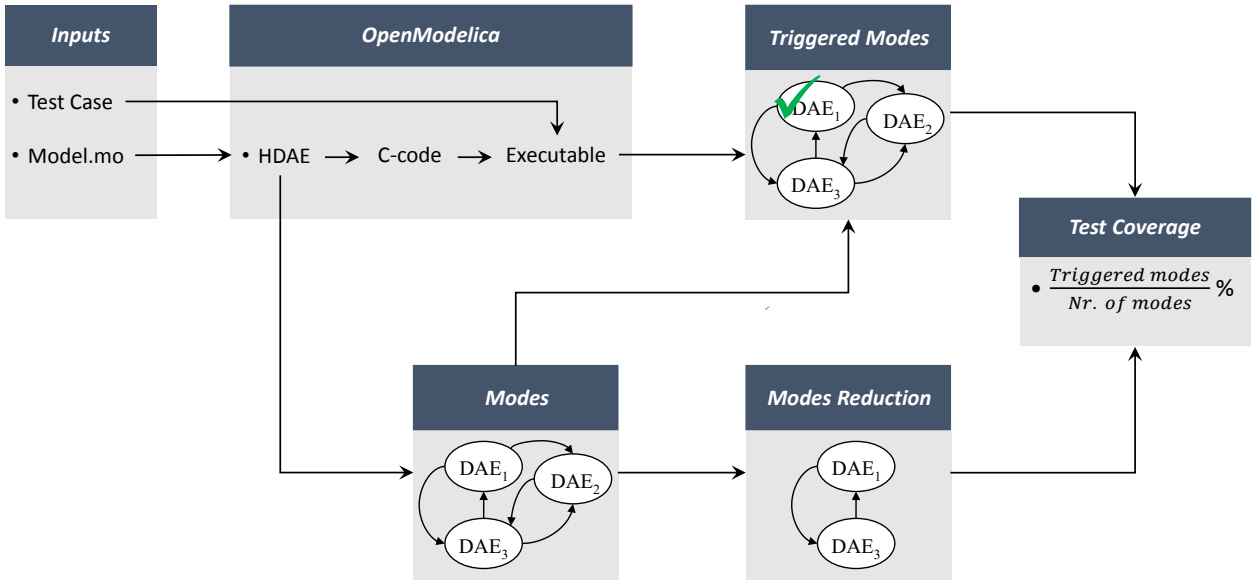
CHALMERS
UNIVERSITY OF TECHNOLOGY



Test coverage of systems with continuous dynamics

Master's thesis in Systems, Control and Mechatronics.

JAVIER GIL CEPEDA

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Master's thesis 2017: EX002/2017

Test coverage for systems with continuous dynamics

JAVIER GIL CEPEDA

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Signals and Systems
Division of Signals and Control
Chalmers University of Technology
Gothenburg, Sweden 2017

Test coverage for systems with continuous dynamics
JAVIER GIL CEPEDA

Supervisor: Sajed Miremadi, Volvo Car Corporation
Examiner: Knut Åkesson, Department of Signals and Systems

Master's Thesis 2017:EX002/2017
Department of Signals and Systems
Division of Signals and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover:

Typeset in LaTeX
Gothenburg, Sweden 2017

Test coverage for systems with continuous dynamics
JAVIER GIL CEPEDA
Department of Signals and Systems
Chalmers University of Technology

Abstract

Testing cyber-physical systems is a crucial part of the development process to ensure correct operation. To determine how well a system is tested, this thesis proposes a novel methodology to measure *test coverage* for systems that contain both continuous and discrete dynamics. The approach is based on using the modelling language Modelica to obtain a mathematical representation of the behaviours of the system that can be translated to a hybrid automaton. Then, a metric called mode coverage is used to measure the exercised dynamical behaviours during the testing process.

This work first describes mechatronic systems and how they can be modelled as hybrid automata, a case-study from the automotive industry is presented as an example. In a second stage, coverability criteria are presented as a metric to evaluate the quality of the test, and two strategies to perform test coverage are identified and analysed. Finally, a strategy based on structural analysis of modes is implemented to the case study and the proposed algorithm is described in detail.

On the basis of the results, it can be concluded that the proposed methodology can be used to pinpoint dynamics that are not examined during the testing process. Such information can be further used to automate the generation of new test cases that trigger the unexplored regions.

Keywords: Test coverage, mode coverage, hybrid automata, Model-Based Testing, Automation.

Acknowledgements

I would like to express my gratitude to all the people who contributed to the work described in this thesis. First of all, I thank my thesis advisor Dr. Knut Åkesson for engaging me in new ideas and guidance throughout the project. I would also like to thank my supervisor at Volvo, Dr. Sajed Miremadi for giving me this thesis opportunity but also supporting me to overcome problems that emerged during the thesis work. Finally, I would like to send my thanks out to Johan Eddeland for his clever and valuable insights.

Javier Gil Cepeda, Gothenburg, January 2017

Acronyms

CPSs	Cyber-Physical Systems
DAE	Differential Algebraic Equations
GCOV	GNU Coverage
HA	Hybrid Automata
HDAE	Hybrid Differential Algebraic Equations
MBT	Model-Based Testing
MC/DC	Modified Condition/Decision Coverage
SMT	Satisfiability Modulo Theories
SUT	System Under Test

Table of Contents

List of Figures

List of Tables

1
Introduction

Test Coverage is a measure extensively used in software engineering to characterise how well a model is tested. This thesis provides a definition of test coverage for systems with continuous dynamics and develops an algorithm to measure test coverage. The algorithm has been evaluated on a use case from Volvo Car Corporation.

This master's thesis is a part of TESTRON [1] project carried out by Chalmers University of Technology, Volvo Car Corporation and Quviq AB and funded by VINNOVA, the Sweden Innovation Agency.

1.1 Background

Cyber-Physical Systems (CPSs) [2] [3] are closed loop feedback systems that present high interconnection between the physical world and one or more computing devices. As the level of capacity of computation has been increasing along the years, more and more physical systems, such as biological systems, smart grids or communications have started to be regulated by means of controllers. These devices perform tasks such as computation or communication through sensors and actuators, reacting in real time to the environment. Cyber-physical systems are today ubiquitous in many different areas, such as transportation, robotics, health care, energy, military or communication.

Applications of CPSs with a high level of inherent risk associated may be called safety-critical and must, thus, be handled with special attention. These systems have as their first requirement the correct (safe) operation rather than cost or performance. Examples of safety critical applications are: a cruise controller in a car, electronic medical equipment and the autopilot in an aeroplane. Thus, an exhaustive analysis and verification of these systems is of crucial importance.

Verification can be defined as the process of determining whether a model meets the specified requirements, ensuring that the system is well-engineered. In the industry, the most common process to verify a model is by testing. Historically, experienced engineers with high knowledge about the system under test did veri-

fication by manual testing. Yet, the actual level of complexity acquired by CPSs and the increased number and detail of specifications make it hard and, in practice, impossible to manually verify these kind of systems. Most important, the continuous increment of costs motivates new testing approaches in the industry to automate testing processes.

One emerging novel technique that will be implemented in the TESTRON project, which has drawn attention in the past decade, is Model-Based Testing (MBT) [4]. This approach is growing importantly in the automotive industry due to the tendency towards the use of the Model-Based Design (MBD) methodology in the development of embedded software, above all to comply with ISO26262 "Road vehicles – Functional safety" [5]. The advantage of the MBT approach is that errors can be detected and fixed already in the early design phases, instead of in integration or marketing stages as is shown in Figure 1.1. Saving costs and facilitating faster developments are important improvements achieved with the MBT implementation [6].

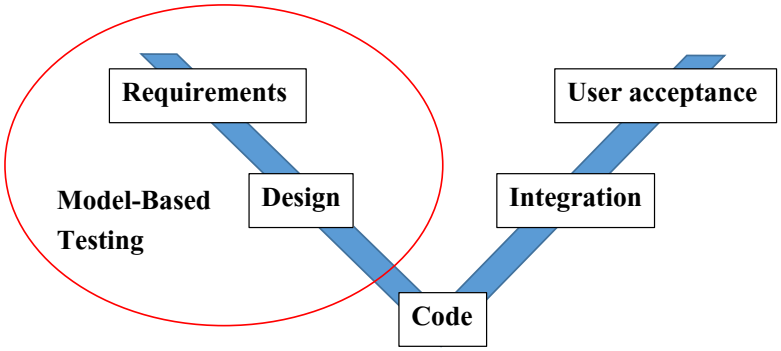


Figure 1.1 – Model-Based Design process. Model-based testing is implemented in the earlier stages of the development process. From D. Firesmith [7].

The overall structure of the MBT technique is presented In Figure 1.2. The motivation of this work lies in the last step of the algorithm where testing quality is evaluated. A test script automates a process where a test case is executed on a System Under Test (SUT) and finally a verdict is assigned to each test case. The question that arises now is how test quality can be measured to generate a verdict. A common method used in software testing is test coverage. Yet, typical coverage criteria for structural analysis at code level as Modified Condition/Decision Coverage (MC/DC) have been questioned in [8] using a simple example of CPSs. This is because analysis for verification of CPSs models is in general complex due to the interaction between continuous and discrete dynamics.

When talking about MBT, the importance of the coverage criterion is not limited to providing a metric to measure test quality. In fact, the objective of test coverage is to find areas that have not been covered by the test cases. The test criterion serves therefore as a guideline in the generation of new test cases to cover such areas. Another important property is that it may be a useful input to a test-stopping rule. When the criterion is fulfilled or it has been reached an acceptable level of coverage the testing may be stopped.

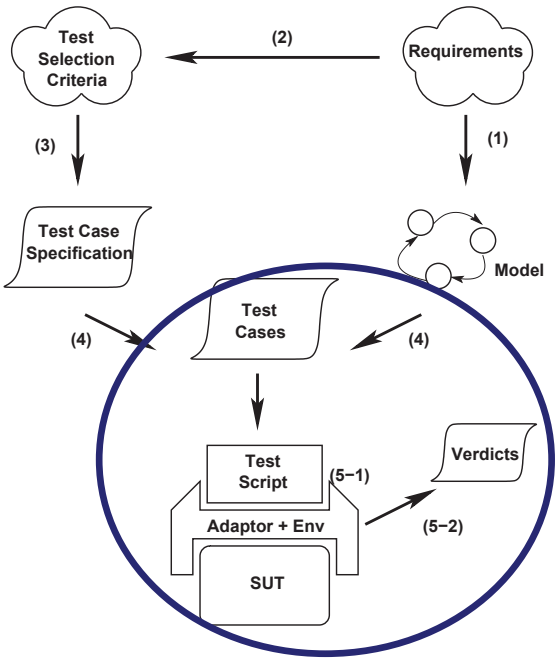


Figure 1.2 – Model-Based Testing work-flow from [4]. This master's thesis focuses in the last steps, 5-1 and 5-2. A test script is used to execute a test case on a SUT and as a result a verdict is achieved.

The aim of this master's thesis is twofold. Firstly, to define a coverage criterion suitable for CPSs which answers the question, "What property of a CPSs must be tested" Secondly, develop an algorithm to evaluate how well a test case tests the property defined by the coverage criterion and returns a verdict. For the first question, mode coverage criterion is used because it fits better with CPSs. Regarding the second question, the algorithm proposed translates cyber-physical systems into Hybrid Automata (HA) [9] [10] to be used for analysing test coverage.

1.2 Purpose

The purpose of this thesis is to create a tool that measures *test coverage* for a system with continuous dynamics. To that end, an algorithm has been developed and implemented that takes as inputs a system under test (SUT) and a test case(s) and returns the test coverage according to a pre-defined criterion.

1.3 Related Work

Hybrid systems have been extensively studied in the literature, in Alur R. et al. [9] the framework of hybrid automata (HA) is introduced to verify linear hybrid systems. Henzinger T. [11] applies model-checking techniques in HA for formal verification concluding that reachability is undecidable even for simple hybrid automata.

Regarding test coverage for hybrid systems two approaches have been reviewed. In A. Agung Julius et al. [12], the infinite set of possible testing scenarios is reduced to a finite set by means of the notion of robust neighbourhoods. The testing aims to cover all these finite scenarios. In the work presented by Tarik Nahhal and Thao Dang [13] the coverage criterion used is state coverage. The state space is divided into regions with the same characteristics and the notion of star-discrepancy is used to describe how well the visited states represent the reachable set of the system. The objective is to test the state space as much as possible without leaving big unexplored areas. This method faces the problem of the infiniteness of states in complex models, which requires a very long time of computation and have no measure of that all modes have been tested.

As a difference, the algorithm proposed in this master's thesis calculates test coverage by reducing the infinite state space to a finite set of *modes*. Such modes are used as a coverage criterion to quantify test coverage. Another difference is that while previous works use hybrid automata to represent hybrid systems, neither of them *describes* the process to generate it. The algorithm presented here establishes a method to automate the creation of a hybrid automaton to be used in the testing process.

The increasing attention and resources laid on the verification of hybrid systems are reflected in the rising number of developed tools using MBT techniques. On one hand, Reactis Tester and TestWeaver are testing and validation tools that try to maximise the degree of coverage by using guided simulation techniques [14]. On the other hand, Simulink Design Verifier uses formal methods and SAT-solving techniques to automatically generate test cases for state coverage [15]. The tool developed in this master's thesis does not use formal methods to verify a model and differs from the other approaches in that, mode coverage is used instead of

state coverage as a metric to evaluate test coverage.

1.4 Research questions

The following questions are answered in this thesis:

RQ1: *What coverage criteria for hybrid systems are useful for testing?*

RQ2: *How can modelling languages as Modelica, be transformed into mathematical descriptions suitable for analysis?*

RQ3: *What are the strength and weakness of the proposed coverage criteria?*

1.5 Method

This subsection firstly justifies the qualitative approach followed throughout this thesis with the aim of answering the research questions defined in the previous chapter. Secondly, the approach is presented.

1.5.1 Justify research approach

1. The acausal modelling approach was used as it efficiently models CPSs and is able to translate them into a mathematical representation. Among all the modelling environments, OpenModelica was chosen since it allows control over all the processes, from model description to the simulation.
2. Hybrid automata is a well studied formal model, suitable to be applied in the analysis of hybrid systems. Research and programming development was carried out to achieve a systematic way to generate hybrid automata from a mathematical description provided by OpenModelica.
3. Two different coverage criteria were studied resulting in *mode coverage* as a criterion to quantitatively measure test coverage.
4. The approach followed along the thesis allows exploiting the advantages of emerging satisfiability modulo theories solvers to efficiently reduce the number of unreachable modes in the HA.

1.5.2 Research approach

The aim of the thesis is to measure test coverage for systems with continuous dynamics. The approach followed can be summarised in Figure 1.3. A CPS system is represented by the control and plant blocks and the feedback loop. The control block describes the discrete dynamics while the plant describes the continuous

dynamics. Thus, the goal is to find the dynamical behaviours or *modes* of the plant block that are excited by a test vector.

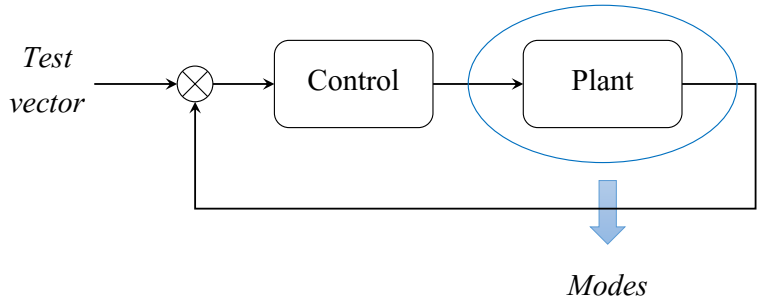


Figure 1.3 – The method followed in this thesis aims to identify modes that are triggered by a test case in a cyber-physical system. The modes describe continuous dynamics in the plant block.

The procedure followed is described below:

1. The modelling framework OpenModelica was used to transform a CPSs system described in Modelica language into a hybrid DAE. After analysing the results obtained from OpenModelica, it was decided to use a hybrid automata as the system from which to measure test coverage, also *mode coverage* was identified as a proper coverage criterion since it is particularly useful in analysing hybrid automata.
2. The hybrid DAE was used to automatically generate a hybrid automaton, which is a mathematical description suitable for test coverage analysis.
3. By using the Z3 solver, the HA was reduced removing the unfeasible modes.
4. Measure test coverage with mode coverage as a criterion.
5. Throughout the process, strength and weakness of the approach were examined.

The application developed has been implemented in Python and two software tools have been used, OpenModelica and the Satisfiability Modulo Theories (SMT) Z3 solver.

1.6 Contributions

To the best knowledge of the authors, this master's thesis proposes a novel methodology to perform test coverage based on the use of the Modelica language to represent CPSs as hybrid automata. Furthermore, it is proposed the definition of mode coverage as a coverage criterion to determine how well a system is tested.

2

Modelling Mechatronic Systems

Modelling mechatronic systems consist of capturing the dynamics of a system in a mathematical description. Among all the possible representations, two are of particular importance for this work: the Hybrid Differential Algebraic Equations (HDAE) and the hybrid automata.

This chapter is organised as follows: Mechatronic systems are first presented to later move on towards modelling concepts as modelling approaches, mathematical representations and different software for modelling and simulation mechatronic systems. Finally, the Dog-Clutch model used through the thesis is presented.

2.1 Mechatronic Systems

Mechatronic systems are a type of cyber-physical systems where the physical part is a mechanical system. An outline of a mechatronic system is presented in Figure 2.1. Typically, a micro-controller running a control algorithm is located in the electronic domain. On the other side, a robot or some mechanical process interacting with the environment is used in the mechanical domain. The communication between the two domains is conducted through actuators to perform the actions decided by the controller and through sensors providing the feedback from the mechanical part. Examples of mechatronic devices are robots and manipulator arms, more complex examples are aeroplanes. The mechatronic system investigated here is a dog-clutch that is explained in detail in section 4. *This device is used to propagate motion by coupling two rotating shafts.*

2.2 Modelling fundamentals

This section describes the two mathematical representations used throughout the modelling process, the hybrid DAE and hybrid automata. The objective is to translate the original model coded in Modelica into hybrid automata, which is a suitable structure to analyse test coverage, a HDAE is used as an intermediate step.

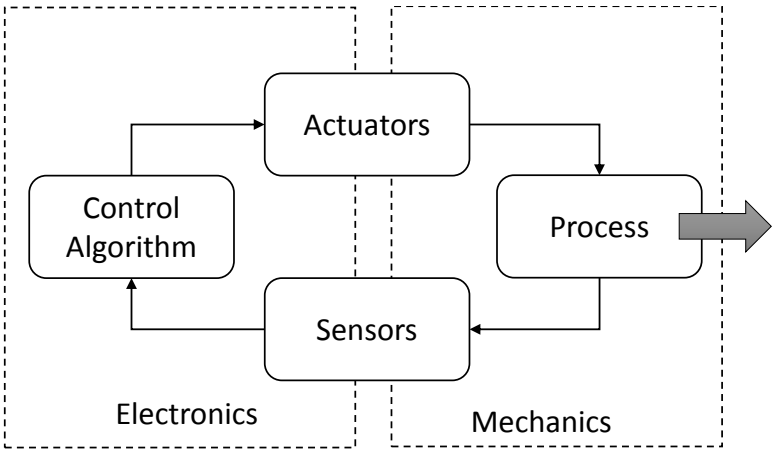


Figure 2.1 – Description of the interaction between the electrical and the mechanical part of a mechatronic system

2.2.1 Modelling approaches

A mechatronic system usually consists of two parts: a controller and a plant. Modelling the controller usually includes sensors and actuators and may experience jitter, delay times and other disturbances. Since control algorithms are designed using a model of the plant, it is thus necessary to model the mechanical system as close as possible to the reality. The implementation of the plant is a difficult task due to the highly complex interaction between the different domains. Two prominent approaches are commonly used to carry out modelling tasks, causal and acausal modelling, both are described below in more details.

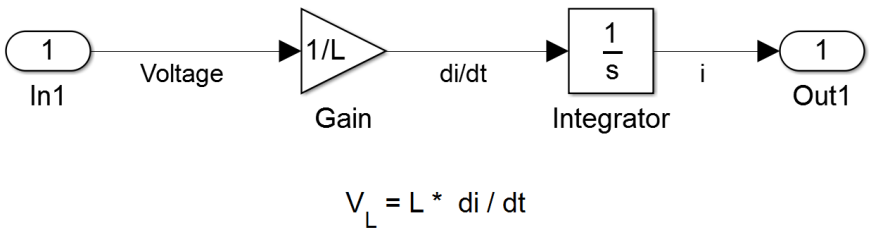
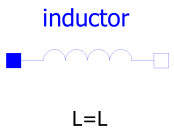


Figure 2.2 – Simulink model of an inductor where the input is the voltage and the output the current

1. **Causal modelling**: Causal models follow the principle of *causality*. Blocks have defined inputs and outputs that impose a *causal* relationship between variables. The interconnection between different blocks is done by links,

which transmit signals at each time step. The model is constructed following the dependencies between variables, generating a determined data flow direction for the signals. Systems that present such behaviour are called causal systems, an example of this is created with Simulink and shown in Figure 2.2, where a mathematical description of an inductor is used to obtain current from a voltage input. One disadvantage of this approach is that the model diagram follows a hierarchy and represents the mathematical relationships rather than the physical representation of the model. Another drawback is the re-usability of the block, in this case, it can only be used to compute current when the voltage and resistnce are given.

2. **Acausal modelling**: This approach also called the Physical Network Approach aims to model physical systems through a mathematical description made up by a set of Differential Algebraic Equations (DAE). Unlike the causal approach, equations neither specify input nor output variables. In this context, equations must be understood as relations among variables, not as assignments, therefore causality is not specified. The directionality or data flow of the variables among blocks is defined during simulation time when the equations are solved. Graphically the acausal approach provides a model with a topology very similar to the real physical model. Figure 2.3 describes a simple example where an ideal inductor is created. The icon represents an inductor that is governed by the relation $L \cdot \frac{di}{dt} = V$, where voltage and current flows are simultaneously simulated.




```
model Inductor "Ideal linear electrical inductor"
extends Interfaces.OnePort(i(start=0));
parameter SI.Inductance L(start=1) "Inductance";
equation
L*der(i) = v;
end Inductor;
```

Figure 2.3 – A script that models an Inductor using Modelica is shown within the frame. Above the script, the graphical representation of the class *Inductor* is depicted.

The second approach, acausal modelling, fits better with the aim of this thesis in the sense that it is possible to extract the equations describing the behaviours

of the model to analyse it. That is in opposition to the causal approach, where only a bunch of C-code files is obtained making it complicated, if not impossible, to analyse behaviours of mechatronic systems. In fact, the equations provided by acausal modelling are expressed in a structure called hybrid DAE that is described more exhaustively in the next section.

2.2.2 Hybrid DAE

A hybrid DAE can be summarised as a mathematical model that switches its behaviour with time and is composed of a set of DAEs, representing continuous dynamics. Below are briefly introduced, from a less to a more complex level, the different systems of equations that define a hybrid DAE. An example of how a HDAE looks is presented in Figure 2.4.

Algebraic equations describe the design and physical constraints among variables. It does not contain derivatives of the system variables.

Ordinary differential equations are equations with derivatives that models rates of change of the system variables and are used to describe natural phenomena.

Differential algebraic equations are a generalization of ODE's including algebraical equations. A DAE can be expressed as $F(\dot{x}, x, t) = 0$.

$$\left.\begin{aligned} \dot{x}_1 &= 2x_1 + u_1 \\ \dot{x}_2 &= 4x_2 + 6u_2 \end{aligned}\right\} \quad \dot{x} = f(x, u) \quad \text{Differential equations}$$

$$x_1 = 3x_2 \qquad h(x) = 0 \qquad \text{Algebraical equation}$$

Figure 2.4 – Example of a DAE composed by two differential equations and one algebraical equation.

Hybrid DAEs naturally arise during the modelling process of physical systems. This mathematical description is the tool used to construct hybrid automata from which test coverage will be analysed.

2.2.2.1 Hybrid DAE in SimCode

As far, the concept of hybrid DAE has been introduced without any mention with regard to its implementation in a computational language. An efficient way to represent HDAEs is through *conditional equations* also called if-equations, an example of two conditional equations is shown in (2.1). In this simple example, the value of the variables a and b defines the expression at which variables $\dot{x}$ and

x are updated. The two conditional equations describes four different DAEs that are shown in Table 2.1.

$$\dot{x}_1 = \begin{cases} 2x_1 + u, & \text{if } a < 0 \\ -(x_1 + u), & \text{if } a \geq 0 \end{cases} \tag{2.1a}$$

$$x_2 = \begin{cases} 1, & \text{if } b < 0 \\ 0, & \text{if } b \geq 0 \end{cases} \tag{2.1b}$$

$\boldsymbol{a, b < 0}$	$\dot{x_1} = 2x_1 + u$ $x_2 = 1$	$\dot{x_1} = 2x_1 + u$ $x_2 = 0$	$\boldsymbol{a < 0; b \geq 0}$
$\boldsymbol{a \geq 0; b < 0}$	$\dot{x_1} = -(x_1 + u)$ $x_2 = 1$	$\dot{x_1} = -(x_1 + u)$ $x_2 = 0$	$\boldsymbol{a, b > 0}$

Table 2.1 – DAEs generated by the conditional equations in (2.1).

The modelling software OpenModelica uses the hybrid DAE mathematical description to collect all the information related to the dynamics of the model in a data structure called SimCode. A simplified version of the SimCode for the case study is shown in Table 2.2[1]. The different DAEs that conforms the hybrid DAE are described by regular equations and the conditional equations (2.2a), (2.2d), (2.2e) and (2.2f). The variables defined by if-equations, update its value according to the guards G_i that are evaluated to true or false each time instant.

2.2.3 Hybrid Automata

A hybrid automaton is a high-level formal model that can be described as an extended finite automata (EFA) with continuous state variables. EFAs are usually modelled using *modes*, edges and guards. In this thesis a mode represent a discrete region or location in a graph that determines a specific continuous dynamic of the plant. Edges and guards are used to represent the transitions between modes.

A HA can evolve in two different ways, by discrete and by continuous evolution. In the first case, a transition is satisfied bringing the system to a new mode and imposing a different behaviour. In the second case, continuous state variables

[1]For the sake of clarity some equations have been reduced

$$engage\_req = \texttt{if } G_1 \texttt{ then 1 else 0} \tag{2.2a}$$
$$f\_spring\_current = \texttt{KI * current + (-k\_spring) * z - DampingBelleville * zv} \tag{2.2b}$$
$$a = \texttt{0.02777777777777778 * sin(36.0 * phi)} \tag{2.2c}$$
$$\$DER.zv = \texttt{if } G_2 \texttt{ then if } G_3 \texttt{ then DIVISION(f\_spring\_n\_current, mass) else DIVISION(f\_spring\_n\_current - stiff2 * (-0.0016 + z), mass) else 1} \tag{2.2d}$$
$$t = \texttt{if } G_4 \texttt{ then if } G_5 \texttt{ then 2 else if } G_6 \texttt{ then HiSpeed2 * w else 1.5 else 0.0} \tag{2.2e}$$
$$eng\_state = \texttt{if } G_7 \texttt{ then if } G_8 \texttt{ then 0.0 else if } C_9 \texttt{ then 1 else 0.0 else 0.0} \tag{2.2f}$$
$$\$DER.z = \texttt{zv} \tag{2.2g}$$
$$\$DER.phi = \texttt{w} \tag{2.2h}$$
$$prev\_eng\_state = \texttt{pre(eng\_state)} \tag{2.2i}$$

Table 2.2 – Set of equations and conditional equations that included in a hybrid DAE in OpenModelica. The guards G_i are evaluated to true or false at each time instant.

evolve according to differential algebraic equations. For mechatronic systems the controller determines the locations of the EFA and the mechanical system determines the continuous dynamics. A more formal definition of Hybrid automaton is presented below:

Definition 2.1. *Hybrid automaton [9] [13]:*
A Hybrid automaton is defined by a five-tuple as:

$$\langle \mathcal{X}, \mathcal{Q}, \mathcal{F}, \mathcal{E}, \mathcal{G} \rangle \tag{2.3}$$

- $\mathcal{X} \subseteq R^n$ is the continuous state space of the system in which continuous state variables are defined. Here $\mathcal{X} \subseteq R^n$
- $\mathcal{Q} = \langle q_1, q_2, ..., q_n \rangle$, $n \in N$ is the *finite* set of ***modes*** or regions. Discrete transitions switch between different behaviours. Associated with each mode is a DAE describing the behaviour (dynamics) of the system.
- $\mathcal{F} : \mathcal{X} \times \mathcal{Q} \to \mathcal{X}$ represents the vector field that defines the continuous dynamics in each mode. In this work continuous dynamics are described by DAEs.
- $\mathcal{E} \subseteq Q \times Q$ is a finite set of edges representing discrete transitions. A transition $e \in E$ is defined by a three tuple (m, m', g) where m is the origin mode, m' is the target mode and $g \in \mathcal{G}$ is the guard of the transition. Transitions determines the discrete dynamics.
- $\mathcal{G} : G_e | e \in \mathcal{E}$ is the set of guards enabling transitions among different modes. When a guard is evaluated to *true*, a new mode becomes active imposing a new behaviour. In this report, guards are represented by predicates.

Definition 2.2. *Hybrid State*

$(q, x) \in \mathcal{Q} \times \mathcal{X}$ is a specific hybrid state of the system which includes discrete and continuous variables.

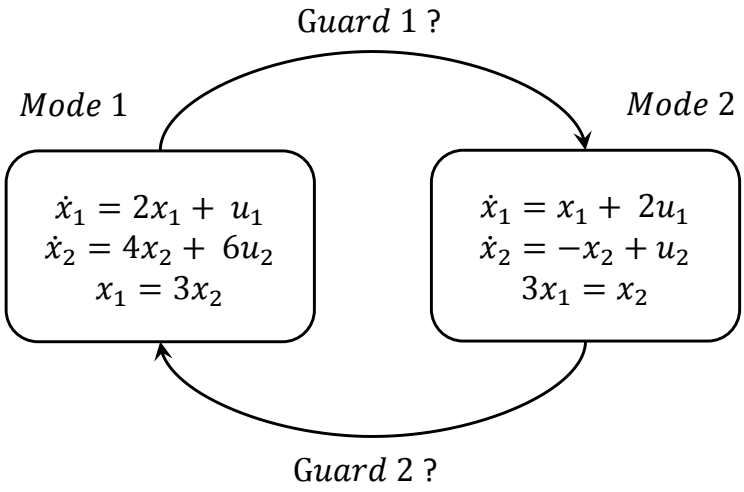


Figure 2.5 – Hybrid Automaton with two modes. Transitions between ***modes*** are enabled when the corresponding guard is evaluated to True. At each mode a DAE defines the dynamics of the system.

2.3 Modelling Software

To carry out the modelling task there are plenty of commercial software available. Below, the modelling environments considered during this project are briefly presented.

- Causal Approach:
  Simulink®: is a block diagram environment for multidomain simulation and Model-Based Design [16].
- Acausal approach:
  Simscape™: is a tool for modelling and simulating multidomain physical systems. [17]. This is the tool used at Volvo Car Corporation to implement the dog-clutch model, this model is discussed later in this report.
  OpenModelica: is an open-source Modelica-based modelling and simulation environment intended for industrial usage [18]. In this thesis OpenModelica was selected to carry out the simulation process.

The decision towards using OpenModelica as the modelling environment was motivated by two qualitative criteria. Mainly because it uses an acausal modelling

approach so that dynamics are described by equations. The other important reason is that OpenModelica provides an open source environment where it is possible to rewrite the internal code and customise all the internal steps involved in the simulation process. This property was really appreciated in the beginning of this work because it gave great versatility to decide among different strategies in order to answer the proposed research questions. For instance, one of the first issues was about to choose between the generated C-code files or the *optimised sorted equations* to identify the different modes or behaviours of the system.

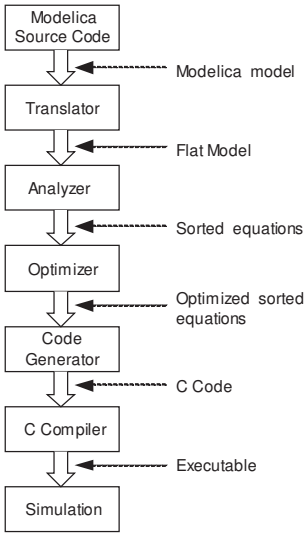


Figure 2.6 – OpenModelica Compiler Translation stages from Modelica code to executable C-code. Figure taken from P. Fritzson et al. [19].

The workflow of OpenModelica is summarised in Figure 2.6. OpenModelica takes as input a model described in the Modelica language and after some internal steps, the model is translated to a structure with optimised equations. This structure is internally called SimCode and is the object used to generate the C-code for simulation. The importance of this structure lies in the fact that the set of equations describe a hybrid DAE which can be dumped into a file or read from the C-file.

3

Test Coverage for CPSs

Two different approaches to perform test coverage are presented in this section. The first is structural analysis of the code and the second is structural analysis of the modes.

After evaluation of the two different options, the second approach was chosen. Finally, the general algorithm is presented to check test coverage using a model and a test case.

3.1 Coverability criteria

As previously mentioned in the introduction, verification is used to assure the correctness and can be done in a variety of ways. However, testing is the most common process to verify a model. A common method used by software engineers to evaluate the quality of testing is *test coverage* analysis.

Test coverage is a technique used within software testing to quantify the degree to which a model is covered during the testing process. Using this technique requires, above all, to decide what property or characteristic of the system should be measured. An adequate metric or *coverage criterion* has to be defined indicating what properties must be exercised to perform a "good" test. Based upon a chosen metric, it is possible to quantify how well the SUT is explored under simulation. For instance, *statement coverage* is a typical coverage criterion in the industry, which describe whether a statement of a code have been executed at least once or not at all. The degree of compliance reached during testing can be measured quantitatively by the percentage of statements executed.

According to [20], many development standards claim *structural coverage* criteria at code level as the best practice in industrial software development. Such types of criteria aim to analyse the structure of the program. Some examples of those criteria are: function coverage, statement coverage, branch coverage and condition coverage [21]. Unfortunately, these criteria do not fit well with CPSs. CPSs are non-deterministic reactive systems i.e. they constantly interact with its environment in a manner that one input to the system can lead to different outputs depending on the state of the environment. This is in opposition to the

classical model of computation where for one input a specific output is expected. In [8], it has been shown that the coverage criterion Modified Condition/Decision Coverage (MC/DC) is not able to exercise all the modes of an elementary case of CPSs even if the MC/DC coverage is fulfilled. Thus, classical software test criteria are inadequate for verification of CPSs.

Other coverage metrics as state coverage have been used [13] to measure test coverage for CPSs. Though, CPSs have continuous states implying that the state space is infinite so that in practice it cannot be entirely explored. Other appropriate coverage criteria should be used.

3.1.1 The proposed criterion: mode coverage

The objective of this master's thesis is to cover/excite all the different finite behaviours of the system. Based on this idea, two coverability approaches have been identified and examined. The first approach studied was structural analysis at code level that involves the GNU Coverage (GCOV) tool. The second approach is intended for structural analysis of modes[1] and requires the use of hybrid automata and Satisfiability Modulo Theories (SMT).

After having examined the two methods, the proposed criterion was *mode coverage* and is defined as follows:

$$\text{Mode Coverage} = \frac{\text{Number of modes exercised during simulation}}{\text{set of reachable modes}} \times 100\% \quad (3.1)$$

In the next sections both approaches to do test coverage are described thoroughly.

3.2 Test coverage analysis: GCOV

This part presents the approach followed towards accomplishing test coverage at code level. To begin with, the GCOV tool is introduced and later the methodology used to analyse C-code for test coverage is explained. To conclude, the disadvantages that led to discarding this approach are explained.

3.2.1 Introduction to GCOV

GCOV [22] is a software tool for coverage analysis of code generated by the GCC compiler (GNU Compiler Collection). This is the compiler used by OpenModelica to generate the C-code for simulation. For the goal of this thesis, the interesting

[1]The definition of mode is in 2.1

functionality of this tool is its ability to count lines of code that are executed during simulation.

3.2.2 Methodology Workflow

The workflow of the method followed is summarized in Figure 3.1. Since it was decided to use OpenModelica, a manual translation from Simscape code to Modelica code was done. Inside OpenModelica a SimCode structure, that contains all the information needed for C-code generation, was generated. SimCode is later passed as input to a process where C-code files are returned. The last step lies in executing the GCOV tool upon the C-code to obtain structural coverage analysis at code level.

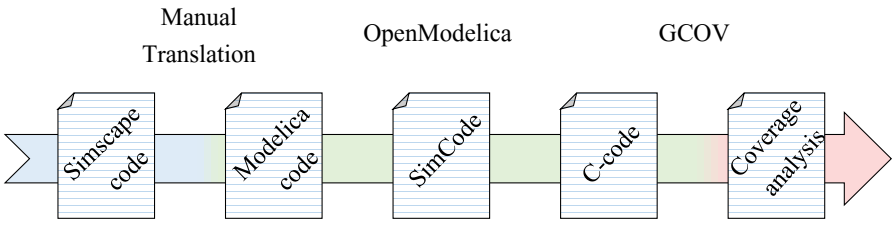


Figure 3.1 – Work-flow followed to perform structural coverage analysis at code level employing GCOV.

The attractive point of this methodology is that dynamics described by equations in the original model are captured by the C-code. Another advantage is that OpenModelica tracks all the processes of equation transformations in such a way that C-code can be easily interpreted and hence, analysed.

3.2.3 Analysis of C-code

The aim of the analysis is to identify, in the C-code, all behaviours of the system triggered during simulation. It is useful to recall that behaviours are described by DAEs which, in turn, are made up of set of equations, as it is shown in section 2.2.2. The algorithm to perform test coverage is summarised in three steps below:

1) to identify lines where the equations that defines dynamics are located.
2) to check whether those lines have been triggered or not.
3) calculate the percentage of equation triggered against the total number of equations

The C-code generated[2] from the conditional equation (2.1a) is used in Figure 3.2 to exemplify how the analysis has been carried out. The highlighted lines 13

[2]For the sake of clarity the code has been slightly modified

and 17 assign one of the two dynamics to the variable $tmp2$, which value is later updated to the variable $\dot{x}$ in line 19. Identification of lines that assign dynamics was done using the libClang [23] package inside the programming environment Eclipse. LibClang parses the source code generating an abstract syntax tree that can be traversed later to find specific parts of the code. GCOV was invoked to identify whether lines were executed during simulation or not. When GCOV completed the analysis, it was returned a file containing, among other information, the times a line was executed. In Figure 3.2 can be seen that both lines 13 and 17 were executed at least one time. In particular, 8808 and 6677 times respectively. In this case, test coverage would be 100% as both behaviours of the variable $\dot{x}$ were exercised.

Several drawbacks were found during the implementation that makes difficult to follow this approach. The most important are listed below:

- This method allows to account how many times the different equations that defines the dynamics are triggered during simulation but is not capable of discerning which of them are triggered concurrently. That is important because the combination of the dynamics is what defines the behaviour or mode at any given moment. As a result it is not possible to identify modes with the GCOV solution proposed.
- The method to detect specific lines in the C-code lacks generality. Actually, parsing C-files could be a complex task if the objective is to automate the process to all possible models.
- It is necessary to modify the OpenModelica compiler to generate C-code files with a suitable format for GCOV. Some necessary modifications are: to remove the use of the C ternary operator "?", to handle different types of linear and non-linear functions to make them explicit, modify the makefile that compiles and links the C-code, etc. Although these adjustments can be done, it could be a source of frequently occurred problems.

Before this method was implemented, and due to the aforementioned disadvantages, it was decided to move on with another approach that is discussed in the next section.

```
1  /*
2  equation index: 15
3  type: SIMPLE_ASSIGN
4  der(x) = if a < 0 then 2x + u else -(x + u)
5  */
6  void clutch_eqFunction_15(DATA *data, threadData_t *threadData)
7  {
8  const int equationIndexes[2] = {1,15};
9  RELATIONHYSTERESIS(tmp0, $Pa, 0, 0, Less);
10 tmp1 = (modelica_boolean)tmp0;
11 if(tmp1)
12 {
13 tmp2 = (2) * (x + u);
14 }
15 else
16 {
17 tmp2 = (-1) * (x + u);
18 }
19 DER(x) = tmp2;
20 }
```

Figure 3.2 – Script showing the C-code generated from the conditional equation in (2.1a). Highlighted lines show the conditional equation in Modelica language and the sentences where the script assigns dynamics to the variable $\dot{x}$

```
1      -:    1: /*
2      -:    2: equation index: 15
3      -:    3: type: SIMPLE_ASSIGN
4      -:    4: der(x) = if a < 0 then 2x + u else -(x + u)
5      -:    5: */
6  15485:    6:void clutch_eqFunction_15(DATA *data, threadData_t *threadData)
7      -:    7:{
8      -:    8:  const int equationIndexes[2] = {1,15};
9  15485:    9:  RELATIONHYSTERESIS(tmp0, $Pa, 0, 0, Less);
10 15485:   10:  tmp1 = (modelica_boolean)tmp0;
11 15485:   11:  if(tmp1)
12     -:   12:  {
13  8808:   13:  tmp2 = (2) * (x + u);
14     -:   14:  }
15     -:   15:  else
16     -:   16:  {
17  6677:   17:  tmp2 = (-1) * (x + u);
18     -:   18:  }
19 15485:   19:  DER(x) = tmp2;
20 15485:   20:}
```

Figure 3.3 – Example of the file generated by GCOV where the script in Figure 3.2 is the input. It shows how often each line is executed during simulation. Highlighted lines show the conditional equation in Modelica language and the sentences where the script assigns dynamics to the variable $\dot{x}$

3.3 Test coverage analysis: Hybrid Automaton

A hybrid automaton is a formal model that may be used to analyse CPSs because of its ability to reduce the infinite state space to a finite set of locations called modes, this property provides a new coverage criterion called *mode coverage*. In the course of this section is described the approach followed to perform test coverage as proposed in (3.1). Initially, the general algorithm to quantify mode coverage is presented, then, two steps of the algorithm are described in more detail: the abstraction from a hybrid DAE to hybrid automata and the operation of the Z3 Solver to reduce the number of modes in a HA.

3.3.1 General algorithm

The algorithm developed during this master's thesis to automate the process of quantifying test coverage for CPSs is hereby presented, see Figure 3.4. The approach taken is based on the translation of a hybrid DAE to a formal model described as a hybrid automaton. Then, test coverage is carried out aiming to exercise all the modes of the system, the different steps are described below.

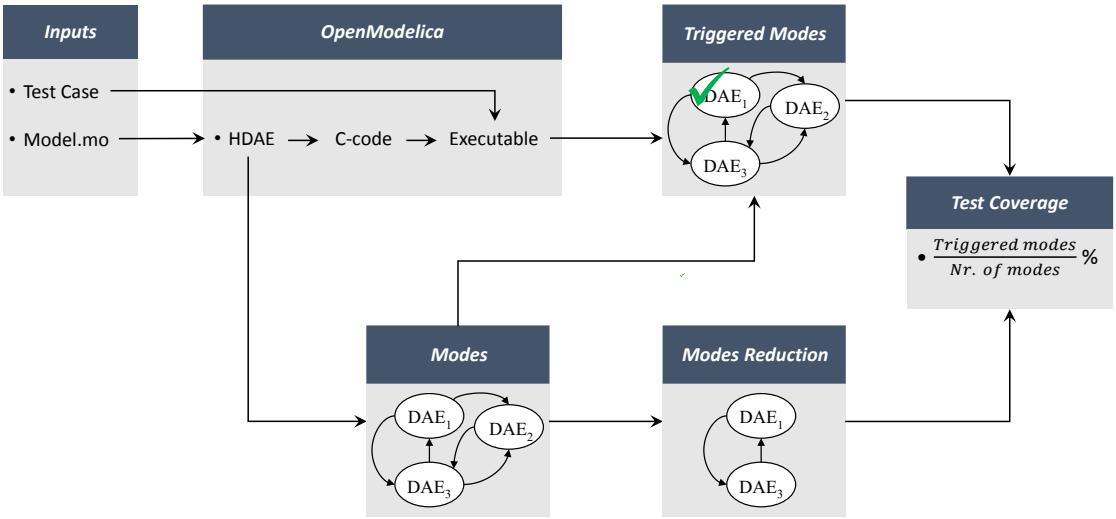


Figure 3.4 – Overview of the approach followed to perform test coverage starting from a test case and a model of a CPSs described in Modelica language.

- **Inputs:** The algorithm begins with a model to be tested and a test case to excite behaviours of the model.

- **OpenModelica:** The environment OpenModelica takes the model.mo and generates a hybrid DAE that is saved in a data structure named SimCode from which modes can be extracted to generate a hybrid automaton. The second input, the test case, and an executable file are used in the last step to simulate the original model. The results of the simulation are saved in a file that is later on used to identify the triggered modes during simulation. The most importance of using OpenModelica resides in the fact that it was transformed a Cyber-Physical system into a Hybrid DAE.
- **Modes:** A hybrid automaton is created from a hybrid DAE. At this step, all feasible and infeasible modes are identified. This process is explained in detail in section 3.3.2. To the extent that this process is automated it supposes an advantage with respect to other approaches as [12] and [13], since they started their approaches with a hybrid model without showing how it is achieved.
- **Mode Reduction:** At this point infeasible modes of the hybrid automaton are removed by means of the Z3 solver, in addition, modes describing the same dynamics are removed. The remaining modes are saved and sent to the last step where test coverage is measured. A more in-depth explanation is presented in section 3.3.3.
- **Triggered modes:** The workflow to retrieve the triggered modes during simulation time is shown in Figure 3.5. The input to the algorithm are the conditions of the hybrid DAE and the results of the simulation. The algorithm loops through the simulation time and all the conditions are evaluated to check whether they are true or false. As mentioned in section 3.3.2.1, solely one condition in each conditional equation will be evaluated to true. Thus, a mode is identified as the logical conjunction of the conditions evaluated to true at each time instant. Following the example of the hybrid DAE in (3.3), at any given time the evaluation of the conditions will give as a solution only one of the modes listed in Table 3.1. When the *for* loop is completed, the repeated modes are filtered. Likewise, modes with different conditions but representing the same dynamics are also eliminated such that the final output is the minimal set of triggered modes.
- **Test coverage:** At this point, it is now possible to measure test coverage produced by executing a test case on the SUT applying (3.1) which is shown here again for clarity:

$$\text{Mode Coverage} = \frac{\text{Number of modes exercised during simulation}}{\text{set of reachable modes}} \times 100\% \tag{3.2}$$

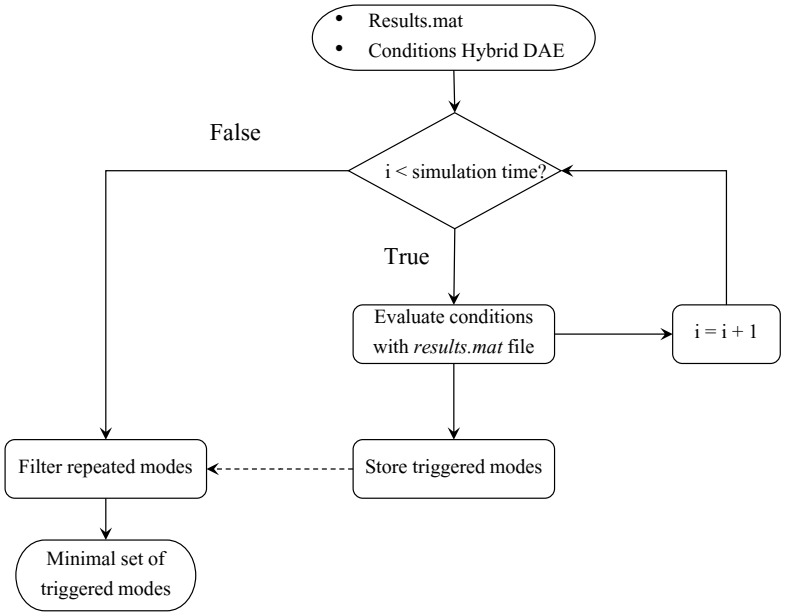


Figure 3.5 – Description of the process *Triggered modes*. The algorithm aims to find the modes triggered during simulation time, starting with the results of the simulation and the conditions retrieved from a hybrid DAE.

3.3.2 Modes: from Hybrid DAE to Hybrid Automaton

The procedure developed for constructing hybrid automata relies on the abstraction of hybrid DAEs. To illustrate the steps carried out, a simple example of a hybrid DAE shown in (3.3) is used throughout this section. The example consists of two conditional equations that can take two and three different values (f_i^j) depending on which of the conditions is true. The conditions, represented as C_i^j, are predicate sentences, as for instance "$a \geq 0$", where a can be of type real, integer or boolean.

$$var_1 = \begin{cases} \text{if} & C_1^1 \text{ then } f_1^1 \\ \text{elseif} & C_2^1 \text{ then } f_2^1 \end{cases} \tag{3.3a}$$

$$var_2 = \begin{cases} \text{if} & C_1^2 \text{ then } f_1^2 \\ \text{elseif} & C_2^2 \text{ then } f_2^2 \\ \text{elseif} & C_3^2 \text{ then } f_3^2 \end{cases} \tag{3.3b}$$

The following simplifications have been made:

- In Modelica, conditional equations must have an *else* statement at the end to cover the case when the rest of conditions are evaluated to false. For clarity purposes, this statement is obviated without loss of generality in the explanation to generate a HA.
- Hybrid DAEs can also contain regular equations that are not conditional, though, that sort of equations always exist in all modes so that they can be omitted in the analysis without missing modes.

For this simple case, the hybrid DAE in (3.3) results in six modes that are listed in Table 3.1. The hybrid automaton generated is depicted in Figure 3.6 where the automaton consists of 6 modes and 30 transitions.

Mode	var_1	var_2	Guard
1	f_1^1	f_1^2	$C_1^1 \wedge C_1^2$
2	f_1^1	f_2^2	$C_1^1 \wedge C_2^2$
3	f_1^1	f_3^2	$C_1^1 \wedge C_3^2$
4	f_2^1	f_1^2	$C_2^1 \wedge C_1^2$
5	f_2^1	f_2^2	$C_2^1 \wedge C_2^2$
6	f_2^1	f_3^2	$C_2^1 \wedge C_3^2$

Table 3.1 – Information that can be extracted from 3.3 to define modes. Each mode has associated a guard as a combination of conditions. Each guard defines the value of the variables var_1 and var_2 that represents the dynamics of the mode.

3.3.2.1 Identification of modes

Hereinafter, the followed procedure identifies modes by its unequivocally associated guard, which is a combination of conditions that must be fulfilled to reach a specific mode.

It is easy to see that the maximum number of possible modes can be calculated by multiplying the number of conditions inside each conditional equation with

each other. In the case of 3.3, the number of modes is $2 \times 3 = 6$. In order to automate the process of mode generation, the set of possible modes is modelled as the cross-product over all the conditions in a hybrid DAE and is represented by the logical formula (3.4). This logical formula is said to be in Conjunctive Normal Form (CNF) and to expand it means to end up doing the cross-product of the elements C_j^i. The usefulness of representing modes through a CNF formula will become more clear when explaining the Z3 solver [24] in section 3.3.3.1.

$$Modes = \bigwedge_{i=1}^{k} \left(\bigvee_{j=1}^{n_i} C_j^i \right) \tag{3.4}$$

where:

$$k \in \mathbb{N} = \text{number of conditional equations}$$
$$n_i \in \mathbb{N} = \text{number of conditions at each conditional equation}$$

As an example, the result of applying the above formula on the hybrid DAE (3.3) is presented in the below equation (3.5) which spans all the possible modes. Obviously, the modes finally calculated coincide with the modes in table 3.1.

$$\begin{aligned} \texttt{Modes} &= \left(C_1^1 \vee C_2^1\right) \wedge \left(C_1^2 \vee C_2^2 \vee C_3^2\right) = \\ &= \left(C_1^1 \wedge C_1^2\right) \vee \left(C_1^1 \wedge C_2^2\right) \vee \left(C_1^1 \wedge C_3^2\right) \vee \\ &\vee \left(C_2^1 \wedge C_1^2\right) \vee \left(C_2^1 \wedge C_2^2\right) \vee \left(C_2^1 \wedge C_3^2\right) \end{aligned} \tag{3.5}$$

A more deep insight into how modes are calculated suggest the following shortcomings to be borne in mind when building the hybrid automata:

1) Hybrid automata are able to capture non-deterministic behaviour, i.e., there may also be situations where two or more conditions in a conditional equation are true at the same time. For instance, staying in mode 1, it could happen that conditions C_1^2 and C_2^2 being true, entailing the possibility that modes 1 and 2 would be active simultaneously, which is physically impossible. In order to achieve deterministic behaviour, the modes should be described in a manner so that two or more conditions can never be true at any given time. In the tool developed along with this work, the procedure followed is illustrated with the guard associated with the mode 1:

$$\text{Mode } 1 : \left\langle \left(C_1^1 \wedge \neg C_2^1\right) \wedge \left(C_1^2 \wedge \neg \left(C_2^2 \vee C_3^2\right)\right) \right\rangle$$

The above mode can solely be reached when C_1^1 and C_1^2 are true and C_2^1, C_2^2 and C_3^2 false.

2) Not all guards are feasible; it can occur that some conditions in a guard contradict each other. For instance, the mode 1 with guard $\langle C_1^1 \wedge C_1^2 \rangle$ where $C_1^1 = current < 0$ and $C_1^2 = current \geq 0$ will be always false, meaning that this mode does not exist. The SMT Z3 tool is used in order to identify such contradictions and remove the infeasible modes generated by (3.4). This process significantly reduces the number of modes in the system.

3) Several conditions C_i^j in (3.3) may update a variable to the same value so that different guards would define the same mode. Looking at table 3.1, it could happen that $f_1^2 = f_3^2$ so that mode 1 = mode 3 and mode 4 = mode 6, thus, the created automaton has four modes instead of 6. This possibility is exploited to further decrease the number of modes in the HA.

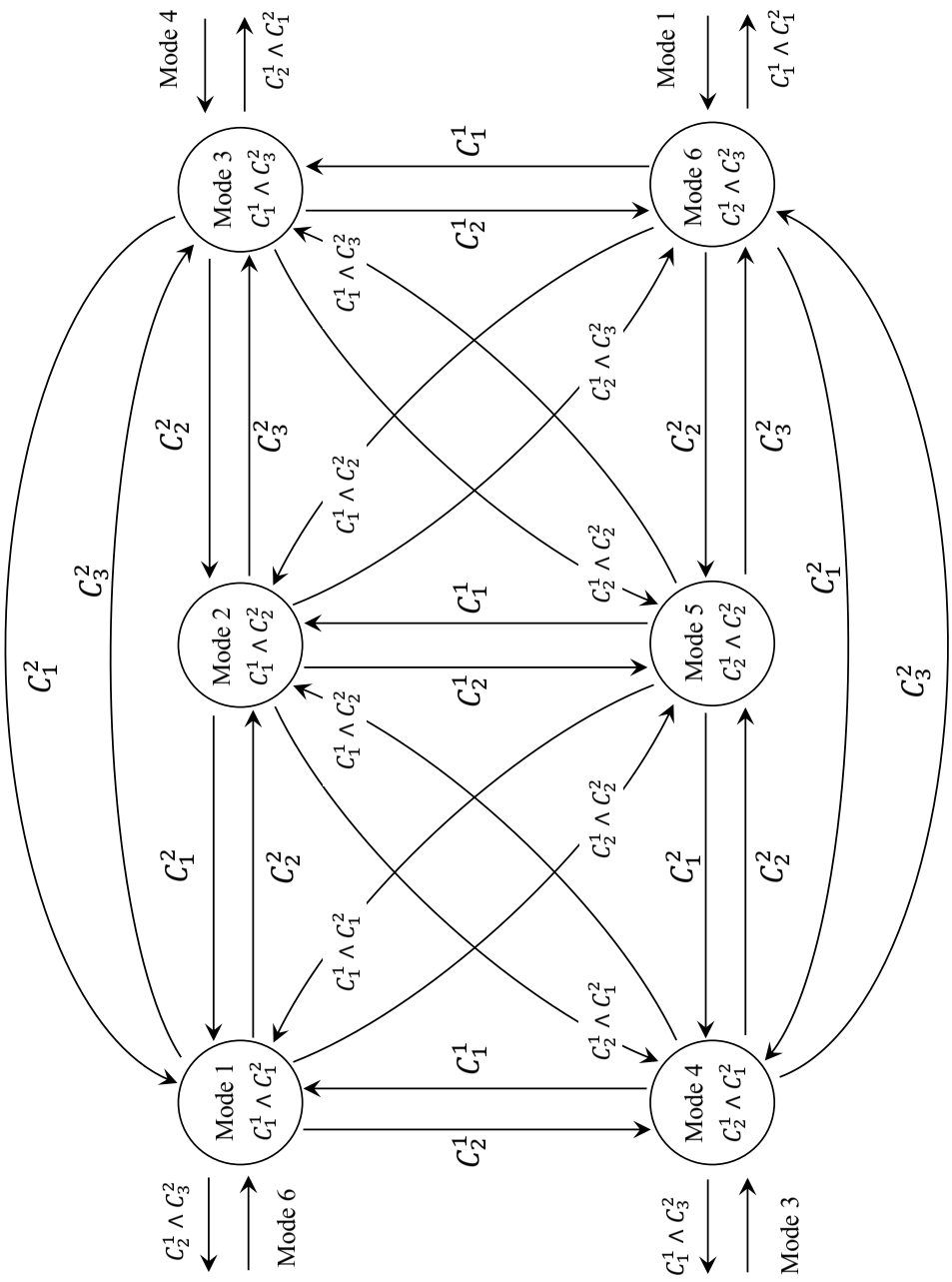


Figure 3.6 – Hybrid automaton abstracted from the hybrid DAE in 3.3.

3.3.3 Mode Reduction

The mode reduction step is based on the use of the Satisfiability Modulo Theories, therefore, first is introduced the Z3 solver and later is presented the workflow to reduce the number of modes.

3.3.3.1 Satisfiability Modulo Theories - the Z3 Solver

The Z3 solver is a *Satisfiability Modulo Theories* (SMT) solver developed by L. de Moura and N. Bjørner at Microsoft Research [24]. An SMT solver is such that given a logical formula in first-order logic, it decides whether it is *satisfiable* or not. A formula $\mathcal{F}$ is said to be satisfiable when exists a valuation that makes $\mathcal{F}$ true. In the SMT context, the input to the solver represents a constraint system that the solver will try to satisfy providing a solution.

In order to illustrate the operation of the solver is presented an example in Figure 3.7, that uses two different formulas. The first case is decided to be satisfiable and a solution is provided while for the second case no solution is returned since the formula is unsatisfiable.

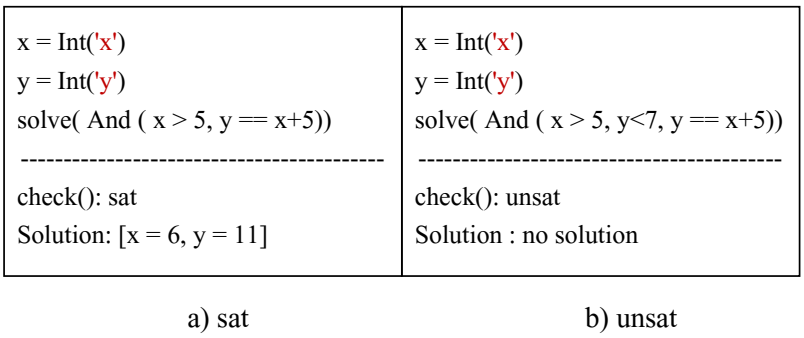


Figure 3.7 – Example of how the Z3 solver checks satisfiability. First are defined the variables x and y, then a constraint is added to the solver. With the *check* command the solver decides whether the formula has a solution or not, if yes a solution is displayed.

The logical formulas used in the fore-mentioned example are the same sort of formula as was used in (3.5) to identify all modes: both are logical formulas that uses predicates. Obviously, this was not a coincidence but a deliberate choice in favor of using the Z3 solver to analyse modes. Primarily, the objective is to check satisfiability for all modes in order to remove those that are false. Among different SMT solvers, Z3 was selected because of the following reasons:

- It is free available for academic purposes. Licensed with Microsoft Research License Agreement (MSR-LA).
- It has interfaces API's in several languages, in particular, in Python which is the language used to develop test coverage in this thesis.
- If a formula is satisfiable, SMT Z3 returns a solution which is an useful property used during the development of the algorithm to perform test coverage.
- It is a high-performance theorem prover. In the SMT Competition at 2014 SMT Z3 participated as a reference for the other competitors.

For the time being, the Z3 solver supports nonlinear polynomial Real arithmetic but cannot handle transcendental functions such as sine, cosine and exponential. [25].

3.3.3.2 Mode Reduction Workflow

The number of modes increases as the level of complexity of the DAEs raise. As commented in the earlier section 3.3.2.1, the combination of conditions easily leads to infeasible modes that can be removed applying the Z3 tool. More accurately, the Z3 solver selects the feasible modes among all possible modes. Figure 3.8 shows the algorithm to perform reduction of modes, which is divided into two main parts: firstly, all possible modes are sent to the Z3 solver as an input, through (3.4). The core of the method is a *while* loop checking whether the set of added constraints is satisfiable or unsatisfiable. If satisfiable, Z3 returns a solution that is mathematically coherent and represents a feasible mode in the model. Later, the mode is negated and forwarded to the solver as a new constraint, such that at the next time the solver is checking satisfiability it will not return the same solution. When the model is decided to be unsatisfiable the search of feasible modes is completed. In the second part of the algorithm, further reduction of modes is achieved by removing those describing same dynamics to obtain the minimal set of modes in the system.

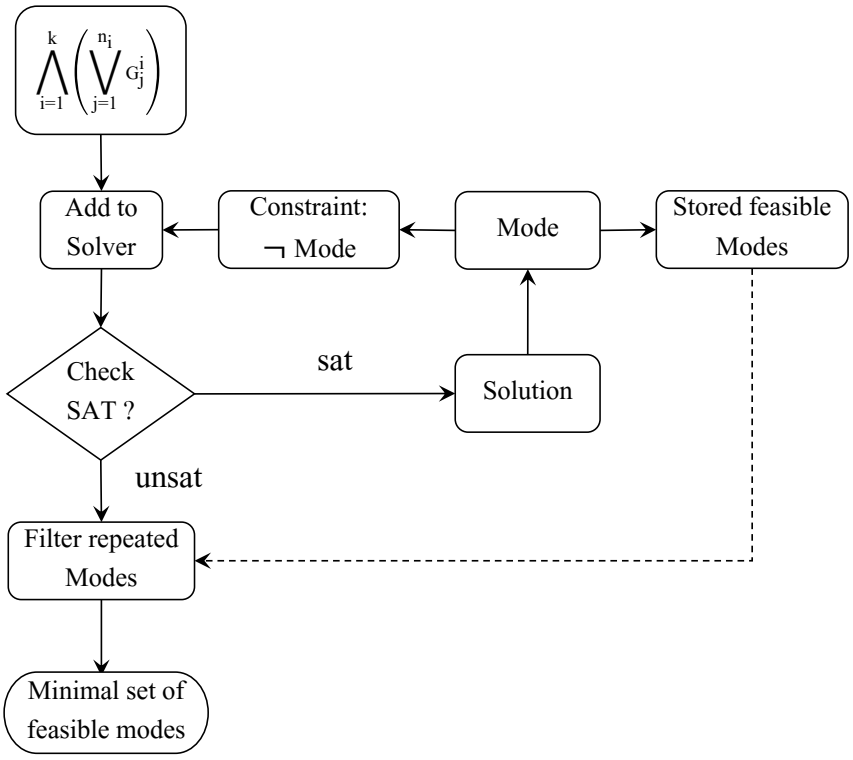


Figure 3.8 – Mode reduction algorithm. A set of constraints representing modes of a HA is added to the solver through a logical formula. The solver checks satisfiability and if satisfiable it returns a solution that is negated and added back to the solver. This loop is repeated until the constraints are unsatisfiable, thus, concluding the process. Finally, modes describing identical behaviours are removed.

A simple example describing the operation of the Z3 solver to identify satisfiable modes is presented in Figure 3.9. The input to the algorithm is the constraint $(x < 0 \vee y < 0) \wedge$

$(x \geq 0 \vee y \geq 0)$ that has only two satisfiable solutions: $\langle x < 0 \wedge y \geq 0 \rangle$ or $\langle y < 0 \wedge x \geq 0 \rangle$. The first time that satisfiability is checked (line 17), the solver returns a solution that is added back to the solver as a constraint using the logical *Not*. The second time the solver tries to solve the constraints, it returns the other solution that is also added to the solver as a constraint. When the solver inspect the constraints to find a solution for the third consecutive time, it is decided to be unsatisfiable as expected.

```
1  >> x, y = Int('x y')                          // Declaration of integer variables
2  >> p1, p2, p3, p4 = Bools('p1 p2 p3 p4')      // Declaration of boolean variables
3
4  >> slv = Solver()                             // creating the object solver
5  >> slv.add(And(Or(x<0,y<0),Or(x>=0,y>=0)))    // adding the input to the solver
6
7  // booleans for keeping track the different conditions previously added
8  >> slv.add(p1 == (x<0), p2 == (y<0), p3 == (x>=0), p4 == (y>=0))
9
10 >> slv.assertions()  // Shows current constraints in the Solver
11 [And(Or(x < 0, y < 0), Or(x >= 0, y >= 0)),
12 p1 == (x < 0),
13 p2 == (y < 0),
14 p3 == (x >= 0),
15 p4 == (y >= 0)]
16
17 >> slv.check()       // checking satisfiability
18 sat                  // returned value. Constraints are Satisfiable
19
20 >> slv.model()       // solving the constraints
21 [x = -1,             // returned solution
22 y = 0,
23 p1 = True,           // feasible mode = ⟨p1 ∧ p4⟩ = ⟨x < 0 ∧ y ≥ 0⟩
24 p2 = False,
25 p3 = False,
26 p4 = True]
27
28 // adding the feasible mode as a constraint to the solver
29 >> slv.add(Not(And(p1,p4)))
30
31 >> slv.check()       // checking satisfiability
32 sat                  // satisfiable constraints
33
34 >> slv.model()       // solving the constraints
35 [x = 0,              // solution
36 y = -1,
37 p1 = False,          // feasible mode = ⟨p2 ∧ p3⟩ = ⟨y < 0 ∧ x ≥ 0⟩
38 p2 = True,
39 p3 = True,
40 p4 = False]
41
42 // adding the feasible mode as a constraint to the solver
43 >> slv.add(Not(And(p2,p3)))
44
45 >> slv.check()                  // checking satisfiability
46 unsat                          // Unsatisfiable
```

Figure 3.9 – Description of the operation of the Z3 solver in the mode reduction algorithm. Starting with the formula $(x < 0 \vee y < 0) \wedge (x \geq 0 \vee y \geq 0)$, the Z3 solver returns all the possible solutions until the constraints are unsatisfiable. Lines started with ">>" are commands written in the Python console, the rest of lines are the solutions provided by the Z3 solver. The script was coded using the Z3 API in Python

4

Case study, Dog-Clutch

This chapter focuses on the results of applying the algorithm described in section 3.3.1 to measure test coverage for the case study, the Dog-clutch model provided by Volvo Cars Corporation. The system proposed is simple but complicated enough to carry on the objectives stated in the beginning of this master thesis. The first part of the section is dedicated to describe the dog-clutch system and the second part evaluates the results obtained.

4.1 Description of the Dog-Clutch

A dog-clutch is a mechatronic system. The following elements characterize its mechatronic nature:

a) **Mechanical system:** The physical system is composed by the following mechanical components: Dog-clutch, Cam-Ring and Spring. This is the plant of the model that determines the dynamical properties of the system.
b) **Computer Device:** All the information from the communication devices is gathered, processed and forwarded to the actuators by an electronic control unit. This implements a closed loop control algorithm that determines the logical attributes of the system.
c) **Communications:** The communication between the computer device and the physical system is done by means of a solenoid playing the role of an actuator and positioning sensors providing feedback to the control algorithm.

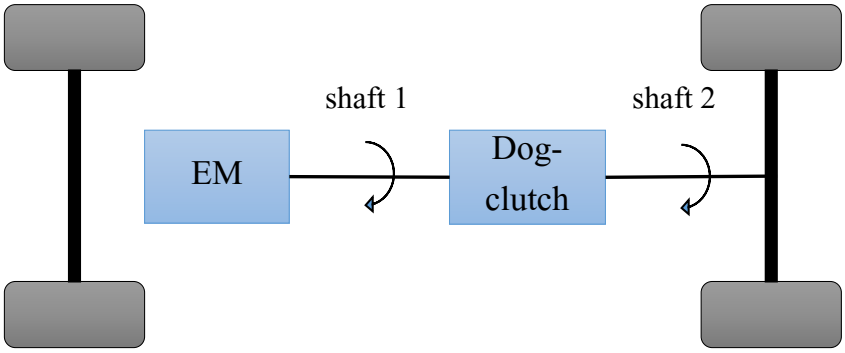


Figure 4.1 – Initially, the electric motor transmits the torque generated to the shaft 1. Then, the function of the dog-clutch is to propagate the torque in the shaft 1 to the shaft 2 that is connected to the wheels.

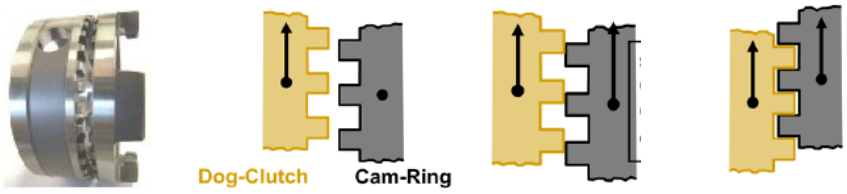


Figure 4.2 – Example of a dog-clutch system. a) Image of a real dog-clutch. b) disengaged state, the cam-ring is static c) synchronizing phase, the cam-ring starts moving in the same direction as the Dog-Clutch. A control algorithm synchronizes the velocity of the two parts to achieve the optimal position to engage. d) engagement phase, both components rotate with the same velocity. Cam-Ring begins to move axially until both parts are engaged. Figure provided by Volvo Car Corporation.

The function of a dog-clutch is to engage two rotating shafts to propagate motion. In the case studied here, the torque generated by an electric motor is transmitted to the wheels in order to move the vehicle as shown in Figure 4.1. The operation of the clutch system is shown in Figure 4.2, where the three main states are described. It consists of two pieces, a dog-clutch and a cam-ring that meshes together by moving the cam-ring towards the dog-clutch. The movement of the cam-ring is produced by an anchor that pushes it when a solenoid is energised. To disengage the system the solenoid must be deactivated. When the velocity of the two parts is synchronised the cam-ring moves axially to engage with the clutch-dog. This axial displacement is done through a solenoid controlled by current.

4.1.1 Modelling the Dog-Clutch

The starting point was the clutch model provided by Volvo Car Corporation, which was implemented in Simscape™. Though, since OpenModelica was the chosen tool for simulation purposes, the model was translated from Simscapre to Modelica language. Since both environments share the acausal modelling approach[1], it facilitated the code translation task. The final version of the code is presented in Appendix A.

It must be mentioned how the acausal approach influences the style of coding. Firstly, for all variables and parameters, it must be indicated the type (real, integer, boolean, ...). Secondly, all the variables must have assigned a physical quantity so that units are consistent in the equations. Finally, there is a specific section called *equation* where, as the name suggests, equations describing the system are added. Recall that equations are not assignments but relations between physical quantities.

The *equation* section is declared using an *if-then-else* structure that imposes the logics of the system, i.e. it represents the discrete dynamics of the system. Each conditional statement (if, elseif, else) is used to define a new situation where a set of equations describes the continuous behaviour at that state. In total, 20 distinct scenarios of the system are modelled and have been commented inside the Modelica code. The following variables and parameters forms the conditions statements defining the fore-mentioned scenarios:

[1]The characteristics of this approach are described in more detail in section 2.2.1

- z (m): Is a variable of the model used to describe the distance between the dog-clutch and the cam-ring as it can be seen in Figure 4.3.
- ω (rad/s): Is an input of the model describing the angular velocity difference between the dog-clutch and the cam-ring
- a (rad): Is a variable of the model describing the backlash phenomenon.
- Z_0, Z_1 (m): Are internal parameters of the model describing distance limits, shown in Figure 4.3.
- a_{lim} (m): Defines a half of backlash. Is a parameter of the model.
- w_{lim} (rad/s): Specifies the maximum value of ω to engage the system. Is a parameter of the model.
- *engage_req*: Defines with a 1 or 0 whether engage of the clutch is required. Is a parameter of the model.
- *prev_eng_state*: Binary number that informs about the state of the clutch in the previous time instant. Is a parameter of the model.

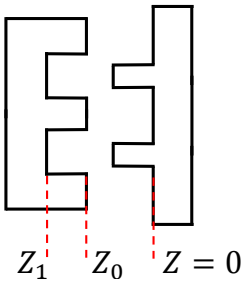


Figure 4.3 – The variable Z defines the distance between the two parts of the system. Z_0 delimits when the dog-clutch reaches the cam-ring. Z_1 delimits when the dog-clutch reaches the final position.

The importance of these variables and parameters lies in the fact that they will form the guards enabling transitions in the hybrid automaton used to analyse test coverage.

4.2 Implementation and evaluation

The Python programming language was used to implement the tool that performs test coverage. It was chosen primarily because of two reasons: (1) it is an efficient language manipulating strings so it is adequate to parse Hybrid DAEs provided by OpenModelica; (2) the Z3 solver has an API in Python so that it is easier to use it. The implementation of the proposed algorithm is presented following the chain of processes described in Figure 3.4 and will help to discuss the shortcomings encountered.

- **Inputs:** A model of the Dog-Clutch implemented in Modelica language and a test case in form of CSV file were the inputs of the algorithm. The first can be seen in the Appendix A and the test case was chosen randomly with the only requirement that the values of the variables should be inside the physical margins of the model.
- **OpenModelica:** The main Python application launched the OpenModelica software and created the hybrid DAE structure shown in Appendix B and a Matlab file with the results of the simulation. In the hybrid DAE two type of equations can be distinguished, one one hand, there are five simple assignment equations as those in lines 2, 3, 4, 5 and 6, in the other hand, four conditional equations in lines 1, 7, 21 and 30 that will be used to generate the modes of the HA. The conditions in these equations are expressions made up from the variables described in section 4.1.1.
- **Modes:** To identify all the possible modes contained in the original model, the conditions in each conditional equation were parsed in two different formats: Python and Z3. The reason for this was that the Z3 solver requires inputs in a specific format while the Python format was used to evaluate whether the conditions were true or false. A future work could be to improve the algorithm so that Z3 format can be used in both cases simplifying the parsing step. Finally, 48 conditions were obtained distributed as follows:

Conditional equation	Nr. of conditions
$Engage\_req$	2
$DER.zv$	20
t	12
eng_state	14

Table 4.1 – Number of conditions for each conditional equation described in the hybrid DAE structure shown in Appendix B.

As example, the conditions obtained after parsing the equation t are shown in Table 4.2.
The total number of modes obtained at this stage is the cross product of the conditions found, for the case study: $\{2\} \times \{20\} \times \{12\} \times \{14\} = 6720$. This number includes the infeasible modes that should be removed.

- **Mode Reduction:** After running the Z3 solver on python, only 34 of the 6720 modes were decided to be logical, which clearly represents a huge reduction in the amount of modes. The execution time of only 0.6 seconds performed by the Z3 solver was another pleasant surprise. Definitely, these two facts evidence the effectiveness reached by the solver. Further reduction of modes was achieved by checking the

dynamics described by those modes, resulting in 20 different behaviours that are shown in Table 4.3 where for each mode is shown the value assigned to each variable. Some problem were found when examining the results:

a) Mode 1 is unreachable and it appears due to the way the Modelica causal approach describes models.
b) 10 modes out of the 20 modes do not exist in the original model.

These two points falsifies the measure of test coverage since 100% will never be reached, this is a task to solve in future work.

- **Triggered Modes:** For the test case created, only nine modes were exercised throughout the simulation, they are identified in Table 4.3 with an asterisk. This is the more time consuming step of the proposed methodology since 48 conditions are evaluated at each time instant.
- **Test coverage:** The result of measuring test coverage is displayed and dumped into a txt file as in Figure 4.4. For the case study and using the inputs described above, only 45% of the feasible modes were exercised, indicating, that more test cases are needed to achieve higher coverage. Besides the numerical results, more information about the model can be dumped into the txt file as the triggered and feasible modes including the conditions that composes them.
  The results obtained here are considered useful for two reasons:
  a) the identified modes no triggered by the test case can be used as clues in an algorithm to automatically generate new test cases that increase the test coverage measure. The automated generation of test cases is the final objective of the model-based testing methods.
  b) numerically quantify how well a mechatronic system is tested.

  It must be noted that the final methodology implemented in this thesis generates modes that are not reachable so that 100% coverage could be impossible to reach as it happens with the case study analysed in this work.

```
1    Mode Coverage: 45.00%
2
3    Feasible Modes: 20   Visited Modes : 9
```

Figure 4.4 – Result of measuring test coverage using mode coverage as a criterion. Feasible modes are those that are mathematically coherent while visited modes are modes that have been exercised by a specific test case during simulation.

t_C0 =	(engage_req == 1 and prev_eng_state == 0) and (z < 0.0016) and (not ((z >= 0.0016 and abs(w) > w_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) <= a_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) > a_lim)))
t_C1 =	(engage_req == 1 and prev_eng_state == 0) and (z >= 0.0016 and abs(w) > w_lim) and (not ((z < 0.0016) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) <=a_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) >a_lim)))
t_C2 =	(engage_req == 1 and prev_eng_state == 0) and (z >= 0.0016 and abs(w) <= w_lim and abs(a) <=a_lim) and (not ((z < 0.0016) or (z >= 0.0016 and abs(w) > w_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) >a_lim)))
t_C3 =	(engage_req == 1 and prev_eng_state == 0) and (z >= 0.0016 and abs(w) <= w_lim and abs(a) >a_lim) and (not ((z < 0.0016) or (z >= 0.0016 and abs(w) > w_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) <=a_lim)))
t_C4 =	(engage_req == 1 and prev_eng_state == 0) and (not ((z < 0.0016) or (z >= 0.0016 and abs(w) > w_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) <=a_lim) or (z >= 0.0016 and abs(w) <= w_lim and abs(a) >a_lim)))
t_C5 =	(engage_req == 1 and prev_eng_state == 1) and (abs(a) >a_lim and z < 0.0032 and z >= 0.0016) and (not ((abs(a) >a_lim and z >= 0.0032 and z >= 0.0016)))
t_C6 =	(engage_req == 1 and prev_eng_state == 1) and (abs(a) >a_lim and z >= 0.0032 and z >= 0.0016) and (not ((abs(a) >a_lim and z < 0.0032 and z >= 0.0016)))
t_C7 =	(engage_req == 1 and prev_eng_state == 1) and (not ((abs(a) >a_lim and z < 0.0032 and z >= 0.0016) or (abs(a) >a_lim and z >= 0.0032 and z >= 0.0016)))
t_C8 =	(engage_req == 0 and prev_eng_state == 1) and (abs(a) >a_lim and z >= 0.0016) and (not ())
t_C9 =	(engage_req == 0 and prev_eng_state == 1) and (not ((abs(a) >a_lim and z >= 0.0016)))
t_C10 =	(engage_req == 0 and prev_eng_state == 0)
t_C11 =	not ((engage_req == 1 and prev_eng_state == 0) or (engage_req == 1 and prev_eng_state == 1) or (engage_req == 0 and prev_eng_state == 1) or (engage_req == 0 and prev_eng_state == 0))

Table 4.2 – Conditions extracted from the conditional equation t after parsing the Hybrid DAE in Appendix B. In this case the conditions were parsed to Python format

	$engage\_req$	$engage\_state$	t	$mass \cdot der(zv)$
Mode 1	0	0	0	0
Mode 2	0	0	0	$f\_spring - DampingBelleville\_extra \cdot zv$
Mode 3 *	0	1	$WBD \cdot w + stiff \cdot (-0.00437 + abs(a)) \cdot (\cdot Real \cdot)(sign(a))$	$f\_spring - WSD \cdot (-0.0016 + z) \cdot zv$
Mode 4	0	0	$HiSpeedDamping2 \cdot w$	$f\_spring - stiff2 \cdot (-0.0016 + z)$
Mode 5	0	0	$LoSpeedDamping2 \cdot w$	$f\_spring - stiff2 \cdot (-0.0016 + z)$
Mode 6	0	1	$WBD \cdot w + stiff \cdot (-0.00437 + abs(a)) \cdot (\cdot Real \cdot)(sign(a))$	$f\_spring - stiff2 \cdot (-0.0032 + z) - WSD \cdot (-0.0016 + z) \cdot zv$
Mode 7	0	0	0	$f\_spring - stiff2 \cdot (-0.0016 + z)$
Mode 8	0	1	0	$f\_spring - stiff2 \cdot (-0.0032 + z)$
Mode 9 *	0	0	0	f_spring
Mode 10 *	0	1	0	f_spring
Mode 11	1	0	0	0
Mode 12	1	0	0	$f\_spring - DampingBelleville\_extra \cdot zv$
Mode 13 *	1	1	$WBD \cdot w + stiff \cdot (-0.00437 + abs(a)) \cdot (\cdot Real \cdot)(sign(a))$	$f\_spring - WSD \cdot (-0.0016 + z) \cdot zv$
Mode 14	1	0	$HiSpeedDamping2 \cdot w$	$f\_spring - stiff2 \cdot (-0.0016 + z)$
Mode 15 *	1	0	$LoSpeedDamping2 \cdot w$	$f\_spring - stiff2 \cdot (-0.0016 + z)$
Mode 16 *	1	1	$WBD \cdot w + stiff \cdot (-0.00437 + abs(a)) \cdot (\cdot Real \cdot)(sign(a))$	$f\_spring - stiff2 \cdot (-0.0032 + z) - WSD \cdot (-0.0016 + z) \cdot zv$
Mode 17	1	0	0	$f\_spring - stiff2 \cdot (-0.0016 + z)$
Mode 18 *	1	1	0	$f\_spring - stiff2 \cdot (-0.0032 + z)$
Mode 19 *	1	0	0	f_spring
Mode 20 *	1	1	0	f_spring

Table 4.3 – Feasible Modes obtained by the developed application. Each mode is defined by four variables that change its dynamic at different modes. The asterisk indicates the modes exercised by the specific test case used during simulation.

5

Conclusion

A new approach to measure test coverage for systems with continuous dynamics is presented. The developed algorithm supports the use of Modelica language to describe CPSs in form of hybrid automata which can be used to define test coverage.

Mode coverage has been identified to be a good candidate as coverage criterion since it is able to capture the infiniteness of the continuous dynamics of the CPSs in a finite set of modes, which allow implementing quantitative methods to measure test coverage.

The study carried out in this thesis has examined the procedure to abstract CPSs implemented by means of an acausal approach into a mathematical description of a hybrid automaton. The results are promising to the extent that Hybrid DAEs created by OpenModelica can be used to automate the process of mode generation and measuring coverage. The results given by the algorithm permit to discover untested areas in the system and provide clues to guide the generation of new test cases to cover such areas, which is the main objective of Model-Based Testing approach. The approach followed can also enhance other testing methods in order to support testing of hybrid systems.

With regard to the reduction mode technique implemented, it is worth to remark the high performance shown by the Z3 solver to decide satisfiability and to remove infeasible modes yielding more accurate results. The approach proposed here could not have been achieved unless the enormous progress and development conducted in SMT solvers in recent times.

The usefulness of this approach still requires more development and testing with more general CPSs models in order to guarantee that software as OpenModelica always produces hybrid DAEs that are suitable to be transformed into hybrid automata. In addition, the Z3 solver also should be assessed against more complex models to verify its performance.

5.1 Limitations

This work succeeded in proving the validity of the methodology proposed, though, some limitations have been found, further described below.

a) The algorithm has been tested in a single, simple but still complex example to be able to assess the developed application.
b) The Z3 solver does not check for reachability, therefore it may happen that the algorithm generates feasible modes that are unreachable or physically impossible.
c) Z3 do not support real transcendental functions, e.g. sine, cosine and exponential. Thus, transcendental functions are not allowed in the conditions.

5.2 Future Work

This section presents the future work that can be done based on the results presented in this master's thesis.

- **Generate test cases:**
  The next natural step would be to develop a tool that automates the process to generate test cases to explore the localised untested areas, i.e. to continue with the implementation of the model-based testing methodology.
- **Verification against more complex systems:**
  The developed application has been tested only against one mechatronic system so more cases are necessary to verify the generation of hybrid automata and more thoroughly evaluate the approach followed.
- **Improve Reduction of modes:**
  In order to achieve an accurate test coverage measure, it is necessary to refine the mode reduction process to remove non-existent modes in the hybrid automata.
- **Analyse HA under other criteria:**
  Besides mode coverage criterion, other criteria such as transition coverage can be carried out to guarantee correctness. In addition, it would be interesting to analyse how well the continuous dynamic is tested within each mode.

Bibliography

[1] S. Miremadi, *Modellbaserad testning av mekatroniska system (testron) - vinnova*, 2016. [Online]. Available: `http://www.vinnova.se/sv/Resultat/Projekt/Effekta/2009-02186/Modellbaserad-Testning-av-Mekatroniska-System-TESTRON/` (visited on 09/02/2016).

[2] R. Alur, *Principles of cyber-physical systems*. MIT Press, 2015.

[3] E. A. Lee, "Cyber physical systems: Design challenges", in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, IEEE, 2008, pp. 363–369.

[4] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches", *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[5] M. Conrad, "Verification and validation according to iso 26262: A workflow to facilitate the development of high-integrity software", *Embedded Real Time Software and Systems (ERTS2 2012)*, 2012.

[6] M. A. Mäkinen, "Model based approach to software testing", PhD thesis, Citeseer, 2007.

[7] D. Firesmith, *Four types of shift left testing*, 2017. [Online]. Available: `https://insights.sei.cmu.edu/sei_blog/2015/03/four-types-of-shift-left-testing.html` (visited on 01/06/2017).

[8] J. Eddeland, J. Gil, R. Fransen, S. Miremadi, M. Fabian, and K. Åkesson, *Automated mode coverage analysis for model-based testing of hybrid automata*, 2016.

[9] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems", in *Hybrid systems*, Springer Berlin Heidelberg, 1993, pp. 209–229.

[10] J.-F. Raskin, "An introduction to hybrid automata", in *Handbook of networked and embedded control systems*, Springer, 2005, pp. 491–517.

[11] T. A. Henzinger, "The theory of hybrid automata", in *Verification of Digital and Hybrid Systems*, Springer, 2000, pp. 265–292.

[12] A. A. Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas, "Robust test generation and coverage for hybrid systems", in *International Workshop on Hybrid Systems: Computation and Control*, Springer, 2007, pp. 329–342.

[13] T. Nahhal and T. Dang, "Test coverage for continuous and hybrid systems", in *International Conference on Computer Aided Verification*, Springer, 2007, pp. 449–462.

[14] J. Kapinski, J. Deshmukh, X. Jin, H. Ito, and K. Butts, "Simulation-guided approaches for verification of automotive powertrain control systems", in *2015 American Control Conference (ACC)*, IEEE, 2015, pp. 4086–4095.

[15] R. Agrawal, "Semi-automated formalization and verification of automotive requirements using simulink design verifier", 2015.

[16] 2016. [Online]. Available: https://se.mathworks.com/products/simulink/ (visited on 11/30/2016).

[17] 2016. [Online]. Available: https://se.mathworks.com/products/simscape/ (visited on 11/30/2016).

[18] 2016. [Online]. Available: https://www.openmodelica.org/ (visited on 11/30/2016).

[19] P. Fritzson and et. al., *Openmodelica system documentation*, 2014-02-01 for OpenModelica 1.9.1 Beta1. Open Source Modelica Consortium, 2014, p. 14. [Online]. Available: https://openmodelica.org/svn/OpenModelica/trunk/doc/OpenModelicaUsersGuide.pdf (visited on 09/12/2016).

[20] A. Baresel, M. Conrad, S. Sadeghipour, and J. Wegener, "The interplay between model coverage and code coverage", in *Proc. EuroCAST*, 2003, pp. 136–190.

[21] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, "The art of software testing .", 2004.

[22] 2016. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html (visited on 12/10/2016).

[23] 2017. [Online]. Available: http://releases.llvm.org/download.html.

[24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[25] L. De Moura, *Z3 supports for nonlinear arithmetics*, 2016. [Online]. Available: http://stackoverflow.com/questions/18064822/z3-supports-for-nonlinear-arithmetics (visited on 12/21/2016).

A

Clutch Model

Listing A.1 – Clutch model implemented in Modelica language

```
encapsulated model clutch
import Modelica.Constants.pi;
import Modelica.SIunits.{Torque,Current,Velocity,Mass,TranslationalSpringConstant,
RotationalDampingConstant,TranslationalDampingConstant,Angle,Length,
RotationalSpringConstant,AngularVelocity,ElectricalForceConstant,
DynamicViscosity,Frequency};

// PARAMETERS
parameter Frequency freqHz = 1;
parameter Mass mass = 0.15;
parameter TranslationalSpringConstant k_spring = 100;
parameter RotationalDampingConstant WallBounceDamping = 10;
parameter TranslationalDampingConstant DampingBelleville = 10;
parameter DynamicViscosity WallScrapeDamp = 1e3;
parameter TranslationalDampingConstant DampingBelleville_extra = 100;
parameter Angle phi_init = 0;
parameter Length z_init = 0;
parameter RotationalSpringConstant stiff = 1e4;
parameter TranslationalSpringConstant stiff2 = 1e5;
parameter RotationalDampingConstant HiSpeedDamping2 = 0.01;
parameter RotationalDampingConstant LoSpeedDamping2 = 0.01;
parameter ElectricalForceConstant KI = k_spring * z1 / Ihold;


// VARIABLES
Integer eng_state(start = 0, fixed = true);
Integer engage_req;
Integer prev_eng_state;
Real a; // Backlash angle (radians)
Real f_spring_n_current; // Force exercised by the spring
Torque t(start = 0); // due to Friction
Length z(start = z_init, fixed = true); // position
Velocity zv(start = 0, fixed = true); // velocity
Angle phi(start = phi_init, fixed = true); // rotational angle

// INPUTS FROM AN EXTERN FILE
input Current current;
input AngularVelocity w;

// PROTECTED VARIABLES
protected
constant Real a_lim = sin(delta / 2);
constant Real wmax = 3;
constant Integer p = 36;
constant AngularVelocity w_lim = 30 * (pi / 30); //Speed treshold for engaging the clutch
constant Angle delta = 0.5 * (pi / 180);
constant Length z0 = 1.6e-3; // distance to stator teeth
constant Length z1 = 3.2e-3; // distance to fully engaged
constant Current Icl = 2.3;
constant Current Ihold = 1.2;
constant Current current_min = 0.01;
constant Length dog2dog_range = 0.2e-3;
```

```
53
54 // EQUATION SECTION
55 equation
56 prev_eng_state = pre(eng_state);
57 w = der(phi);
58 zv = der(z);
59 a = sin(p * phi) / p;
60 f_spring_n_current = KI * current - k_spring * z - DampingBelleville * zv;
61 if current > current_min then
62     engage_req = 1;
63 else
64     engage_req = 0;
65 end if;
66 if engage_req == 1 and prev_eng_state == 0 then
67 // engage requested,  clutch out of the cam-ring
68     if z < z0 then
69     // Rotor has not reached stator
70         eng_state = 0;
71         t = 0;
72         mass * der(zv) = f_spring_n_current;
73     elseif z >= z0 and abs(w) > w_lim then
74     // Rotor has reached stator
75     // Speed difefrence too large, glide on teeth
76         eng_state = 0;
77         t = HiSpeedDamping2 * w;
78         mass * der(zv) = f_spring_n_current - stiff2 * (z - z0);
79     elseif z >= z0 and abs(w) < w_lim and abs(a) <= a_lim then
80     // Rotor has reached stator
81     // Speed difference OK
82     // Tooth within gap bounds, engage clutch and move forward
83         eng_state = 1;
84         t = 0;
85         mass * der(zv) = f_spring_n_current;
86     elseif z >= z0 and abs(w) < w_lim and abs(a) > a_lim then
87     // Rotor has reached stator
88     // Speed difference OK
89     // Tooth outside gap bounds
90         eng_state = 0;
91         t = LoSpeedDamping2 * w;
92         mass * der(zv) = f_spring_n_current - stiff2 * (z - z0);
93     else
94         eng_state = 0;
95         t = 0;
96         mass * der(zv) = 0;
97     end if;
98 elseif engage_req == 1 and prev_eng_state == 1 then
99 // engage requested, Clutch engaged in previous state
100     if abs(a) > a_lim and z < z1 and z >= z0 then
101     // teeth in the gap, hitting the wall bounds, clutch not in final position
102         eng_state = 1;
103         t = stiff * (abs(a) - a_lim) * sign(a) + WallBounceDamping * w;
104         mass * der(zv) = f_spring_n_current - WallScrapeDamp * (z - z0) * zv;
105     elseif abs(a) > a_lim and z >= z1 and z >= z0 then
106      // teeth in the gap, hitting the wall bounds, clutch not in final position
107         eng_state = 1;
108         t = stiff * (abs(a) - a_lim) * sign(a) + WallBounceDamping * w;
109         mass * der(zv) = f_spring_n_current - stiff2 * (z - z1) - WallScrapeDamp*
110                                                                 * (z - z0) * zv;
111     elseif abs(a) <= a_lim and z < z1 and z >= z0 then
112     // teeth in the gap, not hitting the wall bounds, clutch not in final position
113         eng_state = 1;
114         t = 0;
115         mass * der(zv) = f_spring_n_current;
116     elseif abs(a) <= a_lim and z >= z1 and z >= z0 then
117     // teeth in the gap, not hitting the wall bounds, clutch in position
118         eng_state = 1;
119         t = 0;
120         mass * der(zv) = f_spring_n_current - stiff2 * (z - z1);
121     elseif z < z0 then
122     // teeth outside gap
123         eng_state = 0;
124         t = 0;
```

```
125         mass * der(zv) = f_spring_n_current;
126     else
127         eng_state = 0;
128         t = 0;
129         mass * der(zv) = 0;
130     end if;
131 elseif engage_req == 0 and prev_eng_state == 1 then
132 // Detatch clutch. Clutch engaged in previous state
133     if abs(a) > a_lim and z >= z0 then
134     // teeth in the gap, hitting the wall bounds
135         eng_state = 1;
136         t = stiff * (abs(a) - a_lim) * sign(a) + WallBounceDamping * w;
137         mass * der(zv) = f_spring_n_current - WallScrapeDamp * (z - z0) * zv;
138     elseif abs(a) <= a_lim and z >= z0 then
139     // teeth in the gap, not hitting the wall bounds
140         eng_state = 1;
141         t = 0;
142         mass * der(zv) = f_spring_n_current;
143     elseif z < z0 then
144     // clutch was engaged but now has somehow reached outside dog teeth
145         eng_state = 0;
146         t = 0;
147         mass * der(zv) = f_spring_n_current;
148     else
149         eng_state = 0;
150         t = 0;
151         mass * der(zv) = 0;
152     end if;
153 elseif engage_req == 0 and prev_eng_state == 0 then
154 // Detatch clutch.
155     if z < z0 and z >= 0 then
156     // clutch not engaged, i.e. outside dog teeth
157         eng_state = 0;
158         t = 0;
159         mass * der(zv) = f_spring_n_current;
160     elseif z < z0 and z < 0 then
161     // clutch not engaged
162         eng_state = 0;
163         t = 0;
164         mass * der(zv) = f_spring_n_current - DampingBelleville_extra * zv;
165     elseif z >= z0 then
166     // teeth buncing of eachother
167         eng_state = 0;
168         t = 0;
169         mass * der(zv) = f_spring_n_current - stiff2 * (z - z0);
170     else
171         eng_state = 0;
172         t = 0;
173         mass * der(zv) = 0;
174     end if;
175 else
176     eng_state = 0;
177     t = 0;
178     mass * der(zv) = 0;
179 end if;
180 end clutch;
```

B

Hybrid DAE

Listing B.1 – SimCode gernerated by OpenModelica for the case study

```
engage_req=if current > 0.01 then 1 else 0[Integer ]
f_spring_n_current=KI * current + (-k_spring) * z - DampingBelleville * zv[Real ]
a=0.02777777777777778 * sin(36.0 * phi)[Real ]
DER.z=zv[Real ]
DER.phi=w[Real ]
prev_eng_state=pre(eng_state)[Integer ]
DER.zv=if engage_req == 1 and prev_eng_state == 0 then if z < 0.0016 then DIVISION(f_spring_n_current, mass) else if z >= 0.0016 and
abs(w) > 3.141592653589793 then DIVISION(f_spring_n_current - stiff2 * (-0.0016 + z), mass) else if z >= 0.0016 and abs(w) < 3.141592653589793
and abs(a) <= 0.004363309284746571 then DIVISION(f_spring_n_current, mass) else if z >= 0.0016 and abs(w) < 3.141592653589793 and abs(a) >
0.004363309284746571 then DIVISION(f_spring_n_current - stiff2 * (-0.0016 + z), mass) else 0.0 else if engage_req == 1 and prev_eng_state == 1
then if abs(a) > 0.004363309284746571 and z < 0.0032 and z >= 0.0016 then DIVISION(f_spring_n_current - WallScrapeDamp * (-0.0016 + z) * zv, mass)
else if abs(a) > 0.004363309284746571 and z >= 0.0032 and z >= 0.0016 then DIVISION(f_spring_n_current - stiff2 * (-0.0032 + z) - WallScrapeDamp *
(-0.0016 + z) * zv, mass) else if abs(a) <= 0.004363309284746571 and z < 0.0032 and z >= 0.0016 then DIVISION(f_spring_n_current, mass) else if
abs(a) <= 0.004363309284746571 and z >= 0.0032 and z >= 0.0016 then DIVISION(f_spring_n_current - stiff2 * (-0.0032 + z), mass) else if z < 0.0016
then DIVISION(f_spring_n_current, mass) else 0.0 else if engage_req == 0 and prev_eng_state == 1 then if abs(a) > 0.004363309284746571 and z >=
0.0016 then DIVISION(f_spring_n_current - WallScrapeDamp * (-0.0016 + z) * zv, mass) else if abs(a) <= 0.004363309284746571 and z >= 0.0016 then
DIVISION(f_spring_n_current, mass) else if z < 0.0016 then DIVISION(f_spring_n_current, mass) else 0.0 else if engage_req == 0 and prev_eng_state
== 0 then if z < 0.0016 and z >= 0.0 then DIVISION(f_spring_n_current, mass) else if z < 0.0016 and z < 0.0 then DIVISION(f_spring_n_current -
DampingBelleville_extra * zv, mass) else if z >= 0.0016 then DIVISION(f_spring_n_current, mass) else 0.0 else 0.0[Real ]

t=if engage_req == 1 and prev_eng_state == 0 then if z < 0.0016 then 0.0 else if z >= 0.0016 and abs(w) > 3.141592653589793 then
HiSpeedDamping2 * w else if z >= 0.0016 and abs(w) < 3.141592653589793 and abs(a) <= 0.004363309284746571 then 0.0 else if z >= 0.0016 and
abs(w) < 3.141592653589793 and abs(a) > 0.004363309284746571 then LoSpeedDamping2 * w else 0.0 else if engage_req == 1 and prev_eng_state == 1
then if abs(a) > 0.004363309284746571 and z < 0.0032 and z >= 0.0016 then WallBounceDamping * w + stiff * (-0.004363309284746571 + abs(a))
* /*Real*/(sign(a)) else if abs(a) > 0.004363309284746571 and z >= 0.0032 and z >= 0.0016 then WallBounceDamping * w + stiff *
(-0.004363309284746571 + abs(a)) * /*Real*/(sign(a)) else 0.0 else if engage_req == 0 and prev_eng_state == 1 then if abs(a) >
0.004363309284746571 and z >= 0.0016 then WallBounceDamping * w + stiff * (-0.004363309284746571 + abs(a)) * /*Real*/(sign(a)) else 0.0
else if engage_req == 0 and prev_eng_state == 0 then -0.0 else 0.0[Real ]

eng_state=if engage_req == 1 and prev_eng_state == 0 then if z < 0.0016 then 0.0 else if z >= 0.0016 and abs(w) > 3.141592653589793 then 0.0
else if z >= 0.0016 and abs(w) < 3.141592653589793 and abs(a) <= 0.004363309284746571 then 1 else 0.0 else if engage_req == 1 and
prev_eng_state == 1 then if abs(a) > 0.004363309284746571 and z < 0.0032 and z >= 0.0016 then 1 else if abs(a) > 0.004363309284746571 and
z >= 0.0032 and z >= 0.0016 then 1 else if abs(a) <= 0.004363309284746571 and z < 0.0032 and z >= 0.0016 then 1 else if abs(a) <=
0.004363309284746571 and z >= 0.0032 and z >= 0.0016 then 1 else 0.0 else if engage_req == 0 and prev_eng_state == 1 then if abs(a) >
0.004363309284746571 and z >= 0.0016 then 1 else if abs(a) <= 0.004363309284746571 and z >= 0.0016 then 1 else 0.0 else if engage_req == 0 and
prev_eng_state == 0 then -0.0 else 0.0[Real ]
```