



Automatic Failure Explanation in CPS Models

Ezio Bartocci¹, Niveditha Manjunath^{1,2(✉)}, Leonardo Mariani³,
Cristinel Mateis², and Dejan Ničković²

¹ Vienna University of Technology, Vienna, Austria

² AIT Austrian Institute of Technology, Vienna, Austria
niveditha.manjunath@ait.ac.at

³ University of Milano-Bicocca, Milan, Italy

Abstract. Debugging Cyber-Physical System (CPS) models can be extremely complex. Indeed, only detection of a failure is insufficient to know how to correct a faulty model. Faults can propagate in time and in space producing observable misbehaviours in locations completely different from the location of the fault. Understanding the reason of an observed failure is typically a challenging and laborious task left to the experience and domain knowledge of the designers.

In this paper, we propose CPSDebug, a novel approach that combines testing, specification mining, and failure analysis, to automatically explain failures in Simulink/Stateflow models. We evaluate CPSDebug on two case studies, involving two use scenarios and several classes of faults, demonstrating the potential value of our approach.

1 Introduction

Cyber-Physical Systems (CPS) combine computational and physical entities that interact with sophisticated and unpredictable environments via sensors and actuators. To cost-efficiently study their behavior, engineers typically apply model-based development methodologies, which combine modeling and simulation activities with prototyping. The successful development of CPS is thus strongly dependent on the quality and correctness of their models.

CPS models can be extremely complex: they may include hundreds of variables, signals, look-up tables and components, combining continuous and discrete dynamics. Verification and testing activities are thus of critical importance to early detect problems in the models [2, 5, 7, 16, 17], before they propagate to the actual CPS. Discovering faults is however only a part of the problem. Due to their complexity, debugging CPS models by identifying the causes of failures can be as challenging as identifying problems themselves [15].

CPS functionalities are often modelled using the MathWorksTM Simulink environment where *falsification-based testing* can be used to find bugs in Simulink/Stateflow models [2, 22, 25]. This approach is based on quantifying (by monitoring [4]) how much a simulated trace of CPS behavior is close to violate a requirement expressed in a formal specification language, such as Signal

Temporal Logic (STL) [20]. This measure enables a systematic exploration of the input space searching for the first input sequence responsible for a violation. However, this method does not provide any suitable information about which components should be inspected to resolve the violation. Trace diagnostics [10] identifies (small) segments of the observable model behavior that are sufficient to imply the violation of the formula, thus providing a failure explanation at the input/output model interface level. However, this is a black-box technique that does not attempt to delve into the model and explain the failure in terms of its internal signals and components.

In this paper, we advance the knowledge in failure analysis of CPS models by presenting CPSDebug, a debugging technique that combines testing, specification mining, and failure analysis to identify the causes of failures. CPSDebug first exercises the CPS model under analysis by running the available test cases, while discriminating passing and failing executions using requirements formalized as a set of STL formulas. While running the test cases, CPSDebug records information about the internal behavior of the CPS model. In particular, it collects the values of all internal system variables at every timestamp. The values collected from passing test cases are used to infer properties about the variables and components involved in the computations. These properties capture the correct behavior of the system.

CPSDebug checks the mined properties against the traces collected from failed test cases to discover the internal variables, and its corresponding components, that are responsible for the violation of the requirements. Finally, failure evidence is analyzed using trace diagnostics [10] and clustering [12] to produce a time-ordered sequence of snapshots that show where the anomalous variables values originated and how they propagated within the system.

CPSDebug thus overcomes the limitation of state of the art approaches that do not guide engineers in the analysis, but only indicate the inputs or code locations that might be responsible for the failure. The sequence of snapshots returned by CPSDebug provides a step by step illustration of the failure with explicit indication of the faulty behaviors. We evaluated CPSDebug against three classes of faults and two actual CPS models. Results suggest that CPSDebug can effectively and efficiently assist developers in their debugging tasks. The feedback that we collected from industry engineers further confirmed that the output produced by CPSDebug can be indeed valuable to ease failure analysis and debugging of CPS models.

The rest of the paper is organized as follows. We provide background information in Sect. 2 and we describe the case study in Sect. 3. In Sect. 4 we present our approach for failure explanation while in Sect. 5 we provide the empirical evaluation. We discuss the related work in Sect. 6 and we draw our conclusions in Sect. 7.

2 Background

2.1 Signals and Signal Temporal Logic

We define $S = \{s_1, \dots, s_n\}$ to be a set of signal variables. A *signal* or *trace* w is a function $\mathbb{T} \rightarrow \mathbb{R}^n$, where \mathbb{T} is the time domain in the form of $[0, d] \subset \mathbb{R}$. We can also see a multi-dimensional signal w as a vector of real-valued uni-dimensional signals $w_i : \mathbb{T} \rightarrow \mathbb{R}$ associated with variables s_i for $i = 1, \dots, n$. We assume that every signal w_i is piecewise-linear. Given two signals $u : \mathbb{T} \rightarrow \mathbb{R}^l$ and $v : \mathbb{T} \rightarrow \mathbb{R}^m$, we define their parallel composition $u \parallel v : \mathbb{T} \rightarrow \mathbb{R}^{l+m}$ in the expected way. Given a signal $w : \mathbb{T} \rightarrow \mathbb{R}^n$ defined over the set of variables S and a subset of variables $R \subseteq S$, we denote by w_R the projection of w to R , where $w_R = \parallel_{s_i \in R} w_i$.

Let Θ be a set of terms of the form $f(R)$ where $R \subseteq S$ are subsets of variables and $f : \mathbb{R}^{|R|} \rightarrow \mathbb{R}$ are interpreted functions. The syntax of STL with both *future* and *past* operators is defined by the grammar:

$$\varphi ::= \top \mid f(R) > 0 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathcal{U}_I \varphi_2 \mid \varphi_1 \mathcal{S}_I \varphi_2,$$

where $f(R)$ are terms in Θ and I are real intervals with bounds in $\mathbb{Q}_{\geq 0} \cup \{\infty\}$. As customary, we use the shorthands $\diamond_I \varphi \equiv \top \mathcal{U}_I \varphi$ for *eventually*, $\square_I \varphi \equiv \neg \diamond_I \neg \varphi$ for *always*, $\heartsuit_I \varphi \equiv \top \mathcal{S}_I \varphi$ for *once*, $\boxminus_I \varphi \equiv \neg \heartsuit_I \neg \varphi$ for *historically*, $\uparrow \varphi \equiv \varphi \wedge \top \mathcal{S} \neg \varphi$ for *rising edge* and $\downarrow \varphi \equiv \neg \varphi \wedge \top \mathcal{S} \varphi$ for *falling edge*¹. We interpret STL with its classical semantics defined in [19].

2.2 Daikon

Daikon is a template-based property inference tool that, starting from a set of variables and a set of observations, can infer a set of properties that are likely to hold for the input variables [9]. More formally, given a set of variables $V = V_1, \dots, V_n$ defined over the domains D_1, \dots, D_n , an observation for these variables is a tuple $\bar{v} = (v_1, \dots, v_n)$, with $v_i \in D_i$.

Given a set of variables V and multiple observations $\bar{v}_1 \dots \bar{v}_m$ for these same variables, Daikon is a function $D(V, \bar{v}_1 \dots \bar{v}_m)$ that returns a set of properties $\{p_1, \dots, p_k\}$, such that $\bar{v}_i \models p_j \forall i, j$, that is, all the observations satisfy the inferred properties. For example, considering two variables x and y and considering the observations $(1, 3)$, $(2, 2)$, $(4, 0)$ for the tuple (x, y) , Daikon can infer properties such as $x > 0$, $x + y = 4$, and $y \geq 0$.

The inference of the properties is driven by a set of template operators that Daikon instantiates over the input variables and checks against the input data. Since template-based inference can generate redundant and implied properties, Daikon automatically detects them and reports the relevant properties only. Finally, to guarantee that the inferred properties are relevant, Daikon computes the probability that the inferred property holds by chance for all the properties. Only properties that are statistically significant with a probability higher than 0.99 are assumed to be reliable and are reported in the output.

¹ We omit the timing modality I when $I = [0, \infty)$.

In our approach, we use Daikon to automatically generate properties that capture the behavior of the individual components and individual signals in the model under analysis. These properties can be used to precisely detect misbehaviours and their propagation.

3 Case Study

We now introduce a case study that we use as a running example to illustrate our approach step by step. We consider the Aircraft Elevator Control System (AECS) introduced in [11] to illustrate model-based development of a Fault Detection, Isolation and Recovery (FDIR) application for a redundant actuator control system.

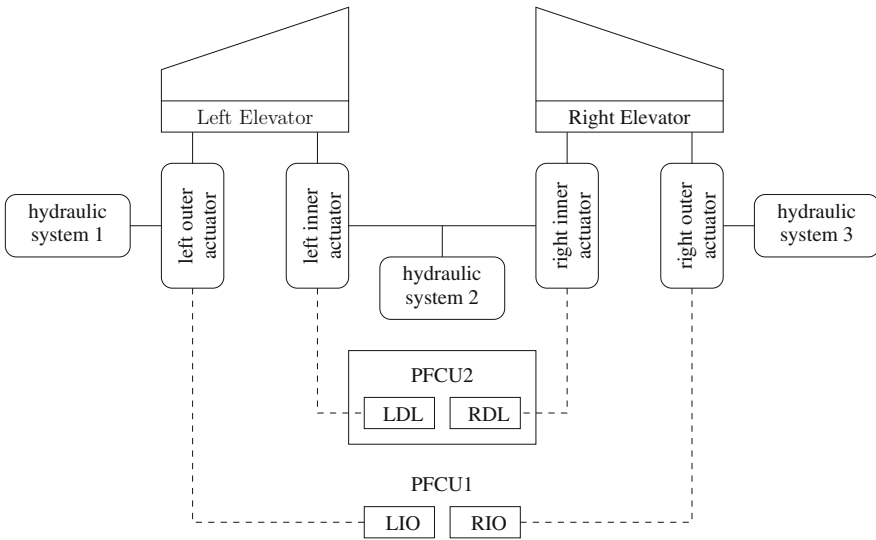


Fig. 1. Aircraft elevator control system [11].

Figure 1 shows the architecture of an aircraft elevator control system with redundancy, with one elevator on the left and one on the right side. Each elevator is equipped with two hydraulic actuators. Both actuators can position the elevator, but only one shall be active at any point in time. There are three different hydraulic systems that drive the four actuators. The left (LIO) and right (RIO) outer actuators are controlled by a Primary Flight Control Unit (PFCU1) with a sophisticated input/output control law. If a failure occurs, a less sophisticated Direct-Link (DL/PFCU2) control law with reduced functionality takes over to handle the left (LDL) and right (RDL) inner actuators. The system uses state machines to coordinate the redundancy and assure its continual fail-operational activity.

This model has one input variable, the input Pilot Command, and two output variables, the position of left and right actuators, as measured by the sensors.

This is a complex model that could be extremely hard to analyze in case of failure. In fact, the model has 426 signals, from which 361 are internal variables that are instrumented (279 real-valued, 62 Boolean and 20 enumerated - state machine - variables) and any of them, or even a combination of them, might be responsible for an observed failure.

The model comes with a failure injection mechanism, which allows to dynamically insert failures that represent hardware/ageing problems into different components of the system during its simulation. This mechanism allows insertion of (1) low pressure failures for each of the three hydraulic systems, and (2) failures of sensor position components in each of the four actuators. Due to the use of redundancy in the design of the control system, a single failure is not sufficient to alter its intended behavior. In some cases even two failures are not sufficient to produce faulty behaviors. For instance, the control system is able to correctly function when both a left and a right sensor position components simultaneously fail. This challenges the understanding of failures because there are multiple causes that must be identified to explain a single failure.

To present our approach we consider the analysis of a system failure caused by the activation of two failures: the sensor measuring Left Outer Actuator Position failing at time 2 and the sensor measuring Left Inner Actuator Position failing at time 4. To collect evidence of how the system behaves, we executed the Simulink model with 150 test cases with different pilot commands and collected the input-output behavior both with and without the failures.

When the system behaves correctly, the intended position of the aircraft required by the pilot must be achieved within a predetermined time limit and with a certain accuracy. This can be captured with several requirements. One of them says that whenever Pilot Command cmd goes above a threshold m , the actuator position measured by the sensor must stabilize (become at most n units away from the command signal) within $T + t$ time units. This requirement is formalized in STL with the following specification:

$$\varphi \equiv \square(\uparrow (cmd \geq m) \rightarrow \diamond_{[0,T]} \square_{[0,t]}(|cmd - pos| \leq n)). \quad (1)$$

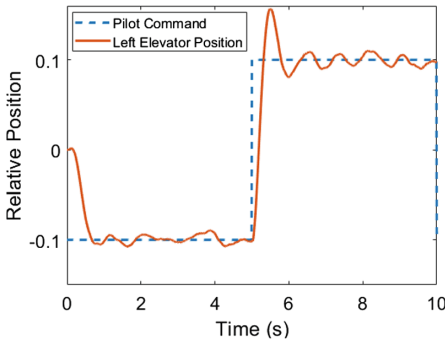


Fig. 2. Expected behavior of AECS.

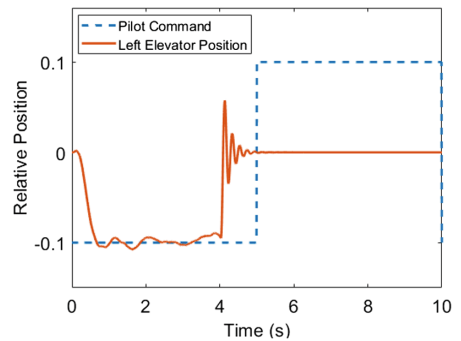


Fig. 3. Failure of the AECS.

Figures 2 and 3 shows the correct and faulty behavior of the system. The control system clearly stops following the reference signal after 4 seconds. The failure observed on the input/output interface of the model does not give any indication within the model on the reason leading to the property violation. In the next section, we present how our failure explanation technique can address this case producing a valuable output for engineers.

4 Failure Explanation

In this section we describe how CPSDebug works with help of the case study introduced in Sect. 3. Figure 4 illustrates the main steps of the workflow. Briefly, the workflow starts from a target CPS model and a test suite with some passing and failing test cases, and produces a failure explanation for each failing test case. The workflow consists of three sequential phases:

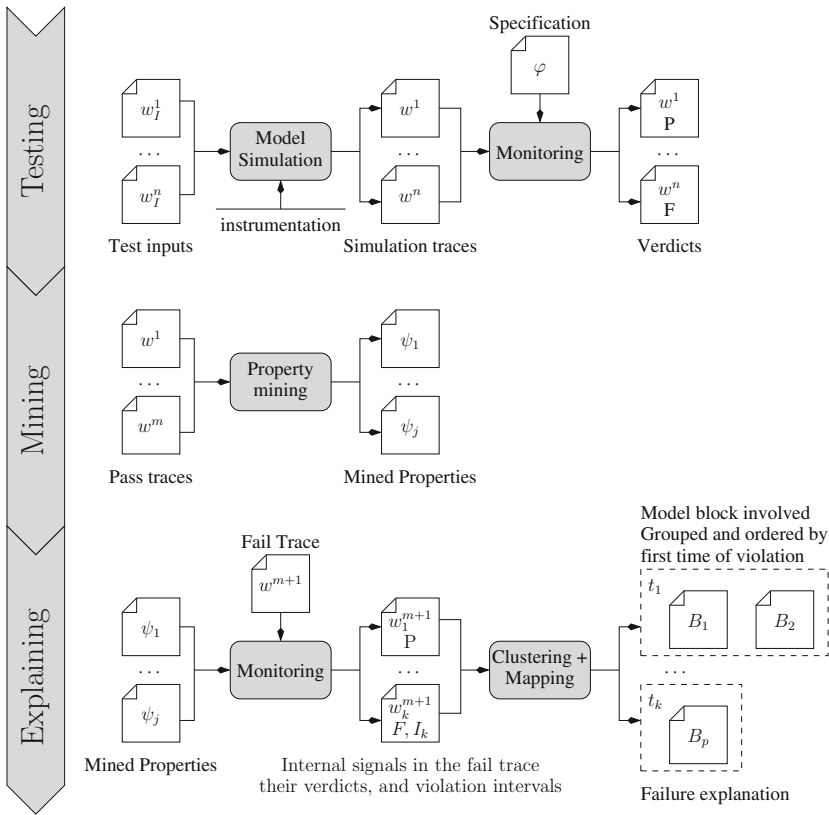


Fig. 4. Overview of the failure explanation procedure.

- (i) *Testing*, which simulates the instrumented CPS model with the available test cases to collect information about its behavior, both for passing and failing executions,
- (ii) *Mining*, which mines properties from the traces produced by passing test cases; intuitively these properties capture the expected behavior of the model,
- (iii) *Explaining*, which uses mined properties to analyze the traces produced by failures and generate failure explanations, including information about the root events responsible for the failure and their propagation.

4.1 Testing

CPSDebug starts by instrumenting the CPS model. This is an important pre-processing step that is done before testing the model and that allows to log the internal signals in the model. Model *instrumentation* is inductively defined on the hierarchical structure of the Simulink/Stateflow model and is performed in a bottom-up fashion. For every signal variable having the real, Boolean or enumeration type, CPSDebug assigns a unique name to it and makes the simulation engine to log its values. Similarly, CPSDebug instruments look-up tables and state machines. Each look-up table is associated with a dedicated variable which is used to produce a simulation trace that reports the unique cell index that is exercised by the input at every point in time. CPSDebug also instruments state-machines by associating two dedicated variables per state-machine, one reporting the transitions taken and one reporting the locations visited during the simulation. We denote by V the set of all instrumented model variables.

The first step of the testing phase, namely *Model Simulation*, runs the available *test cases* $\{w_J^k | 1 \leq k \leq n\}$ against the *instrumented* version of the simulation model under analysis. The number of available test cases may vary case by case, for instance in our case study the test suite included $n = 150$ tests.

The result of the model simulation consists of one simulation trace w^k for each test case w_J^k . The trace w^k stores the sequence of (simulation time, value) pairs w_v^k for every instrumented variable $v \in V$ collected during simulation.

To determine the nature of each trace, we transform the informal model *specification*, which is typically provided in form of free text, into an STL formula φ that can be automatically evaluated by a *monitor*. In fact, CPSDebug checks every trace w^k against the STL formula φ , $1 \leq k \leq n$ and labels the trace with a *pass verdict* if w^k *satisfies* φ , or a *fail verdict* otherwise. In our case study, the STL formula 1 in Sect. 3 labeled 149 traces as passing and 1 trace as failing.

4.2 Mining

In the mining phase, CPSDebug selects the traces labeled with a pass verdict and exploits them for *property mining*.

Prior to the property inference, CPSDebug performs several intermediate steps that facilitate the mining task. First, CPSDebug reduces the set of variables V to its subset \hat{V} of *significant* variables by using cross-correlation. Intuitively,

the presence of two highly correlated variables implies that one variable adds little information on top of the other one, and thus the analysis may actually focus on one variable only. The approach initializes $\hat{V} = V$ and then checks the cross-correlation coefficient between all the logged variables computed on the data obtained from the pass traces. The cross-correlation coefficient $P(v_1, v_2)$ between two variables v_1 and v_2 is computed with the Pearson method, i.e. $P(v_1, v_2) = \frac{cov(v_1, v_2)}{\sigma_{v_1} \sigma_{v_2}}$ which is defined in terms of the *covariance* of v_1 and v_2 and their standard deviations. Whenever the cross-correlation coefficient between two variables is higher than 0.99, that is $P(v_1, v_2) > 0.99$, CPSDebug removes one of the two variables (and its associated traces) from further analysis, that is, $\hat{V} = \hat{V} \setminus v_1$. In our case study, $|V| = 361$ and $|\hat{V}| = 121$, resulting in a reduction of 240 variables.

In the next step, CPSDebug associates each variable $v \in \hat{V}$ to (1) its domain D and (2) its parent Simulink-block B . We denote by $V_{D,B} \subseteq \hat{V}$ the set $\{v_1, \dots, v_n\}$ of variables with the domain D associated with block B . CPSDebug collects all observations $\bar{v}_1 \dots \bar{v}_n$ from all samples in all traces associated with variables in $V_{D,B}$ and uses the Daikon function $D(V_{D,B}, \bar{v}_1 \dots \bar{v}_n)$ to infer a set of properties $\{p_1, \dots, p_k\}$ related to the block B and the domain D . Running property mining per model block and model domain allows to avoid (1) combinatorial explosion of learned properties and (2) learning properties between incompatible domains.

Finally, CPSDebug collects all the learned properties from all the blocks and the domains, and translates them to an STL specification, where each Daikon property p is transformed to an STL assertion of type $\Box p$.

In our case study, Daikon returned 96 behavioral properties involving 121 variables, hence CPSDebug generated an STL property ψ with 96 temporal assertions, i.e., $\psi = [\psi_1 \psi_2 \dots \psi_{96}]$. Equations 2 and 3 shows two examples of behavioral properties inferred from our case study by Daikon and translated to STL. Variables *mode*, *LI_pos_fail* and *LO_pos_fail* denote internal signals Mode, Left Inner Position Failure and Left Outer Position Failure from the aircraft position control Simulink model. The first property states that the Mode signal is always in the state 2 (Passive) or 3 (Standby), while the second property states that the Left Inner Position Failure is encoded the same than the Left Outer Position Failure.

$$\varphi_1 \equiv \Box(\text{mode} \in \{2, 3\}) \quad (2)$$

$$\varphi_2 \equiv \Box(\text{LI_pos_fail} == \text{LO_pos_fail}) \quad (3)$$

4.3 Explaining

This phase analyzes a trace w collected from a failing execution and produces a failure explanation. The *Monitoring* step analyzes the trace against the mined properties and returns the signals that violate the properties and the time intervals in which the properties are violated. CPSDebug subsequently labels with F

(*fail*) the internal signals involved in the violated properties and with P (*pass*) the remaining signals from the trace. To each fail-annotated signal, CPSDebug also assigns the violation time intervals of the corresponding violated properties returned by the monitoring tool.

In our case study, the analysis of the left inner and the left outer sensor failure resulted in the violation of 17 mined properties involving 19 internal signals.

For each internal signal there can be several fail-annotated signal instances, each one with a different violation time interval. CPSDebug selects the instance that occurs first in time, ignoring all other instances. This is because, to reach the root cause of a failure, CPSDebug has to focus on the events that cause observable misbehaviours first.

Table 1 summarizes the set of property-violating signals, the block they belong to, and the instant of time the signal has first violated a property for our case study. We can observe that the 17 signals participating in the violation of at least one mined property belong to only 5 different Simulink blocks. In addition, we can see that all the violations naturally cluster around two time instants – 2 s and 4 s. This suggests that CPSDebug can effectively *isolate in space and time* a limited number of events likely responsible for the failure.

Table 1. Internal signals that violate at least one learned invariant and Simulink blocks to which they belong. The column $\tau(s)$ denotes the first time that each signal participates in an invariant violation.

Index	Signal name	Block	$\tau(s)$
s252	LI_pos_fail:1→Switch:2	Meas. Left In. Act. Pos.	1.99
s253	Outlier/failure:1→Switch:1	Meas. Left In. Act. Pos.	1.99
s254	Measured Position3:1→Mux:3	Meas. Left In. Act. Pos.	1.99
s255	Measured Position2:1→Mux:2	Meas. Left In. Act. Pos.	1.99
s256	Measured Position1:1→Mux:1	Meas. Left In. Act. Pos.	1.99
s55	BusSelector:2→Mux1:2	Controller	2.03
s328	In2:1→Mux1:2	L_pos_failures	2.03
s329	In1:1→Mux1:1	L_pos_failures	2.03
s332	Right Outer Pos. Mon.:2→R_pos_failures:1	Actuator Positions	2.03
s333	Right Inner Pos. Mon.:2→R_pos_failures:2	Actuator Positions	2.03
s334	Left Outer Pos. Mon.:2→L_pos_failures:1	Actuator Positions	2.03
s335	Right Inner Pos. Mon.:3→Goto3:1	Actuator Positions	2.03
s338	Left Outer Pos. Mon.:3→Goto:1	Actuator Positions	2.03
s341	Left Inner Pos. Mon.:2→L_pos_failures:2	Actuator Positions	2.03
s272	LO_pos_fail:1→Switch:2	Meas. Left Out. Act. Pos.	3.99
s273	Outlier/failure:1→Switch:1	Meas. Left Out. Act. Pos.	3.99
s275	Measured Position1:1→Mux:1	Meas. Left Out. Act. Pos.	3.99
s276	Measured Position2:1→Mux:2	Meas. Left Out. Act. Pos.	3.99
s277	Measured Position3:1→Mux:3	Meas. Left Out. Act. Pos.	4.00

The *Clustering & Mapping* step then (1) clusters the resulting fail-annotated signal instances by their violation time intervals and (2) maps them to the corresponding model blocks, i.e., to the model blocks that have some of the fail-annotated signal instances as internal signals.

Finally, CPSDebug generates failure explanations that capture how the fault originated and propagated in space and time. In particular, the failure explanation is a sequence of snapshots of the system, one for each cluster of property violations. Each snapshot reports (1) the mean time as approximative time when the violations represented in the cluster occurred, (2) the model blocks $\{B_1, \dots, B_p\}$ that originate the violations reported in the cluster, (3) the properties violated by the cluster, representing the reason why the cluster of anomalies exist, and (4) the internal signals that participate to the violations of the properties associated with the cluster. Intuitively a snapshot represents a new relevant state of the system, and the sequence shows how the execution progresses from the violation of set of properties to the final violation of the specification. The engineer is supposed to exploit the sequence of snapshots to understand the failure, and the first snapshot to localize the root cause of the problem. Figure 5 shows the first snapshot of the failure explanation that CPSDebug generated for the case study. We can see that the explanation of the failure at time 2 involves the Sensors block, and propagates to Signal conditioning and failures and failures and Controller

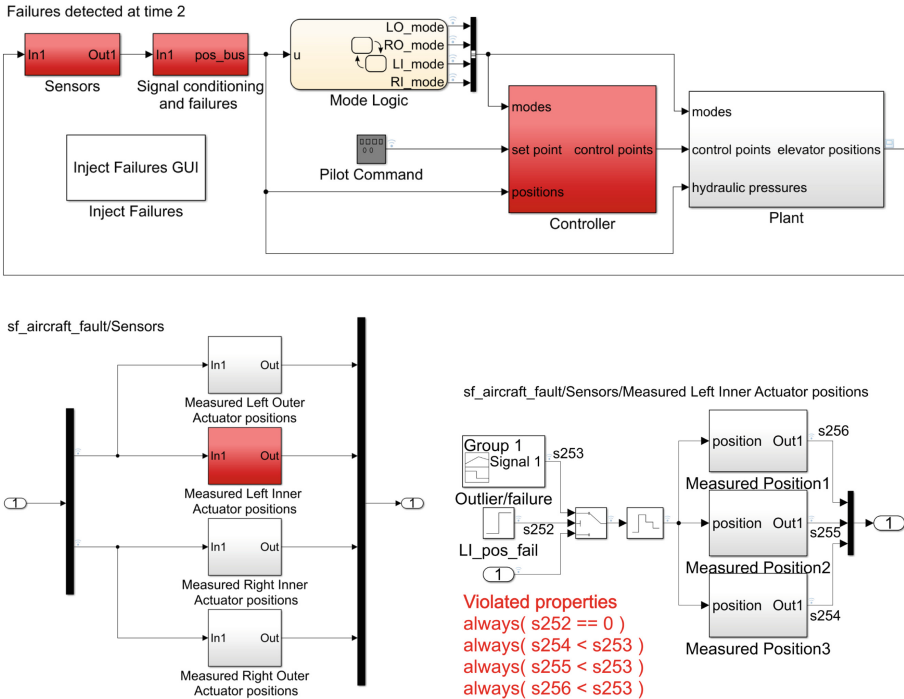


Fig. 5. Failure explanation as a sequence of snapshots - part of the first snapshot.

blocks. By opening the Sensors block, we can immediately see that something is wrong with the sensor that measures the left inner position of the actuator. Going one level below, we can see that the signal s_{252} produced by *LI_pos_fail* is suspicious – indeed the fault was injected exactly in that block at time 2. It is not a surprise that the malfunctioning of the sensor measuring the left inner position of the actuator affects the Signal conditioning and failures block (the block that detects if there is a sensor that fails) and the Controller block. However, at time 2 the failure in one sensor does not affect yet the correctness of the overall system, hence the STL specification is not yet violated. The second snapshot (not shown here) generated by CPSDebug reveals that the sensor measuring the left outer position of the actuator fails at time 4. The redundancy mechanism is not able to cope with multiple sensor faults, hence anomalies manifest in the observable behavior. From this sequence of snapshots, the engineer can conclude that the problem is in the failure of the two sensors - one measuring the left inner and the other measuring the left outer position of the actuator that stop functioning at times 2 and 4, respectively.

5 Empirical Evaluation

We empirically evaluated our approach against three classes of faults: *multiple hardware faults in fault-tolerant systems*, which is the case of multiple components that incrementally fail in a system designed to tolerate multiple malfunctioning units; *incorrect look-up tables*, which is the case of look-up tables containing incorrect values; and *erroneous guard conditions*, which is the case of imprecise conditions in the transitions that determine the state-based behavior of the system. Note that these classes of faults are highly heterogenous. In fact, their analysis requires a technique flexible enough to deal with multiple failure causes, but also with the internal structure of complex data structures and finally with state-based models.

We consider two different systems to introduce faults belonging to these three classes. We use the fault-tolerant aircraft elevator control system [11] presented in Sect. 3 to study the capability of our approach to identify failures caused by multiple overlapping faults. In particular, we study cases obtained by (1) injecting a low pressure fault into two out of three hydraulic components (fault h_1h_2), and (2) inserting a fault in the left inner and left outer sensor position components (fault *lilo*).

We use the automatic transmission control system [13] to study the other classes of faults. Automatic transmission control system is composed of 51 variables, includes 4 look-up tables of size between 4 and 110 and two finite state machines running in parallel with 3 and 4 states, respectively, as well as 6 transitions each. We used the 7 STL specifications defined in [13] to reveal failures in this system. We studied cases obtained by (1) modifying a transition guard in the StateFlow chart (fault *guard*), and (2) altering an entry in the look-up table Engine (fault *eng.lt*).

To study these faults, we considered two use scenarios. For the aircraft elevator control system, we executed 150 test cases in which we systematically changed

the amplitude and the frequency of the pilot command steps. These tests were executed on a non-faulty model. We then executed an additional test on the model to which we dynamically injected h_1h_2 and *lilo* faults. For the automatic transmission control system, we executed 100 tests in which we systematically changed the step input of the throttle by varying the amplitude, the offset and the absolute time of the step. All the tests were executed on a faulty model. In both cases, we divided the failed tests from the passing tests. CPSDebug used the data collected from the passing tests to infer models necessary for the analysis of the failed tests.

We evaluated the output produced by our approach considering four main aspects: Scope Reduction, Cause Detection, Quality of the Analysis, and Computation Time. Scope Reduction measures how well our approach narrows down the number of elements to be inspected to a small number of anomalous signals that require the attention of the engineer, in comparison to the set of variables involved in the failed execution. Cause detection indicates if the first cluster of anomalous values reported by our approach includes any property violation caused by the signal that is directly affected by the fault. Intuitively, it would be highly desirable that the first cluster of anomalies reported by our technique includes violations caused by the root cause of the failure. For instance, if a fault directly affects the values of the signal `Right Inner Pos.`, we expect these values to cause a violation of a property about this same signal. We qualitatively discuss the set of violated properties reported for the various faults and explain why they offer a comprehensive view about the problem that caused the failure. Finally, we analyze the computation time of CPSDebug and its components and compare it to the simulation time of the model.

To further confirm the effectiveness of our approach, we contacted 3 engineers from (1) an automotive OEM with over 300.000 employees (*E1*), (2) a major modeling and simulation tool vendor with more than 3.000 employees (*E2*), and (3) an SME that develops tools for verification and testing of CPS models (*E3*). We asked them to evaluate the outcomes of our tool for a selection of faults (it was infeasible to ask them to inspect all the results we collected). In particular, we sent them the faulty program, an explanation of both the program and the fault, and the output generated by our tool, and we asked them to answer the following questions:

- Q1 How helpful is the output to understand the cause(s) of the failure? (Very useful/Somewhat useful/Useless/Misleading)
- Q2 Would you consider experimenting our tool with your projects? (Yes/Maybe/No)
- Q3 Considering the sets of violations that have been reported, is there anything that should be removed from the output? (open question)
- Q4 Is there anything more you would like to see in the output produced by our tool? (open question)

In the following, we report the results that we obtained for each of the analyzed aspects.

5.1 Scope Reduction, Cause Detection and Qualitative Analysis

Table 2 shows the degree of reduction achieved for the analyzed faults. Column *system* indicates the faulty application used in the evaluation. Column *# vars* indicates the size of the model in terms of the number of its variables. Column *fault* indicates the specific fault analyzed. Column *# ψ* gives the number of learned invariants. Column *# suspicious vars (reduction)* indicates the number of variables involved in the violated properties and the reduction achieved. Column *fault detected* indicates whether the explanation included a variable associated with the output of the block in which the fault was injected.

Table 2. Scope reduction and cause detection.

system	# vars	fault	# ψ	# suspicious vars	fault detected
Aircraft	426	<i>lilo</i>	96	17(96%)	✓
		<i>h₁h₂</i>	96	44(90%)	✓
Transmission	51	<i>guard</i>	41	1(98%)	
		<i>eng_lt</i>	39	4(92%)	✓

We can see from Table 2 that CPSDebug successfully detected the exact origin of the fault in 3 out of 4 cases. In the case of the aircraft elevator control system, CPSDebug clearly identifies the problem with the respective sensors (fault *lilo*) and hydraulic components (fault *h₁h₂*). Overall, the scope reduction ranged from 90% to 98% of the model signals, allowing engineers to focus on a small subset of the suspicious signals. Note that a strong scope reduction is useful also when CPSDebug is not effective with a fault, since engineers could quickly conclude that the fault is not in the (few) recommended locations without wasting their time (such as for the *guard* fault).

In the case of the automatic transmission control, CPSDebug associates the misbehavior of the model with the Engine look-up table and points to its right entry. The scope reduction in this case is 90%. On the other hand, CPSDebug misses the exact origin of the *guard* fault and fails to point to the altered transition. This happens because the faulty guard alters only the *timing* but not the *qualitative* behavior of the state machine. Since Daikon is able to learn only invariant properties, CPSDebug is not able to discriminate between passing and failing tests in that case. Nevertheless, CPSDebug does associate the entire state machine to the anomalous behavior, since the observable signal that violates the STL specification is generated by the state machine.

5.2 Computation Time

Table 3 summarizes computation time of CPSDebug applied to the two case studies. We can make two main conclusions from these experimental results: (1) the overall computation time of CPSDebug-specific activities is comparable to

the overall simulation time and (2) property mining dominates by far the computation of the explanation. We finally report in the last row the translation of the Simulink simulation traces recorded in the Common Separated Values (csv) format to the specific input format that is used by Daikon. In our prototype implementation of CPSDebug, we use an inefficient format translation that results in excessive time. We believe that investing an additional effort can result in improving the translation time by several orders of magnitude.

Table 3. CPSDebug computation time.

	Aircraft	Transmission
# tests	150	100
# samples per test	1001	751
	Time (s)	
Simulation	654	35
Instrumentation	1	0.7
Mining	501	52
Monitoring properties	0.7	0.6
Analysis	1.5	1.6
File format translation	2063	150

5.3 Evaluation by Professional Engineers

We analyze in this section the feedback provided by engineers *E1–E3* to the questions *Q1–Q4*.

- Q1 *E1* found CPSDebug potentially very useful. *E2* and *E3* found CPSDebug somewhat useful.
- Q2 All engineers said that they would experiment with CPSDebug.
- Q3 None of the engineers found anything that should be removed from the tool outcome.
- Q4 *E2* and *E3* wished to see better visual highlighting of suspicious signals. *E2* wished to see the actual trace for each suspicious signal. *E2* and *E3* wished a clearer presentation of cause-effect relations.

Apart from the direct responses to *Q1–4*, we received other useful information. All engineers shared appreciation for the visual presentation of outcomes, and especially the marking of suspicious Simulink blocks in red. *E1* highlighted that real production models typically do not only contain Simulink and State-Flow blocks, but also SimEvent and SimScape blocks, Bus Objects, Model Reference, Variant Subsystems, etc., which may limit the applicability of the current prototype implementation.

Overall, engineers confirmed that CPSDebug can be a useful technology. At the same time, they offered valuable feedback to improve it, especially the presentation of the output produced by the tool.

6 Related Work

The analysis of software failures has been addressed with two main classes of related approaches: fault localization and failure explanation techniques.

Fault localization techniques aim at identifying the location of the faults that caused one or more observed failures (an extensive survey can be found in [27]). A popular example is *spectrum-based fault-localization* (SBFL) [1], an efficient statistical technique that, by measuring the code coverage in the failed and successful tests, can rank the program components (e.g., the statements) that are most likely responsible for a fault.

SBFL has been recently employed to localize faults in Simulink/Stateflow CPS models [5, 7, 16–18], showing similar accuracy as in the application to software systems [18]. The explanatory power of this approach is however limited, because it generates neither information that can help the engineers understanding if a selected code location is really faulty nor information about how a fault propagated across components resulting on an actual failure. Furthermore, SBFL is agnostic to the nature of the oracle requiring to know only whether the system passes or not a specific test case. This prevents the exploitation of any additional information concerning why and when the oracle decides that the test is not conformed with respect to the desired behavior. In Bartocci et al. [5] the authors try to overcome this limitation by assuming that the oracle is a monitor generated from an STL specification. This approach allows the use of the trace diagnostic method proposed in Ferrère et al. [10] to obtain more information (e.g., the time interval when the cause of violation first occurs) about the failed tests improving the fault-localization. Although this additional knowledge can improve the confidence on the localization, still little is known about the root cause of the problem and its impact on the runtime behavior of the CPS model.

CPSDebug complements and improves SBFL techniques generating information that helps engineers identifying the cause of failures, understanding how faults resulted in chains of anomalous events that eventually led to the observed failures, and producing a corpus of information well-suited to support engineers in their debugging tasks, as confirmed by the subjects who responded to our questionnaire.

Failure explanation techniques analyze software failures in the attempt of producing information about failures and their causes. For instance, a few approaches combined mining and dynamic analysis in the context of component-based and object-oriented applications to reveal [24] and explain failures [3, 6, 21]. These approaches are not however straightforwardly applicable to CPS models, since they exploit the discrete nature of component-based and object-oriented applications that is radically different from the data-flow oriented nature of CPS models, which include mixed-analog signals, hybrid (continuous and discrete) components, and a complex dynamics.

CPSDebug originally addresses failure explanation in the context of CPS models. The closest work to CPSDebug is probably Hynger [14, 23], which exploits invariant generation to detect specification mismatches, that is, a mismatch between an actual and an inferred specification, in Simulink models. Spec-

ification mismatches can indicate the presence of problems in the models. Differently from Hynger, CPSDebug does not compare specifications but exploits inferred properties to identify anomalous behaviors in observed failures. Moreover, CPSDebug exploits correlation and clustering techniques to maintain the output compact, and to generate a sequence of snapshots that helps comprehensively defining the story of the failure. Our results show that this output can be the basis for cost-effective debugging.

A related body of research consists of approaches for anomaly detection of Cyber-Physical Systems [8, 26]. However, anomaly detection approaches aim at detecting misbehaviours, rather than analyzing failures and detecting their root causes as CPSDebug does.

7 Future Work and Conclusions

We have presented CPSDebug, an automatic approach for explaining failures in Simulink models. Our approach combines testing, specification mining and failure analysis to provide a concise explanation consisting of time-ordered sequence of model snapshots that show the variable exhibiting anomalous behavior and their propagation in the model. We evaluated the effectiveness CPSDebug on two models, involving two use scenarios and several classes of faults.

We believe that this paper opens several research directions. In this work, we only considered mining of invariant specifications. However, we have observed that invariant properties are not sufficient to explain timing issues, hence we plan to experiment in future work with mining of *real-time temporal* specifications. In particular, we will study the trade-off between the finer characterization of the model that temporal specification mining can provide and its computational cost. We also plan to study systematic ways to explain failures in presence of heterogeneous components. In this paper, we consider the setting in which we have multiple passing tests, but we only use a single fail test to explain the failure. We will study whether the presence of multiple failing tests can be used to improve the explanations. In this work, we have performed manual fault injection and our focus was on studying the effectiveness of CPSDebug on providing meaningful failure explanations for different use scenarios and classes of faults. We plan in the future to develop automatic fault injection and perform systematic experiments for evaluating how often CPSDebug is able to find the root cause.

Acknowledgments. This report was partially supported by the Productive 4.0 project (ECSEL 737459). 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. Abreu, R., Zoeteweyj, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: *Testing: Academic and Industrial Conference Practice and Research Techniques*, pp. 89–98. IEEE (2007)
2. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TALIRO: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) *TACAS 2011*. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
3. Babenko, A., Mariani, L., Pastore, F.: AVA: automated interpretation of dynamically detected anomalies. In: *proceedings of the International Symposium on Software Testing and Analysis (ISSTA)* (2009)
4. Bartocci, E., et al.: Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In: Bartocci, E., Falcone, Y. (eds.) *Lectures on Runtime Verification*. LNCS, vol. 10457, pp. 135–175. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_5
5. Bartocci, E., Ferrère, T., Manjunath, N., Nickovic, D.: Localizing faults in Simulink/Stateflow models with STL. In: *Proceedings of HSCC 2018: The 21st International Conference on Hybrid Systems: Computation and Control*, pp. 197–206. ACM (2018)
6. Befrouei, M.T., Wang, C., Weissenbacher, G.: Abstraction and mining of traces to explain concurrency bugs. *Form. Methods Syst. Des.* **49**(1–2), 1–32 (2016)
7. Deshmukh, J.V., Jin, X., Majumdar, R., Prabhu, V.S.: Parameter optimization in control software using statistical fault localization techniques. In: *Proceedings of ICCPS 2018: the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, pp. 220–231. IEEE Computer Society/ACM (2018)
8. Ding, M., Chen, H., Sharma, A., Yoshihira, K., Jiang, G.: A data analytic engine towards self-management of cyber-physical systems. In: *Proceedings of the International Conference on Distributed Computing Workshop*. IEEE Computer Society (2013)
9. Ernst, M., et al.: The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* **69**(1–3), 35–45 (2007)
10. Ferrère, T., Maler, O., Ničković, D.: Trace diagnostics using temporal implicants. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) *ATVA 2015*. LNCS, vol. 9364, pp. 241–258. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_20
11. Ghidella, J., Mosterman, P.: Requirements-based testing in aircraft control design. In: *AIAA Modeling and Simulation Technologies Conference and Exhibit*, p. 5886 (2005)
12. Hastie, T., Tibshirani, R., Friedman, J.H.: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics, 2nd edn. Springer, Heidelberg (2009)
13. Hoxha, B., Abbas, H., Fainekos, G.E.: Benchmarks for temporal logic requirements for automotive systems. In: *International Workshop on Applied verification for Continuous and Hybrid Systems*, volume 34 of EPiC Series in Computing, pp. 25–30. EasyChair (2015)
14. Johnson, T.T., Bak, S., Drager, S.: Cyber-physical specification mismatch identification with dynamic analysis. In: *Proceedings of ICCPS 2015: The ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, pp. 208–217. ACM (2015)

15. Lee, E.A.: Cyber physical systems: design challenges. In: Proceedings of ISORC 2008: The 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 363–369. IEEE Computer Society (2008)
16. Liu, B., Nejati, S., Briand, L.C.: Improving fault localization for Simulink models using search-based testing and prediction models. In: International Conference on Software Analysis, Evolution and Reengineering, pp. 359–370. IEEE Computer Society (2017)
17. Liu, B., Lucia, Nejati, S., Briand, L.C., Bruckmann, T.: Localizing multiple faults in Simulink models. In: International Conference on Software Analysis, Evolution, and Reengineering, pp. 146–156. IEEE Computer Society (2016)
18. Liu, B., Lucia, Nejati, S., Briand, L.C., Bruckmann, T.: Simulink fault localization: an iterative statistical debugging approach. *Softw. Test. Verif. Reliab.* **26**(6), 431–459 (2016)
19. Maler, O., Nickovic, D.: Monitoring properties of analog and mixed-signal circuits. *STTT* **15**(3), 247–268 (2013)
20. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT -2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30206-3_12
21. Mariani, L., Pastore, F., Pezzè, M.: Dynamic analysis for diagnosing integration faults. *IEEE Trans. Softw. Eng. (TSE)* **37**(4), 486–508 (2011)
22. Nghiem, T., Sankaranarayanan, S., Fainekos, G.E., Ivancic, F., Gupta, A., Pappas, G.J.: Monte-Carlo techniques for falsification of temporal properties of non-linear hybrid systems. In: International Conference on Hybrid Systems: Computation and Control, pp. 211–220 (2010)
23. Nguyen, L.V., Hoque, K.A., Bak, S., Drager, S., Johnson, T.T.: Cyber-physical specification mismatches. *TCPS* **2**(4), 23:1–23:26 (2018)
24. Pastore, F., et al.: Verification-aided regression testing. In: International Symposium on Software Testing and Analysis, ISSTA 2014, San Jose, CA, USA - 21–26 July 2014, pp. 37–48 (2014)
25. Sankaranarayanan S., Fainekos, G.E.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: International Conference on Hybrid Systems: Computation and Control, pp. 125–134. ACM (2012)
26. Sharma, A.B., Chen, H., Ding, M., Yoshihira, K., Jiang, G.: Fault detection and localization in distributed systems using invariant relationships. In: Proceedings of DSN 2013: The 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 1–8. IEEE Computer Society (2013)
27. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. *IEEE Trans. Software Eng.* **42**(8), 707–740 (2016)