CPSDebug: A Tool for Explanation of Failures in Cyber-Physical Systems

Ezio Bartocci
Vienna University of Technology
Vienna, Austria

Niveditha Manjunath
AIT Austrian Institute of Technology
Vienna University of Technology
Vienna, Austria

Leonardo Mariani
University of Milano-Bicocca
Milan, Italy

Cristinel Mateis
AIT Austrian Institute of Technology
Vienna, Austria

Dejan Ničković
AIT Austrian Institute of Technology
Vienna, Austria

Fabrizio Pastore
University of Luxembourg
Luxembourg City, Luxembourg

ABSTRACT

Debugging Cyber-Physical System models is often challenging, as it requires identifying a potentially long, complex and heterogenous combination of events that resulted in a violation of the expected behavior of the system. In this paper we present CPSDebug, a tool for supporting designers in the debugging of failures in MATLAB Simulink/Stateflow models. CPSDebug implements a gray-box approach that combines testing, specification mining, and failure analysis to identify the causes of failures and explain their propagation in time and space. The evaluation of the tool, based on multiple usage scenarios and faults and direct feedback from engineers, shows that CPSDebug can effectively aid engineers during debugging tasks.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Cyber-Physical Systems, Model-based Development, Testing, Specification Mining, Debugging, Failure Explanation

ACM Reference Format:
Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, Dejan Ničković, and Fabrizio Pastore. 2020. CPSDebug: A Tool for Explanation of Failures in Cyber-Physical Systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404369

1 INTRODUCTION

Cyber-Physical System (CPS) models are complex and heterogenous models that combine discrete and continuous dynamics to precisely capture the behavior of CPSs. Faults in the models must be timely revealed and fixed, otherwise the successful realization of the CPS can be compromised.

Indeed, debugging can be an extremely challenging task in this domain. Failures might be caused by multiple heterogeneous sources in the model, such as continuous dynamics, switching mechanisms implemented by Finite-State Machine (FSM) components, violated time constraints, incorrect entries in look-up tables and unexpected component interactions. Failures may also propagate in time (i.e., a misbehaviour might be the cause of a later misbehaviour) and space (i.e,. a misbehaviour in a component might be the cause of misbehaviours in other components) before they become observable.

Debugging tasks can be supported by *fault-localization* and *failure explanation* techniques. Fault-localization consists of suggesting the possible fault locations based on the program elements executed by passing and failing executions [19]. A popular example is *spectrum-based fault-localization* [1] (SBFL), which has been recently extended to Simulink/Stateflow CPS models [3, 6, 10–12]. Although sometime useful, SBFL techniques have a limited explanatory capability [16], since they provide little overview and explanation of the failure, not really helping engineers assess whether the identified components are faulty and how the failure propagated across the components resulted on an actual failure.

Failure explanation techniques support debugging by identifying the sequence of misbehaviors that may explain a failure [2, 5, 14, 17]. These works are not directly applicable to CPS models since they usually leverage 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 analog-mixed signals, hybrid (continuous and discrete) components, and a complex dynamics.

To address this problem, we defined CPSDebug [4], a technique that combines testing, specification mining, and failure analysis to identify the causes of failures in CPS models. CPSDebug outputs a sequence of snapshots that reconstructs how misbehaviors propagate in space and time in the context of the failure. This information can be exploited by engineers to understand the reason of the failure and fix the fault.

This paper describes the tool that implements CPSDebug. It is implemented in MATLAB, it is fully automatic and freely available at https://gitlab.com/nmanjunath/cpsdebug.git for download. Early results with an Automatic Transmission Control System and an

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3404369

Aircraft Elevator Control System, including feedback from engineers who inspected the output of the tool, confirm the value of the tool.

The rest of the paper is organized as follows. Section 2 summarizes how CPSDebug works, and describes the architecture and usage of the tool. Section 3 describes usage scenarios and summarizes the empirical results that we obtained using the tool. We conclude in Section 4.

2 CPSDEBUG

This section describes CPSDebug, its architecture and usage.

2.1 The Approach

CPSDebug produces failure explanations by *Testing* the CPS model, by *Mining* properties from the data collected during testing, and by *Explaining* the failures based on the violations of mined properties and their relationships. The CPS models input to CPSDebug are MATLAB Simulink/Stateflow models.

The *testing* phase first instruments the software so that the values assigned to every signal and variable of the Simulink model at every timestamp are logged, and then runs the available test cases. A Signal-Temporal Logic (STL) specification [13] is used to distinguish failing and passing test cases, and to classify traces accordingly.

The *mining* phase infers properties from the traces recorded during the execution of passing test cases, to capture how the individual signals and components of the Simulink model behave when no failure is reported. CPSDebug performs two pre-processing steps before mining properties: It identifies correlated signals to drop signals that only provide redundant information; and it groups traces per component (i.e., Simulink block) so that the inferred properties refer to the individual components of the model. These steps are useful to generate properties whose violation can be easily interpreted, because each violation refers to a well defined component. In addition, they avoid the combinatorial explosion of the size of input to the mining tools, which are launched multiple times on small groups of variables rather than on all variables at once. CPSDebug infers two classes of properties from traces: properties that represent the *values that can be assigned to signals* inferred with Daikon [7] and timed automata that capture the *timed state-based behavior of stateful components* inferred with TkT [18]. Failures that can be explained in terms of violations of these properties can be addressed with CPSDebug.

The *explaining* phase exploits the mined properties to analyze the traces recorded during the execution of the failing test cases and generate failure explanations. In particular, CPSDebug collects the signals and variables that violate the mined properties at any point in time and reconstructs the history of the failing execution focusing on the propagation of the misbehaviors. To do this, CPSDebug clusters misbehaviors, to compactly report groups of violations (we refer to these groups of violations as a snapshot), and exploits their time and space distribution, to order in time and localize on the components of the Simulink model the snapshots. The engineer can inspect the snapshots from the first one, usually indicating the faulty component, to the last one to reconstruct the behavior of the software during the failure.

2.2 Tool Architecture

Figure 1 illustrates the structure of the tool, which consists of the components described below.

The *Model Instrumenter* is responsible of instrumenting the CPS model under analysis. The objective of instrumentation is to log values assigned to every internal signal and variable in the model. Our Model Instrumenter is implemented as a MATLAB component that inductively navigates the hierarchical structure of the Simulink/Stateflow model in a bottom-up fashion adding tracing capabilities to every element. For every Real, Integer, Boolean value and Enumerated-type signal, it automatically assigns a unique identifier and enables logging of the signal. For every state machine and look-up-table, it adds the logic necessary to record every state transition and access to the table, respectively.

The *Tester* component is a regular off-the-shelves test executor used to run the available test cases. CPSDebug currently uses the MATLAB Simulation Engine to run the test cases. CPSDebug uses an STL formula that is automatically evaluated by a *monitor* against the output traces of the tests. These traces are finally labeled with a *pass* verdict, if the test satisfies the STL formula, or with a *fail* verdict otherwise. CPSDebug uses the RTAMT library [15] and its input language to encode and monitor STL specifications and label the tests with pass/fail verdicts.

The *Property Miner* selects the traces labeled with a *pass* verdict and uses them for mining properties. It is also responsible of dropping the correlated signals and grouping traces per component (i.e., per Simulink/Stateflow block). Our tool is designed to integrate and run a number of mining solutions. The current version integrates Daikon [7] and TkT [18] to derive universal and real-time state-based properties.

The *Monitor + Trace Diagnoser* component produces failure explanations by analyzing the collected failing traces. In this step, when the trace is evaluated against the mined properties, a list of signals that violate the properties and the time at which the signals first violate the properties are reported. CPSDebug uses RTAMT to detect violations of universal properties and TkT to detect violations of real-time state-based properties.

The *Failure Mapper* clusters the fail-annotated signals by their violation times and maps them to their corresponding model blocks, i.e., to the model blocks that the fail-annotated signals originate from. The violated signals with their corresponding origin blocks are used to create a sequence of snapshots, one for each cluster of property violations. The designer can inspect the sequence of snapshots to reconstruct the anomalous behaviors that have been observed during the failing execution, and identify both the root events and root components responsible for the investigated problem.

2.3 Tool Usage

CPSDebug can be executed in any environment equipped with Java 8, Python and MATLAB 2016b to 2018b. It fully supports CPS models designed in MATLAB Simulink/Stateflow, but it does not support models exploiting other MATLAB toolboxes, such as Simscape/SimEvents. The tool can be executed from the command line to be easily integrated within automated scripts.

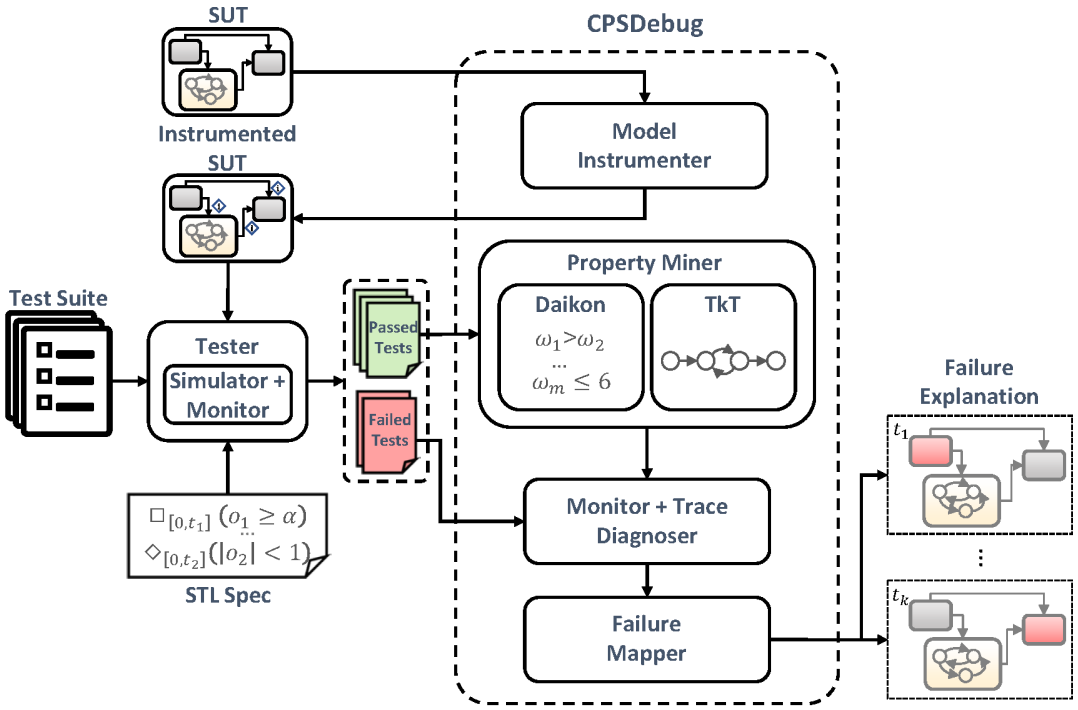

Figure 1: Structure of CPSDebug

The behavior of the tool is influenced by a configuration file that can be used to specify parameters. To launch the tool, it is necessary to specify the model under test, the STL specification used as test oracle and the test suite.

The tool produces as failure explanation output a table for each failed tests. The table stores in tabular form an ordered sequence of the identified snapshots. The data for each snapshot include the signals that violated the mined properties, the time of violation, the block that the signal belongs to and the violated properties. In addition, the tool provides statistics about the executed tests and the mined properties, such as the number of passed and failed test, and the number of properties inferred with Daikon and TkT.

3 TOOL ASSESSMENT

We experimented CPSDebug in the context of both coverage-based and regression testing [4]. We experimented the former scenario with the Automatic Transmission Control System (ATCS) [9] and the latter scenario with the Aircraft Elevator Control System (AECS) [8] model. We ran the experiments on Intel i7-8650 2.11GHz with 8 cores and 32GB RAM. The models chosen to assess CPSDebug contain both continuous and discrete blocks with hierarchies. They are publicly available models of medium complexity that are representative of industrial models with regard to behavioural dynamics.

Coverage-based testing. ATCS is a model of an automatic transmission controller that exhibits both continuous and discrete behavior. The system has two inputs, the throttle u_t and the break u_b, an two continuous-time state variables, the speed of the engine ω (rpm), the speed of the vehicle v (mph) and the active gear g_i. When the system is started, both the vehicle speed and the engine speed are initialized to 0. It follows that the output trajectories depend only on the input signals u_t and u_b, which can take any value between 0 and 100 at any point in time.

The overall ATCS model consists of 55 signals, 4 look-up tables and 2 FSMs running in parallel with 3 and 4 states, respectively. The available coverage-based test suite consists of 100 tests implemented to cover the functionalities of the system. We experimented CPSDebug with two injected faults: (1) a faulty transition in a FSM, and (2) an altered entry in a look-up table.

The outcome of CPSDebug is the set of three signals that explain the failure, two concerning the *throttle* and one concerning the *engine torque*, all having values higher than expected. The signal about the *engine torque* is the output of the component in which the fault was injected, while the two signals about the throttle are the inputs that trigger this anomalous behavior. Since both the triggers of the anomalous behavior and the anomalous output generated by the faulty component have been identified, it is straightforward for the engineer to identify the right component to inspect and reveal the fault. Indeed, failure explanation reduces the scope of the analysis by 95% allowing the engineers to focus on a small subset of the suspicious signals.

Regression testing. The architecture of an AECS contains one elevator on the left and one on the right side, with redundancy. 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 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.

The AECS model consists of 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. We experimented AECS with a fault each in *hydraulic system2* and *hydraulic system3* components. These example faulty components are available with AECS.

The fault in the *hydraulic system 3* in AECS was injected at time 2 and the fault in *hydraulic system 2* was injected at time 5. The output of CPSDebug is a set of 51 signals partitioned into two snapshots. The first snapshot consists of 50 signals that show an anomaly at time 2 and the second snapshot consists of 1 signal that shows an anomaly at time 5. The two snapshots contain output signals from the faulty hydraulic systems 3 and 2, respectively. The engineer simply focuses on the components responsible of the anomalous signals in the two snapshots. It follows that the failure explanation reduces the scope of the investigation performed by the engineer by 88% while detecting the root cause of the failure, thus helping the engineer to quickly identify faulty components.

Out of 171 tests, 166 tests were labeled as fail after the faulty components were introduced in the system. This large number of failures is due to a rigid specification that has strong assumptions about the environment. This setting has been useful to experiment the usage of the tool when most of the tests fail and little information is available to generate properties.

Feedback from Professionals. To confirm that the output produced by the tool is indeed useful to engineers, we submitted the output of the tool to 3 engineers from major tool vendors and automotive OEMs asking them to evaluate the outcomes of our tool for a selection of faults. The engineers found the tool potentially useful since debugging of CPS models can be very difficult. They shared appreciation for the generated output and they were willing to experiment the tool with their projects. More details about our experimental evaluation can be found in the paper by Bartocci et al. [4].

4 CONCLUSION

Debugging faults in Cyber-Physical System models can be extremely challenging due to the complex interactions between discrete and continuous behaviors. Indeed, an anomalous behavior may propagate through many components before resulting in an observable failure. Reconstructing the complex and potentially long chain of events responsible of a failure is of fundamental importance to localize, understand and fix the fault.

To address this challenge, we presented CPSDebug, a tool for explaining failures in MATLAB/Simulink models. The tool combines testing, specification mining, runtime verification and clustering techniques to derive meaningful explanations, which consist of sequences of snapshots that reconstruct the propagation in time and space of the anomalous events responsible for the failure, offering a clear starting point for the analysis of the failure and the correction of the fault.

As part of our future work, we plan to improve CPSDebug along two lines. We will first enhance the tool usability by directly integrating it in MATLAB, so that it can be launched and used without leaving the programming environment. Moreover, we plan to investigate the possibility to integrate guided test generation strategies designed to improve the explanations.

REFERENCES

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

[2] Anton Babenko, Leonardo Mariani, and Fabrizio Pastore. 2009. AVA: Automated Interpretation of Dynamically Detected Anomalies. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ACM, 237–248.

[3] Ezio Bartocci, Thomas Ferrère, Niveditha Manjunath, and Dejan Nickovic. 2018. Localizing Faults in Simulink/Stateflow Models with STL. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control*. ACM, 197–206.

[4] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, and Dejan Nickovic. 2019. Automatic Failure Explanation in CPS Models. In *Software Engineering and Formal Methods (LNCS)*, Vol. 11724. Springer, 69–86.

[5] Mitra T. Befrouei, Chao Wang, and Georg Weissenbacher. 2016. Abstraction and Mining of Traces to Explain Concurrency Bugs. *Formal Methods in System Design* 49, 1-2 (2016), 1–32.

[6] Jyotirmoy V. Deshmukh, Xiaoqing Jin, Rupak Majumdar, and Vinayak S. Prabhu. 2018. Parameter optimization in control software using statistical fault localization techniques. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*. IEEE, 220–231.

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1-3 (2007), 35–45.

[8] Jason Ghidella and Pieter Mosterman. 2005. Requirements-based testing in aircraft control design. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*.

[9] Bardh Hoxha, Houssam Abbas, and Georgios Fainekos. 2015. Benchmarks for Temporal Logic Requirements for Automotive Systems. In *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems (EPiC Series in Computing)*, Vol. 34. 25–30.

[10] Bing Liu, Lucia Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 359–370.

[11] Bing Liu, Lucia Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Localizing Multiple Faults in Simulink Models. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, 146–156.

[12] Bing Liu, Lucia Lucia, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability* 26, 6 (2016), 431–459.

[13] Oded Maler and Dejan Nickovic. 2013. Monitoring properties of analog and mixed-signal circuits. *Software Tools for Technology Transfer* 15, 3 (2013), 247–268.

[14] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Transactions on Software Engineering (TSE)* 37, 4 (2011), 486–508.

[15] Dejan Nickovic and Tomoya Yamaguchi. 2020. RTAMT: Online Robustness Monitors from STL. arXiv:2005.11827 [cs.LO]

[16] Chris Parnin and Alex Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 199–209.

[17] Fabrizio Pastore, Leonardo Mariani, Antti E. Johannes Hyvärinen, Grigory Fedyukovich, Natasha Sharygina, Stephan Sehestedt, and Ali Muhammad. 2014. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 37–48.

[18] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. 2017. Timed k-Tail: Automatic Inference of Timed Automata. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 401–411.

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