Localizing Faults in Simulink/Stateflow Models with STL

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

Thomas Ferrère
Institute of Science and Technology (IST), Austria
thomas.ferrere@ist.ac.at

Niveditha Manjunath
Austrian Institute of Technology (AIT), Austria
TU Wien, Austria
niveditha.manjunath@ait.ac.at

Dejan Ničković
Austrian Institute of Technology (AIT), Austria
dejan.nickovic@ait.ac.at

ABSTRACT

Fault-localization is considered to be a very tedious and time-consuming activity in the design of complex Cyber-Physical Systems (CPS). This laborious task essentially requires expert knowledge of the system in order to discover the cause of the fault. In this context, we propose a new procedure that aids designers in debugging Simulink/Stateflow hybrid system models, guided by Signal Temporal Logic (STL) specifications. The proposed method relies on three main ingredients: (1) a monitoring and a trace diagnostics procedure that checks whether a tested behavior satisfies or violates an STL specification, localizes time segments and interfaces variables contributing to the property violations; (2) a slicing procedure that maps these observable behavior segments to the internal states and transitions of the Simulink model; and (3) a spectrum-based fault-localization method that combines the previous analysis from multiple tests to identify the internal states and/or transitions that are the most likely to explain the fault. We demonstrate the applicability of our approach on two Simulink models from the automotive and the avionics domain.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Model verification and validation*;

ACM Reference Format:
Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Ničković. 2018. Localizing Faults in Simulink/Stateflow Models with STL. In *HSCC '18: 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), April 11–13, 2018, Porto, Portugal.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178126.3178131

1 INTRODUCTION

The development of CPS is a complex process involving several stages. In the concept phase of the design, the engineers often use the MathWorks™ Simulink toolset to model the CPS functionality. These models are sophisticated hybrid systems – they involve an intricate interaction between the continuous and discrete dynamics.

The verification and testing of Simulink/Stateflow CPS models is a challenging task that has received considerable attention in recent years. Falsification-based testing [2, 7, 25, 27] is a well-established technique to efficiently find bugs in Simulink/Stateflow models. This method uses a formal specification, expressed in a language such as Signal Temporal Logic (STL) [17], and a monitor, checking each simulation trace for correctness against the specification and optionally providing an indication as to how far the trace is from violation. While falsification-based testing has proven effective in finding input sequences that result in specification violations, it does not provide additional information that would help to localize the error within the model and thus aid the debugging process.

Conventionally, debugging is a tedious and time-consuming activity during the design of CPS. When observing some faulty behavior in the simulation trace of a model, the engineer first needs to localize the fault in order to correct the model. This manual fault-localization process is a cumbersome activity and generally requires a well-trained and expert engineer to discover the cause of the problem. When the system is modeled in Simulink/Stateflow, the engineer can use the Model Slicer component of the Simulink Design Verifier [19] to identify components that were active during specified time intervals. Nevertheless, this tool still requires considerable manual interaction and in particular requires to visually identify and mark time intervals of interest.

In this paper we propose a novel automated procedure that aids the designer in localizing Simulink/Stateflow faults from simulations that violate STL specifications. We build our approach on top of the trace diagnostics method of [8] that isolates small segments of simulation traces sufficient to imply the violation of STL specification. This black-box technique ignores the model, but provides valuable spatial and temporal information about the interface variables associated with the fault and the times of their critical involvement. We then use model slicing [23] to identify model components that influence the marked variables, and propagate the trace diagnostics information to these components. We repeat this procedure over all the (failed) tests in the test suite, and use spectrum-based fault localization [1] to find the internal states and/or transitions that are the most likely to cause the fault. Spectrum-based fault localization is a lightweight statistical technique that analyzes passed and failed tests to rank model variables according to their suspiciousness. To the best of our knowledge, this is the first application of spectrum-based fault localization in

HSCC '18, April 11–13, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5642-8/18/04…$15.00
https://doi.org/10.1145/3178126.3178131

conjunction with a sophisticated logic-based oracle (here a temporal logic monitor). This enables us to refine the set of variables fed to the method, and substantially improve the outcome.

1.1 Overview

The proposed approach consists of three main steps: (1) specification monitoring and trace diagnostics; (2) model slicing; and (3) spectral analysis. We assume a hybrid system given in the form of a Simulink/Stateflow model, and a set of tests each consisting of an input trace and an STL specification, used as an oracle.

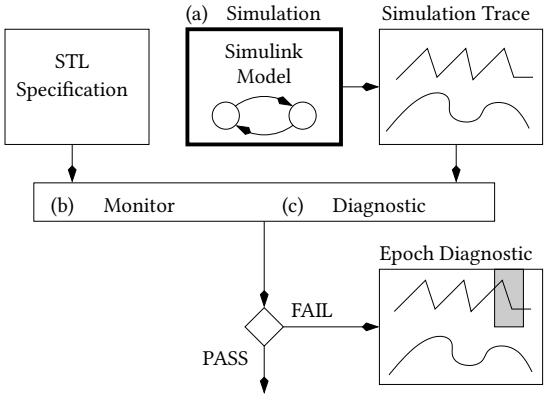


Figure 1: System diagnostics procedure overview – monitoring and trace diagnostics.

Specification monitoring and trace diagnostics. In this step, we evaluate a single test on the model. In other words, we first simulate the model with the input trace, thus generating the output trace (Figure 1 (a)) and use a monitoring procedure to check whether these observable traces satisfy the STL specification (Figure 1 (b)). If we detect a specification violation, we use a trace diagnostics method [8] to localize all the segments in the observable traces that are responsible for the detected fault (Figure 1 (c)). The outcome of this procedure is the localization in time of the fault at the observable interface of the model. Note that the resulting analysis does not relate the fault to the model internal signals.

Model slicing. In this step, we employ model slicing [23] to map the results from the monitoring and trace diagnostics to the internal states and transitions in the model. The trace diagnostics procedure identifies (among other things) a subset of interface variables that are responsible for the detected fault. We use this information to perform model slicing and find all the components in the model, particularly in Stateflow charts, that may be involved in the specification violation (see the light-gray-shaded chart in Figure 2). In the next step, we map the intervals computed by the trace diagnostics procedure to the trace that records the evolution of the chart's states and transitions over time (see the dark-gray band in the trace of the highlighted Stateflow chart in Figure 2). We can now distinguish between internal states and transitions that were active during the correct and faulty test segments.

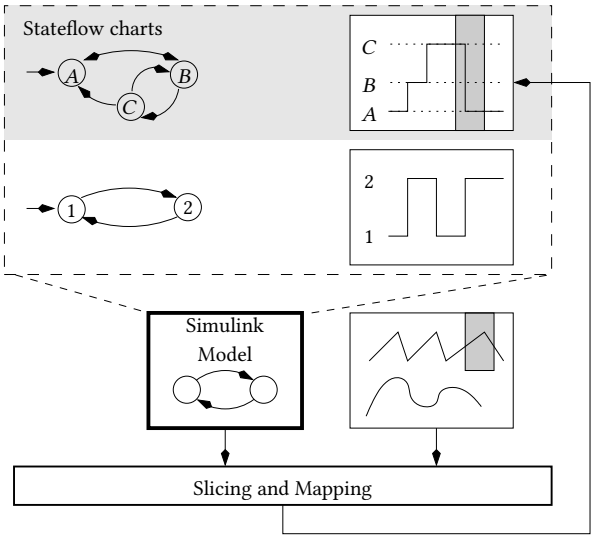
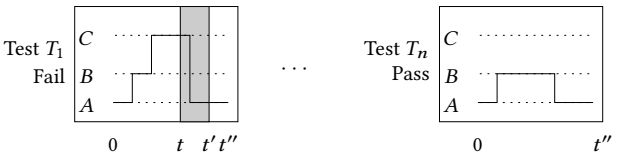


Figure 2: System diagnostics procedure overview – model slicing.

Spectrum-based Fault Localization. We apply the above procedure to all the available tests and collect the information in a tabular form (see Figure 3). Each test is partitioned into a sequence of segments, with each segment labeled with a pass or fail flag. We assign a column in the table to each test segment. The states and transitions appear as rows in the table. Whenever a state or a transition was activated in a given test segment, we mark the corresponding cell with a Boolean flag. Finally, we apply the spectrum-based fault localization [1] procedure to find the states and the transitions that are the most likely to cause the detected fault.



State/Tran.	$T_1[0, t)$	$T_1[t, t')$	$T_1[t', t'')$	...	$T_n[0, t'')$
A	×	×	×		×
B	×				×
C	×	×			
$A \to B$	×				×
$B \to A$					×
$B \to C$	×				
$C \to A$		×			
$C \to B$					×
Verdict	Pass	Fail	Pass	...	Pass

Figure 3: System diagnostics procedure overview – spectrum-based fault localization.

2 SYSTEM DIAGNOSTICS

In this section, we present our approach for localizing faults in Simulink/Stateflow models. In Section 2.1, we define the syntax

and semantics of Signal Temporal Logic (STL) and in Section 2.2 we discuss the monitoring procedure. Section 2.3 introduces the trace diagnostics method. In Section 2.4 we describe the model slicing procedure while in Section 2.5 we show how to apply the spectrum-based fault localization in our framework.

2.1 Signal Temporal Logic

STL [17] extends the continuous-time Metric Temporal Logic [13] with numerical predicates over real-valued variables. STL enables reasoning about real-time properties at the interface between components that exhibit both discrete and continuous dynamics.

We denote by X and P finite sets of *real* and *propositional* variables. We let $w : \mathbb{T} \to \mathbb{R}^m \times \{0,1\}^n$ be a multi-dimensional *signal*, where $\mathbb{T} = [0, d) \subseteq \mathbb{R}$, $m = |X|$ and $n = |P|$. Given a variable $v \in X \cup P$ we denote by w_v the *projection* of w on its component v. Given a value $t \in \mathbb{R}_{\geq 0}$ and interval I, we denote by $t \oplus I$ the Minkowski sum $\{t + t' \mid t' \in I\}$.

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

$$\varphi \quad ::= \quad p \mid x \sim c \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \Diamond_I \varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

where $p \in P$, $x \in X$, $\sim \in \{<, \leq\}$, $c \in \mathbb{Q}$, and $I \subseteq \mathbb{R}_{\geq 0}$ is an arbitrary interval. We omit the subscript $[0, \infty)$ and write $\Diamond \varphi, \Box \varphi$ instead of $\Diamond_{[0,\infty)} \varphi, \Box_{[0,\infty)} \varphi$ respectively.

The semantics of STL is defined in terms of the *satisfiability relation* $(w, t) \models \varphi$, indicating that signal w satisfies φ at time t. Given a signal w over $\mathbb{T}$, relation $\models$ is defined for all $t \in \mathbb{T}$ inductively as follows:

$$\begin{array}{lcl}
(w,t) \models p & \leftrightarrow & w_p[t] = 1 \\
(w,t) \models x \sim c & \leftrightarrow & w_x[t] \sim c \\
(w,t) \models \neg\varphi & \leftrightarrow & (w,t) \not\models \varphi \\
(w,t) \models \varphi_1 \vee \varphi_2 & \leftrightarrow & (w,t) \models \varphi_1 \text{ or } (w,t) \models \varphi_2 \\
(w,t) \models \Diamond_I \varphi & \leftrightarrow & \exists t' \in (t \oplus I) \cap \mathbb{T} : (w,t') \models \varphi \\
(w,t) \models \varphi_1 \,\mathcal{U}\, \varphi_2 & \leftrightarrow & \exists t' \in (t, +\infty) \cap \mathbb{T} : (w,t') \models \varphi_2 \text{ and} \\
 & & \forall t'' \in (t,t') : (w,t'') \models \varphi_1
\end{array}$$

We say that a signal w *satisfies* a STL formula φ, denoted by $w \models \varphi$, iff $(w, 0) \models \varphi$.

Note that standard presentations of STL such as [17] often provide a *timed until* operator $\mathcal{U}_I$ that is primitive in the syntax, other temporal operators deriving from it. According to an observation made in multiple places, the current syntax is as general as the standard one. This observation, known as the *until rewrite* rule, enables us to recover several forms of *timed until* operators as the following abbreviations:

$$\varphi \,\mathcal{U}_{[a,\infty)}\, \psi \equiv \Box_{(0,a)}\, \varphi \wedge \Box_{(0,a]}(\psi \vee (\varphi \wedge (\varphi \,\mathcal{U}\, \psi)))$$
$$\varphi \,\mathcal{U}_{[a,b]}\, \psi \equiv \Diamond_{[a,b]}\, \psi \wedge \varphi \,\mathcal{U}_{[a,\infty)}\, \psi$$

Other forms of operator *until*, timed with open and semi-open intervals, can be defined similarly, see [18].

2.2 Monitoring STL

Following the inductive semantics of STL, in order to know the truth value of a given formula at every time point, it is sufficient to know the truth value of its *direct* subformulas at every time point. This is the basis of the monitoring algorithm of [17], which we outline in this section.

The satisfaction signal of formula φ relative to w, denoted w_φ, is the Boolean signal defined as follows:

$$w_\varphi[t] = \begin{cases} 1 & \text{if } (w,t) \models \varphi \\ 0 & \text{otherwise} \end{cases}$$

This definition is consistent with the notion of projection when $\varphi \equiv p$ for some atomic proposition $p \in P$.

The monitoring algorithm runs offline and is recursive on the formula structure. To compute the satisfaction signal w_φ for a given φ, we assume a partition of the temporal domain $\mathbb{T} = [0, d)$ into

$$[t_1, t_1], (t_1, t_2), \ldots, [t_{n-1}, t_{n-1}], (t_{n-1}, t_n)$$

with $t_1 = 0$, $t_n = d$, such that the satisfaction signal of the subformulas of φ are constant over every (t_i, t_{i+1}) for $i = 1, \ldots, n-1$. The value of w_φ is then computed in a manner specific to each operator, as follows.

- Atomic formulas. The satisfaction signals of a Boolean variable p is given directly by the trace. For atomic formulas of the form $x \sim c$, we first interpolate the times $t_1, \ldots, t_n$ at which signal x crosses threshold c, and compute the Boolean value over every interval of the resulting partition.
- Disjunction. The satisfaction signal of formula $\varphi \vee \psi$ is constant over every interval in the partition, and given by taking the disjunction of w_φ and w_ψ over that interval, since

  $$w_{\varphi \vee \psi}[t] = w_\varphi[t] \vee w_\psi[t]$$

  for all $t \in \mathbb{T}$.
- Negation. Similarly the value of $w_{\neg\varphi}$ is constant over intervals in the partition, and given by:

  $$w_{\neg\varphi}[t] = \neg w_\varphi[t]$$

  for all $t \in \mathbb{T}$.
- Timed eventually. We create a new partition of $\mathbb{T}$ of the form $[t'_1, t'_1], (t'_1, t'_2), \ldots, [t'_{m-1}, t'_{m-1}], [t'_{m-1}, t'_m)$ from the set of times $\{0, d\} \cup \{t_i - \inf I, t_i - \sup I \mid 1 \leq i \leq n\} \cap (0, d)$. The satisfaction signal of $\Diamond_I$ over every interval (t'_i, t'_{i+1}) is constant, and given by:

  $$w_{\Diamond_I \varphi}[t] = \bigvee_{t' \in t \oplus I} w_\varphi[t']$$

- Untimed until. The satisfaction signal of formula $\varphi \,\mathcal{U}\, \psi$ is constant over every interval in the partition. Its value can be determined from that of w_φ and w_ψ using the rule $w_\varphi[r] = w_\varphi[r^+]$ over singular intervals $[r, r]$, and the rule

  $$w_{\varphi \,\mathcal{U}\, \psi}[s^-] = \begin{cases} 1 & \text{if} \quad w_\varphi[s^-] = w_\psi[s^-] = 1 \\ 1 & \text{if} \quad w_\varphi[s^-] = w_\psi[s] = 1 \\ 0 & \text{if} \quad w_\varphi[s^-] = 0 \\ w_{\varphi \,\mathcal{U}\, \psi}[s] & \text{otherwise} \end{cases}$$

  over open intervals (r, s) of the partition. Here $w_\varphi[r^+] = w_\varphi[s^-]$ denote the value of w_φ over the interval (r, s).

There are linear-time algorithms to compute such satisfaction signals, the reader is referred to [18] for more details.

2.3 Trace Diagnostics for STL

In this section, we introduce a novel trace diagnostics procedure for STL. This procedure is based on the notion of *epoch*, defined as a signal subset in terms of time and variables. Given a STL specification φ and a trace w that violates φ, the trace diagnostics procedure computes the epoch of w responsible for the detected fault. We illustrate our procedure with Example 2.1.

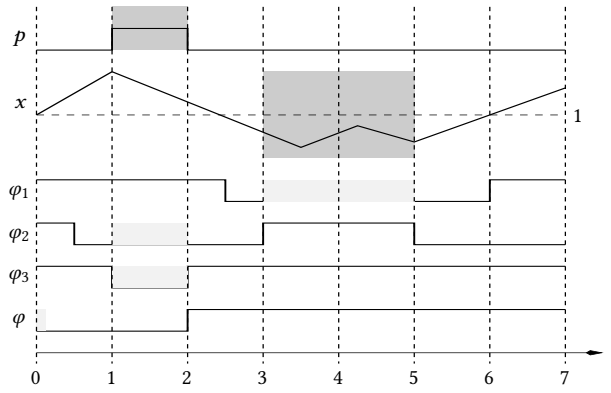


Figure 4: Monitoring and trace diagnostics for formula $\varphi \equiv \square(p \rightarrow \lozenge_{[2,3]}(x \geq 1))$ with subformulas $\varphi_1 \equiv x \geq 1$, $\varphi_2 \equiv \lozenge_{[2,3]}(x \geq 1)$ and $\varphi_3 \equiv p \rightarrow \lozenge_{[2,3]}(x \geq 1)$.

Example 2.1. Consider the STL formula $\varphi = \square(p \rightarrow \lozenge_{[2,3]}(x \geq 1))$. It requires that every time instant t at which p holds is followed by another instant that is within the interval $[t+2, t+3]$ and at which x is greater or equal to 1. Figure 4 depicts a signal that violates φ. Indeed, we can see that p holds during the interval $[1, 2]$, but x consistently stays below 1 throughout $[3, 5]$. We highlight in gray two epochs $[2, 3]$ and $[3, 5]$ associated to p and x respectively that together explain all the reasons for the violation of φ.

We introduce the *localization* operator Λ which is implicitly parameterized by the trace w. It takes as argument a formula φ and gives a mapping $\Lambda(\varphi) : \mathbb{T} \rightarrow 2^{(P \cup X) \times \mathbb{T}}$ associating the violation (dually, the satisfaction) of φ at a given time t to an epoch where the violation (satisfaction) occurs.

Definition 2.2 (Temporal Localization). Let φ be a formula, w a signal, and $t \in \mathbb{T}$. The epoch $\Lambda(\varphi)[t]$ is defined inductively as follows:

$$\Lambda(p)[t] = \{(p, t)\} \qquad \Lambda(x \sim c)[t] = \{(x, t)\}$$

$$\Lambda(\neg\varphi)[t] = \Lambda(\varphi)[t]$$

$$\Lambda(\varphi_1 \vee \varphi_2)[t] = \begin{cases} \Lambda(\varphi_1)[t] \cup \Lambda(\varphi_2)[t] & \text{if } w_{\varphi_1}[t] = w_{\varphi_2}[t] \\ \Lambda(\varphi_1)[t] & \text{else if } w_{\varphi_1 \vee \varphi_2}[t] = w_{\varphi_1}[t] \\ \Lambda(\varphi_2)[t] & \text{else if } w_{\varphi_1 \vee \varphi_2}[t] = w_{\varphi_2}[t] \end{cases}$$

$$\Lambda(\lozenge_I \varphi)[t] = \bigcup_{\substack{s \in t \oplus I \\ w_\varphi[s] = w_{\lozenge_I \varphi}[t]}} \Lambda(\varphi)[s]$$

$$\Lambda(\varphi_1 \,\mathcal{U}\, \varphi_2)[t] = \bigcup_{s \in (t, +\infty) \cap \mathbb{T}} \Lambda(\varphi_1)[s] \cup \Lambda(\varphi_2)[s]$$

Note that the localization operator is self-dual relative to negation. When $w \not\models \varphi$ the set $\Lambda(\varphi)[0]$ can be characterized as being a *violation epoch* of φ in signal w, based on the notion of *sub-model* defined below.

Definition 2.3 (Sub-Model). We call *partial signal* a function $v : (X \cup P) \times \mathbb{T} \rightarrow \{0, 1, \bot\}$, where $v_x[t] = \bot$ means that v is *undefined* over (x, t). A partial signal v is *sub-signal* of w, denoted $v \sqsubseteq w$, when $v_x[t] = w_x[t]$ for all $(x, t) \in (X \cup P) \times \mathbb{T}$ where v is defined. We say that partial signal v is a (*minimal*) *sub-model* of some formula φ when $w \models \varphi$ for all signals $w \sqsupseteq v$ (and v is minimal for order $\sqsubseteq$).

The notion of trace diagnostic proposed by [8] consists of a single sub-model of the negated formula under consideration. The present notion of violation epoch consists of the subset of the signal domain that appears in at least one such trace diagnostic.

Definition 2.4 (Epoch). Given a STL formula φ and a signal w such that $w \not\models \varphi$, we say that $E \subseteq (X \cup P) \times \mathbb{T}$ is a (*minimal*) *violation epoch* of φ relative to w when for all $(x, t) \in (X \cup P) \times \mathbb{T}$ it holds that $(x, t) \in E$ iff there exists a (minimal) sub-model $v \sqsubseteq w$ of $\neg\varphi$ such that $v_x[t] \neq \bot$.

Given a formula φ and signal w such that $w \not\models \varphi$, the set $\Lambda(\varphi)[0]$ is a violation epoch of φ relative to signal w. Note that this provides a small violation epoch, but not necessarily the minimal one. In particular, the localization could be made more precise for operator *until* along the lines of [8]. In general, the problem of finding the minimal violation (equivalently, satisfaction) epoch is at least as hard as satisfiability. This is because tautologies can be recognized as admitting $\emptyset$ as a satisfaction epoch. Deciding Metric Temporal Logic (MTL) [13] (and therefore STL) over bounded time domains has a computational cost at least exponential in space [22]. On the contrary, the violation epoch defined by $\Lambda(\varphi)[0]$ can be computed by the polynomial-time procedure that we now outline.

The idea is to apply operator $\Lambda(\varphi)$ symbolically over finite unions of intervals, by lifting this operator to intervals according to

$$\Lambda(\varphi)[r, s] = \bigcup_{t \in (r, s)} \Lambda(\varphi)[t]$$

assuming the satisfaction signal of φ is constant throughout $[r, s]$. Let us denote by $w_\varphi : \mathbb{T}^2 \rightarrow \{0, 1, \bot\}$ the lifting of w_φ to intervals, defined as

$$w_\varphi[r, s] = \begin{cases} 1 & \text{if } (w, t) \models \varphi \text{ for all } t \in (r, s) \\ 0 & \text{if } (w, t) \not\models \varphi \text{ for all } t \in (r, s) \\ \bot & \text{otherwise} \end{cases}$$

The lifted operator $\Lambda(\varphi)$ is then characterized by inductive rules similar to those of its scalar version, for instance

$$\Lambda(\lozenge_I \varphi)[r, s] = \bigcup_{\substack{t, u \in \mathcal{P}([r, s] \oplus I) \\ w_\varphi[r, s] = w_{\lozenge_I \varphi}[t, u]}} \Lambda(\varphi)[t, u]$$

where $\mathcal{P}([r, s] \oplus I)$ is a set of time points in $[r, s] \oplus I$ such that $w_\varphi[t, u] \neq \bot$ for every consecutive t, u. Starting from $[0, 0]$ for a main formula φ such that $w \not\models \varphi$, we obtain a violation epoch for φ in w that can be expressed using a finite union of intervals.

Example 2.5. We consider again the specification $\varphi = \square(p \rightarrow \lozenge_{[2,3]}(x \geq 1))$ from Example 2.1. We compute the violation epoch

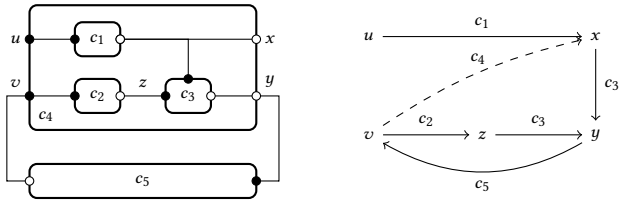


Figure 5: Example of system model: (a) block diagram; (b) signal-flow graph. Full nodes are input ports, shallow nodes are output ports. Components c_1, c_2, c_3 and c_5 are atomic, while component c_4 is hierarchical. Treating c_4 as atomic would yield the spurious edge indicated with a dashed line.

of φ in w as follows (see light- and dark-gray intervals in Figure 4):

$$\begin{aligned}\Lambda(\varphi)[0,0] &= \Lambda(p \rightarrow \Diamond_{[2,3]}(x \geq 1))[1,2] \\ &= \Lambda(p)[1,2] \cup \Lambda(\Diamond_{[2,3]}(x \geq 1))[1,2] \\ &= \Lambda(p)[1,2] \cup \Lambda(x \geq 1)[3,5] \\ &= \Lambda(p)[1,2] \cup \Lambda(x)[3,5] \\ &= \{(p,[1,2]),(x,[3,5])\}\end{aligned}$$

2.4 Model Slicing

We now present a procedure that, given a faulty signal x, finds in a Simulink model M the set of source signals that are upstream of x. We call this set the *cone of influence* of x, and denote it by $\mathsf{cone}_M(x)$. In a signal-flow graph, where signals are nodes and components are edge labels, this amounts to finding the set of back-reachable nodes. The internal representation of Simulink model is more hierarchical in nature, and requires special handling. Ignoring this hierarchical information would lead to over-approximations, as illustrated in Figure 5. In that example, the correct computation gives us $\mathsf{cone}_M(x) = \{u\}$ instead of $\{u, v\}$. One simple way to treat the hierarchy would be to flatten the model, i.e. "unbox" components that are contained in others, prior to applying the reachability procedure. This requires to explore the whole model a priori; instead we unbox components on the fly. Before further describing the procedure, we take some basic definitions.

A Simulink model is given in the form of a hierarchical block diagram with the following features. A diagram contains three types of elements: components, ports, and signals. These elements have the following relations in M:

- A component c is associated with a set of *input* ports, denoted $\mathsf{in}(c)$, and a set of *output* ports, denoted $\mathsf{out}(c)$, such that $\mathsf{in}(c) \cap \mathsf{out}(c) = \emptyset$.
- A port p is associated with a unique component denoted $\mathsf{block}(p)$, such that $p \in \mathsf{in}(\mathsf{block}(p)) \cup \mathsf{out}(\mathsf{block}(p))$.
- A signal x connects with a source port, denoted $\mathsf{src}(x)$, with a set of destination ports, denoted $\mathsf{dest}(x)$. For every port p there is at most one signal x such that $p = \mathsf{src}(x)$, and at most one signal y such that $p \in \mathsf{dest}(y)$.

Note that there are other types of ports, such as *enable* ports, which we ignore. We consider two types of Simulink components: atomic, and hierarchical. An atomic component cannot have its input ports directly connected to input ports of another component; similarly for output ports. A hierarchical component c can have its input ports directly connected to input ports of another component c', and similarly for output ports. Such a component c' which is called a *subsystem* of c.

We treat atomic components by assuming that any signal connected to one of its input port can influence any signal connected to one of its output port. From a signal-flow graph perspective, an atomic component connects each of its input signals to each of its output signals. Given a set of signals X we define the set of driving ports $\mathsf{drv}_M(X)$ as follows:

$$\mathsf{drv}_M(X) = \mathsf{in}(\mathsf{block}(\mathsf{src}(X)))$$

For any set of ports Q, let us write $\mathsf{dest}^{-1}(Q) = \{x \mid \mathsf{dest}(x) \in Q\}$. The set of signals upstream of a given set of signals X is denoted $\mathsf{pre}(X)$ and defined by letting $\mathsf{pre}_M(X) = \mathsf{dest}^{-1}(\mathsf{drv}_M(X))$. The cone of influence of some signal variable x is the reflexive-transitive closure of pre_M, that is

$$\mathsf{cone}_M(x) = \mathsf{pre}_M{}^*(\{x\})$$

Computing $\mathsf{cone}_M(x)$ for a given x is done using a standard fix point computation over pre_M. We build sets of signals X_i and Y_i, $i = 1, \ldots, n$ such that $X_0 = \{x\}$, $Y_0 = \emptyset$ and $X_{i+1} = \mathsf{pre}_M(X_i) \setminus Y_{i+1}$, $Y_{i+1} = Y_i \cup X_i$ until $X_n = \emptyset$. Then, $\mathsf{cone}_M(x) = Y_n$.

2.5 Spectrum-based Fault Localization

We now describe how we apply a spectrum-based fault localization procedure to identify the internal states and transitions in the model that are the most likely to be responsible for the fault. We use the Tarantula indicator [12], a well-known technique for statistical fault localization from the software engineering field. This method assumes a number of test cases that are labeled with pass/fail verdicts, the software components that were executed during each test and the source code of the software. Tarantula correlates the activity of software entities (components, methods, statements, branches, etc.) to the pass and fail verdicts and accordingly computes a *score* for each entity that indicates its likelihood to be responsible for the detected fault. Intuitively, components that are more often activated in failed tests are better "culprit" candidates than components that are more often executed in the passed tests.

We present the Tarantula method in a more formal manner as follows. Let $\mathcal{T}$ be a set of tests and $T \in \mathcal{T}$ a single test. Let $\mathsf{oracle} : \mathcal{T} \rightarrow \{\mathsf{pass}, \mathsf{fail}\}$ be the oracle that maps each test to a pass or fail verdict. Let $\mathcal{E}$ be the set of entities and $\mathsf{active} : \mathcal{E} \times \mathcal{T} \rightarrow \mathbb{B}$ a function that maps an entity $e \in \mathcal{E}$ and a test $T \in \mathcal{T}$ to true iff e is active in T. Let $\mathcal{T}_p = \{T \mid T \in \mathcal{T} \text{and } \mathsf{oracle}(T) = \mathsf{pass}\}$ and $\mathcal{T}_f = \mathcal{T} \setminus \mathcal{T}_p$ denote the sets of passed and failed tests, respectively. Let $\mathcal{T}^e = \{T \mid T \in \mathcal{T} \text{and } \mathsf{activity}(T, e)\}$ denote the set of tests with active entity e. We denote by $\mathcal{T}_p^e = \mathcal{T}_p \cap \mathcal{T}^e$ the passed tests with active entity e and $\mathcal{T}_f^e = \mathcal{T}_f \cap \mathcal{T}^e$ the failed tests with active entity e. Finally, we define $\mathsf{passed}(e) = |\mathcal{T}_p^e|$, $\mathsf{failed}(e) = |\mathcal{T}_f^e|$, $\mathsf{total\_passed} = |\mathcal{T}_p|$ and $\mathsf{total\_failed} = |\mathcal{T}_f|$ as the cardinalities of the respective sets. We then define the score function $\mathsf{score}(e)$ that maps each entity to a number as follows:

$$\mathsf{score}(e) = \frac{\frac{\mathsf{failed}(e)}{\mathsf{total\_failed}}}{\frac{\mathsf{passed}(e)}{\mathsf{total\_passed}} + \frac{\mathsf{failed}(e)}{\mathsf{total\_failed}}}$$

We adapt the Tarantula method to our CPS context and dynamic traces. Let us point out that we will use the model slicing technique from the previous section to identify all internal state signals that influence the observable variables declared in the specification. A test hence consists of a simulation trace over time that contains both the observations of the variables from the specification and the state variables given by the slicing. We define our entities as states and transitions observed in the test. The sets $\mathcal{T}_p$ and $\mathcal{T}_f$ of passed and failed tests are generated in two stages. In the first step, we use the monitoring results to make the first test partition according to the satisfaction/violation verdict of the monitor. All the tests that satisfy the specification are put in the set $\mathcal{T}_p$ of passed tests. However, we use the trace diagnostics procedure to further partition the tests that violate the specification into multiple segments. Each segment is marked according to its involvement in the falsification of the specification. The segments in time that have no relation with the detected fault are considered to be passed tests and are added to $\mathcal{T}_p$. The test segments that are responsible to the falsification of the specification, and only those, are added to the set of failed tests $\mathcal{T}_f$. Finally, for each entity (state or segment), we check its activity in a test (or test segment) by traversing the simulation trace and observing whether the entity is present or absent in the test.

3 EVALUATION

We developed a prototype implementation of the procedures described in Section 2. The test generation, test simulation, model slicing and spectrum-based fault localization were developed as MATLAB functions. We used the AMT tool [21] for offline monitoring of STL properties and implemented epoch trace diagnostics presented in Section 2.3 on top of it.

3.1 Automatic Transmission Controller

Model. We use the automatic transmission controller model proposed in [11], and depicted in Figure 6. It is a model of an automatic transmission controller that exhibits both continuous and discrete behavior. The system has two inputs – the throttle u_t and the break u_b. The system has two continuous-time state variables – the speed of the engine ω (rpm), the speed of the vehicle v (mph) and the active gear g_i. The system is initialized with zero vehicle and engine speed. It follows that the output trajectories depend only on the input signals u_t and u_b, which can take any value between 0 and 100 at any point in time.

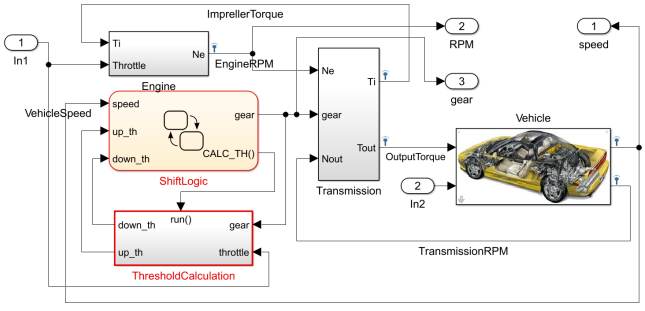


Figure 6: Automatic transmission model.

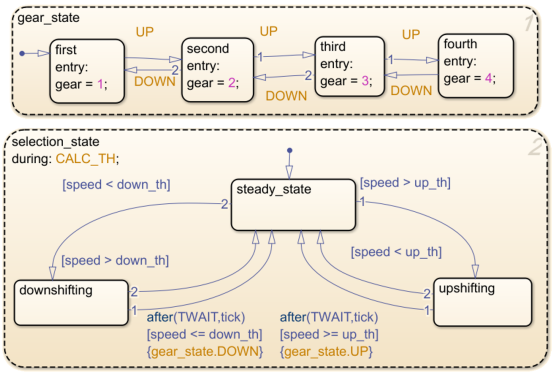


Figure 7: Automatic transmission model – Stateflow chart.

Specification. A requirement of this model is that the vehicle speed v and engine angular speed ω should not exceed maximal values of $\bar{v} = 120$ mph and $\bar{\omega} = 4500$ rpm, respectively. We formalize this requirements as two STL specifications:

$$\varphi_1 \equiv \square(v < \bar{v})$$
$$\varphi_2 \equiv \square(\omega < \bar{\omega})$$

Evaluation Setting. We first generate 100 test cases in which the break signal u_b is set to 0 and the throttle signal u_t is a step function. In each test case, throttle value is initialized to u_t^0 and after some time τ switches to some target value $u_t^\tau > u_t^0$. We obtain different test inputs by varying the values of u_t^0, u_t^τ and τ. We then apply model slicing technique over the signals v and ω and identify the ShiftLogic Stateflow chart and *gear* as the internal state signal that needs to be observed. We simulate all the test cases. Every simulation consists of 3000 time samples of the variables u_t, u_b, v, ω and *gear*.

We next monitor the simulation traces against the specifications φ_1 and φ_2 and apply the epoch diagnostics procedure to the tests that violate either of the two properties. The result of the diagnostics procedure for every test is a list of tuples of the form (var, I), where var is the variable name and I is the interval that is identified as contributing to the property violation. We use this information to further partition the traces that violate the formula into the faulty and non-faulty segments. For every test and every partition, we mark all the active events in a table, and apply the Tarantula formula to compute the fault localization score. We call this diagnostics procedure `mdss-debug` (based on four steps: *monitor*, *diagnose*, *slice*, *score*).

In order to assess our technique, we apply the same procedure, but without doing the trace diagnostics and without partitioning the faulty traces. We call this diagnostics procedure `ms-debug` (based on only two steps: *monitor*, *score*).

Results. We first analyze the system diagnostics results of the procedure `mdss-debug` for the specification φ_1. The outcome of `mdss-debug` for φ_1 is very decisive – the only proposed "culprit" event is the state $gear = 4$. Consider Figure 8, where we collect all the tests that violate φ_1 and plot their speed and gear signals. In addition we show a gray-shaded band that is the convex-hull of all the violation windows reported by the trace diagnostics tool. It is

Table 1: Automatic transmission controller – scores.

	φ_1		φ_2	
Event	mdss-debug	ms-debug	mdss-debug	ms-debug
1	0	0.50	0.16	0.50
2	0	0.50	0.39	0.50
3	0	0.50	0.48	0.50
4	0.51	0.51	0.32	0.51
$1 \longrightarrow 2$	0	0.50	0.16	0.50
$2 \longrightarrow 3$	0	0.50	0.33	0.50
$3 \longrightarrow 4$	0	0.51	0.34	0.51
$3 \longrightarrow 2$	0	0.68	0	0.83
$2 \longrightarrow 1$	0	1	0	1

clear from the figure that in *all* cases the fault happens when the controller is in the gear 4, hence the outcome of mdss-debug turns out to be precise and correct.

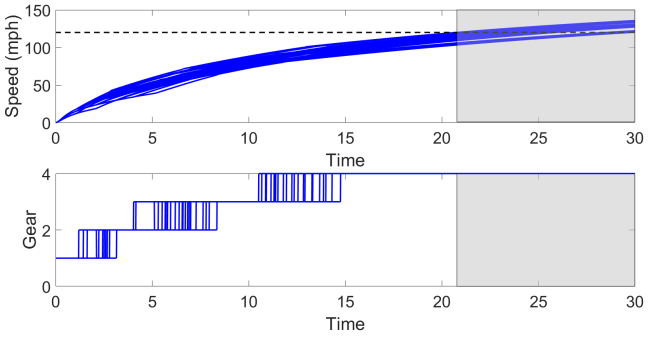


Figure 8: Simulation traces that violate φ_1.

We now analyze the system diagnostics results of the procedure mdss-debug for the specification φ_2. In this case, the results are less decisive. We can see that the gears 2, 3 and 4 are all correlated to the detected fault, as well as the transitions $2 \longrightarrow 3$ and $3 \longrightarrow 4$. If we further combine these results with the analysis of the epoch diagnostics (we depict one trace that violates φ_2 in Figure 9), we see that the violations occur only after the controller stays for sufficiently long time in a gear, and that the transition to the next gear marks the end of the violation. In this example, the computed score is not sufficient on its own to provide a good explanation of the specification violation, but provides a valuable indication that combined with the epoch diagnostics information allows to localize and explain the fault.

We finally note that the procedure ms-debug gives significantly different results from mdss-debug. In particular, ms-debug marks the event $2 \longrightarrow 1$ as the absolute candidate to explain both specification violations with score 1, while mdss-debug marks this same event with score 0. By looking closer at the simulation traces, we find out that the transition from gear 2 to gear 1 happens in a single test case T, depicted in Figure 10. Procedure ms-debug only sees that T fails, and selects $2 \longrightarrow 1$ as one of the potential explanation candidates. Since this event does not appear in any of the passed tests, it is selected as the best possible explanation result. On the

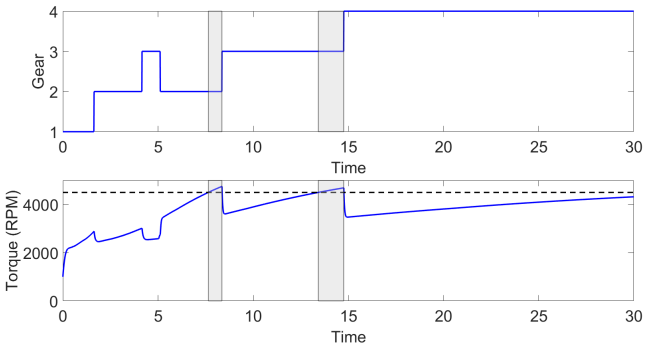


Figure 9: Simulation trace violating φ_2 and its epoch diagnostics.

other hand, the epoch diagnostics procedure used in mdss-debug is able to distinguish that the $2 \longrightarrow 1$ event (marked by the red vertical line in Figure 10) is not related to the actual faults that trigger the violation of φ_1 and φ_2 (marked by the gray bands in Figure 10). Hence, it rightfully excludes this event from the subsequent analysis and gives it the score 0.

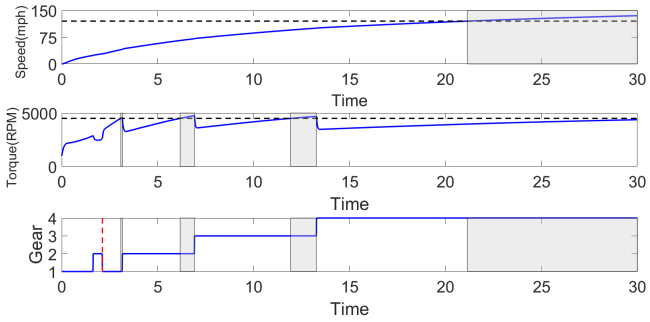


Figure 10: Test featuring the gear transition event $2 \longrightarrow 1$.

Computation Cost. We summarize the computational overhead of our procedure in Table 2. This global overhead is measured to be 2.3 seconds per test, which is an acceptable computational cost. We first observe that the test simulation is relatively efficient, which is explained by the high-level nature of the model. We also see that the monitoring and diagnostics dominate the computation costs. This is mainly due to an invariant overhead of calling an external tool, in addition to processing simulation data.

3.2 Elevator Control System

Model. We consider the Simulink/Stateflow model [20] (see Figure 11) of aircraft's elevators control system. These flight control surfaces are usually attached on the horizontal tails of an aircraft and they are responsible for the control the aircraft's pitch. To assure the safety of the aircraft the system includes several redundant parts in the system as illustrated in Figure 11: (1) four independent hydraulic actuators (two for each elevator); (2) three hydraulic systems for driving the actuators; (3) two Primary Flight Control Units (PFCUs) and (4) two control modules for each actuator – full range

Table 2: Computation cost.

Procedure	Time (ms)
Test generation	23
Model instrumentation	6
Test simulation	250
Monitoring	1400
Diagnostics	550
Slicing	1
Spectrum-based FL	100
Total	2330

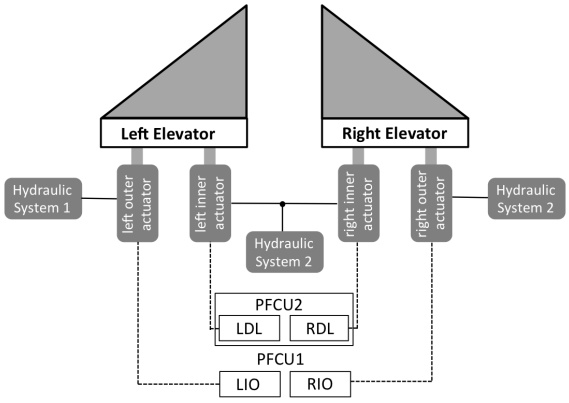


Figure 11: Elevator Control System.

control law and limited range control law. The system takes as an input pilot commands u_p^l and u_p^r that provide the target positions to the left and right surface actuators. These inputs are modeled as square waves. The control system uses the hydraulic actuators to bring the control surfaces to the target position. Each of the actuators has an associated state machine identical to the one in Figure 12. For convenience we assign a numerical identifier to each state that we use in the remainder of the section: 1 – passive, 2 – active, 3 – standby, 4 – off and 5 – isolated. The outputs of the systems are left m^l and and right m^r measured positions.

Specification. This model requires that the control surfaces properly reach the target positions within specified time. In essence, the absolute difference between the pilot command for the left and the right control surfaces and the corresponding measured positions must always be bounded by some d_{max} constant, except when the pilot command changes in which case a delay of at most t_{max} is allowed for the actual position to reach the target one. We formalize this requirement as two STL specifications:

$$\varphi_3 \equiv \Box \Diamond_{[0,t_{max}]}(|u_p^l - m^l| \leq d_{max})$$
$$\varphi_4 \equiv \Box \Diamond_{[0,t_{max}]}(|u_p^r - m^r| \leq d_{max})$$

where $t_{max} = 1$ and $d_{max} = 0.3$.

Given that the properties φ_3 and φ_4 are symmetric, we focus in the remainder of the paper only on the former. In Figure 13, we illustrate the correct behavior of the left wing control surface. We

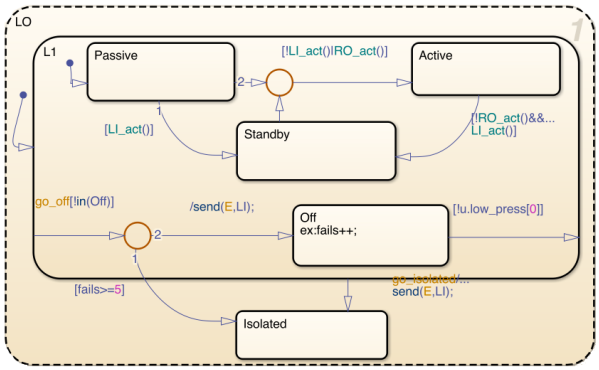


Figure 12: Elevator Control System – Stateflow chart associated to left outer actuator.

can see that the target behavior is followed within a certain amount of time by the actually measured behavior. We can also observe the behavior of the left wing actuators - the left outer actuator is always in the *active* state, while the left inner actuator remains in the *standby* state, implementing a safety redundancy.

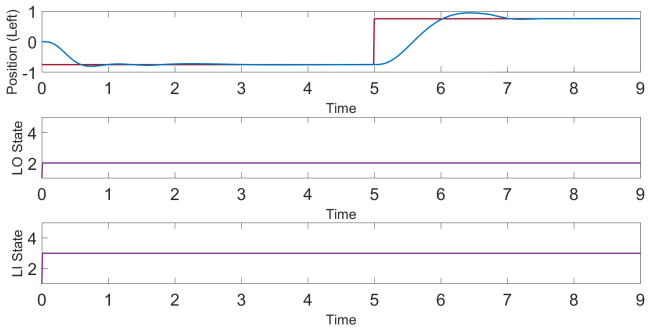


Figure 13: Elevator Control System – ideal behavior.

Evaluation Setting. The elevator control system model comes with the possibility of fault injection. There are two types of faults that can be injected during the model simulation: (1) in the hydraulic system circuits; and (2) in the individual actuators. We create 4 tests with the following properties:

Test 1 no fault injection
Test 2 faults in left and right outer actuators
Test 3 faults in left inner and outer actuators
Test 4 faults in left hydraulic system circuits 1 and 2

Results. The application of the monitoring procedure shows that tests 1 and 2 satisfy, while tests 3 and 4 violate the specification φ_3. Table 3 summarizes the fault localization results. The application of procedure `mdss-debug` clearly indicates that the fault is associated to the fact that both the left inner and the left outer state machines are in the *isolated* state. The procedure `ms-debug` gives a less decisive verdict, giving an equal responsibility likelihood to 2 states and 3 transitions.

We now analyze further these results. Test 1 passes because we do not inject any fault. In Test 2, we inject 2 faults, one in the left

Table 3: Elevator control system – scores.

Event	mdss-debug	ms-debug
$(1,1)$	0	0.50
$(2,3)$	0	0.5
$(4,2)$	0	1
$(5,2)$	0	0.5
$(5,5)$	0.67	1
$(1,1) \rightarrow (2,3)$	0	0.5
$(2,3) \rightarrow (4,2)$	0	1
$(2,3) \rightarrow (5,2)$	0	0.5
$(4,2) \rightarrow (5,5)$	0	1
$(5,2) \rightarrow (5,5)$	0	1

outer and the other in the right outer actuator. The redundancy ensures the correct operation of the system. In Test 3, we inject both faults in the left actuators, thus inhibiting redundancy. Finally, the injection of two faults in the hydraulic system circuits 1 and 2 also affects both left actuators resulting in the failure. The effect of Tests 3 and 4 is that both the left inner and outer state machines go to the *isolated* states which is correctly identified by our approach. The diagnostics results for Test 3 are illustrated in Figure 14.

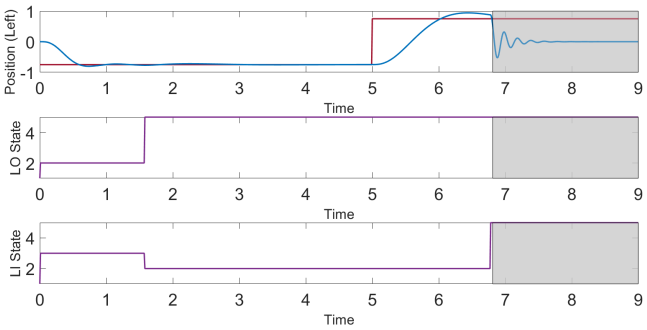


Figure 14: Elevator Control System – incorrect behavior.

The ms-debug procedure only takes into account the pass/fail test verdicts. It is hence not able to properly distinguish between the events that are related to the actual detected fault.

4 RELATED WORK

Since the late 1970s, in the software engineering community there has been a great effort to provide (semi-)automatic techniques assisting the developer to localize and to explain program bugs (we refer the reader to the very comprehensive survey of [26]). Among these techniques, the *spectrum-based fault-localization* (SFL) [1] is a lightweight and well-established statistical technique to measure the code coverage associated with the failed and successful tests. In particular, SFL provides a ranking of the program components that are most likely responsible for the observed fault.

Only very recently, this approach has been applied to also localize faults in Simulink/Stateflow CPS models [14–16], exhibiting a comparable accuracy with the same method applied to software systems [16]. However, the main limitation of this heuristic remains its unpredictability. In fact, it may happen that when applied to a certain test suite, this method assigns the same level of suspiciousness to several components. In such case, one way to improve the precision is to refine the ranking of the suspicious components by automatically generating additional suitable test cases [3, 4, 14, 24].

In this paper, we pursue a new orthogonal and complementary approach to improve fault-localization without the need to generate additional test cases. The classical SFL is agnostic to the nature of the oracle and only requires to know whether the system passes or not a specific test case. SFL does not exploit any additional information about why and when the instrumented system produces an output that is not conformed with respect to a requirement. Here, we assume that our oracle is a monitor generated from an STL requirement. This choice enables us to leverage the trace diagnostic method proposed in [8] and to obtain more information about the failed tests improving the fault-localization.

Other related and complementary approaches can be found in recent papers [6, 10]. The work in [6] investigates the problem of identifying the most important segments in the input of a counterexample that falsifies an STL requirement. Although this method does not provide any information about internal fault-localization, it can be exploited to generate more test cases producing violations and indeed improving even further the precision of our approach.

In [10], the authors address the problem of the controllers synthesis in the model predictive control framework, providing feedback on the reasons of a controller unrealizability whenever the STL specification is infeasible to realize. The particular chosen framework enables to encode the controller synthesis problem as a Mixed Integer Linear Program (MILP). On the contrary, the approach presented here is general and can be applied not only to MPC synthesis problems, but also to other Simulink/Stateflow models.

We are also aware that the problem of fault diagnosis for dynamical systems has been extensively investigated from a different perspective within the control-theory community. In particular, two main families of model-based techniques have emerged: a fault detection system based on output observers [9] and an on-line parameter estimation approach [5] using state-based observers. Our approach is more related to the first one since we monitor the output of the system. Observer-based fault detection systems generally use a class of statistical models called Kalman filters to online estimate *the residual*, a real value representing the difference between the output of the system and the estimated one. In this case modeling errors that are generally difficult to avoid can create unexpected troubles. In our approach (that is offline and simulation-based) we use instead an STL requirement that specifies in a rigorous way the possible set of allowed trajectories of the system, and we exploit this logical framework to refine our search for the cause of a fault.

5 DISCUSSION AND FUTURE WORK

We presented a novel procedure for debugging Simulink/Stateflow models that is guided by STL specifications. The proposed approach uses logic-based reasoning and model slicing to refine spatial and temporal information about detected test violations. This data is then processed by spectrum-based fault localization algorithm to identify the most likely reason for the violation. We demonstrated

the effectiveness of our method on two application examples, from the automotive and avionics domain.

The evaluation of our approach shows promising results. The combination of trace diagnostics with spectrum-based fault localization seems to provide an effective debugging aid to the engineer. The former method enables refining failing tests into smaller, more informative segments that guides the statistical search of the culprit. Nevertheless, application examples reveal possible improvements, and open several directions to pursue as future work.

We first recall that in this paper we identify faults with respect to discrete locations and transitions of the Simulink/Stateflow model. While we are confident that in hybrid systems design faults are typically related to particular locations and mode switches, considering in addition the continuous dynamics can certainly help in identifying the reasons for specification violations.

In our analysis, only basic events (*states* and *transitions*) were considered as candidates to explain faults. We have seen that such analysis alone may not be sufficient. Some applications may require richer structures that take into account duration of being in a state, sequences of events, correlation between the states and some continuous parameters etc. to explain certain faults. In order to tackle this problem, we see a potential need to combine logic-based with machine learning reasoning.

We use static model slicing to identify Simulink/Stateflow entities that can influence the fault. While this static approach is able to discard irrelevant model components and narrow down the search for the right explanation, we also plan to study more dynamic slicing techniques. In particular, dynamic slicing could help by taking into account potential propagation effects in the model and thus further refine our analysis.

ACKNOWLEDGMENTS

This work was partially supported by the Austrian Science Fund (FWF) under grants S11402-N23 and S11405-N23 (RiSE/SHiNE), the CPS/IoT project (HRSM), the EU ICT COST Action IC1402 on Runtime Verification beyond Monitoring (ARVI), the AMASS project (ECSEL 692474), and the ENABLE-S3 project (ECSEL 692455). The CPS/IoT project receives support from the Austrian government through the Federal Ministry of Science, Research and Economy (BMWFW) in the funding program Hochschulraum-Strukturmittel (HRSM) 2016. The ECSEL Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques*. IEEE, 89–98.

[2] Yashwanth Annapureddy, Che Liu, Georgios E. Fainekos, and Sriram Sankaranarayanan. 2011. 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 (LNCS)*, Vol. 6605. Springer, 254–257.

[3] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed Test Generation for Effective Fault Localization. In *International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 49–60.

[4] Benoit Baudry, Franck Fleurey, and Yves Le Traon. 2006. Improving test suites for efficient fault localization. In *International Conference on Software Engineering*. ACM, 82–91.

[5] Laurian Dinca and Tunc Aldemir. 1997. An On-Line Parameter Estimation Scheme for Fault Diagnosis. *IFAC Proceedings Volumes* 30, 18 (1997), 289–294.

[6] Ram Das Diwakaran, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Analyzing neighborhoods of falsifying traces in cyber-physical systems. In *International Conference on Cyber-Physical Systems*. ACM, 109–119.

[7] Alexandre Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *International Conference on Computer Aided Verification (LNCS)*, Vol. 6174. Springer, 167–170.

[8] Thomas Ferrère, Oded Maler, and Dejan Nickovic. 2015. Trace Diagnostics Using Temporal Implicants. In *International Symposium on Automated Technology for Verification and Analysis (LNCS)*, Vol. 9364. Springer, 241–258.

[9] P.M. Frank and X. Ding. 1997. Survey of robust residual generation and evaluation methods in observer-based fault detection systems. *Journal of Process Control* 7, 6 (1997), 403–424.

[10] Shromona Ghosh, Dorsa Sadigh, Pierluigi Nuzzo, Vasumathi Raman, Alexandre Donzé, Alberto L. Sangiovanni-Vincentelli, S. Shankar Sastry, and Sanjit A. Seshia. 2016. Diagnosis and Repair for Synthesis from Signal Temporal Logic Specifications. In *International Conference on Hybrid Systems: Computation and Control*. ACM, 31–40.

[11] Bardh Hoxha, Houssam Abbas, and Georgios E. Fainekos. 2015. Benchmarks for Temporal Logic Requirements for Automotive Systems. In *International Workshop on Applied veRification for Continuous and Hybrid Systems (EPiC Series in Computing)*, Vol. 34. EasyChair, 25–30.

[12] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *International Conference on Automated Software Engineering*. ACM, Long Beach, CA, USA, 273–282.

[13] Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems* 2, 4 (1990), 255–299.

[14] Bing Liu, Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *International Conference on Software Analysis, Evolution and Reengineering*. IEEE Computer Society, 359–370.

[15] Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Localizing Multiple Faults in Simulink Models. In *International Conference on Software Analysis, Evolution, and Reengineering*. IEEE Computer Society, 146–156.

[16] Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Simulink fault localization: an iterative statistical debugging approach. *Softw. Test., Verif. Reliab.* 26, 6 (2016), 431–459.

[17] Oded Maler and Dejan Ničković. 2013. Monitoring properties of analog and mixed-signal circuits. *STTT* 15, 3 (2013), 247–268.

[18] Oded Maler, Dejan Ničković, and Amir Pnueli. 2008. Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday (LNCS)*, Vol. 4800. Springer, 475–505.

[19] MathWorks. 2017. Isolating Problematic Behavior with Model Slicer. https://www.mathworks.com/products/sldesignverifier/features.html#isolating-problematic-behavior-with-model-slicer. (2017).

[20] Pieter J. Mosterman and Jason Ghidella. 2006. Fault Detection Control Logic in an Aircraft Elevator Control System. https://www.mathworks.com/help/stateflow/examples/fault-detection-control-logic-in-an-aircraft-elevator-control-system.html. (2006).

[21] Dejan Ničković and Oded Maler. 2007. AMT: A Property-Based Monitoring Tool for Analog Systems. In *International Conference on Formal Modeling and Analysis of Timed Systems (LNCS)*, Vol. 4763. Springer, 304–319.

[22] Joël Ouaknine, Alexander Rabinovich, and James Worrell. 2009. Time-Bounded Verification. In *International Conference on Concurrency Theory (LNCS)*, Vol. 5710. Springer, 496–510.

[23] Robert Reicherdt and Sabine Glesner. 2012. Slicing MATLAB Simulink models. In *International Conference on Software Engineering*. IEEE Computer Society, 551–561.

[24] Jeremias Rößler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *International Symposium on Software Testing and Analysis*. ACM, 309–319.

[25] Sriram Sankaranarayanan and Georgios E. Fainekos. 2012. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *International Conference on Hybrid Systems: Computation and Control*. ACM, 125–134.

[26] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740.

[27] Aditya Zutshi, Sriram Sankaranarayanan, Jyotirmoy V. Deshmukh, James Kapinski, and Xiaoqing Jin. 2015. Falsification of safety properties for closed loop control systems. In *International Conference on Hybrid Systems: Computation and Control*. ACM, 299–300.