FIM: Fault Injection and Mutation for Simulink

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

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

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

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.


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:


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.

This work is licensed under a Creative Commons Attribution 4.0 International License.

ESEC/FSE ’22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9415-0/22/11
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 (Fault Injection and Mutation 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 programatically:

• 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. SIMULATE [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 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], SIMULATE [12] and FIBlock [5]. The faults Negate, Invert and Absolute and Block mutation operations ROR, LOR and P2S are derived from SIMULATE [12]. S2P and ASR are standard arithmetic operator replacements used in mutation testing.

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
## 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. 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.
We present a case study based on the Aircraft Elevator Control System (AECS) from the avionics-aerospace domain [5]. 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 repack 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).

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

<table>
<thead>
<tr>
<th>Target Location and Fault details</th>
<th>Fault type</th>
<th>( n )</th>
<th>Fault Injection time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller subsystem with parent block 'Left Control Laws'</td>
<td>Bias/Offset</td>
<td>20</td>
<td>16.38, 53.53</td>
</tr>
<tr>
<td>Plant/Actuators/Right Outer Hydraulic Actuator</td>
<td>Negate</td>
<td>17</td>
<td>14.05, 46.31</td>
</tr>
<tr>
<td>Plant/Actuators/Left Outer Hydraulic Actuator</td>
<td>ASR</td>
<td>2</td>
<td>3.73, 11.02</td>
</tr>
<tr>
<td>Controller/Right Control Laws/I0 Control Law</td>
<td>S2P</td>
<td>1</td>
<td>1.83, 3.58</td>
</tr>
<tr>
<td>Plant/Actuators/Right Inner Hydraulic Actuator</td>
<td>P2S</td>
<td>3</td>
<td>2.58, 7.92</td>
</tr>
<tr>
<td>Controller/Right Control Laws/Subsystem</td>
<td>ROR</td>
<td>2</td>
<td>3.05, 6.99</td>
</tr>
</tbody>
</table>

Total: 45, 41.62, 129.35
REFERENCES


