Tampere University

Santeri Vahos

TEST CASE GENERATION FOR EMBEDDED CONTROL SYSTEM

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Jose Luis Martinez Lastra
Luis Enrique Gonzalez Moctezuma
March 2024

ABSTRACT

Santeri Vahos: Test Case Generation for Embedded Control System
Master of Science Thesis
Tampere University
Master’s Degree Programme in Automation Technology
February 2024

Software is arguably one of the most important parts of any embedded control system, as microcontrollers or computers function as the brain of the system. Therefore, ensuring correct operation and quality of the software needs to be verified throughout the development process with different kind of software testing methods. As embedded control systems grow evermore complex to perform all kinds of tasks, creating tools that help developers to produce and maintain good quality software tests are needed.

Creating test cases requires developers to spend valuable development time on tedious, but important, task of figuring out how exactly they can exercise the software as much as possible. Algorithmically creating some of these test cases would speed up development process by moving some of the work usually left for humans to be done during a CI-process. Expecting algorithms to completely replace some human work is not realistic, so part of the software testing process needs to be looked from a different angle to achieve satisfactory results.

Usual way of regression software testing requires close introspection of implemented software and requirements, based on which test cases are then produced by software developers or test engineers. However, code coverage is a metric that can be used to evaluate quality of regression test sets by measuring how much of the program is being executed. Using this insight, algorithmic methods can be applied to produce regression tests that exhaustively exercise some program.

In this thesis, Genetic Algorithms are used in Model-Based software development environment, to generate regression test suites to be used in control systems of forest machines. Requirements that guide some of the different design choices for the algorithm are explored. More specific details are explored in a level that is required to produce the algorithm implemented in this thesis. The proposed algorithm is implemented and evaluated in Model-Based software development environment.

Algorithm was successfully implemented and significant convergence of code coverage was shown to be possible with the proposed approach. Usage as a software development tool, as initially intended, was shown to be possible, but most likely impractical. Biggest reasons for impracticality as a software development tool were long runtime and case-by-case parametrization, without which convergence was slow or low in code coverage. Additional future work might be able to transform the algorithm as tool from impractical to practical.

Keywords: CI-process, Genetic Algorithm, Model-Based Software Development, Test case generation, regression test

TIIVISTELMÄ

Santeri Vahos: Sulautettujen järjestelmien testitapausten tuottaminen algoritmisesti
Diplomityö
Tampereen yliopisto
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Helmikuu 2024

Nykypäivänä yksi tärkeimmistä osista sulautetuissa järjestelmissä on niiden toimintaa ohjaavat ja valvovat ohjelmistot. Näiden ohjelmistojen laatua pitää varmistaa erilaisia ohjelmistotestaus metodeja hyödyntäen kehitysprosessin aikana. Sulautettujen järjestelmien kehittyessä monimutkaisemmiksi, on noussut tarve luoda ohjelmistokehittäjien työskentelyä helpottavia ja nopeuttavia työkaluja.

Testitapausten luominen on aikaa vaativaa ja pikkutarkkaa työtä, jota ohjelmistokehittäjät tai testi-insinöörit tekevät luodakseen tapauksia, jotka suorittavat testin alla olevaa ohjelmisto mahdollisimman kattavasti. Algoritmisesti uusien ohjelmistoa testaavien testitapausten luominen nopeuttaisi ohjelmistokehitysprosessia siirtämällä osan ohjelmistokehittäjien manuaalisesta työkuormasta CI-prosessille. Näiden algoritmisesti luotujen testitapausten ei kuitenkaan voida olettaa täysin korvaavan ihmisten työpanosta, joten osaa ohjelmistotestausprosessista pitää tutkia eri kantilta, jotta testitapaukset tuovat lisä arvoa prosessiin.

Perinteinen tapa tehdä regressiotestausta jollekin ohjelmistolle vaatii kohteena olevan ohjelmiston tarkkaa tutkimista ja sen vaatimusten ymmärtämistä, jotta ohjelmistokehittäjä tai testi-insinööri pystyvät luomaan tarvittavat testitapaukset. Koodikattavuutta voidaan käyttää hyödyksi regressiotestejä luodessa algoritmisesti, sillä koodikattavuus indikoi kuinka suuri osa ohjelmasta on suoritettu. Algoritmi, joka maksimoida koodikattavuuden manipuloimalla ohjelman syötettä iteratiivisesti, pystyy myös luomaan myös regressiotestisettejä.

Tässä opinnäytetyössä geneettisiä algoritmeja hyödynnettiin luomaan regressiotestisettejä mallipohjaiseen ohjelmistokehitykseen. Luotuja regressiotestisettien käyttötarkoitus olisi hyödyntää niitä metsäkoneiden ohjausjärjestelmien laadun varmistuksessa. Yleisen tason vaatimuksia algoritmille ja siitä johtuvia suunnittelupäätöksiä on käsitelty. Tarkempia yksityiskohtia on käsitelty niiltä osin, kun on vaadittu tämän opinnäytetyön toteuttamiseen. Mallipohjaista ohjelmistokehitysympäristöä käytettiin hyödyksi niin toteutuksessa kuin myös algoritmin suorituskyvyn arvioinnissa.

Opinnäytetyössä kuvattu algoritmi implementointiin onnistuneesti ja sillä saavutettiin huomattavaa konvergenssia kohti korkeaa koodikattavuutta. Luodun algoritmin käyttö ohjelmistotyökaluna, joka oli opinnäytetyön lähtöajatus, todettiin olevan mahdollinen, mutta epäkäytännöllinen. Suurimmat syyt tähän olivat algoritmin pitkä suoritusaika, sekä yksittäistapausten vaativa parametrisointi. Ilman parametrisoinnin tekemistä konvergenssi oli hidasta tai koodikattavuus jäi matalaksi. Erinäisiä jatkokehitys mahdollisuuksia listattiin, joilla voisi lisätä algoritmin käytännöllisyyttä ohjelmistokehitystyökaluna.

Avainsanat: CI-Prosessi, geneettinen algoritmi, mallipohjainen ohjelmistokehitys, testitapausten luominen ohjelmallisesti, regressiotesti

PREFACE

I would like to thank my university instructors Professor Jose Martinez Lastra and Luis Moctezuma Gonzalez for their patience on my long thesis writing process, which took about year longer than planned. Similarly, I want to thank my current employer, John Deere Forestry Oy, for their patience on my longer than expected thesis process and giving me time to complete it at my own pace.

Tampere, 1st of February 2024

Santeri Vahos

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
CAN	Controller Area Network
CI/CD	Continuous Integration/Continuous Delivery
ECU	Electronic Control Unit
GA	Genetic Algorithm
IC	Integrated Circuit
LIN	Local Interconnect Network
MBSD	Model Based Software Development
MC/DC	Modified Condition/Decision Coverage
HIL	Hardware-in-the-Loop
MIL	Model-in-the-Loop
OS	Operating System
PC	Personal Computer
PDF	Probability Density Function
RTOS	Real-Time Operating System
PIL	Processor-in-the-Loop
SIL	Software-in-the-Loop
UML	Unified Modeling Language

1. INTRODUCTION

1.1 Background

Modern forest machines are complex pieces of engineering that need to operate in harsh conditions reliably for years. As they handle heavy loads, such as felling, processing, or transporting a tree, it is important to ensure safe and robust operation of a such system. Verifying a proper operation of a forest machine is not an easy task as new and ever more complex features are brought to market.

Embedded control system is in a heart of a modern forest machine, as it handles everything from complex hydraulically controlled movements to gathering, analysing, and updating production data to the cloud. These systems have ECUs running real-time operating systems to handle machine movements and basic operations. Additionally, they also run PCs to handle tasks that are not so real-time dependent.

Developing software for embedded control system of a forest machine is a multi-stage process all the way from defining requirements to doing full system test on an actual machine.

1.2 Problem definition

Creating test cases is one of the most repetitive processes during any kind of software development project. It usually requires meticulous inspection of the software to test all possible scenarios. Especially when creating regression suites, which compare current software to an older, already verified software version. There is a need to speed up the creation of these regression test suites because their creation is time consuming and repetitive.

Usually when creating useful test cases, it is needed to have at least some understanding of the system under test, if some specific functionality of the software component is being targeted with a test. With regression testing, main motivation is to have confidence that systems behaviour is as it was before. So, it is not necessary, even though it is desirable, to have cases that target specific predefined functionality of the software component when creating regression test suites. This insight on regression testing is something that could be utilized to algorithmically generate test cases that still bring value to the development process.

When generating test cases algorithmically, it is important to keep in mind that they will need to be as human readable as possible. That is why the resulting test suite should have good separation between cases so that the test different parts of the software. Also, they shouldn’t have too much noise in the signals if specific point of failure in regression test case needs to be found and analysed by the developer.

1.3 Objectives

This thesis focuses on how to generate regression test suite to be used in model-based system design with genetic algorithms, by trying to answer the following questions:

- How can genetic algorithms be used in test case generation with model-based system development?
- How well do genetic algorithms perform when generating tests for embedded control system software?
- What type of models work best with test case generation with genetic algorithms?

2. THEORETICAL BACKGROUND

2.1 Embedded control systems

Systems, which are used to perform some dedicated functions using microprocessors that interact directly with the real world, are called embedded systems. As the name suggests, computation is embedded into a device and is integral part to achieve required functionality. General definition of embedded systems is considered to be that they assist, control or monitor operation of a machine or device and their presence is not necessarily obvious to a casual observer [1].

Embedded systems interact with the real world by reading some sensor inputs, computing some control logic based on the measured data and then producing a wanted change in output. The output can be anything from radio waves to electronic solenoid controlling hydraulic oil flow [2]. Embedded systems are used to perform specific control functions and tasks, rather than have general-computing capabilities. Usually, embedded systems have lower computing power and can run with much more limited resources as general-purpose computing capable devices [3].

Generally, operating systems (OS) used in embedded systems can be divided to the three following categories:

- **Bare-Metal Operating Environments**, which consists of devices firmware directly running applications without any conventional operating system. This operating environment requires deep integration from application software to run on specific device as there are only few abstraction layers.
- **Embedded Operating Systems** enable the possibility to execute multiple applications simultaneously in a scalable way. Offers abstractions layers for integrating different systems and devices. Are usually referred as real-time operating systems (RTOS).
- **Fully Featured Operating Systems**, have the capability to run anything a general-purpose PC can run. Linux is commonly used OS in systems running Fully Features Operating Systems. [4]

2.1.1 Real-time embedded systems

The term real-time can be defined to be either latency based or schedule based. Latency based systems need to provide output within some predefined time to achieve satisfactory operation. Schedule based systems run control tasks in predefined intervals and often have hard real-time requirement. The terms hard real-time and soft real-time refer to the operation of the system if the real-time requirement is not met. [5]

Soft real-time systems retain their usefulness when operation time is exceeded, even though the value of the operation might decrease. One example is a user interface application. If action takes longer than specified, it is still better to display it than not. [6]

Hard real-time on the other hand does not allow the predefined execution time to be exceeded. Missing the execution deadline can cause catastrophic and dangerous failures in the systems operation. Machine control and safety critical system use scheduled systems with hard real-time requirement to ensure that operation is safe and robust. [7]

To achieve hard real-time requirement, complex embedded systems usually run on microcontrollers with some type of RTOS, which handles the task scheduling for different control loops. Each control loop is assigned to run in a RTOS task with a predefined task-rate. This ensures that sensors signals are read and control logic is processed without any variation in execution interval [8]. Using RTOS allows for multiple different control loops to be run at the same time by providing APIs for software components to interact with each other. This helps to create a layers of abstraction between the software components and the underlying computation system, which in turn helps interoperability and reuse of software components [9].

Interaction with other parts of an embedded system from RTOS point of view is done via devices drivers. Just as RTOS provides API for applications to communicate to each other, drivers provide APIs for RTOS to communicate with hardware. Common uses for device drivers are non-volatile memory management, communication between different ICs, sensor readings, output actuator handling and communications stacks [4].

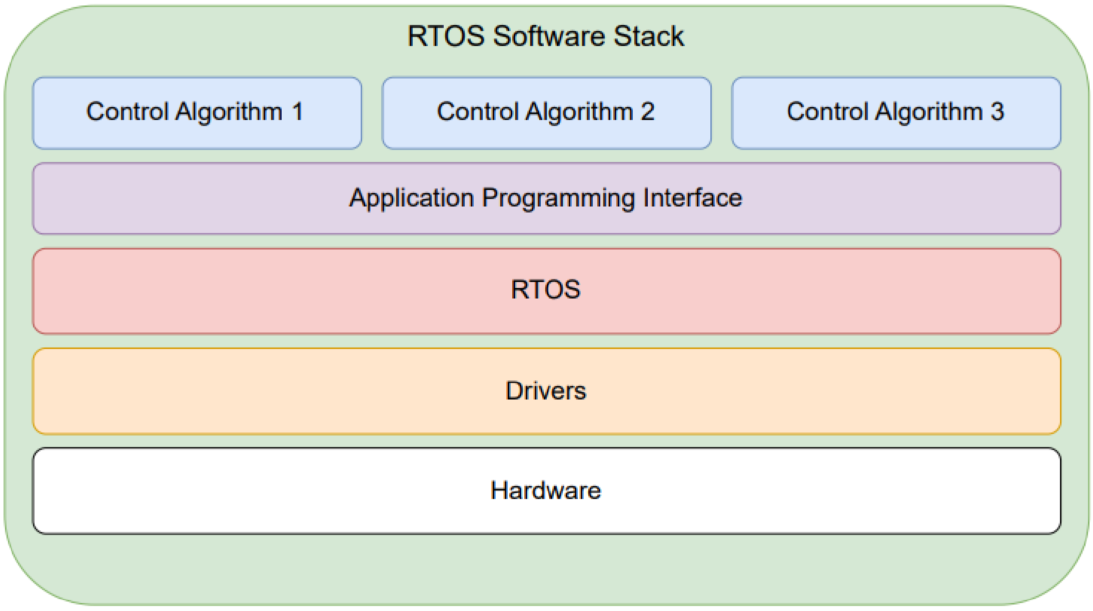


Figure 1. *System scheme of an RTOS system stack.*

Often in large and complex embedded control systems, such as consumer vehicles or heavy work machines, multiple Electronic Control Units (ECU) are needed to perform all required functions. Having multiple ECUs in one system requires some sort of connectivity between them to ensure synchronized and correct operation. This connectivity in automotive solutions is often done via CAN or LIN-buses. They offer robust and cost-efficient communication solutions and their data-transfer rates are in range of tens of kilobits per second to few megabytes per second [10]. This data-transfer rate has generally been enough for sensors and actuators used in automotive solutions, but recent advances is sensor technology and AI have brought a need for faster connectivity options, such as Automotive Ethernet. These newer solutions can offer data-transfer speeds up to 10 Gbps [11].

In the past, embedded control algorithms were developed completely with some low-level programming language, such as C [12]. This required developers and domain experts to have deep knowledge of the underlying computation system. It also posed challenges for individual contributors to visualize and comprehend complex system in its entirety. That is why new control algorithm design paradigm was developed: Model-Based System Development [13]. Its use cases and development workflows are explored in more detail in the next subchapter.

2.2 Model-Based System Development

Models are regarded as a mathematical representation of some system, where unnecessary details are abstracted away. These layers of abstraction can be peeled to reveal underlying building blocks of a model. Combining different models and adding models inside other models is a powerful way to represent some complex system. [14]

In MBSD (also called Model-Based Development, MBD) models are used to represent two sides of a dynamic system, a controller and a plant. Replicating the physical characteristics of a dynamic system into a mathematical model that can be used to represent the actual physical system is called the plant model. Controller on the other hand represents an embedded control system that runs algorithms and logic to control some physical system. Running both controller and plant at early-stage of development in simulation enables the discovery of defects and design problems way before any software is run on real hardware. This way of developing complex algorithms for electromechanical systems is considerably faster, as it enabled faster iterations on software during development [15]. MBSD is widely used in various industries, including aerospace, automotive, signal processing and motion control applications [16].

Engineering process used in MBSD is usually described by the so-called V-model, as shown in figure 1. The V-model is used broadly in development of safety critical systems because it focuses on testing the software at multiple levels. It consists of the following steps:

- First step of the V-model process is defining the requirements that specify what the system should do on a high level without describing any details about the implementation. This step usually involves collaboration between managers, software engineers, analysts, and domain experts.
- Architectural design consists of setting high-level design guidelines for the project such that complexity is minimized, and component reuse is maximized.
- Software design needs to fulfil the requirements specified by the requirements, by following the guidelines asserted in the architectural design. This step is done by software developer with guidance from domain experts. With MBSD it is also common for domain experts do early-stage software development and the responsibility to shift towards the software engineer closer to production the project gets.
- Code generation enables domain experts involved deep into the development process, as it can generate executable code for real hardware straight from the

models. This removes need for a software developer to act as a middleman when generating real-time running software from a model. [17]

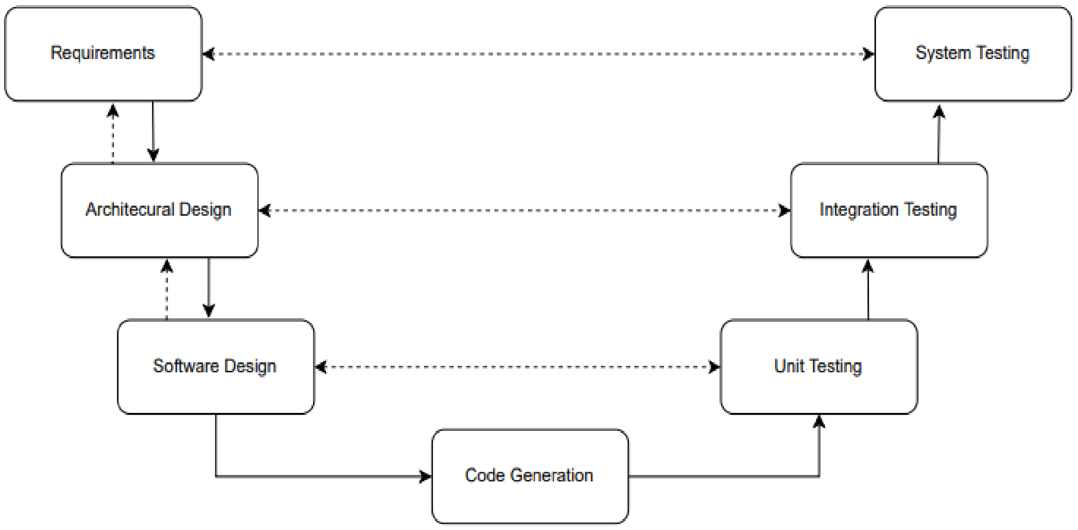


Figure 2. *V-Model used to describe MBSD.*

Software can be verified in many stages during of the development process when using MBSD to test for defects introduced in different stages of integration. The verifying process can be divided into 4 different stages:

1. First stage is to use MIL (Model-in-the-Loop), which means that control software is running still on the modelling software. Control software is developed and tested against plant model, which is a modelled representation of the physical system that control system is designed to control. Plant models can vary greatly in fidelity in MIL, as execution time does not be real-time. MIL is used in early stages of a development cycle as it enables fast development iterations. Control software doesn’t need to be built or installed on real hardware but can be run and tested instantly in simulation.

2. Second stage is to verify software in a SIL (Software-in-the-Loop) system. In this development phase, the model-based representation is generated into a code (usually C/C++ code) that can be compiled and executed in a virtual environment. Having SIL in the development pipeline aids in integration of new features. Source code can be debugged without the need of any external debuggers which also reduces time between iterations while integrating generated code to the system.

3. Third stage is called PIL (Processor-in-the-Loop), where the same generated code as in SIL is now cross compiled to run in a same processor as the final

production ECU will contain. This step is used to discover faults in code compilation process and faults in processor architecture [3]. All software projects do not necessarily need PIL testing as part of their testing process as there exists a lot to third-party systems that enable tested systems straight out of the box.

4. Fourth and final stage is HIL (Hardware-in-the-Loop) simulation, which consists of running the software on actual ECU hardware. In HIL-simulation, ECU is connected to a real-time simulation, where all of the ECU’s functionalities can be verified. HIL-testing cannot always contain same plant models used as in MIL because the real-time requirement of HIL. [17][18]

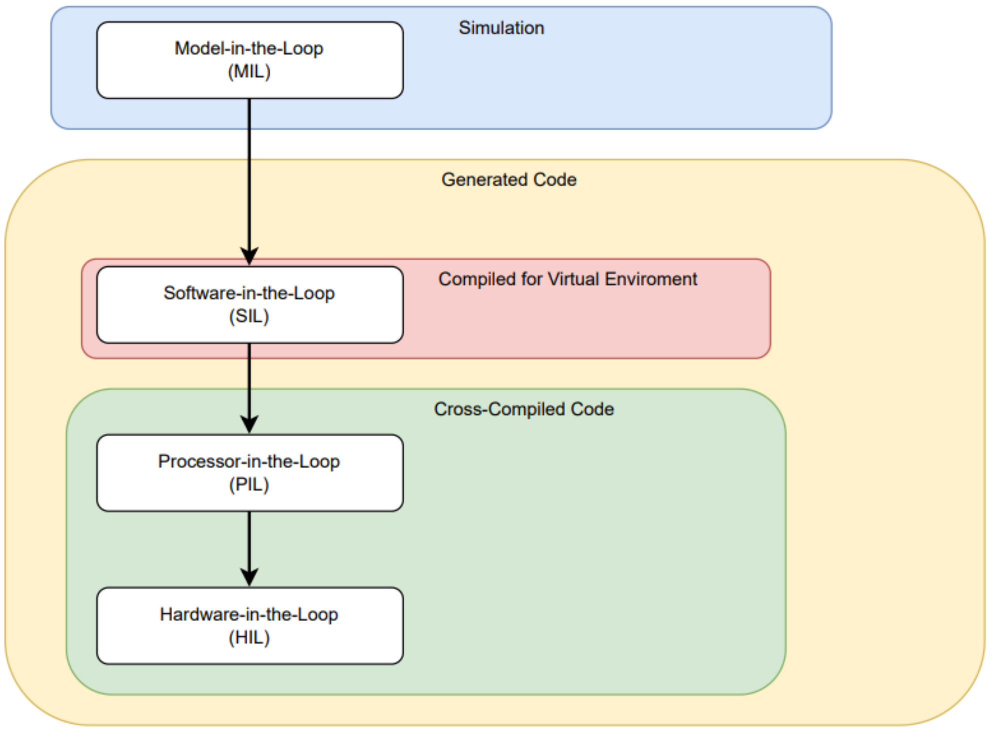

Simulation
Model-in-the-Loop (MIL)
Generated Code
Software-in-the-Loop (SIL)
Compiled for Virtual Enviroment
Cross-Compiled Code
Processor-in-the-Loop (PIL)
Hardware-in-the-Loop (HIL)

Figure 3. *Chart describing MIL, SIL, PIL and HIL*

2.2.1 CI/CD in Model-Based System Development

MBSD can utilize CI/CD (Continuous Integration/Continuous Delivery) pipelines in a similar fashion that is often used in traditional software development processes. Continuous integration of code changes throughout the process leads to faster time to market, higher quality, lower risk, transparency of the software and reduced cost by revealing defects as soon as they are introduced. [19]

The MBSD CI process consists of the following 4 steps: Build, Test, Package and Deploy, which together form so called “CI pipeline”. Whole CI pipeline is usually fully automated and triggered by a version control change made by a software developer. Visualisation of how CI pipeline in MBSD fits on to a local development cycle can be seen in Figure 4.

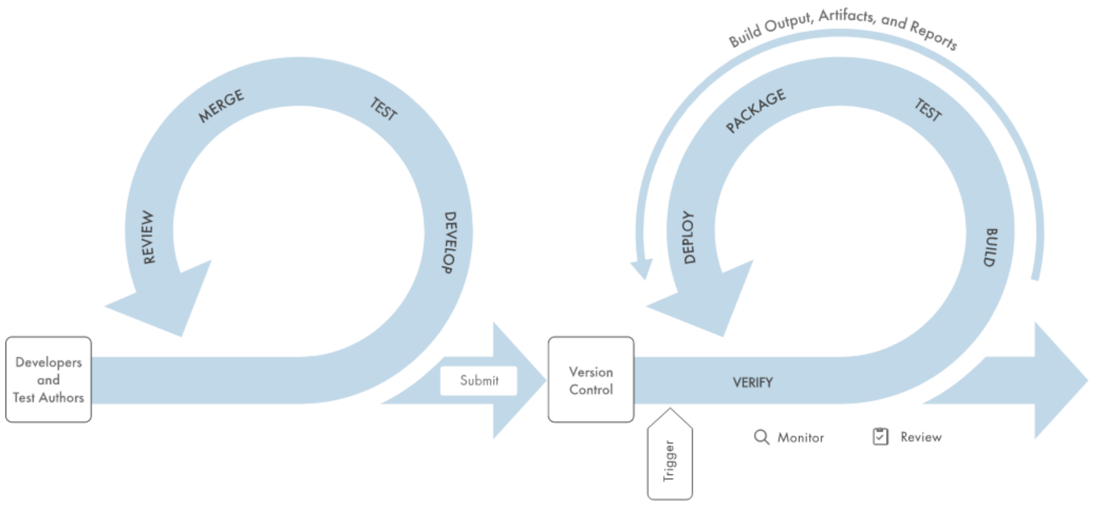


Figure 4. *MBSD CI pipeline*

Important steps of the CI process in MBSD are the testing stages. Local checks consist of quality checks, test results and coverage metrics in simulation environment to verify that quality of the software is adequate before committing to version control system. Tests that are run on the CI pipeline usually include the tests created during development but can also add more expansive tests to verify software more thoroughly. [20]

2.3 Software testing

The term software testing is used to describe the process of finding unintended programming errors, evaluating performance, and validating the intended behaviour of some program. All the complex paths that a program can take during execution can, and most often will be, too complex for a human to handle in their entirety. This is the reason why programs will usually contain at least some sort of unintended behaviour, often called bugs [21]. Having unintended behaviour in a system containing software can cause serious harm to a brands reputation and have large financial ramifications [22]. Therefore, it is important to try to reduce the number of programming errors to a minimum during a software development process.

Doing exhaustive testing, where every possible input is being tested, can be difficult and in most cases practically impossible. If a program has large input domain, the number of possible inputs the system can have grows very quickly to an impractical number. Trying to obtain a relatively small subset of the input domain that gives a good coverage of the program is key in software testing. Naik and Tripahty describe in their book Software Testing and Quality Assurance, activities that an engineer must do to perform test on a program with one test case:

- Identify test objective so that the test case has a clear purpose.
- Select inputs based on software requirements, source code, or with software engineers own expectations.
- Compute the expected outcome in some high level, so that the outcome isn’t a complete surprise.
- Set up program for execution in environment that the test case is defined to work with.
- Execute program with selected inputs and store the outputs.
- Analyse results by comparing the executed output to the expected output by giving a verdict if test case passes, fails or is inconclusive.

Systems are tested on different levels and different stages of a project to verify and validate functionalities that may not be possible or cost-effective to do elsewhere in the process.

Software is tested in multiple stages during the whole development process. Some tests only make sense to execute at a certain phase of the development. Also, some software projects do not always need all steps listed in the next subchapter. Commonly used names for the different stages are Unit testing, Integration testing, System testing and Acceptance testing [24].

2.3.1 Unit testing

Testing component of a program in a way that it isn’t dependent on other components is called unit testing (also sometimes called component testing). For a test to be a unit test, it is often considered to described by the following characteristics: verifies small piece of code, verifies it quickly, and does it in isolation from other pieces of code [24]. Developer who implements some software functionality is usually the person responsible for constructing the unit tests for given functionality. This is because the developer has a good understanding of the implemented component and thus is the best person to write the unit tests. Creating unit tests also help developers gain better understanding of the functionality and design of the component under test [25].

In fully functioning software, there can be dependencies between components and testing in complete isolation might not be possible. These limitations are usually solved by creating so called "dummies". Meaning that some complex functionality is simplified in a way that it is "good enough" for the unit test [26].

In traditional software development smallest piece of program that is being unit tested is usually a function or a class. In MBSD, smallest testable unit is a model, which is similar to a function in that it can contain constants, parameters, inputs and outputs. Additionally, models can contain other models, just as functions can call other functions. MBSD development tools also offer simulation capabilities that can be utilized when running unit tests. [63]

2.3.2 Integration, system and acceptance testing

In integration testing phase, software components are not being run in isolation from other components as in unit testing phase. Having components interact with each other can bring forward defects that unit tests will not be able to reproduce. Integration testing usually involves adding different components incrementally to the software and testing their interfaces. This helps to prepare the software for full system test without major catastrophic errors when everything is introduced incrementally. When every component is integrated to the software and all major errors have been fixed, then integration testing is done.

System testing is done when every implemented component is running in actual real-world environment at the same time. Exhaustive test suite is usually impractical for a full system test and is instead used by test engineers to test specific functionalities against requirements. This also enables the possibility to execute stress tests when system is loaded with real world operation instead of just simulated load.

Final testing stage is acceptance testing, where software requirements can be compared against the output of the system. Acceptance criteria must be defined beforehand and must be met before software is deemed to be adequately tested for full release. [24]

2.3.3 Regression testing

During software development process components under work are usually being developed throughout the whole process. There can be additions on functionality, requirement changes or bug fixes. If a component is adequately tested and verified at some point of development, these new additions to the software shouldn't introduce any new unwanted behaviour. This is where regression testing is used. Regression tests compare the output of a newly modified software to some older version, which behaviour has been verified to satisfy the requirements. If a test passes, it indicates that the outputs fit the acceptance criteria, and that the new software version behaves just as the old. If the test fails, there is too large variation in the output and the root cause will need to be checked by a developer or test engineer. [27]

Implementing new or modifying old functionality can, and most likely will, cause some of the regression tests to fail. This means that the regression test suites need to be updated frequently, so that they retain their usefulness. It is also good practice to always run regression suites when new software version is released, or even during daily software builds. Automating regression suites by integrating them to the CI process is highly recommended because running same tests manually can be tedious and time consuming. [28]

2.3.4 Black-box and White-box testing

Testing a system and having no knowledge in the inner workings of a system is called black-box testing. Only the systems inputs and outputs are available to the tester in this testing methodology. This kind of testing is usually used when verifying some requirement as only inputs and outputs are defined. Black-box testing is also called functional testing. Integration, system, and acceptance tests are usually considered to be black-box tests. Benefits for doing black-box testing are the following: testing the software from end users' perspective and the person doing the test activities doesn't need to have any knowledge of the specifics of the implementation.

White-box testing requires access to the source code of the system under test. This is because white-box testing activities require structural knowledge of the system [29] . With white-box testing there can be more confidence in correct operation of a system because different routes that the code execution can take can be monitored. Unit tests are white-box tests as they are usually written by the same developer who writes the software under test. Doing code coverage analysis is considered white-box testing.

2.3.5 Code coverage

Code coverage analysis is a way to measure how much test cases exercise a system under test. Higher code coverage percentage gives higher confidence that the system does not have some unwanted behaviour. Doing this kind of analysis requires a way to instrument the software during program execution. Meaning that it is possible to measure what points of execution the software reaches [30].

Conceptualizing the extraction of code coverage can be described the following way: If execution of a program is considered so that every decision is an edge on a graph and every vertex is a state of a program on the same graph. Then it is possible to extract Branch Coverage by measuring how many of all possible edges have been traversed during code execution. Every edge is only counted once, even though it can be traversed multiple times during execution of test cases. In figure 4 green arrows represent traversed vertices and black vertices decisions that execution didn’t take. Opposed to branch coverage, Statement Coverage measures percentage of states that the execution has traversed [29]. Figure 4 also represents reached states in blue and states not reached in yellow.

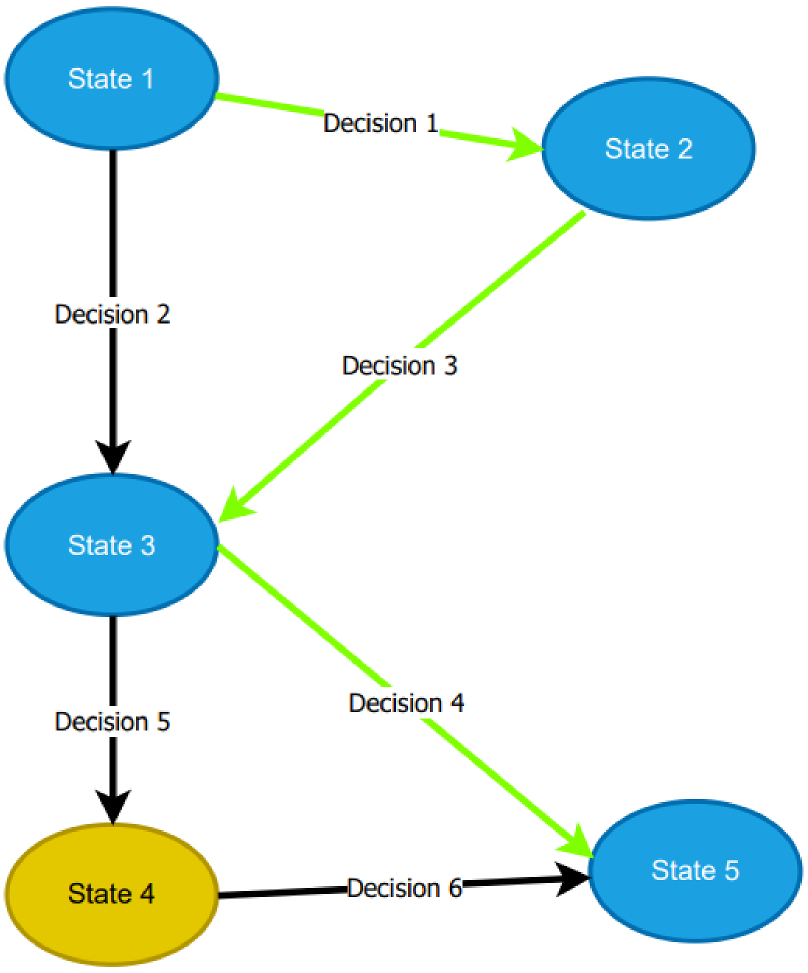


Figure 5. *Branch and Statement coverage visualized.*

Decision/Condition coverage measures the execution of every Boolean sub-expression along a branch. Every output in a sub-expression must be true and false at least once, for it to be counted towards Decision/Condition Coverage [29]. Modified Condition/Decision (MC/DC) coverage differs from normal Decision/Condition coverage by showing that every condition changes independently the decisions outcome. MC/DC requires the largest number of cases from all mentioned coverages to achieve 100% code coverage [31]. It is also sensitive to the implementation of the software structure so achieving 100% MC/DC coverage is not always possible. [32]

Simulink can also perform all the performance analyses mentioned above, but with certain differences. Statement coverage is called Execution Coverage in Simulink and it only measures that a Simulink block has been executed. No change in block inputs is needed to achieve good Execution Coverage. Decision Coverage in Simulink is similar to Branch Coverage mentioned above. Condition Coverage is similar to Decision/Condition Coverage mentioned above.

MC/DC Coverage in Simulink has certain limitations on its capabilities. It cannot analyse the MC/DC Coverage for expression that has more than 12 conditions or different types of logical operators. Some blocks also don't support MC/DC and might support Condition Coverage or Decision Coverage, so it is possible to achieve 100% MC/DC Coverage to some models that might have less than 100% Decision and Condition Coverage [33]. Visualisation of how MC/DC on Simulink's stateflow-charts can be seen in figure 5. It shows decisions with 100% MC/DC coverage in green and decisions with less than 100% MC/DC coverage in red.

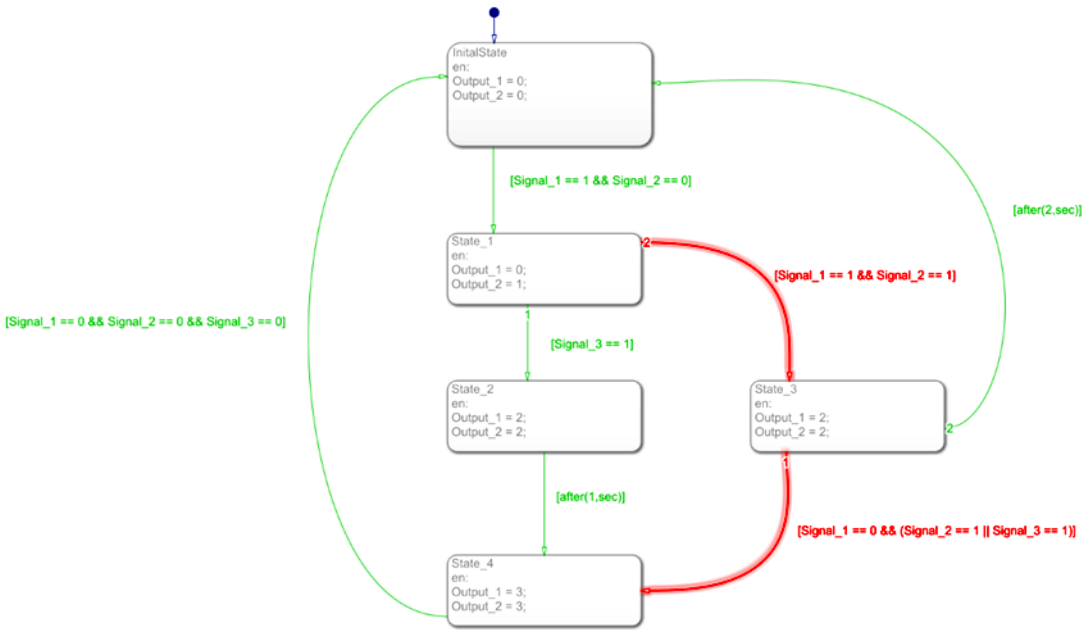


Figure 6. *Visualization of MC/DC coverage in a Simulink's stateflow chart.*

2.4 Metaheuristic algorithms

Metaheuristics are frameworks for algorithms used to solve complex optimization problems that are generally non-linear, stochastic, non-differentiable and discontinuous. They are often inspired by some sort of natural process [34]. The Greek term "meta" is a prefix used to describes subject in a way that transcends its original limits [35]. Heuristics means a way of finding a solution by learning, discovering, using trial-and-error methods. In other words what metaheuristics means, is to have create some heuristic concept and to create a higher-level framework for algorithms to be based on it [36]. Metaheuristic algorithms provide solution to this by being "good enough" and providing this solution in a reasonable execution time. [34]

Two key functions of metaheuristic algorithms are exploration and exploitation. These terms are also referred as diversification and intensification [36]. Exploration is considered as the functionality of the algorithm that explores surrounding areas adjacent to already explored areas. Exploitation refers to the functionality of “breaking through” into new, yet undiscovered areas of the search space. [37]

Different types of metaheuristics algorithms have been classified as local search-based, construction-based and population-based. Local search-based metaheuristics algorithms only search the space within the vicinity of current solution. This type of search is also called Classical Neighbourhood search [38]. Construction-based algorithms build structure from sequences where one move leads to another, which then is able to traverse the search space. They are often used in problems where solution requires traversing a graph, for example optimizing network routing [39]. Population based algorithms are generally called genetic algorithms, which are covered in more detail in the next subchapter. Other classifications also exist, such as trajectory-based, nature-based and non-nature-based metaheuristic algorithms. But algorithms in these classifications also generally fall under the previously listed three classifications [40][38].

2.4.1 Genetic algorithms

One of the first instances of using evolutionary algorithms was described in research article written by R. M. Friedberg in 1958 titled “A Learning Machine: Part 1”. The algorithm was used to find a program to match given inputs and outputs [43]. However, key ideas for the standard generic algorithms were introduced in only in 1962 by John H. Holland in his paper “Outline for a logical Theory of Adaptive Systems”. This paper introduced the concepts of mutation, population generations and “good enough” solutions in one framework [44].

Genetic algorithms are population-based metaheuristic algorithms that work on the Darwinian principle of survival of the fittest. Meaning that, higher the performance of an individual, higher the chance of survival and thus higher change to spread its genes for future generations. Genetic algorithms also pull one other key inspiration from nature, mutation process. This mutation process brings variation to the population and thus helps genetic algorithms to escape local minimum [41].

Genetic algorithms differ from classical search algorithms in the following way:

- Classical search algorithms create only one point per iteration and sequence of points then approaches optimal solution. In genetic algorithms, a population of solutions is created each iteration and best point in population approaches a solution.

- Classical algorithms use deterministic computation in the selection of a new point. Genetic algorithms use random values in computation, so they are non-deterministic [42]

Having a population of solutions, enables creating of a new set of solutions based on old ones where good traits are kept. Terms usually used to describe genetic algorithms are named after to their counterparts in nature:

- Individual – possible solution to the problem. Also often called Chromosome.
- Population – A set of individuals.
- Generation – New population. Iteration of the genetic algorithm.
- Gene – A characteristic of an individual that can be altered.
- Fitness – Performance of an individual as a solution to the problem.
- Crossover – Gene selections from two parents to offspring.
- Mutation – Random variation of gene.
- Offspring – Individual created from two individuals from the previous generation.
- Selection – Parents selected for crossover process. [45]

Standard Genetic Algorithm can be seen described in figure 6.

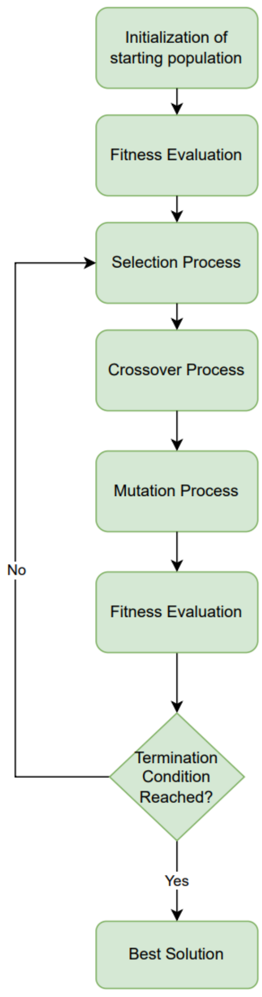


Figure 7. *Flowchart of a standard Genetic Algorithm.*

Genetic algorithms require that there is a way to rank solutions in a population based on their performance on a given problem. This performance score is called fitness and is calculated by fitness function for every individual. Every application has a different fitness function, as the fitness function sets constraints to the problem and enables the algorithm to converge at some optimum result. Convergence of a genetic algorithm requires it to be run for multiple generations. When generation count increases, so does the fitness score of a top performing individual. Therefore, fast as possible execution time is desirable when designing genetic algorithms and their fitness functions.

Fitness threshold is a value that can defined before the execution of the genetic algorithm to indicate that optimal or good enough solution is found. The plateauing of fitness scores of the top performing individuals can be defined to be end condition as well, if no good estimate range can be defined for the fitness score before execution. [46]

2.4.2 Crossover

Creating new individual from two parents is called crossover process. Genes from two parent individuals are selected by some crossover operations in such way that a new individual can be created from those genes. This way the genes that help the individual to score higher fitness scores are passed on to new generations and thus help the algorithm to converge.

Selecting the proper type of crossover operation for every solution is critical to avoid getting premature or slow convergence. Simplest type of crossover operation is 1-point crossover, where all the genes before a point are selected from one parent and all genes after the point are from the second parent. This point can be selected randomly or be predefined. Example of 1-point crossover where point is between genes 3 and 4 can be seen below:

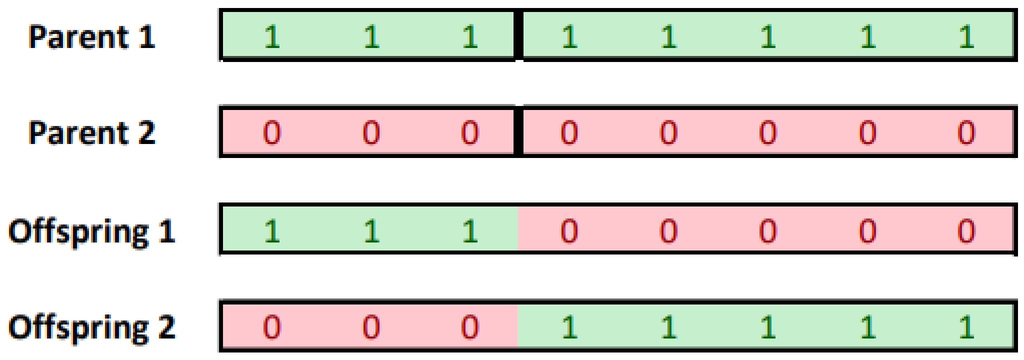


Figure 8. *Example of 1-point crossover operation*

K-point crossover bring more complexity compared to 1-point crossover by adding K-number of points to parents where genes are split [47]. Example of K-point crossover can be seen below, where points are after genes 3, 4 and 6:

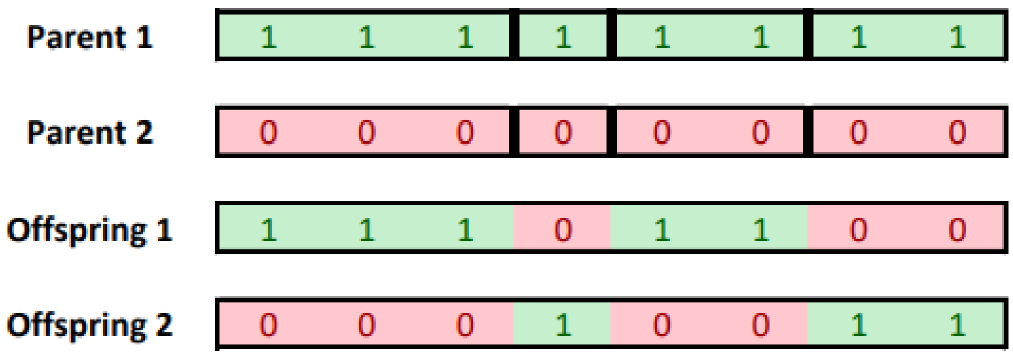


Figure 9. *Example of K-point crossover operation, with K = 3*

Uniform crossover combines genes from two parents by sampling from a uniform distribution and defining bit mask based on result of the sampling. This bit mask then describes which parent the corresponding gene of an offspring should be used from. Another offspring can be formed when the values of the bit mask are inverted. [48]

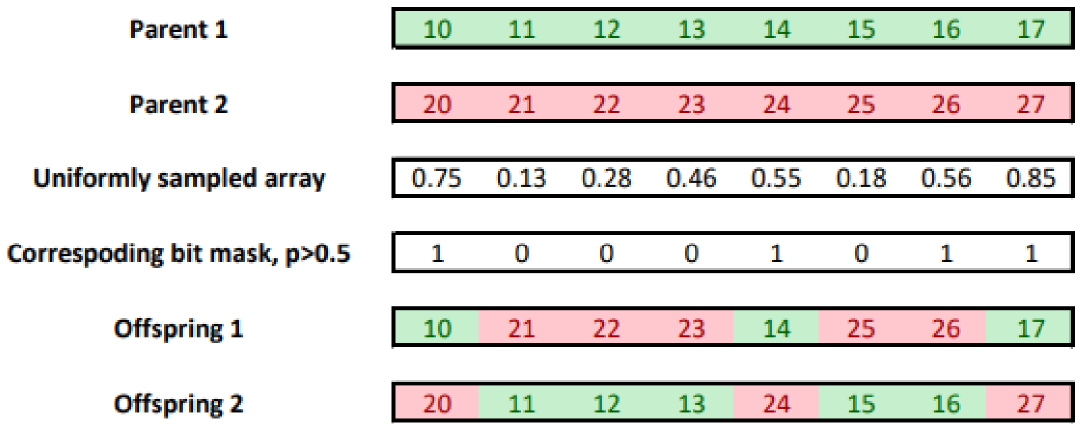


Figure 10. *Example of uniform crossover operation*

2.4.3 Mutation

Just the combination of two parents most likely cannot cover the whole input space and thus significant convergence is very unlikely. Some other operation is need for the algorithm to potentially reach every point in the input space. This operation is called mutation operation, which is brings variety to the population and thus helps drive the algorithm towards optimal solution. For standard genetic algorithms, in practice mutation means that some percentage of genes in the offspring are changed to some random values. Altered genes are selected randomly. [49]

In some problems there isn't just one optimal solution. There might be many solutions that are in practice equally good and in some cases getting multiple solutions might be preferable. Mutation operation enables the possibility to reach to solutions by bringing stochasticity to the algorithm, by changing random genes to random values. In other words, mutation operation makes genetic algorithms non-deterministic. [34]

2.5 Literature review

There have been numerous studies on using genetic algorithms on test case generation. It has been shown that they can successfully bring added value to a software development process by creating useful test cases [50][51][56][53]. They often have some sort of variation based on the standard genetic algorithm to help faster convergence of solution on given problem. Some of the test case generation research is based purely on output domain analysis [52].

Using genetic algorithms poses the challenge of needing large number of executions on the fitness function to achieve satisfactory results. Attempts to alleviate the strain that compute heavy fitness function poses on a genetic algorithm's performance have been researched. One of the ways that this could be achieved is by using some approximation of a fitness function, instead of doing the full calculation. Frameworks for injecting approximate fitness calculations into genetic algorithms to speed up convergence have been proposed and implemented. These implementations have used neural networks for the approximation [53]. Fitness function approximation has been shown to prevent information loss over generations, but it is not capable of producing new information that aids convergence [54][55].

For model base software development, there are studies on test case generation based on generating the cases from UML-charts that describe the system [57]. Also, genetic algorithms have been used to create test cases on Simulink models and stateflow charts [58]. Via mutation testing, genetic algorithms have also been used on Simulink models to perform evaluation of regression test suites. Mutation testing in this case means that models are modified in such a way that they bring out missing coverage of test suites [59]. All of the previously mentioned test case generation research has focused on generating test cases on synthetic benchmarks, not real-world production software.

3. DESIGN

This chapter goes through a proposal on how to apply genetic algorithms in such way that can be used to create regression test suites from scratch or expand already existing suites, with certain limitations in mind. Also, motivations behind certain design decisions are explored. The focus is on creating regression tests suites directly from production Simulink models used in forest machines, but same concepts could be applied in other MIL environments as well, given that there is a possibility to instrument the code in similar way that there is in Simulink.

3.1 Requirements

One of the key concepts in model-based system development is the usage of plant model to test the software against. This approach gives realistic feedback to the software under development; but brings forward also one big hurdle that limits the possible performance of genetic algorithms, large computation time. Minimizing execution time is critical in achieving good results within a reasonable computation time. Methods used in this thesis rely heavily on large number of simulations, so the genetic algorithm was developed to work on models that don't need plant response.

Expecting an algorithm to create exactly something like a human can do with deep understanding of some system is not realistic. That's why it is needed to reframe the usage of regression tests from a different angle: The regression test cases can exercise any part of the system without any correlation to actual machine behaviour, if they give good coverage. This way it is possible to achieve satisfactory results with genetic algorithms.

Basic principle of the proposed algorithm is to find a set of input vectors for a software component with genetic algorithms that fulfil the following conditions as well as possible:

1. Achieve high code coverage.
2. Minimized the complexity of each input vector.
3. Minimized overlap coverage of each input vector with other input vectors.

The motivation behind of the above mentioned 3 characteristics are the following: having high code coverage gives confidence in that the software is being exercised adequately. So, if the given input vector set is used in regression test purposes, the chances of unwanted behaviour slipping through unnoticed is smaller. Complexity of each vector should be small as possible as even though they don’t necessarily correspond to any specific part of the software, they still need to be human readable if a regression test fails. Minimizing coverage overlap as much as possible is needed to have wide set so that all of the test suite doesn’t trigger all at once if a deviation from baseline is detected. Additionally, it is possible that state-machine based logic might end in such states that achieving 100% will not be possible with only on case.

Genetic algorithms are non-deterministic and so by using this attribute it is possible to augment already existing input vectors in such way that their coverage increases. Even when giving the same input twice, the output can be different. This enables the usage of same algorithm with same parameters even when starting from scratch or when starting from an existing set of test cases.

For code coverage metric, any of the different ones mentioned in chapter 2.3.5 can be used as a basis for defining the fitness function. Reaching high coverage in some types of coverages might be harder than others, so termination condition for the genetic algorithm should be the plateauing of the fitness score rather than some desired code coverage value. Having other metrics for individuals’ fitness is also necessary to avoid generation of similar cases. This can be achieved by penalizing each individual by being too similar to other, already generated cases.

3.2 Algorithm proposal

3.2.1 Overview

Base for the whole algorithm proposal is the standard type of genetic algorithm described in Figure 7, but with more functionality added on around it to make it suitable for the problem of generating test cases. The whole algorithm can be seen visualized in Figure 11. Main additions to the standard genetic algorithm are case similarity penalty evaluation, simplifying a generated test case and multiple iterations of the whole algorithm.

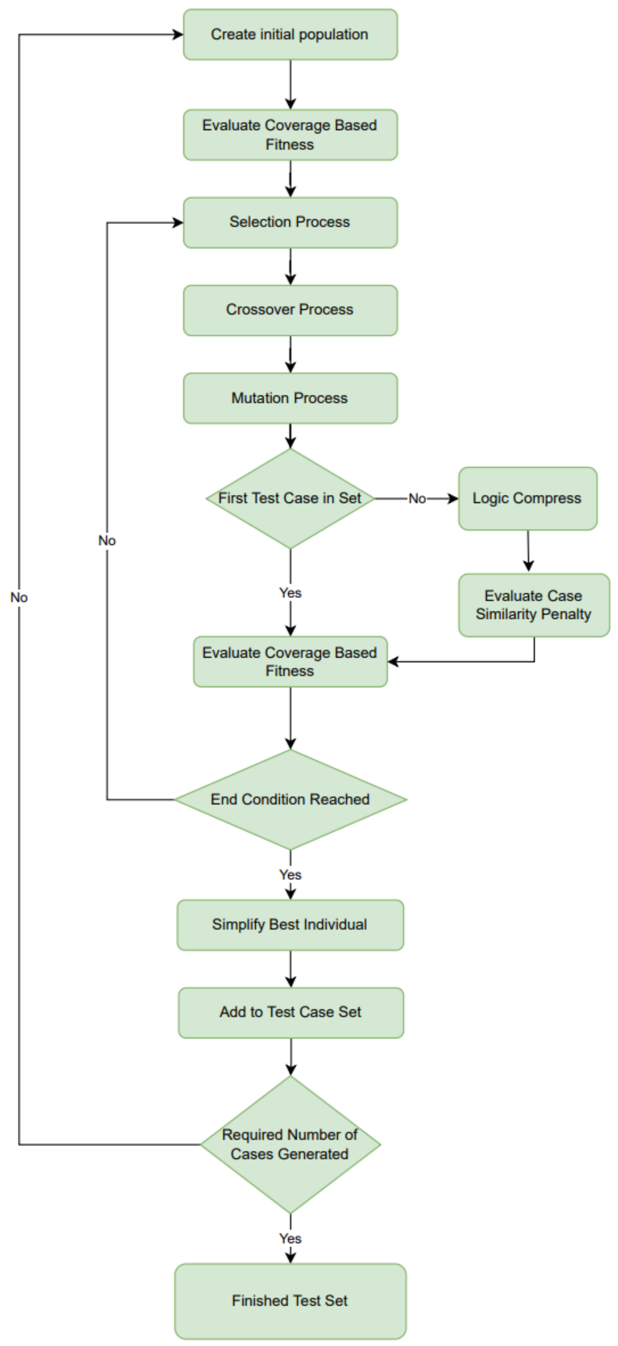


Figure 11. *The proposed algorithm.*

3.2.2 Standard genetic algorithm implementation

Generating one test case starts with by choosing some starting state for the input vector. This state can be vector of zeros, ones, or random values. It can also be some human-defined or previously generated test case. This is possible because the input vector is copied to every individual in the starting population. Gradually the mutation operation will bring variation to population and so homogeneity of the population isn’t a concern at the start.

In practice, the input vector is an array with size of $m\ x\ n$. Number of input signals to the model being m, and number of time steps of the simulation being $n.$ So, each column of the array represents a time series signal. All the values in the array are always between the closed interval $[0,1]$. A gene in this implementation is a row of values in the array. Before the generated input vector is inputted to the target model, each column is first scaled to match expected signal range. This is done achieve interoperability of the same genetic algorithm architecture between different target models, even when the number of signals and their ranges vary.

The extraction of coverage from simulated model isn't exactly white-box or black-box testing, because the full path what the execution takes isn't known, but metrics other than inputs and outputs are extracted. So, it is utilising so-called "grey-box" testing **Virhe. Viitteen lähdettä ei löytynyt.**. Code coverage metrics indicate the fitness scores of test cases in a given model.

Selection process is done in a probabilistic way, by ranking the top performing individuals and assigning a selection score for all of them. Selection score is defined in such way that higher ranking individuals have higher selection score than lower ranking. This score is then used as probabilities for random sampling process, which always picks 2 individuals for mating. In practice this means that higher fitness individuals get selected for mating more often than lower fitness individuals.

Crossover is done by applying generated crossover mask over two individuals and retrieving parts indicated by the mask. The crossover mask is a binary array that gets inverted for the second individual. This is done to ensure that resulting child is the same size array as the parents. Another unique child can be extracted from same by swapping the crossover masks between parents. Because a gene is a step consisting of all the signals, this type of crossover process carries over logical sequences from parents to child, given that the crossover mask contains same value consecutively.

3.2.3 Additions to standard genetic algorithm

When generating the first test case in the whole test set, individuals' fitness score in each generation is calculated only from the code coverage it achieves. All the other test cases will have additional penalty applied to their fitness scores, based on their similarity with already existing suite. This is done to reduce overlap coverage between cases.

After the algorithm has plateaued or reached an end condition for one test case, it is being fed through a simplifying algorithm. This algorithm determines one dominating value for each column of the array. This dominating value is selected by choosing the value that occurs the most in any given signal. After a suitable dominating value is selected, the algorithm fits the dominating value to every sample that doesn't have a dominating value in it. Fitting process is done one-by-one to every sample, then simulated and code coverage is calculated. If the coverage decreases, sample is important, and it cannot be altered. On the other hand, if the code coverage doesn't decrease, that particular sample doesn't bring any value to the solution so it can be set to dominating value. Simplifying process reduces noise on the final output test case, without decreasing its code coverage. This is a brute force algorithm, that can have worst case performance t of $t = \frac{m*n}{2}$. Simplifying algorithm is only run on each test case after the genetic algorithm has completed so it is only scaled by the size of the whole result test suite and not by the generation count of the genetic algorithm. After simplifying algorithm has passed, it is added to the pool of generated cases. All the cases in this pool are used as penalty for similarity for all subsequent cases.

The proposed algorithm is run iteratively until predefined amount of test cases have been generated. Resulting test suite should then have test cases which all have significant coverage individually and as a collective, they should have even higher coverage than any one individual.

4. IMPLEMENTATION

This chapter goes through detailed implementation of the algorithm described in chapter 3. Programming language used for the implementation is Python. Source code for the whole algorithm, utility functions and interface class can be found in appendixes A through D.

4.1 Environment

Environment where target models were implemented and simulated was Simulink as it was the environment where all evaluation models were implemented. All of the target models were modified in such ways that their input could be read from MATLAB workspace as timeseries objects instead of input ports. Example of modifications done to models can be seen in Figure 12. These modifications enabled genetic algorithm execution to be implemented using Python and didn't require implementation to be implemented in MATLAB. Simulink models were setup to simulate with Fast Restart enabled, which reduced the time between each simulation run as model weren't rebuilt between runs.

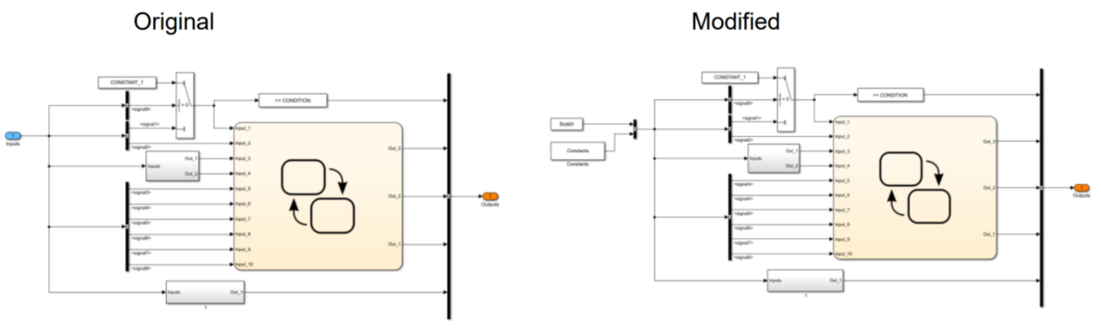


Figure 12. *Production model and modified model.*

Simulink has built-in tool which can used to extract different coverage metrics from simulated models. Having this functionality built-in to the environment removed the need for the genetic algorithm implementation to do analysis on the simulated model. The Simulink coverage tool was set to report MC/DC, Decision Coverage and Execution Coverage from all target models after each run.

An interface class was implemented with python, which handled naming, scaling, and populating the simulation input bus to the model. It also extracted coverage metrics from the target model. The interface class acted as an intermediary between Python and MATLAB. Additionally for every target model a signal map data structure was created, which defined signal ranges and types. Interaction between the model and the genetic algorithm is described in Figure 13.

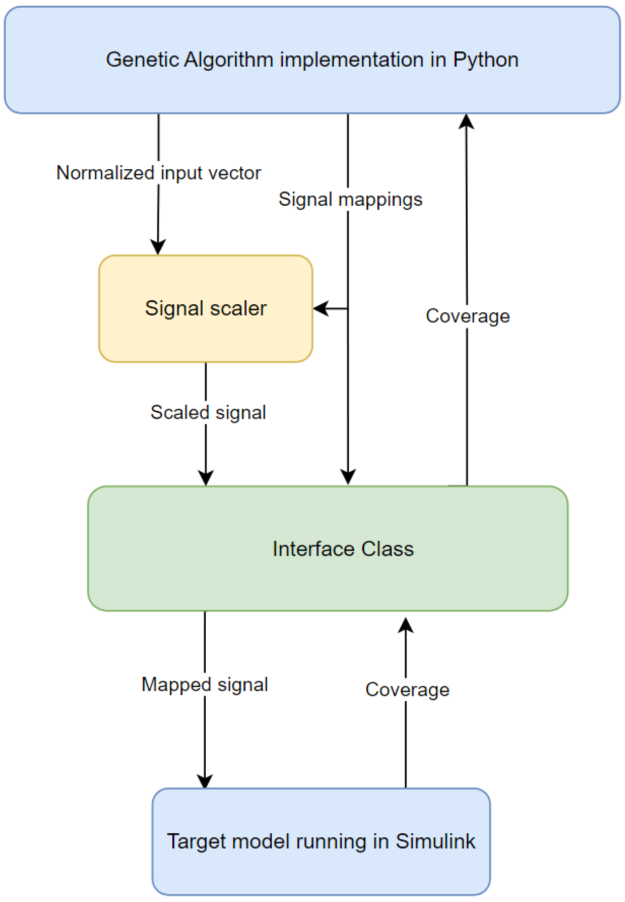
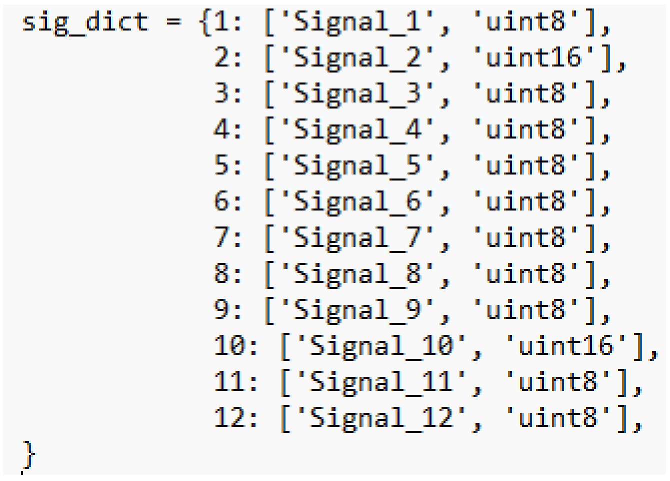


Figure 13. *Interface with target model and genetic algorithm*

4.2 Genetic algorithm implementation

4.2.1 Signal Mappings

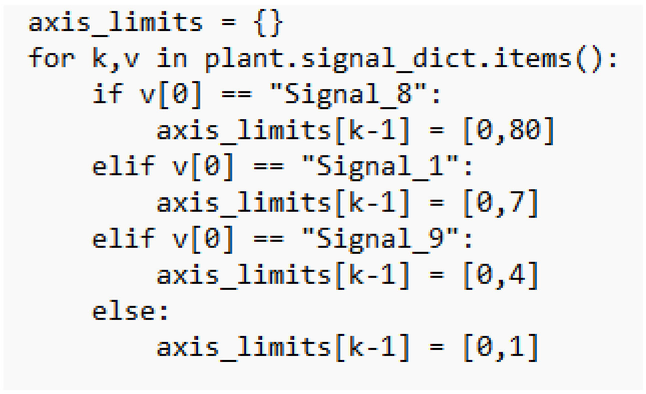
Every target model had mostly different signals from each other, so a signal mapping data structure was made for all models. It consists of an index, signal name and data type. Example can be seen on Figure 14. This signal mapping structure could then be referenced in python implementation to map the array outputted by the genetic algorithm to the Simulink signal bus.



```
sig_dict = {1: ['Signal_1', 'uint8'],
            2: ['Signal_2', 'uint16'],
            3: ['Signal_3', 'uint8'],
            4: ['Signal_4', 'uint8'],
            5: ['Signal_5', 'uint8'],
            6: ['Signal_6', 'uint8'],
            7: ['Signal_7', 'uint8'],
            8: ['Signal_8', 'uint8'],
            9: ['Signal_9', 'uint8'],
            10: ['Signal_10', 'uint16'],
            11: ['Signal_11', 'uint8'],
            12: ['Signal_12', 'uint8'],
}
```

Figure 14. *Example of signal mapping type data structure*

Based on the name of each signal, another data structure was made which contained the valid signal limits. These limits were used to create rules to scale the floating-point values between the closed interval $[0,1]$ so that the genetic algorithm outputted to a range that was valid for the model. Example of the limit definitions can be seen in Figure 15.



```
axis_limits = {}
for k,v in plant.signal_dict.items():
    if v[0] == "Signal_8":
        axis_limits[k-1] = [0,80]
    elif v[0] == "Signal_1":
        axis_limits[k-1] = [0,7]
    elif v[0] == "Signal_9":
        axis_limits[k-1] = [0,4]
    else:
        axis_limits[k-1] = [0,1]
```

Figure 15. *Example of signal mapping data structure*

Both arrays in Figure 16 show signals as columns of the array and rows as timesteps. Each sample has luminance based on its amplitude. Left side shows array with only floating-point values between 0 and 1, which the genetic algorithm is using to perform mutation and crossover. On the right-hand side, the same array is shown but with signal scaled to match the rules defined in Figure 15. This right-hand side array is inputted to the model and then simulated.

In summary, signal mappings enable the genetic algorithm to operate completely with signal values between 0 and 1. Per-model mappings enable interoperability of the algorithm for different models.

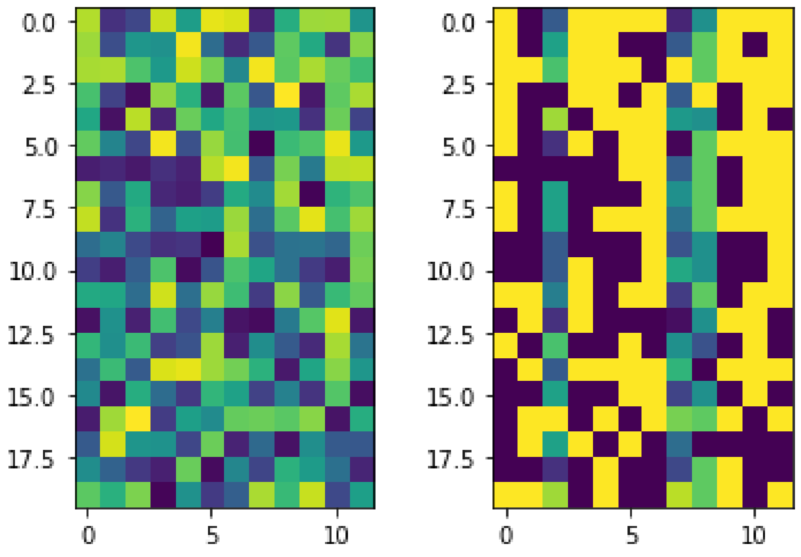


Figure 16. *Unscaled and scaled input array*

4.2.2 Parameters

During development there arose a need for the algorithm to function differently for different target models. So many aspects of the algorithm were made tuneable with parametrisation. Defining these parameters required some introspection to the target model and its functionality. For example, if there was some decision with a timer functionality, the simulation time must be longer than the timer so that the decision would ever be able change states. Parameters which could be defined for all models separately were:

- Model sample rate
- Initial population type
- Signal sample count
- Signal sample size
- Crossover type
- Population size
- Mating pool size
- Finished test case suite size.

Addition to the parameters above, there are also numerous other parameters that could be tuned to reach better performance for every target model, but it was decided to leave them as fixed values. Reasoning behind limiting tuneable parameters, was to keep usage of the algorithm simpler.

4.2.3 Fitness evaluation

Fitness scores consist of code coverage (MC/DC) and penalty score based on similarity to existing cases. To retrieve coverage score, a test vector is inputted to the interface class, which then transforms it to a timeseries signal in MATLAB workspace. This timeseries signal is then mapped to the input block and the model is simulated. After simulation, code coverage is retrieved from the model using the interface class.

Overwhelming majority of the execution time taken in fitness evaluation for each individual is spent on simulation time and calculating coverage. Only a small fraction is spent on penalty calculations, setting inputs and reading outputs. Therefore, the execution speed of the penalty function is not greatly affecting the time to convergence.

There exists one problem for ranking individuals based on their fitness scores, discrete code coverage. MC/DC coverage indicates percentage of executed decision in a way described in 2.3.5. Thus, complexity of a model is inversely proportional to the size of discrete value in MC/DC coverage. Penalty score on the other hand can have continuous values. Total fitness score is calculated by subtracting penalty score from the code coverage. Code coverage values are also logged independently from penalty scores to monitor performance over time.

Error metric for the case similarity penalty score is Euclidean distance of the two arrays begin compared. To enable Euclidean distance-based comparison on test vectors, they first need to be compressed to remove all time steps that don’t have any state changes. This is done to both existing test cases and the current test vector under training. If test arrays are different sizes, the shorter one gets resampled to match the longer one. Example of compressing logic of a test case can be seen in figure 16.

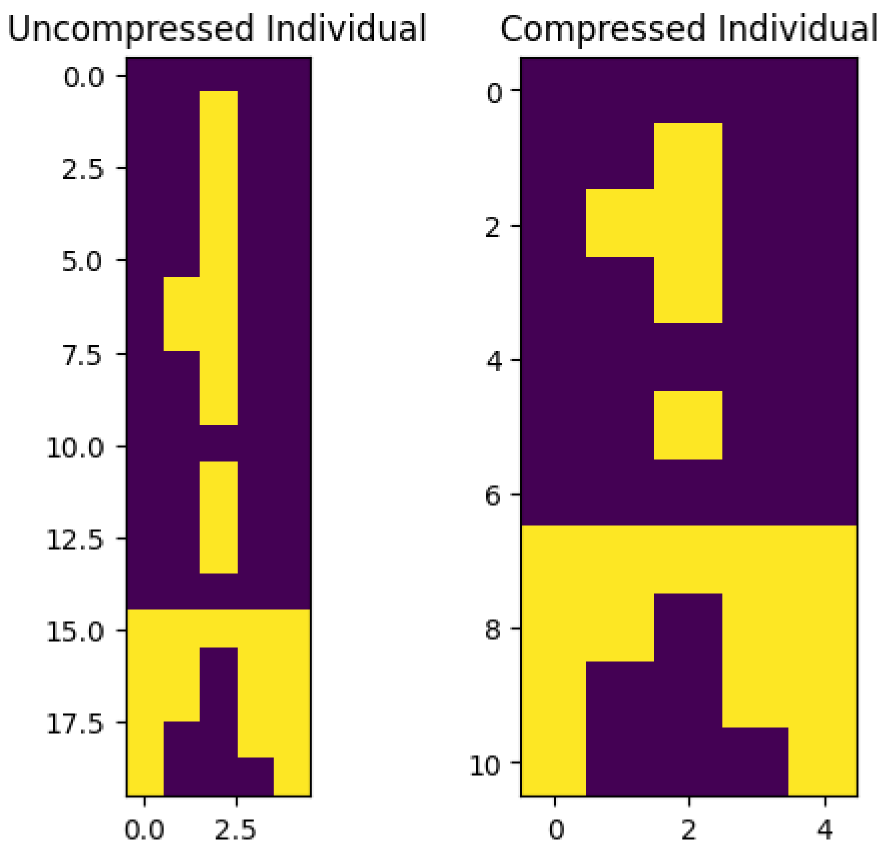


Figure 17. *Individual before and after logic compress*

4.2.4 Selection Process

Mating pool size is a parameter that defines how many of the top performing individuals in each generation get selected for the crossover process. Once the individuals have been selected, they are ranked based on their fitness scores. This is done so that all individuals can be assigned a selection probability. Directly using the fitness scores as selection probabilities would not be useful because fitness scores can be very close to each other or even same as code coverage increases in discrete steps. Selection probability is determined by fitting a gaussian probability density function (PDF) over the ordered individuals. This process is illustrated for six individuals in figure 17.

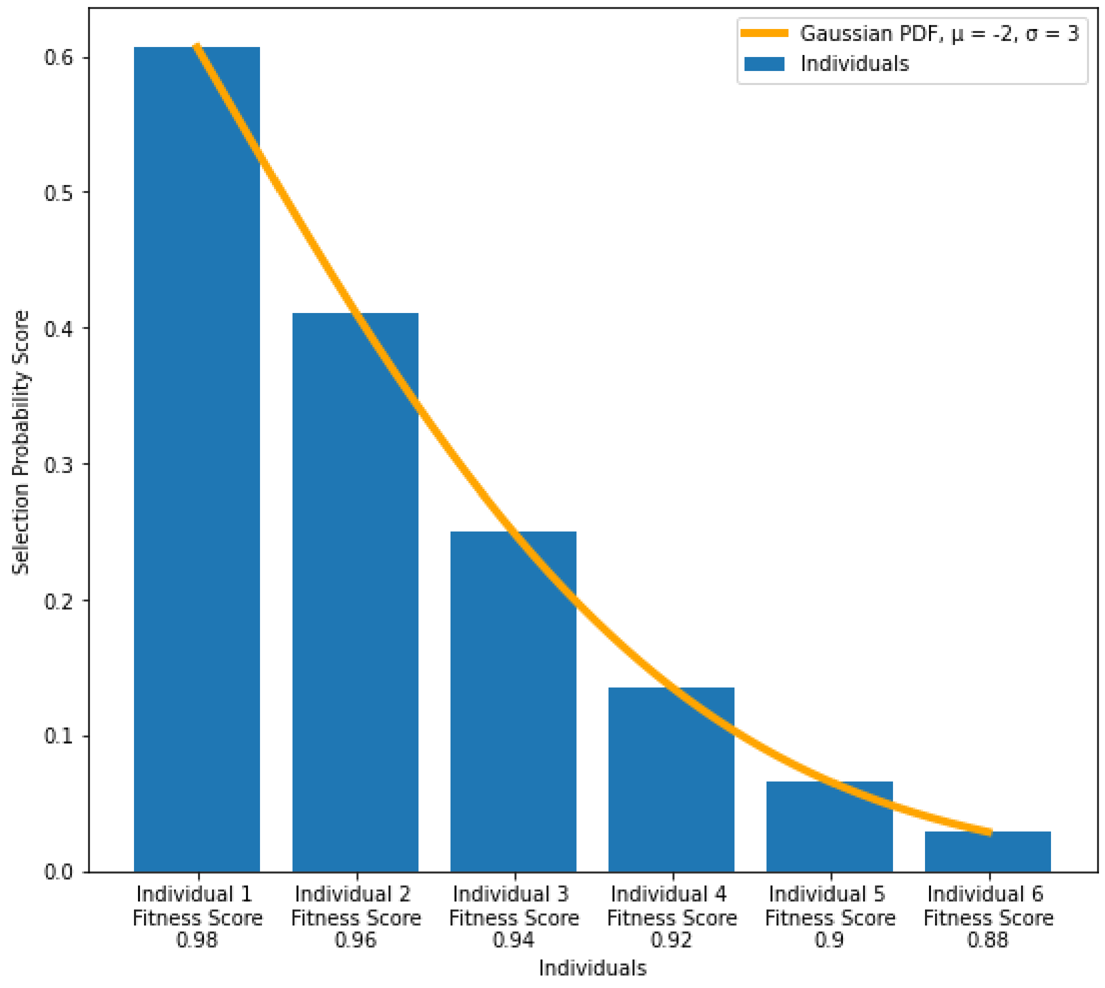


Figure 18. *Selection probability score for six individuals*

After obtaining probability scores for all individuals, scores are normalized. Selection for cross-over process is done by sampling from an array with individuals probability scores being the probabilities of individuals being picked.

When two individuals are selected for cross-over they can be either removed from mating pool, so all individuals get to be selected for cross-over, or they can be left into the pool for another round of selection. Also, there can be done multiple passes on the same mating pool to obtain more individuals for given mating pool. Mating pool size, mating rounds and exclusive mating are all tuneable parameters in the algorithm.

Predefined number of the top performing individuals are passed over from one generation to another after they have been selected for mating. This helps the algorithm to always keep the best performers in the mating pool so that there is no regression is performance of the top performers. Individuals whose performance is good enough to be selected for mating pool but not good enough to be passed on to the next generation are discarded after mating.

4.2.5 Crossover Process

An offspring is produced from two individuals that got selected from selections process. Crossover starts by creation of a bit mask that is based on the methods described in chapter 2.4.2. This bit mask is created as a one-dimensional array even though the individual is two-dimensional. This is done because a gene in this genetic algorithm implementation isn’t one value in the 2-D array that the individuals consist of, but one time step consisting of all the signals. Once the one-dimensional crossover mask is created, it is expanded to two dimensions along the signal count axis to match the size of the test case. Usage of K-fold or Uniform crossover can be selected with a parameter. Example of how the crossover process works for two example cases with 5-fold crossover can be seen on Figure 19.

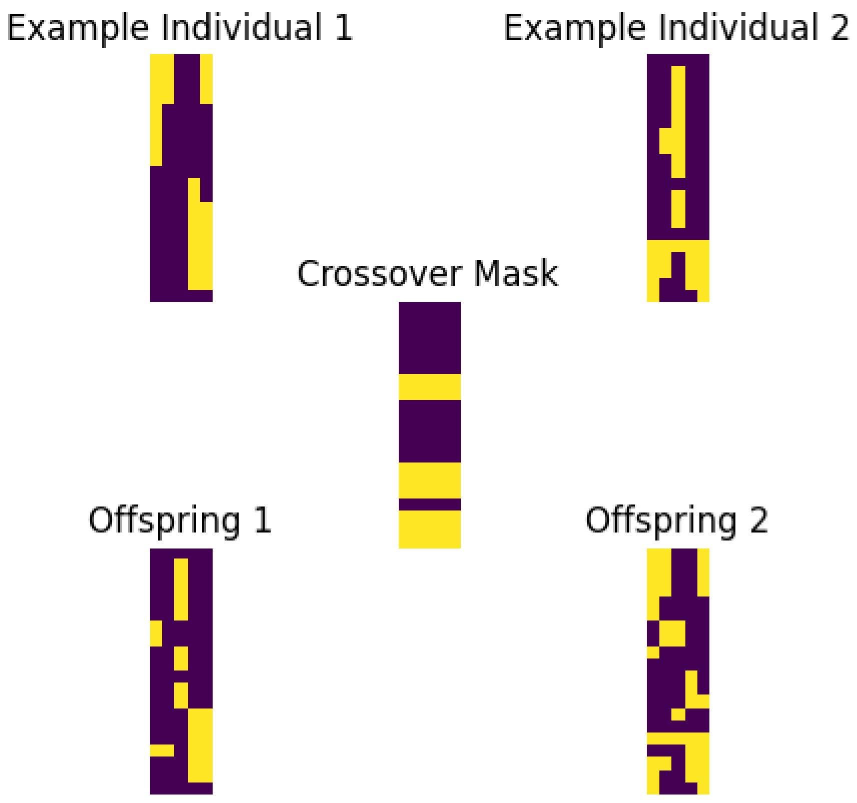


Figure 19. *Example of 5-fold crossover*

4.2.6 Mutation Process

Offspring creation described in previous chapter only has features from both parents and this process does not alter any of the existing genes (which are timesteps consisting of all signals). So, alterations to the gene pool is needed to achieve convergence past certain point.

A mutation percent parameter defines how much of the samples in the individual are changed to a new random value. These random samples are sampled from a uniform distribution. This value might not always change the state of a sample after the mapping process in the simulation, because the random value is sampled from the closed interval $[0,1]$ and might lay between the same interval as the previous value. For example, if a signal is a Boolean signal, its threshold for a true state is 0.5 and if a sample is changed from 0.2 to 0.3, no state change occurs. Example of a mutation process performed by the algorithm can be seen in Figure 20.

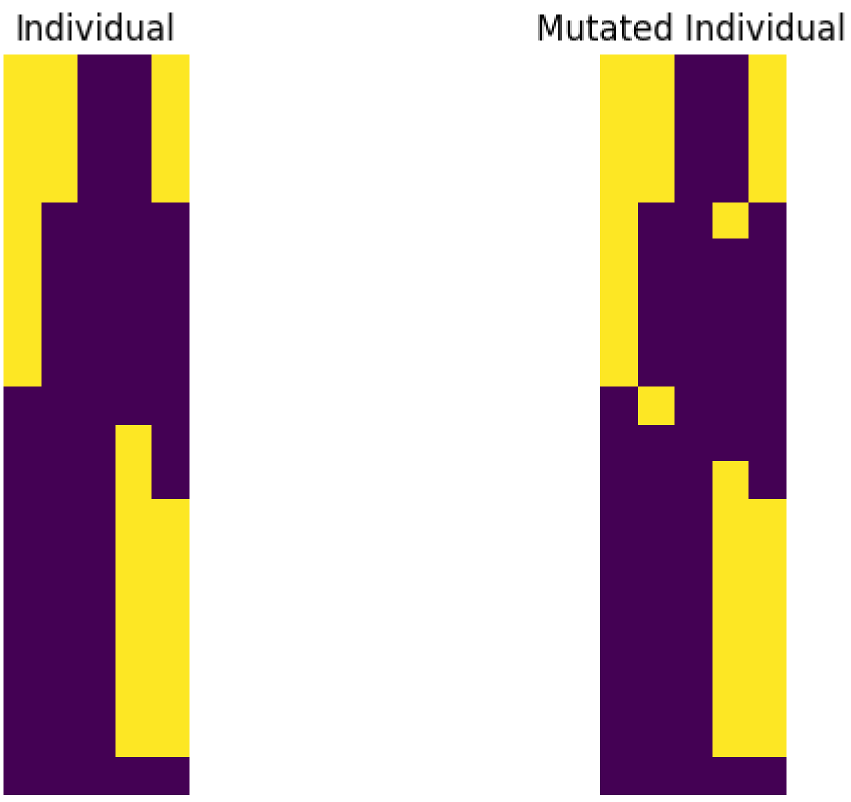


Figure 20. *Individual before and after mutation process*

5. PERFORMANCE ANALYSIS AND DISCUSSION

5.1 Target models

Performance of the created algorithm was evaluated on Simulink models that are in production, used in forest machines. Only a subset of models used in the control system of a forest machine was used. The different models under test were categorized to be the following types:

- Control logic with timers and only Boolean operator as inputs. (Model #1)
- Control logic with state machines, Boolean variables, timers, Enumerations, and sensor signals as inputs. (Model #2)
- Control logic with state machines containing timers, Boolean variables as inputs. (Model #3)
- Control logic with state machines, Boolean variables, and enumerations as inputs. (Model #4)

Selections criteria for the models to be used in performance evaluation, was to have control logic that had the possibility to calculate MC/DC coverage in Simulink. Omitted models consisted mostly of complex control algorithms that required some plant model response to be able to achieve satisfactory results or had such a long simulation time that it was not practical to use them for evaluation. Also because of the long runtime needed for the algorithm to converge, it was not feasible to test all available production models.

Maximum number of outcomes for MC/DC, decision and condition coverage are seen in Table 1. The number of decision coverage outcomes is higher for condition and decision than MC/DC for all models under test. This is because Simulink doesn't have MC/DC defined for all blocks. Performance analysis is done on MC/DC coverage. Table 1 gives some context to the complexity of the models outside of the blocks that have MC/DC coverage defined.

Table 1. *Model complexity descriptions.*

	Model 1	Model 2	Model 3	Model 4
Maximum number of MC/DC–coverage outcomes	23	30	29	15
Maximum number of Decision-coverage outcomes	48	145	42	42
Maximum number of Condition-coverage outcomes	50	120	58	48

5.2 Parameters

As described in chapter 4, the algorithm has multiple tuneable parameters that can affect the convergence speed and quality. Most of these were set to fixed values between different models and simulation runs to give some idea how the algorithm would perform in actual production environment. Some parameters require case by case tuning to even get the algorithm to simulate the evaluation models. The following parameters were needed to be adjusted for each model separately:

- Signal sample size
- Signal sample count
- Maximum generations
- Maximum number of plateauing generations

Parameter sweeps were run for mutation percentage and crossover type, to find suitable values that would perform well in all test models. Selection of these two parameters for closer introspection was the fact that modifying them doesn't affect the per generation simulation time. These parameter sweeps were done by generating only one test case and logging the coverage percentage throughout the process. By generating only one case, similarity penalty between the cases was not a factor and thus had no effect on convergence. Each parameter was run 10 times for each model to remove reduce the stochastic effect of the algorithm on the results. Average fitness score of each generation for every given parameter value in the sweep can be seen in Figure 21 and Figure 22.

Values used to perfrom the mutation parameter sweep were selected so that smallest was the smallest possible value that could be used for the mutation calculation. Largest value was selected to be 40%. Model #2 had less parameters because it was considerably more time consuming to simulate and would take unreasonable time to do same type of sweep as for other models.

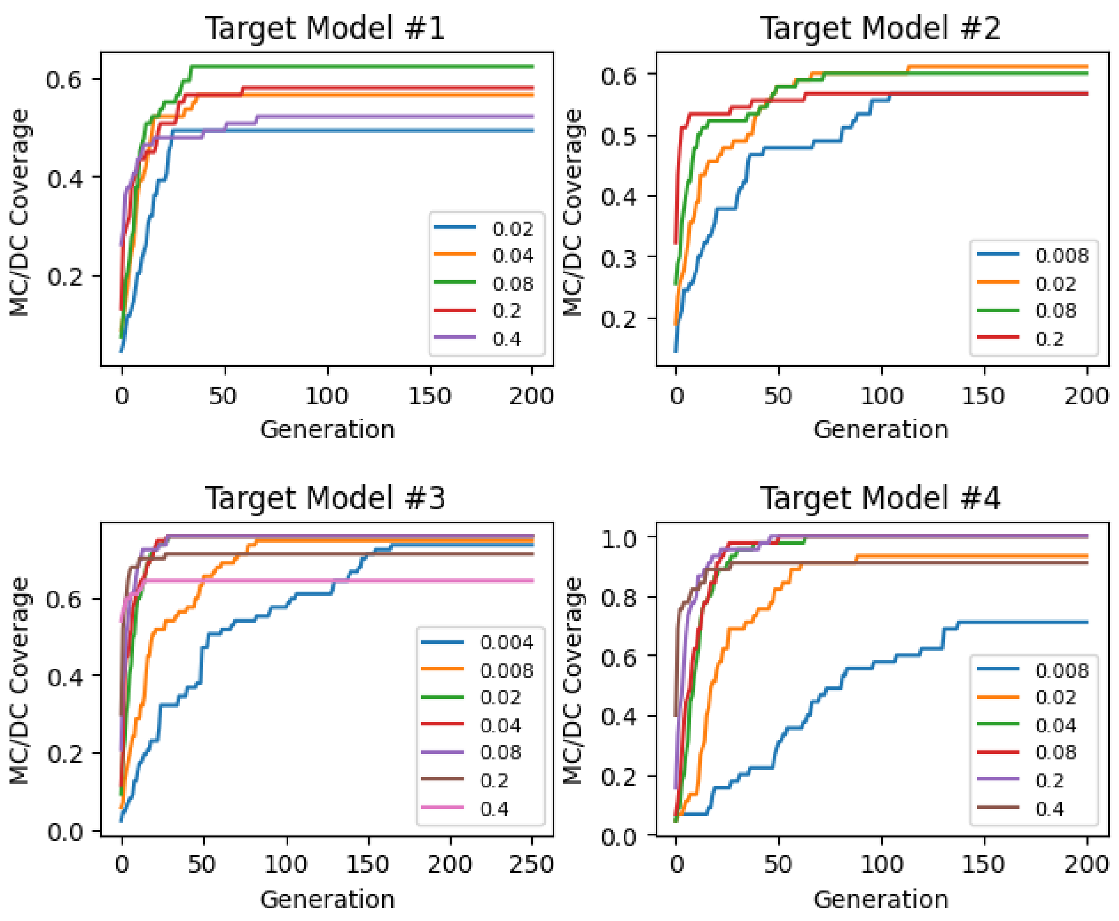


Figure 21. *Mutation sweep for all 4 test models*

Best performing mutation percent was deemed to be 8% for the models under test. It converged almost as quickly as higher percentages but was always equal or best in coverage performance.

Similar sweep was done for the crossover parameter, k-values for k-fold crossover were selected to be 2,4 and 8. Also uniform crossover was used, with the p-value of 0.5. Algorithm was run on all evaluation models with the same parameters. Fitness scores per generation can be seen in Figure 22. Based on the values shown Figure 22 K-value for k-fold-crossover was selected to be 2 for further evaluation.

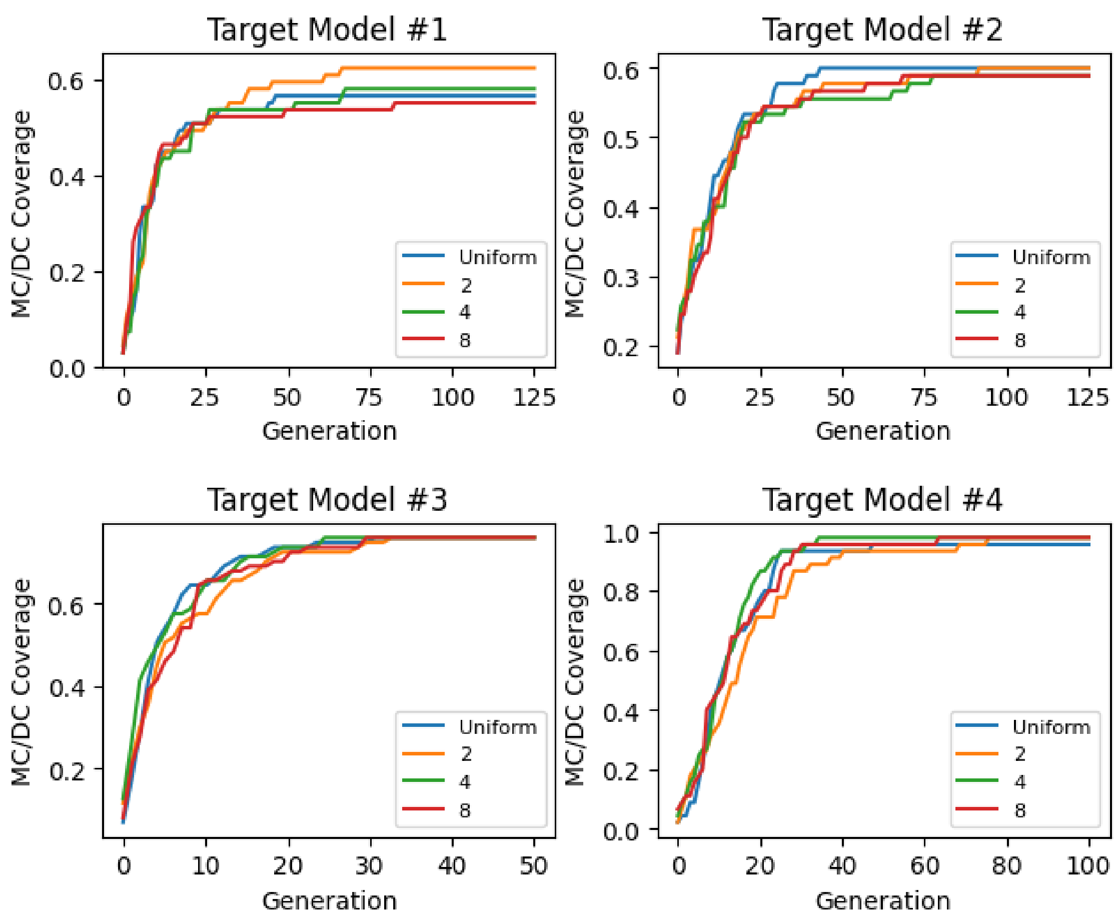


Figure 22. *Crossover sweep for all 4 evaluated models.*

5.3 Performance as a software development tool

Tests to evaluate the algorithms performance were done in such a way that it would reflect its usage as a software development tool. Size of the resulting test suite was set to 5 and consideration was used to limit the generation count so that the whole suite set would be generated in under 3 hours.

By using the optimal parameters found in chapter 5.2 for all 4 models, test suites were created and MC/DC coverage for each case for each generation was logged. This is visualized in graphs shown in figure 22.

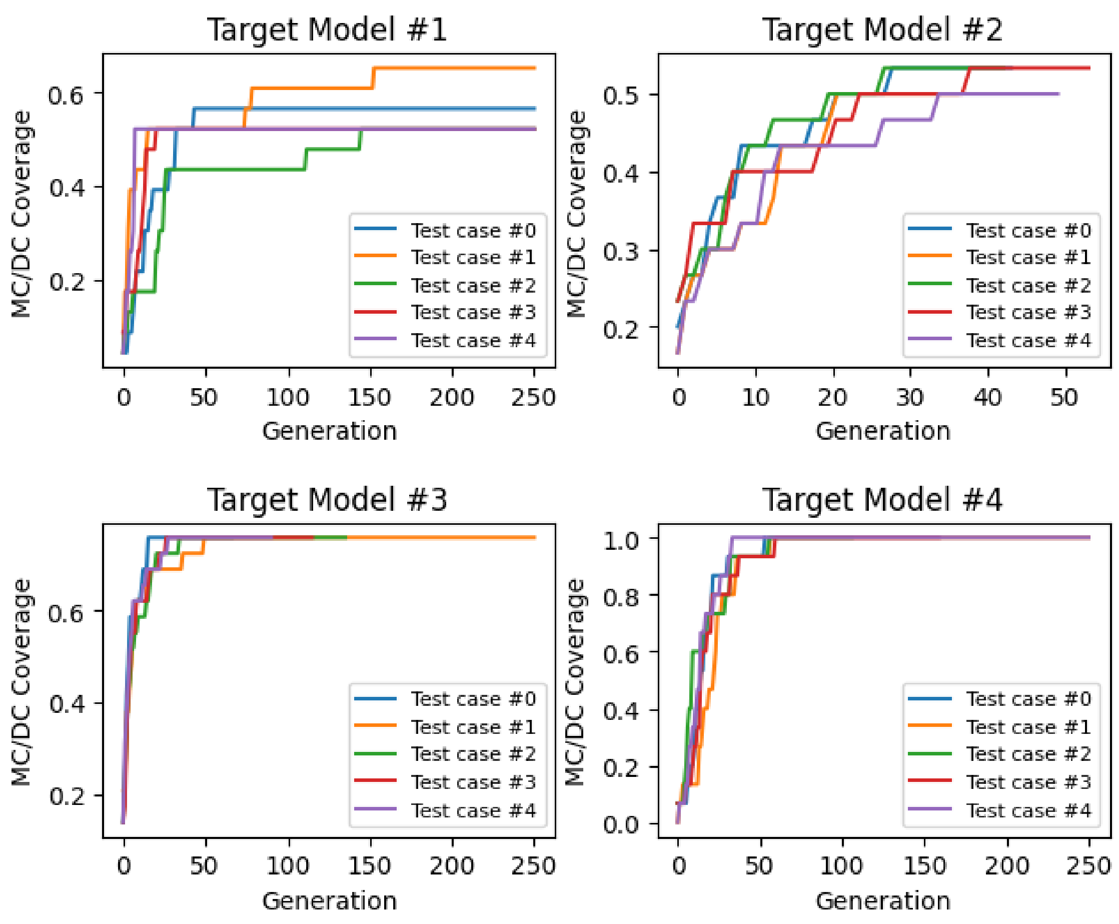


Figure 23. *Test suite coverage scores over generations*

5.4 Case similarity and simplification algorithm evaluation

The difference in logical paths that the cases in resulting test suite had, was evaluated by getting the cumulative coverage of the whole test suite. Meaning that if coverage of the test suite was higher than any individuals coverage by themselves, then as a collective, the test suite was better than any single individual. Cumulative coverage was evaluated on same generated test cases that were used to create Figure 23. This was done using feature in Simulink which would record cumulative coverage over multiple test cases. Minimum, maximum, and cumulative coverage for all test suites for all tested models is shown in table 2.

Table 2. *Minimum, maximum and cumulative MC/DC coverages for tested models.*

	Model #1	Model #2	Model #3	Model #4
Minimum case coverage	52.2%	50.0%	75.8%	100%
Maximum case coverage	65.2%	53.3%	75.8%	100%
Cumulative coverage	69.6%	56.6%	75.8%	100%

To evaluate how well the simplifying part of the algorithm performed, a test case for each model was generated. Output from the algorithm was stored before and after the simplifying was applied. Results from this operation can be seen in appendix E.

To quantify the data seen in appendix E, state changes in each signal was calculated for all generated cases before and after the simplification algorithm. All signal changes in one test case were summed up to calculate one value that represents complexity of the test case. Results can be seen in table 3.

Table 3. *Test case state changes before and after applying the simplification algorithm.*

	Model #1	Model #2	Model #3	Model #4
Original test case state changes	49	819	238	141
Simplified test case state changes	12	31	60	26

5.5 Discussion

5.5.1 Convergence

Results of parameter sweeps shown in Figure 21 and Figure 22, show that the algorithm could converge with all tested parameter values. This shows that mutation rates and crossover types could be used as fine-tuning parameters to improve the algorithms performance on a given model.

Main differentiation between results with mutation parameter sweep shown in Figure 21 was the convergence speed. Generally, higher mutation rates converged faster, which indicates that there are actually quite many “routes” to good convergence and the algorithm didn’t necessarily need to find one of the few good ones to progress. Interestingly, lower mutation percentages also stopped converging after a certain point, even though there was still much room for improvement. This most likely is caused by the need to introduce a lot of changes in one generation to “break through” the barrier of one discrete code coverage step listed in Table 1. Having lower mutation rates could not introduce these changes via mutation in one generation but also needed crossover process to introduce new patterns to the solution. This is an issue that genetic algorithms should overcome given enough generations to iterate over the solution space, but in this case, generation count became limiting factor in some tests runs with low mutation rate. Too high mutation rate on the other hand most likely introduced too much variation to a given solution per generation. This way, most of the good "features" inherited from previous generations were lost to the mutation process, which then limited the quality of the final result. Therefore, it can be said that good balance in mutation rate needs to be found for optimal convergence.

Based on the data in Figure 22, three of the four evaluated models had almost identical convergence when comparing the different crossover types. Only exception begin model #1, that had some divergence between the graphs. This could be explained by the fact that model #1 had only 10 samples of all signals, so higher crossover values would be more significant percentage of the samples and thus "brake up" too much of the logical paths that form a high performing test case. Models which had higher number of samples defined per test case would not suffer from this effect in similar fashion. For example, 8-fold crossover would mix up 80% of the states in model #1 and only 8% in model #2. Uniform crossover should introduce lot of stochasticity in models that required inputs with more time samples and thus negatively affect the performance. This was not the case given the results shown in Figure 22. One reason for this was deemed be the fact that evaluation models were relatively simple and added stochasticity originating from crossover didn't hinder the performance too much.

Evaluating performance as software development tool had case similarity penalty enabled and algorithm produced multiple cases as seen in Figure 23. Even with case similarity penalty applied, convergence speed and quality were mostly consistent with most models. Exception to this is model #1, which had some deviation from within the test suite. Having all cases mostly converge within roughly the same speed and quality hints that the case similarity penalty is not able to penalize the algorithm so that it would create different logical paths for different cases within the same test suite. Another explanation could also be that the algorithm could achieve the measured coverage with diverse set of solutions in evaluations models.

In chapter 2.4 metaheuristic algorithms were deemed to improve solution in two ways: exploration and exploitation. Based on the conclusions above, the implemented algorithm was utilizing more the exploitation rather than exploration. This is because the implemented algorithm's function can be thought to be exploration when trying to optimize the case similarity penalty as the value is continuous and even tiny gains are possible. So "exploring" these nearby values were not shown to affect the result in any meaningful way. On the other hand, exploitation could be thought of "breaking through" to the next discrete step of code coverage and this was shown to work well in Figure 21 and Figure 22.

Utilizing genetic algorithms in such way that could carry good features in test cases over generations can be determined to be achieved. This is most likely because of the design choices of defining a gene to be a timestep of all signals and mutation to only alter part of a gene. This meant that good logical sequences in an individual get carried over generations, but they can be still improved further by mutation.

5.5.2 Case similarity and simplification algorithm

Algorithm should penalize too much similarity between the test cases in a generated suite, to broaden the search space that cases together can cover. This was largely not achieved as only 2 of the 4 generated test suites produced cumulative coverage higher than the maximum coverage of any one case in that test suite. For models #1 and #2, gain in cumulative coverage was only 4.34% and 3.33% respectively. The maximum MC/DC outcomes from model #1 is 23, so the cumulative coverage of the test suite achieved only one more MC/DC outcome that any single test case. Same is true with model #2 with the number of MC/CD outcomes of 30 and increase of 3.33%.

For model #4, 100% MC/DC coverage was achieved even for the worst performing case, so cumulative coverage couldn’t increase coverage further. This means that test suite generation is not even necessary as only one test case could achieve 100% coverage. Model #3 achieved same coverage with minimum, maximum and cumulative coverages. This result implies that the model under test most likely had some amount of “dead logic” that could not be exercised with the inputs provided to the model or condition statements with timers contained in the model could not be satisfied in any case.

Maximum performance of the algorithm in model #2 was actually lower than what was achieved when doing parameter sweeps, shown in Figure 21 and Figure 22. Reason for this is that as seen from Figure 23, coverage score on model #2 was still converging before the algorithm was stopped for every case. Reason for premature stoppage of the algorithm on model #2 was predefined 3-hour time limit on execution, which was reached during evaluation. Extending the allowed simulation time would most likely would have resulted in better performing test cases for model #4 but it would lose its utility as software development because the runtime would grow to be too long for many practical use cases.

Main culprit for bad performance in case similarity penalty was most like the usage of Euclidean distance as the metric. It couldn’t create pressure for the algorithm to generate different logical paths in the evaluation model set for different test cases. It would penalize cases for having signal activation at the exact sample, this however does not consider that same features can still exist, just shifted in time.

As seen in Table 3 and in appendix E, simplifying algorithm can reduce complexity of the generated test case greatly. The amount of simplification that could be applied to a generated test case is shown to be most effective cases where there are large number of samples. For example, in model #4, all of the relevant changes were able to be achieved within the first 6 samples and rest of the samples were able to be simplified, thus making the test case more human readable.

6. CONCLUSIONS

To answer the research question on "*How genetic algorithms can be used in test case generation with model-based system development?*", the following conclusions can be made: A way to generate regression test suite to be used in CI-pipeline of model-based software development process with genetic algorithms, was successfully implemented in this thesis. Implemented algorithm showed significant convergence towards high code coverage and could theoretically be implemented to a CI-pipeline.

More specific requirements for the algorithm described in chapter 3.1 were achieved in for most parts, as seen in performance evaluation in chapter 5. Simplification part of the algorithm was deemed to increase readability of generated cases significantly. Higher collective coverage for a generated suite compared to one case was not deemed to be achieved, as the collective coverage was only marginally higher than one single case. This could be only due to the stochastic nature of the algorithm.

Usage as tool, part of the CI-process, was deemed to be possible but impractical. Biggest reason for this was the parametrisation needed to be done for every single model and the effect of these parameters on convergence speed and quality. Having this level of insight for a software development tool for every software engineer, who would use the CI-pipeline, would most likely not be cost-efficient. Big downside is also the long run time of the algorithm, which was already measured in hours in the limited evaluation model set. For more complex models, runtime could be measured in days rather than hours. Also interfacing with the models would require more large amounts of modifications to an existing MATLAB-based CI-pipeline or complete reimplementation of the algorithm directly in MATLAB instead of python.

Research question about *"How well do genetic algorithms perform when generating tests for embedded control system software?",* can be answered with the following conclusions: It was shown that the proposed and implemented algorithm could, in best case scenario, fully exercise a production model used in forest machines and achieve 100% code coverage, given that the metric was MC/DC-coverage. Even at worst case, 50% MC/DC coverage was achieved. However, extrapolating that the performance of the algorithm would be the same for the whole code base of a forest machine was deemed to be impossible, given the small set of evaluation models. Also, it is important to keep in mind that all of the control software of a modern forest machine cannot be done with only state-machine based control logic.

The final research question was the following *"What type of models work best with test case generation with genetic algorithms?".* To answer this question, the following conclusions can be made: Performance of the algorithm on different types of models was analysed it was shown that in every case, there was significant convergence towards high code coverage. Best performance was achieved on a model which had no timers or sensor signals as input and was overall the least complex from the evaluation set. Model where the algorithm achieved worst performance was the most complex and it contained state machines, Boolean variables, and enumerations as inputs. Overall, it can be said that models which have some temporal dependency and overall higher complexity, have negative effect on the convergence of the algorithm implemented in this thesis.

6.1 Future improvements

Algorithm implemented in this thesis still leaves a lot of room for possible improvements. By addressing previously mentioned major shortcomings, a functioning tool for a Model-Based software development CI-process could be implemented and integrated to produce higher quality software. In this sub-chapter some of the possible future developments are explored.

To improve performance in generating a wider coverage test suite, better case similarity penalty would need to be implemented. This penalty metric would need to take into consideration changes of inputs, in addition to their states. Intuitively, some kind sliding window type of implementation might be a good candidate, as it would be time agnostic. In other words, it doesn't often matter "where" the features in the test cases are, as long as they are present, they could be compared to each other.

Going one step further in the MBSD code generation pipeline shown in Figure 3, from MIL to SIL and executing the algorithm with generated code, would most likely result in significant decrease in per generation runtime. This would enable much larger population sizes and more generations for the same total runtime as the implementation done in this thesis. Having significantly faster simulation time would open the possibility to test wider range of parameters and their effect, which could bring further improvements in convergence speed and quality.

Further testing could be done to see how much the implemented algorithm could increase code coverage of already existing test suites. Mutation testing [59], could be one additional way of assessing generated test case quality. Additionally, to fully assess the usage of the work done in this thesis, data from complete model-based software development project would need to be gathered, where a tool made with the implemented algorithm is in use. From this data, it could be measured how much the generated regression test suites catch errors more than human defined test suites and further cost-benefit analysis could be done.

REFERENCES

[1] S. Abitha, Embedded System Paper Document, International Journal of Engineering Research & Technology (IJERT), Special Issue, 2018

[2] A. Forrari, Embedded Control System Design, A Model Based Approach, 2013

[3] P. Chang, Industrial control systems, Advanced Industrial Control Technology, 2010

[4] J. Shepard, How does embedded software work?, MicrocontrollerTips, Accessed: Jan-15 2023, Available: https://www.microcontrollertips.com/how-does-embedded-software-work/

[5] A. Kejariwal, F. Orsini, On the Definition of Real-Time: Applications and Systems, IEEE Trustcom/BigDataSE/ISPA, 2016

[6] D. Fontanelli, L. Greco, L. Palopoli. Soft real-time scheduling for embedded control systems. Automatica, 49(8), 2330–2338, 2013

[7] Z. Jiang, T. G. Lewis, W. Jackson, R. L Wilson, Scheduling in hard real-time applications. IEEE Software, 12(3), 54–63, 1995

[8] O. Olodeye, A. Akinwole, N. A. Yekini, A. O Akinade, Overview of Embedded Systems & Its Applications, 3RD INTERNATIONAL ACADEMIC CONFERENCE, 2022, Available: https://www.researchgate.net/publication/361562662_OVERVIEW_OF_EMBEDDED_SYSTEM_ITS_APPLICATION

[9] J. Wang, Real-Time Embedded Systems, John Wiley & Sons, Inc, 2017

[10] Z. H. Khan, A. Khan, Perspectives in Automotive Embedded Systems, International Symposium on Automotive and Manufacturing Engineering (SAME), SMME, NUST, Islamabad, Pakistan, Nov 2015

[11] L. B. Lucia, G. Patti, L. Leonardi, A Perspective on Ethernet in AutomotiveCommunications — Current Status and Future Trends, Appl. Sci. 2023,13,1278.

[12] J. Eker, Flexible Embedded Control Systems: Design and Implementation, Department of Automatic Control, Doctoral Thesis, Lund Institute of Technology, 1999

[13] C. Haskins, A historical perspective of MBSE with a view to the future, INCOSE International Symposium, 21(1), 493–509, 2011

[14] C. E Dickerson, D. Marvis - A Brief History of Models and Model Based Systems Engineering and the Case for Relational Orientation, in IEEE Systems Journal 7, pp. 58 1-592, April 2013

[15] What is Model Based Software Development? [Online], Available: https://www.lifecycleinsights.com/tech-guide/model-based-development/, [Accessed: 10-May-2022]

[16] Why Adopt Model-Based Design?, MathWorks, White paper, Available: https://www.mathworks.com/content/dam/mathworks/white-paper/why-adopt-model-based-design-white-paper.pdf

[17] M. Bialy, Handbook of System Safety and Security || Software Engineering for Model-Based Development by Domain Experts, 2017, pp. 39-46.

[18] N. Srinivas, N. Panditi, S. Schmidt, R. Graffels, MIL/SIL/PIL Approach A new paradigm in Model Based Development, Continetal Corporation, Engine Systems, Available: https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/solutions/automotive/files/in-expo-2014/mil-sil-pil-a-new-paradigm-in-model-based-development.pdf, [Accessed: 31-May-2022]

[19] How to Implement a Continuous Integration Workflow with Model-Based Development (MBD), dSPACE, Educational Material, Available: https://www.dspace.com/en/pub/home/learning-center/recordings/how-to-implement-a-continuous-.cfm. [Accessed: 16-Jan-2023]

[20] Continuous Integration for Verification of Simulink Models, MathWorks, Technical Articles and Newsletters, Available: https://se.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html, [Accessed: 16-Jan-2023]

[21] T. Kelemenová, M. Kelemen, L. Mikova, V. Maxim, E. Prada, T. Lipták, F. Menda, Model Based Design and HIL Simulations, American Journal of Mechanical Engineering, Vol. 1, No. 7, pp. 276-28, 2013

[22] P. Jianto, Software Testing, 18-849b Dependable Embedded Systems, Carnegie Mellon University, 1999, Available: http://users.ece.cmu.edu/~koopman/des_s99/sw_testing/

[23] IBM, What is software testing?, Availabe: https://www.ibm.com/topics/software-testing, [Accessed: 12-May-2022]

[24] K. Naik, P. Tripahty, Software Testing and Quality Assurance, Theory and Practice, 2008, Available: http://www.softwaretestinggenius.com/download/staqtpsn.pdf

[25] L. Gren, V. Antiyan, On the Relation Between Unit Testing and Code Quality, 43rd Euromicro Conference on Software Engineering and Advanced Applications, 2017

[26] J. Shcmitt, CircleCI, Unit testing vs integration testing, [Online], https://circleci.com/blog/unit-testing-vs-integration-testing/ [Accessed: August-8-2022]

[27] D. Graham, E. Veenendaal, I. Evans, R. Black, Foundations of Software Testing, ISTQB Certification, pp. 52-53, 2012

[28] What Is Regression Testing? Definition, Tools, Method, And Example, [Online] https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/ [Accessed: 30-June-2022]

[29] S. Nidhra, J. Dondeti, Black Box and White Box Testing Techniques – a Literature Review, International Journal of Embedded Systems and Applications (IJESA) Vol.2, No.2, June 2012, Available: https://asset-pdf.scinapse.io/prod/2334860424/2334860424.pdf

[30] What is Code Coverage Analysis, Linode LLC., Development Guide, [Online], https://www.linode.com/docs/guides/what-is-code-coverage-analysis/ [Accessed: 6-August-2022]

[31] A Practical Tutorial on Modified Condition/Decision Coverage, Nasa, TM-2001-210876, May 2011

[32] A. Rajan, M.P.E. Heimdahl, M.W. Whalen, The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage, ICSE '08: Proceedings of the 30th international conference on Software engineering, 2009, Available: http://se.inf.ethz.ch/old/teaching/2009-S/0276/slides/fiva.pdf

[33] Types of Model Coverage, Mathworks, Accessed: 3. Jan. 2023, Available: https://se.mathworks.com/help/slcoverage/ug/types-of-model-coverage.html

[34] L. Bianchi, M. Dorigem L. M. Gambardella, W. J. Gutjahr, A survey on metaheuristics for stochastic combinatorial optimization, Nat Comput (2009) 8:239–287, Available: http://doc.rero.ch/record/319945/files/11047_2008_Article_9098.pdf

[35] That's So Meta, Meriam-Webster, Available: https://www.merriam-webster.com/words-at-play/meta-adjective-self-referential [Accessed: 9-August-2022]

[36] C. Blum, A. Roli, Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, ACM Computing Surveys, Vol. 35, No. 3, September 2003, pp. 268–308

[37] M. Abdel-Basset, , L. Abdel-Fatah, & A. K. Sangaiah, Metaheuristic Algorithms: A Comprehensive Review. Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications, 185–231, 2018.

[38] I. H. Osman, Meta-heuristics. Metaheuristics: Theory and Applications. 1-21. 1996.

[39] E. Pesch, F. Glover, TSP Ejection Chains, Discrete Applied Mathematics 76, 165-181, 1995.

[40] I. Fister Jr, X.-S. Yang, I. Fister, J. Brest, D. Fister, A brief review of nature-inspired algorithms for optimization, Elektrotehniski Vestnik, 80(3): 1–7, 2013

[41] H. A. Loáiciga, M. Solgi, O. Bozorg-Haddad, Meta-heuristic and Evolutionary Algorithms for Engineering Optimization, pp. 54, 2017

[42] Genetic Algorithms, MathWorks, Available: https://se.mathworks.com/discovery/genetic-algorithm.html [Accessed: 1.8.2022]

[43] R. M. Firedberg, A Learning Machine: Part 1, IBM Journal of Research and Development, 2, 2-13, 1958.

[44] J. H. Holland, Outline for a Logical Theory of Adaptive Systems, Journal of the ACM, Volume 9, Issue 3, pp. 297–314, 1962.

[45] T. Alam, S, Qamar, A. Dixit, M. Benaida, " Genetic Algorithm: Reviews, Implementations, and Applications.", International Journal of Engineering Pedagogy (iJEP), 2020

[46] H. Kour, P. Sharma, P. Abrol, “Analysis of fitness function in genetic algorithms”, Journal of Scientific and Technical Advancements, Volume 1, Issue 3, pp. 87-89, 2015.

[47] A. J. Umbarkar, P. D Sheth, Crossover Operations in Genetic Algorithms: A Review, ICTACT Journal on Soft Computing, October 2015, Vol: 06, Issue: 01

[48] A. P. Engelbrecht, Computational Intelligence, An Introduction, Second edition, 2007, Available: https://papers.harvie.cz/unsorted/computational-intelligence-an-introduction.pdf

[49] F. Saglietti, N. Oster, F. Pinte, White and grey-box verification and validation approaches for safety- and security-critical software systems, information security tec hnical report 13 (2008) 10–16

[50] A. Sharma, R. Patani, A. Aggarwal, Software Testing Using Genetic Algorithms, International Journal of Computer Science & Engineering Survey (IJCSES) Vol.7, No.2, April 2016, Available: https://aircconline.com/ijcses/V7N2/7216ijcses03.pdf

[51] D. Kumar, M, Phogat, Genetic Algorithm Approach For Test Case Generation Randomly: A Review, International Journal of Computer Trends and Technology (IJCTT), Vol 49, No. 4, July 2017, Available: https://ijcttjournal.org/2017/Volume49/number-4/IJCTT-V49P134.pdf

[52] R. P. K. Bhatia, Test case Optimization using Genetic Algorithm, IJRAR December 2018, Volume 5, Issue 4, 2018

[53] R. Zhao, S. Lv, Neural-Network Based Test Cases Generation Using Genetic Algorithm, 13th IEEE International Symposium on Pacific Rim Dependable Computing, 2007

[54] Y. Jin, A comprehensive survey of fitness approximation in evolutionary computation. Soft Computing, 9(1), 3–12, 2003.

[55] A. Ratle, Optimal sampling strategies for learning a fitness model. Proceedings of the 1999 Congress on Evolutionary Computation-CEC, 1999

[56] A. Tamizharasi, P. Ezhumalai, Genetic-based Crow Search Algorithm for Test Case Generation, International Transaction Journal of Engineering, Management, & Applied Sciences & Technologies, 2022, 13(4), 13A4K, 1-11, Available: https://tuengr.com/V13/13A4KM.pdf

[57] R. Hametner, D. Winkler, T. Östricher, S. Biffl, A. Zoitl, The Adaptation of Test-Driven Software Processes to Industrial Automation Engineering, 8th IEEE International Conference on Industrial Informatics, 2010

[58] J. Oh, M. Harman, S. Yoo, Transition coverage testing for simulink/stateflow models using messy genetic algorithms, Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation – GECCO, 2011

[59] L. T. M. Hanh, B. T. Nguyen, T. T. Khuat, Survey on Mutation-based Test Data Generation, International Journal of Electrical and Computer Engineering (IJECE), Vol. 5, No. 5, October 2015, pp. 1164~1173

[60] L. Halduraim, T. Madhubala, R. Rajalakshmi, A Study on Genetic Algorithm and its Applications, International Journal of Computer Sciences and Engineering, Vol.-4(10), Oct 2016,

[61] M. Ivanković, G. Petrović, R. Just, G. Fraser, Code Coverage at Google, 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’19), 2019

[62] V. Khorikov, Unit Testing, Principle, Practices and Patterns, Manning Publications Co, 2020, Availabe: https://sd.blackball.lv/library/unit_testing_(2020).pdf

[63] D. Kamma, P. Maruthi, Effective Unit-Testing in Model-Based Software Development, AST 2014: Proceedings of the 9th International Workshop on Automation of Software Test, 2014

APPENDIX A: GENETIC ALGORITHM CODE

```
import numpy as np
from utility_functions import Penalty
from scipy import signal
import time

class GA:
    def __init__(self,
                plant,coverage_type="MCDC",
                time_step=0.01,
                sample_time=60,
                mutation_percent=0.02,
                inclusive_mating=False,
                crossover_type=1,
                mating_population_size = 6):
        self.mutation_percent = mutation_percent
        self.existing_cases = []
        self.score_graph = []
        self.time_step = time_step
        self.sample_time = sample_time
        self.coverage_type = coverage_type
        self.plant = plant
        self.inclusive_mating = inclusive_mating
        self.k_fold = crossover_type
        self.mating_population_size = mating_population_size

    def get_coverage_function(self):
        if self.coverage_type == "MCDC":
            return self.plant.get_MCDC
        elif self.coverage_type == "decision":
            return self.plant.get_decision
        elif self.coverage_type == "execution":
            return self.plant.get_execution

    def get_init_scores(self, current_pop, formatter, plant, disp):
        """
        Get fitness score on initial population
        """
        return  np.array(self.runSingleTestCasePopulation(current_pop,  for-
matter, plant, disp))

    def gaussian(self, x, mu, sigma, scale):
        return scale*np.exp(-(x-mu)**2/(2*sigma**2))

    def create_mating_pool_probs(self, pool):
        x = np.linspace(0,1,pool.shape[0])
        y = self.gaussian(x,0,1,1)
        return y / np.sum(y)

    def mating_pool(self, pool):
        """
        Select from mating pool
        """
        probs = self.create_mating_pool_probs(pool)
        same = True
        if(self.inclusive_mating):
            pool = pool.copy()
```

```
                while same:
                    choices = np.random.choice(pool, 2, p=probs)
                    if choices[0] != choices[1]:
                        same = False
                pool = pool[pool != choices[0]]
                pool = pool[pool != choices[1]]
            else:
                while same:
                    choices = np.random.choice(pool, 2, p=probs)
                    if choices[0] != choices[1]:
                        same = False
            return pool, choices

        def get_k_point_crossover_mask(self, points, shape):
            """
            Returns a binary mask for crossover, given number of points and size
of the array
            """
            #1-Point mask
            co_mask = np.ones(shape)
            if points == 1:
                    fold_point = np.random.randint(0,  shape, size=points)
                    np.put(co_mask,  np.arange(fold_point,  shape,  1),  np.ze-
ros(fold_point))
            # K-point mask
            else:
                fold_points = np.sort(np.random.randint(0,  shape, size=points))
                #np.put(co_mask,  np.arange(point,  ind_2.shape[0],  1),  np.ze-
ros(ind_2.shape[0]-point))
                for count, point in enumerate(fold_points):
                    if count%2 == 0:
                        if count == 0:
                            np.put(co_mask,  np.arange(0,  point,  1),  np.ze-
ros(point))
                        else:
                            np.put(co_mask,       np.arange(fold_points[count-
1],point,1), np.zeros(point - fold_points[count-1]))
            return co_mask

        def get_uniform_mask(self, threshold, shape):
            """
            Returns uniformly sampled and thresholded boolean array
            """
            uniform_array = np.random.uniform(size=shape)
            return (uniform_array > threshold)

        def train_generation(self, current_pop, scores, formatter, plant, disp):
            """
            Trains one generation of solutions with k-point crossover. Returns
population with best performing individual at
            index 0.
            """
            k_fold = self.k_fold
            # Parameters for mating pop and direct transferred individuals
            mating_pop_size = self.mating_population_size
            direct_pass_size = 2

            # Initialize lists for populations, need to do copy
            current_pop_tmp = []
```

```
          current_pop_tmp = current_pop.copy()
          current_pop = []

          # Order population based on individuals fitness score
          idx = np.flip(np.argsort(scores))
          current_pop_tmp = np.array(current_pop_tmp)[idx]
          scores = np.array(scores)[idx]

          # Get top X for moving it unchanged to next generation
          direct_pass         =         np.linspace(0,direct_pass_size-1,di-
rect_pass_size,dtype="int")
          top_results = []
          for i in direct_pass:
              top_results.append(current_pop_tmp[i])

          # Get top X to get crossover and mutation for nex generation.
          top_x            =            np.linspace(0,mating_pop_size-1,mat-
ing_pop_size,dtype="int")
          repeat_matings = 2
          for j in range(repeat_matings):
              pool = top_x.copy()
              for i in range(0, top_x.shape[0],2):

                  # Choose from mating pool
                  pool, choices = self.mating_pool(pool)

                  ind_1 = current_pop_tmp[choices[0]].copy()
                  ind_2 = current_pop_tmp[choices[1]].copy()

                  if k_fold > 0:
                      # Use k-point mask
                      co_mask    =    self.get_k_point_crossover_mask(k_fold,
ind_2.shape[0])
                  else:
                      # Use unifrom mask
                      co_mask = self.get_uniform_mask(0.5, ind_2.shape[0])

                  # Do crossover
                  co_mask      =      np.repeat(co_mask[:,      np.newaxis],
ind_2.shape[1],axis=1)
                  offspring_1 = np.select([co_mask == True, co_mask == False],
[ind_1, ind_2])
                  offspring_2 = np.select([co_mask == False, co_mask == True],
[ind_1, ind_2])
                  #print(co_mask)
                  #time.sleep(10)

                  # Calculate mutation parameters
                  total_elements     =     offspring_1.shape[0]     *     off-
spring_1.shape[1]
                  mutation_count  =  int(total_elements  *  self.mutation_per-
cent)

                  # Create mutation bit mask
                  mask_1 = np.invert(np.random.randint(0,total_elements/muta-
tion_count,size=ind_1.shape).astype(bool))
                  mask_2 = np.invert(np.random.randint(0,total_elements/muta-
tion_count,size=ind_2.shape).astype(bool))
```

```
                    r_1             =             np.random.random_sample(off-
spring_1.shape)#*np.max(ind_1)
                    r_2             =             np.random.random_sample(off-
spring_2.shape)#*np.max(ind_2)

                    offspring_1[mask_1] = r_1[mask_1]
                    offspring_2[mask_2] = r_2[mask_2]

                    #Add offsprings to new population
                    current_pop.append(offspring_1)
                    current_pop.append(offspring_2)

            for res in reversed(top_results):
                current_pop.insert(0, res)

            scores,   model_scores   =   np.array(self.runSingleTestCasePopula-
tion(current_pop.copy(), formatter, plant, disp))
            self.score_graph.append(np.max(model_scores))
            return current_pop, scores

        def runSingleTestCasePopulation(self, pop, formatter, plant, disp):

            """
            Does all the neccssary stuff for getting the fitness score
            """
            pop_c = []
            pop_c = pop.copy()
            fitness_scores = [0] * len(pop_c)
            model_scores = [0] * len(pop_c)

            for index, individual in enumerate(pop_c):
                # Scale signals to a right output scale
                ind = individual.copy()
                ind = formatter.quantize_data(ind.copy())


                lst = []
                lst.append(ind)
                plant.populate_bus(lst[0], self.sample_time, self.time_step)
                model_score = self.runFitnessScore(plant)
                penalty = self.get_existing_case_penalty(ind)
                combined_score = model_score - penalty
                fitness_scores[index] = combined_score
                model_scores[index] = model_score

            return fitness_scores, model_scores

        def get_existing_case_penalty(self, arr):
            """
            Returns the maximum penalty when compared to all already generated
cases
            """
            smoothed_new_case = self.compress_logic(self.fast_smooth(arr))
            penalty_max = 0
            for case in self.existing_cases:
                compressed_case = self.compress_logic(case)
                pen   =   self.get_similarity_penalty(smoothed_new_case,   com-
pressed_case)
                if penalty_max < pen:
```

```
                penalty_max = pen
        return penalty_max

    def get_similarity_penalty(self, new, old):
        """
        Expand input of smaller array to match larger one and calculate
eucaledian distance
        between them.
        """
        if old.shape[0] == new.shape[0]:
            pass
        elif old.shape[0] > new.shape[0]:
            rep_new_fill = np.repeat(new[-1:,:], repeats=(old.shape[0] -
new.shape[0]), axis=0)
            new = np.concatenate([new, rep_new_fill],axis=0)
        else:
            rep_old_fill = np.repeat(old[-1:,:], repeats=(new.shape[0] -
old.shape[0]), axis=0)
            old = np.concatenate([old, rep_old_fill],axis=0)

        # Eucaledian dist
        dist = np.linalg.norm(old - new)
        return 1/max((dist)/old.shape[0]*old.shape[1],0.001)


    def compress_logic(self, input):
        """
        Compresses input so that duplicate logical states are removed
        """
        arr = input.copy()
        compressable = True
        while compressable:
            length = arr.shape[0]
            compressable = False
            del_indcies = []
            for i in range(length-1):
                if np.array_equal(arr[i,:], arr[i+1,:]):
                    del_indcies.append(i)
                    compressable = True
            arr = np.delete(arr, del_indcies, axis=0)
        return arr

    def fast_smooth(self, arr):
        arr_to_smooth = arr.copy()
        signal_most_frequent = []
        for i in range(arr_to_smooth.shape[1]):
            signal_most_frequent.ap-
pend(np.bincount(arr_to_smooth[:,i].astype("int")).argmax())
        for i in range(arr_to_smooth.shape[0]):
            for j in range(arr_to_smooth.shape[1]):
                arr_to_smooth_copy = arr_to_smooth.copy()
                if arr_to_smooth[i][j] != signal_most_frequent[j]:
                    arr_to_smooth_copy[i][j] = signal_most_frequent[j]
        return arr_to_smooth

    def runFitnessScore(self, plant):
        """
        Runs the fitness score
        """
```

```
            cov_function = self.get_coverage_function()

            cov = cov_function() #plant.get_MCDC()
            return cov

        def reduce_noise_all(self, plant, final_cases, disp):
            """
            Reduces noise from all solutions. Uses the most common value of a
signal to try to minimize state change
            to that, without losing coverage.
            """
            noisy_case = final_cases.copy()
            smoothed_cases = []
            for case in noisy_case:
                case = self.reduce_noise(plant, case)
                #disp(np.clip(case, 0, 4))
                smoothed_cases.append(case)
            return smoothed_cases

        def reduce_noise(self, plant, case):
            """
            Reduces noise from one solution. Uses the most common value of a
signal to try to minimize state change
            to that, without losing coverage.
            """
            cov_function = self.get_coverage_function()

            plant.populate_bus(case, self.sample_time, self.time_step)
            score = cov_function() #plant.get_MCDC()
            signal_most_frequent = []
            for i in range(case.shape[1]):
                signal_most_frequent.ap-
pend(np.bincount(case[:,i].astype("int")).argmax())

            for i in range(case.shape[0]):
                for j in range(case.shape[1]):
                    case_copy = case.copy()
                    if case[i][j] != signal_most_frequent[j]:
                        case_copy[i][j] = signal_most_frequent[j]
                        plant.populate_bus(case_copy,        self.sample_time,
self.time_step)
                        mod_score = cov_function() #plant.get_MCDC()
                        if mod_score >= score:
                            case = case_copy
            return case
```

APPENDIX B: UTILITY FUNCTIONS FOR GENETIC ALGORITHM

```
import numpy as np

class Penalty:
    def __init__(self):
        self.final_cases = []

    def detect_ups_downs(self, y):
        """
        Returns all the state changes of one 1d array. Output size is input
- 1
        """

        s0 = np.flatnonzero(y[1:] > y[:-1])+1
        s1 = np.flatnonzero(y[1:] < y[:-1])+1

        idx0 = np.searchsorted(s1,s0,'right')
        if len(idx0) == 0:
            return [0 for x in range(len(y))]
        s0c = s0[np.r_[True,idx0[1:] > idx0[:-1]]]

        idx1 = np.searchsorted(s0c,s1,'right')
        if len(idx1) == 0:
            return [1 for x in range(len(y))]
        s1c = s1[np.r_[True,idx1[1:] > idx1[:-1]]]

        out = np.zeros(len(y),dtype=int)
        out[s0c] = 1
        out[s1c] = -1
        return out

    def get_state_change_array(self, input):
        """
        Returns array with state changes -1, 0 or 1. One less rows than
input
        """
        change_array = np.empty((0, input.shape[0]), int)
        for i in range(arr.shape[1]):
            # Do derivative penalty
            col = input[:,i].astype(int)
            der_col = np.asarray(self.detect_ups_downs(col.copy()))
            der_col = np.expand_dims(der_col, axis=0)
            change_array = np.append(change_array, der_col, axis=0)
        return change_array

    def get_penalty(self, case):
        """
        Calculates the penalties for state changes and similarities to pre-
viously generated cases
        """
        case_q = quantize_data(case)

        signal_wise_penalty = 0
        for i in range(case_q.shape[1]):
            # Do derivative penalty
            col = case_q[:,i].astype(int)
            der_col = self.detect_ups_downs(col.copy())
            der_changes = der_col[:-1] != der_col[1:]
            x = np.sum(der_changes)

            signal_wise_penalty += x
        return signal_wise_penalty/(case.shape[1]*case.shape[0])

class Formatter:
    """ Formats 0 to 1 inputs to match defined axis limits """
    def __init__(self, axis_limits):
        self.axis_limits = axis_limits

    def quantize_data(self, individual):
        ind = individual.copy()
        for i in range(0, ind.shape[1]):
            # Boolean signal
            if max(self.axis_limits[i]) == 1:
                # Threshold to 0.5, maybe change this to a parameter later
                ind[:,i] = np.where(ind[:,i] > 0.5, 1, 0)

            elif max(self.axis_limits[i]) == min(self.axis_limits[i]):
                ind[:,i] = max(self.axis_limits[i])

            # Signal is not booelan, so scale based on axis limits
            else:
                ax_max = max(self.axis_limits[i])
                ind[:,i] = np.rint(ind[:,i]*ax_max)
        return ind
```

APPENDIX C: INTERFACE CLASS BETWEEN MATLAB AND GENETIC ALGORITHM

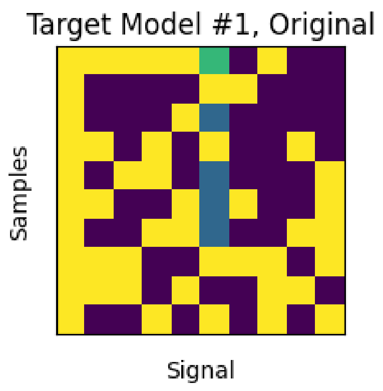
```
import matlab.engine
import matplotlib.pyplot as plt
import numpy as np
import time
#matlab.engine.shareEngine

class SimulinkPlant:
    def __init__(self,signals, modelName, busName):

        self.modelName = modelName
        self.signal_dict = signals
        self.signal_bus_name = busName
        self.ref_models = []

    def connectToMatlab(self):
        """ Connects to a running matlab instance """
        print("Starting matlab")
        self.eng = matlab.engine.connect_matlab()

        print("Connected to Matlab")

        #Load the model
        self.eng.eval("model = '{}'".format(self.modelName),nargout=0)
        self.eng.eval("load_system(model)",nargout=0)

        print("Initialized Model")

    def createBus(self):
        """ Create a bus where to input GAs results for simulation"""
        bus_str = ""

        for k,v in self.signal_dict.items():
            temp_str = "elems({0}) = Simulink.BusElement;\nelems({0}).Name
= '{1}';\nelems({0}).DataType = '{2}';\n".format(k,v[0],v[1])
            bus_str = bus_str + temp_str

        bus_str  =  bus_str  +  "{0}  =  Simulink.Bus;\n{0}.Elements  =  el-
ems;\n".format(self.signal_bus_name)
        self.eng.eval(bus_str,nargout=0)

    def populate_bus_random(self):
        """ Populate bus with random data for testing """
        time_str =   """sampleTime = 0.01;
                     numSteps = 1001;
                     time = sampleTime*(0:numSteps-1);
                     time = time';
                  """
        self.eng.eval(time_str,nargout=0)
        self.eng.eval("clear busin;",nargout=0)

        populate_str = ""
        for k,v in self.signal_dict.items():
            r = (np.random.rand(1001)*10).astype("int").tolist()
            md = matlab.uint8(r)
```

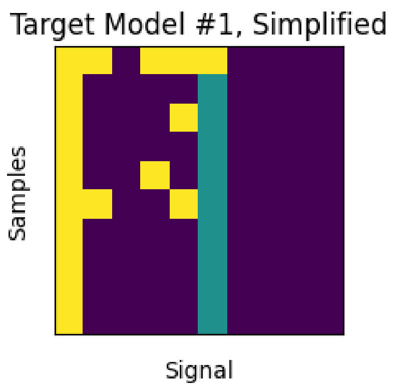
```
                self.eng.workspace['md'] = md
                populate_str = "busin.{0} = timeseries(md,time);".format(v[0])
                self.eng.eval(populate_str,nargout=0)

        def resample_input(self, input, resample_factor):
            """ Resample input so that GAs output matches expected sample
count"""
            return input.repeat(resample_factor,axis=0)

        def populate_bus(self, data, sample_time, step):
            """ Populate bus with GAs output """
            time_str  =     "sampleTime  =  {};  numSteps  =  {};time  =  sam-
pleTime*(0:numSteps*100-1); time = time';".format(step, sample_time)
            data = self.resample_input(data, 100)
            self.eng.eval(time_str,nargout=0)
            self.eng.eval("clear busin;",nargout=0)

            populate_str = ""
            for k,v in self.signal_dict.items():
                r = data[:,k-1].astype("int").tolist()
                if v[1] == 'uint16':
                    md = matlab.uint16(r)
                elif v[1] == "int16":
                    md = matlab.int16(r)
                else:
                    md = matlab.uint8(r)

                self.eng.workspace['md'] = md
                populate_str = "busin.{0} = timeseries(md,time);".format(v[0])
                self.eng.eval(populate_str,nargout=0)

        def get_MCDC(self):
            """ Retrieve MCDC coverage from the simulated model"""
            input_str = "mdl = '{}';\n".format(self.modelName)
            self.eng.evalc(input_str)
            input_str =  "testObj = cvtest(mdl);\n testObj.settings.mcdc = 1;\n
blk_handle = get_param(mdl, 'Handle');\n"
            self.eng.evalc(input_str)
            input_str = "data = cvsim(testObj);\n cov = mcdcinfo(data, blk_han-
dle);\n"
            self.eng.evalc(input_str)
            input_str_top = "cov(1) / cov(2)\n"
            try:
                top_score  = self.eng.eval(input_str_top)
            except:
                top_score = 0

            return top_score

        def get_decision(self):
            """ Retrieve decision coverage from the simulated model"""
            input_str = "mdl = '{}';\n".format(self.modelName)
            self.eng.evalc(input_str)
            input_str =  "testObj = cvtest(mdl);\n testObj.settings.mcdc = 1;\n
blk_handle = get_param(mdl, 'Handle');\n"
            self.eng.evalc(input_str)
            input_str  =  "data  =  cvsim(testObj);\n  dec  =  decisioninfo(data,
blk_handle);\n"
            self.eng.evalc(input_str)
            input_str = "dec(1) / dec(2)\n"
            return self.eng.eval(input_str)

        def get_condition(self):
            """ Retrieve condition coverage from simulated model"""
            input_str = "mdl = '{}';\n".format(self.modelName)
            self.eng.evalc(input_str)
            input_str =  "testObj = cvtest(mdl);\n testObj.settings.mcdc = 1;\n
blk_handle = get_param(mdl, 'Handle');\n"
            self.eng.evalc(input_str)
            input_str  =  "data  =  cvsim(testObj);\n  dec  =  conditioninfo(data,
blk_handle);\n"
            self.eng.evalc(input_str)
            input_str = "dec(1) / dec(2)\n"
            return self.eng.eval(input_str)

        def simulate(self):
            """ Call simulate on model """
            self.eng.set_param(self.modelName,'SimulationCommand','start',nar-
gout=0)

        def disconnect(self):
            """ Dissconect from matlab instance to remove reconnection errors"""
            self.eng.set_param(self.modelName,'SimulationCommand','stop',nar-
gout=0)
            self.eng.quit()
```

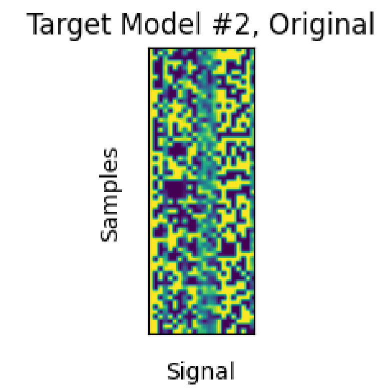
APPENDIX D: EXAMPLE CODE FOR CREATING A TEST CASE

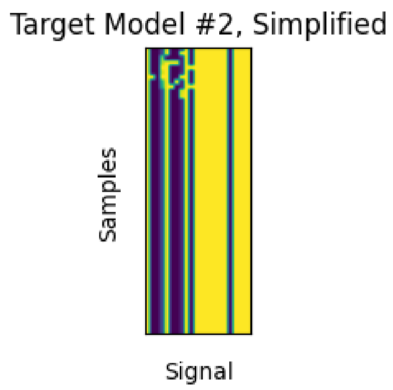
```
import numpy as np
import matplotlib.pyplot as plt
from utility_functions import Penalty, Formatter
from genetic_algorithm import GA
from SimulinkPlant import SimulinkPlant
from scipy import signal, spatial

test_suite_size = 5

# ======= Init live visualizer ======
def display_live_img(arr):
    %matplotlib inline
    ax.clear()
    ax.imshow(arr)
    display(fig)
    clear_output(wait=True)
    plt.pause(0.1)

#====== Define signal datatypes ==========
sig_dict =  {1:["Signal1", 'uint8'],
            2:["Signal2", 'uint8'],
            3:["Signal3",'uint8'],
            4:["Signal4", 'uint8'],
            5:["Signal5", 'uint8'],
            6:["Signal6", 'uint8'],
            7:["Signal7",'uint8'],
            8:["Signal8",'uint8'],
            9:["Signal9",'uint8'],
            10:["Signal10",'uint8']}

#====== Connect to MATLAB instance ==========
plant = SimulinkPlant(sig_dict, "ExampleModel", "ExampleBus")
plant.connectToMatlab()
plant.createBus()

# ======= Define signal axis limits =======--
axis_limits = {}
for k,v in plant.signal_dict.items():
    if k == 1:
        axis_limits[k-1] = [80,80]
    elif v[0] == "Signal1":
            axis_limits[k-1] = [0,4]
    else:
        axis_limits[k-1] = [0,1]

# ==== Initialize penalty, formatter and genetic algorithm ========
penalty = Penalty()
formatter = Formatter(axis_limits)
save_name = "Example_1"
ga_size = (10,10)
ga_init_type = np.zeros
ga = GA(plant,
        coverage_type="MCDC",
        time_step=0.02,
            sample_time=ga_size[0],
            mutation_percent = 0.04,
            inclusive_mating = True,
            crossover_type = 2,
            mating_population_size = 15)

    # ===== Train one test case to completion
    def train_one_case():
        pop_size = 15
        initial_pop = []
        for i in range(pop_size):
             scores, _ = ga.get_init_scores(initial_pop, formatter, plant, dis-
play_live_img)


        current_pop = []

        max_plateau = 50
        plateau_count = 0
        prev_best_score = 0

        generations = 500

        # Do crossover
        for gen in range(generations):
            current_pop, scores = ga.train_generation(current_pop, scores, for-
matter, plant, display_live_img)

            if gen % 1 == 0:

                if np.isclose(max(scores), prev_best_score):
                    plateau_count = plateau_count + 1
                else:
                    plateau_count = 0
                if plateau_count >= max_plateau or max(scores) == 1:
                    break

                prev_best_score = max(scores)
                disp_arr = formatter.quantize_data(current_pop[0])
                disp_arr  =  disp_arr  /  np.clip(disp_arr.max(axis=0),  0.0001,
None)
                display_live_img(disp_arr)
        return current_pop[0]


    # ===== Use the GA multiple times to get the test suite =======
    final_cases = []
    score_graphs = []
    for x in range(test_suite_size):
            ret = formatter.quantize_data(train_one_case())
            ret = ga.reduce_noise(plant, ret)
            final_cases.append(ret)
            ga.existing_cases.append(ret)
            score_graphs.append(ga.score_graph)
            ga.score_graph = []
```

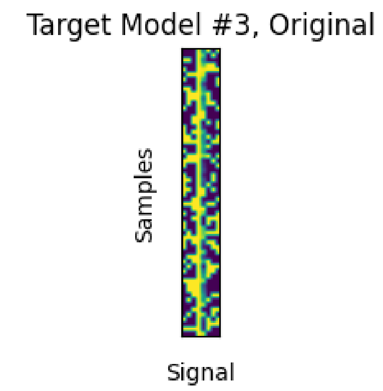
APPENDIX E: SIMPLIFIED AND UNSIMPLIFIED TEST CASE COMPARISON

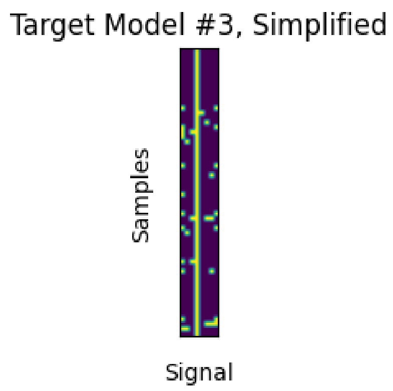

Target Model #1, Original
Samples
Signal

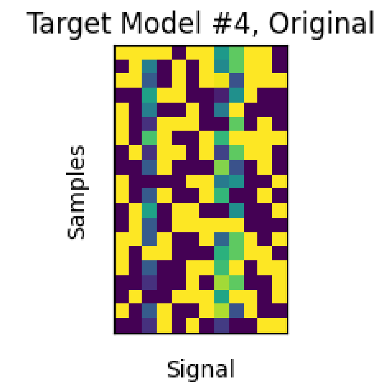

Target Model #1, Simplified
Samples
Signal

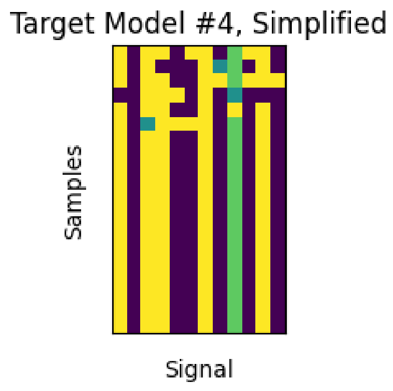

Target Model #2, Original
Samples
Signal


Target Model #2, Simplified
Samples
Signal


Target Model #3, Original
Samples
Signal


Target Model #3, Simplified
Samples
Signal


Target Model #4, Original
Samples
Signal


Target Model #4, Simplified
Samples
Signal