Model-Implemented Hybrid Fault Injection for Simulink (Tool demonstrations)

Mehrdad Moradi[1,2][0000-0001-8748-069X], Bert Van Acker[1,2][0000-0002-3854-5159], Ken Vanherpen[1,2] and Joachim Denil[1,2][0000-0002-4926-6737]

[1] Antwerpen University, Prinsstraat 13, 2000, Antwerpen, Belgium
[2] Flanders Make, Oude Diestersebaan 133, 3920 Lommel, Belgium
{Mehrdad.Moradi, Bert.VanAcker, Ken.Vanherpen and Joachim.Denil}@uantwerpen.be

Abstract. The increasing complexity and certification needs of cyber-physical systems (CPS) requires improved methods of dependability analysis. Fault injection (FI) is an experimental-based way for safety analysis of a system which is mainly divided in *model-based, software-based* and *hardware-based* techniques. For safety analysis during model-based development, FI mechanisms can be added directly into models of hardware, models of software and/or models of the system. This approach is denoted as *model-implemented hybrid FI*. The availability of a modelling environment such as Simulink allows for early stage verification of FI experiments to analyze the correct behavior of the system-under-design. This results in a reduced time and cost by testing at early stages of the development process. This paper presents an automated framework to inject faults in the Simulink model. The framework is not only limited to injection at the model-in-the-loop (MiL) level but is also applicable at other approximation levels in model-based testing such as hardware-in-the-loop (HiL). The modeler instruments the model with FI blocks to specify which faults need to be injected and when they should be injected. This model is converted to a fault-injected model and a FI orchestrator that allows the FI experiment to run automatically. The framework is completely build upon the generative technique of model transformation, allowing it to be ported to other formalisms and tool environments.

Keywords: Cyber-Physical System, Model-Implemented Fault-Injection, Model Transformation, Safety Analysis.

1 Introduction

Humans use their cognitive abilities to make their lives more comfortable. By merging embedded systems with electro-mechanical systems, automation of time consuming or error prone tasks became available. This resulted in e.g. advanced automobiles, trains, medical technology and manufacturing plants. With the increase of market pressure and increased need for more features such as autonomous decision making, safety regulation and more efficiency in terms of power and cost, novel methods, techniques, and tools are required to create this next generation of CPS.

CPS are systems that combine computational and networking components together with physical components [1]. The computational part of a CPS has different components which have a predefined task that comprises e.g. controlling mechanical actuators, computing the new state of the system or monitoring the system context using a set of physical and virtual sensors. Designing a CPS therefore also requires a multi-disciplinary approach.

Because of this complexity and the huge impact on our daily lives, most of the current and future CPS are also dependable systems. Dependable system, and more specifically safety critical system, are systems that need to be reliable and resistant to failures, specially failures that can endanger or damage humans, environment or equipment. To keep the complexity of the system design at bay, engineers adhere to standardized processes to create more dependable systems. These standards define process constraints on all levels of the process such as design, development and testing, etc. to assure that the final product is safe and reliable [2]. An example of such standard is the ISO 26262 standard [3]. It is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production passenger cars. The ISO 26262 standard specifies that FI could be used in the development process. This recommendation raised the issue of the role of FI in the design phase, which is a difficult problem that has not been fully investigated.

FI is a well-studied verification technique that has been applied to many different targets, e.g. operating systems (OS), embedded software, and more recently at the model level. In safety engineering, the use of FI is twofold [4-5]:

- FI is used for testing different fault tolerant mechanisms that are embedded in the system. The safety standard is requirement driven, this means that all requirements have to be verified and that each line in the software can be traced to a (set of) requirement(s). To allow the testing of these fault-tolerance mechanisms, FI techniques at both the level of the hardware (or their models) and the software (and their models) are a necessity.

- Second, FI can be used as a supporting technique to hazard and risk analysis: It allows to identify the different fault modes during fault identification. By analyzing the traces of the FI, the propagation of errors can be seen. This leads to the detection of possible new failure modes and the cause for this failure.

However, this raises the question which faults have to be injected when and where, as it is impossible to inject all possible faults, at all different possible places, in all different scenarios and time windows. A structured technique to reason about a FI experiment is the FARM model [6]. The FARM model describes the set of **F**aults to be injected, the **A**ctivation of the faults, the **R**eadouts of the experiments and finally the **M**easures or analysis of the readouts. The conceptual FARM model can be applied at each level of the design process but as mentioned.

In this paper we model the FARM concept explicitly within a Simulink model such that a domain-expert can easily setup FI experiments without the need for other tooling. The contributions of the paper can be summarized as follows:

- Modelling of FI experiments at the model level using the same visual notation that the engineer is comfortable with.
- The generation of FI experimental setups for both MiL and HiL simulation.
- The consideration of real-time behavior of the application when instrumenting the model for HiL simulation such that the tasks do not overrun its real-time deadline. This is to avoid that faults emerge in the behavior of the system that cannot be traced to the experiments defined by the user.

The rest of the paper is organized as follows. Section 2 introduces the background needed to understand the contributions presented in this paper. Section 3 introduces the running example which is used to show the contribution of this paper. Section 4 shows the approach followed in this paper to instrument the model and create the experimental setups for a FI experiment. Afterwards, in Section 5 we discuss the approach. Section 6, gives an overview of the work related to this work. Finally, we conclude in Section 7.

2 Background

2.1 Fault injection

FI is a verification technique that has been available since the 80’s - early 90’s. Today, FI has been applied to many different targets: OS, middleware, web services, web servers, embedded systems, etc [2]. FI is an experimental approach to evaluate dependability. Dependability is particularly important when system/software being developed is safety critical, where failure can cause a serious hazard or even loss of life. FI techniques can be classified as physical or simulation-based. Based on the implementation of FI mechanisms, the techniques can be classified as hardware-implemented FI (HIFI) or software-implemented FI (SWIFI) techniques [7].

In hardware-based FI, faults are injected at the physical level by controlling the environment parameters. Real-life faults are emulated by injecting voltage sags, disturbing the power supply, heavy ion radiation, electromagnetic interference, etc. Radiation result in the change of data in parts of the memory like microprocessor internal register, program memory and data memory. Production defects and component wear-out can potentially increase latency to execute tasks, wrong measurements and loss of response (for example in sensors). Interface problems will create package loss, latency and even full loss of communication when a short-circuit occurs.

Software-based FI refers to techniques that inject faults by implementing it in the software. Different types of faults can be injected with software-based FI, for example register and memory faults, error conditions and flags, irregular timings, missing messages, replays, corrupted memory, etc. Data memory changes in source code lead to control flow errors [8]. Technically, faults are injected within the application or between the software application and the operating system (OS)/middleware. In case where the target is the OS itself, faults are embedded within the middleware/OS [9]. The faults within the application or middleware/OS are inserted either at compile-time or at run-time. Software implemented FI methods can be adapted to inject faults on various trigger mechanisms such as exception, traps, time-out, code-modification [10].

Finally, simulation-based FI involves constructing a simulation model of the system-under-test and adding faults in this model. In most of the literature, the system-under-test is the computational hardware. In that case, hardware description languages such as VHDL are injected with faults [7]. With the advent of CPS, the techniques of simulation-based FI are also applied to other systems and formalisms. Mostly, a simplified model of a CPS consists of a plant and controller which are connected to each other in feedback loop.

In safety engineering there needs to be a way to establish if a failure occurs (failure mode) when doing FI experiments. One way is to explicitly model the safety requirement of system. These safety requirement models can be co-simulated together with the FI experiments to verify the defined safety properties. We will assume that the safety properties of the system are modelled using an appropriate language that can detect these failures. An example of such a language is a temporal logic such as signal temporal logic (STL). Monitors for STL can automatically be generated for Simulink [11]. Another technique is to compare the trace of a correct system with the trace of the experiment run (where a fault is injected). If there is a difference between traces (using an error metric), the trace has to be further examined by the expert to define if this result in an unsafe or incorrect behavior.

2.2 Model-based techniques

Model-based systems engineering together with model-based design is gaining more interest for the design and development of software-intensive and CPS. This results in a development shift from hand-written code to models from which implementation code is automatically generated through model-to-text transformations (also known as code generation). Furthermore, various disciplines are involved in designing a CPS such as mechanical engineering, control engineering, software engineering, integration engineering [12].

The major advantage of using model-based techniques is that verification and validation steps can be frontloaded. This means that users can use their previous experience and/or other developer's knowledge during the development process of the new system-under-design. In traditional design processes, the verification and validation of the system behavior can only be done when all components are designed and integrated. Model-based techniques allow to check system-level properties much earlier in the design process.

Today, MATLAB/Simulink is a popular tool used in the design, simulation, and verification of software-intensive and CPS. Thanks to extensive automatic code generation facilities, developers often use Simulink as a control programming language based on the causal block diagram (CBD) formalism. Simulink uses two basic entities: blocks (with ports) and links. Blocks represent (signal) transfer functions, such as arithmetic operators, integrators, or relational operators. Links represent the time-varying signals shared between connected blocks. Fig. 1 shows the meta-model of a Simulink model. All models in the Simulink language follow the structural rules defined by the type model. The semantics of continuous-time causal blocks diagrams map to ordinary differential equations.

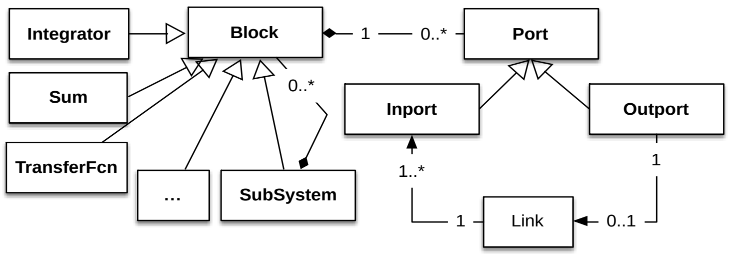


Fig. 1. Simulink meta-model

Model-based techniques are also applied for the testing of systems. Zander in [13] defines model-based testing as the technique where the test specification is derived in whole or in part from both the system requirements and a model that describe selected functional aspects of the system-under-test. In the literature different test execution platforms are defined where the integration with the real computational platforms is increasing. MiL is the first integration level where the control model is attached to a plant model. It is done for testing purposes. Software-in-the-loop (SiL) uses the implementation code in closed loop with the plant model for testing. Processor-in-the-loop (PiL) uses the production microcontroller to execute the control model in closed-loop with the plant model. Finally, HiL simulation, in which the real-time embedded system has interactive simulation with the plant model [14]. Because of its rapidness and high precision, it has been widely used in the testing and verification of embedded systems such as automobiles, ships and aircrafts [15-18].

2.3 Model transformation

With the advent of more models in the design of systems, a systematic way to manipulate models is required. During the process of design, models have to be converted between different representations (in possibly different modelling languages) that allow different features such as code generation, model checking, simulation, performance checking, etc. MTs systematically modify models. Rule-based MT languages work on typed, attributed, and directed graphs that represent the model. The transformation rule language can automatically be generated from the meta-model of the languages [19]. A transformation rule represents a manipulation operation on the represented model. Fig. 2 shows such a MT's schema. A rule consists of a left-hand side (LHS) pattern representing the pre-condition for the applicability of the rule. The right-hand side (RHS) pattern defines the outcome of the operation. A set of negative application condition (NAC) patterns block the application of the rule. A unique label in the pattern elements of the LHS, RHS, and NAC refers to the matched elements. The transformation execution decides the outcome of the rule based on these unique labels.

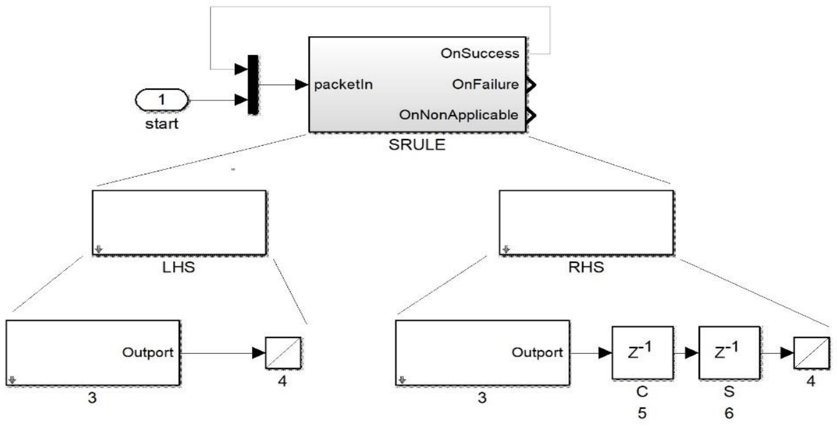


Fig. 2. Example of Simulink transformation

We design the Simulink MT in this work using the Simulink language based on the tool presented in [20]. The transformation language in this tool is based on the T-Core MT language framework [21]. The Simulink MT scheduling language is a simple branch-condition-loop language modelled in Simulink. The Simulink transformation language supports multiple scheduling blocks that have different effects on the model:

- ARULE: The atomic rule matches a single instance of the precondition pattern in the model and rewrites it.
- FRULE: The for-all rule matches all the instances of the precondition pattern in the model and rewrites all of them in one pass. This can create consistency issues when patterns are overlapping because part of the model is already rewritten.
- SRULE: The SRULE is similar to the FRULE but does not create consistency issues. It basically executes the ARULE in a loop until no more instances can be found in the model.

The schedule is shown at the top level. The schedule emulates an SRULE application using a simple loop with an ARULE. The schedule block contains two subsystems, one for the LHS and one for the RHS. Opening the subsystems allows us to specify the pre and post condition patterns. The pre-condition pattern matches all combinations of a constant and product block. The post-condition pattern changes the matched constant product block into a gain block (removing the product and constant block). The parameters of the gain block are set based on the attributes of the matched product and constant block. The color of the new gain block is set to red. The *Pre_InportOfBlock* and *Post_InportOfBlock* (as well as the *Pre-* and *Post_OutportOfBlock*) are abstract blocks. They explicitly refer to the input port of the block. To re-attach the post-condition pattern in the model to the previous (respectively next) blocks in the model, the MT language needs these constructs. Other abstract blocks can match any block, any block with an input port and/or output port, or even only discrete blocks [22].

3 Running Example

We will use a running example to show the contribution of this paper. The example is a case study in Simulink which implements a power window. The model contains the *DRV_Control*, *Object_DRV*, *Debounce_DRV*, *PW_DRV* and the *plant model* in a feedback loop as shown Fig. 3. *DRV_Control* and *Object_DRV* are blocks for providing system inputs. *Debounce_DRV* and *PW_DRV* are our controller. By using FI mechanisms, the safety properties of the control model will be verified. The verification of the plant model will not be examined in this paper.

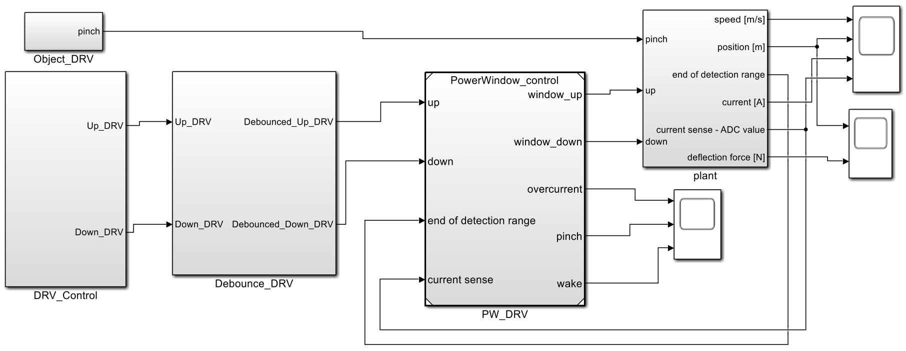


Fig. 3. Use case model

Our framework considers two possible instrumentation levels, MiL and HiL. For HiL experiments, we use a real-time hardware simulator, more specific the dSPACE SCALEXIO HiL box, which enables us to simulate the plant model and interact with the real embedded hardware. The controller is implemented on an Atmel board.

4 Fault Injection Framework

The goal is to inject faults into the executable model. This framework will be described in three main parts. First, we look into the fault library for models in Simulink. Second, we focus on the workflow and experiment orchestrator for MiL. Finally, we show the additional workflow steps when applying our framework in a HiL setting.

4.1 Fault library

By studying possible faults at the hardware level and software level in the literature [5,10], we came up with a list of faults that can be injected at the model level. Because most of these techniques are either for hardware or software, not all are applicable at the model level. For example, for hardware-based FI we cannot emulate power surge, bridging, sporous current and changing program memory or internal hardware component specification at the model level. Also, for software-based FI we cannot introduce compile time faults. However, we can cover other faults and inject them at the model level. We distinguish four categories (data faults, latency faults, open links, code insertion) at three levels of the computational stack (hardware, software and interface). Also, the faults have three different types of nature including permanent fault, temporary fault and intermittent fault. Permanent fault remains until end of each simulation, while temporary change the model in only a short time. Finally, intermittent fault is something between permanent and temporary and they repeat periodically.

- **Data**

Data changing comes from changes of data in memory or when data transfer between layers in the computational stack. Reasons of changes in the data are shown in Table 1. Generally, changes in the data could be permanent or transient phenomena. At the model level in Simulink, we can easily mimic data changing manipulating blocks. These manipulations include changing the value of parameters directly or adding small changes in the computational path that changes these parameter values.

Table 1. Cause of data changing

Hardware	Software	Interface
Noise Bit flip Wrong measurement Stuck at 0/1	Developer/User mistake Buffer overflow Small memory Share resource Resource starvation	Noise Stuck at 0/1 Bus contention

- **Latency**

Cause of latency faults are described in Table 2. Delay does not have a permanent nature. In hardware, latencies are normally intermittent and are transient in interface and software. Adding a delay block enables us to manipulate timing behavioral of hardware and software.

Table 2. Cause of latency

Hardware	Software	Interface
Component wear out Busy component	Work load Synchronizing Share *resource*	Busy line Bus contention

- **Open link**

These faults come from broken wires or broken component in the system. It accrues in hardware and interface not in software. Its nature is permanent, and it remains faulty until end of simulation.

- **Code insertion/drop**

This type of faults happens in software only. It can originate from developer's mistakes and/or manual introduced bugs e.g. missing parameter incrementation within a loop control structure. By adding and/or dropping certain parts of the execution path, framework create control flow errors in software. In Simulink, blocks can be added to define a different control flow or blocks can easily be short-circuited.

4.2 MiL Fault Injection Workflow

Fig. 4 shows the workflow needed to create a MiL FI experiment in Simulink. We use UML activity diagrams to depict the workflow [23]. The process starts at the black circle and ends with the black dot in a circle. White round tangles indicate manual activities and yellow round tangles are automatic actives within the framework. Diamonds show control flow decisions (in our case for looping the different experiment runs within a single experiment). The black bar represents a fork and join node. In between the fork and the join node, the activities are concurrently executed.

- **Define Experiment**

Before starting to annotate the model with FI blocks, the goals of the experiment should be clear. The user must define these goals by modelling the properties she/he wants to check. As previously mentioned, temporal logics can be used to define such properties. The properties and its monitoring of the control model explicitly models readouts and measures of the FARM model. The input scenario also must be selected (or modelled) such that a complete simulation is possible. These models are a small but important part of the activation part of the FARM model.

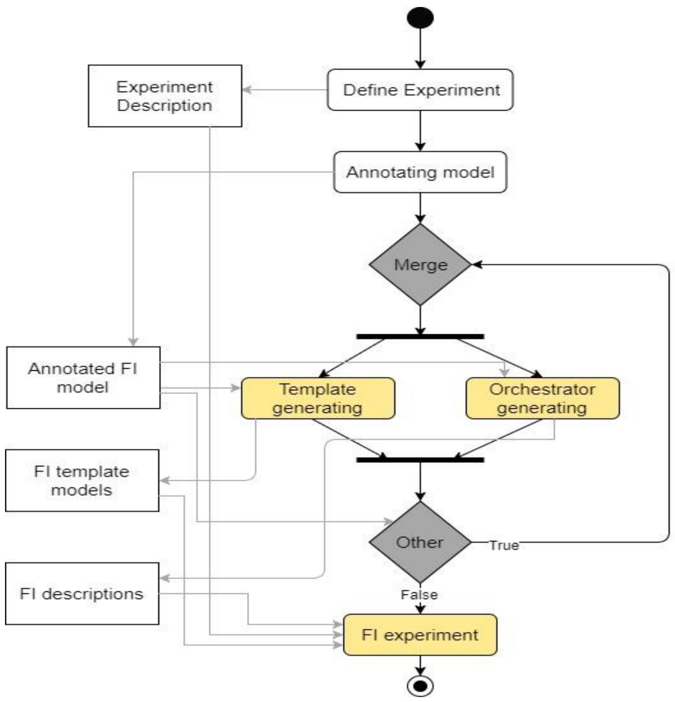
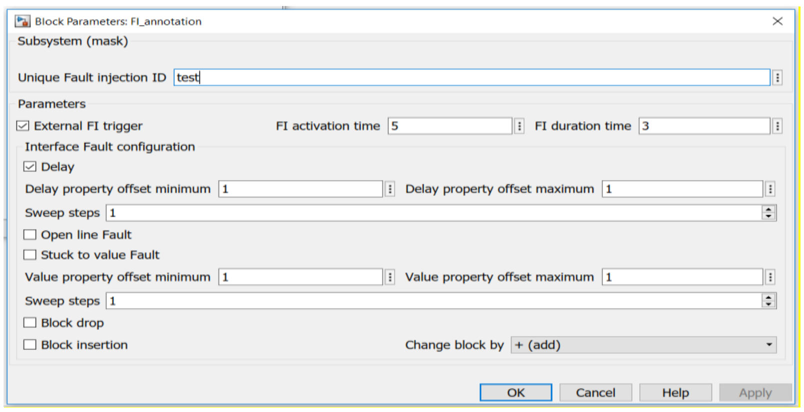


Fig. 4. FI framework workflow

- **Annotation**

The objective of our instrumentation problem is to create an instrumentation of the elements of the Simulink model that are marked for FI. We instrument a block in the Simulink model at the output ports. The framework enables the user to indicate which blocks should be instrumented, with what different types of faults, and the time at which to inject the fault (explicit time, range and/or simple conditions). The activity takes the control model as input. The output is a new annotated model with user defined parameter for the FI experiment. For annotating, custom FI blocks that can be configured, shown in Fig. 5, are available. Users need to dedicate a special ID for each instrumentation block. In addition to the set of faults, users define parameters for each of the selected faults. This includes FI orchestrator settings for activating the saboteur blocks. Saboteurs are the faulty blocks that replace the annotation and are responsible for injection the fault at the correct time instant. In Fig. 6 you can see power window controller which annotated by user.



Block Parameters: FI_annotation

Subsystem (mask)

Unique Fault injection ID: test

Parameters

☑ External FI trigger — FI activation time: 5 — FI duration time: 3

Interface Fault configuration

☑ Delay

Delay property offset minimum: 1 — Delay property offset maximum: 1

Sweep steps: 1

☐ Open line Fault

☐ Stuck to value Fault

Value property offset minimum: 1 — Value property offset maximum: 1

Sweep steps: 1

☐ Block drop

☐ Block insertion — Change block by: + (add)

Fig. 5. Annotate window

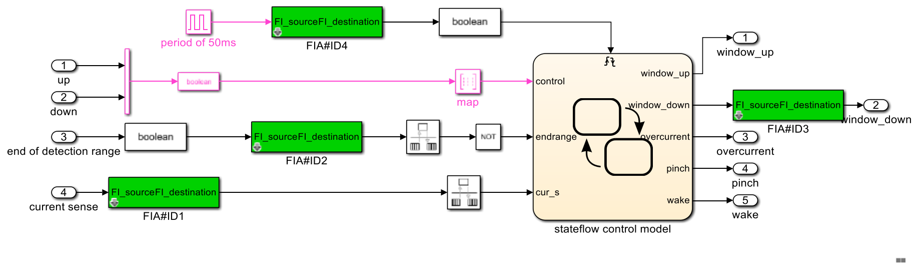


Fig. 6. Annotated controller

- **Generate injection model**

This activity transforms the annotated model into a model with the saboteurs added to the model. As an input, the model with the fault types and parameters is used. The orchestrator goes over each annotation block and adds the correct saboteur blocks in the model. Internally, each of the faults in the fault library has an associated transformation rule attached to it. Fig. 7 shows a part of the transformation rule library in our framework that transforms a changing parameter, an open line and does code insertion. Each saboteur can be activated by the switch block that either takes the normal (without fault) path or the faulty path. These trigger blocks are activated by the experiment FI orchestrator.

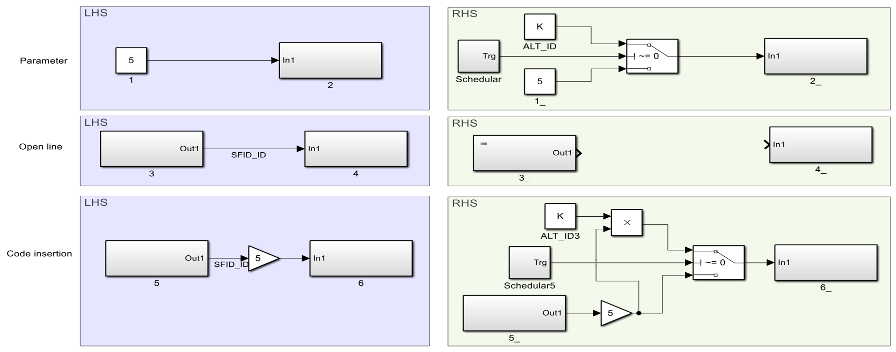


Fig. 7. Part of fault library for MT regarding open line and code insertion

- **Generate model orchestrator**

The orchestrator is generated concurrently with the generation of the annotated model The activity that generated the orchestrator also uses the annotated model as input. As output it generates an activity diagram and parameter files that are an explicit representation of the experiment that will run. From this activity diagram the framework generates a MATLAB m-script that is used for the MiL experiment. Again, MT are used to generate the activity diagram. For this, each fault library rule from the previous activity has a sister rule to generate the orchestrator at the same time as the fault injected model. The next subsection shows some more information about the orchestrator.

- **FI Execute Experiment**

Finally, the experiment can be run using the generated m-script from the activity diagram. The property models provide information about the requirements that are met or violated when faults are injected.

- **Fault injection Orchestrator**

The Orchestrator is responsible for activating the different saboteurs in the correct experiment run. As previously shown, each saboteur has an input which is triggered by the orchestrator. An abstraction of the fault orchestrator is shown in Fig. 8. The following activities are shown:

  - Initializing orchestrator. The orchestrator must load some files to execute the different simulations. First, the FI description contains some fault's parameter such as range and step size of the latency of a saboteur.

Second, the stimuli description contains information of triggering time (or triggering condition). Finally, the property description file contains the requirements, condition or functions which fulfilled. The modeled properties in Simulink allow to automate the process of property checking.

- Preparing FI parameter. In this step, the orchestrator gets the fault parameter description according to the template and the FI scenario, defined within the experiment. This contains the information on how to activate the correct saboteur.

- Load FI template model. In this phase, the orchestrator prepares the FI template model for the experiment by loading the parameter into the FI template model. The experiment setup is almost complete and can be executed to produce the traces for further analysis.

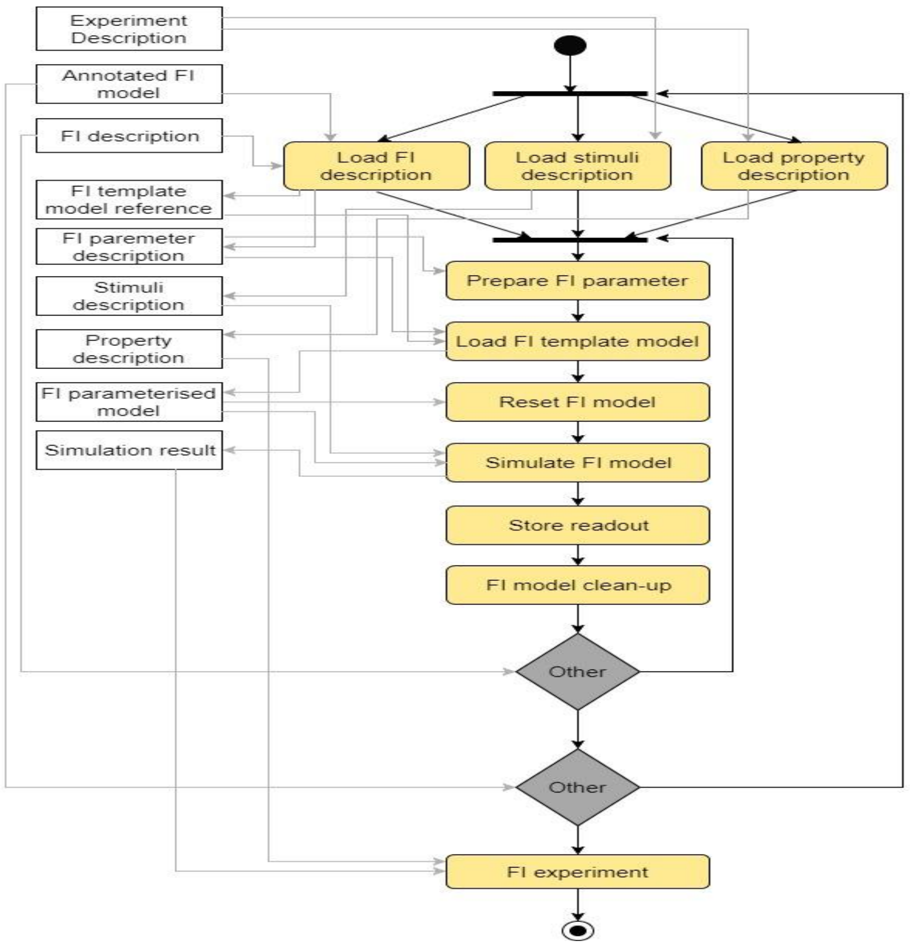


Fig. 8. Orchestrator diagram

- Reset FI model. Before executing the experiment, the orchestrator first resets the plant model and controller model.

- Simulate FI model. According to the parameterized model and the stimuli description, the orchestrator runs the simulation from the start to stop time.

- Store readouts. The results or readouts of the experiment are logged in a MATLAB file, used for further offline analysis.

- FI model clean-up. After running the simulation, the orchestrator will clean and reset the model for the next simulation run. Next, the orchestrator will check its parameters to find out if it is sweeping a parameter finished or not. If there are more sweep parameter, the orchestrator will prepare new FI parameter and execute the sweep of the parameter. Finally, the orchestrator checks if there is another saboteur for execution. If so, it will do the experiment runs of the other faults.

- Analyze readouts. At the end of each experiment, an analysis of the results or readouts is needed. This process is offline and is activated by the orchestrator after the complete experiment suite is simulated. To enable this, all the readouts are stored for each experiment.

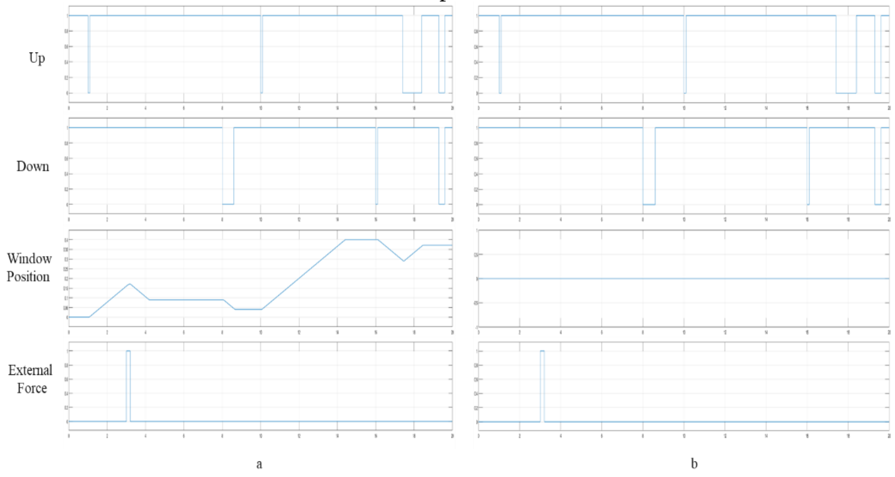


Fig. 9. MiL sample result **a.** Normal model behavioral **b.** Faulty model behavioral

Fig. 9. shows the results of an experiment run. The left-hand side figure is the result of a model without faults and the right-hand figure shows the result from a fault within model. The first row, second row and last row are in inputs from the up button, down button and output from current sensor which will be active when there is something between window and door's frame or if there will be external force toward window. We can see in normal operation that the window position is changing logically when the up button is active. The window position is increasing until the time that the current sensor will be active, then the window stops and goes down for a certain time. Afterward, by pushing the up and down button we see that window position is changing. In the faulty injected model, the window position is not changing anymore.

4.3 HiL Fault Injection

In this subsection we look at the differences in the framework when using it for HiL FI. In HiL, we consider the deployment of the control model onto the hardware platform.[1] This also requires that the orchestrator is either run on the embedded platform or run from an experiment box e.g. the HiL real-time target simulator. Synchronization between the plant model running on HiL box and the controller should be provided such that the plant model and control model are reset and started at the correct times. We implement the orchestrator on the embedded board and synchronize with the HiL box using a dedicated general-purpose input/output (GPIO) pin on the hardware. Furthermore, we consider the real-time properties of the control task such that it does not overrun its real-time bounds, as this could create additional faults that are hard to trace to the injected fault.

Extra workflow steps: To allow for HiL FI, we need to add some extra steps in the workflow model of Fig. 4. These steps have to be done before starting the MiL workflow.

Define hardware deployment. The hardware deployment model is needed to allow us to model the injection of representative faults that occur at the interface level and other hardware faults (e.g. faulty sensor readings). A custom deployment block set in Simulink allows the modeler to model the deployment of the control models onto

[1] Note that the hardware setup can contain multiple interconnected hardware platforms and by this, the control model is distributed over these hardware platforms.

the defined hardware setup. This extra deployment information is used to add extra FI annotation to the system model.

Generating hardware FI deployment model. This step generates the deployment model automatically by referring to the hardware model and deployment description. Deployment descriptions contain information about the timing of the system components for example, the deadline and worst-case execution time. The output of this step is a model with annotation block. In this phase, the framework determines the place for annotation block and generates a new model.

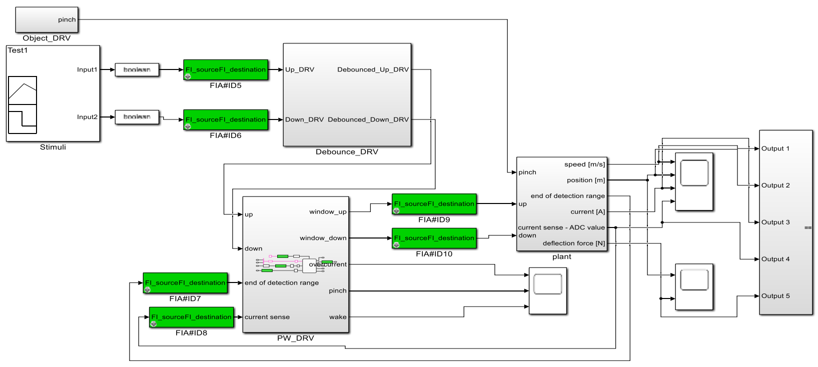


Fig. 10. FI deployed model

Fig. 10 contains FI deployment model from the use case. In this model, the annotation blocks have been added at the interconnections (which they are physical part). These boxes are between inputs, plant and controller indicate hardware-based faults such as stuck to value.

- **Generation of the orchestrator**

As previously mentioned, a different implementation of the experiment orchestrator is needed so it can run on the embedded hardware. This can be done by transforming the activity model into C-code. We build upon the OSEK (Open Systems and their Interfaces for the Electronics in Motor Vehicles) as our real-time OS (RTOS) for running the control model. This is reflected in the generated code.

For representing an example of the generated code, we show a part of orchestrator as pseudo as follow:

```
TASK(InputTaskPeriodic)
{
    if (simtime<=SIM_stop){
        if (simtime==0)
        {
            FIOrchestrator();
        }
        simtime = simtime+stepsize;
    }else{
        if (FI_block <= FI_blockcount){
            simtime = 0;
        }
    }
    TerminateTask();
}
```

The **FIOrchestrator()** function is called within a periodically triggered task and initializes the FI and overall simulation parameters. This orchestrator function will take care of the synchronization between the plant model, running on the HiL box and the controller, running the embedded hardware and will reset them both when needed. This synchronization is a very important step for valid HiL FI simulations. The real-time constraint of the control system tasks is also critical for validity of HiL FI simulations. Despite of the additional FI blocks and the FI activation of different saboteurs, the tasks must assure to finish before the defined deadline, otherwise real-time behavior of system is not met and, as previously mentioned, faults can originate from this real-time constraint violation.

- **Real-time constraints for HiL**

For HiL instrumentation in a real-time environment we want to test the behavioral of model without introducing extra fault because of real-time constraints. Therefore, we must assure that real-time behavior of system is not violated. We use the slack parameter to define how much extra blocks can be added to the execution of the Simulink model. The slack time is explicitly modelled during the modelling of the deployment. Fig. 11 shows an example of slack time of a task that needs to be instrumented with FI. Both "control task 1" and "to instrument task" have the same period (we assume that the deadline of the task is equals the period). The "to instrument task" and "the control task 1" are both released at times n×T1 (with n = 1…∞). In our example, "task control 1" has the highest priority and runs first. Afterwards, the "to instrument task" runs. Because there is some time left in between the end of the execution of "to instrument task" and the start of the new period, this extra time is available for instrumentation. Note, that we use worst-case execution timings for both tasks. The creation and/or extension with other properties, such as memory consumption, is similar.

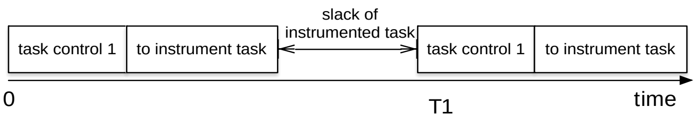


Fig. 11. Slack in execution of tasks

Our model for extending the framework with real-time properties is straightforward. The execution model of Simulink is based on [24]. We characterize each FI pattern (by transformation rules) to add saboteurs to the application. These saboteurs add execution time to the application and this execution time needs to be taken into account in order to fulfil the real-time constraints of the application. In certain cases, this is quite small because stateless paths in the Simulink model execution will not compute when the other path is chosen. Stateful paths however need to be updated and compute every execution step of the Simulink model. The condition blocks also always execute every step.

When all MT rules are characterized, a greedy approach to defining the experiment is created. To avoid overruns, the experiment is no longer one schedule with one model, but rather a set of schedules and its instrumented models. Now, the slack mapper towards creating the FI enabled Simulink model and schedule count the added execution times when applying a rule. It will use a greedy approach for selecting the next instrumentation to apply to the model. When there is no possibility anymore to apply an instrumentation transformation (the slack become too small), a new orchestrator and model with saboteurs is created. The process repeats until all instrumentation requests are fulfilled. For seeing the HiL's result you can refer to [25].

5 Discussion

In our approach, the readouts and measures of the FARM model are represented by the modelling of (safety) properties that can be traced to the safety requirements. The modelling can, in Simulink, be done by the use of explicit blocks that monitor a combination of signals in the model. Another approach is to model these properties using another formalism. Appropriate formalisms for detecting complex temporal patterns are temporal logics such as STL and linear temporal logic (LTL) [26-27]. However, the abstraction level which to specify these logics is at a wrong abstraction level. Therefore, we need express these properties at that same level of abstraction as the model itself. ProMoBox is such a technique that allows engineers to model properties at a more domain-specific level [28]. Properties also specifies what needs to be observed after a specific period, in safety properties referred to tolerance time interval. The simulation orchestrator can take this time into account to check if the properties are violated or not and reduce the simulation time. The detection using STL can be done either online or offline on the traces of the model.

In this work, we only modelled part of the activation concept of the FARM model. The time to activate an event is underspecified in our case. It just uses an explicit list of times to inject the fault or a range of injection timings by specifying a start time and stop time with discrete steps. If we want to cover all scenarios, experimenting overhead will increase and make the approach infeasible. To solve this, domain-knowledge needs to be added to the injection time. This can be done using the same property language. The temporal logic can be used to create fine grained control over the injection time as it is possible to specify the conditions when a fault is activated. For real-time simulation in HiL, we can deploy the monitor on the target or on the HiL box that simulates the plant model. If the monitor is deployed on the target, we need to take the computation time of the STL monitor into account when experimenting. When it is deployed on the HiL box, all the necessary signals must be exposed to the HiL box which is not always feasible.

The proposed method uses MT techniques to implement the injection of faults. However, the Simulink language and its features allows for a much simpler solution by embedding the sabotage blocks immediately in the parametrized fault library block or by using code generation. We have several reasons not to do this. The most important reason is that our method is generic and not depending on language features such as hierarchy, scripting and parametrization. Not relying on such language features but using the generic approach of MT allows us to port the technique to other formalisms and tools. As each formalism is modelled using a type-model or meta-model, the generic technique of ramification can be applied to the meta-model and MTs can be created for this language. The added benefit of the explicit transformation is that the intermediate model can more easily be analyzed for additional properties such as timing or memory consumption. Finally, it also helps in extending the fault library more easily. When using the hierarchy with scripting approach, the expert needs to create scripts in a possibly new language. When using code generation, code generation capabilities of the block have to be altered which is in the solution domain, rather than in the problem domain. The transformation approach is probably closest to the mental model of the expert and thus most appropriate.

6 Related work

Methods in [29] and [30] are the closest approach to our framework. This tool tries to use model-based FI at the beginning level of design to decrease the cost. The main advantage of this tool is providing minimal cut set for hardware-based FI and software-based FI. MODIFI does this by inspiration of hardware fault types. Our framework hasn't provided minimal cut set, but it has two more advantages. First, we considered software FI method as well as hardware FI. We have code inserting and code drop as well as latency (it could be considered as shared fault type between hardware-based and software based). Secondly, our framework covers a wide sweep range for each parameter of fault types. The user can set start point, end point and steps for sweeping data and latency which enables the user to have a better evaluation of dependency while MODIFI covers only 30 different faults. In [31] authors try to combine software-based technique with model-based FI separately. So, this work doesn't cover different aspect of FI, but they focused more on efficiency.

7 Conclusion and Future Work

For having a fast FI-based for dependability evaluation, there are three main approaches: (i) workload-based FI [32-33], (ii) sample-spread-based FI [9, 34], or (iii) introducing error behavioral functions [35] and we are going to use these techniques and combine them to framework to increase efficiency.

In addition, we are going to determine fault types and their values in a matured approach based on system dynamic feature, system architecture and cut set analysis. Adding these techniques will enable user and framework to choose fault wisely in model and increase fault coverage of system as well as fast instrumentation.

Also, in current framework, one fault per model is implemented and triggered. A better approach is that a framework does not limit the user to inject faults. In the next version, we try to add capability of mix FI and activation.

In conclusion, for each type of fault in hardware and software level of design there are a lot of tools, platforms and environments which focus on one or several fault types. You can find them in [36], but there is no such comprehensive environment to automate process specially in beginning of development.

Acknowledgment. This work is partly supported by INES (Innovation in the Development of Electrical Systems For Aeronautics) under project no. 11172. We thank our colleagues at Siemens Industry Software in Leuven, Belgium for supporting us: Stefan Dutre, Jef Stegen, Sweety Pate, Jonathan Menu and Satya Prakash Jha.

References

[1] Lee, E.: Cyber physical systems: Design challenges, Object Oriented Real-Time Distributed Computing (ISORC). In: 11th IEEE International Symposium on IEEE, pp. 363-369 (2008).

[2] Pintard, L., Toulouse, D. E.: Des Analysis de securite a la validation experimentale par injection de faults – le CAS DES systems embarques automobiles. Doctorat de l' université de toulouse (2012).

[3] ISO Homepage, https://www.iso.org/standard/54591.html

[4] Project page, http://www.eurekanetwork.org/project/id/11172

[5] Ziade, H., Ayoubi, R., Velazco, R.: A Survey on Fault Injection Techniques. In: The International Arab Journal of Information Technology, 1(2), pp. 171–186 (2004). doi: 10.1.1.167.966

[6] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J. C., Laprie, J. C., Powell, D.: Fault injection for dependability validation: a methodology and some applications. In: IEEE Transactions on Software Engineering, 16(2), pp. 166–182 (1990). doi:10.1109/32.44380

[7] Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Törner, F.: Improving fault injection in automotive model-based development using fault bypass modeling. In: GI-Jahrestagung, pp. 2577–2591 (2013).

[8] Kooli, M., Bosio, A., Benoit, P., Torres, L., Kooli, M., Bosio, A., Torres, L.: Software testing and software fault injection (2016).

[9] Wang, W., Trivedi, K. S., Profeta, J. A.: The Impact of Fault Expansion on the Interval Detection Coverage Estimate for Fault. In: Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing, pp. 330–338 (1994). doi: https://doi.org/10.1109/FTCS.1994.315627

[10] Hsueh, M.-C., Tsai, T. K., Iyer, R. K.: Fault injection techniques and tools. In: Computer, Vol. 30, pp. 75–82 (1997). doi: 10.1109/2.585157

[11] Balsini, A., Natale, M. Di, Celia, M., Tsachouridis, V.: Generation of simulink monitors for control applications from formal requirements. In: 12th IEEE International Symposium on Industrial Embedded Systems, SIES – Proceedings (2017). doi: 10.1109/SIES.2017.7993389

[12] Vanherpen, K., Denil, J., Vangheluwe, H., Meulenaere, P.: Model transformations for round-trip engineering in control deployment co-design. In: SpringSim (TMS-DEVS), pp. 55-62 (2015).

[13] Zander, J.: Model-based Testing of Real-Time Embedded Systems in the Automotive Domain. PhD-thesis, TU Berlin (2009).

[14] Luo, A., Wu, C., Shen, J., Shuai, Z., Ma, F.: Railway static power conditioners for high-speed train traction power supply systems using three-phase V/V transformers. In: IEEE Trans, Power Electron., vol. 26, no. 10, pp. 2844–2856 (2011).

[15] Hasanzadeh, A., Edrington, C. S., Stroupe, N., Bevis, T.: Real-time emulation of a high-speed microturbine permanent-magnet synchronous generator using multiplatform hardware-in-the-loop realization. In: IEEE Trans. Ind. Electron., vol. 61, no. 6, pp. 3109–3118 (2014).

[16] Ou, K., Rao, H., Cai, Z., Guo, H., Lin, X., Guan, L., Maguire, T., Warkentin, B., Chen, Y.: MMC-HVDC Simulation and Testing Based on Real-Time Digital Simulator and Physical Control System. In: IEEE J. Emerg. Sel. Top. Power Electron., vol. 2, no. 4, pp. 1109–1116 (2014).

[17] Zhang, H., Zhang, Y., Yin, C.: Hardware-in-the-Loop Simulation of Robust Mode Transition Control for a Series-Parallel Hybrid Electric Vehicle. In: IEEE Trans. Veh. Technol., vol. 65, no. 3, pp. 1059–1069 (2016).

[18] Yang, X., Yang, C., Peng, T., Chen, Z., Liu, B., Gui, W.: Hardware-in-the-Loop Fault Injection for Traction Control System. In: IEEE Journal of Emerging and Selected Topics in Power Electronics, 6(2), pp. 696–706 (2018). doi: 10.1109/JESTPE.2018.2794339

[19] Kühne, T.: Explicit transformation modeling. In: International Conference on Model Driven Engineering Languages and Systems. Springer (2009).

[20] Denil, J., Mosterman, P. J., Vangheluwe, H. L. M.: Rule-based model transformation for, and in simulink. In: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative (DEVS'14), (1), Article No. 4 (2014).

[21] Syriani, E., Vangheluwe, H.: A modular timed graph transformation language for simulation-based design. In: Software and Systems Modeling. 12(2), pp. 387–414 (2013). doi: 10.1007/s10270-011-0205-0

[22] Denil, J., Kashif, H., Arafa, P., Vangheluwe H., Fishmeister S.: Instrumentation and preservation extra-functional properties of Simulink Models. In: Proceedings of the Symposium of Theory of Modelling and Simulation. (2015).

[23] UML Homepage, http://www.uml.org/

[24] Di Natale, M., Guo, L., Zeng, H., Sangiovanni-Vincentelli, A.: Synthesis of multitask implementations of Simulink models with minimum delays. In: IEEE Transactions on Industrial Informatics, 6(4), pp. 637–651 (2010). doi: 10.1109/TII.2010.2072511

[25] https://www.youtube.com/channel/UCvfwLU_G0FrbSl1Ef7fbHUg?view_as=subscriber

[26] Bartocci, E., Ferrère, T. (n.d.).: Localizing Faults in Simulink/Stateflow Models with STL. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week) (HSCC '18). ACM, New York, NY, USA, pp. 197-206. doi: 10.1145/3178126.3178131

[27] Balsini, A., Natale, M. Di, Celia, M., Tsachouridis, V.: Generation of simulink monitors for control applications from formal requirements. In: 12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017 – Proceedings (2017). doi: 10.1109/SIES.2017.7993389

[28] Meyers, B., Wimmer, M., Vangheluwe, H., Denil, J.: Towards domain-specific property languages: the ProMoBox approach. In: Proceedings of DSM 2013, pp. 39–44 (2013). doi: 10.1145/2541928.2541936

[29] Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: MODIFI: A MODel-implemented fault injection tool. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6351 LNCS, pp. 210–222 (2010). doi: 10.1007/978-3-642-15651-9_16

[30] Svenningsson, R., Eriksson, H., Vinter, J., Törngren, M.: Model-implemented fault injection for hardware fault simulation. In: Proceedings - 2010 Workshop on Model-Driven Engineering, Verification, and Validation, MoDeVVa 2010, pp. 31–36 (2011). doi: 10.1109/MoDeVVa.2010.11

[31] Guthoff, J., Sieh, V.: Combining software-implemented and simulation-based fault injection to a single fault injection method. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers, (Ii), pp. 196–206 (1995). doi: 10.1109/FTCS.1995.466978

[32] Goswami, K.K.: DEPEND: a simulation-based environment for system level dependability analysis. In: IEEE Transactions on Computers 46(1), pp. 60–74 (1997).

[33] Kanawati, G.A, Kanawati, N.A., Abraham, J.A.: FERRARI: A Flexible Software-Based Fault and Error Injection System. In: IEEE Trans. Computers, vol. 44, no. 2, pp. 248-260 (1995).

[34] Powell, D., Martins, E., Arlat, J., Crouzet, Y.: Estimators for Fault Coverage Evaluation. In: IEEE Trans. Computers, vol. 44, no. 2, pp. 261-273 (1995).

[35] Imén, M., Ohlsson, J., Torin, J.: On Microprocessor Error Behavior Modeling. In: Int. Symp. Fault Tolerant Computing, FTCS-24, IEEE Computer Society, pp. 76-85 (1994).

[36] Yu, Y., Johnson, B. W.: Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation (2003).