International Journal on Software Tools for Technology Transfer (2021) 23:783–796
https://doi.org/10.1007/s10009-020-00599-4

GENERAL

Special Issue: MeTRID

CPSDebug: Automatic failure explanation in CPS models

Ezio Bartocci[1] · Niveditha Manjunath[1,2] · Leonardo Mariani[3] · Cristinel Mateis[2] · Dejan Ničković[2]

Accepted: 1 December 2020 / Published online: 8 January 2021
© The Author(s) 2020

Abstract

Debugging cyber-physical system (CPS) models is a cumbersome and costly activity. CPS models combine continuous and discrete dynamics—a fault in a physical component manifests itself in a very different way than a fault in a state machine. Furthermore, faults can propagate both in time and space before they can be detected at the observable interface of the model. As a consequence, explaining the reason of an observed failure is challenging and often requires domain-specific knowledge. In this paper, we propose approach, a novel CPSDebug that combines testing, specification mining, and failure analysis, to automatically explain failures in Simulink/Stateflow models. In particular, we address the hybrid nature of CPS models by using different methods to infer properties from continuous and discrete state variables of the model. We evaluate CPSDebug on two case studies, involving two main scenarios and several classes of faults, demonstrating the potential value of our approach.

Keywords Cyber-physical systems · Testing · Debugging · Model-based development · Property mining · Failure explanation

1 Introduction

Cyber-physical systems (CPS) are the emergent ICT systems that are characterized by tight interactions between computational and physical components in unpredictable environments. Model-based paradigms are increasingly adopted to cope with the inherent complexity of CPS and to enable their cost-effective design. As a consequence, the quality and correctness of these models are essential characteristics for successful CPS development.

✉ Niveditha Manjunath
niveditha.manjunath@live.com

Ezio Bartocci
ezio.bartocci@tuwien.ac.at

Leonardo Mariani
leonardo.mariani@unimib.it

Cristinel Mateis
cristinel.mateis@ait.ac.at

Dejan Ničković
dejan.nickovic@ait.ac.at

1 TU Wien, Vienna, Austria

2 AIT Austrian Institute of Technology, Vienna, Austria

3 University of Milano-Bicocca, Milan, Italy

The complexity of CPS models is manifold: these models typically combine discrete and continuous dynamics with an interplay between many variables, signals, state machines, look-up tables and components. Detecting problems in the early stages of CPS design [2,5,9,19,20] is of uttermost importance, before they propagate to the actual CPS with potentially catastrophic consequences.

While verification and testing tasks enable such early detection of faults, they are not sufficient on their own to debug CPS models. Indeed, it has been shown that debugging CPS models by identifying the causes of failures can be as challenging as identifying the problems themselves [18].

MathWorks™ Simulink environment is a de-facto standard for modeling and concept design of CPS. *Falsification-based testing* (FBT) [2,4,25,29] is a search-based approach for effectively finding faults in Simulink/Stateflow models.

FBT measures how far is a simulation trace from violating requirements expressed in a formal specification language, such as Signal Temporal Logic (STL) [23]. This measure is then used to systematically explore the input space and find the sequence of inputs that brings the system to violate the requirement.

This approach has been successfully adopted in various applications and applied to many case studies. However, this

method does not provide a useful information for resolving the violation and debugging the model.

Trace diagnostics [13] addresses this limitation by identifying segments of the observable model behavior that are sufficient to imply the violation of the formula. As a result, this method provides a failure explanation at the level of the model's input/output observable interface. However, this black-box technique does not explain faults in terms of the internal model's structure.

In this paper, we propose CPSDebug, a debugging technique that combines testing, specification mining, and failure analysis to identify the causes of failures. CPSDebug first simulates the CPS model under analysis by running the available test suite, while partitioning executions into passing and failing according to their evaluation against requirements formalized as a set of STL formulas.

While running the test cases, CPSDebug instruments the CPS model and records information about its internal behavior. In particular, it collects the values of all the internal system variables at every timestamp. CPSDebug uses the values from passing test executions to infer properties about the variables and components involved in the computations. These properties capture the correct and intended behavior of the system.

The failed test executions are then evaluated w.r.t. the mined properties to identify internal variables, and its corresponding components, that are responsible for the violation of the requirements.

Finally, CPSDebug analyzes the failure evidence to detect the violation times and cluster violations accordingly [15], finally producing a time-ordered sequence of snapshots that shows where the anomalous variables values originated from and how they propagated within the system. CPSDebug thus overcomes the limitation of the existing procedures that only provide a static explanation of the failure, such as the inputs or code locations.

The sequence of snapshots given 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 CPS models. Results show the effectiveness of our approach in supporting debugging activities of CPS models.

This paper extends our preliminary work on CPSDebug [6] as follows:

– We refine and improve our property mining procedure by adopting TkT [28], a timed-automata learning library, in addition to Daikon [12]. We use Daikon to infer invariants of real-valued signals and TkT to learn real-time relations between events originating from state machines and look-up tables.
– We compare these two approaches in two case studies. We demonstrate that by learning properties in the form of timed automata, we can derive meaningful explanations for failures resulting from time-dependent anomalies in enumerated variables. This class of anomalies are not captured and explained by Daikon-generated invariants.

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 $S = \{s_1, \cdots, s_n\}$ to be a set of signal variables. A *signal* or *trace* w is a function $\mathbb{T} \to \mathbb{R}^n$, 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 $w_i : \mathbb{T} \to \mathbb{R}$ associated with variables s_i for $i = 1, \cdots, n$. We assume that every signal w_i is piecewise-linear. Given two signals $u : \mathbb{T} \to \mathbb{R}^l$ and $v : \mathbb{T} \to \mathbb{R}^m$, we define their parallel composition $u \| v : \mathbb{T} \to \mathbb{R}^{l+m}$ in the expected way. Given a signal $w : \mathbb{T} \to \mathbb{R}^n$ 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 $w_R = \|_{s_i \in R} w_i$.

Let Θ be a set of terms of the form $f(R)$ where $R \subseteq S$ are subsets of variables and $f : \mathbb{R}^{|R|} \to \mathbb{R}$ 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*, $\Box_I \varphi \equiv \neg \Diamond_I \neg\varphi$ for *always*, $\diamondminus_I \varphi \equiv \top \mathcal{S}_I \varphi$ for *once*, $\boxminus_I \varphi \equiv \neg \diamondminus_I \neg\varphi$ 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 [22].

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 [12]. More formally, given a set of variables $V = V_1, \ldots, V_n$ defined over the domains $D_1, \ldots D_n$, an observation for these

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

variables is a tuple $\overline{v} = (v_1, \ldots, v_n)$, with $v_i \in D_i$. In our domain, variables are either signals or enumerated variables (e.g., a variable that represents the current state of a stateful component).

Given a set of variables V and multiple observations $\overline{v}_1 \ldots \overline{v}_m$ for these same variables, Daikon can be represented as a function $D(V, \overline{v}_1 \ldots \overline{v}_m)$ that returns a set of properties $\{p_1, \ldots p_k\}$, such that $\overline{v_i} \models p_j \forall i, j$, that is, all the observations satisfy the inferred properties.

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.

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 detect misbehaviors and their propagation. In a case study about an aircraft system that we studied in this paper, Daikon mined several interesting properties about signals. For instance, it discovered that the enumerated variable s_{32}, which represents the state of the component that decides whether the gear should be in steady state or should shift up/down, only covers a subset of the possible states. This is specified by the property:

$$\square_{[0,\infty]}((s_{32} == 0) \vee (s_{32} == 1) \vee (s_{32} == 3)).$$

Daikon also discovered useful relations between continuous signals, such as $\square_{[0,\infty]}(s_{47} \geq 0.0)$, $\square_{[0,\infty]}(s_{51} > s_{47})$ and $\square_{[0,\infty]}(s_{33} \leq s_{47})$. Interestingly, these properties show that when executions terminate correctly, s_{47} cannot be assigned with negative values, and its value is always below the value of signal s_{51} but is greater or equal than the value of signal s_{33}.

2.3 Timed k-Tail

Timed k-Tail (TkT) is an automata learning technique that can generate timed automata from timed traces [28]. A timed trace is an ordered sequence of events $event_1 \ldots event_n$ with $event_i = (op_i, type_i, time_i)$, where op_i indicates an operation whose execution is either starting ($type_i = \uparrow$) or finishing ($type_i = \downarrow$), and $time_i$ indicates the timestamp of the event. The timed trace to be well-formed must list events with non-decreasing timestamp values, every started operation must eventually finish at some point in the trace, and the operations must be properly nested, that is, an operation o_2 started after another operation o_1 must finish before operation o_1 finishes.

TkT only requires positive samples to learn timed automata, that is, the model can be learnt from a set of timed traces that must be accepted by the resulting automaton. TkT starts by generating a tree-shaped automaton, where each branch of the tree accepts exactly one timed trace. In particular, every event in the trace is mapped to a transition in the corresponding branch. Moreover, every time a transition accepts an event that represents the beginning of an operation ($type_i = \uparrow$), a clock reset is added to the transition. Similarly, a transition that accepts an event that represents the conclusion of an operation ($type_i = \downarrow$) is associated with a constraint that measures the duration of the operation, exploiting the clock that is reset at the beginning of the same operation. The specific duration used in the constraint is derived from the timestamps (e.g., if an operation starts at timestamp 10 and finishes at timestamp 15, the constraint requires the duration of the operation to be equal to 5). TkT may also add constraints on a global clock, which is reset on the first transition and checked on all other transitions.

TkT then generalizes the behavior accepted by the tree-shaped automaton merging equivalent states, by following the same heuristics used in the k-Tail algorithm [8]. That is, two states are assumed to be equivalent and redundant representations of a same state if they accept the same sequences of events of length k, where k is a parameter of the technique. As a consequence of the state merging process, the initial tree-like shape evolves into an automaton with branches and cycles, if necessary. During this merging process, also clock constraints can be merged and generalized with intervals of the form $v1 \leq c \leq v2$, where $v1$ and $v2$ are two constants and c is a clock. The resulting finite state model and time constraints are guaranteed to accept all the input traces used for the inference process. The interested reader can learn more about TkT in the paper by Pastore et al. [28].

In the context of this paper, TkT is exploited to monitor stateful components and reconstruct their behavior. A Simulink/Stateflow block may consist of several enumerated signals, each signal leading to an inferred TkT automaton. For example, Fig. 1 shows the inferred timed model obtained by monitoring the stateful component that governs the gear in the Automatic Transmission Control System used as one of the case studies. Each transition of the model represents a change in the value of the enumerated signal s_{32}, and the time constraints capture the normal timing of these state transitions.

Note that TkT produces information complemental to Daikon. Daikon derives properties about both continuous and enumerated signals, but cannot reconstruct the stateful behavior of a component, nor can it derive information about the timing of the operations. On the contrary, TkT works with

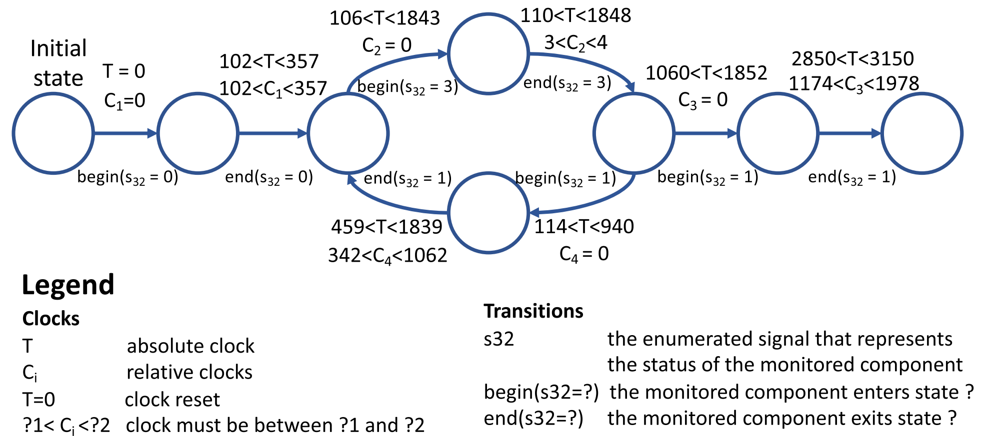


Fig. 1 Timed automaton inferred by TkT for a component that controls the gear in an Automatic Transmission Control System

enumerated signals only, but it can nicely reconstruct stateful computations, including timing information. We show in the evaluation how properties mined by Daikon and timed automata inferred by TkT can be jointly exploited to analyze failures.

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 [14] to illustrate model-based development of a Fault Detection, Isolation and Recovery (FDIR) application for a redundant actuator control system.

Figure 2 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. Either actuator can position, but at most 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 (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:

$$\varphi \equiv \Box(\uparrow (cmd \geq m) \rightarrow \Diamond_{[0,T]} \Box_{[0,t]}(|cmd - pos| \leq n)) \tag{1}$$

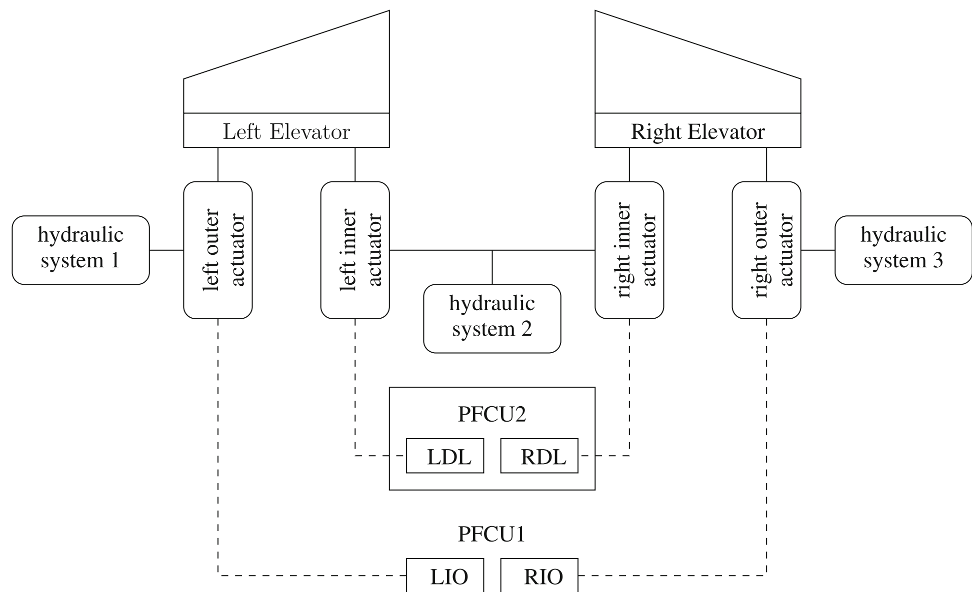


Fig. 2 Aircraft Elevator Control System [14]

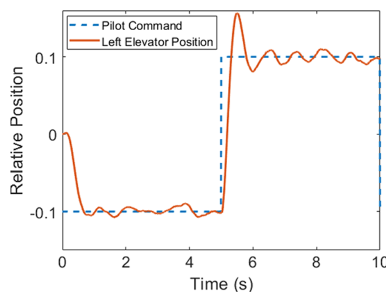


Fig. 3 Expected behavior of AECS

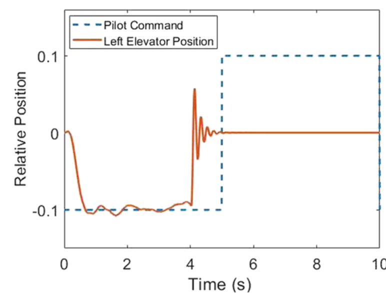


Fig. 4 Failure of the AECS

Figures 3 and 4 show the correct and faulty behavior of the system. In Fig. 4, 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 5 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:

(i) *Testing*, which exercises the previously 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

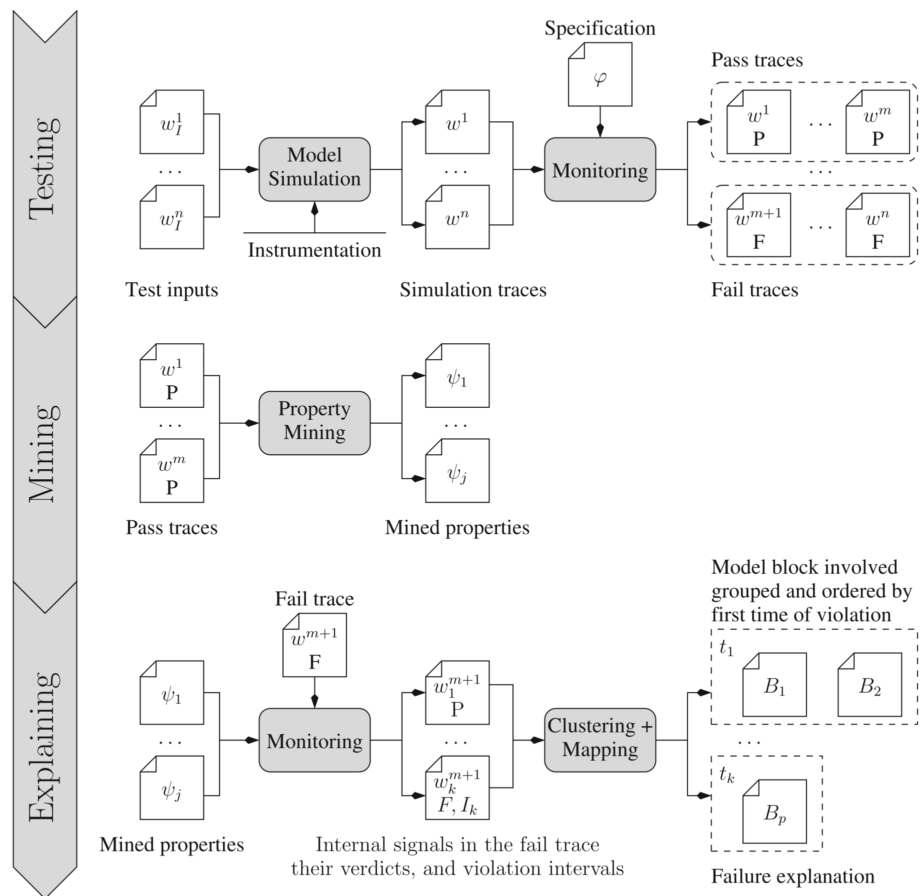


Fig. 5 Overview of the failure explanation procedure

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* $\{w_I^k | 1 \leq k \leq n\}$ 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_I^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 P* if w^k *satisfies* ϕ, or a *fail verdict F* 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 P and exploits them for *property mining*.

Prior to the property inference, CPSDebug performs several intermediate steps that facilitate the mining task. First, CPSDebug uses cross-correlation to reduce the set of variables V to its subset $\hat{V}$ of *significant* variables. 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 non-deterministically removes one of the two variables (and its

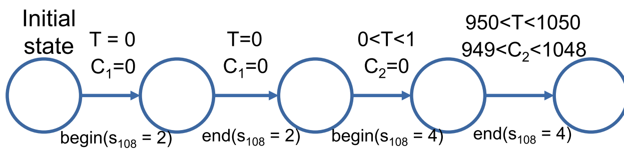


Fig. 6 Timed automaton for the state variable *mode* in the Left Outer Hydraulic Actuator block

associated traces) from further analysis, that is, $\hat{V}$ is updated to $\hat{V} \setminus v_1$. 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 $\{v_1, \ldots, v_n\}$ of variables with the domain D associated with block B. CPSDebug collects all observations $\overline{v}_1 \ldots \overline{v}_n$ from all samples in all traces associated with variables in $V_{D,B}$ and uses the Daikon function $D(V_{D,B}, \overline{v}_1 \ldots \overline{v}_n)$ and TkT automata learning engine to infer a set of properties $\{p_1, \ldots, p_k\}$ related to the block B and the domain D. As mentioned in Sect. 2.3, TkT is used to monitor stateful components which means that TkT infers properties only on state variables. 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. Each Daikon property p is transformed to an STL assertion of type $\Box\, p$. The TkT properties are in form of timed automata describing the behavior of the state variables and do not need any transformation for further use.

In our case study, Daikon returned 96 behavioral properties involving 121 variables and TkT returned 20 timed automata, one automaton for each state variable. Hence, CPSDebug generated an STL property ψ with 96 temporal assertions, i.e., $\psi = [\psi_1\ \psi_2 \ldots \psi_{96}]$, from Daikon properties. Equations 2 and 3 show 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.

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

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

Figure 6 shows the timed automaton for the state variable *mode* in the Left Outer Hydraulic Actuator block generated by TkT.

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 w.r.t. 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 and TkT.

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. That way, CPSDebug focuses on the events that cause observable misbehaviors first to reach the root cause of a failure.

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 seconds and 4 seconds. This suggests that CPSDebug can effectively *isolate in space and time* a limited number of events likely responsible for the failure. Figure 6 illustrates the timed automaton inferred by TkT for the variable *mode* in the Left Outer Hydraulic Actuator block in AECS. In Table 2, we observe that all faults detected by TkT are also captured by Daikon since no *guards* are violated. However, if such faults exist in the model, TkT is able to capture such failures since the time bounds are inferred. This can be observed in Table 2 for ATCS example where TkT efficiently captures the *guard* violation. Since properties mined by Daikon capture the *qualitative* behavior and not *timing*, Daikon does not capture the faulty guard in the example.

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, that is, 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 $\{B_1, \ldots, B_p\}$ that

Table 1 Internal signals that violate at least one learned invariant and Simulink blocks to which they belong

Index	Signal Name	Block	$\tau(s)$
s_{252}	LI_pos_fail:1→Switch:2	Meas. Left In. Act. Pos.	1.99
s_{253}	Outlier/failure:1→Switch:1	Meas. Left In. Act. Pos.	1.99
s_{254}	Measured Position3:1→Mux:3	Meas. Left In. Act. Pos.	1.99
s_{255}	Measured Position2:1→Mux:2	Meas. Left In. Act. Pos.	1.99
s_{256}	Measured Position1:1→Mux:1	Meas. Left In. Act. Pos.	1.99
s_{55}	BusSelector:2→Mux1:2	Controller	2.03
s_{328}	In2:1→Mux1:2	L_pos_failures	2.03
s_{329}	In1:1→Mux1:1	L_pos_failures	2.03
s_{332}	Right Outer Pos. Mon.:2→R_pos_failures:1	Actuator Positions	2.03
s_{333}	Right Inner Pos. Mon.:2→R_pos_failures:2	Actuator Positions	2.03
s_{334}	Left Outer Pos. Mon.:2→L_pos_failures:1	Actuator Positions	2.03
s_{335}	Right Inner Pos. Mon.:3→Goto3:1	Actuator Positions	2.03
s_{338}	Left Outer Pos. Mon.:3→Goto:1	Actuator Positions	2.03
s_{341}	Left Inner Pos. Mon.:2→L_pos_failures:2	Actuator Positions	2.03
s_{272}	LO_pos_fail:1→Switch:2	Meas. Left Out. Act. Pos.	3.99
s_{273}	Outlier/failure:1→Switch:1	Meas. Left Out. Act. Pos.	3.99
s_{275}	Measured Position1:1→Mux:1	Meas. Left Out. Act. Pos.	3.99
s_{276}	Measured Position2:1→Mux:2	Meas. Left Out. Act. Pos.	3.99
s_{277}	Measured Position3:1→Mux:3	Meas. Left Out. Act. Pos.	4.00

The column $\tau(s)$ denotes the first time that each signal participates in an invariant violation

Table 2 Scope reduction and cause detection

sys	**#vars**	**fault**	**Daikon**			**Daikon+TkT**		
			# ψ	**# suspicious vars (reduction)**	**fault detected**	**# ψ**	**# suspicious vars (reduction)**	**fault detected**
AECS	426	*lilo*	96	17(96%)	✓	96 + 20	17 + 15(92%)	✓
		h_1h_2	96	44(90%)	✓	96 + 20	44 + 15(86%)	✓
ATCS	51	*guard*	41	1(98%)		41 + 5	1 + 2(94%)	✓
		eng_lt	39	4(92%)	✓	39 + 5	4 + 2(88%)	✓

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 the 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 7 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 Controller blocks. By opening the Sensors block, we can immediately see that the sensor measuring the left inner position of the actuator is marked as a possible cause of the failure. 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

In order to verify the improvement over the approach from our preliminary work [6] which only used Daikon, we repeated the evaluation from [6] by using also TkT in addition to Daikon and compared the results. More precisely, we empirically evaluated our two approaches, i.e., with and without TkT, 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 use two different systems to introduce faults from these three classes. We use the fault-tolerant aircraft elevator control system (AECS) [14] 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 (ATCS) [16] illustrated in Fig. 8 to study the other classes of faults. The 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 [16] 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*).

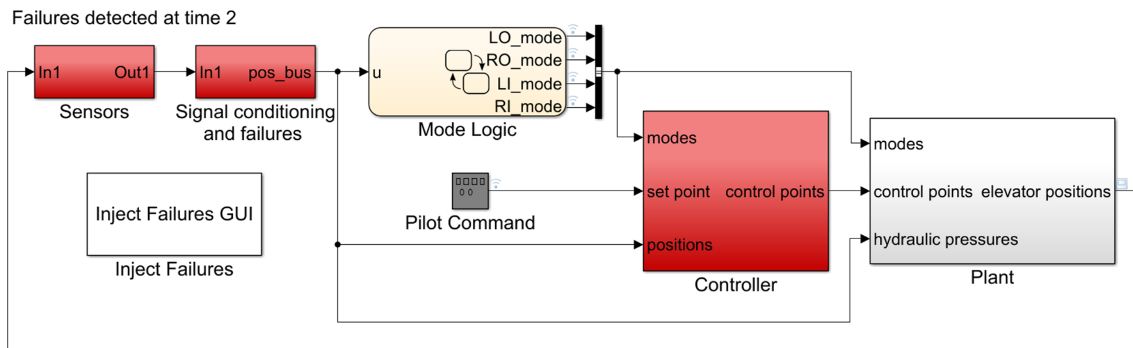
To study these faults, we considered two scenarios. For the aircraft elevator control system, we executed 150 test cases in which we systematically changed the amplitude and the frequency of the pilot command steps. These tests were executed on a non-faulty model. We then executed an additional test 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.

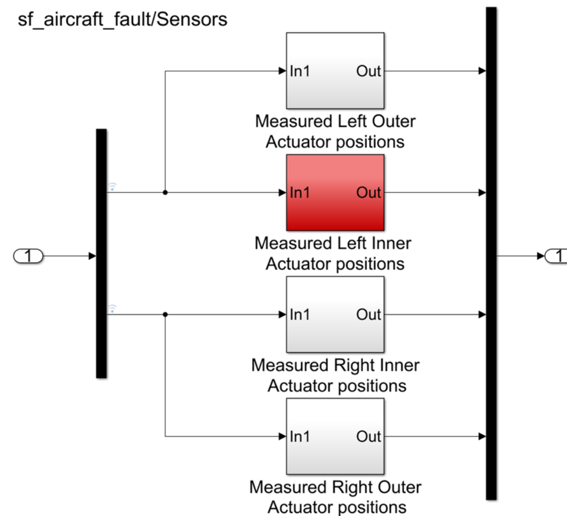
We evaluated and compared the output produced by the two approaches, i.e., Daikon and Daikon + TkT, considering four main aspects: Scope Reduction, Cause Detection, Quality of the Analysis, and Computation Time. Scope Reduction measures how well each approach narrows down the number of elements to be inspected to a small number of anomalous signals that require the attention of the engineer, in comparison with the set of variables involved in the failed execution. Cause detection indicates if the first cluster of anomalous values reported by each 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. In summary, our analysis shows that TkT improves the performance of CPSDebug at low cost in terms of computation time.

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 in a comparative way for the purely Daikon-based approach and its improved version which uses in addition TkT. 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, including both real-valued signals and state signals. Column *fault* indicates the specific fault analyzed. Column # ψ gives the number of learned invariants. We emphasize that, whereas Daikon





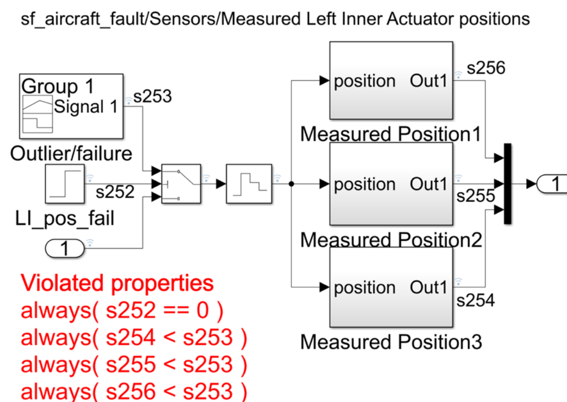


Fig. 7 Failure explanation as a sequence of snapshots—part of the first snapshot

derives invariants from both real-valued and state signals, the TkT-related invariants are timed automata derived from state signals only. 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.

We can see from Table 2 that CPSDebug successfully detected the exact origin of the fault in only 3 out of 4 cases when it used only Daikon. On the other hand, when CPS-Debug used both Daikon and TkT, the fault was detected in all 4 cases. In the aircraft elevator control system, CPS-Debug clearly identified the problem with the respective sensors (fault *lilo*) and hydraulic components (fault h_1h_2) also without the help of TkT. TkT identified 15 additional suspicious variables but none of these turned out to be exclusively responsible for some fault. That is, these (unuseful) additional suspicious variables determined a slightly worse

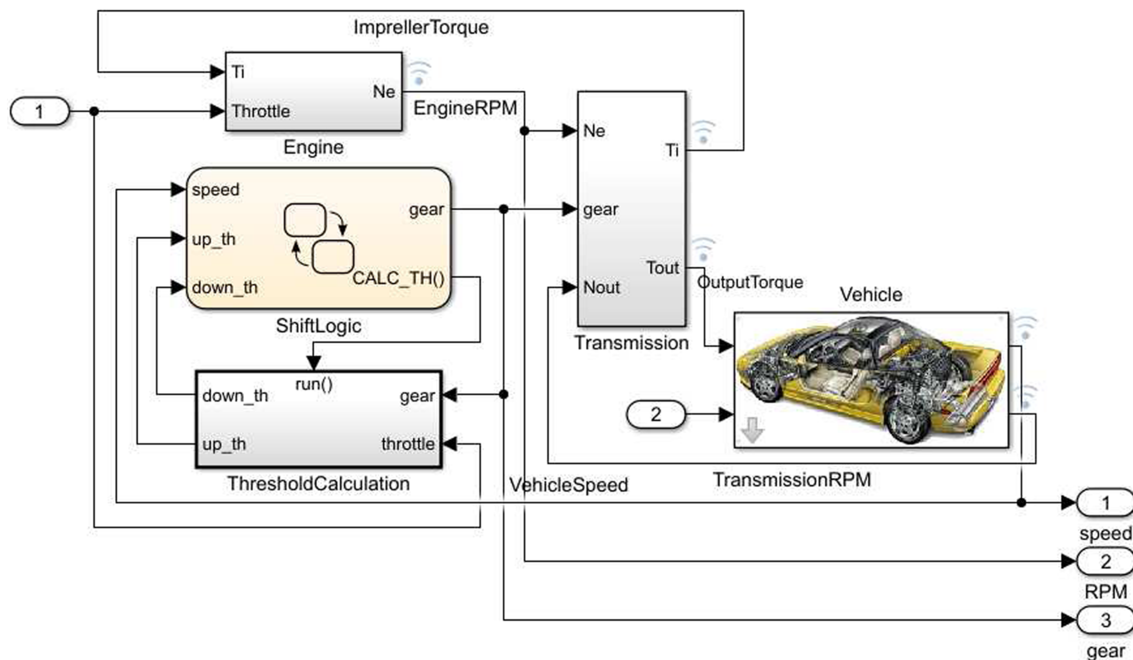


Fig. 8 Automatic Transmission Control System (ATCS)

reduction in the number of variables to be inspected when both Daikon and TkT were used. However, the overall reduction of 92%, resp. 86%, was still considerably high, allowing engineers to focus on a small subset of the suspicious signals.

In the case of the automatic transmission control, CPSDebug associates the misbehavior of the model with the Engine look-up table and points correctly to the faulty entry without the help of TkT, by only using Daikon. However, TkT turned out to be crucial for detecting the exact origin of the *guard* fault which could not be detected by using Daikon alone. 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, the purely Daikon-based approach is not able to discriminate between passing and failing tests in that case. This is exactly where the timed automata provided by TkT can help.

5.2 Computation time

Table 3 summarizes computation time of CPSDebug applied to the two case studies. In order to assess the overhead added by TkT, the computation times of the TkT- and Daikon-related activities are reported separately. The simulation and instrumentation activities are required and have the same costs, regardless of whether CPSDebug uses also TkT or not. There are several conclusions we can make from these experimental results:

- the overall computation time of CPSDebug-specific activities is comparable to the overall simulation time,
- Daikon property mining dominates by far the computation of the explanation, whereas TkT property mining and explanation computation times are similar and low, and
- the overall TkT computation time is by far lower than the overall Daikon computation time.

We finally report in the last row for both Daikon and TkT 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 and TkT, respectively. Since mining properties for TkT are performed only on state-based variables, we provide the time taken by Daikon to mine properties for comparison. In our prototype implementation of CPSDebug, we use an inefficient Daikon format translation that results in excessive time. We believe that investing additional effort can result in improving the Daikon translation time by several orders of magnitude. The computation time required by the TkT format translation is considerably higher than for the TkT mining and explanation activities but still by several magnitude orders lower than the total CPSDebug computation time. Overall, the accuracy of CPSDebug can be increased by using TkT in addition to Daikon without paying any significant cost in terms of computation time.

5.3 Discussion

In CPS engineering, teams are multidisciplinary, for instance they include both control and system engineers. Our results show that CPSDebug can be useful to all the categories of

Table 3 CPSDebug computation time

	Aircraft	Transmission
# tests	150	100
# samples per test	1001	751
time (s)		
Simulation	654	35
Instrumentation	1	0.7
Daikon		
Mining	501	52
Monitoring properties	0.7	0.6
File format translation	2063	150
Daikon (state-based signals)		
Mining	31	4
Monitoring properties	0.7	0.3
File format translation	80	26
TkT		
Mining	9.3	2.1
Monitoring properties	9.5	2.1
File format translation	37	13
Analysis	1.5	1.6

engineers. In the aircraft elevator control system, CPSDebug reports a useful feedback to system engineers by isolating multiple faulty components responsible for an injected failure. In the automatic transmission control case study, CPSDebug provides an insightful feedback to the control engineers by isolating the fault present in the lookup table.

We note that certain faults can be masked by redundant components. This can for instance happen in the aircraft elevator control system. CPSDebug is able to detect masked faults as long as there is an observable violation of the overall specification. For example, if the left inner, left outer and right inner actuator position components fail, CPSDebug identifies anomalies in all three components, including the right inner actuator position component whose fault is masked by the right outer actuator position component. In contrast, CPSDebug is not able to detect a masked fault if it does not impact the satisfaction of the system-level specification. The specification violation is the prerequisite to even start the fault explanation analysis.

Although our evaluation focuses on Simulink models, CPSDebug is a quite general approach. In fact, as long as the individual components of the system under analysis can be observed, CPSDebug can be used to analyze failures. Clearly, some cases might be harder to address than others. For instance, hardware components are sometime difficult to instrument.

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 (see the extensive survey in [31]). *Spectrum-based fault-localization* (SBFL) [1] is 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 [5,9,19–21], showing similar accuracy as in the application to software systems [21]. The explanatory power of this approach is, however, limited, because it generates neither information that can help the engineers understand if a selected code location is really faulty nor information about how a fault is propagated across components. 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 does not conform with 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. [13] 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 identify the cause of failures, understand how faults resulted in chains of anomalous events that eventually led to the observed failures and produce a corpus of information well-suited to support engineers in their debugging tasks.

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 [27] and explain failures [3,7,24]. 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 [17,26], which exploits invariant generation to detect mismatches between an actual and an inferred specification in Simulink models. Specification 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 [10,30]. However, anomaly detection approaches aim at detecting misbehaviors, rather than analyzing failures and detecting their root causes as CPSDebug does.

Finally, we mention the recent work of analyzing neighborhoods of falsifying traces in CPS [11] where the authors use search-based testing to characterize how "robust" is the falsifying trace. This provides an alternative and complementary way of understanding properties of the violation.

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. Our approach differentiates between learning properties from discrete and from real-valued variables -- we use Daikon for inferring invariants over continuous signals and TkT for learning timed automata from time-stamped enumerated sequences of events. We evaluated the effectiveness of CPSDebug on two models, involving two use scenarios and several classes of faults.

We believe that this paper opens several research directions. We plan to study systematic ways to explain failures in the 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. Finally, we will investigate how to combine automated test generation with our approach to improve the quality of explanations.

Acknowledgements 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. This work was also partially supported by the IoT4CPS project funded the Austrian FFG and BMVIT.

Funding Open Access funding provided by TU Wien (TUW)

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abreu, R., Zoeteweij, 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. Annapureddy, Y., Liu, C., Fainekos, G.E., Sankaranarayanan, S.: 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, volume 6605 of LNCS, pp. 254–257. Springer (2011)
3. Babenko, A., Mariani, L., Pastore, F.: AVA: Automated interpretation of dynamically detected anomalies. In: Proceedings of ISSTA 2009: International Symposium on Software Testing and Analysis, pp. 237–248. ACM (2009)
4. Bartocci, E., Deshmukh, J.V., Donzé, A., Fainekos, G.E., Maler, O., Nickovic, D., Sankaranarayanan, S.: Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In: Lectures on Runtime Verification—Introductory and Advanced Topics, volume 10457 of LNCS, pp. 135–175. Springer (2018)
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. Bartocci, E., Manjunath, N., Mariani, L., Mateis, C., Nickovic, D.: Automatic failure explanation in CPS models. In: Proceedings of SEFM 2019: the 17th International Conference on Software

Engineering and Formal Methods, volume 11724 of LNCS, pp. 69–86. Springer (2019)
7. Befrouei, M.T., Wang, C., Weissenbacher, G.: Abstraction and mining of traces to explain concurrency bugs. Formal Methods Syst. Des. **49**(1–2), 1–32 (2016)
8. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. IEEE Trans. Comput. **21**(6), 592–597 (1972)
9. 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)
10. 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 IEEE 33rd International Conference on Distributed Computing Workshop, pp. 303–308. IEEE Computer Society (2013)
11. Diwakaran, R.D., Sankaranarayanan, S., Trivedi, A.: Analyzing neighborhoods of falsifying traces in cyber-physical systems. In: Proceedings of ICCPS 2017: the 8th International Conference on Cyber-Physical Systems, pp. 109–119. ACM (2017)
12. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
13. Ferrère, T., Maler, O., Nickovic, D.: Trace diagnostics using temporal implicants. In: International Symposium on Automated Technology for Verification and Analysis, volume 9364 of LNCS, pp. 241–258. Springer (2015)
14. Ghidella, J., Mosterman, P.: Requirements-based testing in aircraft control design. In: AIAA Modeling and Simulation Technologies Conference and Exhibit, pp. 5886 (2005)
15. Hastie, T., Tibshirani, R., Friedman, J.H.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics, 2nd edn. Springer, Berlin (2009)
16. 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)
17. 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)
18. Lee, E.A.: Cyber physical systems: design challenges. In: Proceedings of ISORC2008: the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, pp. 363–369. IEEE Computer Society (2008)
19. Liu, B., Nejati, S., Briand, L.C.: Improving fault localization for Simulink models using search-based testing and prediction models. In: Proceedings of SANER 2017: the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, pp. 359–370. IEEE Computer Society (2017)
20. Liu, B., Nejati, S., Briand, L.C., Bruckmann, T.: Localizing multiple faults in Simulink models. In: Proceedings of SANER 2016: the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, pp. 146–156. IEEE Computer Society (2016)
21. Liu, B., Nejati, S., Briand, L.C., Bruckmann, T.: Simulink fault localization: an iterative statistical debugging approach. Softw. Test. Verif. Reliab. **26**(6), 431–459 (2016)
22. Maler, O., Nickovic, D.: Monitoring properties of analog and mixed-signal circuits. STTT **15**(3), 247–268 (2013)
23. Maler, O., Ničković, D.: Monitoring temporal properties of continuous signals. In: Joint International Conferences on Formal Modelling and Analysis of Timed Systems, and Formal Techniques in Real-Time and Fault-Tolerant Systems, volume 3253 of LNCS, pp. 152–166. Springer (2004)
24. Mariani, L., Pastore, F., Pezzè, M.: Dynamic analysis for diagnosing integration faults. IEEE Trans. Softw. Eng. (TSE) **37**(4), 486–508 (2011)
25. 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: Proceedings of HSCC 2010: the 13th International Conference on Hybrid Systems: Computation and Control, pp. 211–220 (2010)
26. 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)
27. Pastore, F., Mariani, L., Hyvärinen, A.E.J., Fedyukovich, G., Sharygina, N., Sehestedt, S., Muhammad, A.: Verification-aided regression testing. In: Proceedings of ISSTA 2014: International Symposium on Software Testing and Analysis, pp. 37–48 (2014)
28. Pastore, F., Micucci, D., Mariani, L.: Timed k-Tail: Automatic inference of timed automata. In: Proceedings of ICST 2017: the International Conference on Software Testing, Verification and Validation, pp. 401–411. IEEE Computer Society (2017)
29. Sankaranarayanan, S., Fainekos, G.E.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: Proceedings of HSCC 2012: the 15th International Conference on Hybrid Systems: Computation and Control, pp. 125–134. ACM (2012)
30. 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)
31. 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)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.