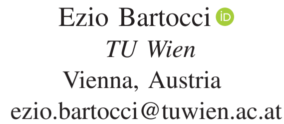
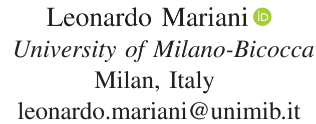
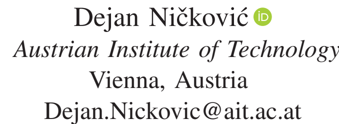
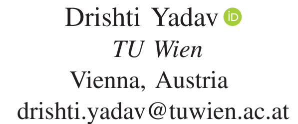
Search-based Testing for Accurate Fault Localization in CPS

Ezio Bartocci 
TU Wien
Vienna, Austria
ezio.bartocci@tuwien.ac.at

Leonardo Mariani 
University of Milano-Bicocca
Milan, Italy
leonardo.mariani@unimib.it

Dejan Ničković 
Austrian Institute of Technology
Vienna, Austria
Dejan.Nickovic@ait.ac.at

Drishti Yadav 
TU Wien
Vienna, Austria
drishti.yadav@tuwien.ac.at

2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-5132-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISSRE55969.2022.00024

***Abstract*—Fault localization plays an important role in the design, verification and debugging of cyber-physical systems (CPS). Finding the exact location of a fault that triggered a failure in a CPS model is however a challenging task, due to the complex structure and data-flow nature of CPS models. In this paper, we propose a method that uses formal specifications and search-based testing to accurately localize faults. Given a CPS Simulink model, a formalized requirement used as a test oracle, and a test case that fails the formalized property, we develop a procedure that uses search-based testing to generate another test case that succeeds on the same formalized property. We then compare our two similar test cases with opposite verdicts to find the accurate location of the fault. We implement our approach and evaluate it on three case studies from automotive and avionic domains. We empirically compare our approach to a state-of-the-art fault localization technique and demonstrate that our procedure (1) is able to considerably narrow down the number of suspicious model variables and blocks compared to the previous work, and (2) remains robust to an increasing number of active faults in the underlying models.**

***Index Terms*—Cyber-Physical Systems, Model-based Development, Debugging, Fault Localization, Equivalence Testing, Signal Temporal Logic (STL), Simulink models**

I. Introduction

The development of safety-critical Cyber–Physical Systems (CPS) is a challenging activity. Model-based platforms are increasingly used by the embedded software industry experts to cope with the inherent complexity of CPS development and facilitate cost-effective design. The MathWorks® MATLAB/Simulink environment has emerged as a *de-facto* standard for the model-based design (MBD) of CPS [1].

While MBD facilitates many design tasks, debugging faulty models remains a cumbersome and labour-intensive activity. Precise identification of the fault location is tedious and often demands considerable expertise from an engineer to reveal the root cause of the resulting failure. Detecting and diagnosing faults quickly and accurately is necessary to ensure that the system is viable and fully operational at all times. In particular, it is important to detect problems in the early stages of CPS design, as undetected failures in safety-critical CPS are not only costly but also have potentially catastrophic consequences [2]–[4]. A precise localization of the detected fault can significantly accelerate the model correction and facilitate the design of safe CPS.

Simulation-based testing and its variants, such as falsification testing, are practical approaches used to efficiently identify bugs in CPS design. Falsification testing [5]–[8] uses a specification expressed in a formal language, such as Signal Temporal Logic (STL) [9] and its quantitative semantics, to guide the search for tests that violate the specification.

While simulation-based testing allows to detect incorrect behavior in the model, it typically provides only the witness behavior as the explanation of the specification violation. More recently, fault localization and explanation methods have been proposed to facilitate debugging of MATLAB/Simulink models [2], [4], [10]–[12]. These gray-box procedures use various heuristics to localize the fault and thus reduce the debugging scope. The common characteristics of the existing localization methods is that they are all detached from the testing activities. It follows that the accuracy of the localization is highly dependent on the quality of the test suite.

In the context of procedural and object-oriented programming, the problem of localizing a fault has been investigated as the problem of comparing two similar test executions with opposite verdicts [13], [14], one passing and the other failing. However, these approaches use test generation strategies and heuristics that cannot be trivially transferred to data-flow oriented computational models, such as CPS Simulink models.

Contributions. We propose an accurate fault localization method that is tightly coupled with (falsification) testing. We consider CPS Simulink models with either explicit or implicit specifications. We adopt STL as formal language for specifying CPS requirements. In absence of explicit specifications, we assume a correct *reference model* and define its *equivalence* to the model-under-test as our implicit specification. In both cases, we use the formal specification as a test oracle. Given a test case that fails according to the formalized property, we develop a procedure that generates an alternative passing test that can be used to extract accurate information about the fault location once compared to the failing test. In essence, we use a global optimizer to perform search-based testing and generate a new passing test case that is close to the original failing test, according to some distance measure. We then exploit our two similar test cases with opposite verdicts to generate accurate information about the fault location, decreasing debugging scope and effort. Our technique characterizes and orders suspicious variables by their time and degree of violation,

2332-6549/22/$31.00 ©2022 IEEE
DOI 10.1109/ISSRE55969.2022.00024

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

presenting them together with their associated model blocks. The list of suspicious blocks helps the engineer to localize in the model multiple (and possibly large number of) faults of possibly different types.

In summary, as our main contributions, we propose (1) a method for the automated generation of a passing test that is close to a failing test for (data-flow) Simulink models using a new search-based approach, and (2) a method for localizing faults in Simulink models based on a failing test and its associated close passing test.

We demonstrate our approach on three case studies from different application domains. We instantiate our approach both with explicit STL specifications and with implicit equivalence checking, and evaluate it on 240 faulty variants of the case study models with multiple faults. We compare its performance to `CPSDebug` [10], [12] a state-of-the-art fault localization technique for CPS. The experimental results show that compared to the baseline, our approach (1) efficiently narrows down the number of suspicious variables and consequently, the number of blocks to be inspected to localize the faults, and (2) correctly identifies the fault locations for multi-fault models. In particular, with multiple faults in the underlying models, the baseline exhibits degraded performance in terms of localization accuracy.

Paper Organization. In Section II, we briefly discuss the preliminary concepts required in the remainder of the paper. We present our fault localization approach in Sections III and IV. In Section V, we provide the empirical evaluation and summarize the results. Section VI presents a refined review of related works. We conclude in Section VII.

II. Background

In this section, we present some preliminaries, fix our notations and briefly review the terminology that we will use in this paper. We provide the necessary background information on STL and key features of Simulink models.

A. Signal Temporal Logic

Functional properties and safety requirements in CPS are usually captured by complex timing relations of their hybrid (discrete/continuous) behaviors. Signal Temporal Logic (STL) [9] is a well-established specification language to express CPS temporal requirements over dense-time and real-valued behaviors. An STL specification can be evaluated against an execution trace of a CPS according to a qualitative or quantitative semantics. The qualitative semantics returns a Boolean value witnessing whether the system under test (SUT) satisfies (true) or violates (false) the specification, while the quantitative semantics [15] provides a real value representing *how robustly* a property (formula) is satisfied or violated.

The syntax of an STL formula Φ defined over a set X of variables is given by the grammar:

$$\Phi := true \mid f(x_1, \ldots, x_m) > 0 \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \mathcal{U}_I \Phi_2$$

where $\{x_1, \ldots, x_m\} \subseteq X$, $f : X^m \to \mathbb{R}$ is a function mapping m variables to a real, and $I \subseteq \mathbb{R}_{\geq 0}$ is an arbitrary interval. The operators $\neg$ and $\vee$ indicate logical negation and logical disjunction. $\mathcal{U}_I$ is the until operator, meaning that at some time j ($j \in I$), Φ_2 becomes true and Φ_1 must remain true until Φ_2 becomes true. From the basic set of operators, we can derive two other temporal operators: *eventually* ($\Diamond_I$) and *always* ($\Box_I$). We use the temporal operators $\mathcal{U}$, $\Diamond$, and $\Box$ to denote $\mathcal{U}_I$, $\Diamond_I$, and $\Box_I$ with $I = [0, \infty)$.

The semantics of STL is defined using the *satisfiability relation* $(w, j) \models \Phi$, implying that the signal w satisfies Φ at time j. The quantitative robust semantics of STL provides a *robustness degree* as the distance between the observed behavior (a signal w) and the set of behaviors defined by the specification Φ. Given an STL formula Φ, the *robustness* is the measure of satisfaction of w with respect to Φ. It is defined as a real-valued quantity denoted by $\rho : R(w, \Phi)$ s.t. (1) $\rho > 0 \Rightarrow w \models \Phi$, and (2) $\rho < 0 \Rightarrow w \models \neg\Phi$. We refer the reader to work by Donze & Maler [15] for deeper insights into the quantitative robust semantics of STL properties.

B. Simulink models

During the model-based design of CPS using the MathWorks™ Simulink environment, an engineer designs a Simulink model $\mathcal{M}$ as a block diagram representation of the actual system. Fig. 1 illustrates an example of Simulink model with its essential elements.

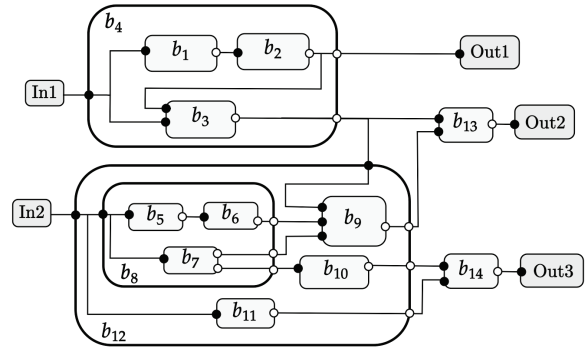


Fig. 1. Example of a Simulink model. Black nodes indicate input ports while white nodes indicate output ports. Blocks b_4, b_8, b_{12} are hierarchical; remaining blocks are atomic.

A Simulink model $\mathcal{M}$ consists of the following elements:

- A set of *blocks* $\mathcal{B}$: These are the basic building units of a Simulink model. The input-output (I/O) behavior of a block represents the cause-effect relationship, characterizing its functionality.
- A set of *ports* $\mathcal{P}$: A block receives data via its *input ports* and transmits the data via its *output ports*. Note that a block may also consist of *trigger*, *action*, *enable*, *reset* ports, which we exclude from our analysis.
- A set of *lines* $\mathcal{L}$: These are connections which indicate data flow or propagation of signals from one block to the other. The block which transmits data is the *source* block while the one which receives the data is the *target* block. Each

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

line $l \in \mathcal{L}$ is uniquely identified by the output ports of the source block and the input ports of the target block.

Each of the above three elements has a unique numeric identifier in the model, known as its *handle*. The handle is, in fact, a pointer to the element that can be accessed to modify and update the model. Typically, a model $\mathcal{M}$ acquires inputs from a set of input blocks or sources (such as Fig. 1's *In1* and *In2* blocks) and yields the output via a set of output blocks or sinks (such as Fig. 1's *Out1*, *Out2* and *Out3* blocks). $\mathcal{M}$ also contains a set of *input-output-internal state* variables V and a set of signals S, defined as the mapping $S\colon \mathcal{L} \rightarrow V$.

Further, there are two types of blocks: *atomic* and *hierarchical*. An atomic block does not encapsulate any other block within it, while a hierarchical block includes atomic blocks and other hierarchical blocks. A block is *parent* of one or more blocks when it includes them as its own subsystems. In Fig. 1, the *top-level* model $\mathcal{M}$ has hierarchy level 1. Further, $b_1 - b_3, b_8, b_9 - b_{11}$ are $\mathcal{M}$'s child at hierarchy level (depth) 2, while $b_5 - b_7$ are $\mathcal{M}$'s child at hierarchy level 3.

In addition, Simulink offers several *libraries* of built-in blocks facilitating the design and the fast development of CPS. A user can also create and add custom libraries and blocks with user-specific functionalities and user-configurable parameters.

After designing the CPS in Simulink, users *compile* and *simulate* the model $\mathcal{M}$ against a given test case t, i.e., an input signal. We define a function $Simulate$ that takes $\mathcal{M}$ and t as inputs, simulates $\mathcal{M}$ and returns traces of all input-internal-output (IIO) signals as the final model simulation output. We assume that the model applies fixed-length sampling that represents a large class of useful Simulink models. Hence, a trace is a sequence of (timestamp, value) pairs, where the distance between any two consecutive samples corresponds to a fixed period. We denote all the traces generated for $\mathcal{M}$ for some test case t by $out(\mathcal{M}, t)$. Mathematically, $out(\mathcal{M}, t) = Simulate(\mathcal{M}, t)$. In general, a *trace* refers to the sequence of states of $\mathcal{M}$ evolving with discrete time steps (from time $q = 0$ to $q = q_T$, where the finite time horizon $q_T > 0$). The values in a simulation trace are stored in the same order in which the signal evolves i.e., from $q = 0$ to $q = q_T$.

III. STL-guided Fault Localization

In this section, we describe how our specification-guided fault localization approach, namely `STL-FL`, works. Fig. 2 visually illustrates the two main phases that compose the fault localization workflow:

1) *Testing*, which first evaluates the SUT against an initial test suite to find the failing test cases and then uses a global optimizer to generate a new passing test case for each failed test case, guided by the STL specification ϕ. The new passing test case is searched to be as similar as possible to its corresponding failing test;
2) *Localizing*, which uses the simulation outputs for a pair of failed and passing test cases to identify the precise location of the faulty component(s).

A. Testing

The testing phase aims to identify a set of representative pairs composed of a failing and a passing test case that can be used for fault localization. Each pair must include two similar executions whose difference is likely exclusively caused by the activation of the fault to be localized. In such a way, the point-to-point comparison of these executions can reveal insights useful to accurately localize the fault.

The testing phase identifies these pairs by first detecting the failing tests (i.e., the tests that fail to satisfy the available STL specification) among the available ones, and then proactively generating new (passing) test cases that are incrementally closer to the failing one until determining the best candidate for fault localization.

We assume that an initial test suite $\mathcal{TS}$ is given, and that it contains at least one test case that results in the violation of a formalized requirement. The test suite $\mathcal{TS}$ can be manually provided by the tester or created using automated test generation tools. The automated test generation methods can rely on coverage-based methods [16] (taking into account input, output and/or structural coverage). They can also target efficient generation of failing test cases, such as falsification testing methods [17].

Algorithm 1 outlines the procedure for selecting the test pairs. Given an initial Test Suite $\mathcal{TS}$, the available test cases are then evaluated against a faulty model. For each test case $t \in \mathcal{TS}$, we simulate the faulty model to obtain the results of the model simulation (Line 4). We then evaluate the output traces against the STL specification ϕ by a $Monitor$ to assign a *pass* or *fail* verdict to the test case (Lines 5-7). The $Monitor$ takes care of 'Trace evaluation' (see Fig. 2) and produces a robustness measure of the trace with respect to ϕ (Line 5). The aim is to identify all the test cases that lead to the violation of the property ϕ: the algorithm groups all the failing test cases that lead to observable failures in the model (Line 7).

For each failing test case, the algorithm finds a close passing test case that could be used to support localization. In particular, for each failing test $t_f \in \mathcal{TS}_{New}$, the algorithm searches for a passing test case t_p which is close (or more similar) to the failing test case t_f (Line 11). Our SearchPT() subroutine is shown in Algorithm 2. The search is cast as an optimization problem of finding a passing test t_p which is close to the failing test t_f s.t. the distance between t_p and t_f is minimum. This problem is formulated as:

Close passing test case search problem

Input: a faulty Simulink model $\mathcal{M}_F$ with a failing test case t_f and a formula ϕ s.t. $out(\mathcal{M}_F, t_f) \not\models \phi$.
Problem: Find t_p s.t. $out(\mathcal{M}_F, t_p) \models \phi$, and $D(t_f, t_p)$ is minimum.

In this work, we employ the recently developed BCA [18] optimizer for the search task. The key to the criterion for selection of the most similar passing test is the definition of the distance $D(t_f, t_p)$ between the failing and passing test

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

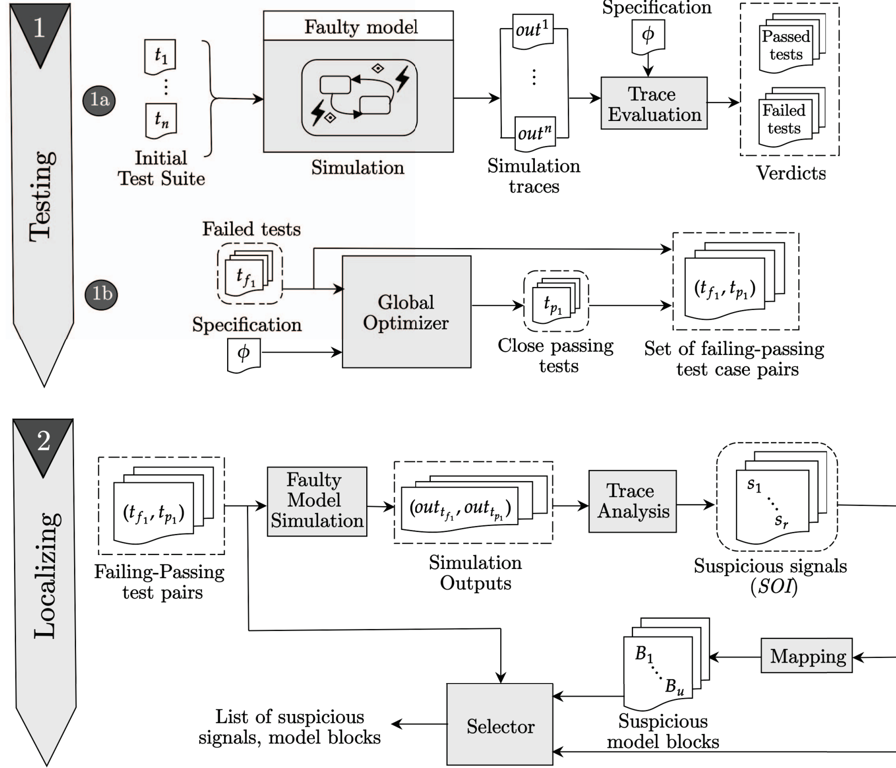


Fig. 2. An overview of STL-guided fault localization procedure (STL-FL).

Algorithm 1: Test Suite Selection for localizing faults.

Input : $\mathcal{TS}$: An initial test suite.
$\mathcal{M}_F$: A faulty model.
ϕ : An STL specification.
Output: $\mathcal{TS}_{FL}$: Set of failing-passing test case pairs.

1 $\mathcal{TS}_{New} = [\ ]$
2 $\mathcal{TS}_{FL} = [\ ]$
3 **for** *each item* $t \in \mathcal{TS}$ **do**
4 $\quad out(\mathcal{M}_F, t) = Simulate(\mathcal{M}_F, t)$
5 $\quad R(out(\mathcal{M}_F, t), \phi) = Monitor(out(\mathcal{M}_F, t), \phi)$
6 $\quad$ **if** $R(out(\mathcal{M}_F, t), \phi) < 0$ **then**
7 $\quad\quad \mathcal{TS}_{New} = \mathcal{TS}_{New} \cup \{t\}$
8 $\quad$ **end if**
9 **end for**
10 **for** *each item* $t_f \in \mathcal{TS}_{New}$ **do**
11 $\quad t_p \leftarrow \text{SEARCHPT}(t_f, \mathcal{M}_F, \phi)$
12 $\quad \mathcal{TS}_{FL} = \mathcal{TS}_{FL} \cup \{(t_f, t_p)\}$
13 **end for**
14 **return** $\mathcal{TS}_{FL}$

---

cases for the faulty model $\mathcal{M}_F$. We compute the distance $D(t_f, t_p)$ as the Euclidean distance (aka *Euclidean norm* or L^2 *norm*) between the signals that correspond to the test cases t_f and t_p. Euclidean distance is one of the most prevalent distance metrics on real vector space that offers the advantage of measuring similarities by computing the regular distance between data points. Since CPS models frequently deal with continuous real-valued variables, the Euclidean distance is a good choice for our analysis. Formally, the Euclidean distance D between two signals $\mathbf{y}$ and $\mathbf{z}$ is expressed as:

$$D(\mathbf{y}, \mathbf{z}) = ||\mathbf{y} - \mathbf{z}||_2 = \sqrt{\sum_{i=1}^{g} (y_i - z_i)^2}$$

where $\mathbf{y}$ and $\mathbf{z}$ are finite-length signals with g samples s.t. $\mathbf{y} = (y_1, \cdots, y_g)$ and $\mathbf{z} = (z_1, \cdots, z_g)$. In general, the number of samples of a signal is a property of the simulation that depends on the *step-size* aka sample time of the model. Since the simulations for test cases t_f and t_p are on the same faulty model, the timestamps (and the signal length) are guaranteed to match among the two executions. The final output of the testing phase is $\mathcal{TS}_{FL}$, a set of pairs of failing test case/close passing test case for the faulty model $\mathcal{M}_F$ (Line 14).

We now describe the SEARCHPT() subroutine outlined in Algorithm 2 that presents BCA adapted to solve the formulated 'Close passing test case search' problem. Given the availability of a plethora of optimization algorithms, we chose BCA over other optimizers because of its (1) intuitive interpretation of parameters and no tuning effort, (2) straightforward implementation, and (3) high computational efficiency. We also tested the performance of BCA against some other state-of-the-art

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

optimizers (including Particle Swarm Optimizer, Harmony Search, Water Cycle Algorithm and Imperialist Competitive Algorithm) for our optimization problem and observed that BCA outperforms others in terms of convergence and speed. Below is the outline of BCA which is a population-based optimizer.

Definition 1. Given an objective function $f(x)$ with n variables, BCA performs the following steps to search for the global best solution of $f(x)$.

1) It creates a randomly (uniformly distributed) initial population of candidates over the search space.
2) It computes fitness of each candidate and finds the candidate fittest on $f(x)$.
3) It updates position of each candidate based on the updating criteria and optimizer parameters. It recomputes fitness values and updates the global best if better solution is found. It repeats until termination criteria is met.

In Algorithm 2, we start by creating an initial population of candidates (Line 2) and then computing the fitness of each candidate i.e. the test case (Line 3). The individuals in the initial population are obtained by generating test inputs that uniformly sample the (numerical) input domain of the SUT. Note that $genPop$ (Line 2) mimics the first step of BCA outlined in *Definition 1*, and ensures that the initial population is a non-empty set of test candidates of a user-defined population size. We do not consider for the moment other input domains since they seldom occur in CPS model, but they could be also addressed as long as a distance metric is provided for their domain. For instance, a Boolean distance function could be used with enumerated signals.

Here, for each candidate in the population, the *fitness* indicates the quantitative robustness measure of the execution trace of the faulty model $\mathcal{M}_F$ with respect to the STL specification ϕ. Line 4 finds the best test case among all test cases in the population which (1) is closest to the failing test case t_f and (2) satisfies ϕ. Line 5 computes the initial Euclidean distance between t_f and the best solution t_p, which is later used by the algorithm as the selection criterion. An important observation is that the algorithm updates the test cases (Line 8) and accepts the new solution as the best one if the selection criteria is met i.e., a closer passing test is found (Lines 12-15). On the guard for the while loop (Line 6), the subroutine terminates if one of the following conditions holds: (1) the maximum number of iterations is reached, (2) there is no improvement after a certain number of iterations, and (3) we reach the timeout.

Algorithm 2: The SEARCHPT() subroutine.

```
Input  : M_F : A faulty model.
         φ : An STL specification.
         t_f : A failing test case.
Output: t_p : Close passing test case.
1  Initialize optimizer parameters
2  InitPop ← genPop() ;  // Initial population
3  FitPop ← Cost(InitPop, φ) ;         // fitness
4  t_p ← Best(FitPop) ;          // best solution
5  D_max = D(t_f, t_p) ;       // Initial Distance
6  while TimeOut() do
7      for each candidate u ∈ InitPop do
8          u_new ← Update(u)
9      end for
10     FitPop ← Cost(InitPop, φ)
11     D ← D(t_f, Best(FitPop))
12     if D < D_max then
13         D_max ← D ;  // update the distance
14         Update t_p ;      // new best solution
15     end if
16 end while
17 return t_p
```

B. Localizing faults

Algorithm 3 outlines our approach for localizing faults by analyzing the anomalous events in the SUT guided by a STL specification. The inputs include a faulty model and a Test Suite $\mathcal{TS}_{FL}$ with a set of pairs of failing and passing test cases. The localization algorithm follows the intuition that the signals that deviate first in time between the passing and the failing test executions are likely to be an artifact of a fault present in the block that originated their values. In practice, it identifies the model variables that misbehave first in time with significantly high differences between the failing test case and the close passing test case, maps these variables to the corresponding blocks and generates a list of suspicious blocks to assist model debugging and repair.

Algorithm 3: Fault Localization by analyzing anomalous events with STL.

```
Input  : M_F - A faulty model.
         TS_FL - Set of failing-passing test case pairs.
Output: SOI_best - Signals of interest.
         blockList_best - List of suspicious blocks.
1  for each pair (t_f, t_p) ∈ TS_FL do
2      out(M_F, t_f) = Simulate(M_F, t_f)
3      out(M_F, t_p) = Simulate(M_F, t_p)
4      d = |out(M_F, t_f) − out(M_F, t_p)|
5      d_m, q_viol ← EVAL(d)
6      d_new = Normalize(d_m)
7      SOI ← GetSignalInfo(d_new)
8      blockList ← Map(SOI)
9  end for
10 SOI_best, blockList_best = Π(TS_FL, SOI, blockList)
11 return SOI_best, blockList_best
```

Engineers start with a test suite $\mathcal{TS}_{FL}$, a set of failing-passing test case pairs. Lines 2-3 perform the job of 'Faulty Model Simulation' block shown in Fig. 2. In particular, Line 2 simulates the faulty model with the failing test case t_f, so as to obtain the corresponding simulation traces of each model variable. Line 3 simulates the faulty model with the passing test case t_p to acquire the simulation traces.

Lines 4-7 perform the 'Trace Analysis' (as shown in Fig. 2).

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

In particular, Line 4 computes the deviations between the simulation traces for the failing and passing tests. We recall that we assume that every trace is periodically sampled, thus enabling pointwise comparison between two traces. Usually, $out(\mathcal{M}_F, t)$ is an array of size $Y \times Z$ where Y denotes the total number of samples of the recorded signals and Z is the total number of signals recorded. The deviation d (an array of size $Y \times Z$) represents the absolute values of the difference among Z recorded signals for t_f and t_p. Note that 'minus' sign in Line 4 indicates element-by-element subtraction of the values of each recorded signal at every timestamp for test case t_p from that of test case t_f.

In Line 5, the function EVAL takes the computed deviation d as input and finds the timestamp q_{viol} at which first misbehavior is observed. It also internally sorts all the deviation values corresponding to q_{viol} in descending order and returns d_m which is an array of size $1 \times Z$. More precisely, d_m indicates an ordered sequence of deviation values of all recorded signals observed at the first timestamp of violation q_{viol}.

The function $Normalize$ returns the vector-wise z-score of d_m with center 0 and standard deviation 1 (Line 6). Consequently, d_{new} is an array of size $1 \times Z$ with normalized deviation values (observed first in time) for each recorded signal. We used data normalization to adjust the deviation values to a notionally common scale and ensure that all the measured deviations lie numerically in the same interval.

In Line 7, the function $GetSignalInfo$ finds the logging information of all those signals which misbehave first in time with *normalized deviations exceeding 0*. This constraint on the normalized deviation values allows to focus only on a small set of signals with significantly high deviations. By that, we localize all the internal signals in the model responsible for the faulty behavior. In Line 8, the function Map (equivalent to 'Mapping' in Fig. 2) maps the *Signals of interest* (SOI) to their corresponding model blocks, so as to mark the suspicious blocks. SOI are the model variables (internal signals) that deviate first in time and with significantly high deviations between t_f and t_p for the faulty model. In essence, SOI indicate those model variables that can best explain the fault.

Finally, the failing-passing test case pair which yields the *minimum* number of SOI is chosen as the best pair. This task is performed by the function Π (Line 10) ('Selector' in Fig. 2). Note that **SOI** and $\mathbf{blockList}$ respectively represent a set of SOI and $blockList$ associated with each pair $(t_f, t_p) \in \mathcal{TS}_{FL}$. Corresponding to the best pair, the variables SOI_{best} and $blockList_{best}$ are shown as the final output to assist identifying the "culprit" signals and components that lead to failure, localizing the faulty units (Line 11). The violation time (q_{viol}) can also be output to assist the understanding of the failure context and the timing modality.

Example 1. We now demonstrate STL−FL with an example. We consider the Simulink model of an automatic transmission controller system described in detail in Section V. We assume a faulty variant with a Bias/Offset fault (with fault value set to 10 units) injected in the signal propagating from the *EngineTorque* block to *Sum* block within the *Engine* subsystem of the model.

TABLE I
DETAILS OF FAILING-PASSING TEST CASE PAIR FOR EXAMPLE 1.

Test case	Throttle parameters			Robustness R	$D(t_f, t_p)$
	ST	IV	FV		
t_f	1	10	68	-1.8022	80.8332
t_p	1	10	65	0.8242	

With constant brake signal (set to zero), we find a failing test case t_f for the model w.r.t. the corresponding STL property ϕ mentioned in Table III, where the test case represents the throttle input signal. We assume that the throttle is a step signal defined using three parameters: *Step Time* (ST), *Initial Value* (IV), and *Final Value* (FV). Using our proposed 'Close passing test case search problem' delineated in Algorithm 2, we found a passing test case t_p (see Table I).

Using Algorithm 3, we identified the SOI to localize the faulty component. We observe that only one internal signal misbehaves first in time with the normalized deviations greater than zero, the details of which are provided in Table II. The identified SOI originates within the *Engine* subsystem wherein we injected the fault, therefore indicating correct localization of the faulty component.

IV. FAULT LOCALIZATION WITH EQUIVALENCE CHECKING

Many CPS do not have explicit specifications. During the various development phases, the design is then typically compared to a *reference model* that is assumed or proved to be correct. Equivalence testing (aka *back-to-back testing*, *differential testing* or *differential fuzzing* in software engineering) is often used to test for equivalence between the reference model $\mathcal{M}$ and its updated or refined version $\mathcal{M}'$. The equivalence is determined by comparing the output signals generated by simulating both against a unique (equivalence) test case. More precisely, the logged output signals are used as the equivalence criteria between $\mathcal{M}$ and $\mathcal{M}'$. It corresponds to an implicit specification formally relating the two models. The non-equivalence of two models can be expressed as:

$$\underbrace{\bigwedge_{i=1}^{m} In_1.i = In_2.i}_{\text{all inputs equal}} \wedge \underbrace{\bigvee_{o=1}^{e} out_1.o \neq out_2.o}_{\text{an output not equal}} \quad (1)$$

Here, In represents the set of model inputs of length m and out indicates the set of traces of all IIO signals of the models with e outputs. (Referring to Fig. 1, $m = 2$ and $e = 3$). Any assignment that satisfies Formula (1) indicates two distinct executions of $\mathcal{M}$ and $\mathcal{M}'$ that produce a different output signal for the same input sequence.

Our approach also supports equivalence checking. Similar to STL−FL, the workflow of our equivalence-driven approach, namely E−FL, starts with testing, explained in the previous section. In this case, a failing test corresponds to a witness of non-equivalence between the two models. For each failing test, we aim to search the close test case that yields the most similar

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

TABLE II
SOI FOR FAULT LOCALIZATION AND THE CORRESPONDING BLOCKS.

Faulty variant	Signal Index	Signal Name	Parent Block	$q_{viol}(s)$
Offset fault in *Engine*	s_2	EngineTorque:1 $\rightarrow$ Offset1:1	Engine	1.00

Note: q_{viol} denotes the first time that the signal exhibits anomalous behavior w.r.t. the test case pair (t_f, t_p).

output as the failing test case. We use a distance metric $dist$ to evaluate the similarity between two outputs. We treat the search task as an optimization problem formulated as: "Given a faulty model $\mathcal{M}_F$ with a failing test case t_f, find t_c (close to t_f) s.t. $dist(c_1, c_2) < \theta$ where $c_1 = out.o(\mathcal{M}_F, t_f)$ and $c_2 = out.o(\mathcal{M}_F, t_c)$". Here, $dist$ indicates the Chebyshev distance (aka *supremum norm*, or L_∞ *norm* or *uniform norm*). Chebyshev distance is a metric on finite-dimensional vector spaces which measures the degree of similarity based on the most significant dimension. We use Chebyshev distance metric induced by the maximum (or supremum) norm that closely resembles the semantics of STL. Formally, the Chebyshev distance between two finite-length signals $\mathbf{y}$ and $\mathbf{z}$ (with g samples) is expressed as:

$$dist(\mathbf{y}, \mathbf{z}) = ||\mathbf{y} - \mathbf{z}||_\infty = \max_g(|y_g - z_g|)$$

In order to speed up the convergence of the search task (using BCA optimizer), we impose a constraint on the distance between the two outputs as $dist(c_1, c_2) < \theta$ where θ is taken as 0.05. The localization procedure is similar to that used in STL-FL (see Lines 1-9 in Algorithm 3), except that the deviations are analyzed for the simulation outputs of the failing test case t_f and the newly discovered test case t_c.

V. Empirical Evaluation

In this section, we describe our research questions, experimental setup, evaluation metrics, and experimental results. We seek to answer the following research questions:

RQ1. [Fault Localization Ability] *How well does our approach narrow down the number of signals and blocks to be inspected to localize the faults?* We evaluate the reduction in the number of suspicious model variables and number of blocks to be inspected to localize the faults. Specifically, we investigate the fault localization ability of our approach and compare our results with those obtained by the baseline technique, CPSDebug [10], which is a state-of-the-art automated fault localization approach for CPS models.

RQ2. [Robustness evaluation] *Is our approach adequately robust when the number of faults in the SUT is large, potentially of different types?* In order to analyze the robustness of our approach with increasing number of faults in the model, we evaluate its fault localization ability against *multi-fault models* and compare the variations with the baseline.

RQ3. [Computational Efficiency] *Is our approach computationally efficient compared to the baseline technique?* To analyze the computational overhead, we report the computation time of our approach and compare the results with that obtained by the baseline technique.

To empirically evaluate our approach and answer the research questions, we conducted systematic experiments on Simulink models of systems across safety-critical domains. For our fault localization approach described in Section III, we developed a prototype implementation. The procedures for test generation, model simulation and model-based fault localization are implemented in MATLAB. We used the RTAMT tool [19] for offline evaluation of STL properties, and implemented our fault localization procedure (presented in Section III) on top of it.

A. CPSDebug

We now provide a brief description of our baseline technique: CPSDebug [10], a state-of-the-art diagnosis solution for Simulink models that localizes bugs in CPS designs and explains the root cause of failures. The workflow of CPSDebug consists of three essential phases: (1) Testing, (2) Specification Mining, and (3) Explaining.

In the testing phase, CPSDebug simulates the CPS Simulink model under analysis against an initial test suite and partitions the test cases into passing and failing ones based on their evaluation against formal STL requirements.

After that, in the specification mining phase, CPSDebug exploits the passing test traces for property mining. The goal is to infer a set of properties that capture the expected behavior of the model. CPSDebug uses (1) Daikon [20], a template-based property inference tool used to infer properties that are likely to hold for the input variables, and (2) Timed k-Tail (TkT) [21], an automaton learning engine that can generate timed automata from timed traces.

Finally, in the explaining phase, CPSDebug exploits the mined properties to analyze failed traces and generate failure explanations. The explaining phase consists of two steps: (1) *Monitoring*: CPSDebug analyzes fail traces and returns the signals violating the properties and the violation time intervals; (2) *Clustering and Mapping*: CPSDebug clusters the fail-annotated signals w.r.t. their violation times and maps them to their corresponding model blocks. CPSDebug uses the violated signals with their corresponding origin blocks to produce, as the final output, a sequence of system snapshots for every cluster of property violations: this way it captures the fault origin and failure propagation in space and time.

B. Experimental Setup

We ran the experiments on a MacBook Pro with Apple M1 chip, 16 GB RAM, macOS Monterey with MATLAB™ R2018b. This section describes the subjects that we use for experiments and evaluation. We also describe fault seeding, test suites and test oracles used in our experiments.

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

1) Case studies: In our experiments, we use three Simulink models from the automotive and avionic domains. Table III provides the size of each model, the STL specification used in the experiments, and the simulation time q_T. The STL specifications in Table III are taken from [5], [10], [22].

Automatic Transmission Controller System is a benchmark from automotive domain [22]. The model has two inputs, the throttle u_t and the brake u_b, that govern the two system outputs: the vehicle speed v (mph) and the engine speed ω (RPM). The inputs lie in the range [0, 100] for all time instants. One of the safety requirements of this system is that the vehicle speed v and engine speed ω must not exceed their respective thresholds $\bar{v}$ and $\bar{\omega}$. We chose these parameters as: $\bar{v} = 120$ mph and $\bar{\omega} = 4500$ RPM.

Aircraft Elevator Control System is a model from the avionics domain which has a redundant actuator control module [23]. The model has two output variables (the positions of left and right actuators) guided by the input variable (Pilot Command). A relevant property is that the desired position of the aircraft must be attained within a preset time. Formally, whenever Pilot Command *cmd* exceeds a threshold *m*, the measured actuator position *pos* must become steady (become at most *n* units away from the *cmd*) within $T + a$ time units. This is expressed by the STL in Table III where $m = 0.09$, $T = 2$, $a = 1$ and $n = 0.02$.

Abstract Fuel Control System is a well-known Powertrain Control Verification Benchmark [24] from the automotive domain modeling the air-fuel controller for an engine. The user inputs throttle command and engine speed to govern both the fuel rate and the air-to-fuel (AF) ratio. The control objective is to maintain the AF ratio at the set-point Ref (i.e., the *ideal* stoichiometric ratio) via closed-loop adjustments of relevant model variables. The STL specification for this model captures constraints on the AF ratio. The values of the parameters are $Ref = 14.7$, $tol = 0.01$, $T_{start} = 10$, and $T_{stop} = 40$.

For our experiments, we set the fundamental sample time of the models {ATCS, AECS, AFCS} to {0.04, 0.01, 0.005} respectively. All the three models are publicly available in the Simulink/Stateflow online documentation of MathWorks [25]–[27]. It is worth mentioning that these models are representative of industrial Simulink models in terms of size, behavioral dynamics and complexity.

2) Fault Seeding: We use the FIM prototype tool [28] to systematically inject faults into a model. We selected the faults to be injected based on the ones typically used in the literature [29]–[32]. In particular, we considered the following categories:

- *Sensor faults*: Stuck-at, Noise, Bias/Offset.
- *Hardware faults*: Bit-flip (single and multiple).
- *Network faults*: Package drop, Time delay.
- *Block mutations*: Wrong relational operator, wrong logical operator, wrong arithmetic operator.

For each of the above-mentioned faults, FIM creates customized fault blocks with flags to control their activation. In particular, FIM seeds faults in a model by (1) inserting fault blocks, and by (2) replacing blocks with fault blocks, exploiting a custom library of faults and mutations. In order to attain heterogeneity, we seeded faults of varied types in different parts (target locations) of the SUTs, particularly at distinct hierarchical depths. Table IV shows the number of seeded faults (and the group to which they belong) for each subject. We seeded a total of 15, 45 and 20 faults in ATCS, AECS and AFCS models, respectively.

Prior to testing the fault model, we instrumented it so as to log all the internal signals. We assigned a unique name to each signal and enabled data logging by the simulation engine. Using the activation flag of the seeded fault block injected by FIM, we activated and deactivated the seeded faults to create different faulty variants. A faulty variant corresponds to the instrumented faulty model with *faults of interest* (FOI) activated. FOI could be one or more depending on the user and the task at hand.

To answer **RQ2**, we created different faulty variants of our subjects, each faulty variant with a different number of active fault blocks. More precisely, for each subject, we created three sets of faulty variants s.t. the number of active fault blocks is 1, 2 and 3 in the first, second, and third set, respectively. For ATCS, each set has 15 faulty variants. For AECS, each set has 45 faulty variants, and for AFCS, each set has 20 faulty variants. Therefore, we created $\{45, 135, 60\}$ faulty variants of {ATCS, AECS, AFCS}, respectively, i.e., 240 faulty variants in total. For fair evaluation of our approach, we ensured that the activated fault blocks are of different types and located at different parts of the models.

3) Setup for Equivalence Testing: In order to evaluate our approach in absence of STL specifications, we assume that the original ATCS, AECS and AFCS are reference models, and that their faulty variants are models-under-test. We then use the non-equivalence witness trace and apply `E-FL` to localize its cause, i.e., the location(s) of the injected fault(s).

4) Test Suite and Test Oracle: We use Adaptive Random Testing (ART) [16] to generate the initial test suite ($\mathcal{TS}$). ART is a baseline method that uniformly samples test cases within valid input ranges. For each case study, we generate an initial $\mathcal{TS}$ of 100 test cases.

In the case of `STL-FL`, we used the monitor generated from the STL specification by the RTAMT library as the test oracle. In the case of `E-FL`, we implemented a procedure that computes the distance between the reference and the tested model behaviors as our test oracle.

5) Evaluation metrics: We evaluated the results obtained using `STL-FL` and `E-FL` on the following aspects: Scope Reduction, Fault Localization Cost, Fault Localization Accuracy and Computation Time. *Scope Reduction* evaluates our approach based on the degree of reduction in the overall model variables to a relatively small number of suspicious variables that can best localize the faults. The metric *Fault Localization Cost* indicates the absolute number of blocks that require the attention of the engineers to localize the fault and fix it. *Fault Localization Accuracy* indicates the efficiency of the approach in terms of fault detection. At last, we analyze the computation time of our proposed approach.

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

TABLE III
KEY INFORMATION ABOUT THE SIMULINK MODELS OF OUR SUBJECTS.

Model Name	$\#Blocks$	$\#Lines$	ϕ	q_T
Automatic Transmission Controller System (ATCS)	65	92	$\Box((v \leq \bar{v}) \wedge (\omega \leq \bar{\omega}))$	30
Aircraft Elevator Control System (AECS)	825	577	$\Box(\uparrow (cmd \geq m) \rightarrow \Diamond_{[0,T]}\Box_{[0,a]}(\lvert cmd - pos\rvert \leq n))$	10
Abstract Fuel Control System (AFCS)	253	283	$\Box_{[T_{start},T_{stop}]}\neg((AF > Ref - tol) \vee (AF < -Ref + tol))$	40

TABLE IV
INFORMATION OF SEEDED FAULTS IN EACH CASE STUDY.

Type	ATCS	AECS	AFCS
Sensor faults	3	20	9
Hardware faults	-	17	6
Network faults	9	-	4
Block mutations	3	8	1
Total	15	45	20

C. Results

We report the experimental results obtained w.r.t. our evaluation criteria. As a running example, we consider the model of ATCS. For the sake of illustration, we evaluate STL-FL and E-FL against the following test settings:

- **Test 1**: [One-fault] 'Stuck-at 0' fault within the *Engine* subsytem.
- **Test 2**: [Two-fault] As in Test 1 and 'Time Delay' fault within the *Transmission/TransmissionRatio* subsystem.
- **Test 3**: [Three-fault] As in Test 2 and 'Sum to Product mutation' within the *Engine* subsystem.

For each of the aforementioned test setting, we use fault localization procedures STL-FL and E-FL to identify the SOI and their corresponding blocks in the SUT. We then analyze the degree of reduction achieved for the fault identification as shown in Table V. Column *Test* points out to the type of test setting. Column *Approach* specifies the fault localization technique. Column $\#SOI$ (Reduction in $\#Vars$) represents the number of Signals of Interest; the degree of reduction achieved in the number of variables for the analyzed fault is indicated in parenthesis. Note that SOI indicates the suspicious variables that misbehave first in time with considerable anomalies. Column FL_Cost (Reduction in $\#Blocks$) denotes the Fault localization cost (abbreviated as FL_Cost) and the corresponding degree of reduction achieved is reported between parentheses. FL_Cost is based on the absolute number of blocks that need to be inspected to localize the fault(s) in the faulty model. Column *Fault(s) detected* specifies whether the faulty component(s) is identified correctly.

From Table V, we can observe that our approach can find out the root cause of the fault by correctly identifying the faulty component for all the three test settings. In Test 1, STL-FL yields only one SOI, signifying a considerably high reduction of 98.5% in the number of model variables. It is noteworthy that the reduction in number of variables and blocks is computed using the respective values for the instrumented fault model. After fault injection and model instrumentation, the total number of blocks in ATCS is 77 and the total number of model variables is 67. Referring to Table V, E-FL identifies two SOI in Test 1 with a reduction of 97%. It is worth mentioning that emphasis on a small subset of the suspicious signals allows significant reduction in the number of variables. This permits the engineers to focus on a relatively lower number of signals, thereby making the debugging process easier. Further, we observe that the fault localization cost is low for all the three tests. Using STL-FL, we achieve 88.3%, 68.8% and 68.8% reduction in the number of blocks to be inspected for Tests 1, 2 and 3 respectively.

Compared to CPSDebug, STL-FL and E-FL not only reduce the scope (for both model variables and blocks), but also provide correct identification of faulty components. We observe that despite higher reductions in the number of suspicious signals for more number of faults (as in Test 3), CPSDebug is unable to correctly identify the fault location.

We summarize the degree of reduction in model variables using STL-FL and E-FL in terms of statistical evaluation metrics in Table VI. The values are obtained for all three sets of faulty variants of each subject. We observe that both STL-FL and E-FL offer considerably high reductions in the number of suspicious model variables. On an average, STL-FL and E-FL respectively provide reductions of nearly 92% and 90% for multi-fault (one- to three-fault) models.

Fault Localization Cost. To answer **RQ1** and **RQ2**, we compute the FL_Cost for all the 240 faulty models using STL-FL, E-FL and the baseline fault localization technique CPSDebug. Fig. 3 shows the distributions of the FL_Cost values for all three benchmarks for one-fault, two-fault and three-fault models. Each box-plot in Fig. 3 consists of 80 points (15 for ATCS, 45 for AECS and 20 for AFCS) corresponding to 80 faulty variants in each of the one-, two- and three-fault models: the horizontal axis shows the localization approach while the vertical axis indicates the FL_Cost.

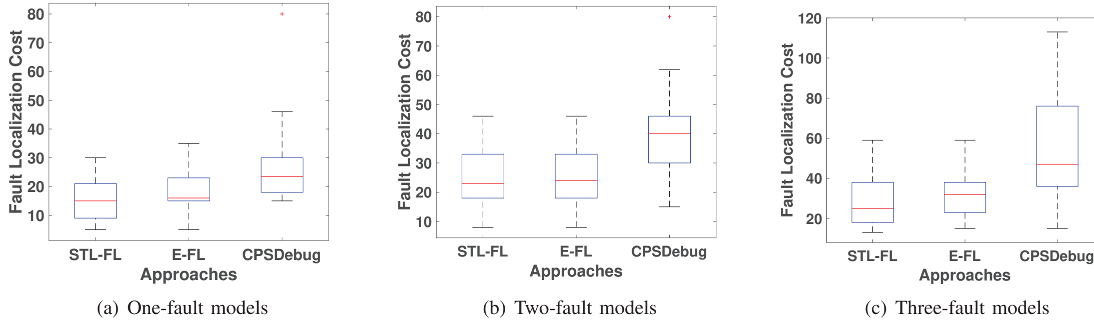
For statistical evaluation of the fault localization ability of our approaches against CPSDebug, we performed the well-known and conventional non-parametric Wilcoxon rank-sum statistical test [33], [34] with 5% degree of significance. The statistical test results reveal that for multi-fault models (one- to three-faults activated), our approaches STL-FL and E-FL are always significantly better than CPSDebug (i.e., the obtained p-values < 0.05).

In summary, our proposed approach exhibits considerable improvement in the fault localization cost compared to CPSDebug. On average, STL-FL reduces the FL_Cost by

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

TABLE V
SCOPE REDUCTION AND FAULT DETECTION IN ATCS.

Test	Approach	$\#SOI$ (Reduction in $\#Vars$)	FL_Cost (Reduction in $\#Blocks$)	Fault(s) detected
1	STL-FL	1 (98.5%)	9 (88.3%)	✓
	E-FL	2 (97.0%)	9 (88.3%)	✓
	CPSDebug	8 (88.1%)	25 (67.5%)	✓
2	STL-FL	5 (92.5%)	24 (68.8%)	✓
	E-FL	7 (89.5%)	24 (68.8%)	✓
	CPSDebug	8 (88.1%)	25 (67.5%)	× (Only one faulty component identified)
3	STL-FL	12 (82.1%)	24 (68.8%)	✓
	E-FL	12 (82.1%)	24 (68.8%)	✓
	CPSDebug	9 (86.5%)	34 (55.8%)	× (Only one faulty component identified)



(a) One-fault models

(b) Two-fault models

(c) Three-fault models

Fig. 3. Distributions of FL_Cost for one-fault to three-fault models.

TABLE VI
EMPIRICAL EVALUATION OF OUR APPROACH.

Model		Reduction in model variables			
		STL-FL		E-FL	
		Mean	SD	Mean	SD
ATCS	One-fault	0.9463	0.0263	0.9328	0.0120
	Two-fault	0.8975	0.0158	0.8892	0.0118
	Three-fault	0.8199	0.0143	0.8003	0.0092
AECS	One-fault	0.9742	0.0019	0.9577	0.0132
	Two-fault	0.9526	0.0142	0.9217	0.0113
	Three-fault	0.9021	0.0155	0.8835	0.0129
AFCS	One-fault	0.9704	0.0037	0.9602	0.0167
	Two-fault	0.9364	0.0111	0.9153	0.0149
	Three-fault	0.8926	0.0237	0.8748	0.0265

*SD: Standard Deviation

approximately 43% while E-FL reduces the FL_Cost by nearly 35% compared to CPSDebug.

Fault Localization accuracy. We analyze the overall fault localization accuracy of the proposed approach and compare the results with the baseline technique. Table VII summarizes the fault localization accuracy indicated as 'total number of fault models for which the faults were correctly identified/the total number of fault models'. The value in parenthesis represents the $\%$ fault localization accuracy. We observe that both STL-FL and E-FL improve the fault localization accuracy over CPSDebug, and are more robust when the number of active faults in the underlying models increases. Specifically,

TABLE VII
FAULT LOCALIZATION ACCURACY.

Approach	ATCS	AECS	AFCS
STL-FL	45/45 (100%)	135/135 (100%)	60/60 (100%)
E-FL	45/45 (100%)	135/135 (100%)	60/60 (100%)
CPSDebug	22/45 (48.88%)	53/135 (39.25%)	25/60 (41.66%)

CPSDebug works well with one-fault models, but fails to localize faults in multi-fault models. Further, referring to Table V, it can be observed that the performance of CPSDebug degrades with increasing number of active faults.

Computational cost. Table VIII provides the computation time of our procedure applied to the three subjects. Note that the costs for *Testing* and *Localizing* denote the average time taken by the approach to localize a bug in a model. The values are based on the analysis of different fault variants (one-to-three fault models) of our subjects for a failing-passing test pair. We observe that *Testing* is more dominant than *Localizing*. This is mainly due to the search task (optimization) for finding the most similar test case. Note that specification monitoring (by calling external tool) involved in STL-FL leads to slightly higher computation cost over E-FL.

Referring to Table VIII, both STL-FL and E-FL have an acceptable global overhead of {23.7, 27.6, 26.7}s and {23.2, 26.9, 25.1}s respectively, to locate a bug in {ATCS, AECS, AFCS} models. The higher computation cost of CPSDebug is mainly due to specification mining using Daikon and

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
COMPUTATIONAL COST (IN SECONDS).

Approach		ATCS	AECS	AFCS
STL-FL	*Testing*	21.4	24.5	23.9
	Localizing	2.3	3.1	2.8
E-FL	*Testing*	20.9	23.8	22.3
	Localizing	2.3	3.1	2.8
CPSDebug		294.1	3394.6	2174.8

TkT. Our approach demonstrated its efficiency, completing the diagnosis in about 25 seconds on average, in contrast to CPSDebug which requires an order of magnitude more time. *In summary*, the answer to **RQ3** is that our approach is more computationally efficient than the baseline and offers a low-cost solution for debugging CPS Simulink models.

VI. RELATED WORK

In the recent past, CPS researchers and practitioners have adopted two key methodologies for evaluating the safety and reliability aspects of CPS: *set-based reachability analysis* [35]–[37] and *rigorous testing* such as differential testing [38] and falsification analysis [5], [6]. Our approaches utilize the characteristics of both the differential testing (in E-FL) and falsification analysis (in STL-FL). From the perspective of software engineering, fault localization has remained to offer continually increasing challenges [39], [40]. Although the presence of faults can be discovered by analyzing the falsifying traces, counterexamples or related manifestations of the fault, yet the identification of the precise fault location in the SUT remains a tedious task. Recent works use traditional and iterative statistical debugging techniques [3], [16], [41] for generating a ranked list of blocks that need the further attention of an engineer to identify fault locations. Our approach E-FL, unlike statistical techniques, uses (1) differential testing to identify the failing test cases and (2) optimization to find the new test candidates to localize faults.

Similar to our approach STL-FL that is guided by a STL specification for detecting bugs, the work in [42] analyzes the neighborhoods of falsifying traces of a CPS for a formalized property. The goal is to identify and analyze the segments of inputs that cause the violation of the specification as a whole. This approach may be possibly used for systematic refinement of the test candidates to assist in debugging. Another work on spectrum-based fault localization combines trace diagnostics with model slicing technique [4]. Since there is no empirical evaluation of this SBFL technique, the expected debugging effort is unclear. Further, the work is carried out under the assumption that the faults are injected in specific components of the SUT. This drastically reduces the problem space and makes it difficult to reckon the potency of the SBFL technique.

CPSDebug [10] is a recently developed tool which employs a gray-box testing approach for localizing bugs and explaining failures in CPS designs. The underlying concepts of CPSDebug, that combines testing, specification mining, and failure analysis for exposing faults, are discussed extensively in [12]. In our empirical evaluation, our approach outperformed CPSDebug. The work by Singh and Saha [11] uses a matrix decomposition-based bug localization procedure based on the falsification of STL properties. The approach offers a set of suspected signals to the engineer as the alleged cause of the falsification of the STL specification.

In contrast to the state-of-the-art, we consider a wider range of fault types and mutation operators for Simulink models. Moreover, we do not make any assumptions on the location of the seeded fault/mutation in the SUT, except that we ignore faults in Stateflow charts. Contrary to the existing fault localization techniques, our approach unifies (1) equivalence testing with search-based testing, and (2) specification monitoring with search-based testing, to identify the *root* cause of anomalous events that eventually led to the observed failures. The result is a tailor-made list of suspicious model variables and blocks, ideally appropriate for engineers to assist in their debugging tasks.

VII. CONCLUSION

We presented a novel procedure to localize faults in Simulink models of safety-critical CPS. The proposed approach, driven by a STL property, uses failing and automatically generated passing executions similar to the failing ones to identify the anomalous signals, and consequently the likely faulty blocks. We instantiated our approach with the notion of equivalence testing where the fault localization is driven by implicit specifications.

We demonstrated the effectiveness of our approach with three use cases. Experimental results show that our approach is able to effectively localize multiple faults with admissible fault localization cost, outperforming CPSDebug.

As part of future work, we envision the extension of our approach by integrating alternative test generation strategies and analyzing their effect on the overall fault localization accuracy. We further aim to explore the interplay between different failing executions for a faulty system that violates a specification, and build refined strategies to pick the most promising test cases from a test suite.

ACKNOWLEDGMENT

This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien.

REFERENCES

[1] Mathworks. (2022) Simulink — simulation and model-based design. [Online]. Available: https://in.mathworks.com/products/simulink.html

[2] B. Liu, S. Nejati, L. C. Briand *et al.*, "Improving fault localization for simulink models using search-based testing and prediction models," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 359–370.

[3] B. Liu, S. Nejati, L. Briand, T. Bruckmann *et al.*, "Localizing multiple faults in simulink models," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 146–156.

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.

[4] E. Bartocci, T. Ferrère, N. Manjunath, and D. Ničković, "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)*, 2018, pp. 197–206.

[5] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 167–170.

[6] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-taliro: A tool for temporal logic falsification for hybrid systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 254–257.

[7] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, J. Kapinski, and X. Jin, "Falsification of safety properties for closed loop control systems," in *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, 2015, pp. 299–300.

[8] S. Sankaranarayanan and G. Fainekos, "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*, 2012, pp. 125–134.

[9] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[10] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, "CPSDebug: Automatic failure explanation in CPS models," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 5, pp. 783–796, 2021.

[11] N. K. Singh and I. Saha, "Specification-guided automated debugging of CPS models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4142–4153, 2020.

[12] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, "Automatic failure explanation in CPS models," in *International Conference on Software Engineering and Formal Methods*. Springer, 2019, pp. 69–86.

[13] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "Mimic: locating and understanding bugs by analyzing mimicked executions," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 815–826.

[14] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[15] A. Donzé and O. Maler, "Robust satisfaction of temporal logic over real-valued signals," in *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 2010, pp. 92–106.

[16] B. Liu, S. Nejati, L. C. Briand *et al.*, "Effective fault localization of automotive simulink models: achieving the trade-off between test oracle effort and fault localization accuracy," *Empirical Software Engineering*, vol. 24, no. 1, pp. 444–490, 2019.

[17] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancic, A. Gupta, and G. J. Pappas, "Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems," in *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, 2010, pp. 211–220.

[18] D. Yadav, "Blood coagulation algorithm: A novel bio-inspired metaheuristic algorithm for global optimization," *Mathematics*, vol. 9, no. 23, p. 3011, 2021.

[19] D. Nickovic and T. Yamaguchi, "RTAMT: online robustness monitors from STL," in *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, 2020, pp. 564–571.

[20] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[21] F. Pastore, D. Micucci, M. Guzman, and L. Mariani, "TkT: Automatic inference of timed and extended pushdown automata," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 617–636, 2022.

[22] B. Hoxha, H. Abbas, and G. E. Fainekos, "Benchmarks for temporal logic requirements for automotive systems." *ARCH@ CPSWeek*, vol. 34, pp. 25–30, 2014.

[23] J. Ghidella and P. Mosterman, "Requirements-based testing in aircraft control design," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2005, p. 5886.

[24] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts, "Powertrain control verification benchmark," in *Proceedings of the 17th international conference on Hybrid systems: computation and control*, 2014, pp. 253–262.

[25] Mathworks. (2022) Modeling an automatic transmission controller. [Online]. Available: https://in.mathworks.com/help/simulink/slref/modeling-an-automatic-transmission-controller.html

[26] Mathworks. (2022) Detect faults in aircraft elevator control system. [Online]. Available: https://in.mathworks.com/help/stateflow/ug/fault-detection-control-logic-in-an-aircraft-elevator-control-system.html

[27] Mathworks. (2022) Modeling a fault-tolerant fuel control system. [Online]. Available: https://in.mathworks.com/help/simulink/slref/modeling-a-fault-tolerant-fuel-control-system.html

[28] E. Bartocci, L. Mariani, D. Nickovic, and D. Yadav, "FIM: fault injection and mutation for simulink," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[29] T. Fabarisov, I. Mamaev, A. Morozov, and K. Janschek, "Model-based fault injection experiments for the safety analysis of exoskeleton system," *arXiv preprint arXiv:2101.01283*, 2021. [Online]. Available: https://arxiv.org/abs/2101.01283

[30] I. Pill, I. Rubil, F. Wotawa, and M. Nica, "Simultate: A toolset for fault injection and mutation testing of simulink models," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 168–173.

[31] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "Modifi: a model-implemented fault injection tool," in *International Conference on Computer Safety, Reliability, and Security*, E. Schoitsch, Ed. Berlin, Heidelberg: Springer, 2010, pp. 210–222.

[32] M. Saraoğlu, A. Morozov, M. T. Söylemez, and K. Janschek, "Errorsim: A tool for error propagation analysis of simulink models," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017, pp. 245–254.

[33] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[34] J. D. Gibbons and S. Chakraborti, *Nonparametric statistical inference*. CRC press, 2014.

[35] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2e2: A verification tool for stateflow models," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 68–82.

[36] G. Frehse, C. L. 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 *International Conference on Computer Aided Verification*. Springer, 2011, pp. 379–395.

[37] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 258–263.

[38] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 621–631.

[39] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

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

[41] B. Liu, Lucia, S. Nejati, L. C. Briand, and T. Bruckmann, "Simulink fault localization: an iterative statistical debugging approach," *Software Testing, Verification and Reliability*, vol. 26, no. 6, pp. 431–459, 2016.

[42] R. D. Diwakaran, S. Sankaranarayanan, and A. Trivedi, "Analyzing neighborhoods of falsifying traces in cyber-physical systems," in *Proceedings of the 8th International Conference on Cyber-Physical Systems*, 2017, pp. 109–119.

Authorized licensed use limited to: TU Wien Bibliothek. Downloaded on May 22,2023 at 14:11:02 UTC from IEEE Xplore. Restrictions apply.