FIM: Fault Injection and Mutation for Simulink

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

Leonardo Mariani
leonardo.mariani@unimib.it
University of Milano-Bicocca
Milan, Italy

Dejan Ničković
Dejan.Nickovic@ait.ac.at
AIT Austrian Institute of Technology
Vienna, Austria

Drishti Yadav
drishti.yadav@tuwien.ac.at
TU Wien
Vienna, Austria

ABSTRACT

We introduce FIM, an open-source toolkit for automated fault injection and mutant generation in Simulink models. FIM allows the injection of faults into specific parts, supporting common types of faults and mutation operators whose parameters can be customized to control the time of fault actuation and persistence. Additional flags allow the user to activate the individual fault blocks during testing to observe their effects on the overall system reliability. We provide insights into the design and architecture of FIM, and evaluate its performance on a case study from the avionics domain.

Tool package and demo: https://gitlab.com/DrishtiYadav/fimtool, https://youtu.be/0EJri93Y_Gg

CCS CONCEPTS

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

KEYWORDS

Cyber-physical systems, Fault injection, Model-based Development, Mutation, Simulink, Testing.

ACM Reference Format:
Ezio Bartocci, Leonardo Mariani, Dejan Ničković, and Drishti Yadav. 2022. FIM: Fault Injection and Mutation for Simulink. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558932

1 INTRODUCTION

Safety-critical cyber-physical systems (CPS) must be reliable both in normal and unexpected circumstances. Hence, it is essential that such CPS comply with relevant industrial standards (such as ISO 26262, IEC 61508, etc.) and that they conform to desired safety requirements.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3558932

To cope with the complexity of modern CPS, researchers and practitioners have adopted a model-based development (MBD) associated with a light verification paradigm based on simulation and testing. The eco-system centered around MathWorks Simulink® is today the *de-facto* standard for MBD of CPS. In the last decade, the formal methods and control theory communities have developed more principled approaches to simulation-based verification, such as falsification testing [11] (this thread of research resulted in many publications, we specifically mention papers on this topic that appeared in the recent past [1, 2, 18, 19]).

While methods like falsification testing have proved successful and effective in finding bugs in CPS designs, we observe that these methods are often evaluated on a small number of examples and in an *ad-hoc* fashion. The typical approach to systematically evaluate a testing strategy is to analyze its effectiveness in detecting bad system behaviors in the presence of faults [4]. This is achieved via fault injection [3] and mutation testing [8], activities recommended by industrial safety standards, especially in safety-critical domains [13]. The main prerequisite for large-scale mutation testing evaluations is a mechanism for injecting faults of different types into the system model in an automated and programmatic fashion, without human intervention during the injection process. To the best of our knowledge, there is no fault injection solution for Simulink that satisfies all the identified requirements, which explains the lack of systematic experiments of CPS testing approaches.

We remedy this situation by introducing FIM (**F**ault **I**njection and **M**utation engine), an open-source tool for injecting faults into Simulink models that satisfies the above requirements. FIM provides the following features, which can all be configured programmatically:

- A rich fault model, including sensor, hardware and network faults, as well as a library of mutation operators;
- Systematic injection of a fault in different parts of a model:
  - generating multiple copies of the original model, with one fault injected per model copy, or
  - generating a single copy of the original model, with all faults injected into that copy;
- Restriction of fault injection to specific parts of the model;
- Dynamic activation and deactivation of faults.

Related work. A large body of fault injection tools and techniques is available according to the fault type, the System-under-test (SUT), and the injection method [7, 9, 10, 15, 17, 20]. In particular, there is a plethora of prior work on tools for fault injection in Simulink models [5, 12, 14, 16], but each has some caveats and limitations. MODIFI [16] and ErrorSim [14] have a very limited choice of fault types and are not publicly accessible. SIMULTATE [12] offers an interactive user interface using Python and MATLAB®. Although practical for beginners, an interactive interface can be a blocking factor for scalable fault injection experiments that may require injecting hundreds or thousands of faults. Another related model-based fault injection method [5] supports the injection of typical faults, but lacks an automated support for fault block placement in the SUT.

Compared to existing tools, FIM (i) integrates a richer variety of faults/mutations (see Section 2), (ii) supports automated fault injection via a transparent user interface that works directly in the MATLAB® environment without any additional setup, and (iii) provides scalable fault injection, allowing the design of experiments via configuration files that can be processed, generating a large number of mutants in seconds.

Paper organization. In Section 2, we provide insights into the architecture, key functionalities and the workflow of FIM. Section 3 summarizes the usage of the tool. In Section 4, we evaluate the performance of the tool on an example from the aerospace domain with some experimental results, and we conclude in Section 5.

2 FIM: DESIGN AND IMPLEMENTATION

We first pinpoint several conceptual problems encountered during the design and implementation of FIM, necessary to motivate and understand its features:

- *Specificities of Simulink*: Prior to designing the tool, we identified the complexities in handling Simulink models and thoroughly understood the programmatic editing of model components, from lines to blocks. We also comprehended the fundamentals of masking in Simulink prior to designing a rich fault injection library with parameterized blocks.
- We tried several injection strategies, before implementing the injection mechanism as the addition of new blocks, since it guarantees the highest level of control with no issues in its capability to observe and alter variable values.
- We focused on the possibility to do large scale experiments, carefully designing an interface that does not require interactions during experiments.
- We considered several strategies to deal with a large set of injected faults, finally supporting two complementary strategies: generating many models with one fault each and generating one model with all the faults included.

We now guide the reader through the tool architecture and workflow, highlighting its key features. Figure 1 shows the architecture of FIM, which consists of three main components: (1) a *Fault Library*, (2) a *Fault Injection Module* and (3) a *Fault Configuration*.

2.1 Fault Library

The *Fault Library* is a custom Simulink library browser consisting of different parameterized blocks for different types of faults and mutation operators (listed in Table 1). FIM allows the user to extend the library browser with other faults/mutation operators. The insertion or replacement of blocks depends on the type of fault

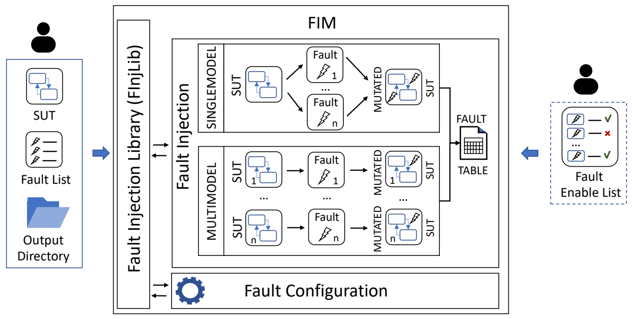

Figure 1: Overview of FIM.

specified by the user in the list of desired mutations *Fault List*. The fault blocks corresponding to 'Faults' in Table 1 indicate insertion of the fault block, while those related to 'Block mutations' indicate replacement of the relevant block by the specified block mutation operator. Some faults and block mutations are illustrated in Figure 2.

The faults and mutation operators supported by FIM are realistic, typical and commonly observed in practice, according to the available literature. In particular, the faults `Stuck-at`, `Package drop`, `Bias/Offset`, `Noise`, `Time delay` and `Bit flips` are derived from ErrorSim [14], MODIFI [16], SIMULTATE [12] and FIBlock [5]. The faults `Negate`, `Invert` and `Absolute` and Block mutation operations `ROR`, `LOR` and `P2S` are derived from SIMULTATE [12]. `S2P` and `ASR` are standard arithmetic operator replacements used in mutation testing.

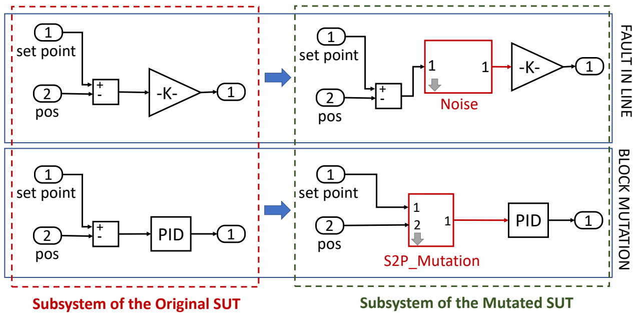

Figure 2: Illustration of some faults/mutations in a SUT (the injected fault blocks are highlighted in red).

Each fault block is implemented as a masked subsystem with different block parameters. Each fault block has a flag *FIEnableflag* that enables the user to turn it on and off. Since FIM supports deterministic faults, all fault blocks have the parameter *FaultOccurenceTime* to specify the time of fault actuation. The block parameter *FaultEffect* specifies the fault effect which governs the fault persistence time. FIM offers two types of fault effects: 1) Infinite time: The fault block produces erroneous output starting from the *FaultOccurenceTime* until the end of the simulation, and 2) Constant time: Fault occurs at *FaultOccurenceTime* and persists for the duration specified by the block parameter *FaultDuration*. The blocks corresponding to `Noise`, `Bias/Offset` and `Package Drop` have an additional block

Table 1: Faults and mutation operators.

Type	Name	Description
Faults	Negate	changes a signal u to $-u$
	Invert	inverts a non-zero signal u to 0; if u is 0, makes 0 to 1
	Stuck-at 0	makes the signal value zero
	Absolute	changes a signal u to $\|u\|$
	Noise	adds a band limited white noise to the input signal based on specified fault value (i.e., noise power)
	Bias/Offset	adds a predefined +ve or −ve offset (bias) value to input
	Stuck-at	the signal value stucks at the last correct value before fault occurrence
	Time Delay	introduces a delay of specified duration
	Bit Flip	Bitwise NOT operation on boolean signal
	Package Drop	replaces the input by the specified fault value
Block mutations	ROR	Relational Operator Replacement
	LOR	Logical Operator Replacement
	S2P	Sum to Product mutation
	P2S	Product to Sum mutation
	ASR	Arithmetic Sign Replacement (for 2-Input Sum Block)

parameter *FaultValue* for fault value adjustment. The blocks related to `ROR`, `LOR` and `ASR` have another parameter *OperatorNum* to select the relevant operator. In fact, there is a mapping between various operators and the operator numbers (more details can be found in the tool package).

2.2 Fault Injection

The *Fault Injection Module* is responsible to inject faults based on the desired mutations (*fault_list*). For SINGLE-MODEL case (*i.e.*, mutations with Single model), FIM first creates a copy of the SUT such that all the desired faults are injected in the copied file. Conversely, for MULTI-MODEL case (*i.e.*, mutations with Multiple models), FIM creates a copy and injects a single fault in the copied file. Then, this process continues until all the desired faults are injected *s.t.* there are multiple copies of the SUT, one fault injected per copy.

The advantage of SINGLE-MODEL over MULTI-MODEL mode is twofold: (a) a single model is generated and compiled, and (b) a mutated SUT with all mutants in a single model permits the user to investigate the simultaneous effect of multiple faults. The advantage of MULTI-MODEL over SINGLE-MODEL is the simplicity and small size of the individual models generated.

The *Fault Injection Module* exploits the basics of programmatic editing of Simulink models to inject faults in the mutated SUT. In essence, it stores the information of line handles, block handles, source-destination blocks (and their respective ports) in a persistent cache. This information is then used to inject the faults in the copied file. While injecting faults, the *Fault Injection Module* automatically retains the logging information of the signal(s) and improves the model layout to abide by modeling guidelines and enhance readability of models.

Once all the desired faults are injected, the *Fault Injection Module* generates a `fault table` (.xlsx file) to provide detailed information of the injected fault blocks. More precisely, the `fault table` provides the following details: (i) name of the mutated SUT, (ii) the unique name of the fault block added in the SUT, (iii) the subsystem within which the fault is injected, (iv) the type of the fault or block mutation operator, (v) full names of the source-destination blocks corresponding to the line in which the fault is injected, and (vi) the respective port numbers of the source-destination. Such a detailed description of the injected faults will assist the user to identify the exact location of the fault block in the mutated SUT. Note that each injected fault block has a unique name, which is later utilized by the *Fault Configuration* component to control its activation.

2.3 Fault Configuration

Based on the generated `fault table`, the user can now configure the faults that must be used in an experiment. FIM allows the user to input another configuration file: *fault_enable_list* which is a table (.csv or .xlsx file) with fields to specify the block(s) to be activated along with the relevant fault parameters. In accordance with the *fault_enable_list*, the *Fault Configuration* component turns on the respective fault blocks in the mutated SUT and configures them for the specified fault parameters. The fault block (to be activated) is specified by its unique identifier (fault number) in the `fault table`.

3 TOOL USAGE

As a special requirement, FIM requires a licensed MATLAB® and Simulink® installation, which must be provided by the user. We have implemented FIM in MATLAB® R2020b to provide a command-line interface which permits the user to invoke the tool from the MATLAB® command prompt. In both SINGLE- and MULTI-MODEL modes, the tasks of *Fault Injection* and *Fault Configuration* are governed by different customized functions. In essence, user-defined configuration files and output directory supervise the behavior of FIM in its entirety.

Injecting faults. As inputs, FIM takes a configuration file and the output directory. Usually, the configuration file includes details of the SUT and the *fault_list*. More precisely, the *fault_list* is a table with fields to specify the target fault location and the desired type of fault/mutation. The target fault location can be specified using the names of source/destination/parent blocks or hierarchy levels of the SUT, thereby giving the freedom to the user to control the fault

Table 2: Fault Injection (mutant generation) time for AECS.

Target Location and Fault details		***n***	**Fault Injection time (in seconds)**	
Target	Fault type		SINGLE-MODEL	MULTI-MODEL
'Controller' subsystem with parent block 'Left Control Laws'	Bias/Offset	20	16.38	53.53
Plant/Actuators/Right Outer Hydraulic Actuator	Negate	17	14.05	46.31
Plant/Actuators/Left Outer Hydraulic Actuator/Hydraulic Actuator	ASR	2	3.73	11.02
Controller/Right Control Laws/IO Control Law	S2P	1	1.83	3.58
Plant/Actuators/Right Inner Hydraulic Actuator/Hydraulic Actuator	P2S	3	2.58	7.92
Controller/Right Control Laws/Subsystem	ROR	2	3.05	6.99
Total		45	41.62	129.35

injection space. As outputs, FIM either produces a single mutated model or a set of mutated models depending on the mode selected, i.e., SINGLE-/MULTI-MODEL. Also, FIM emits the generated `fault table` to the output directory specified by the user.

Configuring faults. After fault injection, the user can look into the generated fault table to identify the fault blocks which he wants to turn on. For the *Fault Configuration*, the user also needs to provide the *fault_enable_list*, which specifies the faults that must be activated in the next run and their activation strategy. As output, the selected fault blocks are switched on and the respective fault parameters are configured after the execution of appropriate command depending on the SINGLE-/MULTI-MODEL mode.

More details on the fault parameters and configuration parameters are available in our tool package. The *step-by-step demonstration* of FIM (with detailed commands) on two usage scenarios is also available online.

4 TOOL EVALUATION

We present a case study based on the Aircraft Elevator Control System (AECS) from the avionics-aerospace domain [6]. This Simulink model consists of various hierarchical subsystems including Controller, Plant, Sensors, etc. The model takes the 'Pilot Command' as the input variable which governs the two output variables (the positions of left and right actuators). Further, the model has different types of signals including real-valued, Boolean and enumerated—state machine—variables. A mutation in any of these signals might trigger a faulty behavior in the model.

To assess the performance of our tool, we conducted experiments on the AECS for both SINGLE- and MULTI-MODEL modes. Our experiments were performed on a MacBook Pro with Apple M1 chip, 16 GB RAM, macOS Big Sur with MATLAB® R2020b. We summarize the fault injection times, measured by averaging over ten independent runs, for the two modes in Table 2. Columns *Target* and *Fault type* respectively indicate the target fault location and fault type. Column *n* shows the number of faults injected. Note that (i) *n* depends on the structure of the target location, and (ii) the fault generation time depends on the fault type as well as the target location. The results show that the fault injection mechanism in SINGLE-MODEL is fast and computationally less expensive compared to MULTI-MODEL. FIM takes {0.92, 2.87} seconds, on average, to inject a single fault in the {SINGLE-, MULTI-} MODEL modes.

Based on our fault injection experiments, we report the following results: (1) Since MULTI-MODEL is more expensive, SINGLE-MODEL might be the best option when efficiency is important. (2) Nevertheless, SINGLE-MODEL is large and messy, and might be more difficult to visualize, debug and analyze. Thus, MULTI-MODEL may facilitate the analysis of the results of fault injection experiments. (3) Contrary to the MULTI-MODEL scenario, SINGLE-MODEL is useful to study interference among multiple faults, potentially activate at different times. (4) For synergistic benefits of both modes, the user could start with the SINGLE-MODEL and then switch to the MULTI-MODEL mode to analyze the fault effects, making the analysis more comprehensible and interpretive.

5 CONCLUSIONS

We presented FIM, a toolkit for the automated injection of faults and generation of mutants when dealing with Simulink models. A tester can control the activation of the fault blocks based on the task at hand. Further, a tester is free to adjust the fault parameters for rigorous testing of the SUT against failures to check its fault tolerance. We speculate that FIM can be an important step towards refining the overall process of verification of safety-critical systems.

Verification experts and testers can reuse and repackage our existing open-source toolkit according to their safety evaluation and testing needs. The users can update the tool by (i) making appropriate modifications in the automated scripts for fault injection and configuration, and (ii) enriching the fault library with additional faults operators as well as conditional and cascaded faults.

At present, FIM can automatically and efficiently generate a large number of models with faults injected, enabling scalable fault injection experiments. We are currently working on enriching the tool with features that can be used to support specific application scenarios, such as fine-grained testing options. Our future work includes an empirical study to investigate the cost and benefit of large-scale mutation testing in Simulink models. We envision the extension of the tool to fault localization and failure explanation in safety-critical systems. Future extensions may also integrate FIM with simulation-based verification engines that encapsulate formal approaches for safety analysis.

ACKNOWLEDGMENTS

This work has been supported by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien. This work has been also supported by the EMELIOT national research project, which has been funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).

REFERENCES

[1] Arvind S. Adimoolam, Thao Dang, Alexandre Donzé, James Kapinski, and Xiaoqing Jin. 2017. Classification and Coverage-Based Falsification for Embedded Control Systems. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, Cham, 483–503. https://doi.org/10.1007/978-3-319-63387-9_24

[2] Takumi Akazaki and Ichiro Hasuo. 2015. Time Robustness in MTL and Expressivity in Hybrid System Falsification. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, Cham, 356–374. https://doi.org/10.1007/978-3-319-21668-3_21

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (2004), 11–33. https://doi.org/10.1109/TDSC.2004.2

[4] Marco Bozzano and Adolfo Villafiorita. 2007. The FSAP/NuSMV-SA Safety Analysis Platform. *Int. J. Softw. Tools Technol. Transf.* 9, 1 (2007), 5–24. https://doi.org/10.1007/s10009-006-0001-2

[5] Tagir Fabarisov, Ilshat Mamaev, Andrey Morozov, and Klaus Janschek. 2021. Model-based Fault Injection Experiments for the Safety Analysis of Exoskeleton System. *CoRR* abs/2101.01283 (2021). arXiv:2101.01283 https://arxiv.org/abs/2101.01283

[6] Jason Ghidella and Pieter Mosterman. 2005. Requirements-based testing in aircraft control design. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*. MathWorks, USA, 5886. https://doi.org/10.2514/6.2005-5886

[7] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82. https://doi.org/10.1109/2.585157

[8] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[9] Maha Kooli and Giorgio Di Natale. 2014. A survey on simulation-based fault injection tools for complex systems. In *Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2014, Santorini, Greece, May 6-8, 2014*. IEEE, USA, 1–6. https://doi.org/10.1109/DTIS.2014.6850649

[10] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. 2016. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–55. https://doi.org/10.1145/2841425

[11] Truong Nghiem, Sriram Sankaranarayanan, Georgios Fainekos, Franjo Ivancić, Aarti Gupta, and George J. Pappas. 2010. Monte-Carlo Techniques for Falsification of Temporal Properties of Non-Linear Hybrid Systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control* (Stockholm, Sweden) *(HSCC '10)*. Association for Computing Machinery, New York, NY, USA, 211–220. https://doi.org/10.1145/1755952.1755983

[12] Ingo Pill, Ivan Rubil, Franz Wotawa, and Mihai Nica. 2016. Simultate: A toolset for fault injection and mutation testing of simulink models. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, USA, 168–173. https://doi.org/10.1109/ICSTW.2016.21

[13] Ludovic Pintard, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. 2013. Fault injection in the automotive standard ISO 26262: an initial approach. In *European Workshop on Dependable Computing*. Springer, Berlin, Heidelberg, 126–133. https://doi.org/10.1007/978-3-642-38789-0_11

[14] Mustafa Saraoğlu, Andrey Morozov, Mehmet Turan Söylemez, and Klaus Janschek. 2017. ErrorSim: A tool for error propagation analysis of simulink models. In *International Conference on Computer Safety, Reliability, and Security*. Springer, Cham, 245–254. https://doi.org/10.1007/978-3-319-66266-4_16

[15] Daniel Skarin, Jonny Vinter, and Rickard Svenningsson. 2014. Visualization of model-implemented fault injection experiments. In *International Conference on Computer Safety, Reliability, and Security*. Springer, Cham, 219–230. https://doi.org/10.1007/978-3-319-10557-4_25

[16] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson, and Martin Törngren. 2010. MODIFI: a MODel-implemented fault injection tool. In *International Conference on Computer Safety, Reliability, and Security*, Erwin Schoitsch (Ed.). Springer, Berlin, Heidelberg, 210–222. https://doi.org/10.1007/978-3-642-15651-9_16

[17] Saša Vulinovic and Bernd-Holger Schlingloff. 2005. Model based dependability evaluation for automotive control functions. In *Invited Session: Model-Based Design and Test, 9th World Multi-Conference on Systemics, Cybernetics and Informatics, Florida*, Vol. 7. IIIS, USA, 75–80.

[18] Zhenya Zhang, Ichiro Hasuo, and Paolo Arcaini. 2019. Multi-armed Bandits for Boolean Connectives in Hybrid System Falsification. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, Cham, 401–420. https://doi.org/10.1007/978-3-030-25540-4_23

[19] Zhenya Zhang, Deyun Lyu, Paolo Arcaini, Lei Ma, Ichiro Hasuo, and Jianjun Zhao. 2021. Effective Hybrid System Falsification Using Monte Carlo Tree Search Guided by QB-Robustness. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, Cham, 595–618. https://doi.org/10.1007/978-3-030-81685-8_29

[20] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. 2004. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.* 1, 2 (2004), 171–186.