Mondragon Unibertsitatea Biblioteka Biblioteca

biblioteka@mondragon.edu

This is an Accepted Manuscript version of the following article, accepted for publication in:

P. Valle and A. Arrieta, "Towards the Isolation of Failure-Inducing Inputs in Cyber-Physical Systems: is Delta Debugging Enough?," 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 549-553.

DOI: https://doi.org/10.1109/SANER53432.2022.00072

© 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Towards the Isolation of Failure-Inducing Inputs in Cyber-Physical Systems: is Delta Debugging Enough?

Pablo Valle
Electronics and Computing Department
Mondragon University
Mondragon, Spain
pablo.valle@alumni.mondragon.edu

Aitor Arrieta
Electronics and Computing Department
Mondragon University
Mondragon, Spain
aarrieta@mondragon.edu

***Abstract*—Cyber-Physical Systems (CPSs) combine digital cyber technologies with parallel physical processes. On the one hand, verification methods of such systems mostly rely on (system level) simulation-based testing. This technique is expensive because complex mathematical models are used to model the physical part of CPSs. On the other hand, test cases of CPSs are usually formed by long test inputs that aim at mimicking real-world scenarios. As a result, when a failure is exhibited, it is highly important to isolate the failure-inducing inputs to provide the developers with the minimal test input. This allows reducing debugging costs by (1) reproducing the failure in the minimal time and (2) reducing the test coverage of the system, making the fault localization easier. In this paper we adapt the well-known delta debugging algorithm to isolate the failure-inducing inputs of CPSs modeled in Simulink. By means of three Simulink models, we analyzed whether Delta Debugging is effective enough to isolate failure-inducing inputs in CPSs.**

***Index Terms*—Simulink Models, Debugging**

I. Introduction

Cyber-Physical Systems (CPSs) combine digital cyber technologies with physical processes [1]. The software of such systems is embedded in a real-time target, which allows for interacting with the physical environment through sensors and actuators. Simulation is the driving technology to test the software of such systems, as it allows for raising the abstraction level at which testing is permitted [2]. However, executing tests in simulation is time consuming because the physical layer of the CPS is modeled through complex mathematical models [3]. For instance, in the industrial example reported in [4], when simulating autonomous vehicles, a simulation run is taken down to 10% of real-time factor (e.g., simulating 1 minute takes 10 minutes). Similar to testing, debugging is a cumbersome activity too, due to the same problems of requiring a long test execution time.

MATLAB/Simulink has become the leading tool for modeling and simulating CPSs [5]. The test inputs in simulation-based testing using Simulink are signals over time that stimulate the CPS's model [3], [5]. Typically, such test inputs are long in time because (1) they intend to reproduce realistic scenarios of the CPS and (2) in CPSs the software system interacts with the physical layer and the environment in which the system operates, thus, the system might need to get into certain states by means of these interactions. This makes debugging hard due to two main reasons. Firstly, the test cases are long, and developers might need to wait long times every time they intend to reproduce a failure. Secondly, these long test inputs might lead to large test coverage, which can result on difficulties when aiming to localize the fault by using state-of-the-art techniques (e.g., Spectrum-based Fault Localization [6], [7]).

This problem might be partially solved simplifying the test cases by automatically isolating the difference that causes the failure. In this paper, we adapt the technique Delta Debugging [8] to the context of CPSs. This way, CPS developers are provided with the minimal test input to reproduce the fault. Specifically, we adapt this algorithm for CPSs modeled in MATLAB/Simulink. To assess the effectiveness of this technique, we carried out an empirical evaluation by using three Simulink models.[1] Furthermore, we discuss the limitations of our approach and open challenges to adapt the technique in other kind of CPS simulators.

II. Delta Debugging Algorithm for Simulink

Context: Simulation is referred to the process of creating a digital model to predict its performance in the real world. CPS modeling and simulation tools (e.g., Simulink) are dataflow models, where each model contains a set of blocks [9]. Each block accepts data through its inputs and may pass output through its output ports after performing a set of operations (e.g., mathematical, logical, etc.). In our context, we limit the adapted version of the Delta Debugging algorithm to Simulink-like models, where a test case is referred to a set of input signals stimulating the model.

Formalization: Let $SM = (I, O)$ be a simulation model, where $I = \{i_1, i_2, ..., i_N\}$ is a subset of inputs and $O = \{o_1, o_2, ..., o_M\}$ is a subset of outputs [10]. Each input and

[1]Both, the experimental material and the algorithm are publicly available in Zenodo: https://zenodo.org/record/5841608#.Yd783WjMKUk

output of the simulation model is a signal (i.e., a function of time), which is stored as a vector where elements are indexed by time [10]. The simulation time (T) is divided into a set of equal sample time steps (ΔT) [10]. A signal (sig) is a function in time of a set of k number of observed simulation steps (i.e., $sig : \{0, \Delta T, 2 \times \Delta T, ..., k \times \Delta T\}$). For instance, a simulation of 10 seconds (i.e., T=10), with a sample time of 0.05 seconds (i.e., ΔT = 0.05) would have a total of 10/0.05+1 (i.e., k=201) simulation steps. The lower the sample time is, the higher the precision of the simulation. However, the time required to simulate the system will also be lengthier. In our study, for each model being tested, we consider a fixed simulation sample time (i.e., the same for all the test cases).

Isolation algorithm: Algorithm 1 describes the adapted version of the Delta Debugging algorithm to the context of CPSs modeled in Simulink. As input, it receives the initial failure inducing Test Input (TI), and its Failing Time (FT), which indicates the exact time at which the Oracle returned the failing verdict. As output, it provides TI', which corresponds to the minimal failure-inducing test input. This is obtained by following the next procedures. In the first step (Line 1, Alg. 1), the algorithm splits the test input so that the test execution stops by the time it fails. Figure 1 illustrates this first function for a test input of a single signal taking 8 seconds and with a step size of 1 second, where the Failing Time (FT) is 6. As can be seen, the test input is reduced from 8 to 6 seconds, which reduces 2 simulation steps out of 9.[2] The provided Test Input (TI') will fail, because from second 0 to second 6 of the test input, everything remains the same as TI.

The following process is to Split this Test Input (TI') by minimizing it (Line 2), obtaining TI_{NEW}. In such case, the test input signal from second $\lceil FT/2 \rceil$ to FT is taken (in the example from Figure 1, from second 3 of TI' to second 6). After, the algorithm enters into a while loop that follows the following process. First, it executes the test TI' in MUT, which returns a verdict.[3] If the verdict reveals there is a Failure, we assign the test input in TI_{NEW} to TI' (Line 6), as its size (i.e., number of simulation steps) is less, and thus, the test input is more isolated. If the verdict does not reveal any failure, the test input is splitted by maximizing it (Line 9), which is done by the *splitMaximizing* function, which takes as input parameters TI' and TI_{NEW}. Figure 1 depicts an example of the returned test input by this function when considering a test input of 7 simulation steps and another one of 4. As can be seen, the function returns the last $\lceil (size(TI') + size(TI_{NEW}))/2 \rceil$ steps of TI', $size$ being the number of simulation steps of the test case. This process is iteratively repeated until the simulation steps of TI' and TI_{NEW} are the same.

III. Preliminary Evaluation

This section explains the preliminary evaluation we carried out to assess the performance of delta debugging for isolating faults in Simulink models. To this end, we aimed at answering the following two Research Questions (RQs):

- *RQ1 – To what extent can Delta Debugging isolate failure-inducing inputs in CPSs when considering the test input size?* Since in CPSs test cases are usually long, this RQ aims at analyzing the reduction rate of the test inputs provided by the algorithm. From the practical perspective, this would provide the developers with the minimal test case to reproduce the fault.
- *RQ2 – To what extent do the test inputs provided by Delta Debugging reduce test coverage?* In CPSs test cases are usually long and they achieve a relatively high coverage. This RQ aims at investigating to what extent the test input provided by the algorithm reduces test coverage. From the practical perspective, this would help narrowing down the search space to localize the fault.

Algorithm 1: Delta Debugging for Simulink Models

```
   Input: MUT //Model Under Test
   TI //Initial failure inducing test input
   FT // Failing Time
   Output: TI' //Minimized input signal over time
1  TI' = split(TI,FT);
2  TI_NEW = splitMinimizing(TI');
3  while size(TI') ≠ size(TI_NEW) do
4      Verdict=executeTest(TI_NEW, MUT);
5      if Verdict == Failure then
6          TI'=TI_NEW;
7          TI_NEW = splitMinimizing(TI');
8      else
9          TI_NEW = splitMaximizing(TI',TI_NEW );
10     end
11 end
```

A. Experimental set-up

1) Studied models: We used three study models involving three open-source Simulink models. Table I summarizes the key characteristics of these three models, which have been used in other prior studies [3], [11], [12]. The first model involves a two tanks system where a controller regulates the incoming and outgoing flows of the tanks. The second model is related to a model of a Cruise Controller (CC). The third model is an open source industrial model developed by Bosch [13], which involves an Electro Mechanical Break (EMB). We chose two relatively large model (i.e., TwoTanks and EMB) and a smaller one (i.e., CC) to see how the algorithm performs with models of different characteristics.

TABLE I: Key characteristics of selected models.

Model	# of Blocks	# of Inputs	# of Outputs	# of Mutants
TwoTanks	498	11	7	34
CC	31	6	2	20
EMB	315	1	1	17

[2] Notice that second 0 also computes as simulation time

[3] We assume that a test oracle is available

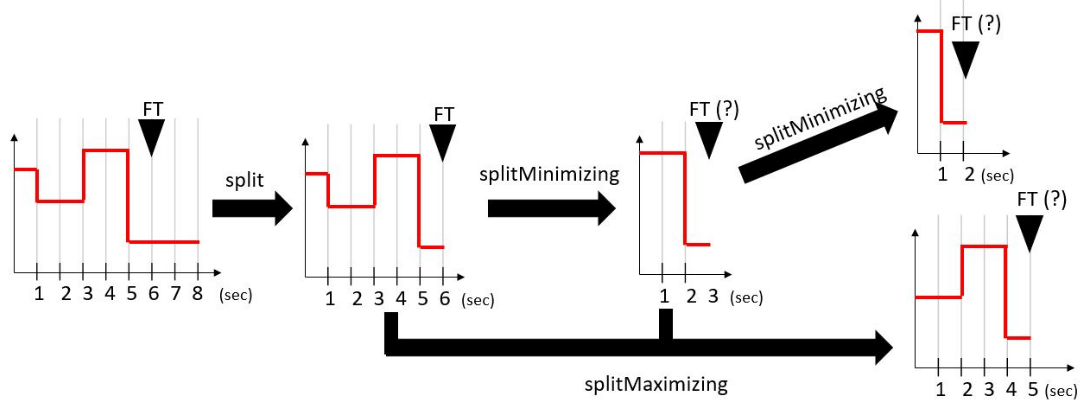

Fig. 1: The different test input transformations that are performed by our algorithm for a test input of a single signal and a duration of 8 seconds. For illustration purposes, we consider a sample time of 1 second, having a total of 9 sample times. FT refers to the time at which the system reveals a Failure.

2) Fault seeding and test generation: We used mutation testing to simulate the faults in the CPSs' models. Specifically, the mutants generated by Arrieta et al. [3] were re-used. These mutants used the operators by Hanh et al., for Simulink models [14]. The Delta Debugging algorithm assumes the availability of a test oracle to determine the test verdict. As we used mutation testing, for each model we employed the original Simulink model and compared whether the outputs differed with respect to the mutants. Thus, as in traditional mutation testing, we cataloged the verdict of a test case as fail if the outputs from the mutant and the original Simulink model differed, and as passed otherwise.

We generated a total of 25 test cases per model. This was because we wanted to have longer test cases than those used by Arrieta et al. [3]. These test cases were randomly generated by adapting some functions defined in a prior work [12]. After generating these tests, we executed them in the mutants for pairing the failing test cases with their detected mutants. These pairs were used in our evaluation to assess the proposed adaption of the delta debugging algorithm. For the TwoTanks model, 79% of the mutants were detected, for the CC all of them and for the EMB 88% of the mutants. In total, we had 131 pairs of test cases and mutants for the TwoTanks model, 135 pairs for the CC model and 244 pairs for the EMB. All these pairs were the inputs for the Delta Debugging algorithm. Notice that in this classification, we filtered those pairs where the tests detected the mutant at second 0, and therefore, isolation algorithm was not necessary.

3) Evaluation Metrics: We used two evaluation metrics to answer the RQs. On the one hand, the **Test Input Reduction Ratio (TIRR)**, which measures the test input reduction rate obtained by the delta debugging algorithm. We used two variants of this metric: firstly, the test case reduction ratio with respect to the original test case ($TIRR_{orig}$), which measures the test execution time difference between the initial Test Input (TI) and the Test Input provided by the delta debugging algorithm (TI_{NEW}). Secondly, the test case reduction ratio with respect to the original failing time ($TIRR_{ft}$), which measures the percentage of time reduction obtained by the Test Input provided by the delta debugging algorithm until the failure is detected with respect to the time taken by the original Test Input (TI) to fail (i.e., FT). The higher these metrics, the higher the performance of the proposed isolation algorithm, as it means that a higher isolation has been achieved.

On the other hand, we used the **Test Coverage Reduction Ratio (TCRR)**, which measures the reduction percentage obtained by the delta debugging algorithm. This metric is also important because the lower the test coverage is for the isolated test input, the lower the number of lines of code (or Simulink blocks in this case) that need to be checked by the debugger. We used two coverage metrics, which were accessible through the MATLAB API to measure the coverage on Simulink models: the Decision Coverage and the Condition Coverage. Thus, we have also divided this metrics into TCRR for Decision Coverage ($TCRR_{DC}$) and TCRR for Condition Coverage ($TCRR_{CC}$). The higher these metrics, the higher the performance of the proposed isolation algorithm, as it means that less Simulink blocks are exercised to reproduce the fault, thus, reducing the search space for its localization.

B. Analysis of the Results and Discussion

Table II summarizes the obtained results and Figure 2 shows the distribution of the different metrics for each of the models. As can be seen, the test input reduction ratio was quite high when considering the original test case (i.e., $TIRR_{orig}$). The average ranged from 0.75 to 0.98 depending on the model. This ratio was reduced when considering the original failing time (i.e., $TIRR_{ft}$). Specifically, while the delta debugging algorithm managed to have a reduction rate of 0.79 for the CC model and 0.86 for the EMB model, the average reduction rate for the TwoTanks model was 0.34. Nevertheless, in this case, the standard deviation was higher, meaning that at some cases, the reduction was high too. A possible reason behind these

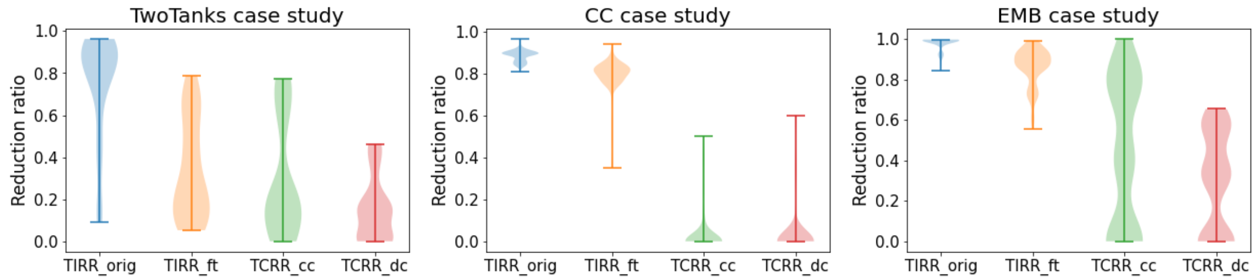

Fig. 2: Distribution of the test input and test coverage reduction ratio obtained for the selected three models

TABLE II: Summary results of the reduction ratios for the performed experiment (Mean (μ), Median (m) and Std. Dev. (σ))

	TIRR_orig			TIRR_ft			TCRR_cc			TCRR_dc		
	μ	m	σ	μ	m	σ	μ	m	σ	μ	m	σ
TwoTanks	0.75	0.84	0.25	0.34	0.26	0.23	0.30	0.17	0.28	0.16	0.15	0.15
CC	0.88	0.89	0.03	0.79	0.81	0.07	0.03	0.00	0.10	0.04	0.00	0.14
EMB	0.98	0.99	0.03	0.86	0.87	0.10	0.44	0.40	0.37	0.29	0.31	0.24

results might lie on the complexity of the TwoTanks model, where more logic is involved, and more test steps might be required to trigger some failures. Thus, the first RQ can be answered as follows:

> The overall test input can be significantly reduced, between 75 and 98% on average for the models used in our evaluation. However, this reduction might not be that significant when considering the time to reproduce the fault. While two models showed an average reduction between 79 to 86%, another one showed only an average reduction of 34%. This could be due to the complexity of this model.

To answer RQ2, we measured the test coverage reduction ratio by considering two coverage metric (i.e., decision and condition coverage). The reduction ratio was higher in the large models (i.e., TwoTanks and EMB). When considering the condition coverage, the average reduction rate for the codition coverage was 0.3 for the TwoTanks model and 0.44 for the EMB. The reduction rate for the decision coverage was lower, 0.16 for the Two tanks model and 0.31 for the EMB. Nevertheless, for the CC model, on average, the reduction rate was marginal for both metrics, although it showed a high standard deviation, meaning that at some cases, the reduction was high. The hypothesis behind these results is that since the CC model is small, it is easy to obtain a high test coverage even when the test inputs are minimal. Subsequently, we can answer the second RQ as follows:

> Based on our evaluation, for large models, the average coverage reduction rate is modest (on average, 30 and 44% of reduction for the condition coverage, 16 and 29% for the decision coverage in the case of our models). However, if the model is small, the reduction ratio is marginal.

IV. Open Challenges and Limitations

In this paper we have adapted the delta debugging algorithm for the context of CPSs modeled in Simulink. This adaption considers the idiosyncrasy of test cases being signals. Nevertheless, another particularity of CPSs when compared to general-purpose software is its tight interaction with the environment. By means of a preliminary evaluation based on three different models, we have shown that in some cases, this approach is able to provide a significant reduction rate on the time for reproducing the fault. However, in some models (e.g., TwoTanks), the reduction was not that significant. A similar thing happened with test coverage, which the maximum average reduction was 44% for the condition coverage and 29% for decision. Towards this direction, we identified three main challenges.

Challenge 1 – To consider the environment to further help the isolation of test inputs: To trigger a failure, the system might need to enter in certain state, which might depend on the environmental status in which the CPS operates. For instance, in the TwoTanks model, one of the tanks might need to be filled in order to trigger the fault, therefore, needing this scenario to be modeled by the test case. Our hypothesis is that if we monitor the environment and have access to changing it (e.g., changing the level of the tank), the failure-inducing input can be reduced even more.

Challenge 2 – To consider further states of the system: Similar to the previous point, it might be possible that the system depends on other states of the system to trigger the fault. These states are not usually changeable from the outside, and therefore, different test sequences signals might be necessary in order the CPS to enter those states. A potential solution could be to monitor system states prior these to fail. These states could be later obtained by a search algorithms which generates test cases in order the system to get into them. After, some of the test sequences of the original test case would be executed to trigger the fault. A challenge for this

case could be that (1) domain knowledge is needed to extract relevant system states and (2) the search approach might suffer scalability problems because CPSs are compute-intensive [12].

Challenge 3 – Other kind of simulators: This study is centered on those CPSs that are modeled in Simulink. Nevertheless, other simulators are also used for testing CPSs. Simulink is based on signals, but other CPS simulators are based on events over time. Different strategies need to be considered to adapt the delta debugging algorithms to these simulators (e.g., iterate based on events or based on the time that events happen). We believe that further investigation is required to isolate faults of CPSs modeled in other simulators.

V. Related Work

Different techniques have been proposed in the last few years in the context of automated debugging [7], including fault localization and test input reduction. For the latter, the Delta Debugging algorithm [8] has shown to be an effective technique to isolate the failure inducing inputs in different type of systems. However, CPSs face several idiosyncrasies when compared to other software systems, such as the tight interaction of the software with the physical part of the CPS and the interaction of the CPS with its environment. In this paper we have adapted the algorithm to the context of CPSs modeled in Simulink and analyze its performance, which, to the best of our knowledge, has not been tackled before.

Debugging of CPSs modeled in Simulink has also been studied in the last few years. Liu et al., focused on fault localization [6], [15]. Deshmukh et al., aimed at localizing faults provoked by missconfigurations in the context of Simulink models by using Spectrum-based Fault Localization [16]. Bartocci et al. [17] proposed CPSDebug, which combines testing, specification mining, and failure analysis, to automatically explain failures in Simulink models. All these areas are different to what we are tackling in this paper, which is centered on the automated isolation of failure-inducing inputs.

VI. Conclusion and Future Work

In this paper we proposed an adaption of the Delta Debugging algorithm for the isolation of failure-inducing inputs in CPSs modeled in Simulink. A preliminary evaluation shows that the reduction rate based on this technique might be large in some cases, but not in others. We summarize in a set of challenges potential possibilities to further improve this technique. In the future, we would like to extend this work from different perspectives. Firstly, we would like to compare the performance of the algorithm in terms of running time when compared with other baselines. Secondly, we would like to apply this technique to other CPS simulators that are based on events. Lastly, we would like to explore novel strategies that consider the system environment to further reduce the failure-inducing inputs in the context of CPSs.

Acknowledgment

Project supported by a 2021 Leonardo Grant for Researchers and Cultural Creators, BBVA Foundation.

References

[1] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13 – 28, January 2011.

[2] L. Briand, S. Nejati, M. Sabetzadeh, and D. Bianculli, "Testing the untestable: Model testing of complex software-intensive systems," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. ACM, 2016, pp. 789–792.

[3] A. Arrieta, S. Wang, U. Markiegi, A. Arruabarrena, L. Etxeberria, and G. Sagardui, "Pareto efficient multi-objective black-box test case selection for simulation-based testing," *Information & Software Technology*, vol. 114, pp. 137–154, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.06.009

[4] C. Gladisch, T. Heinz, C. Heinzemann, J. Oehlerking, A. von Vietinghoff, and T. Pfitzer, "Experience paper: Search-based testing in automated driving control applications," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 26–37.

[5] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Test generation and test prioritization for simulink models with dynamic behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 9, pp. 919–944, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2018.2811489

[6] B. Liu, S. Nejati, L. C. Briand *et al.*, "Effective fault localization of automotive simulink models: achieving the trade-off between test oracle effort and fault localization accuracy," *Empirical Software Engineering*, pp. 1–47, 2018.

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

[8] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[9] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 981–992.

[10] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous simulink models," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 595–606.

[11] R. Matinnejad, S. Nejati, and L. C. Briand, "Automated testing of hybrid simulink/stateflow controllers: Industrial case studies," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 938–943. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117770

[12] C. Menghi, S. Nejati, L. Briand, and Y. I. Parache, "Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 372–384.

[13] T. Strathmann and J. Oehlerking., "Verifying properties of an electro-mechanical braking system," in *In 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, 2015, pp. 49–56.

[14] L. T. M. Hanh, N. T. Binh, and K. T. Tung, "A novel fitness function of metaheuristic algorithms for test data generation for simulink models based on mutation analysis," *Journal of Systems and Software*, vol. 120, no. C, pp. 17–30, 2016.

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

[16] J. Deshmukh, X. Jin, R. Majumdar, and V. Prabhu, "Parameter optimization in control software using statistical fault localization techniques," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 220–231.

[17] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Ničković, "Cpsdebug: Automatic failure explanation in cps models," *International Journal on Software Tools for Technology Transfer*, pp. 1–14, 2021.