Universitá degli Studi di Verona

DEPARTMENT OF COMPUTER SCIENCE
Master Degree in Computer Science and Engineering

Fault Models for Non-electrical Disciplines in Verilog-AMS

Candidate:
Alessandra Castiglioni
VR424616

Supervisor:
Franco Fummi

Advisors:
Graziano Pravadelli
Nicola Dall'Ora
Enrico Fraccaroli

Academic Year 2019–2020

Contents

List of Figures

Chapter 1

Introduction

Analyzing the faults that can occur in a system allows us to act in advance and create a device that can be fault-tolerant.
However, the situation is becoming more complex every day with the creation of more and more smart devices that interface 360 degrees with the world around them.
Smart systems integrate so many different features, and while it may seem at first that the accouterments used for other systems are sufficient, they need more careful analysis.
By interfacing with a wide variety of physical quantities while collecting and processing data, smart systems can incur an increasing number of different types of faults. That's why it's necessary to analyze in more detail the types of failures that may occur and be able to standardize them by creating failure models that can be reused in all circumstances.
Multi-domain faults analysis is done in this case in the language that best lends itself to smart system modeling and simulation, that is Verilog-AMS, as can be read in chapter 4.
To do this, it is also useful to have tools that speed up the injection and simulation procedures, to facilitate the study of the available models, so a possible tool for this purpose is proposed in chapter 5.

1.1 Thesis Outline

The thesis is organized as follows:

chapter 2 Background and State of the Art introduces the main concepts needed to understand this work and where it is positioned in relation to the state of the art of modeling and simulate non-electric faults with the Verilog-AMS language.

chapter 3 Objectives analyzes the problems present in this field of study and explains the main goals of this work in detail.

chapter 4 Modeling Faults on Heterogeneous Domains describes the methodology used to identify the multi-domains faults and to create a fault taxonomy.

chapter 5 A Framework for Injecting and Simulating Faults describes the implementation of the framework to inject and simulate multi-domain faults.

chapter 6 Experimental Validation shows experimental results of all the work, obtained by applying what has been seen in the previous chapters to the study models.

chapter 7 Concluding Remarks and Future Works summarizes the work and conclusions are drawn about it, also presenting possible future tasks.

Chapter 2

Background and State of the Art

This chapter presents the concepts necessary to better understand this thesis, starting with the basic concepts and then describing the technologies mentioned and used. Finally, the state of the art on the subject is presented.

2.1 Basics

2.1.1 MEMS

MEMS, which stands for Micro-electro-mechanical Systems, are miniaturized sensors, actuators, devices and systems with a critical dimension: the electromechanical functions are brought to the nanometric level, and everything needed to interface with the host microcontroller is into a single device.
Even though many initial concepts for MEMS were based on silicon, a variety of other materials and fabrication techniques have been developed over the last two decades for applications in mechanical, electrical, chemical, biological, and other disciplines.
MEMS technology is progressively replacing products made with traditional technologies: MEMS devices such as accelerometers, gyroscopes, high performance mirror displays, pressure sensors, micro motors, micro engines, RF switches, valves, pumps, ultra sensitive membranes, single-chip microfluidic systems such as chemical analyzers or synthesizers, single-chip micro total analysis systems (also referred to as lab-ona-chip) and many more devices and systems have been designed and fabricated over the last one to two decades.
The impetus to use MEMS devices derives from potential cost benefits and size advantages as well as performance considerations.
The cost benefits have a common premise: more specifically, incorporating

MEMS devices directly on a communications IC or package is a powerful approach to eliminate assembly steps and reduce fabrication costs.
The global microelectromechanical systems market includes products such as automotive airbag systems, display systems, pressure and temperature sensors, and popular inkjet cartridges.

2.1.2 Smart Systems

Smart systems are miniaturized self-sufficient devices, that incorporate functions of sensing, actuation, and control to describe and analyze a situation and make decisions based on the available data in a predictive or adaptive manner (smart actions that make the systems smart).
The concept of smart system has been introduced to identify energy-efficient, autonomous miniaturized devices performing sensing, actuation, communication, and control through computation.
This class of systems requires integrating many different technologies within the same device; in order to obtain a smart system, more and more skills are combined to be placed on SoCs. Digital HW and SW performing computation must co-exist with analog HW for wireless communication, as well as integrated sensors and actuators, usually implemented as Micro-Electro-Mechanical Systems.

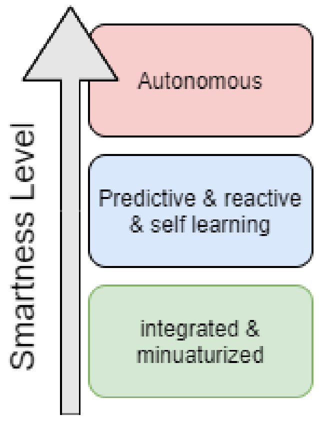

Autonomous
Predictive & reactive & self learning
integrated & minuaturized
Smartness Level

Figure 2.1: Smart systems evolution overview

2.1.3 Hardware Description Language

A Hardware Description Language (HDL) is a specialized programming language used to describe the behavior and the structure of electronic circuits

and digital logic circuits. A hardware description language enables a precise, formal description of an electronic circuit that allows for the automated analysis, simulation, and simulated testing of an electronic circuit.
This type of programming language allows, unlike others, to distinguish between structural, behavioral, and data flow components of a project and accurately describe how the different parts connect to each other.
Also, a substantial difference with other programming languages is that HDLs explicitly include the notion of time.
Instead of creating an executable file, HDL compilers provide a net-list ready to be fully translated into an integrated circuit, a procedure called synthesis.
A description of the hardware created using HDLs is ready to be analyzed and simulated using specific software.
As a result of the efficiency gains realized using HDLs, most modern digital circuit design revolves around them. The process of writing the HDL description is highly dependent on the nature of the circuit and the designer's preference for the coding style.
The most popular and widely used hardware description languages are VHDL and Verilog, which allow both analog and digital designs to be modeled. In particular, analog simulation relies on simulators or ad hoc models, expressed using analog HDL extensions, like Verilog-AMS, VHDL-AMS, SystemC-AMS.

2.1.4 Faults

A fault is defined as a physical failure mechanism due to some defects. As a consequence, an error is the condition (or state) of a system containing a fault (i.e. a deviation from a correct state).
Depending on their temporal characteristics, a fault can be classified in:

- *Permanent fault:* a fault that has existed long enough to be observed at test time;
- *Transient fault:* a fault that appears and disappears in short intervals of time;
- *Intermittent fault:* a fault that appears and disappears at regular time intervals.

2.1.5 Fault Model

A fault model is a logical abstraction describing the functional effect of a physical failure.
There are several faults models according to the level of abstraction in which

the design with faults is described. A fault model fits into one of the following assumptions:

- *simple fault assumption:* only one fault occur in a circuit. If we define k possible fault types in our fault model, the circuit has n signal lines, by single fault assumption, the total number of single faults is $k * n$.
- *multiple fault assumption:* multiple faults may occur in a circuit.

A *stuck-at fault* is a particular fault model in which individual signals and pins are assumed to be stuck at Logical "1", "0", and "X".
The stuck-at-0 model represents a signal that is permanently low regardless of the other signals that normally control the node.
The stuck-at-1 model represents a signal that is permanently high regardless of the other signals that normally control the node.

2.1.6 Fault Tolerance

A system can be defined as fault-tolerant when it is able to continue to function properly even if a failure occurs in one or more of its components.
The ability to maintain its functionality when a portion of the system collapses is called graceful degradation.
A fault-tolerant design allows a system to continue its operations, although at a reduced level, rather than fail completely when some part fails.
When creating a system, fault tolerance can be achieved by anticipating exceptional fault conditions and building the system so that it can respond to them.
To create a fault-tolerant system it is important to understand how the system would react to them. To do this it is fundamental to inject the faults in a model, in order to simulate them and understand how to go to intervene on them.

2.1.7 Fault Injection

Fault injection is a methodology for verifying the robustness of a circuit i.e. its tolerance to faults, through the definition of a library of faults of interests plus some mechanism for actually injecting the faults.
Fault injection is used as a dependability validation technique for hardware or software applications. It is realized by implementing (or even automatically generating) testbenches by which the target system is stimulated in such a way that a faulty behavior is observed.
The stimulation process is realized by an injector, whereas the observations

are realized by a monitor.
The injector is a configurable component similar to a regular stimulus generator and can be configured to drive new values onto variables in a target system. As opposed to a regular stimulus generator which is connected directly to the target system's inputs, the injector is also connected to internal variables. A generic injector takes as configuration arguments the attributes which make up the target system's fault space.
The fault space [2.2] is made up of:

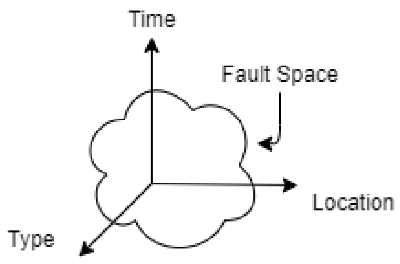


Figure 2.2: Fault space

- *Location:* the variable inside the target system where the fault should be injected;
- *Time:* the simulation time value or condition assertion that activates the fault injection;
- *Type:* the type of fault (e.g. stuck-at-0/1, single event upset, single event transient, etc...) which should be injected.

Faults injection techniques can be classified into two categories:

- *Software-based fault injection*, whereas the fault is injected in the simulation model of the system and can be injected at circuit, logic, or system-level abstraction;
- *Hardware-based fault injection*, where a physical system is used for the experiment.

In traditional simulation-based fault injection, faults are usually injected at the same level of abstraction as the design (mostly logic level or higher). The main advantage of simulation-based fault injection is the high controllability and observability it provides over the injection process.

2.1.7.a Fault injection techniques

The basic approaches to implementing fault injection are:

- *Hardware-implemented fault-injection:* utilizes specially created hardware, which allows the implementation of faults in the respective model. It can be intrusive/with contact or non-intrusive/without contact.

- *Software-implemented fault-injection:* uses additional code and/or software tools to intrusively emulate fault injection in the model either by manipulating objects (e.g. signals, ports, payloads) in the model or by modifying the modeled code altogether (e.g. simulation platform, code mutators).

- *Simulation-implemented fault-injection:* uses commands provided by the simulator or debugger to non-intrusively add faults in the model. The models are compiled only once and faults can be injected repeatedly in any sequence during a regression run by simply changing the test-case stimuli.

Fault injection based on code instrumentation relies on saboteurs or mutants. A saboteur is an artificial HDL component added to the original design, which does not affect the Device Under Verification (DUV) behavior during the normal operation.
The goal of a saboteur is to alter the properties of the target signal (e.g. value, timing, response) when the corresponding fault is injected.
A mutant is a component replacing another component with the following behaviors. When the mutant is not activated, it has exactly the same behavior as the original component. When the mutant is activated, it introduces a given faulty behavior with respect to the original component. The mutant can affect the values of the internal variables.

2.2 Technologies

2.2.1 Verilog-AMS

Verilog-AMS is the analog extension of Verilog; it combines Verilog-A, the analog-only subset, and Verilog-HDL, the digital one [2.3].
Verilog-A/MS allows to cover different levels of details and to describe physical processes and multiple domains through the definition of disciplines [2.4] and natures.
A discipline is a type used when declaring analog nodes, ports, or branches.

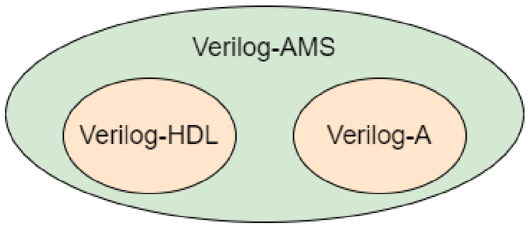
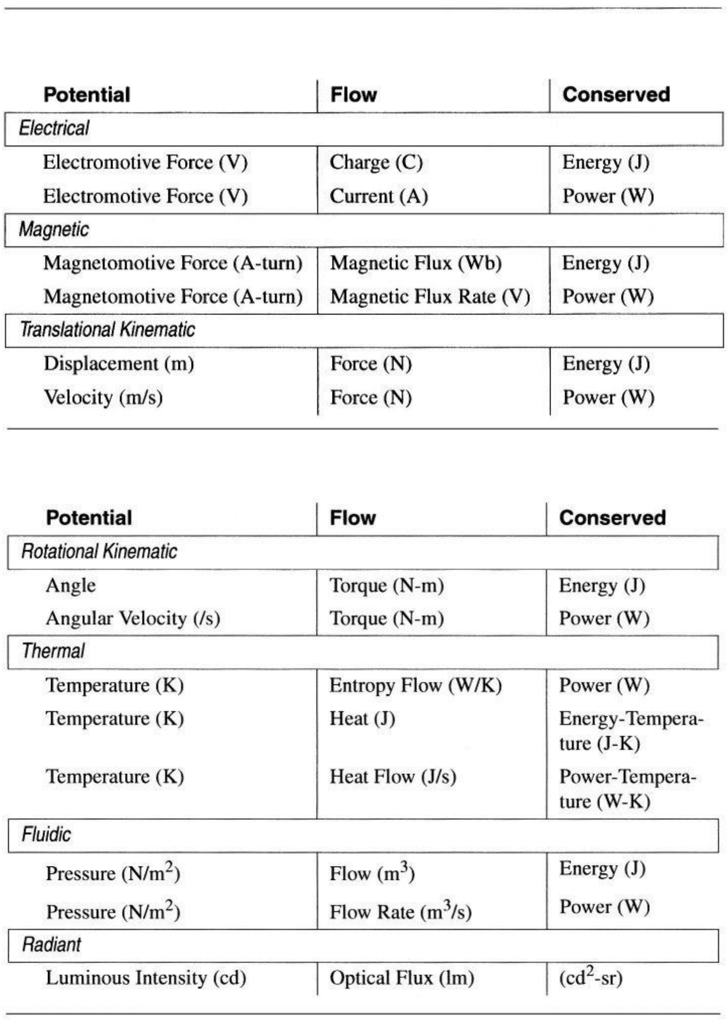


Figure 2.3: Verilog-AMS language



Potential	Flow	Conserved
Electrical		
Electromotive Force (V)	Charge (C)	Energy (J)
Electromotive Force (V)	Current (A)	Power (W)
Magnetic		
Magnetomotive Force (A-turn)	Magnetic Flux (Wb)	Energy (J)
Magnetomotive Force (A-turn)	Magnetic Flux Rate (V)	Power (W)
Translational Kinematic		
Displacement (m)	Force (N)	Energy (J)
Velocity (m/s)	Force (N)	Power (W)
Rotational Kinematic		
Angle	Torque (N-m)	Energy (J)
Angular Velocity (/s)	Torque (N-m)	Power (W)
Thermal		
Temperature (K)	Entropy Flow (W/K)	Power (W)
Temperature (K)	Heat (J)	Energy-Temperature (J-K)
Temperature (K)	Heat Flow (J/s)	Power-Temperature (W-K)
Fluidic		
Pressure (N/m^2)	Flow (m^3)	Energy (J)
Pressure (N/m^2)	Flow Rate (m^3/s)	Power (W)
Radiant		
Luminous Intensity (cd)	Optical Flux (lm)	(cd^2-sr)

Figure 2.4: Disciplines defined by the Verilog-AMS language

They can also be used to declare physical types, which are referred to as natures.
Examples of disciplines are: electrical, thermal, rotational, magnetic, etc...

Natures are used to describe the signals used in the discipline. A nature is a collection of attributes shared by a class of signals.
The attributes include the units, the name used when accessing the signal, absolute tolerance, related natures and pherhaps other user personalization. Examples of natures are voltage, current, charge, flux, etc...
It is also possible to define disciplines and personalized natures to express any other physical quantity not already defined in the standard of the language.
A system is considered to be a collection of interconnected components, that are acted upon by a stimulus and produce a response. In Verilog-A/MS components are instances of modules. The components might also be systems, so a hierarchical system is defined in the project, and components can have sub-components (components with no sub-components are called primitive components).
In order to simulate systems, it is necessary to have a complete description of the system and all of its components.
Descriptions of systems are given structurally, writing how components are interconnected to each other.
Descriptions of primitive components instead are given behaviorally with mathematical formulas that relate the signal at the ports of the component.

2.2.2 Modelica

Modelica is an object-oriented, declarative, multi-domain modeling language for modeling complex physical systems containing, for example, mechanics, electronics, hydraulics, thermal, and controls. It was created for modeling the dynamic behavior of complex multi-domain systems, using differential, algebraic, and discrete equations.
Modelica classes are not compiled like other object-oriented programming languages, because it is a modeling language. They are translated into objects which are then exercised by a simulation engine.
Moreover, although classes may contain algorithmic components similar to statements or blocks in programming languages, their primary content is a set of equations. In contrast to a typical assignment statement, an equation may have expressions on both its right and left-hand sides. Equations do not describe assignment but equality. In Modelica terms, equations have no pre-defined causality. The simulation engine may manipulate the equations symbolically to determine their order of execution and which components in the equation are inputs and which are outputs.
Modelica is used in a wide variety of applications, such as fluid systems (steam power generation, hydraulics, etc.), automotive applications (powertrain especially), and mechanical systems (multi-body systems, mechatronics, etc.).

2.2.3 MATLAB

MATLAB (an abbreviation of "matrix laboratory") is a multi-paradigm numerical computing environment and proprietary programming language developed by MathWorks.
MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, interfacing with programs written in other languages, and model system designs.
MATLAB supports object-oriented programming including classes, inheritance, virtual dispatch, packages, pass-by-value semantics, and pass-by-reference semantics. However, the syntax and calling conventions are significantly different from other languages.

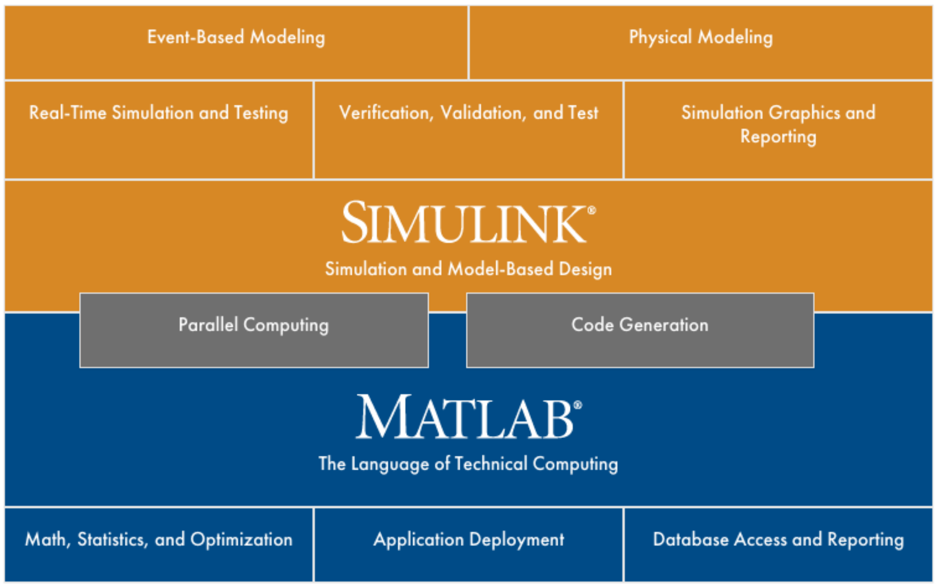

Event-Based Modeling
Physical Modeling
Real-Time Simulation and Testing
Verification, Validation, and Test
Simulation Graphics and Reporting
SIMULINK®
Simulation and Model-Based Design
Parallel Computing
Code Generation
MATLAB®
The Language of Technical Computing
Math, Statistics, and Optimization
Application Deployment
Database Access and Reporting

Figure 2.5: Matlab environment overview

2.2.4 Simulink

Simulink is a MATLAB-based graphical programming environment for modeling, simulating, and analyzing multidomain dynamical systems.
It provides many predefined modules, with the possibility for the user to create new ones.
Its primary interface is a graphical block diagramming tool and a customizable set of block libraries.
The various elements are brought into the workspace by simply dragging and

dropping them into it as if they were icons.
The libraries are Read-only: to change the parameters of a block, you must first drag it into the workspace.
It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it.
Simulink is widely used in automatic control and digital signal processing for multidomain simulation and model-based design.

2.2.5 Simscape

Simscape extends Simulink with tools for modeling and simulation of multi-domain physical systems.
Simscape enables you to rapidly create models of physical systems within the Simulink environment.
With Simscape you build physical component models based on physical connections that directly integrate with block diagrams and other modeling paradigms.
Simscape add-on products provide more complex components and analysis capabilities.
It is also possible to create custom component models using the MATLAB® based Simscape language, which enables text-based authoring of physical modeling components, domains, and libraries.
It is also allowed to quickly assemble models that span multiple domains. By assembling a schematic of the system to be modeled with lines representing physical (acausal) connections, the equations for the network of mechanical, electrical, hydraulic, etc. components are derived automatically. In fact, the Simscape libraries include models of more than 10 physical domains, including mechanical, electrical, and two-phase fluid.

2.2.6 Tool Command Language

TCL (Tool Command Language) is a high-level, general-purpose, interpreted, dynamic programming language. It is suitable for a very wide range of uses, including web and desktop applications, networking, administration, testing, and many others.
It contains two parts: a language and a library.
First, TCL is a simple scripting language, mainly used to release several interactive programs such as text editor, debugger, and shell. It has a simple syntax and powerful scalability, and it can create a new process to extend the capacity of its built-in command.
Second, TCL is a library package that can be embedded in the applications.

TCL's library contains a parser, some routines that can implement built-in commands, and library functions that allow users to expand.

2.2.7 HIF Suite

HIF Suite is a set of tools and application programming interfaces (APIs) that provide support for the modeling of HW/SW systems.
Its core is the HDL Intermediate Format which is a proprietary language used from all the tools, that is a kind of rewriting of the initial code in a XML-like format.
This framework has many functionalities: one of these is the manipulation of HDL designs to obtain their functional equivalent C++ version; another one is the possibility to translate a design from an HDL language to another.
The HIF Suite framework contains three different kinds of tools:

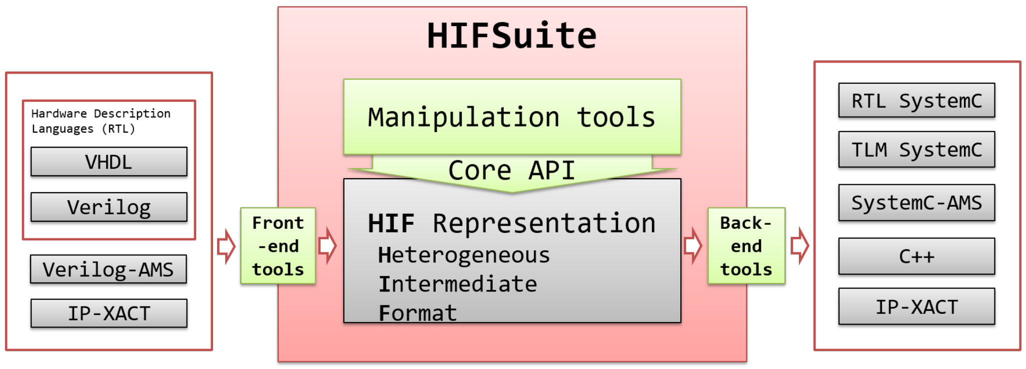


Figure 2.6: HIF Suite overview

- *front-end tools* parse HDL descriptions to generate the corresponding HIF representations;
- *back-end tools* parse HIF representations and convert them into HDL or pure C++ descriptions;
- *manipulation tools* manipulate the HIF representation introducing faults, abstracting data types, abstracting processes, etc.

Some front-end tools are vhdl2hif and verilog2hif, while some back-end tools are hif2vhdl, hif2sc, and hif2vp.
The verilog2hif tool can generate a HIF representation of the Verilog model (an XML tree) which can be more easily manipulated later by the framework.

2.3 State of the Art

This section presents the state-of-art approaches for the topics related to this thesis.
There is not a large amount of work in the scientific literature that analyzes the modeling of non-electrical faults in smart systems using verilog-ams, as it is challenging to model faults suitable for any physical model representation.

2.3.1 Multi-Discipline Modeling Technique

The two papers [1][2] analyzed use verilog-ams to model faults (electrical and mechanical) within cyber physical systems .
These works show how to model and inject faults in the different physical domains through the Verilog-AMS language and apply the proposed strategies to a DC motor, to prove their potentiality and effectiveness on a real case study.
According to their analysis, two approaches exist at the state-of-the-art to represent and simulate physical systems.
One alternative is to adopt graphical tools, e.g., Modelica-based tools and Simulink/Simscape. With this solution, the system is constructed through the connection of pre-defined blocks belonging to domain-specific libraries. The internal dynamics of such pre-defined blocks are not visible, and the underlying equations are visible only to the solver that computes system evolution over time. Blocks are nonetheless parametrized and thus can be customized by the designer.
The complementary approach is to adopt description languages, like Verilog-AMS, VHDL-AMS, SystemC-AMS. Such solutions require to explicitly model system evolution as differential and algebraic equations enriched with the application of energy conservation laws. This imposes a higher effort during model construction. On the other hand, the designer can choose the level of detail of the model, and it becomes possible to inject faults and faulty behaviors at any level of detail.

2.3.2 Modeling Faults with Verilog-AMS

Verilog-AMS allows to describing multiple domains through the definition of disciplines and natures. Each discipline defines two quantities for nodes: a potential (also known as the across value) and a flow (the through value). Verilog-AMS descriptions are thus defined as equations over nodes and branches

of possibly different disciplines. Non-conservative disciplines define only either the potential or the flow dimension. Conservative disciplines associate a nature to both the potential and flow dimension (e.g., in the case of the electrical discipline potential is voltage and flow is current).
Verilog-AMS is also able to interface analog and digital domains by means of specific timing functions (e.g., *cross()*, *timer()*, *transition()*) that allow an efficient synchronization between discrete-time and continuou-stime simulations.
Even if Verilog-AMS supports different disciplines, specific physical phenomena affect each discipline in different ways, so defining generic fault models that can have meaning in any physical discipline is difficult, if not impossible. Faults are injected in the Verilog-AMS description by adding the corresponding equations to add components or relationships in parallel or series. This is allowed in Verilog-AMS, with few restrictions.
Verilog-AMS allows defining parallel components between the same pair of nodes only if both have on the left-hand-side the flow between the pair of nodes, and it allows placing components in series between the same pair of nodes only if both have on the left-hand-side the potential between the pair of nodes. Mixing equations that have potential and flow on the left-hand-side between the same node is allowed by Verilog-AMS. However, in this case, the Language Reference Manual states that the value retention rule is applied [3]. According to this rule, when a contribution of a specific nature is defined on a branch of the circuit, introducing a new contribution of the same nature will result in a summation of the effects, while a new contribution of opposite nature will discard the previous contribution. For instance, if we write the voltage contribution between two nodes and the afterward, we define a current contribution between the same two nodes, the voltage one is discarded. To add the current contribution is thus necessary to find an alternative solution, e.g., by adding a new circuit node.

Chapter 3

Objectives

3.1 Open Issues

Ensure fault tolerance into systems is one of the crucial parts to build systems that can react in presence of faults and, to do this, many faults modeling and injecting techniques are been studied.
Much of the research work on fault modeling focuses mostly on analyzing techniques for modeling electrical faults in the analog and digital domains. However, when we consider a real device we realize that digital HW and SW coexist with analog components and Micro-Electro-Mechanical systems capable of sensing and controlling the physical environment, so the world with which it interfaces is vast and faults of any physical domain can arise, not only electric ones.

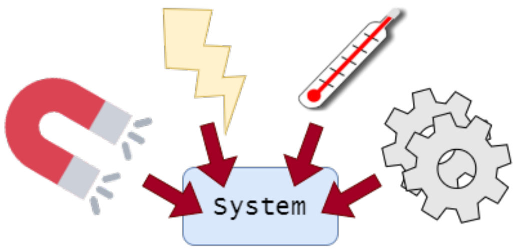


Figure 3.1: Different physical processes interact with the system

For this reason, one of the main challenges is to inject and simulate faults belonging to different physical domains into heterogeneous systems, to refine the fault tolerance of the system under consideration.
Another problem with multi-domain faults is to be able to standardize them so that they can be injected and simulated in different situations and systems.

In fact, until now, only electrical faults have been standardized and can therefore be modeled in different circumstances and different models.
Remaining in the field of multidisciplinary faults, it must also be said that, at the current state of the art and with the tools used so far, it has not yet been possible to generate an efficient and systematic simulation of these faults automatically.

3.2 Proposed Solution

This thesis aims to examine all these previously described issues, using an approach based on Verilog-AMS. Indeed, Verilog-AMS is the Verilog extension that allows us to model and simulate many different physical processes in a very simple way, compared with the others HDL.
Other tools, like Modelica and Simulink, use a block-based notation that limits the fault injection capabilities inside the model, so those faults that alter the internal dynamics of the system can not be reproduced.
Instead, Verilog requires to define the evolution of the system as a set of differential and algebraic equations: this modeling style explicit all system behaviors and thus allows to inject faults and faulty behaviors at any level of detail.
This system description method allows us to define faults changing parameters or using equations that interfere with the physical process under consideration and inject them into the system directly from Verilog-AMS code, adding or modifying a line of code representing the equation in question.

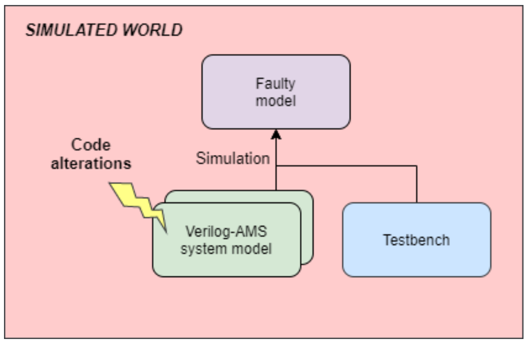


Figure 3.2: Fault injection with Verilog-AMS - Overview

This type of representation has allowed us to better analyze how the physical environment interacts with our systems and this has helped us to identify similarities between the various types of faults, starting from electrical ones

and then considering the mechanical and kinematic ones.
In this way, we have been able to create a taxonomy of the faults that occur in different physical domains (which will be shown in detail and analyzed in the next chapter), and so we can inject faults of the same physical type on different heterogeneous systems.
Besides, using a TCL script, we can set up an architecture to inject and simulate faults, and, using EDA Lab HIF Suite, we can create it for each system model in a systematic way: these aspects are considered and discussed in chapter 5.
Finally, chapter 6 shows the application of the multi-domain faults obtained on some reference models and the respective results obtained.

Chapter 4

Modeling Faults on Heterogeneous Domains

4.1 Known Fault Types

In the electrical discipline, analog faults generally accepted at state of the art are categorized into three main classes, depending on the reproduced physical behavior of electrons in the circuit: short, open, and parametric[2][1].
In detail:

- *parametric faults* model the modification of some quantity with respect to the initial behavior of the system or the specifications, e.g., a variation of the nominal voltage of a voltage source. Such faults reflect many defects in the analog circuit, e.g., different levels of metal caused by the manufacturing process or by aging.
- *short circuit and bridge faults* are caused by an additional communication between two wires (or nodes) that was not present in the initial description (short), or an additional connection of a wire (or a node) to ground with a near-zero resistance value (bridge)[31].
- *open circuit faults* are caused by a missing contact, or by break in a circuit line, so that the current that flows in correspondence to the fault is limited, not completely blocked. In literature, this fault is modeled through a high resistance in series with an edge of the circuit[32].

Such faults will be modeled in the Verilog-AMS code by injecting a resistance or a capacitance, depending on desired effects.
Both open and short circuit faults are implemented as *saboteurs* by adding

a resistor to the description through the following equation:

$$I(p, n) < +V(p, n)/r \tag{4.1}$$

In an open circuit, the r value is extremely high, thus modeling a wiring break that causes the current to cease (even if the electric field will still be present).
In the case of a short circuit, the r value becomes extremely low (a nearly zero ohms path), resulting in a large current flow and a potential difference between the pins of the new path equal to zero.
A different class of faults is *parametric faults*, which represent an alteration in the electrical model's parameters (e.g., of capacitance or a resistance).

4.2 Fault Similarity Analysis

If we abstract the concept of fault as a deviation of the system from its normal behavior, then faults can be applied to any physical domain.
Many analogies exist between conservative domains, and in particular with the electrical domain, so that they can often be simulated through electrical-equivalent circuits.
To look for parallels between the electrical and mechanical (or kinematic) domains, we need to consider the simplification of the two networks. In fact, the networks work on the quantities related to the different disciplines: voltage and current for electrical, corresponding to angular velocity and angular force for mechanical (position and force for kinematic).
The parallelism between quantities allows to define *the dual of electrical faults* in other domains, mapping faults onto the corresponding electrical dual, by looking at the quantity(i.e. potential or flow) that is impacted by the fault. Therefore, with the following fault patterns, it is possible to model the most common alterations of a system.

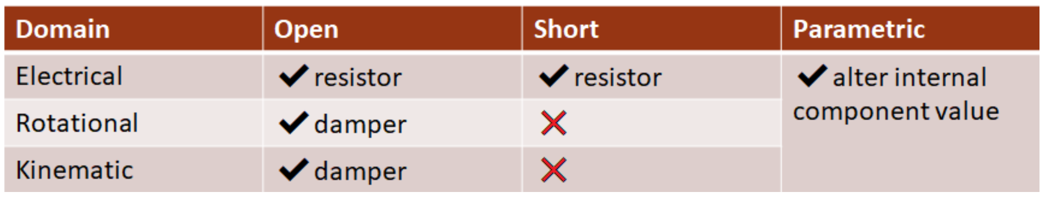
- *Parametric fault:* Parametric-faults are alterations of the parameters that compose the system (e.g. adding or removing connections) but mostly change the configuration parameters. In a mechanical motor model, a possible parametric fault is a variation of the value of the friction coefficient, while in a model of RF MEMS switch can be the variation of the thickness of the bridge. Often these variations are related to the production process, to anomalies in the composition of the materials, or to the effect of system aging, like the wear and tear of materials due to the internal dynamics of the system or external factors caused by the environment in which the system is working[1].

- *Short-dual-fault:* Short-dual-fault represents the dual of the electrical short fault; this fault model is injected into the system by adding new relations (corresponding to new wire connections in the electrical circuit) that reflect the short electrical fault. The short-dual-fault limits the flow of the through variables and changes the potential of the across variable of a system.

- *Open-dual-fault* Open-dual-fault represents the dual of the electrical fault; this fault model is injected into the system by adding new relations that corresponding to a missing contact or by a break in a circuit. In the mechanical domain, injecting an open-dual-fault between a drive shaft and a clutch means that the mechanical connection between these two components is broken; in the kinematic domain instead, injecting a dual open fault means breaking the link between the velocity and the position of the object.

4.3 Fault Taxonomy

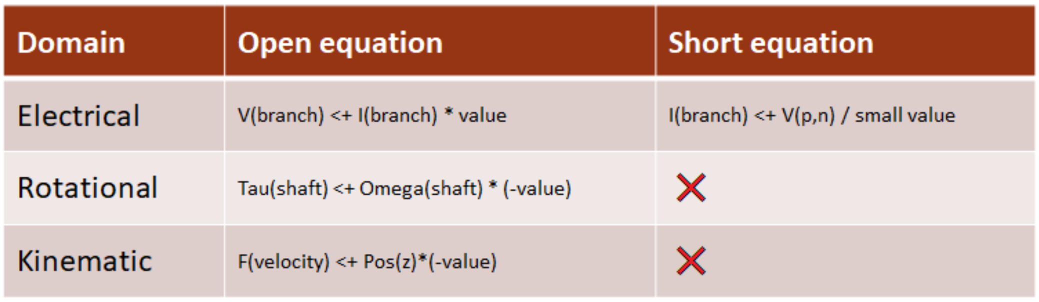
By analyzing the available models with the considerations made above, we were able to obtain a (partial) taxonomy regarding the rotational and kinematic domains.



Domain	Open	Short	Parametric
Electrical	✓ resistor	✓ resistor	✓ alter internal component value
Rotational	✓ damper	✗	
Kinematic	✓ damper	✗	

Figure 4.1: Multi-Discipline Fault Taxonomy

The equivalent of the electrical open is then modeled in the rotational and kinematic domains with a damper, based on the impedance analogy, which is a method of representing a mechanical system by its electrical analogue. Faults in Verilog-AMS models are then injected with the following equations, explained in the next chapters:



Domain	Open equation	Short equation
Electrical	V(branch) <+ I(branch) * value	I(branch) <+ V(p,n) / small value
Rotational	Tau(shaft) <+ Omega(shaft) * (-value)	✗
Kinematic	F(velocity) <+ Pos(z)*(-value)	✗

Figure 4.2: Faults Modeled in Verilog-AMS language

Chapter 5

A Framework for Injecting and Simulating Faults

The creation of the framework for injecting and simulating faults can be divided into two fundamental parts: a first one that aims to the simultaneous and automatic simulation of more different failures on the same model using a TCL script, and a second that instead tries to standardize this procedure, using HIF Suite, in order to make it applicable to various models.

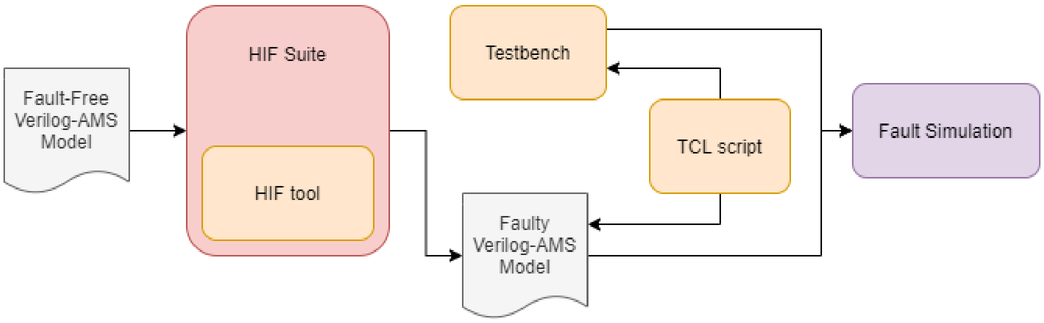


Figure 5.1: Framework for injecting and simulating faults

5.1 Fault Injector Architecture

The first part of the framework creation consists of realizing a performant, fast, and effective procedure to inject multiple faults simultaneously in the same system model. As you can see from the following image, this was accomplished through multiple components:

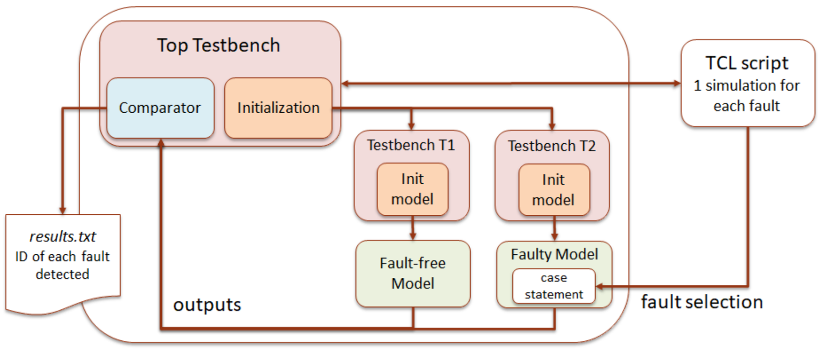


Figure 5.2: Fault injector architecture

To the command of the simulation, there is a simple TCL script that, through a testbench (called top testbench), initializes the models to take into consideration for the simulation.

```
for {set i 0} {$i < 7} {incr i} {
  change :module:ytop:T2:M2:fault_type $i;
  change :module:ytop:CMP:fault_type $i;
  run -all;
  restart;
}

```

The top testbench, as you can see from code:

```
‘include "disciplines.vams"
‘include "constants.vams"
‘timescale 1us / 1us

module comparator(n1, n1_gnd, n2, n2_gnd);
  electrical n1, n2, n1_gnd, n2_gnd;
  input n1, n2, n1_gnd, n2_gnd;
  parameter integer fault_type = 0;
    [...]
endmodule

module top;
  model_test T1();
  model_test_faulty T2();
  comparator CMP(T1.gate, T1.gnd, T2.gate, T2.gnd);
endmodule

```

it actually initializes two testbenches and a comparator.
The first testbench initializes a fault-free model

```
testbench 1
`include "disciplines.vams"
`include "constants.vams"
`timescale 1us / 1us

module mems_test;

  electrical source, drain, gate, gnd;
  ground gnd;
  parameter real val = 1;

  analog begin

    V(source, gnd) <+ 1.65 + 1.65 * sin(2 * `M_PI * val *
   $abstime); // 1 -> freq
    V(drain, gnd) <+ I(drain, gnd) * 1e09;
    V(gate, gnd) <+ 1.65 + 1.65 * sin(2 * `M_PI * 4 *
   $abstime);

  end

  rf_mems_switch M1 (source, drain, gate); // drain -> output
      mems

endmodule

```

while the second one initializes a model that contains within it a case statement, thanks to which the TCL script, shown above, manages to inject faults.

```
testbench 2
`include "disciplines.vams"
`include "constants.vams"
`timescale 1us / 1us

module mems_test_faulty;

  electrical source, drain, gate, gnd;
  ground gnd;
  parameter real val = 1;

  analog begin

    V(source, gnd) <+ 1.65 + 1.65 * sin(2 * `M_PI * val *
   $abstime); // 1 -> freq
```

```
    V(drain, gnd) <+ I(drain, gnd) * 1e09; // load resistance
    V(gate, gnd) <+ 1.65 + 1.65 * sin(2 * `M_PI * 4 *
   $abstime);

  end

  rf_mems_switch_faulty M2 (source, drain, gate); // drain ->
    output mems

endmodule

```

The TCL script executes a simulation for every type of fault entered into the switch case statement of the fault model; at the end of every simulation, the comparator initialized by the top testbench compares the outputs of the fault-free model with those of the faulty model, so that it can determine whether or not the fault has been detected.
At the end of each simulation, the top testbench communicates to the TCL script the termination of the same one, so the script TCL can changes the value of reference for the case statement in order to go to select another fault type for which to make the simulation.

5.2 HIF Tool for Automatic Fault Injection

Trying to simplify things for programmers and to standardize the procedure for choosing the faults to simulate, we used the HIF Suite.

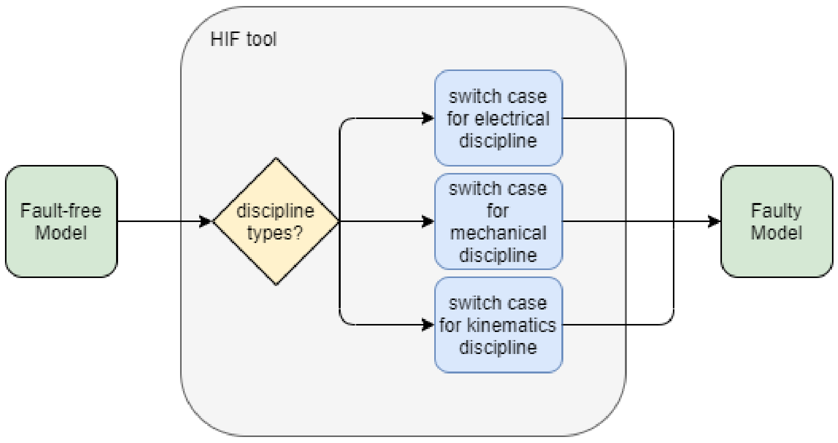

HIF tool
Fault-free Model
discipline types?
switch case for electrical discipline
switch case for mechanical discipline
switch case for kinematics discipline
Faulty Model

Figure 5.3: HIF tool overview

The tool accepts as input a faultless model written in Verilog-AMS and

transforms it in the form of a HIF tree, written in XML.
The design, thus written, can be manipulated by the HIF Suite to be modified in a precise and standard way.
In particular, our tool adds to the model a case statement that can, if necessary, inject the desired faults inside the model.
It recognizes what types of disciplines are being used within the design and, based on that can create the faults that are most appropriate for the situation.

```

Visitor visitor;
visitor.visitSystem(*system);
// Create a copy of the hif input system.
hif::System *copySystemObj = dynamic_cast<hif::System *>(hif
    ::copy(system));~
if (copySystemObj == nullptr) {
  messageError("Error obtaining a copy of System object",
    nullptr, nullptr);
}
/**
* Here call the functions to manipulate the copySystemObj hif
    design.
*/
messageInfo("Visit the manipulated copySystemObj\n");
// Create a visitor to scan the HIF copy design description.
visitor.visitSystem(*copySystemObj);

// Write the manipulated design description to file calling
    Hif::writeFile
std::cout << "\nWriting manipulated design description to
    copy.hif.xml\n";
hif::writeFile("copy.hif.xml", copySystemObj, true);

return 0;

```

The HIF tool needs to have a Visitor, that takes the Verilog-AMS model to examine it.
Below are pieces of code from the part of the tool that creates and injects faults into the hif file based on the type of discipline:

```

auto faultSwitch = new hif::Switch();
faultSwitch->addComment("Switch to inject faults");
faultSwitch->setCondition(_factory.identifier("faultSelector"
    ));
auto switchAlt = new hif::SwitchAlt();

```

Electrical fault:

```
if (electrical){
  std::string branchEl;
  std::string parEl;

  // condition case = 0
  switchAlt->conditions.push_back(_factory.intval(0,_factory.
   integer(nullptr, false)));

  std::cout << "Where do you want to inject the electrical
   fault? (branch name)\n";
  std::cin >> branchEl;
  std::cout << "For which parameter do you want to test it?\n
   ";
  std::cin >> parEl;

  // fault V(branchEl) <+ I(branchEl) * parEl
  switchAlt->actions.push_back(_factory.procedureCallAction("
   hif_verilog_ams_contribution_statement",_factory.instance(
   _factory.library("hif_verilog_vams_standard",  nullptr,
   nullptr, false, true), "vams_standard", _factory.
   noPortAssigns()),
  (_factory.noTemplateArguments()),(_factory.
   parameterArgument("param1", _factory.functionCall("
   hif_verilog_V", _factory.instance(_factory.library("
   hif_verilog_vams_disciplines", nullptr, nullptr, false,
   true), "vams_disciplines", _factory.noPortAssigns()),
   _factory.noTemplateArguments(), _factory.parameterArgument
   ("param1",_factory.identifier(branchEl.c_str())))),
   _factory.parameterArgument("param2", _factory.expression(
   _factory.functionCall("hif_verilog_I", _factory.instance(
   _factory.library("hif_verilog_vams_disciplines", nullptr,
   nullptr, false, true), "vams_disciplines", _factory.
   noPortAssigns()), _factory.noTemplateArguments(), _factory
   .parameterArgument("param1",_factory.identifier(branchEl.
   c_str()))), hif::Operator::op_mult,  _factory.realval(std
   ::stoll(parEl)))))));
}

```

Mechanical fault:

```
if (rotational_omega){
  std::string branchMec;
  std::string parMec;

  // condition case = 1
  switchAlt->conditions.push_back(_factory.intval(1,_factory.
   integer(nullptr, false)));

```

```cpp
  std::cout << "Where do you want to inject the mechanical fault? (branch name)\n";
  std::cin >> branchMec;
  std::cout << "For which parameter do you want to test it?\n";
  std::cin >> parMec;

  // fault Tau(branchMec) <+ Omega(branchMec) * parMec
  switchAlt->actions.push_back(_factory.procedureCallAction("hif_verilog_ams_contribution_statement", _factory.instance(_factory.library("hif_verilog_vams_standard", nullptr, nullptr, false, true), "vams_standard", _factory.noPortAssigns()), (_factory.noTemplateArguments()), (_factory.parameterArgument("param1", _factory.functionCall("hif_verilog_Tau", _factory.instance(_factory.library("hif_verilog_vams_disciplines", nullptr, nullptr, false, true), "vams_disciplines", _factory.noPortAssigns()), _factory.noTemplateArguments(), _factory.parameterArgument("param1",_factory.identifier(branchMec.c_str()))))), _factory.parameterArgument("param2", _factory.expression(_factory.functionCall("hif_verilog_Omega", _factory.instance(_factory.library("hif_verilog_vams_disciplines", nullptr, nullptr, false, true), "vams_disciplines", _factory.noPortAssigns()), _factory.noTemplateArguments(), _factory.parameterArgument("param1", _factory.identifier(branchMec.c_str()))), hif::Operator::op_mult, _factory.realval(std::stoll(parMec)))))));
}

```

Finally, the commands that inject the switch case into the hif tree:

```cpp
faultSwitch->alts.push_back(switchAlt);
o.states.findByName(hif::Str2Name("process"))->actions.push_back(faultSwitch);

```

The HIF tree is then modified, ready to be reconverted by the HIF Suite into Verilog-AMS code so that it can be simulated.

Chapter 6

Experimental Validation

6.1 Motor Model

This section applies the workflow to a DC motor model, which is an electromechanical system that converts electrical energy into mechanical energy through the interaction between the magnetic field of the motor and the electric current in a wire winding[4].

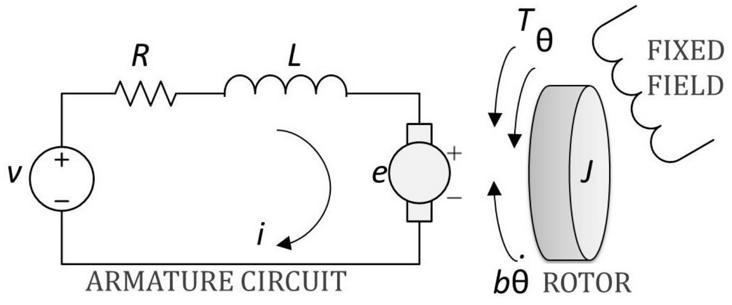


Figure 6.1: DC motor represented as an equivalent circuit of the armature windings and the free-body diagram of the rotor.

Modeling the DC motor in VAMS requires the adoption of electrical, rotational, and rotational_velocity disciplines provided by the standard.
The following constitutive relations model the dynamics of the motor:

$$v = K_M \cdot w + R \cdot i + L \cdot \frac{di}{dt} \tag{6.1}$$

$$\tau = K_T \cdot i - d \cdot w - j \cdot \frac{dw}{dt} \tag{6.2}$$

where v is the input voltage source applied to the motor's windings to control the velocity of the motor; variable i represents the current through the windings; variable w is the angular velocity of the shaft; the variable τ is the torque of the shaft.
K_M e K_T are motor coefficients used to dimension the size of the motor.
The electrical circuit of the DC motor is modeled through an electrical network made of three components: a voltage source (Vm), a resistor (R), and an inductor (L). The three components correspond to the three VAMS equations below.

$$V(p,n) < +Km * Omega(shaft) \tag{6.3}$$

$$V(p,n) < +R * I(p,n) \tag{6.4}$$

$$V(p,n) < +L * ddt(I(p,n)) \tag{6.5}$$

The mechanical behavior of the DC motor is modeled through equations (reported below) that exemplify the interconnection between the electrical domain and the domain of rotational mechanics.

$$Tau(shaft) < ++kt * I(p,n) \tag{6.6}$$

$$Tau(shaft) < +-d * Omega(shaft) \tag{6.7}$$

$$V(p,n) < +-j * ddt(Omega(shaft)) \tag{6.8}$$

To run the motor, the model needs a voltage source, taken from [7]:

```
`timescale 1us / 1us

`include "disciplines.vams"

module vsrc (p, n, dc);

  // PARAMETERS -----------------------
  //parameter real dc=0;  // dc voltage (V)

  // PORTS ---------------------------
  input [9:0] dc;
  output p, n;

  //ground n;

  // NODES ---------------------------
  electrical p, n;

  real vin;

  // BEHAVIOR ------------------------
  analog begin

    @(cross(dc-0.5))

    if (dc > 0.5)
      vin = 3.0;
    else
      vin = 0.0;

    V(p, n) <+ transition(vin, 0, 50u, 50u);

  end
endmodule
```

Then there is the actual engine model:

```
`include "disciplines.vams"
`include "constants.vams"
`timescale 1us / 1us

module motor(shaft_position, p, n);

  // PARAMETERS -----------------------
  parameter real km = 4.5;  // motor constant (V-s/rad)
  parameter real kf = 6.2;  // flux constant (N-m/A) = Kt
  parameter real j = 1.2;   // inertia of shaft (N-m-s2/rad) 0.004
  parameter real d = 0.1;   // drag (friction) (N-m-s/rad) 0.1 -> to 1.5
  parameter real r = 5.0;   // motor winding resistance (Ohms)
  parameter real l = 0.02;  // motor winding inductance (H)

  // PORTS ---------------------------
  output shaft_position;
  input p, n;
  // NODES ---------------------------
  rotational shaft_position, rognd;
  electrical p, n;
  // Internal nodes.
  electrical n1, n2, n3;
  rotational_omega shaft, rgnd;
  // Reference nodes.
  ground rgnd, rognd;
  // BRANCHES ------------------------
  branch (p, n1)  Vm;
  branch (n1, n2) R1;
  branch (n2, n)  L1;
  branch (n2, n3) open01;
  branch (p, n) short01;
  branch (shaft, rgnd) bshaft;
  branch (shaft_position, rognd) bshaftp;

  // BEHAVIOR ------------------------
  analog begin
    // Electrical model of the motor winding.
    V(Vm) <+ km * Omega(bshaft);
    V(R1) <+ r * I(R1);
    V(L1) <+ l * ddt(I(L1));

    // Physical model of the shaft (keep like this).
    Tau(bshaft) <+ +kf * I(Vm);
    Tau(bshaft) <+ -d * Omega(bshaft) -j *ddt(Omega(bshaft));

    // Equation for conversion to degrees.
    Theta(bshaftp) <+ (180 *idt(Omega(bshaft), 0)) / `M_PI;

  end
endmodule
```

6.1.0.a Faulty model creation

The first step is to take our model and inject the faults using the automatic tool created in the HIF Suite environment.
This is the result achieved:

```
`include "disciplines.vams"
`include "constants.vams"
`timescale 1us / 1us

module motor_faulty(shaft_position, p, n);

  // PARAMETERS -----------------------
  parameter real km = 4.5;  // motor constant (V-s/rad)
  parameter real kf = 6.2;  // flux constant (N-m/A) = Kt
  parameter real j = 1.2;   // inertia of shaft (N-m-s2/rad)
   0.004
  parameter real d = 0.1;   // drag (friction) (N-m-s/rad)
   0.1 -> to 1.5
  parameter real r = 5.0;   // motor winding resistance (Ohms
   )
  parameter real l = 0.02;  // motor winding inductance (H)
  parameter real fault = 0;

  // PORTS ---------------------------
  output shaft_position;
  input p, n;
  // NODES ---------------------------
  rotational shaft_position, rognd;
  electrical p, n;
  // Internal nodes.
  electrical n1, n2, n3;
  rotational_omega shaft, rgnd;
  // Reference nodes.
  ground rgnd, rognd;
  // BRANCHES ------------------------
  branch (p, n1)  Vm;
  branch (n1, n2) R1;
  branch (n2, n)  L1;
  branch (n2, n3) open01;
  branch (p, n) short01;
  branch (shaft, rgnd) bshaft;
  branch (shaft_position, rognd) bshaftp;

  // BEHAVIOR ------------------------
  analog begin
    // Electrical model of the motor winding.
    V(Vm) <+ km * Omega(bshaft);
    V(R1) <+ r * I(R1);
    V(L1) <+ l * ddt(I(L1));

    // Physical model of the shaft (keep like this).
    Tau(bshaft) <+ + kf * I(Vm);
    Tau(bshaft) <+ -d * Omega(bshaft) -j *ddt(Omega(bshaft));

    // Equation for conversion to degrees.
    Theta(bshaftp) <+ (180 * idt(Omega(bshaft), 0)) / `M_PI;

    case (fault)
      0: begin
        $display("0");
      end
      1: begin
        V(open01) <+ I(open01) * 1e06;
        $display("1");
      end
      2: begin
        I(short01) <+ V(short01) / 1;
        $display("2");
      end
      3: begin
        Tau(bshaft) <+ Omega(bshaft) * -1e06;
        $display("3");
      default:
        $display("Error case analog");
    endcase

  end
endmodule
```

6.1.0.b Simulation and results

Below we can see the graphs representing the simulation of the fault-free model:

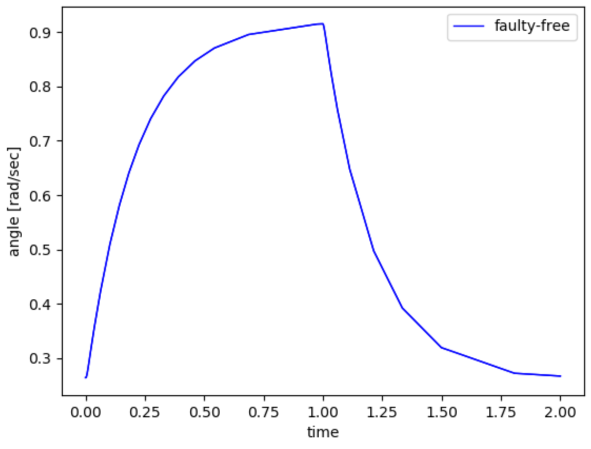


Figure 6.2: Angular velocity of the rotor of the faulty-free description

In particular, we can observe the waveform of the angular velocity and its evolution.
Now we can analyze the differences with the first injected fault: the open fault.

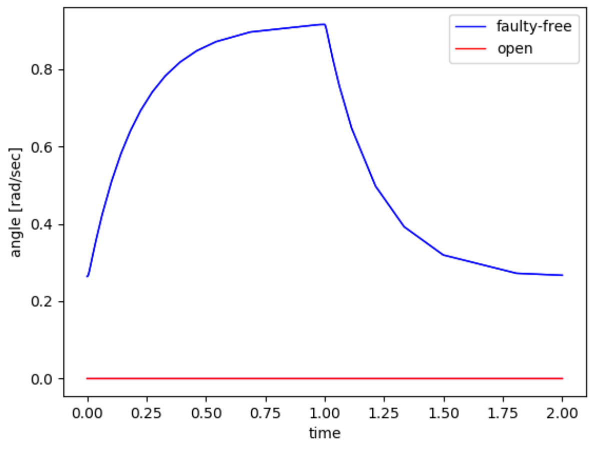


Figure 6.3: Angular velocity of the rotor with open fault

As can be seen, the open fault is destructive, in fact, the rotor does not rotate, despite the applied input voltage.
Analyzing instead the second one, the short fault:

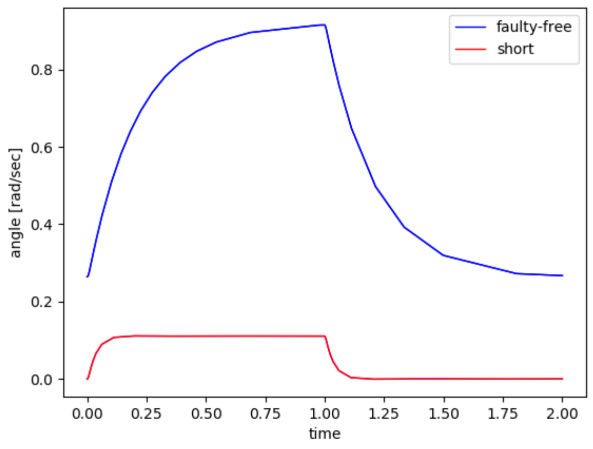


Figure 6.4: Angular velocity of the rotor with short fault

The short fault causes variations in respect to the faulty free behavior, decreasing the rotor velocity and varying also how quickly the new velocity is reached. This fault is not destructive for the motor, that still rotates, even if at different velocities.
Finally, the last proposed failure (the mechanical fault):

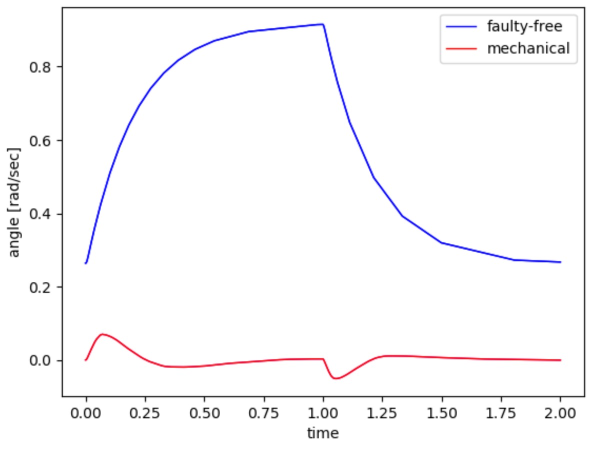


Figure 6.5: Angular velocity of the rotor with mechanical fault

This fault in the mechanical domain causes the engine to break, as the graph shows that the rotor rotates unexpectedly.

6.1.0.c Further Considerations

Looking at the state-of-the-art results (and trying to replicate them) we also find that different values used to inject the same fault cause different engine output behaviors.

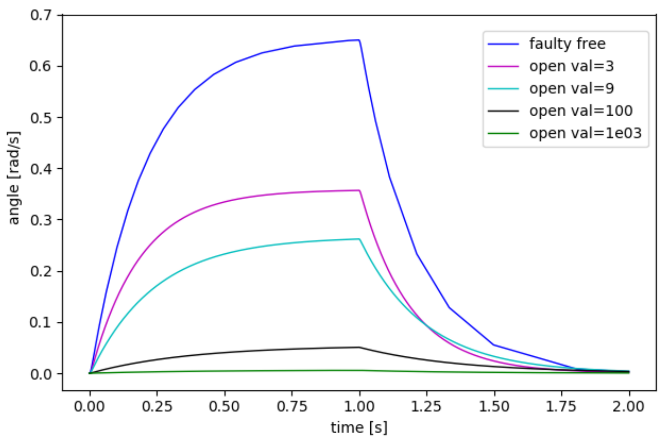


Figure 6.6: Angular velocity of the rotor of the DC motor for the faulty-free description and in presence of an open circuit injected in the electrical part with different levels of severity

This shows us how the parameter value of the fault can impact on the motor response.

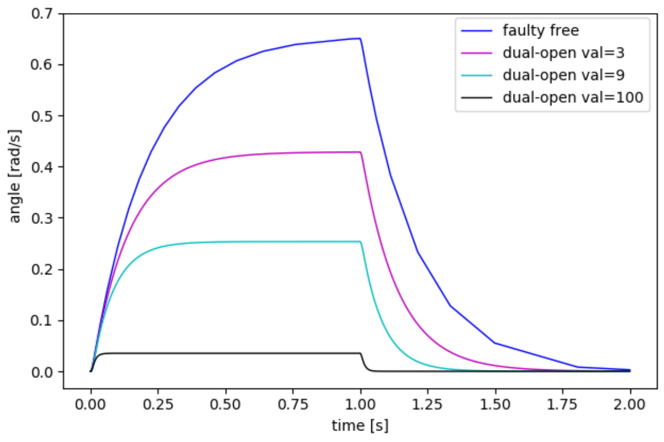


Figure 6.7: Angular velocity of the rotor of the DC motor for the faulty-free description and in presence of an open-dual fault injected in the mechanical part with different levels of severity

Injecting and simulating mechanical failure with different levels of intensity shows us how the value of the failure can impact the entire engine. The graph shows that a higher value of the fault causes less movement of the crankshaft.

6.2 MEMS Switch Model

This section applies the workflow to an ohmic cantilever RF MEMS Switch, Which is an asymmetrical device with the clamp designated as a *source*, the bias electrode designated as a *gate*, and the contact electrode designated as a *drain*.

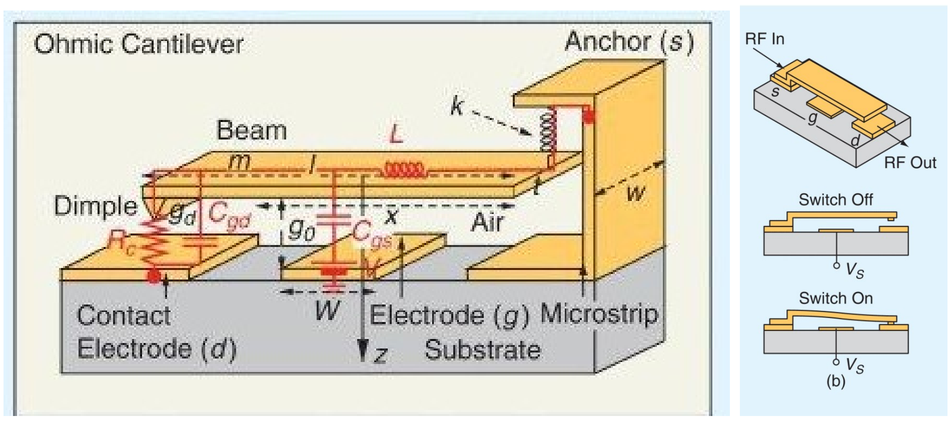


Figure 6.8: RF MEMS switch overview

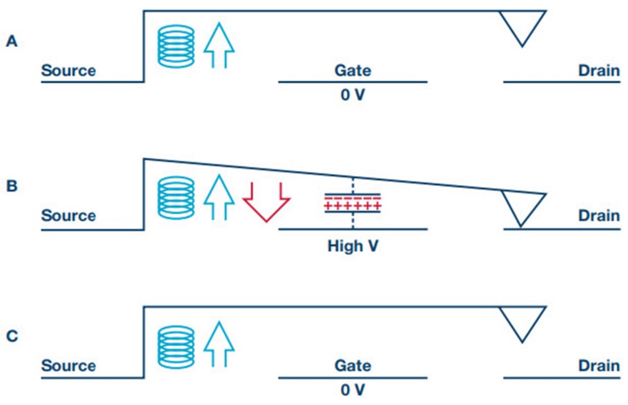


Figure 6.9: RF MEMS switch functioning

It is a multi-disciplinary model, which combines electrical and kinematic disciplines.
Velocity and gate position can be represented by the formulas:

$$\dot{z} = v \tag{6.9}$$

$$\dot{v} = (-b(z)v - F_s(z) + F_e(z, V) + F_c(z) + F_n(z))/m \tag{6.10}$$

Where v is the velocity of the gate, z his position; F_s, F_e, F_c and F_n are forces acting on the bar.
These equations can be modeled in Verilog-AMS as:

$$Vel(velocity) < +ddt(Pos(z)) \tag{6.11}$$

$$Pos(z) : ddt(Pos(velocity)) == 1/m*(-b*Pos(velocity)-F_s+F_e+F_c+Noise\_Elements \tag{6.12}$$

The RF MEMS switch model is taken from [7]:

```
`include "disciplines.vams"
`include "constants.vams"

module rf_mems_switch(s,d,g);
  inout s, d, g;
  electrical s, d, g;
  kinematic z;
  kinematic_v velocity;

  // Dimensions
  parameter real g_0=0.6e-6 from (0:inf); // nominal gap
    height [m]
  parameter real g_d=0.2e-6 from (0:inf); // nominal gap
    height minus dimple thickness [m]
  parameter real l=75e-6 from (0:inf); // beam length [m]
  parameter real t=5e-6 from (0:inf); // beam thickness [m]
  parameter real w=30e-6 from (0:inf); // beam width [m]
  parameter real W_c=10e-6 from (0:inf); // contact width [m]
  parameter real W=10e-6 from (0:inf); // electrode width [m]
  parameter real x=30e-6 from (0:inf); // beam anchor to
    electrode edge distance [m]
  // Electrical parameters
  parameter real R_c=1.0 from [0:inf); // contact resistance
    [Ohm]
  // Material parameters: beam (gold)
  parameter real rho=19.2e3 from (0:inf); // mass density [kg
    /m^3]
  parameter real E=78e9 from (0:inf); // Young's modulus [Pa]
  // Material parameters: gas (air):
  parameter real lambda=1.5e-7 from [0:inf); // mean-free
    path [m]
  parameter real mu=1.845e-5 from (0:inf); // viscosity
    coefficient [kg/(m.s)]
  // van der Waals and nuclear force coefficients
  parameter real c_1=10e-80 from [0:inf); // van der Waals
    force coefficient [N.m]
  parameter real c_2=10e-75 from [0:inf); // nuclear force
    coefficient [N.m^8]

```

```
real A, b, C_ds, C_gs, C_u, F_c, F_e, F_s, f_m_0, I1, I2, IIP3, k_1, m, Q_e, Q_m, t_s_max, t_s_min, V_H, V_P, V_S;

//branch (z, velocity) b1;

analog begin

  @ ( initial_step ) begin
    A=W*w;
    $strobe("\n%M: A (electrode area [m^2]) = %E",A);
    m=0.4*rho*l*t*w;
    $strobe("%M: m (effective beam mass [kg]) = %E",m);
    k_1=2*E*w*pow(t/l,3)*(1-x/l)/(3-4*pow(x/l,3)+pow(x/l,4));
    $strobe("%M: k_1 (spring constant [N/m]) = %E",k_1);
    V_H=sqrt(2*k_1*(g_0-g_d)*pow(g_d,2)/(`P_EPS0*A));
    $strobe("%M: V_H (hold-down voltage [V]) = %E",V_H);
    V_P=sqrt(8*k_1*pow(g_0,3)/(27*A*`P_EPS0));
    $strobe("%M: V_P (pull-in voltage [V]) = %E",V_P);
    f_m_0=sqrt(k_1/m)/(2*`M_PI);
    $strobe("%M: f_m_0 (mechanical resonant frequency [Hz]) = %E",f_m_0);
    Q_m=(sqrt(E*rho)*pow(t,2)*pow(g_0,3))/(mu*pow(w*l,2));
    $strobe("%M: Q_m (mechanical quality factor []) = %E",Q_m);
    V_S=V_P;
    t_s_max=27*pow(V_P,2)/(4*2*`M_PI*f_m_0*Q_m*pow(V_S,2));
    $strobe("%M: t_s_max (maximum switching time [s], V_S = V_P) = %E",t_s_max);
    t_s_min=3.67*V_P/(V_S*sqrt(k_1/m));
    $strobe("%M: t_s_min (minimum switching time [s], V_S = V_P) = %E",t_s_min);
    C_u=`P_EPS0*W_c*w/g_0;
    $strobe("%M: C_u (up-state capacitance [F]) = %E",C_u);
    IIP3=10*log(2*k_1*pow(g_0,2)/(`M_PI*10e9*pow(C_u*50,2)))+30;
    $strobe("%M: IIP3 (third order intercept [dBm], C=C_u, delta_f < f_m_0, f=10 GHz, Z=50 Ohm) = %E",IIP3);
  end

  // b: damping coefficient [N.s/m] (displacement-compensated squeeze-film damping)
  Q_e=Q_m*pow(1.1-pow(g_d*tanh(Pos(z)/g_d)/g_0,2),1.5)*(1+9.9638*pow(lambda/(g_0-g_d*tanh(Pos(z)/g_d)),1.159));
  $strobe("\n%M: Q_e (displacement-compensated mechanical quality factor []) = %E",Q_e);
  b=k_1/(2*`M_PI*f_m_0*Q_e);
  $strobe("%M: b (damping coefficient [N.s/m]) = %E",b);

```

```
    // Forces
    F_c=c_1*W_c*w/pow(g_d-g_d*tanh(Pos(z)/g_d)+1e-9,3)-c_2*W_c*w/pow(g_d-g_d*tanh(Pos(z)/g_d)+1e-9,10);
    $strobe("%M: F_c (attractive van der Waals and repulsive nuclear forces [N]) = %E",F_c);
    F_e=`P_EPS0*A*pow(V(s,g),2)/2*1/pow(g_0-Pos(z),2);
    $strobe("%M: F_e (electrostatic force [N]) = %E",F_e);
    F_s=k_1*Pos(z);
    $strobe("%M: F_s (spring force [N]) = %E",F_s);

    Vel(velocity)<+ ddt(Pos(z));
    Pos(z):ddt(Vel(velocity))==1/m*(-b*Vel(velocity)-F_s+F_e+F_c+white_noise(4*`P_K*$temperature*b, "BROWNIAN_NOISE"));

    C_ds=`P_EPS0*W_c*w/(g_0-Pos(z));
    I1=ddt(C_ds*V(d,s));
    I2=V(d,s)/R_c+white_noise(4*`P_K*$temperature/R_c, "JOHNSON_NYQUIST_NOISE_R_c");
    if (Pos(z)<g_d)
      I(d,s)<+I1;
    else
      I(d,s)<+I2;

    C_gs=`P_EPS0*A/(g_0-Pos(z));
    I(g,s)<+ddt(C_gs*V(g,s));
  end
endmodule
```

6.2.0.a Faulty model creation

The first step is to take our model and inject the faults using the automatic tool created in the HIF Suite environment.
This is the result achieved:

```
`include "disciplines.vams"
`include "constants.vams"

module rf_mems_switch_faulty(s,d,g);
  inout s, d, g;
  electrical s, d, g;
  kinematic z;//, velocity;
  kinematic_v velocity;

  parameter integer fault_type = 0;

  // Dimensions
  parameter real g_0=0.6e-6 from (0:inf); // nominal gap height [m]
parameter real g_d=0.2e-6 from (0:inf); // nominal gap
  height minus dimple thickness [m]

parameter real l=75e-6 from (0:inf); // beam length [m]
parameter real t=5e-6 from (0:inf); // beam thickness [m]
parameter real w=30e-6 from (0:inf); // beam width [m]
parameter real W_c=10e-6 from (0:inf); // contact width [m]
parameter real W=10e-6 from (0:inf); // electrode width [m]
parameter real x=30e-6 from (0:inf); // beam anchor to
  electrode edge distance [m]
// Electrical parameters
parameter real R_c=1.0 from [0:inf); // contact resistance
  [Ohm]
// Material parameters: beam (gold)
parameter real rho=19.2e3 from (0:inf); // mass density [kg
  /m^3]
parameter real E=78e9 from (0:inf); // Young's modulus [Pa]
// Material parameters: gas (air):
parameter real lambda=1.5e-7 from [0:inf); // mean-free
  path [m]
parameter real mu=1.845e-5 from (0:inf); // viscosity
  coefficient [kg/(m.s)]
// van der Waals and nuclear force coefficients
parameter real c_1=10e-80 from [0:inf); // van der Waals
  force coefficient [N.m]
parameter real c_2=10e-75 from [0:inf); // nuclear force
  coefficient [N.m^8]

real A, b, C_ds, C_gs, C_u, F_c, F_e, F_s, f_m_0, I1, I2,
  IIP3, k_1, m, Q_e, Q_m, t_s_max, t_s_min, V_H, V_P, V_S;

analog begin

  @ ( initial_step ) begin
    A=W*w;
    $strobe("\n%M: A (electrode area [m^2]) = %E",A);
    m=0.4*rho*l*t*w;
    $strobe("%M: m (effective beam mass [kg]) = %E",m);
    k_1=2*E*w*pow(t/l,3)*(1-x/l)/(3-4*pow(x/l,3)+pow(x/l,4)
  );
    $strobe("%M: k_1 (spring constant [N/m]) = %E",k_1);
    V_H=sqrt(2*k_1*(g_0-g_d)*pow(g_d,2)/(`P_EPS0*A));
    $strobe("%M: V_H (hold-down voltage [V]) = %E",V_H);
    V_P=sqrt(8*k_1*pow(g_0,3)/(27*A*`P_EPS0));
    $strobe("%M: V_P (pull-in voltage [V]) = %E",V_P);
    f_m_0=sqrt(k_1/m)/(2*`M_PI);
    $strobe("%M: f_m_0 (mechanical resonant frequency [Hz])
   = %E",f_m_0);
    Q_m=(sqrt(E*rho)*pow(t,2)*pow(g_0,3))/(mu*pow(w*l,2));
    $strobe("%M: Q_m (mechanical quality factor []) = %E",
Q_m);
    V_S=V_P;
    t_s_max=27*pow(V_P,2)/(4*2*`M_PI*f_m_0*Q_m*pow(V_S,2));
    $strobe("%M: t_s_max (maximum switching time [s], V_S =
 V_P) = %E",t_s_max);
    t_s_min=3.67*V_P/(V_S*sqrt(k_1/m));
    $strobe("%M: t_s_min (minimum switching time [s], V_S =
 V_P) = %E",t_s_min);
    C_u=`P_EPS0*W_c*w/g_0;
    $strobe("%M: C_u (up-state capacitance [F]) = %E",C_u);
    IIP3=10*log(2*k_1*pow(g_0,2)/(`M_PI*10e9*pow(C_u*50,2))
)+30;
    $strobe("%M: IIP3 (third order intercept [dBm], C=C_u,
delta_f < f_m_0, f=10 GHz, Z=50 Ohm) = %E",IIP3);
 end

 // b: damping coefficient [N.s/m] (displacement-
compensated squeeze-film damping)
 Q_e=Q_m*pow(1.1-pow(g_d*tanh(Pos(z)/g_d)/g_0,2),1.5)
*(1+9.9638*pow(lambda/(g_0-g_d*tanh(Pos(z)/g_d)),1.159));
 $strobe("\n%M: Q_e (displacement-compensated mechanical
quality factor []) = %E",Q_e);
 b=k_1/(2*`M_PI*f_m_0*Q_e);
 $strobe("%M: b (damping coefficient [N.s/m]) = %E",b);

 // Forces
 F_c=c_1*W_c*w/pow(g_d-g_d*tanh(Pos(z)/g_d)+1e-9,3)-c_2*
W_c*w/pow(g_d-g_d*tanh(Pos(z)/g_d)+1e-9,10);
 $strobe("%M: F_c (attractive van der Waals and repulsive
nuclear forces [N]) = %E",F_c);
 F_e=`P_EPS0*A*pow(V(s,g),2)/2*1/pow(g_0-Pos(z),2);
 $strobe("%M: F_e (electrostatic force [N]) = %E",F_e);
 F_s=k_1*Pos(z);

 Vel(velocity)<+ ddt(Pos(z));
 Pos(z):ddt(Vel(velocity))==1/m*(-b*Vel(velocity)-F_s+F_e+
F_c+white_noise(4*`P_K*$temperature*b, "BROWNIAN_NOISE"));

 C_ds=`P_EPS0*W_c*w/(g_0-Pos(z));
 I1=ddt(C_ds*V(d,s));
 I2=V(d,s)/R_c+white_noise(4*`P_K*$temperature/R_c, "
JOHNSON_NYQUIST_NOISE_R_c");
 if (Pos(z)<g_d)
    I(d,s)<+I1;
 else
    I(d,s)<+I2;

 C_gs=`P_EPS0*A/(g_0-Pos(z));
```

```
    case (fault_type)
      0: begin
        I(g,s)<+ddt(C_gs*V(g,s));
        $display("fault 0: no faults");
      end
      1: begin
        I(g,s)<+ddt(C_gs*V(g,s));
        F(velocity)<+Pos(z)*-1e06;
      end
      2: begin
        I(g,s)<+V(g,s)/1; //short;
      end
      3: begin
        I(g,s)<+ddt(C_gs*V(g,s));
        V(d,s)<+I(d,s)*1e03; //open
      end
      default:
        $display("Error case analog");
    endcase

  end
endmodule
```

6.2.0.b Simulation and results

Below we can see the graphs representing the simulation of the fault-free model:

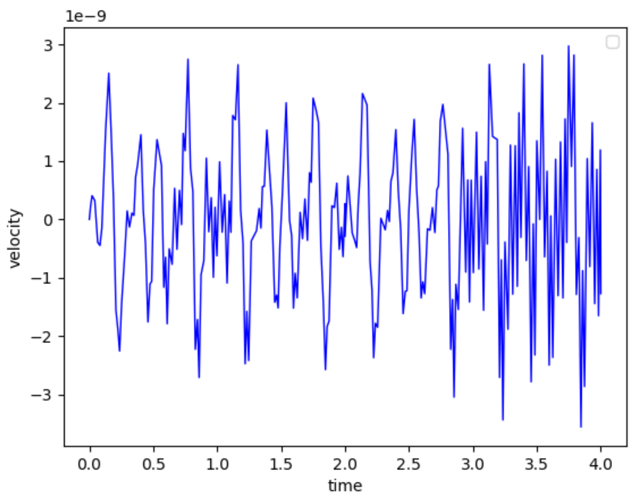


Figure 6.10: Velocity fault free model

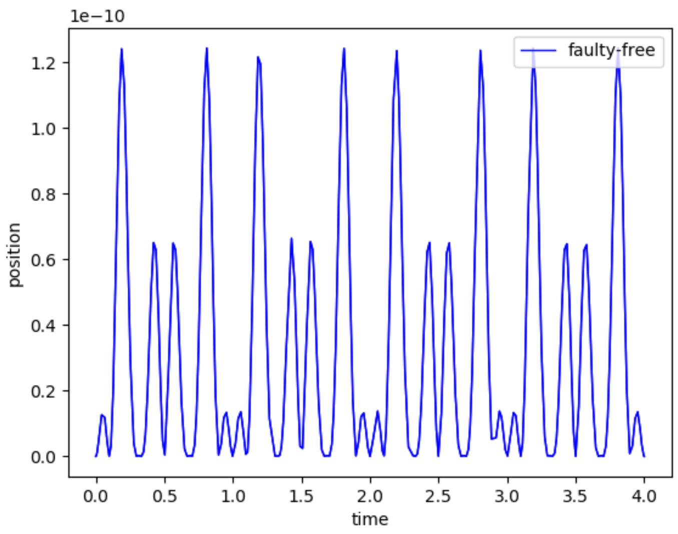


Figure 6.11: Position fault free model

In particular, we can observe the waveform of velocity and position that, when varying, modify the component of electricity transmitted inside the model as can be observed from the Verilog-AMS code.

Now we can analyze the differences with the first injected fault: the kinematic fault.

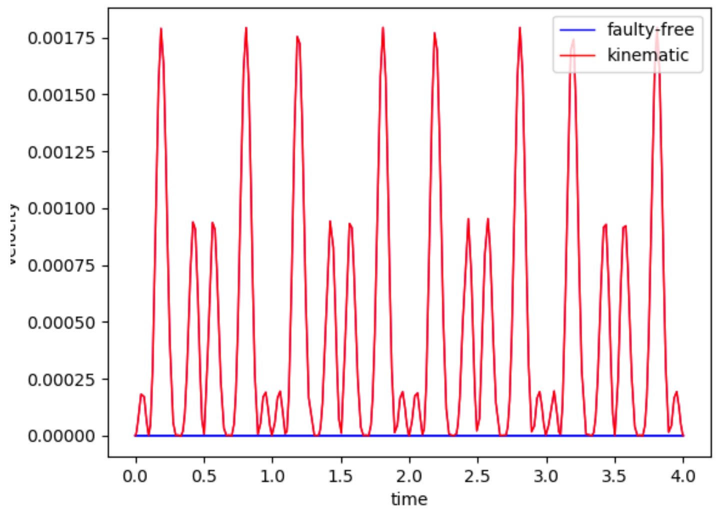


Figure 6.12: Velocity kinematic fault

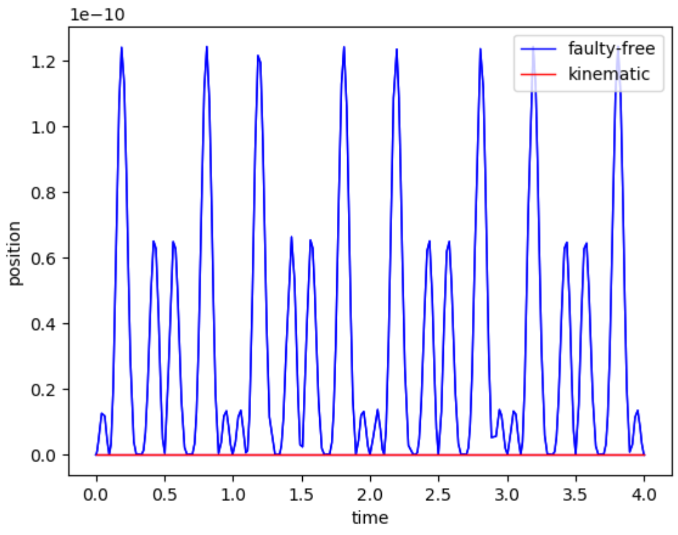
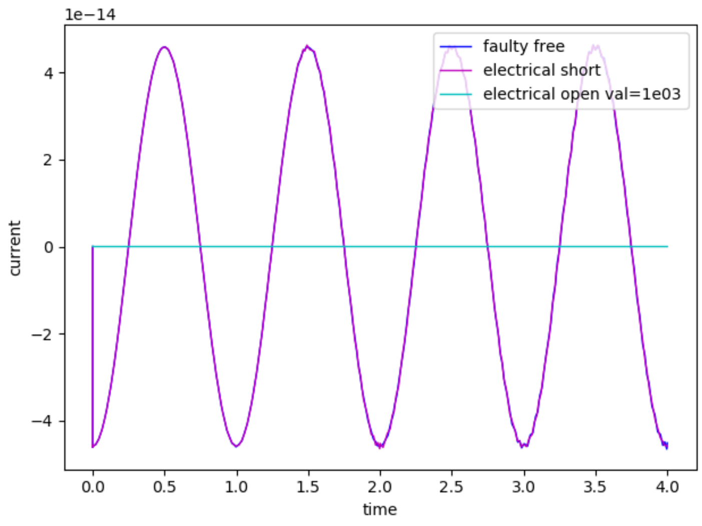


Figure 6.13: Position kinematic fault

As can be seen, the mechanical fault modifies both the speed and position of the MEMS stick, causing as reaction a different current contribution in the device. This makes us understand that the MEMS is broken.

Instead, analyzing the electrical faults (second and third injected faults)



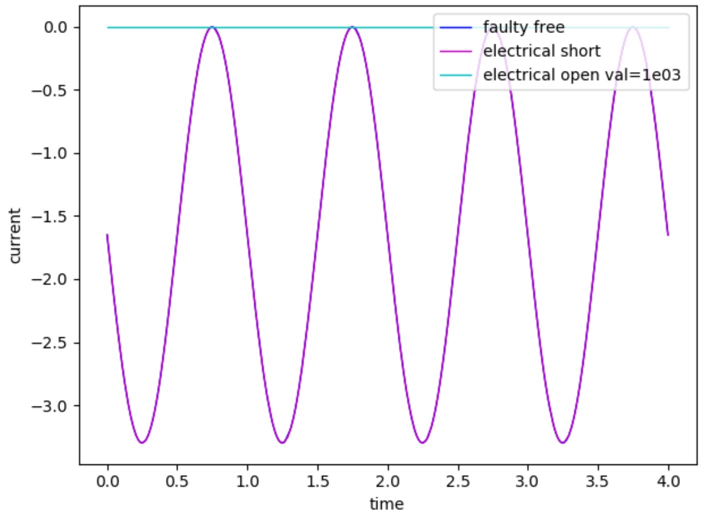


Figure 6.14: Open and short fault change the current contribution

The two graphs represent the two current contributions that are transmitted inside the device at different times depending on the position of the rod.
As can be seen, all injected faults produce values that are discordant with what should be the normal behavior of the device.

Chapter 7

Concluding Remarks and Future Works

This work aims to analyze multi-domain faults in order to create a standard, from which to analyze and simulate them more effectively.
To do this, known state-of-the-art faults are analyzed and similarities with faults from different disciplines are identified.
Subsequently, there is a need to create an automated tool to systematically inject and simulate this multitude of faults within a model.
This is done first by creating a HIF Suite tool that, given a fault-free Verilog-AMS model as input, outputs a faulted Verilog-AMS model based on the types of disciplines present within.
In a second moment, with the model correctly created by the tool, a TCL script allows to conduct a systematic simulation of all the multi-domain faults, analyzing and comparing them with the starting model, and being able to define if the fault is tested or not by the equation injected in the model.
The results obtained from the use of our models with this simulation framework, allow us to state that the tool produces correct results with respect to expectations on multi-domain faults and, moreover, that it can be an advantageous tool for the systematic simulation of multiple faults of different types.

7.1 Next Steps

Future work will focus on improving the tool implemented in order to increase the level of automation and expand the knowledge of multi-disciplinary models.
The main points are:

- model faults from other disciplines;
- expand knowledge of non-electrical faults by completing the taxonomy created;
- take into account for simulations real-life failures;
- increase the degree of automation of the framework to inject and simulate faults;

Bibliography

[1] N. Dall'Ora, S. Vinco, and F. Fummi, "Functionality and fault modeling of a dc motor with verilog-ams," 2020.

[2] N. Dall'Ora, S. Vinco, and F. Fummi, "Multi-discipline fault modeling with verilog-ams," 2020.

[3] *Verilog-A Reference Manual.* Agilent Technologies, 2005.

[4] S.H.Kim, "Control of direct current motor," *Electric Motor Control*, pp. 39–93, 2017.

[5] F. Fraccaroli, M. Lora, F. Fummi, and P. Montuschi, "A fast simulation environment for smart systems validation in presence of electromagnetic interferences," 2016.

[6] O. Kenneth, S. Kundert andd Zinke, *The Designer's Guide to Verilog-AMS.* 2004.

[7] [online], "The designer's guide to verilog-ams," `www.designers-guide.org`.

[8] *Verilog-AMS Language Reference Manual.* Accellera Systems Initiative, 2014.

[9] [online], "Cnr," `www.cnr.it`.

[10] K. V. Caekenberghe, "Modeling rf mems devices," 2012.

[11] D. Maithripala, J. M. Berg, and W. Dayawansa, "Control of an electrostatic mems using static and dynamic output feedback," 2004.

[12] M. Donna, "Interfacing vhdl and verilog designs to c++ models," 2002.

[13] [online], "Verilog pli tutorial," `www.asic-world.com/Verilog/veritut.html`.

[14] S. Sutherland, "The verilog pli is dead (maybe) long live the systemverilog dpi!," 2004.

[15] [online], "Using tcl to create a virtual component in verilog," `https://www.eetimes.com/using-tcl-to-create-a-virtual-component-in-verilog/`.

[16] M. G. Corporation, *EZwave User's Guide.* 2009.

[17] [online], "Tcl basics," `https://classes.engineering.wustl.edu/ese461/Lecture/week5b.pdf`.

[18] [online], "Tcl tutoral," `https://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html`.

[19] *Eldo User's Manual.* Mentor Graphics Corporation, 2009.

[20] *Questa ADMS User's Manual.* Mentor Graphics Corporation, 2009.

[21] *Eldo RF User's Manual.* Mentor Graphics Corporation, 2009.

[22] [online], "Conditional statements - documentation," `https://verilogams.com/refman/modules/analog-procedural/conditional.html`.

[23] [online], "Eda lab suite," `https://edalab.it/hifsuite/`.

[24] F. Pecheux, C. Lallement, and A. Vachoux, "Vhdl-ams and verilog-ams as alternative hardware description languages for efficient modeling of multidiscipline systems," *IEEE Trannsactions on Computer-Aided Design of multidiscipline systems*, vol. 24, no. 2, pp. 204–225, 2005.

[25] K.Kundert and O.Zinke, *The designer's guide to Verilog-AMS.* Springer Science & Business Media, 2006.

[26] P.Frey and D.O'Riordan, "Verilog-ams: Mixed-signal simulation and cross domain connect modules," *in Proc. of IEEE/ACM International Workshop on Behavioral Modeling and Simulation*, 2000.

[27] e. a. C.Chen, "Research on mechanical fault injection method based on adams," *2019 IEEE 3rd Advanced Information Management, Comunicates, Electronic and Automation Control Conference (IMCEC). IEEE*, 2019.

[28] G. McCann, S. Herwald, and H. Kirschbaum, "Electrical analogy methods applied to servomechanism problems," *Transactions of the American Institute of Electrical Engineers*, vol. 65, no. 2, pp. 91–96, 1946.

[29] P.Kroes, "Structural analogies between physical systems," *The British Journal for the Philosophy of Science*, vol. 40, no. 2, pp. 37–50, 1976.

[30] E. Fraccaroli, M. Lora, and F. Fummi, "Automatic generation of analog/mixed signal virtual platforms for smart systems," *IEEE Transactions on Computers*, pp. 1–1, 2020.

[31] H. Vierhaus, W. Meyer, and U. Glaser, "Cmos bridges and restrictive transistor faults: Iddq versus delay effects," *Proceedings of IEEE International Test Conference - (ITC)*, 1993.

[32] C. Henderson, J. Soden, and C. Hawkins, "The behavior and testing implications of cmos ic logic gate open circuits," *Proocedings. International Test Conference*, 1991.