Technical University of Eindhoven

DC 2017.040
Internship report

Coverage Verification Framework for ADAS Models

Author:
Samir Achrifi (0908391)

Supervisors:
Prof.dr. H. Nijmeijer (TU/e)
Ir. PDEng F.P.A. Benders (TNO)

March 3, 2017

Contents

Chapter 1

Introduction

TNO Automotive in Helmond is a research center with a strong track record in applied automotive research. Together with their clients, TNO creates innovative, integrated and tailor-made solutions for safety and powertrain related technologies. One of their iconic projects is the 'EcoTwin' project, in which a strong basis is formed for cooperative truck platooning. The platooning trucks rely heavenly on Advanced Driving Assistance Systems (ADAS) for automated driving. These systems are subjected to strict safety protocols, like ISO 26262 to secure and maintain safety in public traffic.
The Integrated Vehicle Safety (IVS) department of TNO is responsible for the development of ADAS functions. Before the ADAS functions are implemented in real-life vehicles, they are extensively tested in simulated environments like MATLAB Simulink. However, due to the lack of verification framework for MATLAB Simulink, the question arises whether models receive complete test coverage. And even with complete coverage, there is no method to assess and compare quality among different generations of ADAS models.
This project is a continuation of a preceding project. The IVS department has researched the feasibility of creating a code coverage verification framework for arbitrary ADAS models using the MATLAB Verification and Validation (VnV) toolbox. The preceding project's conclusion confirmed the verification framework for ADAS models to be plausible.

The main goal for this project is to improve the overall quality of the ADAS Simulink models produced by the IVS department. In order to achieve this, a code coverage verification framework will need to be created. Code coverage is a measure that indicates how much model elements have been executed during a simulation. In verification processes, a high code coverage means the model has been thoroughly tested. A complete code coverage means every single model element has been executed and tested.
The framework should accept arbitrary ADAS models and verify their code coverage. In addition to the verification framework, a method needs to be developed that assesses and compares quality between different generations of ADAS models. Having achieved this goal, the IVS department receives two tools which improves quality of the ADAS model by allowing verification in the early part of the development.
The project will have the following approach. The ADAS models are created in MATLAB Simulink. A verification framework will be created using the MATLAB VnV toolbox. The verification framework will assess the models' code coverage. The quality comparison between different ADAS models will be done through a MATLAB code algorithm, which will highlight signals that significantly differ between models.
The code coverage tool will eventually be tested through monkey testing, black-box testing and white-box testing. Based upon the conclusion of the testings, recommendations will be given on efficient code coverage testing strategies.

This report contains six chapters. Its structure is as follows: The report starts with an introduction. Chapter two provides background information on the ISO 26262 standard. The third chapter will review VnV tool platforms which allow code coverage in coherence with ISO 26262. Chapter four describes the code coverage in MATLAB using the VnV toolbox. The fifth chapter recommends testing strategies to increase coverage. The report will be finalized with conclusions and recommendations. The appendix features in-detail code descriptions of the created MATLAB functions.

Chapter 2

Safety relevance of ISO 26262

The automotive branch is experiencing exponential growth in the complexity of electrical and electronic (E/E) systems. Nowadays, the development of automotive software isn't restricted to only car manufacturers in general. Additionally, third-party developers are producing more and more software, which find their way into vehicles meant for the general consumers.
To maintain the functional safety of the systems, and with that, the safety in public traffic, safety requirements are becoming more regulated. World-wide standards exist that apply on the design and testing of products. The International Organization for Standardization (ISO) is an independent, non-governmental organization that provides specifications on products, services and systems for every industry. One of the international standards is ISO 26262 [9].
ISO 26262 provides specifications and protocols for safety critical components in automotive-specific products. This chapter will provide information on the background, key aspects like Automotive Safety Integrity Levels (ASIL), Tool Confidence Levels (TCL) and the relevance of ISO 26262 to this project.

2.1 Background on ISO 26262

On the 15th of November 2011 ISO introduced the protocol ISO 26262, titled "Road Vehicles - Functional Safety". ISO 26262 is an international standard regarding functional safety of E/E systems for automotive vehicles that fall into the category "series production passenger cars" with a maximum gross vehicle weight of 3500 kilograms.
ISO 26262 is based on the IEC 61508 standard, titled "Functional Safety of Electrical/Electronic/programmable Electronic Safety-related Systems (E/E/PE or E/E/PES)" [3]. As ISO describes it, "Like its parent standard, ISO 26262 is a risk-based safety standard, in which the risk of dangerous operational situations are assessed and safety measures are defined to avoid or control systematic failures and to detect or control random hardware failures, or mitigate their effects."[9].
Take drive-by-wire systems as an example. Electrical or electro-mechanical by-wire systems, like steer-by-wire and throttle/brake-by-wire are set to replace their mechanical counterpart systems in modern automotive vehicles. This has the advantage that it saves weight and space by eliminating massive mechanical parts. However, steering, throttling and braking are deemed safety critical systems. Failures in these systems can significantly compromise driver and passenger safety. For drive-by-wire systems, a control unit is placed as the governing body between the user's input and the actuator. The control unit governs the safety critical systems based on many inputs. It is therefor a challenge to test and validate systems like these. ISO 26262 provides safety standards for systems like these and other E/E system categories as shown in figure 2.1.

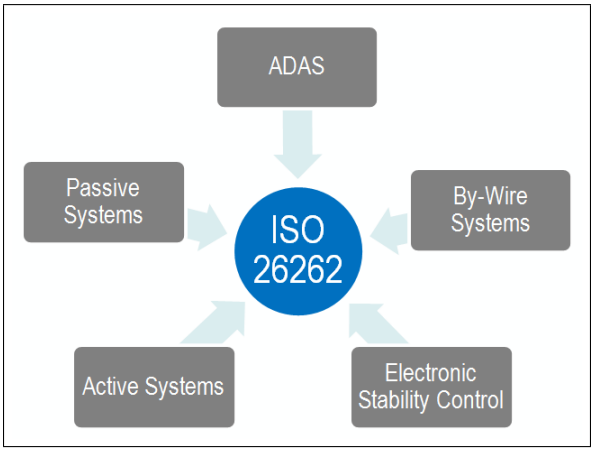


Figure 2.1: This figure [8] shows E/E system examples which are recommended to be designed under the principles of ISO 26262.

2.2 Key aspects

This section will describe the key aspects of the ISO 26262 standard. The standard provides regulations and recommendations for the whole product, service or system development process, from the concept design to decommissioning. The main goals of the standard are the following:

- Provides an automotive safety life cycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these life cycle phases.
- Covers functional safety aspects of the entire development process (including such activities as requirements specifications, design, implementation, integration, verification, validation and configuration.
- Provides an automotive-specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs).
- Uses ASILs for specifying the system's necessary safety requirements for achieving an acceptable residual risk.
- Provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety is being achieved [17].

In short, the standard's process is to define, assign and manage safety risks. Vehicle-level Hazardous events are defined that relate to the operational situations of the product, service or system. The Automotive Safety Life cycle (ASLC) process provide tools to judge the hazardous events and assign an Automotive Safety Integrity Level (ASIL) to the events. Safety goals are defined based on the events with ASILs, which in turn leads to safety requirements. The ASLC and ASIL are further described in the following subsections.

2.2.1 Automotive Safety Life Cycle (ASLC)

The Automotive Safety Life Cycle (ASLC) is a process in ISO 26262. It provides tools to recognise hazardous events for the eventual end product. ASILs are used to establish safety requirements for the end product that decrease the safety risks to an acceptable level. The following is a summary of the ASLC procedure.

- An item (a particular automotive system product) is identified and its top level system functional requirements are defined.
- A comprehensive set of hazardous events are identified for the item.

- An ASIL is assigned to each hazardous event.
- A safety goal is determined for each hazardous event inheriting the ASIL of the hazard.
- A vehicle level functional safety concept defines a system architecture to ensure the safety goals.
- Safety goals are refined into lower-level safety requirements. In general, each safety requirement inherits the ASIL of its parent safety requirement/goal. However, subject to constraints, the inherited ASIL may be lowered by decomposition of a requirement into redundant requirements implemented by sufficiently independent (redundant components).
- Safety requirements are allocated to architectural components (subsystems, hardware components, software components). In general, each component should be developed in compliance with standards and processes suggested/required for the highest ASIL of the safety requirements allocated to it.
- The architectural components are then developed and validated in accord with the allocated safety (and functional) requirements. [1][12]

2.2.2 Automotive Safety Integrity Level (ASIL)

As mentioned before, the Automotive Safety Integrity Level (ASIL) is a grade that is assigned to previously determined possible hazardous events that are related to the functionality of the end product. Each hazard is analysed with ASIL through the question: "If a failure occurs in this situation, what will be the consequences to the driver, passengers and other road users involved?". [10]

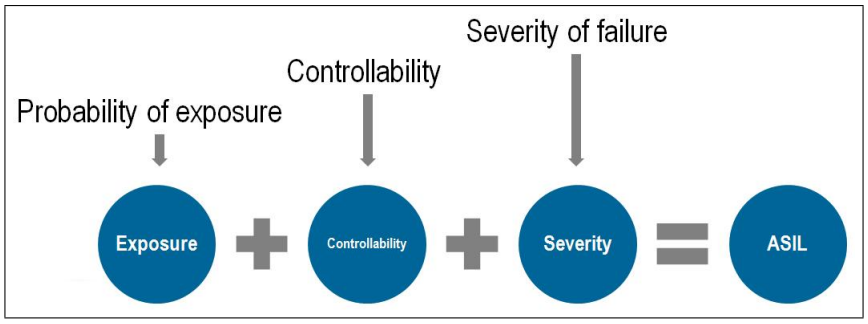


Figure 2.2: This figure [8] displays the combination of probability of exposure, controllability and the severity of a failure that determine the ASIL.

The level determination of the potential risk is, as figure 2.2 displays, based on the several aspects. The probability of the exposure asks how likely it is for the hazard to occur. The controllability indicates how likely it is for the user to control the consequences of the hazard. The severity of the failure estimates how many people will become involved in the hazard and how dire the consequences are for those people. These factors combined determine the ASIL.
For an example, compare a headlight system with a steer-by-wire system. The probability of a headlight system failing is relatively common, though, assuming the driver is aware of the situation, the user still maintains control over the vehicle so the severity of the failure is generally low. A failure in a steer-by-wire system is uncommon, though a loss of the driver's steering control increases the severity of the failure enormously.

In the process to an ASIL, the system is rated independently for exposure, controllability and severity. These ratings are taken into account in the estimation of the ASIL. The ratings for the classifications contain the first letter of its title, with a number grading the proportion.

Severity Classifications

- **S0** No injuries
- **S1** Light to moderate injuries
- **S2** Severe to life-threatening (survival probable) injuries
- **S3** Life-threatening (survival uncertain) to fatal injuries [9]

Exposure Classifications

- **E0** Incredibly unlikely
- **E1** Very low probability (injury could happen only in rare operating conditions)
- **E2** Low probability
- **E3** Medium probability
- **E4** High probability (injury could happen under most operating conditions) [9]

Controllability Classification

- **C0** Controllable in general
- **C1** Simply controllable
- **C2** Normally controllable (most drivers could act to prevent injury)
- **C3** Difficult to control or uncontrollable [9]

The general ASIL rating is based on the classification ratings and can be either QM, A, B, C or D, with Automotive Safety Integrity Level D being the case of a life-threatening to fatal injuries(S3), with a high probability on injury (E4) and with little to no controllability for the driver (C3). ASIL A is the lowest safety risk level. ASIL QM rates lower than ASIL A and applies when there is no safety relevance, only Quality Management.
A demotion of one level in each classification, demotes the ASIL with one level (with an exception for a demotion from E1 to E0). A rating combination of S3, E4, C2 results in ASIL C. A situation which is life-threatening (S3), uncontrollable (C3), but has a low probability (E1), has an ASIL A rating.
The ISO 26262 specifies the minimum testing requirements needed to ensure an acceptably safe sytem based on the ASIL. The more critical the ASIL, the more effort is required to test the system. This is key in determining the testing method which is required to ensure a safe system.

2.3 ISO 26262 on Verification and Validation

As ASILs become more critical, the regulations become more stricter for testings in order to ensure the system will operate within safety requirements. This section will discuss the regulations of ISO 26262 on Verification and Validation principles for model-based software.

The increasing complexity of automotive E/E systems also raises the motive for testing early in the development process of the safety-related system. It is estimated that an system failure would cost a ten fold less if it is discovered during the production phase, instead of when released. The costs are reduced another ten times if the failure is discovered during the design phase instead of during the production phase [8]. Because of the increasing complexity and cost consequences, model-based design is extensively used in the automotive branch for the development of E/E systems that are required to meet the requirements of the ISO 26262 standard. Introducing verification and validation early in the system development process not only leads to increased safety for the eventual user, but it also increases efficiency and cost reduction [4].

ISO 26262 places demands on application-specific verification and validation (VnV), regardless of used development tools and processes. Even though ISO 26262 is derived from its parent standard IEC 61508, they handle software tool qualifications differently. When using model-based design with the intention of code generation for the production system, the ISO 26262 standard asks the application-specific verification and validation the following questions:

- Does the model implement its (textual) requirements?
- Does the object code to be deployed in the ECU implement the models' full design? [4]

Within these questions ISO 26262 provides guidance on software tool qualifications [11]. The guidance ensures the VnV software tool is acceptable for use in the development of ASIL-assigned safety systems.

2.3.1 Tool Confidence Level (TCL)

The ISO 26262 standard demands the VnV tool to be analysed using documented use cases. The analysis shows the effect of the VnV tool on the safety requirements by asking: is it possible that an erroneous output or a malfunction in the VnV tool can breach the safety requirements? Additionally, the VnV tool needs to analyse the exposure of the malfunction or error: what is the probability the VnV tool prevents or detects errors in the software of the system? The ISO 26262 standard summarizes the analysis in two points:

1. Whether a malfunctioning software tool and its erroneous output can lead to the violation of any safety requirement allocated to the safety-related software to be developed.
2. The probability of preventing or detecting such errors in the output of the tool.

These analyses combined estimate the Tool Confidence Level (TCL). The TCL is a grade assigned to the VnV tool, like an ASIL, it is assigned to the system. The TCL is a grade used in the ISO 26262 standard to provide the appropriate tool qualification methods [5].
The TCL is based on a combination of two classifications: Tool of Impact (IC) and Tool Error Detection (TD), with which the combination leads to a Tool Confidence Level (TCL). Figure 2.3 shows the procedure schematically in determine the TCL of the VnV tool.

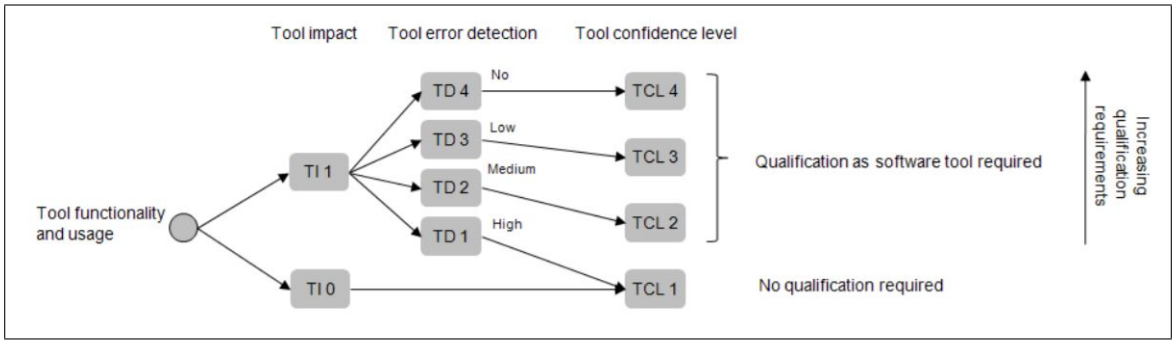


Figure 2.3: This figure [5] displays the ISO/DIS 2626-8 Tool Classification Scheme to determine the TCL

The Tool Impact (TI) is a grade that is assigned to the VnV tool in the process of determining the TCL. The TI is based on analysis on whether a malfunction or erroneous output of the VnV tool can cause a breach of the safety requirements for the system. If it is possible to breach a safety requirement, the grade TI1 is assigned. In case there is no threat of a safety requirement breach, TI0 is assigned [11].

The Tool Error Detection (TD) is a grade that rates the VnV tool in the probability of the tool to detect or prevent its own malfunctions or erroneous output. The TD grade has four classifications, TD1 to TD4. The TD1 grade indicates a high confidence level, TD3 medium, TD2 Low and TD4 no confidence level at all. A system with a TD4 grade is not able to perform systematic verification measures and can only detect malfunctions and errors randomly. [5][11].

The TCL is based on the TI and TD gradings, as shown in the schematics of figure 2.3. The TCL can either be 1, 2, 3 or 4, with TCL1 requiring no qualifications and TCL4 requiring the most strict qualification requirements. Based on the TCL and the ASIL, the ISO 26262 standard recommends methods to be used during specific phases of the system development process.

2.3.2 Tool Qualification Work Products

Additionally to the TCL grading, in order to be classified as a VnV tool, the tool requires a set of tool qualification work products. The work products need to proof the VnV tool is reliable in use and should serve as information for users. The Tool Classification Work Products should contain a Software Tool Qualification Plan, Software Tool Documentation, a Software Tool Classification Analysis and a Software Tool Qualification Report. [11]

Software Tool Qualification Plan (STQP) The Software Tool Qualification Plan is a plan, defined in the early stages of the VnV tool development. It defines the general tool features and provides information on the intended applications, configurations and its situations in which it will operate in. The STQP states the use cases which will be used for the required TCL analysis, the derived qualification methods and its means of detecting malfunctions and erroneous output of the VnV tool [5].

Software Tool Documentation (STD) The Software Tool Documentation should serve as a user's manual to users of the VnV tool. It features information the user is required to know to operate the tool, such as: features overview, operations, installation, limitations, etc...
The STD also provides references to the use cases, configurations and environments used in the STQP for estimating the TCL [5].

Software Tool Classification Analysis (STCA) The Software Tool Classification Analysis is the previously described analysis which is required to determine the TCL. The TCL is based on the TI and TD grading, which rates its performance in preventing or detecting malfunctions and errors and wether these malfunctions or errors can cause a safety requirement breach. The analysis uses the use cases defined in the STQP. Based on the TCL and the ASIL, the required qualification methods are determined for the system in development [5].

Software Tool Qualification Report (STQR) The Software Tool Qualification Report features proof the tool has achieved the qualifications of the ISO 26262 standard as planned in the STQP. Tool limitations, constraints and malfunctions identified during the qualification are documented in this report as well [5].

2.4 ISO 26262 Summary

This section summarizes the most relevant items that have been mentioned in this chapter. This chapter has described the aspects of the ISO 26262 protocol that relate to this project. ISO 26262 is a protocol that provides methods and processes which guard the functional safety of automotive electronic and electric systems. The protocol focusses on systems implemented in series production passenger cars with a gross weight of less than 3500 kilograms.
The Automotive Safety Life Cycle (ASLC) procedure is used to establish scenarios for the to-be-designed system. The system is placed in a theoretical hazardous event. The behaviour of the system during the hazard is estimated in terms of exposure, controllability and severity. Based on its behaviour, an Automotive Safety Integrity Level (ASIL) is assigned. The ASIL is a grade which determines the recommended verification procedure for the system.
The protocol also sets requirements for verification and validation tools. An analysis is described that grades the tool in its ability in handling malfunctions and errors. A Tool Confidence Level (TCL) is provided to the tool and functions as a tool qualification. Furthermore, a tool should contain the following Tool Classification Work Products: a Software Tool Qualification Plan, Software Tool Documentation, a Software Tool Classification Analysis and a Software Tool Qualification Report.

Chapter 3

Verification and Validation tool platforms

The previous chapter described the background and key aspects of ISO 26262 and its relevance to verification and validation. This chapter will focus on finding software platforms with tools for code coverage verification and validation that comply to the requirements of the ISO 26262 standard, described in the previous chapter.
A commercial study regarding software companies has concluded five industry-accessible software platforms with tools that allow for code coverage verification and validation. The software programs are listed below.

- MATLAB/Simulink
- Vector Software
- Parasoft
- CertTech
- LDRA

3.1 ISO 26262 compliant software platforms

The five listed software programs are further described in this chapter. Their relation to the ISO 26262 standard is described as well as their features in the area of code coverage verification and validation. Another important aspect is the platform's potential to further extend VnV activities in the future. These following sections inform the reader in which ways the software platform are compatible with this project.

3.1.1 MATLAB Simulink

The key products MathWorks provides for use in model-based design and verification certified with the ISO 26262 and IEC 61508 standards are listed below.

- IEC Certification Kit (for ISO 26262 and IEC 61508)
- Simulink
- Stateflow
- Embedded Coder
- Simulink Verification and Validation
- Simulink Design Verifier
- Simulink Polyspace code verification products [16]

On the twelfth of April 2011, MathWorks has announced that their MATLAB verification toolboxes, Simulink Design Verifier and Simulink Verification and Validation, have been TÜV SÜD certified for use in ISO 26262 regulated development processes. This allows automotive embedded systems to be developed and verified within the model-based environment of Simulink, whilst maintaining the quality and confidence the ISO 26262 standard provides [16].

TÜV SÜD is an international technical service organisation that provides technical guidance, consultation, testing and certification to developers in finding balance between quality, profitability and sustainability [13]. The certification regarding MathWorks' Simulink toolboxes indicates the tools agrees with the ISO 26262 qualification requirements for VnV tools, described in the previous chapter.

MathWorks' Simulink Design Verifier toolbox provides automated test case generation and the Verification and Validation toolbox focusses on model coverage measurements. TÜV SÜD assessed the classifications and qualification results, whether it complies with the ISO 26262-8 standard.
The ISO 26262 and IEC 61508 certifications of the MathWorks MATLAB verification toolboxes makes them an option to consider for development and verification of automotive E/E systems using the Simulink model-based design and verification tool products.

The strong advantage of VnV on the MATLAB platform is that its code coverage verification can be used on Simulink models. It is unique to its competitors in the sense that it can provide model coverage, opposed to the strict code coverage the majority of the other platforms use.

3.1.2 Vector Software

The VectorCast VnV tools that comply with the ISO 26262 standard are listed below.

- VectorCast/C++
- VectorCast/Cover
- VectorCast/RSP (Runtime Support Package)
- VectorCast/Ada
- VectorCast/QA
- VectorCast/Analytics [19]

Vector Software has introduced the tool VectorCast, which has a products series that satisfy the VnV requirements specified in the ISO 26262 and MISRA C/MISRA C++ standards [19].
MISRA stands for Motor Industry Software Reliability Association. It provides guidelines for the use of C and C++ language for vehicle based software and critical systems [14] [15]. Both standard serve to maintain code safety, security, probability and reliability for embedded systems.

The VectorCast product series provides tools for automating testing and code coverage verification. The platform's features include: Methods for Software Unit Testing, Methods for Deriving Test Cases for Software Unit Testing and Structural Coverage Metrics at the Software Unit Level. For each type of testing, ISO 26262 specifies a recommended action based on the system's ASIL. Tests can either be recommended or highly recommended. VectorCast provides traceability between tests and ISO recommendations.

The Methods for Software Unit Testing feature is a utility to test software in certain aspects. Requirement-based testing is a method which uses test cases to determine whether software requirements are met with the current code.
Interface testing tests the robustness of a system's interface. Various methods exist to achieve this: boundary, illegal or median test values can be opted for. Both requirements-based testing as interface testing are recommended for all ASILs.

Fault injection tests assess the system's behaviour when introducing voluntarily faults. The system should not breach its safety requirements during these tests. Fault injection tests are highly Recommended for ASIL-D systems.
Resource tests are used to determine the software's required allocations and back-to-back tests between model and code for testing auto-generated code more seamless.

The Method for Deriving Test Cases for Software Unit Testing focusses generally on providing guidance in the set-up of test cases. Test cases are generated based on analysis of requirements, equivalence classes, boundary values or estimated error values. Each classification provides a different strategy approach to test the system.

As ISO 26262 strongly recommends the usage of code coverage as a metric to measure whether a system has been sufficiently tested [11], VectorCast also includes the feature for Structural Coverage Metrics at the Software Unit Level. The code coverage exists in three aspects: Statement Coverage, Branch Coverage and Modified Condition/Decision Coverage (MC/DC). MC/DC coverage is highly recommended for ASIL-D, whilst the Brand and Statement Coverage are highly recommended for ASIL-B,-C,-D and -A,-B,-C and -D respectively. Each level in coverage is more difficult to achieve. [19]

In terms of code coverage, this tool is extensive. The tool also provides the ability to animate code coverage activities, providing a visual real-time interpretation of code performance. The tool is limited to C and C++ codebases and is limited in the amount of types of coverage. Additionally, another attractive feature is the compatibility between requirements management tools such as IBM Rational DOORS.

3.1.3 Parasoft

The tools required for VnV activities on the Parasoft platform are listen below.

- Parasoft Development Testing Platform (DTP)
- Parasoft Process Intelligence Engine (PIE) [17]

In response to the ISO 26262 standard for automotive E/E systems, Parasoft has introduced two standard-compliant analysis and verification tools: The Parasoft Development Testing Platform (DTP) and the Parasoft Process Intelligence Engine (PIE). Both are tools that provide the ability to automatically monitor and report the software verification processes [17].

The Parasoft DTP tool provides ISO-compliant software quality tasks in the form of static analysis, data flow static analysis, metrics analysis, peer code review, unit testing and runtime error detection. The tools are designed to detect, prevent and correct in MISRA C/MISRA C++ critical automotive safety system. DTP collects data generated from static tests, code metrics, coverage analysis, source control check-ins, defect tracking tests and other kind of tests, and processes the results in reports, showing correlated and prioritized data [17] [18].

The Process Intelligence Engine further extends the capabilities of the DTP tool. It performs additional position analysis on systems in order to point-out risks and provide recommendations to improve development processes. The engine also provides the ability to animate code coverage activities, providing a visual real-time interpretation of code performance.

Like with VectorCast, Parasoft allows for static, branch and MC/DC coverage. The tools can do so for industry standards rules like MISRA C/MISRA C++, JSF and HIS source code. Next to code coverage, the platform also allows for further possibilities in VnV activities.

3.1.4 CertTech

CertTech provides code coverage VnV possibilities in a variety of regulatory environments, the most noteworthy being formal qualification of software development and the use of verification tools to increase product development and certification processes efficiency. The tool CertTech proposes for VnV activities is as follows:

- BullseyeCoverage Tool Qualification Kit (TQK) [2]

CertTech provides one singular, but extensive tool for the purpose of verification activities that are in accordance with DO-178B/C, DO-254, IEC 62304, ISO 26262 and other regularity guidelines [2]. The Tool Qualification Kit is a tool that is used as a code coverage analyzer for C and C++ codes. It is CertTech's answer for the request of coverage analyzer tools for safety and mission critical industries. It evaluates the completeness and correlation of software requirements and the associated testing activities.

The TQK tool contains the following features:

- Generic Tool Qualification Plan (TQP)
- Generic Tool Operational Requirements (TOR) including:
  - Development environment and test integration requirements
  - Coverage measurements requirements
  - Code construct, expressions, operator and function requirements
  - Report file generation requirements
  - Metrics and build coverage reporting requirements
- Requirements-based Verification Procedures
- Generic Tool Qualifications Accomplishment Summary (TQAS)
- Test Trace Matrix providing correlation of requirements with specific tests
- Full documentation of how to run the qualification tests

NOTE: CertTech states that content will vary due to specific industry regulatory guidelines. [2]

The contents of the TQK tool set requirements based on the most commonly used features. It provides a full suite of tests verifying the provided requirements and it does so in a framework that allows coverage to be extended as needed. However, it limited to only code coverage and has little compatibility with further VnV extensions.

3.1.5 LDRA

LDRA Software Technology provides a platform to automate approaches to meet any MISRA, ISO 26262 and other standards. The platform is focused to introduce automated checks on a wide range of programming standards, including:

- MISRA C:1998/2004/2012
- MISRA C++:2008
- CERT C Secure Coding Standard
- HIS
- JPL safety critical C
- GJB Chinese Military standard

- the Embedded C Coding standard
- JSF++ AV standard
- High-integrity C++ Coding Standard
- LM Train Control Program (LMTCP) [20]

LDRA's platform provides tools for structural code coverage analysis and extensive features like automated unit testing, test case generation and management, qualification guidance and object code verification. LDRA provides the following tools for VnV purposes:

- TBrun
- TBobjectbox
- Tool Qualification Support Pack (TQSP)
- LDRAcover
- LDRAunit [21]

In order to meet strict coverage requirements, LDRA's tool package provides coverage analysis at both the source code as well as the object code level. The tools automatically generate test cases, execute them and generate reports that visually display the results. The types of coverage these tools can provide are statement, branch/decision, procedure/function call, MC/DC and dynamic data flow. The tools support C, C++, Java, Ada and Assemblers and is able to run on wide range of platforms like 64-, 16- and 8-bit micro-controllers [21].

3.2 Reasoning for the MATLAB Simulink platform

The main desire for a software platform is that it allows for code coverage verification within the requirements of the ISO 26262 standard. Another desire is to keep financial costs and learning/operation time to a minimum. It would also be beneficial to use a system which has other extensive VnV supporting tools, besides code coverage. This would allow for further extensions of VnV in the development of systems in the future.

From all the previously mentioned software platforms, it is undeniable that the MATLAB Simulink platform is the most compatible for use in this project. As mentioned in the introduction of this report, TNO Automotive already uses the MATLAB platform for the development of its ADAS models. ADAS functions are designed and tested in Simulink models before implementation into the target hardware. It is realistic to state that TNO is well known with how the MATLAB platform works. Additionally, TNO has active MATLAB licenses. This has the advantage that it mostly negates the *learning curve* that would cost time and money had it been a new software platform.
Aside costs and time, the feature for model coverage directly in Simulink, opposed to code coverage on other platforms, is practical in the sense that TNO's ADAS models are eligible for coverage verification, without the need to convert the model into a format that is required for code coverage. The possibility of extending MATLAB toolboxes is useful for expanding the VnV activities within the platform in the future.
It is for the afore mentioned reasons that MATLAB Simulink has been chosen as the platform to develop a code/model coverage verification framework.

Chapter 4

MATLAB's code coverage verification

This chapter describes code coverage verification and validation in MATLAB Simulink. It will highlight the reasoning behind the need of verification and its potential for practical use in the model design process. MATLAB allows coverage gathering through two methods: through in-loop verification and harness verification. Both methods' functionality will be explained in this chapter as well as which situation they are practical in.
Furthermore, this chapter will describe the different types of code coverage possible using the MATLAB VnV toolbox. The toolbox generates a report containing the code coverage details and an interactive model view which provides the coverage information in the model structure. Both examples and their potential will be discussed in the last section.

4.1 MATLAB verification in general

Verification is defined as the process of determining whether a model or simulation implementation accurately represents the designer's conceptual descriptions and specifications. A good verification is therefor critical to ensure model quality before producing the algorithm on a large scale. Unwanted system behaviour can be rooted out and/or functionalities can be added or removed in case of redundancy. The result should be a system simple enough, that still contains all desired features.

The MATLAB VnV toolbox provides the ability to gather coverage data on MATLAB Simulink models. The coverage data is displayed in a generated HTML report or directly in the model structure, depending on the user's preference. The presented data aids the user in the verification process of a model. It helps in uncovering untested elements in the model structure.

The toolbox gathers coverage data on the model based on the model's behaviour on the input. It is recommended to provide test scenarios to the model as inputs. The test scenarios should contain situations similar to the expected operating situation the system is designed for. The verification will provide coverage on the system's behaviour for the provided situations. The gathered coverage is expressed in percentages. Generally speaking, the percentile score expresses the ratio of the model that is untested during the test scenarios. A model coverage of 100% would mean all elements within the model structure have been executed at least once. A model coverage of around 10% would mean a big part of the model is unexecuted and, therefor, untested and possibly redundant.

The code coverage from the toolbox is an aid in the verification process in the sense that it uncovers elements and subsystems in the model structure that are untested. If the situation may arise where a model's coverage is not enough, the model designer still has to choose the appropriate course of action. The coverage data can aid in the designer's choice. For example, if the goal is to thoroughly test a model, more test scenarios have be created in order to execute every element in the model. On the other side,

a designer can state a set of test scenarios to be all the possible situations the system will operate in. In this case any unexecuted element in the model is redundant and can be removed. As mentioned before, the desire is for a system simple enough, but with all the desired features.

4.2 Verification methods

There are two methods possible for verifying a MATLAB Simulink reference model: in-loop verification and harness verification. The in-loop verification is able to verify multiple reference models which operate inside the structure of a host model. The reference models operate on the inputs of the host model, thus do not require specific test scenarios for verification. The generated report contains verification results for the host model and each of its underlying reference model.
The harness verification method verifies reference models specifically, without the need of a host model. During the harness verification, a harness model is created for the reference model. The harness model allows custom input data to verify the reference model under specific test scenarios. The generated report features verification results only for the verified reference model.

The following sections will give a more detailed insight into the two verification methods, their pros and cons, their conditions and the situations in which they are the most practical.

4.2.1 In-loop verification

The In-loop verification method verifies reference models whilst being inside the structure of host models. It can provide coverage on multiple reference models and their host model at once. However, this method has its limitations and conditions. Below follows a list with the most relevant aspects related to this method.

- The in-loop verification method gathers coverage data on implemented reference models and their host models.
- The method can gather coverage on multiple reference models within the host model at once.
- The only input required for the verification function is the string name and location of the host model, assuming the model is compilable.
- The method has no way of handling inputs for the verification. The model designer is free to choose his way of handling test scenario inputs. The inputs will have to be loaded into the model before the verification.
- All reference models' simulation mode must be set to 'normal'. Reference models operating with an 'accelerated' simulation mode will not be covered by the verification.

The in-loop method only requires the model's name as a string and for it to be added to the current MATLAB directory. The host model should be compilable before hand, which means any inputs the host model relies on should be implemented before the verification. Test scenarios are provided as input for the host model. The reference model's inputs are determined by the actions of the host model.

The reason to exclude a fixed input format is to prevent the model designer to be subjected to a strict design format whilst designing the model. Think of blocks, signal names and the use of either to-workspace blocks, signal builders or other ways of implementing test scenarios into the model. Even with a pre-designed format, manual operations for the loading of data are unavoidable. Hence in this case freedom is granted to the model designer in which way test scenarios are loaded in the model.

The ability to verify multiple reference models and their host model in one go is a strong pro for the in-loop method verification. Though on the other side, it may be assumed that before reference models are implemented in larger complexes models, the reference models have already been subjected to thorough testing. If this is the case, then this verification method is only useful if the tester is focused on the behaviour of the reference model whilst it is operating inside the host model.
This verification will be interesting in the situation where there is an untested model with one or multiple reference models within and the behaviour of the host model and reference models is desired based on the host model's inputs. In such a situation the in-loop method verification is recommended.

4.2.2 Harness verification

The harness verification method is designed to verify singular models and reference models. The method creates a test harness structure for the to-be-verified model. Test scenarios are applied directly to the to-be-verified model, thus the scenarios can be designed specifically and verification can be done early in the design progress. Though, this method too has its conditions and limitations. The following list shows the most relevant aspects.

- The harness method gathers coverage data on singular models and reference models.
- The method places the to-be-verified model into a test harness with which inputs are directly connected on the to-be-verified model.
- The test scenarios require a strict format to be loaded as input.
- Inputs are applied directly on the to-be-verified models through signal builders.
- The model must have 'In' input blocks on its highest level, otherwise no inputs can be loaded. (Reference models have in/out blocks by default).
- The loaded amount of test scenarios must be equal to the number of model inputs. Otherwise the method will either remove input signals if too many or add constant-zero signals if too few. This in order to prevent MATLAB crashes.

The harness method applies input data directly on the to-be-verified model, instead of going through a host model. The input is applied to the model through an automatically generated and connected signal builder. The process of loading data in a signal builder requires a strict procedure. The function script *'input2builder.m'* converts input data to a format that is accepted by the signal builder. The input for the script is a structure with test cases. Each test case should contain a matrix with an input signal in each column. The first column requires to be the time vector. From then on, any column may contain an input signal, similar to the presented table below. The input data matrix can be created on the spot or loaded through a .mat file, as long as there is a structure with one or more test cases provided to the function.

$$'input' = \begin{pmatrix} time & signal\,1 & signal\,2 & \cdots & signal\,n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & \vdots \end{pmatrix} \tag{4.1}$$

The input data is checked for its amount of signals. If the amount of signals exceed the amount of model inputs, the excess input signals are removed. If the amount of signals are too few with regards to the model input, constant-zero signals will be added to the input data matrix to compensate for the lack of signals. This feature has been added to the method in order to increase robustness. Loading an incorrect amount of signals would otherwise result in MATLAB errors. During testing, these errors had the consequence to stop the process whilst the harness model is being created. The result was an incomplete harness model which caused errors in the following verifications. With the algorithm to add or remove signals in case its required, this problem has been solved.

Once scaled up or down, the input data matrix is converted into a cell array format suitable for the signal builder within the function script. The script loads the input data into the signal builder as a new group named 'Test Case' and sets that group as the active group. Previously similarly named groups will be removed and overwritten with the most recent test data.

The harness verification method requires the to-be-verified model to have input blocks. Reference models have in/out blocks by default. This means any reference model is suitable for this type of verification. There is no format required during the design of the model. The only format to be upheld is for the input data matrix.
The custom defined test scenarios are directly connected to the to-be-verified model. This requires specific test scenario for each reference model, however, this does allow for verification early in the design process of the reference model. This highlights the situation in which this method is effective: specific verification for reference models early in the design process.

4.3 Types of coverage

The MATLAB VnV toolbox is able to collect different types of coverage. Each coverage focusses on a different definition to asses the coverage of a model element. The toolbox user can decide which coverage types to take into account in the verification. For example, one can uncheck Look-up Table Coverage if one is not interested in that particular coverage. Or one could choose only Decision Coverage if that is the focus of the verification.
Below follows a list featuring all the types of coverage possible with the MATLAB VnV toolbox.

- Execution Coverage (EC)
- Cyclomatic Complexity
- Condition Coverage (CC)
- Decision Coverage (DC)
- Modified Condition/Decision Coverage (MCDC)
- Look-up Table Coverage
- Relational Boundary Coverage* (only in MATLAB 2016a or newer)
- Saturate on Integer Overflow Coverage
- Signal Range Coverage
- Signal Size Coverage
- Simulink Design Verifier Coverage* (only in MATLAB 2016a or newer)
- S-function Coverage
- External MATLAB files Coverage

The following subsections will describe the definition of each type of coverage. After reading these subsections, the reader should be able to determine which coverage type to use in which situation.

4.3.1 Execution Coverage (EC)

Execution Coverage is the most straight-forward type of coverage. It applies to all Simulink elements. It simply determines whether or not the element has been executed during the simulation.

4.3.2 Cyclomatic Complexity

Cyclomatic complexity is a ratio assigned to a model to assess its structural complexity. The value for the cyclomatic complexity is determined by an approximation of McCabe's complexity measure for code generated from the model. MATLAB VnV computes the cyclomatic complexity of an element using the following formula:

$$c = \sum_{1}^{N} (o_n - 1) \tag{4.2}$$

N is the number of decision points of the element and o_n is the number of outcomes for the nth decision point.

4.3.3 Condition Coverage (CC)

Condition coverage analyses Simulink elements that give a logic output, either TRUE or FALSE. For example, logical Operator Blocks and Stateflow transitions. MATLAB VnV defines full condition coverage for a logical element when each of the block's input trigger both a TRUE and FALSE statement at least once during the simulation.

4.3.4 Decision Coverage (DC)

Decision coverage analyses elements that contain decision points. Block examples that get this coverage are Switch blocks and Stateflow states. The decision coverage is the percentage of the amount of traversed simulation paths opposed to the amount of possible simulation paths. When verifying a switch block with four outputs and three of the four options are selected during the simulation, the decision coverage of the block would be 75%.

4.3.5 Modified Condition/Decision Coverage (MCDC)

Modified Condition/Decision coverage is a collaboration of condition and decision coverage for elements that output logic and Stateflow transitions. It further extends the previous two types for the sake of gaining information on independency of logical block inputs and transition conditions.

- A block achieves full coverage when a change in each input, independent of other inputs, triggers a change in the output during a simulation.
- A Stateflow transition achieves full coverage when a change of each individual condition triggers the transition least once during a simulation.

It has to be noted that the MATLAB VnV MCDC coverage may not achieve full decision or conditions coverage. It is possible to achieve 100% MCDC coverage *without* achieving 100% decision coverage. Their definitions do not have a relation to each other.

4.3.6 Look-up Table Coverage

Look-up Table coverage records coverage for blocks that provide values from a pre-defined n-dimensional table, interpolating between and extrapolating from table entries. Blocks that fall under this type of coverage are for example 1-,2-, n-D Lookup table blocks. Full coverage is given to look-up tables that use each interpolation and extrapolation interval at least once during the simulation. On top of that, the coverage also provides details on the frequency each interval is used. The information is displayed in the generated HTML report with a colour map that displays each used interval. A colour map is generated for every block that can receive look-up table coverage.

4.3.7 Relational Boundary Coverage

The Relational boundary coverage checks the implicit and explicit relational operation of blocks, Stateflow charts and MATLAB function blocks. Blocks like the Relational Operator and If are explicit relational operators. Blocks that fall under implicit relational operations are blocks such as Abs and Saturation.

- MATLAB VnV tests the relational operators in the situation of equal operand values (only applicable if both operands are either integers or fixed-point numbers).
- MATLAB VnV test operand values that differ by the defined tolerance (Default: Absolute tolerance is 1e-05. Relative tolerance is 0.01).

The latter part of the relational boundary coverage applies to all operands. Integer and fixed-point numbers have a fixed tolerance. For floating-point numbers, pre-defined or custom-defined tolerances can be taken into account.

4.3.8 Saturate on Integer Overflow Coverage

The Saturate on integer overflow coverage applies to blocks that saturate on integer overflow, like the Abs, saturation or common math block. It occurs when a value's bit size is greater than its allocated size.
Other blocks with the saturate on integer overflow parameter will be taken into the coverage, if the parameters is enabled. The coverage records the amount of times the element saturates on integer overflow. Full coverage is reached when the element saturates at least once and does not saturate at least once during the simulation.

4.3.9 Signal Range Coverage

Signal range coverage records the minimum and maximum signal value of an element during the simulation. Only blocks that have an output signal qualify for this type of coverage. Blocks which provide control outputs, outputs that initiate other elements in the model, are excluded from this signal range coverage. The software can not record signal range coverage if the total amount of signals exceeds 65535 or if a signal has a width that exceeds 65535 signals.

4.3.10 Signal Size Coverage

Signal size coverage is meant for blocks that provide variable-sized output signals. The minimum, maximum and allocated size of a block's variable-sized output signal is recorded. Similar to the signal range coverage, the software cannot record this type of coverage if the total amount of signals exceed 65535 or if a signal's width exceeds 65535.

4.3.11 Simulink Design Verifier Coverage

Simulink Design Verifier coverage gathers coverage data on specific blocks and code generation functions that were introduced with the MATLAB VnV toolbox. These blocks are designed for design verification goals. Below follows a list of all the MATLAB VnV specific blocks and code generation functions that this type of coverage analyses.

- Test Condition block
- Test Objective block
- Proof Assumption block
- Proof Objective block
- sldv.condition function
- sldv.test function

- sldv.assume function
- sldv.prove function

These blocks and functions require the Simulink Design Verifier license. Without the MATLAB SDV license, the model can't be analysed using the aforementioned blocks and functions. However with the MATLAB VnV license, it is possible to gather coverage data on the aforementioned blocks and functions. The blocks and function are used to test wether a condition, objective or assumption becomes true during the simulation. Full coverage is achieved when the state of the block becomes true at least once during the simulation.

4.3.12 External coverage options

MATLAB VnV also has the ability to gather code coverage on external sources of code like C/C++ S-Functions and external MATLAB Functions. To gather coverage on S-functions and MATLAB functions, the functions will have to be present as a block in the Simulink model structure. When enabled, coverage is recorded C/C++ S-Functions. An specific chapter for all S-functions results is created in the generated HTML report. Multiple S-functions are weighed into a single coverage value. For example S-function block 1 achieves a decision coverage of (3/4) 75%. S-function block 2 has a decision coverage of (10/20) 50%. The combined result for S-functions would become (13/24) 54%. S-Function can receive the following types of coverage.

- Cyclomatic Complexity
- Decision Coverage (DC)
- Condition Coverage (CC)
- Modified Condition/Decision Coverage (MCDC)
- Percentage of statements covered

External MATLAB functions used in Simulink blocks will receive coverage on its written code. The following types of coverage are applicable to MATLAB functions.

- Decision Coverage (DC)
- Condition Coverage (CC)
- Modified Condition/Decision Coverage (MCDC)
- Simulink Designer Verifier Coverage
- Relational Boundary Coverage

The decision coverage covers if, switch, for and while commands. Condition and MCDC gathers coverage on if and while statements. Simulink Designer Verifier Coverage enables coverage on the sldv functions introduced with the MATLAB VnV toolbox. If the MATLAB function contains relational operators, the toolbox will gather relational boundary coverage on the written code.

4.4 Results review

The MATLAB VnV toolbox has the feature to automatically generate a verification HTML report at the end of a verification simulation. The verification report displays the gathered coverage for each coverage type and system/subsystem inside the verified model. The generated report stores the details about the model, the time and date of the verification and the test details. It allows itself to be stored and used as reference work. The generated report consists of the following five sections.

- Analysis Information

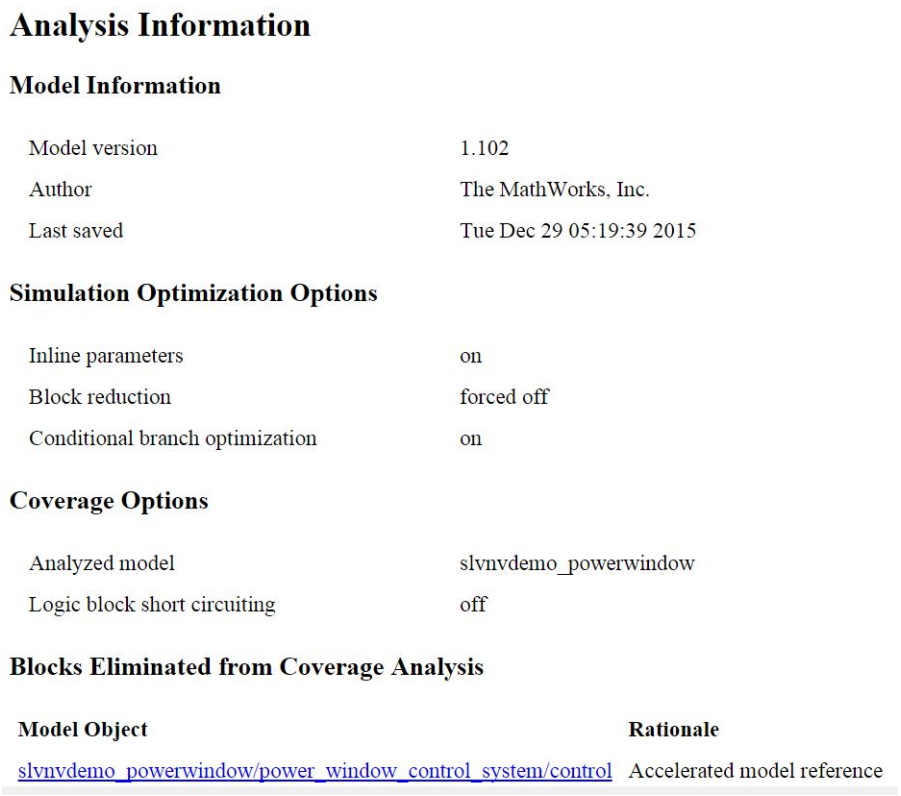
- Tests
- Summary
- Details
- Signal Ranges

Each of the five sections provides information on a part of the verification. 'Analysis information' contains general information on the verified model and the verification options. The 'Tests' sections provides information on the tests that have been used to gather the coverage data. The 'Summary' includes the percentile score of all the Simulink elements in the model. The 'Details' section contains details on the elements that have not reached full coverage. 'Signal Ranges' presents the minimum and maximum values for each element in the model.
Each of the five sections of the generated report are described in more detail in the following five subsections.

4.4.1 Analysis information

The section 'Analysis information' contains information on the verified model: its version, author and time and date of its last changes. It also displays several optimisation and coverage options selected for the test and notifies the reader incase a system/sub-system or specific Simulink blocks have been excluded from the verification.

Figure 4.1 shows the 'Analysis information section' for an electrical window system model. The section shows the model information and which options have been used during that particulare simulation. For example: The block reduction optimization had been forced off for this case. It also mentions the elimination of a reference model from the verification with the reason that it uses an accelerated simulation mode (coverage can only be gather on 'normal' simulation mode).



Analysis Information

Model Information

Model version	1.102
Author	The MathWorks, Inc.
Last saved	Tue Dec 29 05:19:39 2015

Simulation Optimization Options

Inline parameters	on
Block reduction	forced off
Conditional branch optimization	on

Coverage Options

Analyzed model	slvnvdemo_powerwindow
Logic block short circuiting	off

Blocks Eliminated from Coverage Analysis

Model Object	Rationale
slvnvdemo_powerwindow/power_window_control_system/control	Accelerated model reference

Figure 4.1: The 'Analysis' section shows background information on the verified model and coverage options.

4.4.2 Tests

The 'Tests' section provides information on tests that have been performed in order to gather the coverage displayed in the report. The amount of tests are presented as well as the time and data of their execution start and end.
Within MATLAB VnV, the collected coverage is stored in a cvdata object. The cvdata object can consist of structures, doubles and character arrays from within. The cvdata objects are summable with each other. This allows the possibility to determine the total coverage over multiple different tests. Different tests are generally used to determine the verification impact for a certain set of inputs. Figure 4.2 shows a verification that has been run with multiple tests.

Tests

Test#	Started execution	Ended execution
Test 1	24-Oct-2016 16:51:40	24-Oct-2016 16:51:41
Test 2	24-Oct-2016 16:51:51	24-Oct-2016 16:51:51
Test 3	24-Oct-2016 16:51:40	24-Oct-2016 16:51:51
Total	24-Oct-2016 16:51:40	24-Oct-2016 16:51:51

Figure 4.2: The 'Tests' section in the generated report shows information on the tests used to gather the coverage.

4.4.3 Summary

The percentile score of each Simulink element in the verified model is displayed in the 'Summary' section. This section contains a list of all the Simulink elements (system, sub-systems and blocks), together with their percentile coverage score in each of the selected coverage type. The score is visually displayed using blue/red coloured bars. The percentage of coverage is also displayed as an integer. Simulink elements that do not apply to the selected coverage type (like Switch blocks can not apply for Look-up Table coverage) receive the status 'NA'.

Figure 4.3 displays the 'Summary' part for the report generated for a Cruise Control model. The first item in the list shows the complete model and its total coverage for each type of coverage (33% for decision, 100% for condition, etc...). Every item below the main model, represents either a subsystem or a block. Their indent is dependent on their level location in the model.

Summary

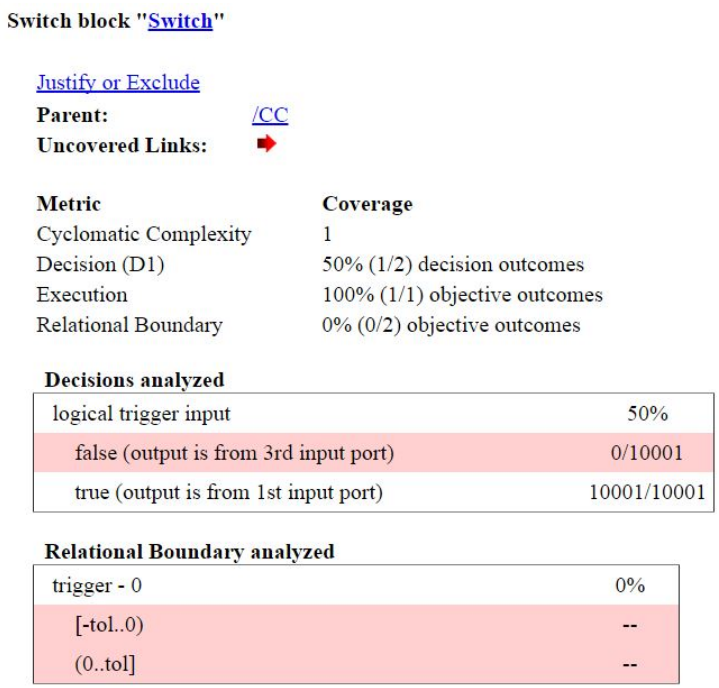
Model Hierarchy/Complexity	Test 1											
	D1		C1		TBL		Execution		Relational Boundary		Saturation on integer overflow	
1. CC	60	33%	100%		1%		30%		13%		9%	
2. . . . Curve Handling	5	63%	100%		2%		80%		75%		50%	
3. . . . . Dynamic Limiter	2	50%	NA		NA		100%		NA		50%	
4. . . . Power Limitation	6	80%	NA		NA		100%		50%		50%	
5. . . . . Saturation Dynamic	2	100%	NA		NA		100%		NA		NA	
6. . . . Setpoint Profile Generation	6	89%	NA		NA		100%		25%		NA	
7. . . . . Acceleration Profile	6	89%	NA		NA		100%		25%		NA	
8. . . . . Acceleration Profile	6	89%	NA		NA		NA		25%		NA	
9. . . . pre-defined profiles	41	12%	NA		0%		10%		2%		0%	
10. . . . . logic	NA		NA		NA		100%		0%		NA	

Figure 4.3: The 'Summary' section displays the percentile coverage score for each of the Simulink elements in the verified model.

4.4.4 Details

The 'details' section contains detailed information for Simulink elements that have not received full coverage. The lay-out for the section depends on the type of coverage and on the Simulink element. In general, the argument why this element hasn't received full coverage is highlighted together with the element's execution numbers or percentages. The element's information is given like its name, parent and whether it contains elements that have not received full coverage. When viewing the report in the MATLAB web browser, the location of the element in the model is linked to the location of the element in the HTML report. With this connection enabled, the elements can be justified or excluded for future verification simulations.

Figure 4.4 shows the details for a Switch block that hasn't achieved full coverage. The Switch block has been tested for the decision, execution and relational boundary type of coverage. The percentages for the types of coverage are displayed together with the amount of requirements needed for full coverage. Another feature is the output per simulation step. As can be seen for the Switch, it fails to produce a false output (10001 times out of 10001 simulation steps, the output has been true). For its relational boundary aspect, it fails to operate within toleration regions, hence it scores 0%.



Switch block "Switch"

Justify or Exclude

Parent: /CC

Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	1
Decision (D1)	50% (1/2) decision outcomes
Execution	100% (1/1) objective outcomes
Relational Boundary	0% (0/2) objective outcomes

Decisions analyzed

logical trigger input	50%
false (output is from 3rd input port)	0/10001
true (output is from 1st input port)	10001/10001

Relational Boundary analyzed

trigger - 0	0%
[-tol..0)	--
(0..tol]	--

Figure 4.4: The 'details' section provides details on the conditions of Simulink elements that have not gathered full coverage.

4.4.5 Signal Ranges

The 'signal ranges' section provides information on the minimum and maximum value that each Simulink element has produced during the simulation. Figure 4.5 shows an examples of the signal ranges for a Cruise Control model. The list of Simulink elements is presented with indents corresponding to their hierarchy in the model structure. The minimum and maximum values are readable in the two right-side columns.

Signal Ranges

Hierarchy	Min	Max
CC		
. . . proportional gain	-12.4	2
. . . Sum	-68.8889	11.1111
. . . Sum1	-12.4	2
. . . Constant	1	1
. . . Constant1	50	50
. . . Curve Handling		
. . . . . Abs	0	0
. . . . . 1-D Lookup Table	12.7224	12.7224
. . . . . Switch		
. . . . . out[1]	0	0.99
. . . . . out[2]	0	12.7224
. . . . . Relational Operator	0	1
. . . . . Constant1	0	0
. . . . . Constant2	0	0

Figure 4.5: The 'Signal ranges' section displays the minimum and maximum values all blocks have achieved during the simulation.

4.5 Verification summary

This chapter has provided insight in the process of verification through the MATLAB VnV toolbox. Code coverage verification is required to assess how many model elements are executed during a simulation. Code coverage is expressed in a percentile score. A 100% coverage score indicates the model has executed all model elements during the simulation. The code coverage is a tool that aids in the verification process by highlighting untested elements. The model designer still has to decide on the appropriate action to increase coverage. In case coverage is not enough, one can add test scenarios that execute the untested model elements. Another option is to remove the untested elements if they do not contribute to the system's functionality. The untested elements could also be replaced by better optimised code.

Coverage verification in MATLAB is possible through two methods: either in-loop verification or harness verification. In-loop verification focusses on nested models. Test scenarios are not required for in-loop verification, as the models are executed by the behaviour of the root model. In-loop verification provides an indication on the nested model's behaviour during operating conditions. Harness verification verifies singular Simulink reference models. It does so by placing the model in a harness format. The harness format allows the model to be verified using custom defined test scenarios. This allows the harness verification method to be used early in the development phase.

Different definitions of coverage are available within the MATLAB VnV toolbox. In total, thirteen different types of coverage are available to assess models. Each coverage type focusses on a particular group of Simulink model elements. Decision coverage focusses on Simulink elements that trigger a decision, like Switch blocks. Condition coverage for example, assess the coverage of Simulink elements that output a logic value, like AND, OR, and IF blocks.

The verification framework presents its results in either a generated HTML report or directly in the model structure through the Interactive Model View. Both options can be turned on or off, depending on the preferences. The HTML report displays the general information of the model, the coverage score in percentages for the model and each individual subsystem and block, arguments as to why certain elements have not received full coverage and the peak signal values during the simulation.
The Interactive Model View is able to present the same information in the model structure. The coverage score for a model element is displayed in an information window when clicking on the element. The information displays the coverage score of each coverage type for that particular model element and arguments if the element has not received full coverage. Signal properties are displayed in the information window when the signal line in the model structure is selected.

Chapter 5

Strategies in reaching coverage

Chapter 3 has described the general features of the verification framework and explained the definition of coverage and types of coverage. This chapter will focus on the influence of three different testing strategies on a model's gained coverage. The question this chapter will answer is: "what is the best strategy to increase model coverage?". To answer this question, a series of tests have been performed on an arbitrary Simulink reference model. The gathered model coverage as well as the output has been analyzed. What has to be mentioned is that the ultimate desire is for a tested system to have 100% coverage, whilst having complete transparency in its behaviour. Coverage is a good aspect, as it shows all the untested elements, but it is worth little if the system does not fulfill its functional purpose.
The three testing strategies are the following:

- Monkey testing
- Black-box testing
- White-box testing

The following section will discuss each of the three testing strategies. Their general definition will be mentioned as well as their impact on model coverage, as it is noticed during the tests.
An important thing to note: these sections will not present the actual coverage percentage and model outputs. Because each model's results is unique. The actual coverage gain due to each test strategy is very specific to the model's structure and algorithm. Focusing on specific values for this model, will most likely lead to false expectations for other models. The analyses lies on general influence to determine a recommended testing approach.

5.1 Monkey testing

Monkey testing is a method of automated software testing by providing the system with random inputs. It is predominantly used to find bugs or system crashes, but can also be used for checking the behaviour of the system on random inputs.
Many believe the name monkey testing is derived from the *infinity monkey theorem*. The theorem states that "if a thousand monkeys with a thousand typewriters type keys for an infinite amount of time, then it is almost surely they will type out the entire works of Shakespear." [7].

Distinction can be made between two types of monkeys. Smart monkeys, which have a brief idea about the systems and how to test it, and dumb monkeys, who operate with no knowledge whatsoever on the system.

5.1.1 Dumb monkey testing

A dumb monkey has no knowledge of the system or how to operate it. Its behaviour is purely stochastic and will operate the system without knowing whether its actions are valid or not.

The characteristics of a dumb monkey are identified below:

- They have no idea about the application
- They do not know whether the inputs they are providing are valid or invalid
- They test the application randomly and are not aware of any starting point of the application or the end to end flow.
- Though they are not aware of the application, they can identify bugs like environmental failure or hardware failure.
- They don't have any idea about the UI and functionality. [7]

The principle of a dumb monkey is its complete random behaviour. It provides random inputs to test a system. It can not distinguish legal or illegal inputs and it is not aware where to start in the system and were to end. The number of bugs it can find is not extensive as a smart monkey, however it will catch the important bugs that causes the system to hang, break or crash. A troubling matter is that it is often hard to reproduce a bug found by the dumb monkey, as it has been initiated fully randomly. The dumb monkey lacks in intelligence, so is easier to set-up for automated testing.

Translating the general concept of a dumb monkey to this situation of code coverage testing, would be similar to using a stochastic signal like white noise for input. White noise mimics the dumb monkey's behaviour in that it is a random signal with different amplitudes (in time) occurring at different frequencies. It is similar to pressing random keys at different time intervals.
A good aspect of using a dumb monkey strategy to gain coverage is that with little random input a decent amount of coverage can be gained. Because the input reaches a different value for every sample in time, it can execute a lot of Simulink elements in the model in a relative short time span.
As mentioned before, reproducing bugs found by dumb monkeys is a difficult matter. The same has been experienced during coverage tests. Using random noise as input for code coverage test has a high chance of covering a significant part of the model, though the exact percentage scores fluctuate for each different test. As expected from a fully random signal, it is predominantly chance whether certain elements are triggered during each simulation.

5.1.2 Smart monkey testing

A smart monkey has a general idea of how the system works and how to initiate/test it. In the case of a calculator, a smart monkey would know what the button is to start the application, to use the operators and to exit the application.

A smart monkey is identified by the following characteristics:

- Have a brief idea about the application
- They know where the pages of application will redirect to
- They know whether the inputs they are providing are valid or invalid
- They work or focus to break the application
- In-case they find an error, they are smart enough to file a bug
- They are aware of the menus and the buttons
- They are suited for stress and load testing [7]

A smart monkey has a general sense of the system and uses that to provide random input within bounds of what is realistic for the system. It has knowledge of the function of the system, where it starts and where it ends. This allows the smart monkey to go into depth of the system and find and file minor bugs. It does however require time to set-up as it takes effort to program the AI for automated testing, making it a more time-consuming and costlier than a dumb monkey.

A smart monkey has the ability to know the general features of the system and whether inputs are valid or invalid. So, for code coverage verification it would know what the inputs and outputs of the system are and what they mean. Take for example an automotive Adaptive Cruise-Control (ACC) system. A velocity path is required for its set-speed reference. A smart monkey would understand the realistic bounds of speed for an automotive vehicle and suggest random values between 10 to 250 km/h. A dumb monkey would enter impossible speeds reaching up to a 10.000 km/h or negative velocities, something out of the range of the system.

Both monkeys have their uses. A dumb monkey blindly executes elements in the model which causes coverage randomly throughout the model. A smart monkey is more focussed and is a slightly better option to cover specific elements through its input-awareness. Though, its advantage is compensated by its set-up time and effort.
Because of the random inputs of both monkeys, the system's behaviour is difficult to analyse. With random inputs, the outputs become random as well. However, measuring both the input and output, one can perform a frequency analysis and analyse the system's behaviour in frequency domain.

5.2 Black-box testing

Black-box testing is a testing strategy that focusses on the system's functionality. Its name comes from the fact that during these tests, the system is being treated as a black box. It involves testing without knowledge about the internal structure of the system. The tests proceed by determining specific inputs for the system and analysing the output behaviour. As long as the functionality of the system does not change, test cases remain the same, even if the internal structure of the system changes [6].
The test cases used for functionality tests are often based on early-development requirements or safety-criteria. These define an operating scenario and have certain expectations of how the system should behave.

The advantage of black-box testing is the separation between the design and testing activities. The strategy is commonly used in industry where testing and development are separate departments. It can provide an external perspective on the functionality of the system, even if the tester is not proficient with the internals of the system or does not have access to the internal code.
The disadvantages of black-box testing is that it is only limited to functionality tests due to the lack of access of the system's internal structure. Although this isn't always the case, it may possibly allow for inefficient testing or blind coverage if the tester is not known with the system.

In terms of code coverage, black-box testing is useful in determining redundancies or a lack of function features. It is similar to the smart monkey's strategy, in that the tester is aware of the inputs and outputs but not of the internal system's structure. Opposed to the monkey testing, black-box testing tests with specific test scenarios.
Take the ACC system again as an example. The system controls the longitudinal behaviour of a vehicle in, mostly, highway driving situations. It controls the vehicle based on several inputs that specify its spatial freedom in front of it. Applying black-box tests for this system would require test scenarios that would trigger the system's features, like scenarios in which the car in front is driving at a constant speed, traffic jams, situations that call for an emergency stop, etc... These test scenarios are derived from the early-development functional requirements. Using these scenarios allows for proper output analyses. The gathered code coverage would also reflect a more realistic scenario.
Having defined the test scenarios and verifying the system, the chance exists that the code/model does not receive full coverage. This is possible due to the following reasons:

- The test scenarios are not extensive enough to cover the complete model; ergo, test scenarios have to be added that compensate the lack of coverage.
- The model has redundant or unnecessary features, which lie outside the boundary of the functional requirements. To solve this, features should be removed.

The amount of gained coverage using this method is dependent on how much the system reflects the functional requirements. If the system has no redundancies and fully defined test scenarios the system should reach full code coverage.

5.3 White-box testing

White-box testing is also known as clear-box, glass-box or transparency-box testing. As the name indicates, it is testing with a visual on the internals of the system. The test strategy focusses on structural testing methods. It is often used by a person who is known with the system's internal structure, often someone involved in the development [6].
Testing is done with respects to the system's structure. For example, if a sub-system of the model is only executed for specific conditions, than a test is defined to reproduce those conditions. White-box testing provides an internal perspective on the system.

The advantage of white-box testing is that it is an efficient strategy in finding errors and problems as specific parts of the model can be targeted. When done by a tester with internal knowledge of the system, the strategy is generally thorough and close to maximum coverage. It also helps for better code optimisation.
The disadvantages of white-box testing is that it is difficult to find redundancies in the system as there is no link to the functionality described by the early-development requirements. Another disadvantage is that it requires code access and high-level knowledge of the internal structure. These tests are often performed by the same department that develops the system.

Replicating this strategy to suit code/model coverage, it is similar to targeting specific Simulink model elements and triggering their condition. The test cases involve inputs that are designed specifically to access certain parts of the model, opposed to being based on the functional requirements as is the case with black-box testing. For example, the ACC model contains a full-stop logic that realizes smooth braking to a full stop of the host vehicle, if its preceding vehicle comes to a full stop. The full-stop logic is a complex system which features many conditions. An internal perspective allows for inputs specifically designed to trigger these features.
On the down side, the system's behaviour becomes indefinite due to the lack of realistic test scenarios. Output analysis is difficult in this case. This strategy basically trades in functional focus for structural focus. Nevertheless, it is the strategy that reaches full coverage with the least effort.

5.4 Recommended approach

Monkey testing, black-box testing and white-box testing all have their use in terms of valid strategies for coverage tests.
Monkey testing is useful as it provides random inputs that cover model elements without setting up an exact testing procedure. It can highlight bad model design or mistakes outside the scope of the requirements. It provides an out-of-the-box perspective.
Monkey testing is useful to form a basis of coverage for a system, although coverage may fluctuate between different tests due to the random inputs. It is especially useful for a first indication or in the case there are no test scenarios defined for the system. Measuring both the input and outputs allows for frequency analysis, which indicates the behaviour of the system in frequency domain.

Black-box testing is best to be used as a functional approach. It derives test scenarios from the functional requirements of the system and tests without required knowledge of the internal structure.
The black-box strategy is particularly useful in gathering coverage whilst being able to analyse the system's functional behaviour in realistic scenarios. Scenarios should either be added or redundancies should be removed until the point where the model is fully covered and the system behaves as expected.

White-box testing is the case wherein testing is done to specifically cover a part or all model elements. It requires a focus on the internal structure of the model.
White-box testing is recommended when there are no test scenarios available for a model. Coverage is gathered by designing inputs to specifically trigger conditions in the model.
This strategy is best used for code/model optimisation. It has no focus on system functionality and behaviour. Therefor, it is recommended to be paired with black-box testing.

The best approach would be a combination of all three tests: monkey testing for random load tests and frequency analysis, black-box testing for functionality verification and white-box testing for code/model optimisation. However, the choice of strategy is dependent on whether there are test scenarios available and/or testers with knowledge of the internal structure of the system.

Chapter 6

Conclusions and Recommendations

6.1 Conclusions

The goal of this project was to improve the overall quality of the Advanced Drivers Assistance Systems (ADAS) Simulink models, produced by TNO Helmond's Integrated Vehicle Safety (IVS) department. In order to achieve this, a ISO 26262 compliant code coverage verification framework has been created for the verification of MATLAB Simulink reference models. Additionally, a comparison function has been created which allows comparison between similar generations of Simulink models. The function allows quick feedback in whether output signals between different generations of models significantly differ.

The ISO 26262 standard provides specifications and regulations for the development of electrical and electronic (E/E) systems for automotive vehicles. The international standard helps to maintain quality and functional safety for series production passenger cars. ISO 26262 introduces Automotive Safety Integrity Levels (ASIL), ranging from A to D. The ASIL of a system is determined by analysing the probability of exposure, controllability and severity of failure of the system in hazardous events. The ASIL determines the amount of verification procedures required to guarantee the functional safety of the system.
Another relative aspect ISO 26262 introduces is the Tool Confidence Level (TCL). The TCL is a grade that rates the capability of a verification tool. Like with the ASIL, the TCL determines the demands on tool qualification.

Analysis has been performed on five notable software platforms. These five notable software platforms allow code coverage verification with compliance to the requirements of ISO 26262. The five platforms are: MATLAB Simulink, Vector Software, Parasoft, CertTech and LDRA. From these five, MATLAB Simulink is the most compatible choice as it allows model coverage on TNO's ADAS models without the need for conversion. It is also software that is familiar to TNO, essentially reducing the time to conquer the *learning curve*.

The code coverage verification framework can gather code/model coverage on the models through two methods: in-loop verification or harness verification. In-loop verification is a method which is designed for one or multiple reference models that operate within a host model. Harness verification is specifically for singular models that require to be verified using test scenarios. Both methods gather coverage and are able to present the results in an HTML report or directly in the model structure through the interactive model view. The results show the coverage as a percentile score. The overall percentile score gives an indication in how many model elements have been executed during the verification.

There are three recommended strategies in order to improve a model's coverage; either through monkey testing, black-box testing or white-box testing. Monkey testing is a strategy where random signals are provided as inputs. Monkey testing is easy to set-up and gains a solid base of coverage, though coverage may fluctuate among different tests due to randomness. Black-box testing increases coverage by focussing on the functionality of the system. The strategy requires specific test scenarios based on

early-development functional requirements. White-box testing increases coverage by designing inputs to execute specific parts of the internal structure. It requires internal code knowledge, though is useful for code optimisation. The choice of strategy is dependent on the available resources, though all three are recommended for the optimal coverage results.

6.2 Recommendations

The created model/code coverage functions require the MATLAB Verification and Validation tool. Using these functions would requires an active license to the Verification and Validation toolbox. A recommendation would be to extend the use of the Verification and Validation in the future, using the toolbox's additional features. Model/code coverage is merely one feature of the toolbox. To use the toolbox's license to its full potential, one might consider the following features of the toolbox.

- Requirements Management Interface (RMI)
- IBM Rational DOORS Surrogate Module Synchronization
- Model Advisor

The Requirements Management Interface (RMI) is a tool that helps establish links between Simulink and Stateflow elements to requirements. The management interface is used to create the requirement traceability with the Simulink and Stateflow elements. After a link has been created, navigation is possible from the requirements document to the model and vice versa. Embedded link in the requirements document opens the specific element which the requirement is related too. In the model, elements are highlighted and/or tagged which relate to requirements in an external document. The RMI feature is designed for system development through model-based design. It allows close monitoring on requirements during any stage of the development.

Next to Microsoft Word, Microsoft Excel, PDF or HTML files, the requirements traceability tool also supports integration with IBM Rational DOORS. DOORS creates a surrogate module after synchronizing with the model. The module shows a representation of the Simulink model hierarchy and the related requirements. It works as a stand alone program, so requirements are able to reviewed, edited, verified and navigated without active Simulink software. It compliments the model/code coverage functions, in that DOORS allows for analysis of requirements coverage. DOORS acts as an intermediate navigation tool between the requirements and the Simulink and Stateflow model and is beneficial to further extend the model-requirements traceability.

Another tool that aids in the development of systems through model-based design, is the Model Advisor. The Model Advisor allows the user to enforce standards or guidelines for model development. Custom-defined standards or guidelines can be created through the tool's Configuration Editor and applied on any Simulink or Stateflow model. The Model Advisor can be set-up to provide an error or an automated fix if the model structure violates the enabled standard's format. This tool is beneficial for consequent model development on a large scale. Its main principle is that each produced model contains the same standard-bounded format. When performed correctly, it reduces most of the conversion required for the testing and implementation stage.

Bibliography

[1] Juergen Belz. The iso 26262 safety lifecycle. pages 0–1, 2011.

[2] CertTech. Bullseyecoverage tool qualification kit (tqk). *CertTech handout*, pages 0–1, 2014.

[3] IEC 61508:2010 CMV. Functional safety of electrical/electronic/programmable electronic safety-related system. *International Organization for Standarization*, 2, 2010.

[4] Mirko Conrad. Verification and validation according to iso 26262: A workflow to facilitate the development of high-integrety software. *The MathWorks, Inc.*, pages 0–8, 2011.

[5] Mirko Conrad, Patrick Munier, and Frank Rauch. Qualifying software tools according to iso 26262. *The MathWorks, Inc.*, pages 0–12, 2011.

[6] Viktor Farcic. Black-box vs white-box testing. *Technology Conversations*, pages 0–1, 2013.

[7] Exforsys Inc. What is monkey testing? *IT Training and Consulting - Exforsys*, pages 0–1, 2011.

[8] National Instruments. What is the iso 26262 functional safety standard? *NI White papers*, 1, 2013.

[9] ISO-26262-1:2011. Road vehicles - functional safety. *International Organization for Standarization*, 1, 2011.

[10] ISO-26262-3:2011. Road vehicles - functional safety: Concept phase. *International Organization for Standarization*, 1, 2011.

[11] ISO-26262-8-11:2011. Road vehicles - functional safety: Software tool qualification. *International Organization for Standarization*, 1, 2011.

[12] Min Koo Lee, Sung-Hoon Hong, and Hyuck Moo Kwon. Incorporating iso 26262 development process in dfss. *Proceedings of the Asia Pacific Industrial Engineering and Management Systems Conference*, pages 0–1, 2013.

[13] Refayet Ullah Mirdha. German certification firm plans expansion in bangladesh. *The Daily Star*, pages 0–1, 2009.

[14] MISRA-C:2004. Guidelines for the use of the c language in critical system. *MISRA Limited*, pages 0–116, 2004.

[15] MISRA-C++:2004. Guidelines for the use of the c++ language in critical system. *MISRA Limited*, pages 0–220, 2008.

[16] Mass Natick. Simulink verification tools qualified to iso 26262. *The MathWorks, Inc.*, pages 0–1, 2011.

[17] Parasoft. Iso 26262 software compliance with parasoft: Achieving functional safety in the automotive industry. *Parasoft White papers*, pages 0–11, 2016.

[18] Parasoft. Satisfying asil requirements with parasoft c++test: Achieving functional safety in the automotive industry. *Parasoft White papers*, pages 0–7, 2016.

[19] Vector Software. Using vectorcast to satisfy software verification and validation for iso 26262. *Vector Software Inc White Paper*, pages 0–15, 2015.

[20] LDRA Software Technology. Automate misra coding standards compliance. *LDRA leaflet*, pages 0–2, 2013.

[21] LDRA Software Technology. Ldra: Code coverage analysis. *LDRA leaflet*, pages 0–1, 2013.

Appendix

This appendix contains a collection of all the MATLAB function scripts that have been created throughout the project. This appendix is written as a reference work with the code details for all the created functions. The following lists highlights the function scripts that are presented in this appendix.

- Main function: In-Loop Verification Function
- Main function: Harness Verification Function
- Main function: Compare Models Function
- Sub-function: Test Case to Signal Builder Format Converter
- Sub-function: Output Data to Workspace Converter
- Sub-function: Simulation Output Plot Function
- Sub-function: Significant Difference Plot Function

Each function is described in its following respective section. The function's particular goal is explained as well as its syntax. The inputs required for the use of each function is listed as well as explained which format they require and to what end they are used. The same is done for the outputs the functions produce. What are they and in which format are they retrieved? The more peculiar parts of the function's code are highlighted and reasons are given for the opted code structure.

Main function: In-loop Verification Function

The In-loop Verification Function script uses the in-loop verification method to gather coverage on the to-be-verified model. This method focusses on Simulink models with one or multiple reference models or subsystems within. Coverage is gathered for all reference models and subsystems within the root model and displayed individually for each system.

This script is particulary useful in the situation where multiple Simulink reference models need to be verified or when nested models require verification. It is also useful in the situation where there are no test scenarios defined specifically for the nested model. The verification gathers coverage on the behaviour of the nested models within the root model.
The downside of this method is that there is no possibility to verify the nested models with specific defined inputs. The nested models operate within the root model and are therefor dependent on the host model's behaviour. This method does not specify host model inputs. The reason being that it would subject the model designer to a strict format with many unavoidable manual operations. In order to allow this method to be applicable for a wide range of models, inputs for the host model are left out and should be loaded-in through any of the model designer's format, as long the host model is compilable before the verification simulation starts.

The inputs that this script requires are the following:

- The to-be-verified model's name in string and, if applicable, the path to the model.
- Settings which specify the coverage types and optimization options to be taken into account for the verification.

The script outputs the following:

- An html report containing the analysis information, coverage score summary and the verification details of the simulation.
- Interactive model view containing colour masks and verification details inside the model.
- Data plots of every data signal of the model connected to an out-block.

This method is applicable for a wide range of model formats. However, it does have its limitations and conditions. Below follows a list with important notes for using the method.

- The Verification and Validation toolbox is required for the use of this script.
- The in-loop verification method gathers coverage data on implemented reference models and their host models.
- The method can gather coverage on multiple reference models and subsystems within the host model at once.
- The only input required for the verification is the string name and location of the host model, assuming the model is compilable.
- The method has no way of handling inputs for the verification. The model designer is free to choose his way of handling test scenario inputs. The inputs will have to be loaded into the model before the verification.
- In order to collect and plot output data, the model should contain out-blocks.
- The function can cope with 2D and 3D simout structures. Higher dimension structures are not applicable.
- All reference models' simulation mode must be set to 'normal'. Reference models operating with the 'accelerated' simulation mode will not be covered by the verification.

The model name of the to-be-verified model is required first in the script. The model name should be provided as a string and can refer to the root model or a specific reference model or subsystem within the root model. A nested model can be specified if the user is only interested in coverage of that specific system.

The coverage settings is a structure which should be provided to the function. It allows the possibility of setting-up the output, coverage and optimisation options for the verification. All aspects have been previously explained in chapter 3. The following list recaps the most relevant aspects regarding the coverage settings.

- The script produces an html report, interactive model view and output data plots. These outputs can be turned either on or off by the user in this part.
- Either all or specific coverage types can be selected in this part by en/disabling the coverage type. The complete list coverage types and their definition has been described in chapter 3.
- Coverage on reference models within the root model can be disabled or filtered. In case the user chooses the filter option, a comma-separated string list with the names of the reference models to exclude has to be provided. The root model can also be excluded from the coverage, if the focus lies on the nested models only.

- The script allows the option to include or exclude verification of external program files called by MATLAB functions and C/C++/S-functions.
- Block reduction optimisation can forced on. When on, the verification will simplify clusters of Simulink elements into a single element in order to speed up/simplify the verification.

The verification process begins with the to-be-verified model or subsystem, which is loaded and opened. A label is created based on the model's file name. A test object is created for the to-be-verified model and named after the created label. The test object is created as a structure with the default options for the verification simulation. The default settings of the created test object is overwritten with the user-defined settings. After this part of the script, the test object structure contains the specified coverage types, optimisation settings and information on nested models and external files to include or exclude from the verification.
Additionally, another structure is created. The parameter structure *paramStruct* is created and is used for the simulation to enable output and state saving. It specifies an absolute tolerance for the simulation and provides the variable name in which the output data is saved in.

The simulation is executed with the verification using the user-defined test object and parameter structure. The simulation gathers coverage for the specified options. It outputs a Simulink Simulation Data Object containing the output data and a data object structure with the coverage data. What happens with the results is depended on the provided coverage settings.

- An html report is generated from the coverage data in the data object. The function *cvhtml* is used to present the report. The report is saved in the current MATLAB directory with a filename based on the verified model's file name.
- An interactive model view is started using the *cvmodelview* function. The Simulink model structure is opened and an information window will appear, which will display the coverage details for every Simulink element selected in the model. Coloured mask will highlight untested elements and elements that have achieved full coverage.
- Plots of all the signal outputs using the Simulation Output Plot Function. The output data in the Simulink simulation data object and the simulation time is used as an input for the function. The input is a matrix with an output signal in each of its columns. It's size is strongly dependent on the simulation, length, step size and the amount of model outputs. The function iteratively plots each model output signal in its own figure and presents it in MATLAB figures.

```
1  function [] = COV_loop_16a(model_name,set)
2  %% In loop verification.
3  % S.Achrifi, TNO, December 2016
4
5  % This script verifies the code coverage for any given MATLAB Simulink
6  % model that contains none, one or more reference models. This script will
7  % generate a code coverage report with coverage data on the root model and
8  % every reference model within it. Inputs are generated for the root model.
9
10 %% Notes.
11 % The model is subjected to certain conditions to be eligible for verification:
12 % -  The Verification and Validation toolbox is required for use of this script!
13 % -  The model should compilable before using this script.
14 % -  The to-be-verified model should be added to the current directory path.
15 % -  In order to collect and plot output data, the model should contain out-blocks
16 % -  The function can cope with 2D and 3D simout structures. Higher
17 %    dimension structures are not applicable.
18 % -  All reference models' simulation mode must be set to 'normal',
19 %    otherwise no coverage can be collected on that reference model.
20 %    Use command: set_param('reference_block_path','SimulationMode','normal');
21
22 % The function is called with the command: COV_loop(model_name,set);
23 % The function requires both input arguments.
24
```

```
%% Model file name.
% The 'model_name' is a variable that should provide the model's file name as a string.
% Example: "model_name123".

%% Model inputs.
% Test cases can not be specified for the nested models. The nested models
% recieve their input from the behaviour of the root model. Inputs for the
% root model should be specified within the Simulink model.

%% Coverage settings.
% The 'set' variable is used to specify the coverage settings used for
% the coverage verification. The settings can determine:

% - Function's output (plots, report, model view)
% - Types of coverage to include
% - Nested models to in/exclude
% - External files or s-functions to in/exclude
% - Simulation optimization

% The coverage settings should be provided in a structure, with each field
% being a setting with a value attached to it. A test object is created
% with the provided settings in the "Creating the test object" part of this
% script.
% An example format can be found in the provided demo scripts to run a
% specific Simulink function.

%% Running the verification.

root_name = strtok(model_name,'/');          % Determines the root model name.
open_system(root_name);                      % Opens the root model in the Simulink window.
open_system(model_name);                     % Opens the subsystem in the Simulink window.
mdl_label = '_Verification Test Object';     % Model label name.
label = strcat(model_name,mdl_label);        % Creates a label for the test object.
test_object = cvtest(model_name,label);      % Creates a test object of the model.

% The settings for the simulation are set below.

test_object.settings.condition               = set.CON;  % Condition coverage
test_object.settings.decision                = set.DEC;  % Decision coverage
test_object.settings.mcdc                    = set.MCDC; % MC/DC coverage
test_object.settings.overflowsaturation      = set.OS;   % Integer overflow coverage
test_object.settings.sigrange                = set.SR;   % Signal ranges coverage
test_object.settings.sigsize                 = set.SS;   % Signal size coverage
test_object.settings.tableExec               = set.LT;   % Look-up table coverage
test_object.settings.designverifier          = set.DV;   % Design verifier coverage
test_object.settings.relationalop            = set.RO;   % Relational boundary coverage

test_object.modelRefSettings.enable          = set.MRS;  % Enable reference model coverage
test_object.modelRefSettings.excludedModels  = set.EM;   % Exclude models from coverage
test_object.modelRefSettings.excludeTopModel = set.ETM;  % Exclude top model from coverage
test_object.emlSettings.enableExternal       = set.EE;   % Enable external files coverage
test_object.sfcnSettings.enableSfcn          = set.ES;   % Enable S-functions coverage
test_object.options.forceBlockReduction      = set.FBR;  % Enable block reduction

set_param(model_name,'ConditionallyExecuteInputs',set.IBO); % Enables/disables input
                                                            % branch optimisation.
set_param(model_name,'SaveFormat','Array'); % Forces the model to output the data in
                                            % an array format.
paramStruct.AbsTol            = '1e-5';     % Defines the simulation absolute tolerance.
paramStruct.SaveState         = 'on';       % Saves the output states.
paramStruct.SaveOutput        = 'on';       % Saves the output data.
paramStruct.OutputSaveName    = 'yout';     % Sets the variable name for the output data.
paramStruct.SaveTime          = 'on';       % Saves the simulation time.
paramStruct.TimeSaveName      = 'tout';     % Sets the name for the simulation time.

[data_object,simout] = cvsim(test_object,paramStruct); % Simulates the created test object.

%% Generating the coverage report.

if set.html == 1;
```

```
95      rep_str = '_coverage_report';                % Part of the report label string.
96      rep_costr = strcat(model_name,rep_str);      % Creates the report label.
97      rep_label = strrep(rep_costr, '/', '_');     % Replaces the '/' characters with the
98                                                   % '_' characters
99                                                   % '/' can't be in document file names.
100     cvhtml(rep_label,data_object);               % Generates the coverage report.
101 end
102
103 if set.cmv == 1;
104     cvmodelview(data_object)      % Generates interactive model view.
105 end
106
107 %% Simulation output data.
108
109 ymat = simout.get('yout');        % Saves the output data structure into a matrix.
110 t = simout.get('tout');           % Returns the simulation time as a vector.
111
112 if set.sop == 1;                  % Executes if simoutplot is enabled.
113     simoutplot(t,ymat);           % Plots the output signals in individual graphs.
114 end
115
116 end
```

Main function: Harness Verification Function

The Harness Verification Function script automates the process of gathering coverage on Simulink reference models or Simulink models that process their in/outputs through in/out-blocks. The script verifies the elements in the model and produces a coverage score based on all the untested elements in the model's structure. The script uses the method of harness verification, in which the to-be-verified model is recreated in a harness model. The harness model features a signal builder with which inputs for test scenarios are simulated. The harness model is simulated and through this, coverage is gathered for the verification.

This method is designed for the singular reference models or models that are depended on in/out-blocks for their in/outputs. The method of creating the harness model, relies on the model containing in/out-blocks. The fixed format of the created harness model makes it ideal for automating. Unlike the in-loop verification method, test scenarios can be loaded in for the verification. This allows specifically designed test scenarios for the to-be-verified model, which has the benefit of verification early in the design process of the model design.
A downside of this method is that specific model test scenarios are required for the verification. Also, the input format in which test scenarios are loaded follows a strict format, compromising the model designer's freedom in loading model inputs. A limitation of this model, opposed to the in-loop verification method, is the difficulty with which it deals with nested (reference) models. Specifically verifying nested models is only possible if the nested model is made 'atomic' and extracted as a new model. This is possible, though it requires additional procedures. It is recommendable to use the in-loop verification method, when dealing with nested models.

The following inputs are required for the verification:

- The to-be-verified model's name in string and, if applicable, the path to the model.
- Settings which specify the coverage types and optimization options to be taken into account for the verification.
- An arbitrary amount of test cases with inputs that resemble test scenarios for the models.

The script accepts only a specific format of inputs. Inputs should be in a matrix format, with each data signal being a column. The first column of the matrix is reserved for the time vector. Any columns after that, are interpreted as data signals. The input is allowed to have an arbitrary amount of data signals, though it should equal the amount of model inputs. Any excess data signals will be removed. In

the situation when there are too few data signals defined, constant-zero signals are added for the model inputs that lack a data signal. The input matrix format is displayed in table 6.2. Multiple test cases can be provided by providing them in a structure format with each field a specific test case containing the input signals.

$$'input' = \begin{pmatrix} time & signal\,1 & signal\,2 & \cdots & signal\,n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \cdots & \vdots \end{pmatrix} \tag{6.1}$$

This script has the following outputs:

- An html report containing the analysis information, coverage score summary and the verification details of the simulation.
- Interactive model view containing colour masks and verification details inside the model.
- Data plots of every data signal of the model connected to an out-block.

The harness verification method its limitations and conditions for use. Below follows a summary of the aspects to take into account when using this method.

- The MATLAB Verification and Validation toolbox is required for the use of this script.
- The harness method gathers coverage data on singular models and reference models.
- The method places the to-be-verified model into a test harness with which in/outputs are directly connected on the to-be-verified model.
- The test scenarios require a strict format to be loaded as input.
- Inputs are applied directly on the to-be-verified models through a signal builder.
- The model must have 'In' input blocks on its highest level, otherwise no inputs can be loaded. (Reference models have in/out blocks by default.)
- The loaded amount of test scenarios must be equal to the number of model inputs. Otherwise the method will either remove input signals if too many or add constant-zero signals if too few. This in order to prevent MATLAB crashes.
- The function can cope with 2D and 3D simout structures. Higher dimension structures are not applicable.

To select a model for verification, one should provide the model's name as it is named in the current folder. The model name should be provided as a string and should represent a reference model or a model that uses in/out blocks for its in/output processing.
Another input that is required for the function, are the test scenarios. The script is able to accept as many test cases as desired, though each test case has to be saved and prvided to the function as a structure with its field names 'inputX', with 'X' being a positive integer. For example, for six different cases, one should name the structure field names as: 'input1','input2'...'input6'. Inputs inside the test cases have to be given in matrix format, with each column a data signal. The first column is should be the time vector. The remaining columns are interpreted as a data signal.
Output, coverage and optimisation options for the verification are defined by provided a structure with the desired setting. All aspects have been previously explained in chapter 3.

- The script produces an html report, interactive model view and output data plots. Incase these outputs are undesired, they can be turned off by the user in this part.
- Either all or specific coverage types can be selected in this part by en/disabling the coverage type.

- Coverage on reference models within the root model can be disabled or filtered. In case the user chooses the filter option, a comma-seperated string list with the names of the reference models to exclude has to be provided. The root model can also be excluded from the coverage, if the focus lies on the nested models only.
- The script allows the option to include or exclude verification of external program files called by MATLAB functions and C/C++/S-functions.
- Block reduction optimisation can forced on. When on, the verification will simplify clusters of Simulink elements into a single element in order to speed up/simplify the verification.

For this verification the warnings for unnecessary data-type conversions are temporarily turned off. The harness model automatically generates size-type conversion blocks between the inputs and the to-be-verified model. This is often unnecessary as the size types are the same most of the time, resulting in consistent false warnings for unnecessary data-type conversions. In case type conversion is required, the blocks perform there function and no warnings are given. This means the warnings are always unwanted. Hence, they are turned off temporarily for this verification. Before the actual verification starts, empty cell arrays for output and data object storage are created. This function will create multiple output sets and data objects within loops, cell arrays are created to iteratively write the outputs and data objects into their respective cells.

The whole simulation is a loop that spans the entire verification process: from creating the harness model and loading test scenarios to simulation of the harness model and processing of the output. The verification process is looped as many times as there are test cases. Each test cases is simulated once. The gathered coverage of each test scenario is summed up in the end.
Inside the loop the to-be-verified model is opened and loaded. A harness model is created that features a signal builder for input loading, size-type conversion blocks, the to-be-verified model in a subsystem and out blocks for retrieving outputs. As soon as the harness model is created, the full directory path towards the harness model and the harness model name are saved as strings in individual variables.
The Test Case to Signal Builder Converter function is used to convert the input data from matrices to a cell array format that is accepted by the signal builder. Additionally, the function script gives a warning in case the amount of data signals in a test scenario does not match the to-be-verified model's amount of inputs. Excess signals are removed and constant-zero signals are added when there are too few defined.

Like with the in-loop verification method, a test object structure with default parameters and options for the verification is created. The default test object is overwritten with the test object's default values with the user-defined parameters and options.
With the test object ready, the function simulates every test case in the harness model and logs the input signals to the test unit block (to-be-verified model) in the harness model. An options structure is created named 'runopts'. The structure enables the option to gather coverage during the simulation and enables referencing to the test object to use those options for verification.
The simulation is run with the options of 'runopts' and the test objects structure. The simulation outputs an array of Simulink.SimulationOutput objects with within signal outputs and data objects containing the gathered coverage.
The created harness model is not of use anymore. To prevent the current MATLAB directory from flooding with unnecessary files, the harness model is deleted after the simulation. However, the harness model is used for simulation, so elements may have changed. It can also be loaded, whilst not visible. The harness model can become shadowed (conflicting filenames) or dirty (unsaved elements) as MATLAB calls it. This can lead to erratic behaviour and errors, like models who appear to be deleted, though still open and unable to close. To prevent errors with deleting these models, the harness model is first saved and then closed.
The gathered coverage from the simulation is stored in a cell array, with each row element of the cell containing a data object.

The Simulink.SimulationOutput object is converted to a matrix array with column-separated signals using the Output to Workspace Converter Function. The function takes the Simulink.SimulationOutput

as input and outputs the simulation time, a cell array with the signal names as strings and the matrix with the output signals. Each verification iteration generates an output signal matrix. These are stored in the cell array 'cdat'. Every row element in the cell contains an output signal matrix.
The data objects stored in the cell array 'cdat' are summated. This is done through a for loop that iterates as many time as there are elements in the cell, minus one. Summating the data objects creates a cumulative data object. Redundant coverage is ignored, whilst only unique coverage is added for the cumulative data object.

This function can provide the same actions with the results as the in-loop verification method. Dependent on the provided coverage settings, the function generates an HTML report with the coverage details, starts the interactive model view to present the coverage results in the model's structure and plots all the output signals in MATLAB figure windows.

```
function [] = COV_harness_16a(model_name,instruc,set)
%% Harness Coverage Verification.
% S.Achrifi, TNO, December 2016.

% This function verifies the code coverage for any given MATLAB Simulink
% model, normal or reference model. This function gathers coverage based on
% multiple sets of input data. This function outputs a verification report,
% plots of the output data and an interactive model view.

%% Notes:
% The model is subjected to certain conditions to be eligible for verification:
% - The Verification and Validation toolbox is required for use of this script!
% - The to-be-verified model should be added to the current directory path.
% - The model should be compilable before using this script.
% - The test input data requires a strict format! (see: "Model inputs")
% - The model should contain in/out-blocks for this script's in and output processing.
% - The function can cope with 2D and 3D simout structures. Higher
%   dimension structures are not applicable.

% The function is called using the command: COV_harness(model_name,instruc,set)
% The function requires all three input arguments.

%% Model file name.
% The 'model_name' is a variable that should provide the model's file name as a string.
% Example: "model_name123".

%% Model inputs.
% 'instruc' is the variable that contains the inputs required for
% gathering coverage. The inputs should represent test scenarios similar
% to the expected operating conditions of the system.

% The model inputs should be defined in a structure, with each structure
% field being a test case. Each field name should be named as "inputX",
% with X being an integer from 1, counting up, to an arbitrary value.

% instruc.input1 = ...
% instruc.input2 = ...
% instruc.input3 = ...
%      ...       = ...
%      ...       = ...

% Each test case may contain multiple input signals. A test case should be
% defined as a matrix, with an input signal in each column. The first
% column of the matrix should be reserved for the time vector. The
% remaining columns can take an arbitrary amount of signals from signal 1,
% counting up, to signal n.

%  time    signal 1   signal 2 ... signal n
%    .        .          .           .
%    .        .          .           .
%    .        .          .           .

%% Coverage settings.
```

```
% The 'set' variable is used to specify the coverage settings used for
% the coverage verification. The settings can determine:

% - Function's output (plots, report, model view)
% - Types of coverage to include
% - Nested models to in/exclude
% - External files or s-functions to in/exclude
% - Simulation optimization

% The coverage settings should be provided in a structure, with each field
% being a setting with a value attached to it. A test object is created
% with the provided settings in the "Creating the test object" part of this
% script.
% An example format can be found in the provided demo scripts to run a
% specific Simulink function.

%% Creating the harness model.

invars = length(fieldnames(instruc));
warnid = 'Simulink:blocks:DTConversionUncessary';   % Warning ID: Unnecessary Data Type
                                                    % Conversion.
warning('off',warnid);          % Turns off warnings for unnecessary data type
% conversions.
cmat = cell(length(invars),1);  % Creates an empty cell array for the output data.
cdat = cell(length(invars),1);  % Creates an empty cell array for data object storage

for i = 1:invars;    % Executes the verification as many times as there are inputs.

    root_name = strtok(model_name,'/'); % Determines the root model name.
    open_system(root_name);             % Opens the root model in the Simulink window.
    open_system(model_name);            % Opens the subsystem in the Simulink window.

    % Creates and retrieves the file path of the harness model.
    harness_model_file_path = slvnvmakeharness(model_name);

    % Retrieves the name of the harness model.
    [~,harness_model_name] = fileparts(harness_model_file_path);

    %% Loading test scenarios.

    % Calls the script to load the input into the signal builder. Two sets
    % of inputs are loaded in.

    [test_sc_time, test_sc_data] = input2sigbuilder(harness_model_name,...
    model_name,instruc.(['input' int2str(i)]));

    %% Creating the test object.

    mdl_label = '_Verification Test Object';
    label = strcat(model_name,mdl_label);            % Creates a label for the test object.
    test_object = cvtest(model_name,label);          % Creates a test object of the model.

    % The settings for the simulation are set below.

    test_object.settings.condition               = set.CON;  % Condition coverage
    test_object.settings.decision                = set.DEC;  % Decision coverage
    test_object.settings.mcdc                    = set.MCDC; % MC/DC coverage
    test_object.settings.overflowsaturation      = set.OS;   % Integer overflow coverage
    test_object.settings.sigrange                = set.SR;   % Signal ranges coverage
    test_object.settings.sigsize                 = set.SS;   % Signal size coverage
    test_object.settings.tableExec               = set.LT;   % Look-up table coverage
    test_object.settings.designverifier          = set.DV;   % Design verifier coverage

    test_object.modelRefSettings.enable          = set.MRS;  % Enable reference models
    test_object.modelRefSettings.excludedModels  = set.EM;   % Exclude models from coverage
    test_object.modelRefSettings.excludeTopModel = set.ETM;  % Exclude top model
    test_object.emlSettings.enableExternal       = set.EE;   % Enable external files
    test_object.options.forceBlockReduction      = set.FBR;  % Enable block reduction

    %% Logging signals and simulating the verification.
```

```
124
125     logged_signals_harness = slvnvlogsignals(harness_model_name); % Logs the signals of the
126                                                                   % harness model.
127
128     runopts = slvnvruntestopts;            % Creates an options structure for simulation.
129     runopts.coverageEnabled = true;        % Enables coverage.
130     runopts.coverageSetting = test_object; % Simulates with the test object settings.
131
132     % Executes the simulation.
133     [simout,data_object] = slvnvruntest(model_name,logged_signals_harness,runopts);
134
135     % Saves the harness model in order to close it.
136     save_system([harness_model_name '.slx']);
137     % Closes the harness model in order to delete it.
138     close_system(harness_model_name);
139     % Removes the created harness model after the simulation.
140     % Prevents the current folder from flooding with models.
141     delete([harness_model_name '.slx']);
142
143     cdat{i,1} = data_object;
144
145     %% Simulation output data.
146
147     % Converts the simulation output structure into a matrix format.
148     [tsim,ymat,outlabels] = output2workspace(simout);
149
150     cmat{i,1} = ymat;  % Creates a cell array with output data matrices for each input.
151
152 end
153
154 %% Generating the coverage report.
155
156 data_object = cdat{1};                     % Creates a cell array with the first element
157                                            % being cdat{1}
158 for c = 1:length(cdat)-1;                  % Executes for the amount of data objects - 1.
159     data_object = data_object + cdat{c+1}; % Sums up all the data objects for each input.
160 end
161
162 if set.html == 1;                          % Executes if coverage report is enabled.
163 rep_str = '_coverage_report';              % Part of the report label string.
164 rep_costr = strcat(model_name,rep_str);    % Creates the report label.
165 rep_label = strrep(rep_costr, '/', '-');   % Replaces the '/' characters with the
166                                            % '_' characters.
167                                            % '/' can't be in document file names.
168 cvhtml(rep_label,data_object);             % Generates the coverage report.
169 end
170
171 %% Interactive model view
172
173 if set.cmv == 1;                    % Executes if interactive model view is enabled.
174     cvmodelview(data_object)        % Generates interactive model view.
175 end
176
177 %% Output data plots
178
179 if set.sop == 1;                    % Executes if simoutplotter is enabled
180 simoutplot(tsim,cmat,outlabels);    % Plots the output signals in individual graphs.
181 end
182
183 warning('on',warnid);  % Turns back on warnings for unnecessary data type conversions.
184
185 end
```

Main function: Compare Models Function

The Compare Models Function compares similar reference models of different development generations to check whether there are significant differences between them. Basically, reference models can be

compared with each other if they contain the same amount of input and outputs and if they simulate under the very same model settings and parameters, like sampling frequency. The basic goal of the function is to compare an arbitrary amount of reference models with each other and highlight the output signals that are significantly different than that of the default model. This function should help in the model-based development of functional systems. Different stages of models can quickly be compared with each other to check the differences and verify the development in system behaviour.

This function uses the same method of simulation as the Harness Verification Method. Due to the harness structure, a lot of similarities will be found between the two functions. Because of the same structure, this section will only describe the goal, inputs and outputs of the function.

The inputs that this function requires are almost similar to the Harness Verification Method. The added features are that multiple models can be provided for simulation and threshold factors that specify the significant differences. A description of the required inputs is listed below.

- A cell vector containing all the model names as strings that are to be simulated.
- A structure containing the test cases which act as inputs for the models to be tested to.
- A structure containing simulation settings like threshold factors, plot and store settings.

The cell vector should contain all the model names as strings of the models that are desired to be compared to each other. The models are compared to a default model, which is the first model specified in the provided cell vector. The simulation loops for every model and every test case.
The structure format that is required to provide the inputs is similar to format described for the Harness Verification Method. The structure should contain fields for each test case, named "inputX", with X being a positive increasing integer. Each test case should be a matrix with each input signal specified in a column. The first column is reserved for the time vector, the rest of the column may be taken for input signals.
The settings structure contains values for settings and parameters required for the simulation. The upper and lower limit threshold factors should be a vector with as many factors as there are output signals. They are described in the next paragraph.
Two more entries in the settings structure are the plot and write to Excel setting. The plot setting determines the plot action for the results of the simulation. A value of '2' will cause the function to only plot output signals of models that produce significantly different signals than that of the default model. A value of '1' will ensure the function plots all the output signals of all models, regardless if they are significantly different or not. A '0' value will not plot anything.

The upper and lower limit threshold factors are used to determine the boundary where a signal can be classified as significantly different. Each threshold value is multiplied with its respective output signal of the first model. This creates a new boundary signal for each of the first model's output signal. The boundary signals are compared with the output signals of the other provided models. If any output signal of another model is higher than the upper limit times the first model or lower than the lower limit times the first model, than that signal is deemed significantly different. The function compares all output signals of the remaining models this way. A logic matrix is the result, in which true and false values are stored upon the comparison. A true indicates significant difference, whilst a false indicates no difference. The logic matrix is used for the generation of plots. The plot loop sequence is initiated for every true entry in the logic matrix.

An additional feature of this function is the ability to store the results in an Excel file. The results of the simulation are the logic matrix that indicates which output signals of which models are significantly different and the plotted figures of the output signals. An Excel file is created using ActiveX. The file is stored in the created folder "Results" on the current folder directory. The logic matrix and images of the output signals are written on the first sheet of the file. In case the "Results" folder already exists, the function will remove the folder and its contents to replace it with a fresh folder and the contents of the current simulation. This is done in order to prevent conflicted copies.

```
function [] = COMP_refMdl(mdl_names,instruc,set)
%% Reference models Comparison

% This script compares the simulation outputs of an arbitrary amount of
% SIMILAR reference models (see notes). Multiple test cases are used for
% performance comparison between the provided models. The function checks
% whether the models produce significantly different ouputs than the first
% model.

%% Notes:
% The models are subjected to certain conditions to be eligible to compare.
% - The models must be set to fixed step.
% - Both models must have the same sampling frequency.
% - The models must contain in/out-blocks for in/output processing.
% - Both models must contain the same amount of output signals (out blocks)
% - The models should be added to current directory path.
% - The Excel results file can not be overwritten when it is open.
% - The function can cope with 2D simout structures. Higher
%   dimension structures have not been tested.

% This function is used with the command: COMP_refMdl('mdl_names,instruc,set);
% This function requires all three input arguments.

%% Model names
% The variable mdl_names should be a cell array vector with the strings of
% the to-be-compared models.

% ['model_1' , 'model_2' , 'model_3' , ... , 'model_n'];

%% Model inputs
% 'instruc' is the variable that contains the inputs required for
% the comparison. The inputs should represent test scenarios similar
% to the expected operating conditions of the system.

% The model inputs should be defined in a structure, with each structure
% field being a test case. Each field name should be named as "inputX",
% with X being an integer from 1, counting up, to an arbitrary value.

% instruc.input1 = ...
% instruc.input2 = ...
% instruc.input3 = ...
%      ...       = ...
%      ...       = ...

% Each test case may contain multiple input signals. A test case should be
% defined as a matrix, with an input signal in each column. The first
% column of the matrix should be reserved for the time vector. The
% remaining columns can take an arbitrary amount of signals from signal 1,
% counting up, to signal n.

%  time    signal 1   signal 2 ... signal n
%    .        .          .           .
%    .        .          .           .
%    .        .          .           .

%% Comparison settings
% The 'set' variable should be a structure with fields for values of
% settings required for the comparison. Information should be provided on
% the upper and lower treshold factor, the plot settings and the Excel file
% setting.
% Output signals are plotted if one of the models' output signal is
% larger/smaller than the first model times a threshold factor. An Excel
% file, containing the plots and a logic table, is saved in the 'Results' folder
% The upper and lower threshold factors should be provided in a row vector,
% with as many elements as there are output signals. The first element in
% the vector corresponds to signal 1, the second element to signal 2,
% etc...

% set.up_thres  = [1.2; 1.2; 1.2];           % Upper trigger treshold factor
```

```
% set.low_thres = [0.8; 0.8; 0.8];           % Lower trigger treshold factor

% set.pset = 2;    % Configure plot settings. Input can be either 0,1 or 2.
                   % - 0: no plots are generated.
                   % - 1: all plots are generated.
                   % - 2: only plots that significantly differ are generated.

% set.exf = 1;     % Enable generation of excel result file in the Results folder.
                   % - 0: disable Excel file.
                   % - 1: enable Excel file.

%% Creates the harness model

warnid = 'Simulink:blocks:DTConversionUncessary';    % Warning ID: Unnecessary Data
                                                     % Type Conversion.
warning('off',warnid);                               % Turns off warnings for unnecessary
                                                     % data type conversions.
lna = length(mdl_names);                             % Returns the amount of models.
lin = length(fieldnames(instruc));                   % Returns the amount of inputs.
cmat = cell(lna,lin);                                % Creates an empty cell for
                                                     % output storage.

for z = 1:lin;                                       % Loops for the amount of inputs.
    i = 1;                                           % Initial condition for the
                                                     % iteration counter.
    for n = mdl_names;                               % Loops for the amount of models.

        % Converts the cell string to a single character array.
        model_name = cell2mat(n);
        % Creates and retrieves the file path of the harness model.
        harness_model_file_path = slvnvmakeharness(model_name);
        % Retrieves the name of the harness model.
        [~,harness_model_name] = fileparts(harness_model_file_path);

        %% Loads the input into the models.

        % Loads the inputs into the signal builder and returns the format.
        [test_sc_time, test_sc_data] = input2sigbuilder(harness_model_name,...
        model_name,instruc.(['input' int2str(z)]));

        %% Simulates the models.

        test_object = cvtest(model_name);   % Creates a test object of the model.

        % Logs the signals of the harness model.
        logged_signals_harness = slvnvlogsignals(harness_model_name);

        % Executes the simulation.
        [simout,~] = slvnvruntest(model_name,logged_signals_harness);

        % Saves the harness model in order to close it.
        save_system([harness_model_name '.slx']);
        % Closes the harness model in order to delete it.
        close_system(harness_model_name);
        % Removes the created harness model after the simulation.
        % Prevents the current folder from flooding with models.
        delete([harness_model_name '.slx']);

        %% Retrieves the Output data.

        % Converts the simulation output structure into a matrix format
        [tsim,ymat,outnames] = output2workspace(simout);

        cmat{i,z} = ymat;          % Writes the output to a new variable
        i = i+1;                   % Loop iteration counter.

    end
end

%% Plots signals with significant difference.
```

```
140
141 % Plots the output signals in individual graphs.
142 tab_flag = differplot(tsim,set.up_thres,set.low_thres,outnames,cmat,set.pset);
143
144 %% Outplot table.
145
146 [~,os] = size(cmat{1});                 % Retrieves the amount of output signals.
147 stuff_cell = cell(os,1);                % Creates an empty cell for the stuff cell.
148
149 for j = 1:os;                           % Creates a stuffarray as long as the number
150                                         % of outputs.
151 stuff_cell{j,1} = '-';                  % Fills '-' in the cell for every iteration.
152 end
153
154 ltable = array2table(stuff_cell);       % Inserts a stuff cell in the first column of the
155                                         % table.
156 l = 1;                                  % Initializes the iteration counter.
157 [~,sig] = size(cmat{1,1});              % Returns the amount of signals.
158 signame  = cell(sig,1);                 % Creates an empty cell for the signal names.
159
160 for in = 1:lin;                         % Loops for the amount of inputs.
161                                         % Converts and adds the differplot
162                                         % output to the table and changes the names of the
163                                         % stuff cell columns.
164
165     ltable = [ltable array2table(tab_flag{in,1}')];
166     ltable.Properties.VariableNames{'stuff_cell'} = ['Input_' int2str(in)];
167
168     for h = 1:lna                       % Loops for the amount of models.
169                                         % Changes the column names from
170                                         % 'Var' to 'Model'.
171
172         ltable.Properties.VariableNames{['Var' int2str(h)]} =...
173          ['In_' int2str(in) '_Model_' int2str(h)];
174         % l = l+1;
175     end
176
177     for u = 1:sig;                      % Loops for the amount of signals.
178                                         % Creates a cell vector with
179                                         % the signal names as strings.
180
181     signame{u,1} = ['Signal_' int2str(u)];
182     end
183
184     if  in < lin            % Adds stuff cells after each set of model names.
185         ltable = [ltable array2table(stuff_cell)];
186     end
187 end
188
189 ltable.Properties.RowNames = signame;   % Adds the signal names as the row names.
190 sigltable = [ltable signame];
191 [~,sltc] = size(sigltable);
192 sigltable.Properties.VariableNames{['Var' int2str(sltc)]} = 'Signals';
193
194 %% Writing to Excel
195
196 if set.exf == 1;    % Executes only if the Excel setting is enabled.
197
198 writetable(sigltable,'Compare_results.xlsx')        % Displays the logic output table.
199
200  % Returns all the files in the folder Results in a structure.
201 s = dir('Results');
202
203 % Converts the field names of the structure to a cell array.
204 fignames = {s.name}';
205
206 % Determines which files are not figures.
207 [rows,~] = find(~strncmp('figure',fignames,6));
208
209 for b = 1:length(rows)
```

```
210     fignames(rows(1),:) = []; % Removes the rows of files that are not figures iteratively.
211     [rows,~] = find(~strncmp('figure',fignames,6));
212 end
213
214 % Moves the results excel file to the folder Results
215 movefile('Compare_results.xlsx',[pwd '\Results']);
216
217 % Saves the generated figures into the result excel file.
218 img2xlsx(fignames,'Compare_results.xlsx',[pwd '\Results'],1,[0,1,1,180,450,352]);
219
220 end
221
222 warning('on',warnid);  % Turns back on warnings for unnecessary data type conversions.
223
224 end
```

Sub-function: Test Case to Signal Builder Format Converter

The Harness Coverage Verification Method and the Compare Models Function allows the loading of test cases as inputs for the simulation. For both cases a harness model is created that is used for the simulation. The input is provided through a signal builder block. The signal builder only loads test cases with a strict format. Test cases should be provided in two separate cell arrays. One time column vector and a series of data row vectors. This sub-function has been created to convert the user-defined test cases into a format the signal builder block in the harness model accepts.

The function requires three inputs to work:

- the harness model's name as a string
- the to-be-verified model's name as a string
- the test cases input as a matrix

The function requires the harness model's name and the to-be-verified model's name in order the select and load the models. The to-be-verified model is used to determine the amount of input blocks on the main level of the model. This is verified with the amount of provided input signals. In case the provided signals do not match the amount of input blocks, excess input signals are removed if there are too many or constant-zero signals are added in case there are too few.
The function converts a singular test case into the required cell format. As mentioned before, the test case should be provided in a matrix format with a column for each test signal. The first column being the time vector and the remaining columns the data signals. Because the test case can take up any amount of rows and columns, a for loop is used to write each column over into an element of the cell array iteratively.

Test cases are added as a new signal group in the signal builder. At first a list is retrieved with the existing group names in the respective signal builder block. If there exist a group with the name "Test Case", that group is removed. Afterwards a new "Test Case" group is appended to the signal builder. This in order to prevent confliction between equal named groups. The group names are retrieved again to set the "Test Case" group as the active group.

The main goal of this function is to load the test case into the signal builder of the respective signal builder. Additionally, this function provides the converted time and data signals in the cell format and stores it in the workspace.

```
1 function [time, data] = input2sigbuilder(harness_model_name,model_name,input)
2
3 %% Test case to signal builder format converter.
4
5 % This script converts the test input data from the workspace variable
6 % 'input' into a cell format that is loaded into the Simulink
```

```
7  % signal builder block of the harness model.
8
9  % The input data of 'input' must be a matrix with column vectors as
10 % signals. The first column at index 1 must be the time signal. The
11 % remaining columns must be input signals.
12
13 %  time    signal 1   signal 2 ... signal n
14 %    .        .          .           .
15 %    .        .          .           .
16 %    .        .          .           .
17
18 % The script's inputs are:
19 % - Name of the harness model,                  [harness_model_name]
20 % - Name of the to-be-verified model,           [model_name]
21 % - Test input matrix,                          [input]
22
23 % The script's outputs are:
24 % - Time data in signal builder format,         [test_sc_time]
25 % - Signal data in signal builder format,       [test_sc_data]
26
27 % S.Achrifi, TNO, September 2016.
28
29 %% Loading the system.
30
31 open_system(harness_model_name);                % Opens the root model
32 blk_path = strcat(harness_model_name,'/Inputs'); % Generates the path to the signal builder
33 open_system(blk_path);                          % Opens the signal builder block's mask
34
35 %% Converting the input data.
36
37 [~,columns] = size(input);         % Determines the amount of columns of the input matrix
38
39 test_sc_time = (input(:,1))';      % Isolates the time vector from the input data
40 test_sc_data = cell(columns-1,1);  % Creates an empty cell vector
41
42 for i = 1:columns-1;
43 test_sc_data(i,1) = {input(:,i+1)'}; % Fills each element of the cell vector
44 end                                  % with the input data vectors
45
46 %% Resizing input data to the amount of model inputs.
47
48 [~,Mdlinfo] = sldiagnostics(model_name,'Sizes'); % Gathers information about the model.
49 Numin = Mdlinfo.NumInputs;                       % Determines the number of inputs in the
50                                                  % model's main level.
51
52 if Numin < columns-1;                            % Checks wether there are too much
53                                                  % input signals.
54
55     test_sc_data = test_sc_data(1:Numin,1);      % Scales the input signals down to
56                                                  % the amount of inputs.
57
58     warning('There are more input signals than model inputs.
59     The excess input signals have been dropped!');
60
61 elseif Numin > columns-1;                   % Checks whether there are too few
62                                             % input signals.
63
64     expansion = cell(Numin-(columns-1),1);  % Creates a cell to compensate for the
65                                             % missing input signals.
66     fill = zeros(1,length(input));          % Creates a constant zero vector at the
67                                             % length of the input signals.
68
69     for z = 1:Numin-(columns-1);            % Fills the expansion cell with constant
70                                             % zero signals.
71        expansion{z,1} = fill;
72     end
73
74     test_sc_data = [test_sc_data; expansion];  % Adds the constant zero signals to the
75                                                % input data cell.
76
```

```
77      warning('There are less input signals than model inputs.
78      Constant-zero signals have been added to the input data!');
79  end
80
81  %% Loading inputs into the signal builder
82
83  sigind = 1:i;                                          % Determines the amount of signals
84                                                         % in a group.
85  [~,~,~,groupnms] = signalbuilder(blk_path);            % Retrieves the groupnames in the
86                                                         % signalbuilder in row vector.
87  idflag = strmatch('Test Case',groupnms,'exact');       % Finds the exact string 'Test Case'
88                                                         % and retrieves the column its in.
89
90  if idflag > 0;
91      signalbuilder(blk_path,'SET',sigind,'Test Case',[],[]);  % Removes the group
92                                                               % 'Test Case', if it exists.
93  end
94
95  % Loads the input data into the signal builder block.
96  signalbuilder(blk_path,'append',test_sc_time,test_sc_data,{},'Test Case');
97
98  [~,~,~,groupnms] = signalbuilder(blk_path);      % Retrieves the groupnames once again.
99  index = strmatch('Test Case',groupnms,'exact'); % Determines the index of the 'Test Case'
100                                                 % group.
101 signalbuilder(blk_path,'activegroup',index);     % Sets the group 'Test Case' as the active
102                                                 % group.
```

Sub-function: Output Data to Workspace Converter

Performing a verification simulation with each of the three main scripts will result in output data. Simulink provides the output data in a Simulink Simulation Data Structure. These structures depend on the amount and length of the output signals, thus, a process is required in retrieving the right amount of data. The process is made generic and placed in this function. The function requires only one input, which is the Simulation Data Structure.

The function provides the following outputs:

- the output signals as result of the simulation
- the simulation time as a vector
- the label names of the output signals as strings

The Simulation Data Structure is accessed for three of its contents: the output signals, the simulation time and the label strings of the output signals. It is determined whether the output data is 2-dimensional or 2-dimensional. This function won't be able to deal with higher dimensional output structures. In the case of higher dimensional structures, it is advised to revert back to either 2D or 3D structures. The data is extracted from the data structure and stored in separate variables which are retrieved by the function and stored into the workspace.

```
1   function [tsim,ymat,outlabel] = output2workspace(simout)
2
3   %% Output data to workspace converter
4
5   % This script converts the Simulink simulation output data object from the
6   % verification into a matrix format with each column vector a different
7   % output signal.
8
9   % signal 1   signal 2   signal 3 ... signal n
10  %    .          .          .           .
11  %    .          .          .           .
12  %    .          .          .           .
```

```
13
14 % The script's inputs are:
15 % - Simulation time,                                [tsim]
16 % - Data matrix with the simulation outputs,       [ymat]
17 % - cell array with signal names as strings,       [outlabel]
18
19 % The script's outputs are:
20 % - Simulink simulation output data object,        [simout]
21
22 % S.Achrifi, TNO, October 2016.
23
24 %% Accessing the SimulationOutput structure.
25
26 tsim = simout(2).get('tout_slvnvruntest');   % Saves the time vector in a variable.
27 yout = simout(2).get('yout_slvnvruntest');   % Accesses the output data structure.
28 ycell = {yout.signals.values};               % Creates a cell format for the data.
29 outlabel = {yout.signals.blockName};         % Saves the out block label names in a cell.
30 [~,columns] = size(ycell);                   % Determines the amount of cell columns.
31
32 if length(size(ycell{1,1})) == 2;
33     [rows,~] = size(ycell{1,1});             % Determines the amount of rows for a
34                                              % 2-dimensional output.
35 elseif length(size(ycell{1,1})) == 3;
36     [~,~,rows] = size(ycell{1,1});           % Determines the amount of rows for a
37                                              % 3-dimensional output.
38 else
39     warning('Dimensions of the output signal makes no sense.
40     Either it is lower than 2, or higher than 3');
41 end
42
43 ymat = zeros(rows,columns);                  % Creates a zero matrix 'ymat'.
44
45 for i = 1:columns;
46     ymat(:,i) = ycell{:,i}(1,:);             % Writes each element in the cell into
47                                              % a column of the data matrix.
48 end
49
50 end
```

Sub-function: Simulation Output Plot Function

Each of the three main functions produces results in the form of output data. The Output Data to Workspace Converter accesses the data in the Simulink Output Data Structure and stores it in variables in the MATLAB workspace. This function, the Simulation Output Plot Function, uses the converted results of the simulation and plots them in individual MATLAB graphics.

The function requires three inputs to plot the output signals.

- the simulation time as a column vector
- the converted simulation output data in either a matrix or cell array format with each element a matrix
- a cell row vector with the strings of all the output signal label names

The inputs are all provided by the Output Data to Workspace Converter in the correct format. Using the converted variables from the workspace will result in each output signals being plotted. The function distinguishes whether the provided output data is a matrix or cell array format with each element a matrix. This feature is required as the in-loop verification method stores its output data in a singular matrix format. The harness verification method and the compare function allow the possibility of multiple test cases. Hence output data is stored in a cell array with each element the output data of that particular test case.

In case the provided output data is in a matrix format, a simple loop is executed that adds each output signal (columns) to a MATLAB figure window individually. The loop ends when all signals have been added. If the provided output data is in a cell array format, first a loop goes over each element in the cell, checking whether all elements have equal sized matrices. All matrices are compared to the size of the matrix in the first element. When one element does not match up, an error is given highlighting that the outputs differ in width and/or length. This is possible due to different sized test cases, when using multiple models, different sample frequencies or different amount of outputs. When the sizes are correctly a multiple loop sequences ensures that all output signals of each cell element are plotted in their respective MATLAB graphics window.

Due to signals being added to existing plot windows and legend entries, a warning often pops up informing that extra legend entries are ignored. Due to the iterative added, the error is unavoidable and false, since at the end of the loop all the legend entries are added. It is for this reason the warnings for ignoring extra legend entries have been temporarily disabled during this function.

```
1
2  function [ ] = simoutplot(t,mat,outlabels)
3
4  %% Simulation output plotter
5
6  % This function script plots the output signals from the verification
7  % simulation. The input is either one or two 'ymat' data matrices from the
8  % output2workspace function script. The result are plots for each output
9  % signal in the 'ymat' data matrix. If two ymat matrices are provided, the
10 % similar variables are plotted in the same plot for comparison reasons.
11
12 % The script's inputs are:
13 % - Simulation time,                              [t]
14 % - Output data in either an array or cell,      [mat]
15 % - Cell array with the signal names,            [outlabels]
16
17 % The script's outputs are:
18 % - MATLAB plots of each output signal,
19
20 % S.Achrifi, TNO, October 2016.
21
22 %% Cmat/Ymat decisioning.
23
24 warnid = 'MATLAB:legend:IgnoringExtraEntries';  % Defines the ID for ignoring
25                                                 % extra legend entries warnings.
26 warning('off', warnid);                         % Disables warnings for ignoring
27                                                 % extra legend entries.
28
29 ic = iscell(mat);   % Checks whether the input is the cmat cell or ymat matrix.
30
31 %% Array ymat plotting.
32
33 if ic == 0;          % Executes if the input is the ymat matrix.
34
35     [~,c] = size(mat);        % Returns the amount of output signals.
36     for i = 1:c;              % Operates a loop for each column.
37
38         % Generates a plot for each output signal.
39         figure(i)
40         plot(t,mat(:,i),'Color',[0 0 0]); title(['Output signal #' int2str(i)]);
41         ylabel(['Signal: ' int2str(i)]); xlabel('Simulation time [s]'); grid on
42     end
43 end
44
45 %% Cell Array cmat plotting.
46
47 if ic == 1;         % Executes if the input is the cmat cell array.
48
49     [r,~] = size(mat);                      % Returns the amount of inputs.
50     [~,c] = size(mat{1,1});                 % Returns the amount of signals.
```

```
51
52      for i = 1:r;
53          if size(mat{1,1}) == size(mat{i,1});      % Checks whether sizes of all matrices
54                                                    % in the cell element are equal.
55          else
56              error('Simulation unsuccessful! Output data differs in width and/or length.
57                     Please check whether the models have the same sampling frequency
58                     and amount of outputs.');
59          end
60      end
61
62      legendinfo = cell(r,1);                       % Creates an empty cell for the legend
63                                                    % information.
64
65      for n = 1:r
66          legendinfo{n,1} = ['Test Case' int2str(n)];  % Writes the legend information for
67                                                       % each model.
68      end
69
70      q = 1;              % Loop iteration counter
71
72      for x = 1:c;
73          for z = 1:r;         % Generates a plot for each output signal of every input.
74              figure(q)
75              plot(t,mat{z,1}(:,x));
76              title(['Output signal #' int2str(x) ' for all test cases']);
77              xlabel('Simulation time [s]'); ylabel(['Signal: ' outlabels{1,x}]);
78              legend(legendinfo);
79              grid on; hold all;
80          end
81          q = q+1;       % Iterative figure counter
82      end
83  end
84
85  warning('on', warnid);      % Re-enables warnings for ignoring extra legend entries.
86
87  end
```

Sub-function: Significant Difference Plot Function

The Significant Difference Plot Function is a sub-function for the Compare Models functions. The sub-functions contains the algorithm to determine whether the output signals of the different models are significantly different and the algorithm to plot the output signals.

The function uses the following inputs to distinguish significant differences and plot the output signals that do differ.

- the time vector that determines the duration of the simulation
- the threshold factors for the upper limit for each output signal
- the threshold factors for the lower limit for each output signal
- the output signal labels as strings in a cell vector
- a cell array with the output signals stored for each test case in each element
- a pset variable which contains the setting for the desired plot action.

The algorithm determines the significant differences based on the threshold factors provided. Threshold factors should be given for both the upper and lower limit. Both are given as a vector with as many elements as there are output signals. The elements in the vector correspond with the output signals in that the first element applies to the first output signal, the second element on the second output signal, etc...
The upper and lower threshold factors are multiplied with each of the first model's respective output

signal, creating an upper and lower bound signal. These bound signals represent the default behaviour. Any model that produces a higher value in time than the first model times the upper threshold factors or a lower value in time than the first model times the upper threshold factor, is marked as a significant difference.

The simulation time and the output signal labels are used for the generation of plots. The pset value determines the plotting procedure. A provided value of '2' ensures plots are only shown if a model has produced a significantly different output signal. A value of '1' plots all the output signals of all models, regardless of whether they are significantly different or not. A '0' value will prevent anything from being plotted.
Aside of the plots being presented in the MATLAB windows, they are also individually saved in a "Results" folder created on the current folder path. Upon repeat, this function checks whether the "Results" folder exist. If so, it removes it to recreate it again with the figures of the new simulation. This is done in order to prevent conflicting copies.

Similar to the Simulation Output Plot Function, this function checks whether the sizes of the output signals match for each test case. It does so by comparing every test case in each element with the data size of the test case in the first element. If all sizes are the same, the function continues. If not, an error will be displayed mentioning the different width and/or length of the output signals.
The function also checks whether the amount of threshold factors match the amount of output signals. After that, the function will start on determining whether there are output signals of models that significantly differ of the first default model. A matrix is produced with logic values. For each measured signal that crosses the bound signals, a logic value of '1' is written in the logic matrix. If there is no measured significant difference, a '0' value is written in the matrix.

If the plot all setting is selected, the function will plot every output signal of every model in a similar way as the Simulation Output Plot Function. In case it is opted to only plot the output signals that significantly differ, the logic matrix will be used. For every true value in the logic matrix, a loop sequence is executed that adds all the model's results for that particular output signal, the bound signals and their specific legend entries and labels to the plots.

```
1  function [tab_flag] = differplot(t,up_thres,low_thres,outlabels,cmat,pset)
2
3  %% Significant difference plotter.
4
5  % This script plots the output signal of all models if one of the output
6  % signal is significantly larger/smaller than the output of the first
7  % model.
8
9  % If one model's output is larger than model 1 times the upper treshold
10 % factor, that signal will be plotted for all models.
11
12 % If one model's output is lower than model 1 times the lower treshold
13 % factor, that signal will also be plotted for all models.
14
15 % The script's inputs are:
16 % - Simulation time,                                    [t]
17 % - upper threshold factor,                            [up_tres]
18 % - Lower threshold factor,                            [low_tres]
19 % - Cell array with signal names,                      [outlabels]
20 % - Cell array with the model's simulation outputs,    [cmat]
21 % - Plot setting,                                      [pset]
22
23 % The script's outputs are:
24 % - Cell array with a logic table for every input,     [tab_flag]
25
26 %% Output size-checker.
27
28 warnid2 = 'MATLAB:legend:IgnoringExtraEntries';      % Disables the warnings for extra
29                                                      % legend entries.
30 warning('off',warnid2);                              % Normally unavoidable using this
```

```
                                                    % legend plot method.

[r,~] = size(cmat);          % Returns the rows of cmat (amount of models).
[~,in] = size(cmat);         % Returns the columns of cmat (amount of inputs).
[~,c] = size(cmat{1,1});     % Returns the # columns of the cmat (amount of output signals).
s_check = size(cmat{1,1});   % Size of model 1. Used to size-check the other models.
q = 1;                       % Initial value for the plot counter.

legendinfo = cell(r,1);              % Creates an empty cell to store the legend labels.
legendinfo{1,1} = 'Upper limit';     % Legend entry for the upper limit label.
legendinfo{2,1} = 'Lower limit';     % Legend entry for the lower limit label.
tab_flag = cell(in,1);               % Creates an empty cell for logic matrix.

if exist('Results','dir') > 0;       % Checks whether the 'Results' folder exist.
    rmdir('Results','s');            % Removes the 'Results' folder and its contents.
end                                  % This is done to prevent conflictions with
                                     % existing files.
mkdir('Results');                    % Creates the 'Results' folder.

for a = 1:in;

    % This loops checks whether all model outputs have the same size as the
    % first model. If not, it returns an error.
    for i = 1:r;
        if any(size(cmat{i,1}) ~= s_check);
            error('Simulation unsuccessful! Output data differs in width and/or lenght.
                   Please check whether the models have the same sampling frequency
                   and amount of outputs.');
        end
    end

    % This loop checks whether the amount of threshold factors equal the
    % amount of output signals.
    if length(up_thres) == c && length(low_thres) == c;
    else
        error('The amount of threshold factors do not equal the amount of outputs!');
    end

    %% Upper/lower-limit significant difference check.

    state = zeros(r,c);      % Creates an empty matrix to store the flags of
                             % significant differences.

    % This loop checks whether there are significant differences in all model's
    % signals compared to the first model's signals.
    for z = 1:r;
        for i = 1:c;
            state(z,i) =  any(abs(cmat{z,a}(:,i)) > abs(cmat{1,a}(:,i))*up_thres(i))...
                || any(abs(cmat{z,a}(:,i)) < abs(cmat{1,a}(:,i))*low_thres(i));
        end
    end

    tab_flag{a,1} = state;       % Creates a logic matrix with flags for signal differences.

    %% Positive flag signal plotting.

    if pset == 2;        % Only executes this part when plot setting is set to 2.

        for j = 1:r;
            legendinfo{j+2,1} = ['Model ' int2str(j)];    % Writes model labels into
                                                          % the cell every iteration.
        end

        [~,col] = find(state);  % Returns the columns which have a significant difference.
        col = unique(col);      % Removes all duplicate columns.

        for i = col';           % Loops for the positive flag signals.

            % Plots the upper and lower limits of the positive flag signals for model 1.
            figure(q);  plot(t,cmat{1,a}(:,i)*up_thres(i),'k--',t,cmat{1,a}(:,i)*...
```

```
            low_thres(i),'k--'); hold all;

            for z = 1:r;        % Loops for the amount of models.

                % Iteratively plots the positive flag signals from the columns in 'col'
                plot(t,cmat{z,a}(:,i));
                title(['Output signal ' int2str(i) ' based on input ' int2str(a)]);
                ylabel(['Signal: ' outlabels{1,i}]); xlabel('Simulation time [s]');
                legend(legendinfo);
                grid on;
            end

            % Saves the current figure in a bmp format.
            saveas(gcf,['figure' int2str(q)],'bmp')

            % Moves the figures to the new folder
            movefile(['figure' int2str(q) '.bmp'],[pwd '\Results']);

            q = q+1;    % Loop iteration counter.
        end
    end
end

%% Plotting all signals.

if pset == 1;            % Only executes this part when plot setting is set to 1.

    for a = 1:in         % Loops for the amount of inputs
        for j = 1:r;     % Loops for the amount of models
            legendinfo{j+2,1} = ['Model ' int2str(j)];    % Writes model labels into
                                                          % the cell every iteration.
        end
        for i = 1:c;
            % Plots the upper and lower limits of the positive flag signals for model 1.
            figure(q);  plot(t,cmat{1,a}(:,i)*up_thres(i),'k--',t,cmat{1,a}(:,i)*...
            low_thres(i),'k--'); hold all;

            for z = 1:r;        % Loops for the amount of models.

                % Iteratively plots the positive flag signals from the columns in 'col'
                plot(t,cmat{z,a}(:,i));
                title(['Output signal ' int2str(i) ' based on input ' int2str(a)]);
                ylabel(['Signal: ' outlabels{1,i}]); xlabel('Simulation time [s]');
                legend(legendinfo);
                grid on;
            end
            % Saves the current figure in a bmp format.
            saveas(gcf,['figure' int2str(q)],'bmp')

            % Moves the figures to the new folder
            movefile(['figure' int2str(q) '.bmp'],[pwd '\Results']);

            q = q+1;    % Loop iteration counter.
        end
    end

%% Doesn't plot anything

if pset == 0;       % This part only executes if plot setting is set to 0.
    warning('Plots have been turned off. Enable the setting to generate plots.');
end

warning('on',warnid2);      % Enables the warnings for extra legend entries.

end
```

Differences between the MATLAB 2014a and 2016a versions

Mathworks monitors their software platform MATLAB and tool boxes including Simulink regularly. They are quick to note and fix bugs and glitches within the software, but are also as frequent in producing new features to make their software platform extensive and practical to use. Updates are released in new versions of MATLAB and Simulink multiple times a year (2014a, 2014b, 2015a, 2015b, etc...). However, the frequent updating of software also has the disadvantage of backwards-incompatibility. Models, stateflows, functions and other code designed in older versions are automatically converted to be compatible in the new MATLAB versions. However, software designed in newer versions are very difficult to maintain operational when converted to the older versions.

The Integrated Vehicle Safety (IVS) department of TNO practices model-based design for the development of automotive safety systems. They do so using MATLAB version 2014a. However, every so often, the models are converted to a newer version. Shortly after this project, a start will be made with introducing MATLAB 2016a as the standard. The coverage functions have been created and tested whilst Keeping a careful eye on the differences between MATLAB version 2014a and 2016a. The lack of the new features and principles can hamper the software functionality. The same also applies to the created coverage functions, however, not as significant.

Speaking in functional terms, there is a slight difference. The 2016a version of MATLAB introduces two new coverage types to gather coverage on. The coverage types are Relational Boundary coverage and Simulink Design Verifier coverage. Relational Boundary coverage focusses on blocks like the Abs, Saturation and any logic operator by gathering coverage on their behaviour when dealing with values that lie in the tolerance range of trigger values. Simulink Design Verifier coverage introduces coverage on the blocks of the Simulink Design Verifier toolbox. If the desire is to gain coverage on these types, version 2016a is required.
Additionally, there are slight difference in the default settings for the blocks. For example, when enabling Saturation on Integer overflow, the 2016a version gathers this coverage on simple math blocks per default. The 2014a version does not and has to be enabled in the block's parameter if the desire is to allow the coverage.

An important thing to note that there has been experienced some difficulties in converting 2014a reference models to version 2016a and applying the Harness Verification Method. In this situation the verification errors on the creation of the harness model, with the argument "can not access method in class". A direct solution to this problem has not been found, though it is expected the problem lies in the model settings/parameters of the 2014a model. When copying the whole contents of the 2014a model and pasting it in a fresh 2016a model window, with default model settings/parameters, the Harness Verification Methods works correctly. It is therefor expected an entry in the model settings/parameters are in some way prohibiting the creation of a harness model of 2014a to 2016a converted models.