y]$
5 $z \leftarrow$ `plot_robust_sat` (ϕ_{min})
6 $N \leftarrow [\ ]$; $j = 1$
7 **for** $i = 1 :$ `length` (z) **do**
8   **if** $z[i] < 0$ **then**
9         $N[j] \leftarrow [\tau[i]\ z[i]]$; $j \leftarrow j + 1$
10 $\mathcal{P} \leftarrow$ `join`(M, N)
11 **return** $\mathcal{P}$

signals in γ for all time-stamps. In line 5, we plot the robustness of minimal violating sub-specification ϕ_{min}. In line 7- 9, we store all the time-stamps where specification violation occurs in matrix N. The plot in line 3 differs from the plot in line 5. While the earlier is a simple portrait of values of a signal, the latter is the robust satisfaction of a signal. In line 10, we join (INNER JOIN) M and N on the field time and store it in matrix $\mathcal{P}$. The INNER JOIN 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.

The complexity of this algorithm is $\mathcal{O}(|C| * |\tau|)$, since $\gamma \subseteq C$ and $|\tau| = |z|$. Here, τ (or $|z|$) is the length of trace of the model(line 5).

Example. Using algorithm `FindStateAtViolation`, the matrix M (line 26) for the Automatic Transmission model is given by

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

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 lin_speed signal (within Vehicle subsystem).

The matrix N is given by

$$\begin{array}{cc} \tau & \rho \\ \left(\begin{array}{cc} \vdots & \vdots \\ 6.11 & -251.41 \\ 6.12 & -254.8 \\ 6.13 & -258.18 \\ 6.14 & -261.5 \\ 6.15 & -264.92 \\ 6.16 & -268.28 \\ \vdots & \vdots \end{array}\right) \end{array}.$$

In the table below, we show only part of $\mathcal{P}$ due to lack of space. Here, the first column is the timestamp(τ) and last column is the robustness value(ρ).

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

3.1.5 Linear Independence

In spite of using the technique of model slicing, we still have a large set of signals. We need to remove redundancy from this set for which we use the concept of *linear independence* of vectors. Here, we represent each signal of the Simulink model as a vector and then find the linearly independent ones amongst them. This minimal set of *vectors* (signals) that contains no dependent vector forms the *basis* of the vector space as described in Section 2. We use matrix decomposition technique to find the set of independent signals. *Matrix decomposition* [57] is representing a large matrix (say A) as a product of simpler matrices.

Algorithm 6: `FindIndependentSignals`

Input: A matrix $\mathcal{P}$, the sliced set γ, and a tolerance limit *tol*
Output: The list of linearly independent signals s_{ind}

```
1 [Q, R, E] ← matrix_decomposition (P)
2 diagr ← abs (diag(R))
3 maxindex ← find_max_index (diagr, tol)
4 s_ind ← [ ]
5 for  i = 1 to maxindex do
6     s_ind[i] ← γ[E[i]]
7 return s_ind
```

In algorithm `FindIndependentSignals`, we find the linearly independent columns in the matrix $\mathcal{P}$. In line 1, we perform the Matrix Decomposition of $\mathcal{P}$. Specifically, we used Orthogonal triangular decomposition [53] of matrix $\mathcal{P}$. It expresses the matrix as the product of a real orthonormal or complex unitary matrix and an upper triangular matrix. It produces an economy-size decomposition in which E is a permutation vector, so that $\mathcal{P}(:, E) = Q \times R$. Here $\mathcal{P}$ is an $m \times n$ matrix which produces an $m \times n$ unitary matrix Q and an $n \times n$ upper triangular matrix R. In line 2, we store the diagonal elements of the matrix R in *diagr*. In line 3, we find the maximum index *maxindex* corresponding to elements satisfying the condition $diagr \geq tol{*}diagr(1)$. This condition implies that the diagonal elements in *diagr* should be greater than the tolerance value *tol*. In line 4, we create an empty vector s_{ind}. In line 5-6, we add elements to vector s_{ind} by mapping the indices from E to the set γ. The vector s_{ind} contains the set of linearly independent signals of $\mathcal{P}$. Intuitively, the s_{ind} forms the *basis* of the vector space represented by $\mathcal{P}$. The important point here is that the number of independent vectors is the same in matrix $\mathcal{P}$ and R. Thus, we have simplified our problem of finding independent vectors in a complex matrix $\mathcal{P}$ to a simple matrix R

The complexity of this algorithm is $\mathcal{O}(|\tau| * |C|^2)$, since $\gamma \subseteq C$. Note that, for qr decomposition the complexity is $\mathcal{O}(m * n^2)$ where $\mathcal{P}$ is an m×n matrix and $m = |\tau|$ and $n = |\gamma|$.

The Linear Independence is an important part of the bug localization algorithm. As we can see in Table 4.2, it significantly reduced the number of signals that we get after model slicing. One point worth mentioning here is that the "independent signals" phrase doesn't mean that the signals are independent and hence give us different bugs. Instead, what we mean is that the SET consists of signals that these have significant impact on the localized bug. Intuitively, we used matrix decomposition method to split the system into a set of linearly independent components. An analogy to this can be found in image compression where a digital image is represented by a matrix of pixel values. The Matrix Decomposition is employed to the image matrix. By deleting some of nonzero singular values and keeping only the first k large singular values out of the r singular values, the image size can be reduced.

Example. Using algorithm `FindIndependentSignals`, we get the *diagr* (line 2) as:

$$diagr = [343670\ 126540\ 96160\ 47880\ 24080\ 6530\ 2150\ ....\ 0]^{\mathsf{T}}$$

Here, the tolerance value *tol* is 0.01. The matrix E is given as $E = [24\ 1\ 22\ 13\ 3\ 5\ 7\ ....\ 18]$. The *maxindex* value that we get is 6 (since diagr(1) ... diagr(6) are $\geq$ 3436). Finally, we get s_{ind} = {`lin_speed`, `Eii`, `l8`, `l12`, `Nin`, `RPM`}

Recursive flattening of the subsystem $Vehicle$ gives us the signal *lin_speed*, and of subsystem *Engine* gives us signal *Eii*. This implies much better bug localisation in the Simulink Model. Hierarchical flattening in this case is computationally expensive than complete flattening, since we would need to repeat falsification process a number of times.

3.2 Model Repair

The bug localisation algorithm provides us with a set of signals that are root cause of the falsification. Since the value of a signal depends on its source block, we focus on the parameters of source block of the suspected signals and attempt to fix the model by tuning its parameters. In our repair mechanism, we work with a single-fault assumption, which ensures that the value of at most one parameter is erroneous.

We start with the default value of parameter p ($p.val$) of model $\mathcal{M}_f$. Our aim is to find an appropriate parameter value v for the parameter that enables us to repair the model. We denote by $\mathcal{M}_f(p.val = v)$ the model that is obtained by setting the value of the parameter p to v.

We explore the neighbourhood of $p.val$ iteratively by unit δ until we find v such that the model cannot be falsified for this new value for parameter p any more. We use a proposer scheme (`PS`) [1] that generates a new value v for the parameter p in the neighbourhood of its current value, such that $p.val = p.val + \delta * \lambda$. Here, λ is the controlling factor that

Algorithm 7: `ParameterTuning`

Input: A model $\mathcal{M}_f$, set of signals s_{ind} and an STL specification ϕ
Output: A model $\mathcal{M}_r$ that satisfies ϕ

1 **for** $s \in s_{ind}$ **do**
2     $par \leftarrow$ `param` (`src`(s))
3   **for** $p \in par$ **do**
4         $curr\_rob \leftarrow \texttt{sim}(\mathcal{M}_f) \models \phi)$
5         $default\_val \leftarrow p.val$
6         $v \leftarrow \texttt{PS}\ (p)$
7   **while** $v \neq \emptyset$ **do**
8             $new\_rob \leftarrow \texttt{sim}(\mathcal{M}_f(p.val = v) \models \phi)$
9   **if** $new\_rob > 0$ **then**
10                 $\mathcal{M}_r \leftarrow \texttt{save\_system}\ (\mathcal{M}_f(p.val = v))$
11   **return** $\mathcal{M}_r$
12   **if** $new\_rob > curr\_rob$ **then**
13                 $p.val \leftarrow v$
14                 $curr\_rob \leftarrow new\_rob$
15             $v \leftarrow \texttt{PS}\ (p)$
16         $\mathcal{M}_f \leftarrow \mathcal{M}_f(p.val = default\_val))$
17   **return** $\emptyset$

determines the direction of exploration. For instance, $\lambda = +1$ gives us parameter values greater than $p.val$ and $\lambda = -1$ gives us parameter values less than $p.val$.

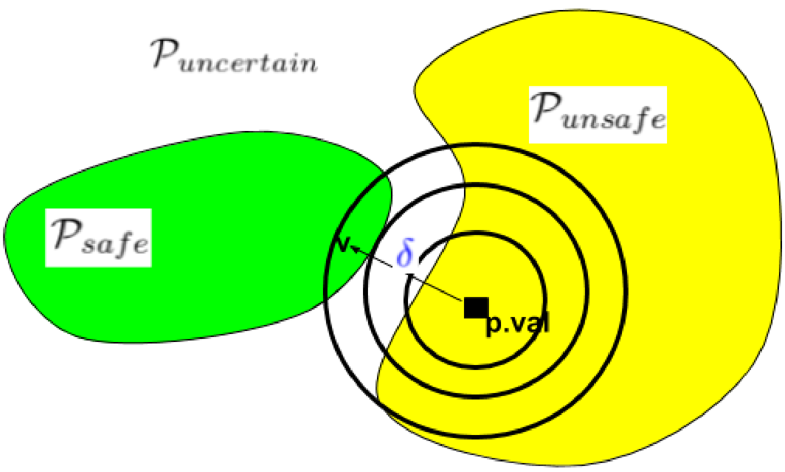


FIGURE 3.4: Parameter Tuning

In Algorithm 7, we present the model repair procedure formally. We start with a signal s from the set s_{ind} (line 1). In line 2, we extract the set of model parameters par from the source block of signal s and explore each of the parameter p in it iterativelty (line 3). In line 4, we get the robustness of the model with parameter p with respect to specification ϕ. In line 6, we generate a new value v of parameter p using the proposer scheme `PS`. If the proposer scheme is not capable of producing a new value for parameter p, we restore the model with the default value for parameter p (line 16) and attempt to fix the model by tuning the value of another parameter. If all the parameters of the source block of the current signal have been examined, we attempt the same procedure with another signal in s_{ind}.

In line 7-15, we compute the robustness value for the model with the new value v for parameter p. If the robustness value is positive, we have been able to repair the model successfully (line 9-11). Otherwise, if the robustness value of the model with respect to property ϕ is better for the new value (line 12), we accept it as the current value of parameter p (line 13), and the search for a new value is continued in the neighbourhood of the new value of p.

Example. In Section 3, 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 $\Box(r < r_{max})$ is 6.90 (robustness = 45.48).

In Algorithm 7, 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 as discussed in [17].

Chapter 4

Experiments

4.1 Implementation

Our bug localization algorithm relies on runtime monitoring of an STL specification. In our implementation, we use BREACH toolbox [15] for this purpose. However, there are other tools such as S-Taliro [3] and AMT [44] 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 also implement some components of our tool as Python code, which we call from Matlab script directly, making the whole process automated.

For processing the Simulink model to produce the graph G in the procedure `SliceSimulinkModel`, we convert the standard Simulink *slx* file into an *xml* file using Matlab functions. Then we parse the information from this *xml* file using a Python script. In the implementation of the procedure `FindStateAtViolation`, we retrieve the time-series data of the signals using Matlab functions. In implementing the `FindIndependentSignals` function, apart from the specific decomposition technique(QR) mentioned there, we also explored other matrix decomposition techniques like LU-decomposition and Singular Value Decomposition. Cholesky decomposition is not applicable in this case as the matrix is not a positive definite matrix. We found that QR-decomposition technique was better than or at least as good as other techniques in all the cases.

4.2 Benchmarks

We carry out our experiments on five Simulink models — Automatic transmission (aka Autotrans) [42], Abstract fuel control (aka AFC) [Jin et al.], Neural Network based magnetic levitation (aka NN-maglev) [39], Anti-lock braking system (aka Absbrake) [41] and a helicopter model [21]. Here we provide a brief introduction to the models.

Autotrans. In the Autotrans model (Figure 2.2), a Stateflow performs the function of gear selection. The inputs to the model are *throttle* and *brake*. Based on these inputs, the vehicles speed and rpm changes and the Stateflow shifts the vehicle into different gears. Here, the specifications captures different requirements based on speed of the vehicle(v), gear and rpm(ω) signals (refer Section 2 for more details).

AFC. In the AFC model (Figure 4.1), 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, specifications captures various constraints on the signal air-fuel ratio (AF).

NN-maglev. In the NN-maglev model (Figure 4.2), the controller transforms nonlinear system dynamics into linear dynamics by canceling 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 reference Ref signal.

Absbrake. In Absbrake model (Figure 4.3), the input to the model is the desired relative slip des_slip. The signal Ww and Vs refers to the Wheel speed (angular) and vehicle speed respectively. The goal here is the prevent skidding as much as possible when brakes are applied.

Helicopter. The helicopter model (Figure 4.4) is one of the simplest model for feedback control. The input to the model is the reference angular velocity psi. Here, we want that the angular velocity $theta\_dot$ (actual behavior) is consistent with the reference signal psi (desired behavior).

The 10 STL specifications used in our experiments are presented in Table 4.1. For the specifications in Table 4.1, we use the following values for parameters: $v_{max} = 160$, $\omega_{max} =$

Simulink Model	*Spec*	*Description*	*Meaning*
Automatic Transmission [42] (70 blocks, 61 lines)	ϕ_1	$\Box\ (\ v < v_{max})\ \wedge \Box\ (\ \omega < \omega_{max})$	The speed and rpm should remain below a threshold.
	ϕ_2	$\Box\ \neg\ ((gear == 3) \wedge (v < v\_low\ ))$	The speed is never below v_low while being in the third gear.
	ϕ_3	$\Box_{[0,25]}\ (v < v_{max}) \wedge \Box_{[25,50]}\ (v > v_{min})$	For the first 25 seconds, speed remains below v_{max} and for the next 25 seconds it remains above v_{min}.
Abstract Fuel Control [Jin et al.] (253 blocks, 187 lines)	ϕ_4	$\Box_{[t\_start,t\_end]}\ AF\_ok$, where $AF\_ok \equiv \neg(AF\_above\_ref \vee AF\_below\_ref)$, $AF\_above\_ref \equiv AF > af\_ref - tol$, $AF\_below\_ref \equiv AF < -af\_ref + tol$	AF is always within permissible range
	ϕ_5	$\Box_{[t\_0,t\_end]}(AF\_above\_ref \implies \Diamond_{[0,t\_stab]}(\ AF\_abs\_ok))$, where $AF\_above\_ref \equiv AF > af\_ref - tol$, $AF\_abs\_ok \equiv AF < af\_ref + tol$	If AF is outside the permissible range, then it eventually converges within this range in a given time
	ϕ_6	$\Box_{[t\_0,t\_end]}\ (control\_mode\_check \implies AF\_ok)$, where $control\_mode\_check \equiv$ (controller_mode $== 1$)	If controller mode is 1 then AF is within the permissible range
	ϕ_7	$\Box_{[t\_0,t\_end]}(AF\_will\_be\_stable$, where $AF\_will\_be\_stable \equiv \Diamond_{[0,9]}(\ AF\_stable)$, $AF\_stable \equiv \Box_{[0,1]}\mathrm{AF\_settled}$, $AF\_settled \equiv abs(AF[t+dt] - AF[t]) < epsi * dt$	AF will be stablilised eventually
Neural Network based MagLev [39] (103 blocks, 93 lines)	ϕ_8	$\Box_{[0,t\_end-tau]}\ ((\neg\ close\_ref) \implies reach\_ref\_in\_tau)$, where $reach\_ref\_in\_tau \equiv \Diamond_{[0,tau]}\ (\Box_{[0,tau0]}\ (close\_ref))$ $close\_ref \equiv$ (Pos - Ref) $\leq c\_abs + c\_rel * abs(Ref)$	The position values always converges to Ref in tau seconds
	ϕ_9	$\Box_{[0,t\_end]}\ (\neg\ (far\_ref))$ where $far\_ref \equiv (Pos - Ref) \geq max\_over$	Position value is never far from Ref value
Anti-Lock Braking [41] (41 blocks, 50 lines)	ϕ_{10}	$\Diamond_{[0,t\_end]}\ (\Box_{[0,3]}(\texttt{abs}\ (Vs - Ww) < thold\ ))$	The wheel speed Ww eventually becomes equal to Vehicle speed Vs and remains stable for at least 3s
	ϕ_{11}	$\Box_{[0,tau]}(slp < 1) \wedge (Ww > tol))$	There is no locking while the vehicle is still moving
Helicopter [21] (6 blocks, 5 lines)	ϕ_{12}	$\Diamond_{[0,tau]}\ (\Box_{[0,3]}(\texttt{abs}\ (theta\_dot - psi) < 0.01\ ))$	Angular velocity ($theta\_dot$) converges to the psi value

TABLE 4.1: Specifications of different Simulink models

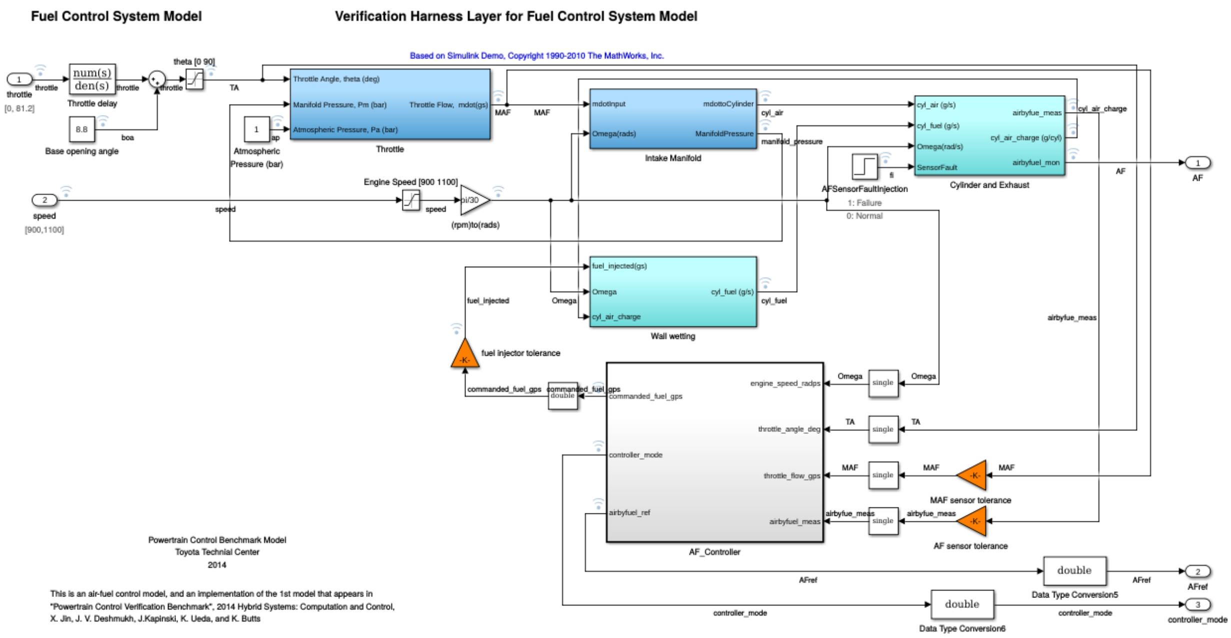


FIGURE 4.1: Fault tolerant Fuel Control model

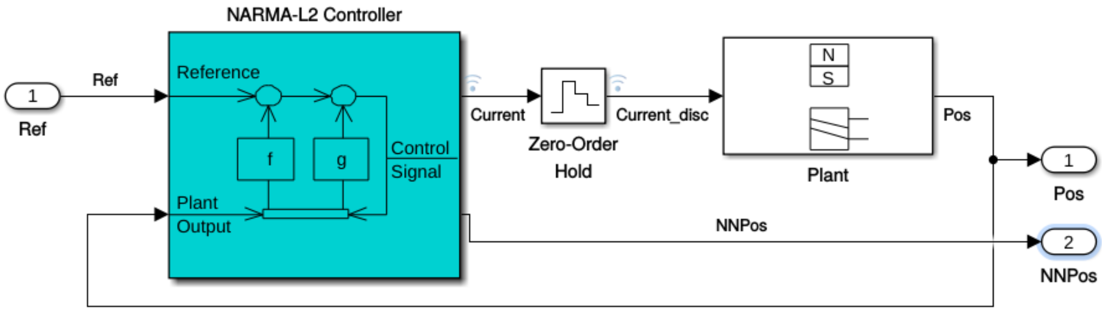


FIGURE 4.2: Neural Network based Magnetic Levitation model

4500, $v_{min} = 0$, $v_{low} = 30$ (Autotrans); $af\_ref = 14.7$, $tol = 0.01$, $epsi = 0.01$, $dt = 0.1$, $t\_start = 10$, $t\_0 = 5$, $t\_end = 40$ (AFC); $c\_abs = 0.01$, $c\_rel = 0.1$, $max\_over = 2$, $t\_end = 20$, $tau\_0 = 1$, $tau = 1$ (NN MagLev); $t\_end = 15$, $tau = 10$, $thold = 1$, $tol = 0.1$ (Absbrake); $tau = 5$ (Helicopter).

4.3 Results

We apply our algorithms described in Section 3 to find the root cause of falsification of the specifications defined in Table 4.1. The results are presented in Table 4.2. We

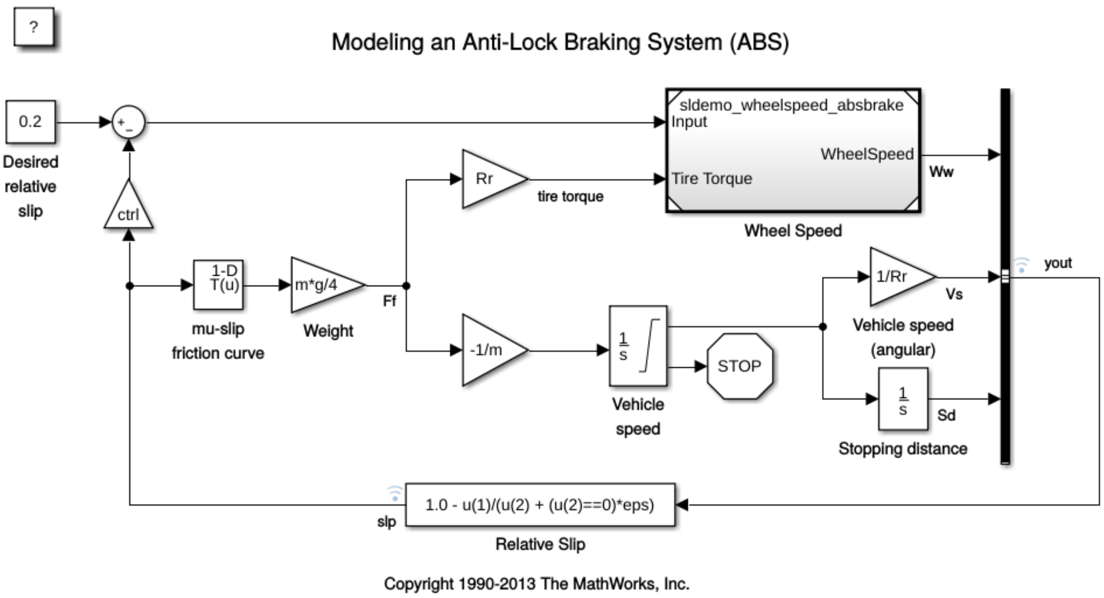


FIGURE 4.3: Anti-Lock Braking model

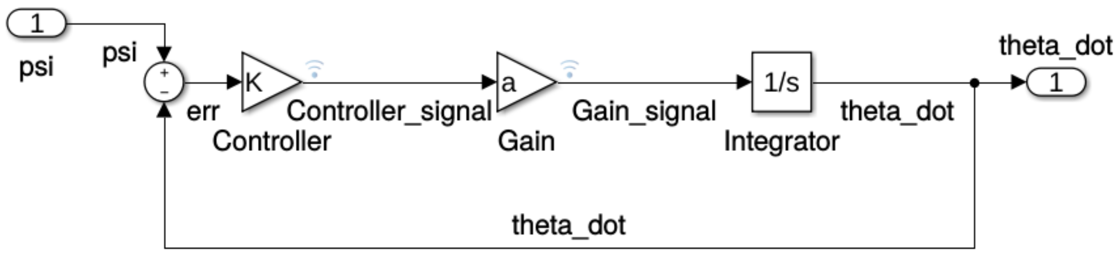


FIGURE 4.4: Helicopter model

have a column *Explanation* in this table, where we attempt to provide the reason of the falsification based on the values of the system states at the minimum robustness value. For example, ϕ_1 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_1.

In case of Automatic transmission model, the parameters that have been used in repair are Iei and Rw referring to $\underline{E}$ngine & $\underline{I}$mpeller $\underline{i}$nertia and $\underline{R}$adius of the $\underline{w}$heel respectively.

Spec	*err_signal*	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{X}\rvert$	$\lvert s_{ind}\rvert$	*FaultExplanation*	*Source Block*	*Repair/Parameter Tuning*
ϕ_1	`RPM`	51	29	4	High RPM	Gain block (2*pi*Rw)	change parameter Rw=1 to Rw=1.1
ϕ_2	`gear, speed`	51	29	2	Low Output Torque	Gain block (2*pi*Rw)	change parameter Rw=1 to Rw=1/3
ϕ_3	`speed`	51	29	1	Low RPM	Gain block (1/Iei)	change value Iei=0.02 to Iei=5
ϕ_4	`AF, AFref`	71	48	1	High speed	Gain block (30/pi) =9.55	Change the parameter from 9.55 to 8
ϕ_5	`AF, AFref`	71	48	2	Low throttle	Function block (2.8-0.05*u + 0.10*u*u - 0.00063*u*u*u)	Change parameter 0.10 to 0.05
ϕ_6	`controller mode`	71	47	2	High speed	Gain block (30/pi) =9.55	Change parameter 9.55 to 8
ϕ_7	`AF`	71	47	2	Low throttle	Function block (2.8-0.05*u + 0.10*u*u - 0.00063*u*u*u)	Change parameter 0.10 to 0.05
ϕ_8	`Pos, Ref`	54	30	2	Position value not converging	Gain block (15)	Change parameter 15 to 2
ϕ_9	`Pos, Ref`	54	30	2	Position value away from ref	Gain block (15)	Change parameter 15 to 2
ϕ_{10}	`Vs, Ww`	31	17	3	Wheel speed doesnt converge with Vehicle speed	Gain block(Kf)	Change parameter Kf=1 to K=0.2
ϕ_{11}	`slp`	31	17	3	Wheel gets locked	Transfer function (param num=100, denom =0.01)	Change parameter 100 to 500
ϕ_{12}	`psi, thetadot`	4	4	1	Thetadot doesn't converge with psi	Gain block(K)	Change parameter K=10 to K=20

Table 4.2: Results for each specification

One point worth noting here is that for specification ϕ_1 and ϕ_2, we tune the same parameter Rw, but end up with different values. In this thesis, we consider all the specifications independently. However, we can always combine two or more specifications into a single specification.

In case of Abstract Fuel Control model, the parameters that have been tuned are constants within the gain block and the function block. In case of ϕ_5, the suspected signal depends on the function block which represents a function of throttle angle θ [Jin et al.]. This function is given as a polynomial with four coefficients. The violation of ϕ_5 is attributed to low throttle, which doesn't make AF eventually within permissible range (Table 4.1). Hence, tuning one of the coefficients fixes the bug. For the specifications ϕ_5 and ϕ_7, the source block is a `function` block. Since this block consists of multiple parameters, we convert this block into a subsystem in the base model and then flatten it. The suspected signal generated by the bug localisation algorithm are among the signals inside the `function`

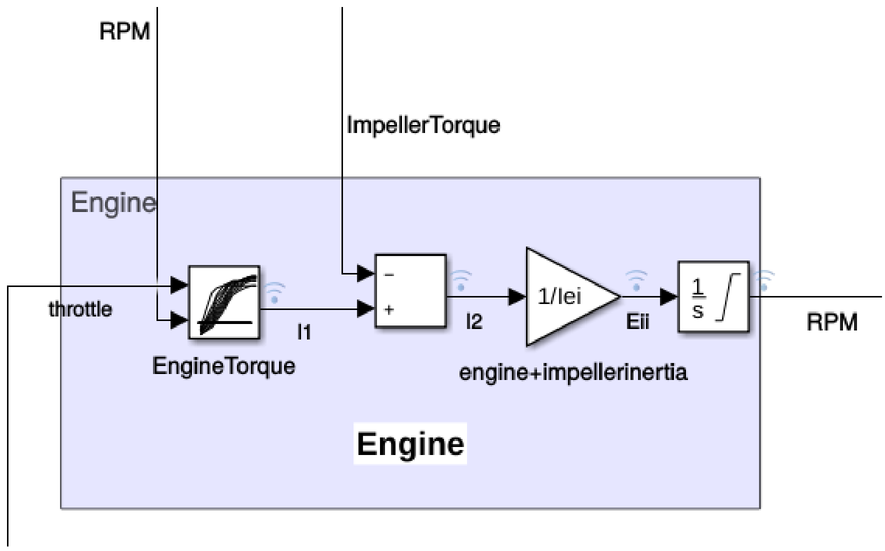


FIGURE 4.5: Parameter Iei and suspected signal Eii in flattened Automatic transmission model

block subsystem.

In Neural Network based maglev model, the parameters used for repair are constants within the 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.

In the Anti-lock braking model, one of the parameters used in repair is piston area and radius with respect to the wheel (denoted as Kf). The other parameter is the numerator coefficient of the transfer function for hydraulic lag. For instance, in case of ϕ_{10}, the suspected signal is $braketorque$ whose source block is a gain block with parameter Kf. The wheel speed Ww is given as $Ww = \int \frac{tiretorque - braketorque}{l} dx$. Since, ϕ_{10} demand the convergence of Ww and Vs, tuning the source block (Kf) of *braketorque* fixes the model.

In the helicopter model, the parameter used for repair is the controller scaling factor K. For instance, the violation of ϕ_{12} is attributed to deviation of $thetadot$ from psi. Since $thetadot = K * a * \int (psi - thetadot)dt$ (refer [21]), tuning the parameter K fixes the model.

It is worth noting that multiple signals in the set s_{ind} could be useful to repair the model.

In Table 4.3, we have presented the break-up of time units spent on each step in the falsification process - namely, Interfacing with the Simulink model (`Intf`), Input Generation (`Inp`), Falsification (`Falf`), procedures FindErrorSignal (`P1`), SliceSimulinkModel

Spec	Computation Time (ms)							
	Intf	Inp	Fals	P1	P2	P3	P4	Total
ϕ_1	1766	14	1047	55	1626	8360	4	12872
ϕ_2	1766	14	39288	38	3219	8493	2	52820
ϕ_3	1766	14	10819	57	1596	8107	3	22362
ϕ_4	7924	17	12822	31	3212	18635	9	42650
ϕ_5	7924	17	13493	39	3284	19084	5	43846
ϕ_6	7924	17	12904	39	1596	18654	3	41137
ϕ_7	7924	17	46361	24	1609	18632	2	74569
ϕ_8	2272	8	3618	36	3092	10605	2	19633
ϕ_9	2272	8	1946	48	3066	10291	3	17634
ϕ_{10}	1749	9	831	15	3098	7680	3	13385
ϕ_{11}	1031	9	679	50	1525	8287	2	11583
ϕ_{12}	620	9	347	17	1433	879	1	3306

Table 4.3: Computation Time

(`P2`), FindStateAtViolation (`P3`) and FindIndependentSignals (`P4`). The maximum computation time was required for ϕ_7, which is 74.569s.

Chapter 5

Conclusion

In this chapter, we position the contribution of this thesis with respect to the related work. We also outline the scope of future work.

5.1 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 [19] which is based on set-based reachability analysis techniques. Some of the popular tools in this domain are SpaceEx [24], C2E2 [20] and Flow* [9]. The other approach is rigorous testing which is often carried out based on falsification of a specification. The prominent tools in this domain are S-TALIROs [3] and BREACH [15]. We have used the latter approach for bug localisation in this thesis.

Bug localization [58, 46] 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 [54].

Recently, various research work have been carried out to develop heuristics for fault localization in Simulink models using statistical debugging techniques iteratively [34, 35]. However, these techniques have been of limited use due to their unpredictability, due to which they need generation of additional test cases [33]. In [6], the authors use model slicing and spectrum-based fault-localization technique [2] 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 thesis, we do not make any assumption on the location of the bug. Though the above-mentioned papers deal with bug localization of Simulink models, unlike our work, 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 violation of complex real-time specifications, which has been lacking in the contemporary papers.

Some of the other recent approaches related to dealing with errors in the models of cyber-physical systems are also worth mentioning here. In [14], the focus is on identifying the subset of inputs that are responsible for a counterexample as a whole. The identification of important inputs may help us in generating more test cases producing violations which may in turn help in fault localization.

In [12, 18], the authors focus on parameter synthesis using reachability analysis to find the set of parameters such that $\mathcal{M} \models \phi$, which is although exhaustive but is computationally expensive. Other computationally complex methods like sensitivity analysis [22, 17] and abstraction based methodology [8] have been studied for parameter synthesis in CPS models. Unlike these papers, we do not treat a model as a black-box, rather we analyse 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 [13], 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.

In [23], the authors explain the falsification of a formula using its prime implicants. The algorithm presented in the paper is more powerful in sense that it can give better unsat cores(prime implicants) than our Algorithm 3 but it is computationally more expensive than our algorithm. Nevertheless, this is similar to what we do in Algorithm 3 and can be incorporated into our framework. In [7] the authors perform property mining from passing traces and use them to analyze the failures in the model. Comparing this with our work, the time required for bug localisation is less in our case. For the Automatic transmission model, the computation time is approx. 65 % less (we have used our worst case computation time for comparison).

5.2 Contribution

In this thesis, 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 mechanism to repair the Simulink models using the information generated from bug localization when the bug is due to inappropriate value for some model parameter. Our approach is based on the rigorous analysis of traces generated by Simulink models, model slicing, and most importantly, finding the linearly independent signals. 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.

5.3 Future Work

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. We also plan to apply our current tool to many more Simulink models with specifications of varied complexity.

Bibliography

[1] Abbas, H., Fainekos, G., Sankaranarayanan, S., Ivancic, F., and Gupta, A. (2013). Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12.

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

[3] Annpureddy, Y., Liu, C., Fainekos, G. E., and Sankaranarayanan, S. (2011). S-taliro: A tool for temporal logic falsification for hybrid systems. In *TACAS*.

[4] Ball, T., Naik, M., and Rajamani, S. K. (2003). From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105.

[5] Bartocci, E., Deshmukh, J., Donze, A., Fainekos, G., Maler, O., Nickovic, D., and Sankaranarayanan, S. (2018a). Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, pages 135–175.

[6] Bartocci, E., Ferrère, T., Manjunath, N., and Ničković, D. (2018b). Localizing faults in simulink/stateflow models with stl. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, pages 197–206. ACM.

[7] Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., and Nickovic, D. (2019). Automatic failure explanation in CPS models. In Ölveczky, P. C. and Salaün, G., editors, *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724 of *Lecture Notes in Computer Science*, pages 69–86. Springer.

[8] Bogomolov, S., Schilling, C., Bartocci, E., Batt, G., Kong, H., and Grosu, R. (2015). Abstraction-based parameter synthesis for multiaffine systems. In *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, pages 19–35.

[9] Chen, X., Ábrahám, E., and Sankaranarayanan, S. (2013). Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, pages 258–263. Springer Berlin Heidelberg.

[10] Cleve, H. and Zeller, A. (2000). Finding failure causes through automated testing. In *AADEBUG*.

[11] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351.

[12] Dang, T., Dreossi, T., and Piazza, C. (2015). Parameter synthesis through temporal logic specifications. In *FM 2015: Formal Methods*, pages 213–230. Springer International Publishing.

[13] Deshmukh, J., Jin, X., Majumdar, R., and Prabhu, V. (2018). Parameter optimization in control software using statistical fault localization techniques. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 220–231.

[14] Diwakaran, R. D., Sankaranarayanan, S., and Trivedi, A. (2017). Analyzing neighborhoods of falsifying traces in cyber-physical systems. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ICCPS '17, pages 109–119.

[15] Donzé, A. (2010). Breach, a toolbox for verification and parameter synthesis of hybrid systems. In Touili, T., Cook, B., and Jackson, P., editors, *Computer Aided Verification*, pages 167–170. Springer Berlin Heidelberg.

[16] Donzé, A., Ferrère, T., and Maler, O. (2013). Efficient robust monitoring for stl. In Sharygina, N. and Veith, H., editors, *Computer Aided Verification*, pages 264–279.

[17] Donzé, A., Krogh, B., and Rajhans, A. (2009). Parameter synthesis for hybrid systems with an application to simulink models. In *Hybrid Systems: Computation and Control*, HSCC '09, pages 165–179. Springer Berlin Heidelberg.

[18] Dreossi, T. and Dang, T. (2014). Parameter synthesis for polynomial biological models. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14, pages 233–242.

[19] Duggirala, P. S., Fan, C., Potok, M., Qi, B., Mitra, S., Viswanathan, M., Bak, S., Bogomolov, S., Johnson, T. T., Nguyen, L. V., Schilling, C., Sogokon, A., Tran, H., and Xiang, W. (2016). Tutorial: Software tools for hybrid systems verification, transformation, and synthesis: C2e2, hyst, and tulip. In *2016 IEEE Conference on Control Applications, CCA 2016, Buenos Aires, Argentina, September 19-22, 2016*, pages 1024–1029.

[20] Duggirala, P. S., Mitra, S., Viswanathan, M., and Potok, M. (2015). C2e2: A verification tool for stateflow models. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–82, Berlin, Heidelberg. Springer Berlin Heidelberg.

[21] E.A.Lee and S.A.Seshia (2017). Introduction to embedded systems - a cyber-physical systems approach,second edition.

[22] Eschenbach, T. G. (1992). Spiderplots versus tornado diagrams for sensitivity analysis. *Interfaces*, 22(6):40–46.

[23] Ferrère, T., Maler, O., and Nickovic, D. (2015). Trace diagnostics using temporal implicants. In Finkbeiner, B., Pu, G., and Zhang, L., editors, *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, volume 9364 of *Lecture Notes in Computer Science*, pages 241–258. Springer.

[24] Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., and Maler, O. (2011). Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer Berlin Heidelberg.

[25] Ghosh, S., Sadigh, D., Nuzzo, P., Raman, V., Donzé, A., Sangiovanni-Vincentelli, A. L., Sastry, S. S., and Seshia, S. A. (2016). Diagnosis and repair for synthesis from signal temporal logic specifications. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, HSCC '16, pages 31–40.

[26] Groce, A., Kroening, D., and Lerda, F. (2004). Understanding counterexamples with explain. In Alur, R. and Peled, D. A., editors, *Computer Aided Verification*, pages 453–456.

[27] Hildebrandt, R. and Zeller, A. (2000). Simplifying failure-inducing input. *SIGSOFT Softw. Eng. Notes*, 25(5):135–145.

[Jin et al.] Jin, X., Deshmukh, J. V., Kapinski, J., Ueda, K., and Butts, K. Powertrain control verification benchmark. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control*, HSCC '14.

[29] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477.

[30] Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299.

[31] Legay, A., Delahaye, B., and Bensalem, S. (2010). Statistical model checking: An overview. In *Runtime Verification*, pages 122–135.

[32] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26.

[33] Liu, B., Lucia, Nejati, S., and Briand, L. C. (2017). Improving fault localization for simulink models using search-based testing and prediction models. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 359–370.

[34] Liu, B., Lucia, Nejati, S., Briand, L. C., and Bruckmann, T. (2016a). Localizing multiple faults in simulink models. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 146–156.

[35] Liu, B., Lucia, Nejati, S., Briand, L. C., and Bruckmann, T. (2016b). Simulink fault localization: an iterative statistical debugging approach. volume 26, pages 431–459.

[36] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. (2005). Sober: Statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 286–295.

[37] Maler, O. and Nickovic, D. (2004). Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg. Springer Berlin Heidelberg.

[38] Maler, O. and Nickovic, D. (2013). Monitoring properties of analog and mixed-signal circuits. volume 15, pages 247–268.

[39] MathWorks. Design narma-l2 neural controller in simulink.

[40] MathWorks. Isolating problematic behavior with model slicer.

[41] MathWorks. Modeling an anti-lock braking system.

[42] MathWorks. Modeling an automatic transmission controller.

[43] MathWorks. Simulink-simulation and model-based design.

[44] Nickovic, D. and Maler, O. (2007). AMT: A property-based monitoring tool for analog systems. In *Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'07, pages 304–319.

[45] Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209.

[46] Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D., and Keller, B. (2017). Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 609–620.

[47] Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57.

[48] Reicherdt, R. and Glesner, S. (2012). Slicing matlab simulink models. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 551–561.

[49] Renieris, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE'03, pages 30–39.

[50] Sankaranarayanan, S. and Fainekos, G. (2012). Falsification of temporal properties of hybrid systems using the cross-entropy method. In *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '12, pages 125–134. ACM.

[51] Strang, G. (2009). *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Wellesley, MA.

[52] Tip, F. (1995). A survey of program slicing techniques,. In *Journal of Programming Languages, vol. 3*, pages 121–189.

[53] Vandenberghe, L. (2019). Qr decomposition.

[54] Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. volume 23, pages 459 – 494.

[55] Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449. IEEE Press.

[56] Wikipedia. Linear independence.

[57] Wikipedia. Matrix decomposition.

[58] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. volume 42, pages 707–740.

[59] Younes, H. L. S. and Simmons, R. G. (2006). Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409.

[60] Zutshi, A., Sankaranarayanan, S., Deshmukh, J. V., Kapinski, J., and Jin, X. (2015). Falsification of safety properties for closed loop control systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, pages 299–300. ACM.