AUTOMATED TESTING OF ROBOTIC SYSTEMS IN SIMULATED ENVIRONMENTS

Sebastian Andersson & Gustav Carlstedt
san13015@student.mdh.se & gct14001@student.mdh.se

Examiner: Daniel Sundmark
Mälardalen University, Västerås, Sweden

Supervisors: Alessandro Papadopoulos & Eduard Paul Enoiu
Mälardalen University, Västerås, Sweden

Company supervisor: Pär Muhr
ABB Robotics, Västerås, Sweden
June 24, 2019
Abbreviations

ABB Asea Brown Boveri
ASEA Allmänna Svenska Elektriska AB
BIC Bayesian Information Criterion
DC Direct Current
DoF Degrees of Freedom
GA Genetic Algorithm
GM General Motors
GMM Gaussian Mixture Model
HD Hausdorff Distance
HIL Hardware In the Loop
HW Hard Ware
I/O Input/Output
IRB Industrial Robot
LR Linear Regression
MIL Model In the Loop
ML Machine Learning
MOC Motion Configurations
MS Mutant Score
PC Personal Computer
PDM Point Distribution Model
PLS Partial Least Square
RTS Regression Test Selection
RQ Research Question
RS RobotStudio
SDD Simulation Driven Development
SLR Switching Linear Regression
SVM Support Vector Machines
SIL Software In the Loop
TCP Tool Center Point
VC Virtual Controller
VCSIM Virtual Controller Simulation
# Contents

1 Introduction ................................. 2

2 Background ................................. 3
   2.1 Industrial Robotics ......................... 3
      2.1.1 Industrial Robot ...................... 3
      2.1.2 Joints and Degrees of Freedom ........ 3
      2.1.3 Workspace .......................... 3
      2.1.4 IRB120 ................................ 4
      2.1.5 Trajectories .......................... 4
   2.2 Trajectory Analysis ....................... 4
      2.2.1 Euclidean Distance ..................... 5
      2.2.2 Hausdorff Distance ..................... 5
      2.2.3 Fréchet Distance ....................... 5
   2.3 Input Space ............................. 6
      2.3.1 RAPID Instructions .................... 6
      2.3.2 Robot Configuration .................... 7
   2.4 Output Signals ......................... 7
      2.4.1 Tool Center Point ..................... 8
      2.4.2 Velocity ............................ 8
      2.4.3 Motor Position ....................... 8
   2.5 Software Testing ......................... 8
      2.5.1 What is a Bug ........................ 9
      2.5.2 Test Case and Test Set ................ 9
      2.5.3 Mutation Testing ..................... 10
      2.5.4 Software In the Loop ................. 10
      2.5.5 Hardware In the Loop .................. 10
      2.5.6 Black Box Testing ..................... 10
   2.6 Regression Testing ....................... 11
      2.6.1 Selection ........................... 11
      2.6.2 Minimisation ......................... 11
      2.6.3 Prioritisation ......................... 11

3 Problem Formulation ....................... 12

4 Related Work ............................. 13
   4.1 Simulation Environment .................... 13
   4.2 Regression Testing ....................... 13
      4.2.1 Selection ........................... 14
      4.2.2 Prioritisation ......................... 14
   4.3 Test Case Generation ..................... 14
      4.3.1 Test Case Generation Evaluation ........ 14
   4.4 Similarity Analysis ...................... 15
   4.5 Bug Detection .......................... 16

5 Method ..................................... 17
   5.1 Research Methodology ..................... 17
   5.2 Simulation Environment .................... 18
      5.2.1 Signal Extraction ...................... 18
   5.3 Test Case .............................. 19
      5.3.1 Auto Generation ....................... 19
      5.3.2 Mercedes Path ........................ 20
   5.4 Creation of Artificial Bugs ............... 21
   5.5 Similarity Analysis ...................... 21
   5.6 Test Set Reduction ...................... 22
5.6.1 Test Set Evaluation ........................................... 22
5.6.2 Selection .................................................. 23
5.6.3 Prioritisation ............................................... 23
5.6.4 Minimisation ............................................... 23
5.6.5 Test Set Validation Model ............................... 23

6 Implementation .................................................. 25
6.1 Simulated Environment ..................................... 25
6.2 Implementations to the Test Engine .................. 25

7 Experimental Setup ............................................ 27
7.1 Simulation Deviations ...................................... 27
7.2 Controller restart .......................................... 27
7.3 Verification of Test Case Creation ....................... 28
  7.3.1 Target Rotation Experiment ............................. 28
  7.3.2 Test Case Generation Type Experiment .............. 28
7.4 Test Set Reduction and Real Bug Detection .......... 28
  7.4.1 Artificial Bug Detection ................................. 29
  7.4.2 Real Bug Detection ...................................... 29
  7.4.3 Compressed Test Set Validation ......................... 29

8 Experimental Results ........................................ 30
8.1 Simulation Deviations ...................................... 30
8.2 Controller Restart .......................................... 30
8.3 Verification of Test Case Creation ....................... 31
  8.3.1 Target Rotation Experiment ............................. 31
  8.3.2 Test Case Generation Type Experiment .............. 32
8.4 Test Set Reduction and Bug Detection .................. 32
  8.4.1 Artificial Bug Detection ................................. 33
  8.4.2 Real Bug Detection ...................................... 33
  8.4.3 Compressed Test Set Validation ......................... 33

9 Discussion ..................................................... 37
9.1 Future Work ................................................ 37
  9.1.1 Bug Detected ........................................... 37
  9.1.2 No Bug Detected ........................................ 38

10 Conclusion .................................................... 39

Appendix A Experiment Data ................................. 45
Abstract

With the simulations tools available today, simulation can be utilised as a platform for more advanced software testing. By introducing simulations to software testing of robot controllers, the motion performance testing phase can begin at an earlier stage of development. This would benefit all parties involved with the robot controller. Testers at ABB would be able to include more motion performance tests to the regression tests. Also, ABB could save money by adapting to simulated robot tests and customers would be provided with more reliable software updates. In this thesis, a method is developed utilising simulations to create a test set for detecting motion anomalies in new robot controller versions. With auto-generated test cases and a similarity analysis that calculates the Hausdorff distance for a test case executed on controller versions with an induced artificial bug. A test set has been created with the ability to detect anomalies in a robot controller with a bug.
1 Introduction

The demand for industrial robots on the market is greater than ever. According to the international federation of robotics, the sales of industrial robots has increased drastically since 2007 [1]. In the most recent sales statistics, 2017 is reported as a record-breaking year with 381,335 industrial robots sold. In the near future sales are predicted to continue to increase with a mean of 14% per year until 2021. The source of this demand can be traced back to 1961 when the first industrial robot, Unimate, was installed at General Motors, created by George Devol. This robot was hydraulically powered and designed to assist a die-casting machine. The first robot was a great success in production, but the real break came when General Motors installed 66 robots in the mid-1960s [2], [3]. At this point, the potential in robot manufacturing grew and ASEA, later merged with BBC A.G. to form ABB, introduced the well known Industrial Robot (IRB) series [4]. Nowadays, almost 60 years since the first robot was installed, the number of industrial robotics manufacturers continues to increase.

As simulation software is advancing and can handle more complex tasks, it can be applied in more applications. The benefits of including simulation modelling at an early stage of development have been studied for decades. Early studies forecast a bright future in simulation tests for software development [5], which will enable developers to have a test platform up at an early stage, meaning that the test phase can begin before the hardware is available [6]. Research has shown that simulations have improved software development of robotic applications. As a whole, the results in adapting simulation modelling have shown an overall increase in performance and a decrease in the cost of the end product [7], [8]. This has caught the eye of the larger companies as a possibility to improve software development and get ahead of the competitors on the market. For many companies, this opens new possibilities in software testing but also introduces new areas of research.

Today, most software tests regarding testing of industrial robots involve hardware components. This is how most of the bug detection is performed when executing a regression test on the robots available in the test rigs. The test set for the regression tests is carefully selected from a test case database. Test cases are prioritised if they have recently failed or not been executed in a while. This way of testing is reliable in the sense that all tests are executed on actual robots but also have a few drawbacks. Tests are expensive and limited to execute depending on the number of test rigs available. Also, it is difficult to extract precise data samples regarding the movement of the robot.

It has been highlighted that software updates regarding the performance of the robot controllers are neglected by customers. The updates are rejected due to the uncertainty of remaining the same robot behaviour in the customer specific environment. This has raised a request to widen the software tests of the robot controllers. In some occasions, a bug may have gone undetected to the customer. This could be due to that the in-house test rigs are insufficient to evaluate some customer specific requirements. By introducing simulation-based software test the robot controller software is believed to be more thoroughly tested before being released to customers.

This thesis presents a feasibility study evaluating if, or how, simulation-based software testing could be applied to improve bug detection. This will be done by creating regression tests that can participate in a nightly test run on the most recent controller versions. This could potentially increase bug detection at an earlier stage of development. If this simulation-based software test method should prove to be reliable a future application could be to build simulations of actual customer applications. This would make it possible to simulate the behaviour of new software with the same setup as the system receiving the update. The scientific contributions presented in this thesis are:

1. A method for auto-generating RAPID test cases, that can be used to create an initial test set, later used for regression testing.
2. A method for evaluating the similarity between two versions of the same robot controller handling the same task. This is used to highlight bugs by identifying abnormalities in robot movement.
3. A process to reduce the test set to make the regression test execution more effective and less time-consuming.
2 Background

This section will go through the basic concepts and terminology that will be used throughout this thesis. The different areas that are described here are the fundamentals in industrial robotics, software testing, and methods for trajectory analysis.

2.1 Industrial Robotics

The fundamentals of industrial robotics are introduced, e.g., how the robot is constructed, its movements, restraints, and reach.

2.1.1 Industrial Robot

An industrial robot is a programmable robot typically located in manufacturing applications. The robot is constructed with a number of links, joined together with electrical motors. These are utilised to actuate the links. Robots can be equipped with tools to be applied in various applications [9]. For instance, these applications could include welding, gluing, and painting. A simple description of how the user can communicate with the robot, and decide its functionality can be seen in Figure 1. This figure demonstrates the structure of how the robot is configured and operate. The mechanical unit is located to the right in the figure. The logic and controlling decisions are made in the controller. To the left is the Flexpendant, which is an interface in which the user can choose to load programs and operate.

![Figure 1: Basic components of an industrial robot system.](image)

In some situations, these robots are simply programmed to perform the same execution every time. For instance, welding at a certain location, for every part that needs welding. In other applications, the movement of the robot could depend on the object of interaction. This implies, for instance, sorting items according to a specific requirement.

2.1.2 Joints and Degrees of Freedom

An industrial robot is constructed with links connected through joints [10]. The basic joints can either be prismatic or revolute. Other types of joints exist, e.g., the universal joint or the saddle joint, but they can be re-conducted to the two basic joints. A prismatic joint is typically a pneumatic or hydraulic system and a revolute joint is usually a servo. The Degrees of Freedom (DoF) define the minimum number of coordinates that is needed to determine the position and orientation of a rigid body in space. The basic joints allow for 1-DoF motion, i.e., either a motion along or rotation about a single axis. For an industrial robot, the DoF is decided based on how many basic joints it consists of.

2.1.3 Workspace

The workspace of the robot is defined by the robot’s reach [10]. The reach of the robot differs depending on how it is constructed, the type of joints it has, how many DoF it has, the number of links it consists of, and physical constraints of the actuation. The reach of the robot is measured
from the base of the arm to the end-effector in positions where the robot is fully stretched, see Figure 3. To reach the point in space the end-effector of the robot has to be positioned in the given coordinates. However, to reach a point the robot can be manipulated in infinite different poses. It is beneficial to pay attention to how the robot actually approaches a target. Since a revolute joint only performs circular movements the robot has to move several joints at the same time in order to manipulate the end-effector as desired. A robot can be restricted by the environment when performing a certain movement. This is because the robot will be halted if it is out of reach.

2.1.4 IRB120

The test platform selected for this thesis is the IRB120, which is ABB’s smallest multi-functional robot [11].

The IRB120 consists of six links and six revolute joints, as can be seen in Figure 2, each joint has an angular reach limit, see Table 1. The end-effector is initially located at the end of axis six, if a tool is attached at the end of axis six the end-effector position will be moved.


Table 1: The IRB120 angular restrictions for each joint.

<table>
<thead>
<tr>
<th>Axis</th>
<th>Range (Degrees)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 rotation</td>
<td>+165 to -165</td>
</tr>
<tr>
<td>2 arm</td>
<td>+110 to -110</td>
</tr>
<tr>
<td>3 arm</td>
<td>+70 to -110</td>
</tr>
<tr>
<td>4 wrist</td>
<td>+160 to -160</td>
</tr>
<tr>
<td>5 bend</td>
<td>+120 to -120</td>
</tr>
<tr>
<td>6 turn</td>
<td>+400 to -400</td>
</tr>
</tbody>
</table>

Figure 2 shows the workspace of the IRB120 robot. The left-hand side of the figure shows the area that the robot arm can be manipulated to reach. On the right-hand side is the base rotation restrictions of the robot. The IRB120 has a reach of 0.58m see Figure 3, and a position reputability of 0.01mm it is designed to operate in compact production facilities.

2.1.5 Trajectories

A robot moves along a trajectory by constantly manipulating the position of the Tool Center Point (TCP). A trajectory is defined within the workspace of the robots reach as a path between a start and an endpoint in the room or space. The path is divided into several intermediate points and is given a start and end time [12]. For the evaluation of a robot controller trajectory tracking behaviour, there exists a predefined standard path that ABB utilise. The path is called the Mercedes path and is a commonly used test trajectory at ABB. It consists of sharp edges, zigzag turns, and circular movements, see Figure 4.

2.2 Trajectory Analysis

The similarity between two given trajectories can be a useful measurement when comparing two different robot control versions. By measuring the gradient distance between all data points along
two trajectories, a similarity measurement can be extracted. Three different measurement methods are to be briefly described.

2.2.1 Euclidean Distance

One method for assessing the quality of trajectory tracking is to compare the Euclidean distance between a reference trajectory and the performed trajectory [13]. This approach is to map points in one trajectory to the corresponding point in the other trajectory and calculate the distance between them. The mathematical definition of the Euclidean distance between two points $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$ in the Euclidean space, is

$$d(p_1, p_2) = \|p_2 - p_1\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}, \quad (1)$$

where $\|x\|$ indicates the 2-norm of a vector $x$.

2.2.2 Hausdorff Distance

Hausdorff is a method used to find similarities between trajectories. This method determine the maximum of the minimum distances between the sets $A = \{a_1, a_2, \ldots, a_n\}, B = \{b_1, b_2, \ldots, b_n\}$, where $A, B \in E^n$. The dimensions can be stretched linearly by extending the method for calculating distances with the aid of linear algebra. The minimum distance is calculated by counting the distances from each point in one set to all points in the other set. The maximum of these minimum distances is chosen. The mathematical definition of the Hausdorff distance between sets $A$ and $B$ is

$$\tilde{\delta}_H(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\|. \quad (2)$$

$\tilde{\delta}_H$ is the one-sided Hausdorff distance [14]. Hausdorff is executing with a linear complexity when calculating between polygons, according to $O(m+n)$, where $m$ and $n$ denote the number of vertices in the trajectory [15]. The Hausdorff distance where the input sets are the same, will output the distance zero, formally defined as the identity property, i.e.

$$\tilde{\delta}_H(A, B) = 0 \Leftrightarrow A = B$$

The Hausdorff distance has been used for analysing the discrepancy between different trajectories in robotics applications [16], [17], as well as more complex systems, such as hybrid systems [18].

2.2.3 Fréchet Distance

The Fréchet distance $\delta_F$, described in a paper by Eiter and Mannila, is used to calculate the similarity between two curves in the geometric space [19], [20]. One example used to describe how
it works by considering a person walking a dog on a leash. The person’s path acts as one of the curves and the dog as the other. Either may change speed but may not reverse. The idea is to calculate the shortest length of the leash to be able to traverse both paths. Formally, let \( f(\alpha(t)) \) and \( g(\beta(t)) \) be the two trajectories. The Fréchet distance is computed as

\[
\delta_F = \inf_{\alpha, \beta} \max_{t \in [0,1]} d(f(\alpha(t)), g(\beta(t)))
\]

A known issue when calculating the exact Fréchet distance is the high search complexity in the algorithm. Similarly to the Hausdorff distance, the Fréchet distance has been already used for analysing the discrepancy between different trajectories in different applications [16]–[18]. In an article by Holladay and Srinivasa, it is described how the Fréchet distance is used to decide paths for a robot’s arm movements [21]. The Fréchet distance is very computationally heavy to calculate, due to its high complexity. The complexity for finding the Fréchet distance can be as best proportional to \( O(m \times n) \), where \( m \) and \( n \) are the numbers of vertices in the two trajectories [22].

2.3 Input Space

The input space determines which input parameters can be utilised for the testing phase, and to what extent. The input space in this thesis refers to all parameters that influence the controller, i.e controller inputs. These inputs can be modified with robot controller code written in RAPID\(^3\) and configuration files. They are divided into two separate sets, one set with inputs that are changed in the RAPID move instructions and the other set with inputs that are changed in the robot configuration file.

2.3.1 RAPID Instructions

RAPID is a programming language to describe the functionality and movement of an industrial robot. An example of a simple RAPID code can be seen in Listing 1. The program possesses a module to define the tools, targets and global variables. The targets, which are points in the robot environment, are defined by their global position, orientation, robot configuration, and position of

external axes. In the figure, a tool called tSpintec is defined, as well as four targets. A tSpintec tool is a custom tool that is provided from the ABB tool library. The tool possesses more possible settings, where some are orientation, weight, and position of the centre of gravity. In the main procedure, the move instructions are defined. A move instruction is a line of code describing how the robot is to move between two points. It can be linear, circular, or absolute. The absolute instruction is provided with the angle for each motor in which it should be positioned. The controller settings that can be modified are speed and accuracy. The speed determines how fast the robot move, in mm/s, between two points and the accuracy is how close to a point, in mm, the TCP has to be considered it has reached. The "\V" and "\Z" in the move instructions are options to specify the chosen velocity and zone parameters, to choose customised values.

```
MODULE TC
    PERS tooldata tSpintec :=[TRUE, [[31.793, 0.229, 639], [0.945518576, 0.325568154, 0]], [1.706, [-51.132, 0, 99.658], [1, 0, 0], [0, 0, 0]]];
    VAR robotarget Target_1 :=[[48.392, 428], [0, -0.707106781, 0.707106781], [-1.0, -1.0], [9E+09, 9E+09, 9E+09, 9E+09, 9E+09, 9E+09]];
    VAR robotarget Target_2 :=[[40.402, 434], [0, -0.707106781, 0.707106781], [-1.0, -1.0], [9E+09, 9E+09, 9E+09, 9E+09, 9E+09, 9E+09]];

PROC main()
    MoveL Target_1, v1000\V:=1100, x1\Z:=0, tSpintec\WObj:=wobj0;
    MoveL Target_2, v1000\V:=1034, x1\Z:=2, tSpintec\WObj:=wobj0;
ENDPROC
ENDMODULE
```

Listing 1: This is an example of what a simple RAPID program may look like. This example moves the robot linearly to Target_2. Initially, the robot is positioned with its TCP in a pre-defined home point.

2.3.2 Robot Configuration

There exist ways to configure the robot that may alter its performance. The existing configurations are divided into 5 topics [23]:

- **Communication**: Parameters regarding the serial channels and file transfer protocol.
- **Controller**: Include safety parameters for RAPID specific functions.
- **I/O System**: Handles in- and outputs for boards and signals.
- **Man-Machine Communication**: Parameters for customising lists for instructions and I/O signals.
- **Motion**: Includes parameters regarding motion control for the robot and external equipment.

Each topic has a set of types, which in turn possess a set of parameters. These parameters alter a specific aspect of functionality in the controller. There are 40 parameters in the motion domain that alter the motion configurations of the controller. The configurations are utilised by loading them to the controller. Many of the configuration parameters exist for the end user to alter the robot behaviour to its desired functionality and application, while some are only visible to the employees at ABB. These can be harmful to the robot if they are changed without the knowledge of their consequences.

2.4 Output Signals

Behavioural data from each simulation execution is extracted from signals defined in the RAPID program. These signals are extracted at two separate stages of the execution, one based on algebraic kinematic calculations (VC-signals), and the other from the Algoryx simulation (VCSIM-signals). Plenty of different data describing the behavioural aspects of the simulation can be monitored using these signals.
2.4.1 Tool Center Point

The TCP data contains data points describing the coordinates and orientation of the robots TCP, with respect to any attached tool, in each data point. It is sampled over time and comes in sets of seven, three values describing the position (X-, Y-, and Z-coordinates) and four values describing the orientation in quaternions.

2.4.2 Velocity

Velocity for a robot can be described as displacement over time and can be derived from the TCP data. Each entry in the data from the output signals has a time stamp and the corresponding position in space. From this, the time difference and distance between points can be calculated as

\[ v(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \]

where \( v \) is the velocity between two points \( p_1 = (x_1, y_1, z_1) \) and \( p_2 = (x_2, y_2, z_2) \) in Euclidean space, and \( t \) is the time difference between two data points. This can be performed on any of the mentioned TCP data sets.

2.4.3 Motor Position

When monitoring motor positions the data contain the angle, in degrees, for each motor axis. For the robot used in this thesis, it results in six sets of data per simulation run.

2.5 Software Testing

Software testing is applied during the development of a product to verify that the end-product acts as expected. This is done by creating a well-structured general purpose test criteria that fit the model or structure under development. There exist a few ways to produce well-designed test approaches, depending on the level on which the software is to be tested. Deciding the designing approach for requirements and their tests for software testing is important, to make the end product more reliable. As described by Ammann and Offutt [24], some areas of software testing utilised in accordance with software development are:

- Acceptance Testing: A method for evaluating product behaviour according to certain requirements.
- System Testing: A method when the tester evaluates the software in its intended environment.
- Integration Testing: A similar method to system testing, but with more focus on subsystem interactivity.
- Module Testing: A method of software testing with respect to more detailed design.
- Unit Testing: A method of evaluating the software with regards to a certain implementation.

These areas have different objectives and are used for different applications. How these test methods can be mapped to software development can be seen in Figure 5, referred to as the V-model. Each testing method correlates a certain part of software development. The entire chain is structured in a layered format, where the topmost layer is where requirements are tested to be accepted or not. In the lowest level, unit tests are run on the raw implementation. This structure enables the development team to choose the appropriate number of layers, depending on the intention of the test phase.

The goal with requirement analysis is to design a test structure that can be used to verify that the software will meet customer’s needs. The customer needs in this thesis are to test daily software updates and notify changes in the behaviour. This will be used to verify that the next software update released to customers has the same behaviour as the current software.
2.5.1 What is a Bug

A software bug is an event in the code that triggers an unexpected behaviour [25]. It could, for instance, be a faulty operation or a logical error. Bugs are commonly found in the source code of applications or, for example, as timing errors in I/O operations. Bugs are important to find since they pose a threat to the operation of the robot and may change the operational functionality of the robot. A common cause of bugs is when an upgrade of the controller is performed. The new version can potentially introduce new behaviours in some parts of the system that was not intended. Thus, the effects need monitoring.

2.5.2 Test Case and Test Set

A test case is used to test a specific function of software and contains the necessary information to perform an execution: steps, input values, expected results, prefix values, and postfix values. A test set is a set of test cases [24]. The test cases are developed to investigate the behaviour of the software that is under test. Defining sets of cases is a way of gathering important test cases together. One approach used in software testing is input partitioning. It is based on the operation of partitioning the test cases that has similar effects together. It is useful since one can effectively choose a test set for the selected application, and intended test.

2.5.3 Mutation Testing

Mutation testing was first mentioned by Hamlet in 1977 as a method in software testing [26]. It evaluates the quality of the test suite by observing whether the test cases can identify artificially injected bugs, i.e. mutants [27]. For instance, Haga and Suehiro explain that a function for adding two integers is tested by introducing a faulty version of the same function [28]. In the faulty version, the addition operation is substituted for multiplication which alters the operation completely, see an example in Figure 6. The performance is assessed by counting the number of bugs that are identified, i.e killed and then compared to the total amount of added bugs. The resulting score, called Mutation Score (MS), give an indication of what the test sets can handle. The equation for finding the MS can be seen in Equation 5, where \( K_m \) is the number of killed mutants, and \( T_m \) the total amount of mutants. An MS of one means that all mutants were killed, according to the identity property e.g.

\[
MS = 1 \Leftrightarrow K_m = T_m
\]

\[
MS = \frac{K_m}{T_m}
\]
2.5.4 Software In the Loop

Software In the Loop (SIL) is used to validate complex physical systems as software components, e.g. the controller for an industrial robot [7]. An advantage with this approach is that there is no need for a hardware target on which to perform the execution. An example can be seen to the left in Figure 7.

2.5.5 Hardware In the Loop

Hardware In the Loop (HIL) is a notation for when the embedded part of a system is tested on a hardware target. It could e.g. mean that one is utilising the hardware of a controller to regulate a simulation model. For this application, the controller hardware feeds the plant model with inputs and acts upon the received simulated outputs. An example can be seen to the right in Figure 7.

2.5.6 Black Box Testing

Black box is a software testing technique used when the code of the test object is unknown [24]. The test files used as the input space to the black box are created from external descriptions. Test files are created based on the function block under tests specifications, requirements, and design. In this thesis, the controller is considered as the black box. Depending on the content of the input space, the output will vary accordingly, as visualised in Figure 8.
2.6 Regression Testing

A commonly used method is regression testing which is a method suited for systems with frequent software updates [29]. When some part of a system is changed, there may be effects in another distant part of the system. This is what regression testing try and avoid. Regression testing is required when something of the following has occurred [30]:

- The code has been altered in a way that change customer requirements.
- An update has been made to improve the functionality of the code.
- A bug from the previous code iteration has been corrected.
- An out-dated functionality is removed.

Regression testing is not shown in Figure 5, but have a maintenance role in the development process of software. By using a set of test cases, also know as a test set, several functionalities can be tested at the same time. This method has three main strategies to streamline testing; minimisation which removes unnecessary test cases, selection to only include relevant test cases, and prioritisation to execute test cases in order of relevance [31]. Tests are usually time-consuming and are preferably performed overnight, which is why streamline techniques are important to take into account.

2.6.1 Selection

In a test set, there might exist obsolete test cases that can detect similar bugs in the code [32]. By selecting the test cases most suited for the purpose, a test set can be reduced to run a more efficient regression test in terms of execution time and test coverage.

2.6.2 Minimisation

Minimisation is applied in order to reduce the cost of the regression test while keeping the test coverage [32]. This technique minimises the size of either a whole test set or a sub-set by removing redundant test cases. Test sets can be minimised by selecting test cases that can highlight changes to specific parts of the code or a more general test set that can cover all of the code.

2.6.3 Prioritisation

A technique that allows the tester to prioritise the test cases based on some test criterion, such as time constraints or specific functionalities [32]. Those test cases with the highest priority are executed at an early stage and are considered to have a higher relevance.

Figure 8: This shows how black box testing is applied. The input space show the files that can be modified to alter the output of the controller.
3 Problem Formulation

The hypothesis for this thesis is that by applying simulations to software testing regression tests can be designed to enhance the motion testing and thereby increase the bug detection in a new controller version.

The way software testing is performed at ABB today is with a test engine, Vera, and a regression test software, DynTest. Vera is the software testers tool used to create, debug, and analyse test cases. DynTest handles the continuous regression test execution and test case prioritisation. It can execute either automatically every night or manually by the testers. Test cases for the test set are selected based on parameters such as time constraints and performance history. Most of the test cases included in the test set are more related to testing of functionality than to testing of motion performance. A functionality test case can, for example, check how the robot reacts to an emergency stop. Whereas a motion performance test case includes testing of the movement precision, like attaching a pencil to the end-effector and following a trajectory. These kinds of tests are, with the current setup, manually applied to a higher degree when a new release is approaching and testing is more intense, which is about two times per year. It is sought after to include more motion performance-oriented test cases at an earlier stage of development. With the current hardware dependent software test setup, this is unavailable. Mostly due to low data feedback availability from the sensors and measurement tools in the test rigs. With customers relying on robust robot systems, performing the same task with high precision, an updated controller version must not alter the motion performance. For this reason, customers can be reluctant to perform software updates, to ensure that the current motion performance. Thus, there is a need for a method to extend the possibility to test motion performance earlier in the development process.

The proposed solution is to introduce simulations in software testing. An adaptation to a test framework utilising simulations for software testing is believed to increase the test accessibility and scalability. Simulation-based software tests would allow testers to execute test cases faster and to run parallel tests on different systems. Also, simulations can provide feedback data from the robot with high precision, without extensive use of expensive sensors. This would enable testers to perform software tests with fewer limitations and make it possible to introduce motion performance testing at an earlier stage of development.

This thesis will conduct a feasibility study analysing how the movement of the robot changes between software versions. By altering the input space for the controller and search for anomalies in the output. It is believed to be utilised as a tool to evaluate and create suitable motion performance test cases for finding bugs at an earlier stage in the controller development. It is desired to develop a test set that can be used for regression testing. The future vision of this thesis is to include this test set, executing on virtual systems, as a part of the existing DynTest.

The following research questions have been derived from the problem description and are to be answered:

- **RQ1.** How can a simulated test environment be used to develop suitable test cases to detect the velocity anomalies and motion discrepancies of industrial robots?
- **RQ2.** What is the best way to generate sensible test cases to automatically identify potential bugs in the robot controller?
- **RQ3.** Can artificial bugs be used to verify the generated test cases ability to find real bugs?
- **RQ4.** How can the design of test cases be improved by the fact that the software functionalities to be tested are robot controllers and not generic software systems?
- **RQ5.** How can a criterion be defined to know that sufficient testing has been performed?
4 Related Work

This section will describe the related works for the different parts of the thesis. The different research areas are simulation environment, trajectory tests framework, regression testing and error evaluation.

4.1 Simulation Environment

Ayed, Zouari, and Abid describe in a study on how to implement the SIL simulation technique to improve the control code [7]. The paper describes how the validation of controllers can be made during development with the aid of simulated software. The combination of SIL and Model In the Loop (MIL) was used to measure the speed and position error. This was chosen due to the lack of available hardware access. It was declared that SIL was an appropriate method to avoid the issue of hardware required solutions. The study concludes that implementation of SIL can guarantee software component verification as well as increase performance and error detection at a lower cost.

Another study has been conducted by Muresan and Pitica evaluating the reliability of SIL compared to HIL with a modelled DC motor when used with a microcontroller system [8]. It was described that the setup consisted of software and simulations, apart from the Device Under Test (DUT), which was a hardware target, in this case, the microcontroller. In the SIL approach, the hardware target was substituted with a simulation part. Both the HIL and SIL tests were set up with the same conditions regarding the motor model, controller specifications, sample time, and angular velocity set point. Step size had to be a compromise between the desired accuracy and the host PC performance. Comparison between HIL and SIL performance shows that SIL is reliable when evaluating software functionality. It also showed that the reliability was lacking for when measuring properties of hardware parameters. For instance, the processor load, by evaluating the idle task.

To sum up, the use of SIL is a proper method for software evaluation in simulation. The use of SIL was more defensible than HIL, in regards to prize and functionality. It showed, that it was reliable in the context of software functionality. It was described as an approved method when hardware access is not a possibility.

4.2 Regression Testing

In a study by Engström, Runeson, and Skoglund the adopted method for regression testing is not general for any solution but needs to be researched for the specific application [33]. Coding language and required input are two examples of parameters that need consideration before a selection of method is performed. The reason for the diversity between the methods is that the evidence of disparity between methods proved to be clear. To perform regression testing on a new updated, with an existing test set, is something that testers should do, as described by Beizer [34].

Three main methods with regression testing are described in detail by Reuter et al. in their article from 2010 [35]. It is clear that the use of regression testing has increased drastically since the 80s. In their research, several papers were included exploring new methods in regression testing. It is concluded that the method shows overall indications of maturing, but also that prioritisation still needs to be further researched.

An article by Strandberg et al. [29] propose a method for increasing the efficiency in regression testing. According to the study, the main problem with most regression testing is time consuming. Thus, the article investigates a method for prioritising the order of testing cases, depending on previous performances. The different failures or affecting attributes are related to corresponding weights, which give some cases higher priority depending on recent errors.

In an article by Biggs, it is described how regression testing is an important tool for evaluating a system after a feature change or repair [36]. Some examples of attributes for evaluation that can be performed on software is the functionality and efficiency [37]. These attributes are also examples of software attributes that are necessary to remain unaltered after an update. When tests are done in a hardware interactive environment, it is important to perform the test multiple times and then apply statistical analysis. The reason is due to the uncertainty that some errors or
behaviours only might occur in a specific test iteration. In a large number of test runs, the chance of identifying patterns increase.

4.2.1 Selection

Regression testing is an effective method to reduce the number of faults in software development but comes at a high cost [38]. Selection is an important aspect in order to reduce the number of test cases in the suit. There exist several selection techniques, a few of them is presented in a paper by Ngah et al. [32]. Depending on the area in which Regression Test Selection (RTS) is performed, different techniques are suggested. When evaluating the behaviour of a code block the strategy is to create a component model based regression test. In a survey produced by Biswas et al. several different RTS techniques are explained and in which area-specific techniques are preferably applied [39]. Specification-Based RTS techniques are explained to fit for industrial testing applications when no access to the source code of the test object exists. Implementing a specification-based RTS three methods are mentioned: Activity diagram-based selection, requirement coverage matrix-based, and critical evaluation.

4.2.2 Prioritisation

Marijan et al. have proposed a method, called ROCKET, to prioritise a test set [40]. The method prioritises a test set based on the execution time for test cases fault rate over a certain threshold. This method is automated and when implemented in the industry it proved to increase efficiency and fault detection, compared to manually and randomly created test sets.

In a study by Rothermel et al. nine different prioritisation techniques are implemented and evaluated with the goal to increase the fault detection [41]. The conclusion of the tests is that an ordered optimised test set get higher fault detection but with low coverage. Six of the techniques are focused to cover faults based on code statements, branches, and probabilities which perform quite evenly with a wider coverage but not as high detection rate as the optimised test set. The test sets that was random and unorganised performed the worst.

To sum up, the use of regression testing is vital with the introduction of software updates. The order of the test cases is an important consideration as well. Another aspect that could be of importance with testing is that hardware interacted testing is performed in multiple iterations, to give statistical reliability.

4.3 Test Case Generation

A study by Wang et al. describes how to design experiments to map the TCP accuracy for an industrial robot [42]. The experiments were conducted to find optimal work areas for the robot. To find the optimal orientation in specific positions, the robot performed an experiment that follows straight lines with different joint settings in a 2D environment with defined accuracy settings. It was shown that joint error affected the end-effector error greatly. Simulated experiments were then placed in several locations within reach of the robot. The trajectory consisted of a zig-zag path in a cube. 45 cubes were placed in the workable area with even spacing. In addition, the test was repeated with different angle approaches. Furthermore, each experiment resulted in a defined accuracy, which was recorded for that position. It was concluded that the developed methodology for accuracy mapping could be further validated in simulation, as an example. It could also be used as a joint error estimator since that was one affecting a parameter that was found in the tests of the robot.

4.3.1 Test Case Generation Evaluation

Saha and Kanewala have in a paper tested how well four different test case generation strategies perform in terms of fault detection coverage [43]. The different strategies where, line coverage, branch coverage, weak mutation, and random testing. Test cases were generated using the EvoSuite\(^4\) tool which allows creating test cases for higher code coverage. The techniques were tested

\(^4\)http://www.evosuite.org/
on 77 functions found in three open-source projects, which with µJava\textsuperscript{5} mutation tool produced 7446 code mutations. The result showed that weak mutation found most mutants on its own, and could be improved by combining it with line and branch coverage.

To evaluate how well a generated test case performs can be evaluated with a fitness function. Papadakis and Malevris have in a paper conducted two experiments, one static and one dynamic to evaluate the mutant-killing effectiveness of three different fitness functions and random testing \cite{44}. From eight program units with a varying number of code rows, 2759 mutants were created. After 150000 fitness evaluations, a benchmark for the proposed fitness utilisation can be reached to kill a high number of mutants. The framework described in this paper concludes that the use of fitness functions can improve the generation of mutation based test cases to kill almost all mutants.

In summary, the articles demonstrate different approaches. One explains how generated trajectories are comparable according to shape, whereas one evaluate the accuracy depending on a specific trajectory, and where it is positioned in the available space for the industrial robot.

### 4.4 Similarity Analysis

With big architectures, the topology is often built by a sizeable amount of smaller sub-systems. An article by Baumann, Pfitzinger, and Jestädt propose a method of integration between them with the aid of simulations \cite{45}. The article describes how a huge amount of vehicles undergo regular system updates of software where the Simulation Driven Development (SDD) method is used as a reliable justification for design updates. As described, the method is expected to provide a more fundamental understanding of the system and its architecture. The expectation is that the number of errors generated will decrease due to the heightened system awareness.

An article was written by Roduit, Martinoli, and Jacot compare different controllers on a single mobile robot \cite{46}. The authors conclude experiments to generate test data. The different executions generate different trajectories depending on iteration and controller. The trajectories are evaluated by introducing them to the statistical approach Point Distribution Model (PDM). The controllers were then compared with regard to the results from the PDM process. The article concluded that the statistical method can be utilised with a comparison between real experiments and their simulated counterpart.

The knowledge of how small errors affect the position of the end-effector could be useful when analysing precision. This has been done by systematically apply a small error to each joint individually. This way it can be monitored how it affects the position of the end-effector, this method is described by Wang et al. \cite{42}. The way it is done is by keeping all joints fixed and then move them one at a time $0.01^\circ$ and measure how the position of the end-effector is changed. This method could then identify which joint that most sensitive to an error.

To measure the trajectory differences between two executions is something that has been useful when finding the optimal trajectory. Seok et al. evaluate a new path planning algorithm for auto-generated trajectories \cite{47}. As a comparison measurement, the Hausdorff metric is used as a part of a fitness equation to find the best auto-generated trajectory. The experiment was successful and proved to work out better than the compared method.

Chen et al. describe how Hausdorff distances were utilised to cluster trajectories\cite{48}. The data that was processed was hurricane data. The paper suggested that the trajectories could be clustered, not only according to geometrical representation but also according to the direction. The approach clustered the trajectories into five representative trajectories.

Another method for measuring the similarity between two trajectories is the Fréchet distance. In a paper published by Holladay et al. the Fréchet distance is used as a measurement to evaluate the distance between two trajectories \cite{49}. The objective solved by implementing the Fréchet distance algorithm is to find the optimal trajectory for a robot arm compared to a reference trajectory. The results were promising and resulted in a final trajectory that converges towards the reference.

In conclusion, the aid of simulations can provide an early justification for certain design decisions. Furthermore, the error registered in the different joint will vary. Methods were described to find which joint that provides the highest distribution to an error.

---

\textsuperscript{5}\url{https://cs.gmu.edu/~offutt/mujava/
4.5 Bug Detection

A method used in conjunction with regression testing is mutation analysis for fault detection. Sun et al. propose a new technique that increased the efficiency of regression testing [50]. The method begins with creating a subset of mutants then through iterations a set of test cases is created. Every time there exists live mutants in the subset another test case is added to the set until all mutants are killed and after that, a new subset of mutants are chosen. This would eventually result in an ordered and reduced set of test cases. The mutation was made with a mutation tool for C-code called Proteum [51].

An empirical study by Papadakis et al. raises the question of how MS affects the detection of real faults [52]. This study applies state-of-the-art test generation tools to create 10,000 test suites of random sizes from large real-world C and Java programs. Regression analysis is used to point out that both test size and MS play a role in fault detection. The correlation between MS and fault detection was done with the Pearson, Kendall correlation coefficients. It is concluded that fault detection is influenced by the test set size and MS. Also, fault detection increases for test suits with a high MS, but can not provide a good representation of the real fault detection.

In a study by Haga and Suehiro, a method is described, where random test cases are processed with a Genetic Algorithm (GA), and evaluated with the MS [28]. The generated mutations include changes in the source code, such as logical changes or differences in operations. The method showed that refining the cases with GA proved to have higher rates of killing mutants, e.g., higher MS.

In a paper written by Ciabattoni et al. three different statistical methods are evaluated, Linear Regression (LR), Switching Linear Regression (SLR), and Partial Least Square (PLS), detecting faults of nonlinear processes [53]. The methods were evaluated with three case studies and through all three the best practice method was the SLR. The SLR uses a clustered estimation of several LR models. The clustering algorithm used for the SLR was the Gaussian Mixture Model (GMM) and find the optimal number of clusters with the Bayesian Information Criterion (BIC).

In conclusion, the articles take advantage of simulated environments, as well as thorough testing, to justify changes or updates. In addition, testing accuracy can be performed to show where a robot can perform with the highest precision.
5 Method

This section will describe which methodology model this thesis follows, and the method developed to solve the problem and to answer the previously stated research questions. Figure 9 is a flow chart over the developed method used for bug detection.

For this thesis it was out of scope to implement a machine learning algorithm, due to insufficient access to training data and documentation about old bugs. Therefore it would be time-consuming and difficult to set up a machine learning solution.

![Flow chart](image)

Figure 9: This shows a flow chart of the different parts of the regression test. It is based on to the proposed method and how each part contribute to eventually verify if a bug exists in a new update.

5.1 Research Methodology

The thesis aims to construct and conduct an empirical study. This approach will provide the ability to test and analyse the effect of an implemented solution for a problem. The workflow of an empirical study is constructed in a cycle [54], as can be seen in Figure 10. Each state of the cycle has a certain purpose:

- **Observation**: The phase where the researchers can make initial observations. For instance, they can be of an environment or object to investigate.

- **Induction**: The researchers define the study. Write out an initial hypothesis, and some research questions to be answered.

- **Deduction**: In this phase, the approaches for answering the questions are defined. For instance, deciding which tests that are to used for the testing phase.

- **Testing**: In this phase, the results and data is produced and gathered.

- **Evaluation**: Analysis of the data is performed, to try and answer the hypothesis and the research questions.

From the investigation of the related work, the papers that acted as scientific contributions for this thesis are listed in Table 2.

Table 2: The articles that have provided inspiration for the method proposed in this thesis.

<table>
<thead>
<tr>
<th>Research topic</th>
<th>Cite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation Environment</td>
<td>[7], [8]</td>
</tr>
<tr>
<td>Regression Testing</td>
<td>[29], [33], [35], [40], [50]</td>
</tr>
<tr>
<td>Test Case Generation</td>
<td>[28], [43]</td>
</tr>
<tr>
<td>Mutation</td>
<td>[28], [43], [44], [50]</td>
</tr>
<tr>
<td>Similarity Analysis</td>
<td>[42], [47]</td>
</tr>
</tbody>
</table>


---
5.2 Simulation Environment

The simulation environment is created using a SIL approach, meaning that the whole execution is happening on the same PC. The robot controller is a Virtual Controller (VC) and the robot is represented with a simulation model [8]. A simulation model of the robot is provided from the ABB-Library. To give the simulation model a behaviour more true to reality a simulation software, Algoryx\(^7\), will be utilised. Algoryx will provide physical properties to the robot and a graphic client of the simulation. This setup will enable extraction of data in the form of both VC- and VCSIM-signals. In Figure 11 is a flowchart describing how the simulation and signal extraction is connected.

The robot model that will be utilised is the IRB120 and it will be used to run all tests. A simulation model of the IRB120 robot is loaded to the Algoryx simulation environment, see Figure 12. Algoryx will provide gravitational forces to the robot and make simulations more true to reality and also show a graphic client of the simulation run.

The simulation is an additional functionality that is applied to the existing controller system, enabling extensive access to sensors with high precision in the measurements based on the robots simulated movements.

5.2.1 Signal Extraction

Data from the simulation are extracted by enabling VCSIM-signals. These signals log the robots simulated movement behaviour based on the Algoryx simulation. Which signals to monitor are defined in a text file located in the VC installation folder. Though, the selected signals have to be activated and deactivated in the test case RAPID script. The VCSIM-signals used in this thesis are:

\(^7\)https://www.algoryx.se/
Figure 12: This is the graphic client that Algoryx providing a visualisation of how the robot moves when executing a test case in the simulation environment.

- Motor Position - Motor position of all six axes of the robot.
- Measured TCP - X, Y, and Z position of the TCP.
- Reference TCP - Reference X, Y, and Z of the TCP position.
- Quaternions - The quaternions that represent the orientation of the TCP.

VC-signals are defined in the RAPID script and can sample up to 16 different signals. These signal types correspond to the VCSIM-signals: reference TCP, measured TCP, and motor positions.

5.3 Test Case

A test case is defined as a trajectory and will be written in RAPID. It can be modified in infinite ways using different paths, speed, and zone precision. Utilising this, test cases will be auto-generated making it possible to create large test sets. Besides auto-generated test cases more static test cases will be added using the Mercedes path. These will be manually placed in different locations and orientations within the robot's workspace. Mixing different types of test cases is thought to increase the diversity among the test cases and to verify which type of test case is preferred. These test cases will be used as one of the inputs to the robot controller. It is desired to create a diverse test set, manipulating the robot to various positions. To create a diverse test set is mainly to find bugs that can only be detected when the robot is in a specific position.

The test set will include different test cases, which are RAPID programs. In the RAPID program is the trajectory defined as well as the tool and its properties. The test set includes a diversity of programs to increase the chance of finding a bug. It will be described how the test cases for the test set has been defined and implemented.

5.3.1 Auto Generation

A system for automatically generating RAPID program files has been developed in Matlab\(^8\). The restrictions to the test case generation can be found in Table 3. This system can generate three types of test cases referred to as:

1. Three Areas
2. Line

\(^8\)https://se.mathworks.com/products/matlab.html
3. Rotation

The *Three Areas* type is defined by generating a random amount of targets within the interval of 1 and 30 targets, and move linearly between them. More than 30 targets would exceed the amount of time available for sampling data. The target positions are decided under constraints, so that no target is located where the robot is unable to reach. After the creation of the targets, each target is rotated around one of its three axes. The amount of rotation, in degrees, is set to be a random value between 0 and 25 degrees. Too much rotation will trigger built-in safety functions that restrict the execution of the test case. Thus, 25 degrees was chosen, which was considered the highest value possible, while being able to execute the cases without triggering errors. Three main areas have been defined where targets can be generated. Which can be seen in Figure 13 with one area in front of the robot, and one area on each side. Three areas were defined due to the fact that the area of operation is limited. One large area would require more control in how the movement would be planned, to not plan a movement over an unreachable area. Therefore, movement within the defined areas can be done freely. However, when moving from one area to another, an absolute joint instruction is used instead, to prevent unfeasible paths. It is based on joint positions in the target to reach, and will only move the robot in the operational area.

The *Line* type of test cases will generate line-shaped trajectories in front of the robot. The endpoints of the lines are randomised, but will be placed with equal spacing from the Y-axis. The cases were developed to move linearly with constant velocity from one target to another. However, due to the position of the trajectory, and the limitations of the robot in regards of movement, the velocity will decrease at the middle of the line. In addition, the targets are randomly rotated in a similar fashion as with the three areas type. However, the rotation amount is slightly smaller. The rotation vary between 0 and 15 degrees.

Finally, the *Rotation* type was created. These test cases were designed to create one specific target in the space, and then rotating randomly around that point. These cases were developed to put more stress on the links and motors. These cases will not focus on TCP movement since the point is stationary. These cases possess higher rotational possibilities, since that is the main function of the test set. The possible rotation values lies between 0 and 35 degrees. However, the area that the target is able to be generated in is more restricted.

Table 3: These are the values used for defining the parameters used when generating test cases. The parameters are for the TCP position(mm), the velocity(mm/s), and the zone sizes(mm).

<table>
<thead>
<tr>
<th>Test Case Type</th>
<th>X Min</th>
<th>X Max</th>
<th>Y Min</th>
<th>Y Max</th>
<th>Z Min</th>
<th>Z Max</th>
<th>Velocity Min</th>
<th>Velocity Max</th>
<th>Zone Min</th>
<th>Zone Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three Areas - Front</td>
<td>300</td>
<td>425</td>
<td>-180</td>
<td>180</td>
<td>250</td>
<td>400</td>
<td>1000</td>
<td>2000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Three Areas - Left</td>
<td>-100</td>
<td>125</td>
<td>370</td>
<td>420</td>
<td>300</td>
<td>450</td>
<td>1000</td>
<td>2000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Three Areas - Right</td>
<td>-100</td>
<td>125</td>
<td>-420</td>
<td>-370</td>
<td>300</td>
<td>450</td>
<td>1000</td>
<td>2000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Line</td>
<td>180</td>
<td>220</td>
<td>-200</td>
<td>200</td>
<td>370</td>
<td>520</td>
<td>1000</td>
<td>2000</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Rotation</td>
<td>300</td>
<td>390</td>
<td>-150</td>
<td>150</td>
<td>350</td>
<td>390</td>
<td>1000</td>
<td>2000</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

The generated move instructions are generated with randomly chosen velocities and zone sizes, within defined intervals. For instance, the movements can not be performed under 1000mm/s or over 2000mm/s. Whereas the zone sizes need to be between 0mm and 2mm. Each target belong to one of the three areas. When generating the targets, it is done sequentially. It was early observed that the move instructions was very frequently moving between areas. Therefore, it was implemented that new targets would have a 15% probability to be generated in a different area than the current one.

The RAPID files are generated in such a fashion that they are executable right after creation. No additional user interference is required. This is to minimise the time needed to load new programs to the controller.

5.3.2 Mercedes Path

Ten manually created versions of the Mercedes path has been created. Manipulating the robot in different locations and orientations. No lengths, speeds, or zones has been altered from the default
Mercedes path program. The parameters are set to 1000mm/s and the zone precision set to one, for all move instructions.

5.4 Creation of Artificial Bugs

Without access to the controller source code, it is not possible to inject bugs in the controller code. Information has been provided by the experts at ABB that ”bug-like” behaviour can be achieved by making changes in the controller’s MOC-file. This might make it possible to have the robot behave like there is a bug in the system, even though its not an actual bug in the controller code. A list with suggestions on MOC modifications has been provided by ABB. These modifications will at first be used to create an experiment to verify that these artificial bugs can be detected. Furthermore, creating modifications to the MOC-file would work similarly to mutation testing [26]. If proven successful it can be used as a substitute to real bugs. Thereby be used to help detect the test cases with a higher bug detection ability, as described in the mutation paper written by Papadakis and Malevris in [44].

Two MOC categories will be modified: MOTION PLANNER and SPLINE PARAMETERS. Changes made in MOTION PLANNER changes how the robot calculate movements, and changes made to SPLINE PARAMETERS handle how many points to calculate in advance by the motion planner [55].

5.5 Similarity Analysis

It has been noted that the data points in the VCSIM-signals are not equally spaced. The logging process begin before the robot initiate its movements. The results of this behaviour originate from the fact that the initial data points are clustered, before representing the path. This is one of the reasons why the Euclidean distance method is not suitable for the given application. The distance will not represent the actual distance since there is an offset distance which can not be altered properly.

For this reason, the Hausdorff distance method is applied to perform this task. By analysing the output of a test case it can evaluate the execution according to an expected output. This can be achieved by comparing the output data with a reference execution. This evaluation will determine its similarity.

The comparison between output files are made between files that contain the same data type. For example, a VC-signal with TCP data is only compared with another VC-signal containing TCP data.
When the output to analyse consist of six dimensions, each one is for one motor in the robot. The Euclidean distance in a six dimensional space is denoted according to

\[ d(A, B) = \| B - A \| = \sqrt{(a_2 - a_1)^2 + (b_2 - b_1)^2 + (c_2 - c_1)^2 + (d_2 - d_1)^2 + (e_2 - e_1)^2 + (f_2 - f_1)^2} \]

where \( d(A, B) \) is the distance between sets \( A = a_1, b_1, c_1, d_1, e_1, f_1 \) and \( B = a_2, b_2, c_2, d_2, e_2, f_2 \).

Firstly, the Hausdorff distance was used in Matlab, to evaluate its performance. The function already existed, and was added to the Matlab workspace from MathWorks\(^9\). With the transition to the test engine, the function needed additional implementation in C# through Visual Studio. However, the performance of the Hausdorff function in Matlab overcame the one in C#. The execution time in C# was too long i.e. approximately 10 minutes for one specific test case. For that reason, all similarity analysis is performed in Matlab.

The data for each test case and configuration, artificial bug, was analysed against the corresponding reference execution. For instance, the output from a reference execution will be evaluated with that execution for all configuration executions from the target controller. This is visualised in Figure 14. Each configuration is an artificial bug, alternating of the robots move parameters. The Hausdorff values for each configuration showed which configurations and test cases that can give a diverse output.

![Figure 14: This work flow describe the process of sampling data and processing it, to ultimately receive an intuition of how each individual test case can perform. It is the process of performing similarity analysis, creating a requirement analysis matrix.](https://se.mathworks.com/matlabcentral/fileexchange/26738-hausdorff-distance)

### 5.6 Test Set Reduction

The outline of this thesis is to create a motion performance test set. That can be used for regression testing to detect motion related bugs earlier in the development of the controller software. Regression testing is a costly and time consuming method to execute [29], [35]. The cost comes when a large active test set contain one or more test cases that check the same behaviour. A test set can with the use of three efficiency techniques be trimmed down and still remain the same test coverage but with fewer test cases [56]. With access to a requirement analysis matrix the initial test set can be reduced. Reduction of a test set will allow the regression test to execute faster without decreasing the test coverage. In this section it will be defined how a test set is evaluated and how the test set was reduced to make the regression test more efficient.

#### 5.6.1 Test Set Evaluation

The test set was evaluated based on the output data after being executed on two different controller versions. First the test set was executed on an unmodified controller version, creating test set reference data. Then, the same procedure was performed but executing the test set on a modified version.

---

\(^9\) https://se.mathworks.com/matlabcentral/fileexchange/26738-hausdorff-distance
controller, with an artificial bug active. With these two output sets of data a similarity analysis algorithm creates a requirement analysis matrix, holding the Hausdorff distances of all test cases in the set [39].

### 5.6.2 Selection

Test cases are selected based on the likelihood of detecting a bug. The likelihood is based on the Hausdorff distance measurement between two executions of the same test case. If the Hausdorff distance measurement is high the test case is considered to be more likely to find anomalies, or bugs, based on the requirement analysis matrix with the collected test set data from the controller executions. Selection can create a specific sub-set from a test set to focus the search for bugs based on configuration impact on the controller [39]. This will be applied to the extent that the best performing test cases, those with the highest Hausdorff value, are selected. One test case is selected per signal type and configuration, denoted artificial bugs. So, if three signals have been monitored in the simulation the best test case from each signal type is selected, meaning that three test cases will be selected from each configuration. This way a selected test set is created containing the best performing test cases from each configuration, see the first evolution in Figure 15. This suggested approach is based on selecting the optimal test cases and is commonly compared to selecting test cases randomly [40], [41].

### 5.6.3 Prioritisation

This should be used to in future regression tests to prioritise test cases that has a history of finding bugs [40]. This have to be a careful procedure to make sure that there still exists test cases so all of the bugs contribution are included in the compressed sub-set.

### 5.6.4 Minimisation

Minimisation is a common regression test technique used to remove, what is considered to be, redundant test cases from the test set [31]. This can be done using the selected Hausdorff performance matrix, selected test set, to create a sub-set by removing obsolete test cases, in the last step in Figure 15 is an example of how an obsolete test case is removed [57].

<table>
<thead>
<tr>
<th>INITIAL TEST SET</th>
<th>SELECTED TEST SET</th>
<th>SUB-SET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gf1</td>
<td>Gf2</td>
</tr>
<tr>
<td>TC1</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>TC2</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>TC3</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>TC4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>TC5</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 15: This describes how selection, prioritisation, and minimisation works. Starting with an initial test set a performance matrix is created, which in this example rate test cases between [1 10]. A selected test set is extracted when selecting those test cases with the highest performance value for each configuration. In this example the top two test cases are selected. Test cases are prioritised based on a performance value threshold, which in this example is prioritise those test cases with a performance value greater than or equal to 5. The last step is to remove redundant test cases, in this example $TC_4$ has been out prioritised form the sub-set.

### 5.6.5 Test Set Validation Model

A reduced test set was validated against a controller version with a real bug. The output was ran through the similarity analysis algorithm, comparing the output from the controller version with a bug and the controller version that was used to create the test set, see Figure 16. This model was used to determine if a test set can detect real bugs. Also, to determine if the reduced test set could provide the same coverage as the initial test set.
Figure 16: How a test set is validated. The test set is validated by executing the test set on a reference controller and a controller with a known bug. The output files are analysed to validate if the controller with bugs can be distinguished.
6 Implementation

This section describes the required implementation for the developed method. The process of including the simulations is explained, as well as the introduction of the test engine Vera.

6.1 Simulated Environment

With the initialisation process of the VC, the necessary simulation files are loaded to the controller. These files are defining the simulations and define where the simulation installation is located. Apart from defining the simulation environment, these files describe the properties of the simulation model. For instance, it can describe the lengths, weights, and forces to simulate correct physical behaviour.

Additionally, some modifications to the test cases that are needed to activate sensors in the simulation. The function of initialising the data logging process had to be defined. Furthermore, in the last stages of a test case, it had to be marked where the test was to cease the logging process.

The final action to perform to be able to utilise the simulations was to load a set of parameters with the start-up of the controller. This gives the VC all the information it needs to properly start the simulations at the execution of a test case. Three files was created to perform this task:

- **startVC.exe** - This file takes a string of the path to the controller as input, and then start the VC.
- **VCstart.cmd** - This file loads the Algoryx parameters, and then execute ”startVC.exe” and pass the path to the controller as an input argument.
- **run.vbs** - This file starts ”VCstart.cmd” and passes the path to the controller as an input argument. The intention of this file is to disable that ”startVC.cmd” open a terminal window, since it can interfere with the user, and in worst case generate issues with the overall execution of the test set.

6.2 Implementations to the Test Engine

The regression test is built in Visual Studio using the in-house developed test engine. The test engine is the software testers tool when working with software testing. It enables testers to create automated regression tests. The test engine handles communication between the test cases, VC, and robot providing a pass or fail indication on each test case to the test engineers. This way a regression test can be automated and executed in an efficient manner. The test engine enables the possibility to automatically install specific robot controller versions to execute the test suit with. All test cases are executed with the Algoryx simulation active and save both VC- and VCSIM-data.

First, the test set is executed with the unaltered, reference, controller version. The main function of this test will execute all test cases located in the program folder. Each test case is executed, and the output data is moved from the virtual controller’s installation folder to a remote local folder, to be used for further analysis. The main loop can be seen in Algorithm 1.

**Algorithm 1** Reference execution - main loop

```
procedure Reference
    for each Testcase do
        Execute Testcase
        Download output data to local folder
```

Secondly, the test set is executed with the altered, target, controller version. This execution is slightly different, all test cases are executed on a controller with a modified configuration file, artificial bug. The number of output files varies depending on the number of configuration files and test cases that have been used. When a configuration change is made a system restart is necessary otherwise the configuration will not be active. The main loop for this execution can be seen in Algorithm 2.
Algorithm 2 Target execution - main loop

procedure Target
    for each Configuration do
        for each Testcase do
            Load Configuration
            Restart controller
            Execute Testcase
            Download output data to local folder
        end for each Testcase
    end for each Configuration
end procedure

Finally, when the data has been collected from the reference and the target controller, the data is due for analysis. A command script is called from Visual Studio launching a Matlab script analysing the test set data.

The Matlab analysis script will perform a test set reduction that will be used for validation. The reduced test set is validated against a controller version with a real bug. After execution similarity analysis is performed once again, in Matlab, providing performance feedback on the reduced test set. This operation is implemented in Visual Studio. The main algorithm for this procedure can be seen in Algorithm 3. The complete workflow can be seen in Figure 17.

Algorithm 3 Validation execution - main loop

procedure Validation
    for each Selected Test Set do
        for each Testcase do
            Execute Testcase
            Download output data to local folder
        end for each Testcase
    end for each Selected Test Set
end procedure

Figure 17: This is the workflow describing where the different blocks have been implemented, beginning in the test cases generation block. The crossing between platforms are made four times, and the reason is that the analysis prototype was developed in Matlab.
7 Experimental Setup

This section will describe the different experiments that have been created for validation and verification of the methods. These experiments have different approaches. Some are to investigate how the simulations and controllers should be utilised. Others are targeted for the creation and execution of test cases.

7.1 Simulation Deviations

It has been problematic to replicate test results. Sometimes the output between two simulations can differ even though the inputs are exactly the same. This problem occurs when a trajectory is being executed repeatably. An experiment has been conducted to verify how the output varies between executions. By creating a RAPID program that loops the same trajectory, a conclusion about how the output changes over iterations can be drawn.

The test is designed by inserting the function call for the trajectory in a loop and defining the loop amount to 500 iterations. The logging is performed for each iteration. Thus, 500 log files are generated. The evaluation will analyse the Hausdorff distance for the TCP positions, against the TCP reference positions.

The expectation is to either see a pattern in the distances or that the values are varying in a small closed interval. The latter would then imply that there is no great deviation in position if the trajectory execution is simply looped. This test was developed to:

- Evaluate the simulation, to find if any affecting behaviours exist.
- Find if the design of a test case can affect the operation in simulation

7.2 Controller restart

In various occasions, the controller needs to perform a restart. For instance, if new configuration files have been added, a restart will ensure they are properly loaded. However, the behaviour of the simulation behave differently depending on in what manner the controller is restarted. Thus, an experiment is conducted to find how different approaches will alter the simulation results.

The test is designed to execute a RAPID program twice and analyse the similarity of the output. Between executions, the program will be reloaded in different ways, and the controller will be restarted in different ways.

The first test will investigate how the execution will be altered if the program is simply reactivated. That implies that it is not re-loaded, and the controller is not re-started. In the second test, the program will be loaded again to the controller right after execution. No restart of the controller is performed. In the third one, the controller will perform an ordinary restart between the executions, referred to as a warm start. Since the program will initially be loaded after the warm start, it will only need to be re-activated. Finally, in the fourth test, the controller will undergo a restart referred to as an I-start. This procedure is equivalent to stopping the controller entirely and starting it again. The program will not initially be loaded after that procedure, and need to be loaded again.

The Hausdorff distance is to evaluate how the secondary execution will deviate from the first, over the different approaches. It is desired that there is no difference between iterations since the program execution is the same. The expected outcome is that at least one of the methods can provide that behaviour. That would imply that there is a way to replicate an execution and that there is no uncertainty in how to restart the controller in an appropriate way. This test will investigate:

- How the controller manages restarts, in regard to maintaining files and structures from previous executions.
- How a method for operating the VC can be developed to avoid misleading results.
7.3 Verification of Test Case Creation

This section will describe the tests defined to evaluate the selection of test cases. The different methods will be tested, as well as the selection of test cases with different structures. In addition, the tests will evaluate how the performance of test cases can be improved by adding certain versions or modifications. For instance, evaluate the performance after the addition of rotation to the targets. The version used for these tests is the 6.09.0126.1 controller.

These tests were designed to draw conclusions in test case structure, to answer RQ1, RQ2, and RQ4. The contributions from these tests are targeted to

- Provide intuition in performance of generated test cases.
- Find how the introduction of rotation affects the output of the black box testing.
- Evaluate how test cases perform, and how they differ between types
- Conclude how the design of test cases can be improved.

7.3.1 Target Rotation Experiment

A test was defined to find if the introduction of slight rotation to the targets would enhance the functionalities of the targets. The intuition is that the rotations will increase the movement of the joints closest to the TCP e.g. 4, 5, and 6. Thus, the "wrist" of the robot will move more, and then increase the efficiency of the test cases. It is wanted that all joints are moved, to not exclude movements that could potentially prove vital for the bug detection ability.

The test was designed to execute 200 test cases, 100 three area and 100 line test case types. The test set will be run with and without a controller bug, as well as with and without rotations included. The expectation is that a clear realisation can be seen, that one execution perform better than the other. The quantifying value will be produced by evaluating the Hausdorff distance as described in Section 5.5.

7.3.2 Test Case Generation Type Experiment

Since there exist multiple types of test cases, a test will be defined to investigate whether there exist any significant differences between them. For instance, one type may be more adequate in a certain application than another. In addition, they operate in slightly different operating areas, which also may be an affecting factor. Another aspect is the fact that each test case generates several data types. The diversity between data types and test case types was investigated.

The test is designed to generate 300 test cases in total, 100 of each type. The test cases are to be executed on a reference controller, and then the same controller with a bug induced. Then the analysis will generate the Hausdorff distances for all the test cases. The distances are to be evaluated against each other, to see which set of cases that provide a higher value on each data type.

7.4 Test Set Reduction and Real Bug Detection

This experiment is designed to verify if artificial bugs can be used to create a compressed sub-set for real bug detection. The idea is to find similarities, in the VC- and VCSIM-signals, between artificial and real bugs. This experiment was evaluated by creating a sub-set from the artificial bug data, and validate it against a real bug.

The first step is to verify that artificial bugs, in the form of modified configuration files, can be detected. The second step will do the same but with a real bug. To make sure that all simulations are executed with the same prerequisites the same controller version, 6.09.0126.1, is used and the controller is restarted between every test case execution. The same test set is used for both experiments containing 160 test cases divided as follows: 50 three-area, 50 rotation, 50 line, and 10 Mercedes track trajectories.

The goals of this experiment were to:
• Find if modified configuration files can be used as artificial bugs to select a sensible sub-set able to detect future bugs.

• Find out if artificial and/or real bugs can be detected with the proposed method.

• Verify if it is necessary to monitor both VC- and VCSIM-signal when identifying real bugs in the controller.

7.4.1 Artificial Bug Detection

This experiment is conducted to verify that modifications in the configuration files can be detected using the proposed method. The modifications to the configuration file are made by tuning some of the parameters. Examples on how and which parameters are tuned have been provided by ABB. For this experiment, 12 manually modified configurations files have been created.

7.4.2 Real Bug Detection

ABB has provided a bug that is said to be a bug that changes the behaviour of the robot. It is unknown how the bug affects the controller and, thereby, the robots movement behaviour. The bug activated by manually swapping a file in the controller installation folder. This experiment will be conducted to verify if the proposed method can detect a real bug, and also which type of test case that is preferred.

7.4.3 Compressed Test Set Validation

To execute a large test set is very time consuming and thereby unfeasible to execute in nightly regression tests. This experiment will evaluate how a test set could be reduced to only include those test cases proving to perform best. The test set reduction was performed based on loading 12 different MOC files to the robot controller. This means that the same test set is executed 12 times, but with a slightly modified controller. The loaded MOC files acted as active artificial bugs to the controller, or mutations of the controller. Each of the output data from the MOC runs was compared against a test set output from a non-modified controller using the Hausdorff distance algorithm creating a performance matrix. Assessing the values in the Hausdorff distance performance matrix, containing how each individual test case from the all output sets performed, are three regression test reduction techniques applied; selection, prioritisation, and minimisation. The reduced test set is then to be evaluated by executing the new test set on the controller with a real bug active.
8 Experimental Results

This section will demonstrate the data and results gathered from the tests described in Section 7.

8.1 Simulation Deviations

A test has been conducted, as described in 7.1, where the deviation of the TCP position over iterations has been recorded. The data recorded for the TCP trace has been analysed against its corresponding reference through the Hausdorff approach. In Figure 18 the resulting values are visualised. The Hausdorff distance starts at around 2.25, and increase to about 2.33. This implies an increase of about 0.076 (3.4%).

![Figure 18](VCSIM_TCP_Deviation_Test.png)

Figure 18: This scattered line demonstrate how the Hausdorff distance of the TCP data vary over loop iterations. The executed path is a circle, with constant speed. The legend demonstrate the graph value of the first and last sample.

8.2 Controller Restart

The test results for evaluating various ways of re-initialising a program to the controller is demonstrated in this section. The results for this test can be seen in Figure 19. The graph that is to the left demonstrates the results for the Mercedes track, and the right graph demonstrates the results for a test case RAPID program.

The resulting Hausdorff values are listed in Table 4. The only entries that resulted in zero were in the fourth row, which is represented by the I-Start method.

Table 4: This is the results from the different restart methods. The Hausdorff distance has been collected by evaluating the TCP positions. The two first rows show when the program has been re-introduced without restarting the controller. The last two rows show two different restart techniques.

<table>
<thead>
<tr>
<th>Restart Variant</th>
<th>Mercedes Path [HD]</th>
<th>Random Targets [HD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Play Again</td>
<td>0.01892</td>
<td>0.05525</td>
</tr>
<tr>
<td>Reload Program</td>
<td>0.01889</td>
<td>0.05525</td>
</tr>
<tr>
<td>Warm-Start</td>
<td>0.15068</td>
<td>6.5868</td>
</tr>
<tr>
<td>I-Start</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 19: This demonstrates the results from a restart test. The left scatter plot shows the data from executing the Mercedes trajectory, and the right scatter plot to show the execution of a trajectory with randomly generated targets. The test was performed four times per trajectory. Each iteration had a different approach to re-loading the same trajectory to the controller for execution. The measurement on the Y-axis is the Hausdorff distance for the TCP trace, and the X-axis shows the experiment number. The fourth execution showed to be zero for both trajectories, when the controller was restarted with an I-start.

8.3 Verification of Test Case Creation

This section will demonstrate the results from the tests analysing prospects of creating test cases. Firstly the rotation test, to see if target rotations are beneficial, and secondly a test to find properties of different test case types.

8.3.1 Target Rotation Experiment

After executing 200 test cases twice, once with rotation on the targets, and one time without, the resulting Hausdorff distances can be seen in Figure 20. The distances recorded are based on the TCP positions and motor positions.

In table 5, all the results from this test are presented.

Figure 20: The results for the rotation experiment, showing the TCP results to the left, and motor position data to the right.
Table 5: The results from the target rotation experiment.

<table>
<thead>
<tr>
<th>Test Set</th>
<th>Signal Type</th>
<th>Data Type</th>
<th>Mean [HD]</th>
<th>Std [HD]</th>
<th>Min [HD]</th>
<th>Max [HD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Rotation</td>
<td>VC</td>
<td>TCP</td>
<td>0.000132</td>
<td>0.000392</td>
<td>0</td>
<td>15.882333</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>TCP</td>
<td>0.058123</td>
<td>0.230868</td>
<td>0</td>
<td>3.115984</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.058123</td>
<td>0.230868</td>
<td>0</td>
<td>3.115984</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000132</td>
<td>0.000392</td>
<td>0</td>
<td>15.882333</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>TCP</td>
<td>0.058123</td>
<td>0.230868</td>
<td>0</td>
<td>3.115984</td>
</tr>
<tr>
<td>Without Rotation</td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000056</td>
<td>0.000515</td>
<td>0</td>
<td>0.007099</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>TCP</td>
<td>0.058123</td>
<td>0.230868</td>
<td>0</td>
<td>3.115984</td>
</tr>
</tbody>
</table>

8.3.2 Test Case Type Generation Experiment

An experiment has been conducted to try and find how the test case types are differing from each other. An amount of 100 cases of each type has been executed, where the target execution had a bug introduced to the controller.

The generated results are visualised in Figure 21. The first 100 cases are the three areas type, the next 100 is the line cases, and the 100 final cases demonstrate the rotation type. The representing results for this test are presented in Table 6.

![Figure 21: The Hausdorff measurements for the TCP and motor positions from the type experiment. To the left is the TCP results, and the motor position data is to the right. The VC-signals are displayed in the top, and the VCSIM-signals in the bottom graphs.](image)

8.4 Test Set Reduction and Bug Detection

Here are the results from three experiments designed to verify that the proposed method is able to detect abnormal behaviours. The first experiment shows the results from how a controller reacts to 12 modified MOC-files. The second experiment shows how a real bug is affecting the controller. In the third and last experiment is an evaluation of two reduced test sets, one with the best performing test cases and the other with randomly selected test cases.

All experiments have been executed on the 6.09.0126 controller version and with the same initial test set. The initial test set used have 160 test cases and consists of four different test case types; Three-area [1-50], Rot [51-100], Line [101-150], and Mercedes Track [151-160].
Table 6: The results from the type experiment.

<table>
<thead>
<tr>
<th>Test Case Type</th>
<th>Signal Type</th>
<th>Data Type</th>
<th>Mean [HD]</th>
<th>Std [HD]</th>
<th>Min [HD]</th>
<th>Max [HD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three Areas</td>
<td>VC</td>
<td>TCP</td>
<td>0.000164</td>
<td>0.000528</td>
<td>0</td>
<td>0.002686</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.000638</td>
<td>0.000851</td>
<td>0</td>
<td>0.000384</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.014370</td>
<td>0.037534</td>
<td>0</td>
<td>0.186180</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.000938</td>
<td>0.002476</td>
<td>0</td>
<td>0.020625</td>
</tr>
<tr>
<td>Line</td>
<td>VC</td>
<td>TCP</td>
<td>0.000224</td>
<td>0.000534</td>
<td>0</td>
<td>0.005312</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.018728</td>
<td>0.137349</td>
<td>0</td>
<td>1.056074</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.002837</td>
<td>0.028064</td>
<td>0</td>
<td>0.280665</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.043823</td>
<td>0.062840</td>
<td>0</td>
<td>0.411416</td>
</tr>
<tr>
<td>Rotation</td>
<td>VC</td>
<td>TCP</td>
<td>0.043823</td>
<td>0.062840</td>
<td>0.000053</td>
<td>0.411416</td>
</tr>
</tbody>
</table>

Table 7: A demonstration of the time variations between test case types. The values have been extracted from the test set from the type experiment i.e. 100 cases of each type.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Three Area</td>
<td>10.952605</td>
<td>6.275826</td>
<td>1.165248</td>
<td>21.547008</td>
<td>1095.260544</td>
</tr>
<tr>
<td>Line</td>
<td>10.777818</td>
<td>5.344980</td>
<td>1.145088</td>
<td>20.188224</td>
<td>1077.781824</td>
</tr>
<tr>
<td>Rot</td>
<td>6.306653</td>
<td>3.272559</td>
<td>1.314432</td>
<td>12.341952</td>
<td>630.665280</td>
</tr>
</tbody>
</table>

8.4.1 Artificial Bug Detection

The results shown here are based on 12 manual modifications done in the motion configuration file for the robot controller, see Section 7.4.1. This experiment provided promising results as a verification of the proposed method. In Figure 22 are how the Hausdorff distance change when performing changes to the controller motion configurations. This experiment provided verification to those certain modifications to MOC files are detectable with the proposed Hausdorff distance measurement. Additional result tables are available in Appendix A showing an evaluation of each of the MOC-files Hausdorff distance and time consumption.

Data outliers were found in a few test cases and were noticed in some of the TCP measurements that the VCSIM provided. These outlier test cases were found when calculating the TCP velocity. The outliers pointed to an error in the simulation environment, VCSIM-signals, which occur at the beginning of a run. The TCP-data from the controller with a modified MOC-file active has a different initial starting position. An example of such a faulty test case can be seen in Figure 23. However, a test case showing these kinds of behaviour is marked and is not considered to be a valid test case when a sub-set is later selected.

8.4.2 Real Bug Detection

The results presented in this section will describe how a real bug is detected. Those test cases that have the largest bug detection ability are those with the highest spikes in Figure 24. An evaluation of the test set are summarised in Table 8 and 9.

8.4.3 Compressed Test Set Validation

The initial test set was reduced from 160 to 13 test cases. Among these 13 best performing test cases are three of the four highest spikes included, see those which stand out in Figure 24. The TCP Hausdorff distance of the reduced test set is displayed in Figure 25 and in Tables 10 and 11 are an evaluation of the reduced test sets. Additional tables from evaluating the reduced MOC-file test sets are found in Appendix A.
Figure 22: Two plots showing the results on how a controller reacts to parameter changes in the configuration file. The results are created by calculating the Hausdorff distance between a run with the default configuration file and one with a modified configuration file. The plot on top shows how the VCSIM-signals vary and the plot underneath shows the same for VC-signals.

Table 8: This shows a test set evaluation based on a comparison between a reference controller and a controller with an active bug, both executed the initial test.

<table>
<thead>
<tr>
<th>Config. Type</th>
<th>Signal Type</th>
<th>Data Type</th>
<th>Mean [HD]</th>
<th>Std [HD]</th>
<th>Min [HD]</th>
<th>Max [HD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug VC</td>
<td>TCP</td>
<td></td>
<td>0.000201</td>
<td>0.000585</td>
<td>0</td>
<td>0.003569</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td></td>
<td>0.155431</td>
<td>0.323281</td>
<td>0</td>
<td>1.593051</td>
</tr>
<tr>
<td>Bug VCSIM</td>
<td>TCP</td>
<td></td>
<td>0.000073</td>
<td>0.000117</td>
<td>0</td>
<td>0.000928</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td></td>
<td>0.034340</td>
<td>0.054827</td>
<td>0</td>
<td>0.288537</td>
</tr>
</tbody>
</table>

Table 9: This shows an evaluation over the initial test set execution time. Note, the total time is the effective execution time of the test set.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug VC</td>
<td>VC</td>
<td>12.018031</td>
<td>7.351488</td>
<td>1.153152</td>
<td>23.341248</td>
<td>1922.884992</td>
</tr>
</tbody>
</table>
Figure 23: This is how the appearance of an outlier test case is detected, showing the first 10 out of 4000 samples. The sudden jump occurring at Sample = 7 gives a spike in velocity and the offset before returns a very high Hausdorff value. The same behaviour can be seen in the Y- and Z-data.

Figure 24: Two plots show the results on how a controller reacts to a real bug. The results are created by calculating the Hausdorff distance between a run without a bug and one with a bug. The plot on top shows how the VCSIM-signals vary and the plot underneath shows the same for VC-signals.
Figure 25: These four graphs show how the TCP Hausdorff distance performed on the selected test set. On the left is the VCSIM-signals and on the right is the VC-signals. The two graphs on top are how the selected test set performed with different MOC-files, and the two on the bottom are with a real bug loaded to the controller.

Table 10: This is a summary of the Hausdorff distance on the reduced test set, now 13 test cases. Best TCs, is a summary when the 13 best test case have been selected. Random, is a summary from 13 randomly chosen test cases.

<table>
<thead>
<tr>
<th>Test Set</th>
<th>Config.</th>
<th>Signal Type</th>
<th>Data Type</th>
<th>Mean [HD]</th>
<th>Std [HD]</th>
<th>Min [HD]</th>
<th>Max [HD]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best TCs</td>
<td>Bug</td>
<td>VC</td>
<td>TCP</td>
<td>0.000657</td>
<td>0.001244</td>
<td>0</td>
<td>0.003292</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.299237</td>
<td>0.443463</td>
<td>0</td>
<td>1.423017</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VCSIM</td>
<td>0.000157</td>
<td>0.000270</td>
<td>0</td>
<td>0.000928</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.086378</td>
<td>0.114061</td>
<td>0</td>
<td>0.288537</td>
</tr>
<tr>
<td>Random TCs</td>
<td>Bug</td>
<td>VC</td>
<td>TCP</td>
<td>0.000023</td>
<td>0.000039</td>
<td>0</td>
<td>0.000120</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.137330</td>
<td>0.263021</td>
<td>0</td>
<td>0.755218</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>VCSIM</td>
<td>0.000052</td>
<td>0.000057</td>
<td>0</td>
<td>0.000133</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.031003</td>
<td>0.040995</td>
<td>0</td>
<td>0.11128</td>
</tr>
</tbody>
</table>

Table 11: This is a summary of the test set time distribution on the reduced test set, now 13 test cases. Best TCs, is a summary when the 13 best test case have been selected. Random, is a summary from 13 randomly chosen test cases. Note, the total time is the effective execution time of the test set.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Best TCs</td>
<td>Bug</td>
<td>VC</td>
<td>11.892229</td>
<td>7.861275</td>
<td>1.237824</td>
<td>22.506624</td>
<td>154.598976</td>
</tr>
<tr>
<td>Random TCs</td>
<td>Bug</td>
<td>VC</td>
<td>12.154309</td>
<td>8.589182</td>
<td>1.237824</td>
<td>22.845312</td>
<td>158.006016</td>
</tr>
</tbody>
</table>
9 Discussion

The approach of applying simulations to software testing has to be further investigated before it can be said to find bugs. However, a reduced test set could be created from running controller versions with artificial bugs. When validated against a controller version with a real bug, three of the four best performing test cases was preserved. The available validation possibilities of the proposed method are believed to be insufficient to be able to draw a conclusion regarding the bug detection ability in general. Nevertheless, the fact that the one bug in possession has been found provides the intuition that future work of the developed method is promising. It has been unclear throughout the thesis work how the characteristics of a real controller bug can act out. That is why the modified configuration files have played such a big part. They were believed to be a substitute for the insufficient number of real bugs provided for this work.

It has been noted that there were differences between the VC and the VCSIM signal outputs. With the available tools, it was not possible to conclude why. For instance, in Figure 24 it can be seen that the VC data is about four times as large as the data from the VCSIM-signal output. Another difference that has been noted is between artificial and real bugs. If looking at the data from the artificial bugs, the scale of the Y-axis is about 200 times the scale of the Y-axis from the real bugs.

9.1 Future Work

The suggested continuation of the thesis includes various developments. For one, the integration of the method into the test engine would be beneficial. The separation of platforms is not intuitive, and produce the need for user interaction during the workflow. It would also be beneficial for the time consumption, to optimise the solution into one system.

One important process that should be conducted is validation. It has been performed, but to a very limited extent due to the restricted access to real bugs. With more real bugs this process can be performed, to properly validate that the method can find new bugs.

In the future, it would be beneficial to explore the possibilities of the usage of simulations. The environment has been empty in the simulation since it was the motion performance of the robot that has been investigated. The bug that was received originated from the motion department. With more bugs that are simulation-oriented, there exist research possibilities in how these differ from the current environment. In addition, there exist extensively more possibilities with robot interaction in the simulation that also can be tested. For instance, the interaction or collision with an object.

The developed test case generation system is primitively created. In the future, it would be beneficial with a more sophisticated generation of test cases, to further strive for those that perform best. Additional areas of the environment need to be investigated, and with different robot configurations. There were many areas of the environment that could not be accessed with the current method.

In addition, the process of generating artificial bugs could prove important. Mostly depending on the access to real bugs in the future. Test cases that are chosen according to the performance with real bugs will have properties more related to finding real bugs, even though they have similar appearances as artificial ones.

In the process of developing a more sophisticated generation process for artificial bugs, one possibly important addition is the sensitivity analysis. This will provide knowledge of how each parameter will affect the controller. Then the generation process can provide more weight in those parameters, to find more suitable configurations.

9.1.1 Bug Detected

In the event that a test set has evaluated a new controller version and identified that a bug has appeared. The most recent controller update has to be searched and refined for this bug. With a structured test set, it could be possible to identify which behaviour has been altered, giving an indication to where the bug might be located in the source code. The following stages include
applying additional testing to find if similar output is generated. If the same behaviour can be seen, the refinement has most likely not been in the area where the bug may be located.

9.1.2 No Bug Detected

In another situation, where the controller seemingly is in the same state as the reference controller, there might be a scenario where there is indeed no new bug introduced. However, there is also the alternative where a bug has been introduced, but the test set was unable to detect it. It may be that the bug is not providing enough difference in performance to be noticed, or the current test set is not evaluating the part of the robot that is affected by the bug.

Thus, in these situations, it is vital that the set of test cases are improved and revised, to decrease the risk of missing a bug. Also, the development process is continuously improved to widen how many aspects of the robot that are tested.
10 Conclusion

This thesis has proposed, implemented, and evaluated a new method with the purpose to find more bugs in software updates of a robot controller. This method has been developed with the goal to find motion anomalies in daily robot controller software updates. It was desired to create a regression test that can be executed nightly to find bugs in robot controller updates. This has included to design test cases, find a similarity metric, and extract an effective test set.

RQ1. How can a simulated test environment be used to develop suitable test cases to detect the velocity anomalies and motion discrepancies of industrial robots?

The simulation environment is used in conjunction with the already existing controller system at ABB. While the VC is executing a test set, the Algoryx physics simulation can be included, giving high precision feedback from the simulation environment. Throughout this thesis, two signal types have been used exclusively, TCP and motor position. This is due to that both signal types are motion oriented and can be extracted from both the VC and Algoryx physics simulation. Generating this data from two different versions of the same controller, an analysis method has been developed to distinguish anomalies between them. The velocity can be derived from the TCP data, which has been a useful measurement to find test case outliers in the Algoryx simulation data. However, the TCP and motor position data has given better results when finding motion anomalies between two controller versions.

RQ2. What is the best way to generate sensible test cases to automatically identify potential bugs in the robot controller?

One way that test cases are developed is by auto-generation. This process has the advantage of finding many combinations of the variable parameters. In addition, the process is time efficient and can create many different test cases fast. Another approach to test case creation is to manually construct them. It is clear that the automatically generated test cases have a significantly greater performance with artificial bug detection. This can be visualised in Figure 22, where the last 10 cases are the manual cases, which are barely visible in comparison to the rest. However, with real bug detection, seen in Figure 24, the manually created test cases performed similarly as the generated ones.

The orientation of the TCP has proven to be an important property of the test cases. The impact of target rotation can be seen in Table 5, where a test has been constructed to find how to target rotation improve test cases. In addition, in Table 6, it can be seen that the test case represented by rotation operation generate greater deviations, in TCP as well as motor position.

Moreover, the time duration has not proven to be of importance in the prospect of test case performance. In a test set with a considerably lower mean for test case execution time, the results did not deviate greatly. This can be seen in Table 9 which demonstrate the times for the initial set of test cases before the reduction has been performed. In Table 11, the times for the reduced test set can be seen. The best performing test cases showed to have a similar mean execution time as the original test set.

RQ3. Can artificial bugs be used to verify the generated test cases ability to find real bugs?

In the process of isolating those test cases with the best real bug detection ability, artificial bugs have played an important role. From the experimental results, it is clear that artificial bugs show indications on being able to distinguish test cases with high real bug detection ability, see Figure 25. The current prerequisites are not enough to be able to give a precise answer, more real bugs need to be obtained and validated against.

RQ4. How can the design of test cases be improved by the fact that the software functionalities to be tested are robot controllers and not generic software systems?

Creating motion performance test cases for a robot controller would have been more similar to a generic software if it was done without access to an object for the controller to actuate. With a mechanical structure for the controller to actuate, it has enabled access to feedback data from the robot when an invalid move has been made. Several underlying safety functions, located in different parts of the controller architecture stopped the execution if something went wrong. This is something that has improved the design of the test cases.

The development of the test case generation system utilised these safety functions. The simulation was a tool which provided the feedback data and gave a realisation of boundaries that the robot possesses. Thus, the test cases design has had many restrictions, due to the fact that it is a
robot controller under test. The restrictions for the areas, in which targets were generated, were found through analysing the feedback from the robot in its workspace.

**RQ5. How can a criterion be defined to know that sufficient testing has been performed?**

To define a criterion for when a bug exists in a new controller version has not been possible to achieve. This is due to the strictly limited access to real bugs to validate the reduced test set against. With access to more real bugs, it is believed to be possible to derive some criterion for when a bug exists. To create a criterion based on the performance of the artificial bugs would be misleading. As the results indicate is the real bug harder to detect than the artificial bugs, see the magnitude difference in Figure 25.
References


[38] A. A. Saifan, M. Akour, I. Alazzam, and F. Hanandeh, “Regression test-selection technique using component model based modification: Code to test traceability”,


A Experiment Data

Table 12: This shows an evaluation of the time distribution of the 12 MOC-files that the initial test set was executed on.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Signal Type</th>
<th>Mean (sec)</th>
<th>Std (sec)</th>
<th>Min (sec)</th>
<th>Max (sec)</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOC1</td>
<td>VC</td>
<td>11.857280</td>
<td>7.469947</td>
<td>1.112832</td>
<td>23.300928</td>
<td>1897.140672</td>
</tr>
<tr>
<td>MOC2</td>
<td>VC</td>
<td>11.204928</td>
<td>7.797092</td>
<td>1.124028</td>
<td>23.345280</td>
<td>1922.505984</td>
</tr>
<tr>
<td>MOC3</td>
<td>VC</td>
<td>11.203668</td>
<td>7.796370</td>
<td>1.124928</td>
<td>23.345280</td>
<td>1922.304384</td>
</tr>
<tr>
<td>MOC4</td>
<td>VC</td>
<td>11.220250</td>
<td>7.779846</td>
<td>1.205568</td>
<td>23.337216</td>
<td>1925.715456</td>
</tr>
<tr>
<td>MOC5</td>
<td>VC</td>
<td>11.217352</td>
<td>7.772670</td>
<td>1.128960</td>
<td>23.292864</td>
<td>1924.272000</td>
</tr>
<tr>
<td>MOC6</td>
<td>VC</td>
<td>11.225063</td>
<td>7.782155</td>
<td>1.128960</td>
<td>23.345280</td>
<td>1925.505792</td>
</tr>
<tr>
<td>MOC7</td>
<td>VC</td>
<td>11.224660</td>
<td>7.782806</td>
<td>1.128960</td>
<td>23.345280</td>
<td>1925.437248</td>
</tr>
<tr>
<td>MOC8</td>
<td>VC</td>
<td>11.210018</td>
<td>7.784279</td>
<td>1.128960</td>
<td>23.341248</td>
<td>1923.155136</td>
</tr>
<tr>
<td>MOC9</td>
<td>VC</td>
<td>11.209666</td>
<td>7.784450</td>
<td>1.128960</td>
<td>23.341248</td>
<td>1923.098688</td>
</tr>
<tr>
<td>MOC10</td>
<td>VC</td>
<td>11.209892</td>
<td>7.784187</td>
<td>1.128960</td>
<td>23.341248</td>
<td>1923.134976</td>
</tr>
<tr>
<td>MOC11</td>
<td>VC</td>
<td>11.209892</td>
<td>7.784147</td>
<td>1.128960</td>
<td>23.341248</td>
<td>1923.134976</td>
</tr>
<tr>
<td>MOC12</td>
<td>VC</td>
<td>11.209867</td>
<td>7.784176</td>
<td>1.128960</td>
<td>23.341248</td>
<td>1923.130944</td>
</tr>
</tbody>
</table>

Table 13: This shows a time evaluation of for the 12 MOC-files that the reduced test set, containing 13 test cases, was executed on.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Signal Type</th>
<th>Mean (sec)</th>
<th>Std (sec)</th>
<th>Min (sec)</th>
<th>Max (sec)</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOC1</td>
<td>VC</td>
<td>11.900603</td>
<td>8.015166</td>
<td>1.112832</td>
<td>22.542912</td>
<td>154.707840</td>
</tr>
<tr>
<td>MOC2</td>
<td>VC</td>
<td>11.877031</td>
<td>7.932893</td>
<td>1.124928</td>
<td>22.546944</td>
<td>154.401408</td>
</tr>
<tr>
<td>MOC3</td>
<td>VC</td>
<td>11.868037</td>
<td>7.929158</td>
<td>1.124928</td>
<td>22.546944</td>
<td>154.284480</td>
</tr>
<tr>
<td>MOC4</td>
<td>VC</td>
<td>11.948677</td>
<td>7.865881</td>
<td>1.314432</td>
<td>22.563072</td>
<td>155.332800</td>
</tr>
<tr>
<td>MOC5</td>
<td>VC</td>
<td>11.921383</td>
<td>7.889222</td>
<td>1.237824</td>
<td>22.413888</td>
<td>154.977984</td>
</tr>
<tr>
<td>MOC6</td>
<td>VC</td>
<td>11.910528</td>
<td>7.887780</td>
<td>1.23632</td>
<td>22.413888</td>
<td>154.836864</td>
</tr>
<tr>
<td>MOC7</td>
<td>VC</td>
<td>11.920763</td>
<td>7.892541</td>
<td>1.23632</td>
<td>22.413888</td>
<td>154.969920</td>
</tr>
<tr>
<td>MOC8</td>
<td>VC</td>
<td>11.892539</td>
<td>7.865465</td>
<td>1.237824</td>
<td>22.482432</td>
<td>154.603008</td>
</tr>
<tr>
<td>MOC9</td>
<td>VC</td>
<td>11.889438</td>
<td>7.864961</td>
<td>1.201536</td>
<td>22.482432</td>
<td>154.562688</td>
</tr>
<tr>
<td>MOC10</td>
<td>VC</td>
<td>11.892539</td>
<td>7.865473</td>
<td>1.237824</td>
<td>22.482432</td>
<td>154.603008</td>
</tr>
<tr>
<td>MOC11</td>
<td>VC</td>
<td>11.892229</td>
<td>7.865360</td>
<td>1.237824</td>
<td>22.482432</td>
<td>154.598976</td>
</tr>
<tr>
<td>MOC12</td>
<td>VC</td>
<td>11.892229</td>
<td>7.865360</td>
<td>1.237824</td>
<td>22.482432</td>
<td>154.598976</td>
</tr>
</tbody>
</table>
Table 14: This table shows an evaluation of the 12 MOC-files that the initial test set was executed on.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Signal Type</th>
<th>Data Type</th>
<th>Mean</th>
<th>Std</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOC1</td>
<td>VC</td>
<td>TCP</td>
<td>0.054234</td>
<td>0.091814</td>
<td>0.000000</td>
<td>0.521628</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>7.358323</td>
<td>8.577127</td>
<td>0.000000</td>
<td>28.169923</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.038167</td>
<td>0.050475</td>
<td>0.000000</td>
<td>0.235353</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>6.240843</td>
<td>6.984058</td>
<td>0.000000</td>
<td>23.104329</td>
</tr>
<tr>
<td>MOC2</td>
<td>VC</td>
<td>TCP</td>
<td>0.072025</td>
<td>0.129530</td>
<td>0.000000</td>
<td>0.570924</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>5.913361</td>
<td>6.288982</td>
<td>0.671333</td>
<td>22.141470</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.029489</td>
<td>0.043102</td>
<td>0.000000</td>
<td>0.237627</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>4.901886</td>
<td>4.869191</td>
<td>0.117925</td>
<td>17.141862</td>
</tr>
<tr>
<td>MOC3</td>
<td>VC</td>
<td>TCP</td>
<td>0.072025</td>
<td>0.129530</td>
<td>0.000000</td>
<td>0.570924</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>5.912613</td>
<td>6.288982</td>
<td>0.117891</td>
<td>23.104329</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.012999</td>
<td>0.026788</td>
<td>0.000000</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>3.390464</td>
<td>5.247895</td>
<td>0.000000</td>
<td>18.772361</td>
</tr>
<tr>
<td>MOC4</td>
<td>VC</td>
<td>TCP</td>
<td>0.001303</td>
<td>0.001169</td>
<td>0.000000</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>3.693781</td>
<td>5.059429</td>
<td>0.000000</td>
<td>18.721216</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.002739</td>
<td>0.002346</td>
<td>0.000000</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>3.390464</td>
<td>5.247895</td>
<td>0.000000</td>
<td>18.772361</td>
</tr>
<tr>
<td>MOC5</td>
<td>VC</td>
<td>TCP</td>
<td>0.001303</td>
<td>0.001169</td>
<td>0.000000</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>3.693781</td>
<td>5.059429</td>
<td>0.000000</td>
<td>18.721216</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.002739</td>
<td>0.002346</td>
<td>0.000000</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>3.390464</td>
<td>5.247895</td>
<td>0.000000</td>
<td>18.772361</td>
</tr>
<tr>
<td>MOC6</td>
<td>VC</td>
<td>TCP</td>
<td>0.001303</td>
<td>0.001169</td>
<td>0.000000</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>3.690307</td>
<td>5.247820</td>
<td>0.000000</td>
<td>18.772383</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.001302</td>
<td>0.001167</td>
<td>0.000000</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>3.693781</td>
<td>5.059429</td>
<td>0.000000</td>
<td>18.721225</td>
</tr>
<tr>
<td>MOC7</td>
<td>VC</td>
<td>TCP</td>
<td>0.002737</td>
<td>0.002346</td>
<td>0.000000</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>3.391171</td>
<td>5.247129</td>
<td>0.000000</td>
<td>18.772394</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000041</td>
<td>0.000390</td>
<td>0.000000</td>
<td>0.004449</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>3.697791</td>
<td>5.102406</td>
<td>0.000000</td>
<td>1.092855</td>
</tr>
<tr>
<td>MOC8</td>
<td>VC</td>
<td>TCP</td>
<td>0.002618</td>
<td>0.002346</td>
<td>0.000000</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>3.181766</td>
<td>3.915760</td>
<td>0.000000</td>
<td>2.587884</td>
<td></td>
</tr>
<tr>
<td>MOC9</td>
<td>VC</td>
<td>TCP</td>
<td>0.000033</td>
<td>0.000297</td>
<td>0.000000</td>
<td>0.003090</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>0.004585</td>
<td>0.845445</td>
<td>0.000000</td>
<td>0.810894</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000005</td>
<td>0.000005</td>
<td>0.000000</td>
<td>0.000450</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.002680</td>
<td>0.025412</td>
<td>0.000000</td>
<td>0.291561</td>
</tr>
<tr>
<td>MOC10</td>
<td>VC</td>
<td>TCP</td>
<td>0.000040</td>
<td>0.000371</td>
<td>0.000000</td>
<td>0.004182</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>0.009940</td>
<td>0.089236</td>
<td>0.000000</td>
<td>0.880085</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000029</td>
<td>0.000337</td>
<td>0.000000</td>
<td>0.004248</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.007330</td>
<td>0.073245</td>
<td>0.000000</td>
<td>0.881251</td>
</tr>
<tr>
<td>MOC11</td>
<td>VC</td>
<td>TCP</td>
<td>0.000013</td>
<td>0.000170</td>
<td>0.000000</td>
<td>0.002146</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>0.004389</td>
<td>0.055523</td>
<td>0.000000</td>
<td>0.702319</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000002</td>
<td>0.000029</td>
<td>0.000000</td>
<td>0.000362</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.001822</td>
<td>0.023050</td>
<td>0.000000</td>
<td>0.291561</td>
</tr>
<tr>
<td>MOC12</td>
<td>VC</td>
<td>TCP</td>
<td>0.000013</td>
<td>0.000170</td>
<td>0.000000</td>
<td>0.002146</td>
</tr>
<tr>
<td></td>
<td>MPOS</td>
<td>0.004389</td>
<td>0.055523</td>
<td>0.000000</td>
<td>0.702319</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000002</td>
<td>0.000029</td>
<td>0.000000</td>
<td>0.000362</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPOS</td>
<td>0.001822</td>
<td>0.023050</td>
<td>0.000000</td>
<td>0.291561</td>
</tr>
</tbody>
</table>
Table 15: This shows an evaluation of the Hausdorff distance distribution for the 12 MOC-files that the reduced test set, containing 13 test cases, was executed on.

<table>
<thead>
<tr>
<th>Config.</th>
<th>Signal Type</th>
<th>Data Type</th>
<th>Mean</th>
<th>Std</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOC1</td>
<td>VC</td>
<td>TCP</td>
<td>0.062004</td>
<td>0.105378</td>
<td>0</td>
<td>0.521628</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>7.742736</td>
<td>8.876493</td>
<td>0</td>
<td>28.169923</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.038378</td>
<td>0.048653</td>
<td>0</td>
<td>0.234146</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>6.521505</td>
<td>7.215029</td>
<td>0</td>
<td>23.104329</td>
</tr>
<tr>
<td>MOC2</td>
<td>VC</td>
<td>TCP</td>
<td>0.080044</td>
<td>0.139211</td>
<td>0</td>
<td>0.570924</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>6.170817</td>
<td>6.539497</td>
<td>0.001847</td>
<td>22.141470</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.030465</td>
<td>0.043199</td>
<td>0.000497</td>
<td>0.234640</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>5.081116</td>
<td>5.042297</td>
<td>0.117925</td>
<td>17.174186</td>
</tr>
<tr>
<td>MOC3</td>
<td>VC</td>
<td>TCP</td>
<td>0.080444</td>
<td>0.139211</td>
<td>0.001847</td>
<td>0.570924</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>6.170058</td>
<td>6.537522</td>
<td>0.000496</td>
<td>22.141466</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.030432</td>
<td>0.043216</td>
<td>0.000497</td>
<td>0.234640</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>5.629651</td>
<td>5.479316</td>
<td>0.008767</td>
<td>17.728877</td>
</tr>
<tr>
<td>MOC4</td>
<td>VC</td>
<td>TCP</td>
<td>0.001238</td>
<td>0.001168</td>
<td>0</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>4.024813</td>
<td>5.343435</td>
<td>0</td>
<td>18.721214</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.002816</td>
<td>0.002878</td>
<td>0</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>3.755290</td>
<td>5.337006</td>
<td>0</td>
<td>18.773631</td>
</tr>
<tr>
<td>MOC5</td>
<td>VC</td>
<td>TCP</td>
<td>0.001251</td>
<td>0.001184</td>
<td>0</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>4.034889</td>
<td>5.347328</td>
<td>0</td>
<td>18.721216</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.002814</td>
<td>0.002880</td>
<td>0</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>3.756861</td>
<td>5.539796</td>
<td>0</td>
<td>18.772383</td>
</tr>
<tr>
<td>MOC6</td>
<td>VC</td>
<td>TCP</td>
<td>0.001232</td>
<td>0.001172</td>
<td>0</td>
<td>0.003843</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>4.039902</td>
<td>5.343294</td>
<td>0</td>
<td>18.721225</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.002815</td>
<td>0.002879</td>
<td>0</td>
<td>0.016735</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>3.756002</td>
<td>5.535377</td>
<td>0</td>
<td>18.772394</td>
</tr>
<tr>
<td>MOC7</td>
<td>VC</td>
<td>TCP</td>
<td>0.000082</td>
<td>0.000548</td>
<td>0</td>
<td>0.004449</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.002245</td>
<td>0.143946</td>
<td>0</td>
<td>1.928556</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.002696</td>
<td>0.002937</td>
<td>0</td>
<td>0.016728</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.199870</td>
<td>0.436376</td>
<td>0</td>
<td>2.887884</td>
</tr>
<tr>
<td>MOC8</td>
<td>VC</td>
<td>TCP</td>
<td>0.000065</td>
<td>0.000417</td>
<td>0</td>
<td>0.003090</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.018915</td>
<td>0.118808</td>
<td>0</td>
<td>0.810894</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000010</td>
<td>0.000060</td>
<td>0</td>
<td>0.000560</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.005359</td>
<td>0.053757</td>
<td>0</td>
<td>0.291561</td>
</tr>
<tr>
<td>MOC9</td>
<td>VC</td>
<td>TCP</td>
<td>0.000079</td>
<td>0.000521</td>
<td>0</td>
<td>0.004182</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.019880</td>
<td>0.125409</td>
<td>0</td>
<td>0.880857</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000058</td>
<td>0.000475</td>
<td>0</td>
<td>0.004248</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.014660</td>
<td>0.103061</td>
<td>0</td>
<td>0.882151</td>
</tr>
<tr>
<td>MOC10</td>
<td>VC</td>
<td>TCP</td>
<td>0.000027</td>
<td>0.000239</td>
<td>0</td>
<td>0.002146</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.008779</td>
<td>0.078274</td>
<td>0</td>
<td>0.702319</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000005</td>
<td>0.000040</td>
<td>0</td>
<td>0.000362</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.008779</td>
<td>0.078274</td>
<td>0</td>
<td>0.702319</td>
</tr>
<tr>
<td>MOC11</td>
<td>VC</td>
<td>TCP</td>
<td>0.000027</td>
<td>0.000239</td>
<td>0</td>
<td>0.002146</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.008779</td>
<td>0.078274</td>
<td>0</td>
<td>0.702319</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000005</td>
<td>0.000040</td>
<td>0</td>
<td>0.000362</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.008779</td>
<td>0.078274</td>
<td>0</td>
<td>0.702319</td>
</tr>
<tr>
<td>MOC12</td>
<td>VC</td>
<td>TCP</td>
<td>0.000027</td>
<td>0.000239</td>
<td>0</td>
<td>0.002146</td>
</tr>
<tr>
<td></td>
<td>VC</td>
<td>MPOS</td>
<td>0.008779</td>
<td>0.078274</td>
<td>0</td>
<td>0.702319</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>TCP</td>
<td>0.000005</td>
<td>0.000040</td>
<td>0</td>
<td>0.000362</td>
</tr>
<tr>
<td></td>
<td>VCSIM</td>
<td>MPOS</td>
<td>0.008779</td>
<td>0.078274</td>
<td>0</td>
<td>0.702319</td>
</tr>
</tbody>
</table>