Universität Bremen

Investigation and Automation of Verification Methods for Model-Based Development of GNC Systems

Master's Thesis

submitted at

University Bremen

Faculty 4: Production Engineering

within the study program

Space Engineering M. Sc.

Author: Philipp Kayser

6254563

Examiners: Dr.-Ing. Stephan Theil

Leonardo Borges Farconi

Bremen, May 15, 2025

Abstract

Objective: The presented work considers the development of guidance, navigation and control (GNC) software in the context of reusable launch vehicles. More specifically, the considered workflows involve modeling in MATLAB/Simulink and automated code generation with the goal of deployment to real-time embedded system processors. The objective is to investigate, evaluate and automate the use of several compatible software verification tools at different points of this process.

Methods: An example project is developed to resemble the software that is developed in a project context. The considered tools are introduced and investigated in that context for their usability and limitations. Both static and dynamic verification are considered, taking current formal methods into account. Methods for static analysis of MATLAB code, Simulink models and generated code are presented as well as an alternative method of testing in Simulink.

Results: The work results in a review of the tools capabilities and limitations. A basic framework of wrapper functions to use the tools is developed in the process. This encompasses an automation concept for the use in GitLab with an automated evaluation of verification results. Lastly, an initial qualitative evaluation of the tools is provided.

Conclusion: The presented suite of tools is able to significantly increase the level of confidence in software quality, when it is used correctly. This means that they require to be used in conjunction within a well-defined process and respecting their individual limitations. Concluding, recommendations for next steps and a prioritization for implementation in a project context are given.

Contents

List of Figures

List of Tables

List of Symbols

$\mathbf{x}$	state vector
$\mathbf{u}$	input vector
$\mathbf{y}$	output vector
$\mathbf{A}$	state matrix
$\mathbf{B}$	input matrix
$\mathbf{C}$	output matrix
$\mathbf{D}$	direct transmission matrix
$G(s)$	transfer function
$U(s)$	Laplace transform of the input vector
$Y(s)$	Laplace transform of the output vector
K_P	proportional gain
K_I	integral gain
K_D	derivative gain
$r(s)$	reference input signal
$e(s)$	error signal
$u(s)$	controller output signal
$v(s)$	actuator output signal
$y_m(s)$	measured plant output signal
$y(s)$	sensor output signal
$\mathbb{Z}$	set of all integers
S	set of integers
$\alpha_{\pm}$	abstraction function
$\gamma_{\pm}$	concretization function
$\perp_{\pm}$	bottom element
$\top_{\pm}$	top element
Q	state space
q	initial state

Σ	input alphabet
h	transition relation
F	set of final states
AP	set of atomic propositions
a, b	atomic propositions
r_m	mixture ratio
$\dot{m}_o$	oxidizer mass flow
$\dot{m}_f$	fuel mass flow
$\dot{m}_t$	total mass flow
P_c	chamber pressure
V_c	chamber volume
T_c	combustion temperature
R	specific gas constant
ρ_c	combustion density
A_t	throat area
c^*	characteristic velocity
x_v	valve opening
τ_v	valve response time
u_o	oxidizer control command
u_f	fuel control command

Acronyms

API	application programming interface
AUTOSAR	Automotive Open System Architecture
CALLISTO	Cooperative Action Leading to Launcher Innovation in Stage Toss-back Operations
CD	continuous delivery
CERT	Computer Emergency Response Team
CI	continuous integration
CTL	computation tree logic
CWE	Common Weakness Enumeration
DLR	German Aerospace Center
DOM	Document Object Model
EN	European Standard
ERT	embedded real-time
FSM	finite state machine
GCC	Guidance and Control Computer
GNC	guidance, navigation and control
GPT	generative pre-trained transformer
GRT	generic real-time
GUI	graphical user interface
HIL	hardware-in-the-loop
HIS	Hersteller Initiative Software
HISM	High Integrity Systems Modeling
HNS	Hybrid Navigation System
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
ISTQB	International Software Testing Qualifications Board

JMAAB	Japan MathWorks Automotive Advisory Board
JSF AV	Joint Strike Fighter Air Vehicle
JSON	JavaScript Object Notation
LLM	large language model
LTI	linear, time-invariant
LTL	linear temporal logic
MAAB	MathWorks Automotive Advisory Board
MAB	MathWorks Advisory Board
MIL	model-in-the-loop
MISRA	Motor Industry Software Reliability Association
MPCV	Multi-Purpose Crew Vehicle
NASA	National Aeronautics and Space Administration
NESC	NASA Engineering & Safety Center
NIST	National Institute of Standards and Technology
OASIS	Organization for the Advancement of Structured Information Standards
OBC	On-Board Computer
PIL	processor-in-the-loop
ReFEx	Reusability Flight Experiment
RTCA	Radio Technical Commission for Aeronautics
SARIF	Static Analysis Results Interchange Format
SEI	Software Engineering Institute
SIL	software-in-the-loop
VCS	version control system
XML	Extensible Markup Language

1. Introduction

On the morning of June 4th, 1996 the maiden flight of the newly developed Ariane 5 launch vehicle ended in catastrophic failure. An independent inquiry determined that an unexpectedly high value of an internal alignment function caused an operand error due to an unprotected data type conversion in the internal software of the inertial reference system. The on-board computer interpreted the resulting diagnostic data output wrongly as flight data and commanded full engine nozzle deflections, leading to an abnormally high angle of attack and the subsequent self-destruction of the vehicle after 39 seconds of flight [1].

The investigation revealed that the missing protection in the software as well as the subsequent behavior of the inertial reference system did not violate the specification. Further, system testing was conducted insufficiently based on false assumptions. According to the report, these decisions were made out of a general philosophy that software can be considered correct until it is shown to be at fault. In fact, the authors point out that software should instead be assumed to be faulty until the currently accepted best practices demonstrate that it is not [1].

Since the 1990s, the development of control systems has changed considerably. Development philosophies and life cycles have emerged that are more concurrent in nature and embrace early and consistent verification [2], [3]. Development and verification are more strongly driven by more diverse and capable tools and their automation capabilities. Advances in computer science have brought forth more rigorous verification techniques, that have been successfully adopted and scaled to industrial applications [4], [5].

The MathWorks Inc. is tightly associated with this process. Being founded in 1984, the company has grown to marketing and supporting more than 130 individual products associated with the MATLAB/Simulink development ecosystem [6]. Spacecraft flight software development by the Institute of Space Systems at the German Aerospace Center (DLR) is just one of many example applications of this software suite.

The thesis at hand aims to investigate the use of several MathWorks software verification products in this context. As preliminaries, fundamentals of GNC systems and the context for flight software development are introduced. The term *verification* for the scope of this thesis is defined in line with its meaning in software development. The associated technical fundamentals are therefore introduced too. This thesis follows an emphasis on static analysis and formal verification, but aspects of conventional software testing are considered too. Based on this, the individual tools are described with a focus on how they might be useful in the given context. In the process, a basic usage and automatization framework around those tools that integrates with the existing workflows at the institute is developed. Concluding, an evaluation of the tools capabilities and recommendations for future use are given.

2. Fundamentals

2.1. Guidance, Navigation and Control Modeling

On a spacecraft, the objective of the GNC subsystem is to achieve the desired motion of the vehicle. This task is divided functionally into *guidance*, referring to determining the required trajectory of the spacecraft; *navigation*, referring to determining the spacecraft position, velocity and attitude; and *control*, referring to determining the commands needed to achieve the required trajectory [7]. Some fundamentals in the area of control system engineering are required to introduce these principles.

Any dynamic systems is characterized by its behavior that changes the system's condition over time. This behavior is described by a system of differential equations that can oftentimes be derived from the physical laws governing the system. As laid out in reference [8], these differential equations can be converted into the *state space representation*. By defining state variables, a higher-order differential equation can be reformulated as a system of first-order differential equations. Arranging these in a compact and standardized form leads to the state space representation where

- The state vector $\mathbf{x}$ contains the state variables of the system, which represent the system's condition at a given time
- The input vector $\mathbf{u}$ contains the system inputs, which represent the external influences on the system
- The output vector $\mathbf{y}$ contains the system outputs, which represent observable quantities derived from the state
- The state equation defines how the system's internal state changes over time

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{2.1}$$

- The output equation defines how the internal state and inputs influence the system's output

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}, \mathbf{u}, t) \tag{2.2}$$

The state space is the n-dimensional space whose coordinate axes represent the state variables [8].

A dynamic system exhibits *linear* behavior when it can be described by linear differential equations: The dependent variable and its derivatives have to appear only in first-degree. Following [8], a linear system satisfies two principles:

- superposition: the response to a sum of inputs equals the sum of the responses to each input individually
- homogeneity: If the input to a system is scaled by a constant factor, the output must be scaled by the same factor

Linear systems are generally easier to analyze, control and simulate. If a dynamic system displays nonlinear behavior, it is usually linearized by approximating it around a useful operating point. There are specialized analysis methods for nonlinear methods which however fall well beyond the scope of this thesis. For the linear case, the state space representation is

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\,\mathbf{x}(t) + \mathbf{B}(t)\,\mathbf{u}(t) \tag{2.3}$$

$$\mathbf{y}(t) = \mathbf{C}(t)\,\mathbf{x}(t) + \mathbf{D}(t)\,\mathbf{u}(t) \tag{2.4}$$

where $\mathbf{A}(t)$ is called the state matrix, $\mathbf{B}(t)$ the input matrix, $\mathbf{C}(t)$ the output matrix, and $\mathbf{D}(t)$ the direct transmission matrix [8].

In the general form of the state space representation above, the system parameters may change: vector function $\mathbf{g}$ in equation 2.1 and vector function $\mathbf{g}$ in equation 2.2 are both a function of time t. If they are not, the system is called a *time-invariant system* and the state space representation can be simplified to

$$\dot{\mathbf{x}}(t) = \mathbf{A}\,\mathbf{x}(t) + \mathbf{B}\,\mathbf{u}(t) \tag{2.5}$$

$$\mathbf{y}(t) = \mathbf{C}\,\mathbf{x}(t) + \mathbf{D}\,\mathbf{u}(t) \tag{2.6}$$

A different system description is the *transfer function* $G(s)$. It is defined as the ratio of the Laplace transform of the system's output $Y(s)$ to the Laplace transform of the system's input $U(s)$, under the assumption that all initial conditions involved are zero, with the complex frequency $s = \delta + j\omega$ [9].

$$G(s) = \frac{Y(s)}{U(s)} \tag{2.7}$$

Performing the Laplace transformation and thereby transferring the system from the *time domain* into the *frequency domain* allows to express system dynamics by algebraic equations instead of differential equations, which is highly useful in system analysis and control design. As [8] notes, the transfer function includes the units necessary to relate the input to the output, but does not provide any information concerning the physical structure of the system. As such, a transfer function can be established experimentally by introducing known inputs and studying the outputs. In other cases, it is possible to derive it from the physical laws governing the system, as is demonstrated in chapter 4.

The transfer function methodology is only applicable to linear, time-invariant (LTI) systems [8]. For such systems, it however allows to describe complex system dynamics with the use of a limited number of elemental transfer-function forms. Reference [9] names

- the proportional element P with $G(s) = K_P$
- the integral element I with $G(s) = \frac{K_I}{s}$
- the derivative element D with $G(s) = K_D\ s$
- the proportional–derivative element PD_1 with $G(s) = K_{PD_1}(T\ s + 1)$
- the first-order low-pass element PT_1 with $G(s) = \frac{K_{PT_1}}{Ts+1}$
- the second-order oscillatory element PT_2 with $G(s) = \frac{K_{PT_2}}{(T^2\ s^2+2\ D\ T\ s+1)}$.

System and associated control dynamics can be visualized easily in the frequency domain, with transfer functions represented as blocks in block diagrams. The layout of a basic control loop is shown in figure 2.1, which has been adapted from reference [9], where

- $r(s)$ is the reference input
- $e(s)$ is the error signal
- $u(s)$ is the controller output

- $v(s)$ is the actuator output
- $y_m(s)$ is the measured output from the plant
- $y(s)$ is the feedback signal from the sensor.

The objectives of spacecraft GNC systems are realized with the same underlying structure, yet have a much higher complexity. The related implications for this thesis are addressed in chapter 3 and revisited in chapter 4.

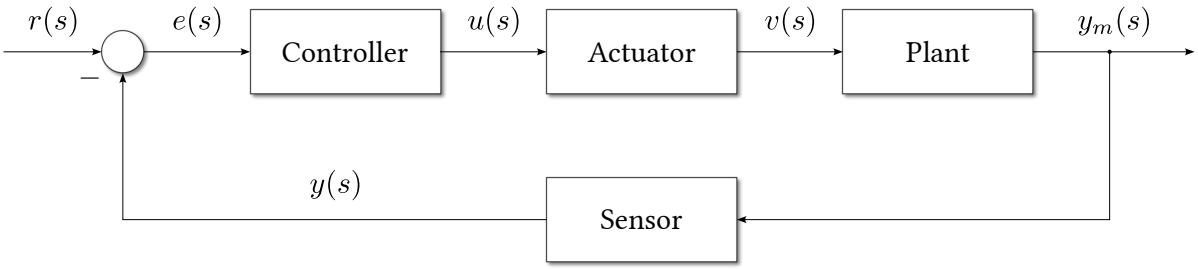

Figure 2.1.: Basic Control Loop (modified from [9])

MATLAB/Simulink is the state-of-the-art tool for dynamic system simulation and control design in a variety of application domains. It allows to hierarchically model system dynamics and the associated data manipulation to very high levels of complexity and serves as the basis for the work done within the scope of this thesis.

The conventional way of modeling with Simulink is using the extensive range of blocks that allow to model virtually any mathematical operation on data represented as signal lines. In complex applications such as spacecraft GNC it can be beneficial to perform considerable parts of the required computation in MATLAB functions embedded in Simulink models. This background is important to note, as the model verification should take this prioritization into account.

An important aspect of modeling with Simulink is the model configuration. The configuration parameters of a model form a configuration set and govern how a model is run. Among others, they contain settings regarding how the simulation is executed, how data is imported and exported, and what the target hardware for simulation and code generation is [10]. They are just as important as the correctness of the model itself and thus should also be subject to verification.

MATLAB fully supports object-oriented programming. *Objects* here are structures that contain data as *properties* and functions that operate on that data as *methods*. They are instances of a *class*. When using MATLAB/Simulink and its toolboxes programmatically, they usually have to be interacted with in the same way.

2.2. Software Management

To work effectively on a shared set of files, software developers typically use version control systems (VCS). VCS provide a systematic way of documenting and managing how files change over time. A VCS stores backups of files and shows who made what changes to what file at which point in time. This allows for different developers to safely make changes on a code base without inadvertently losing work, or to review and understand changes made by others [3].

2.2.1. Version Control

At the time of writing, Git is the most widely used VCS. As opposed to other VCS, it uses a distributed architecture, meaning that when working with Git, all team members have all project-associated files and metadata on their local machine instead of a central server [3]. In the following, the related basic concepts relevant for this thesis are introduced.

Any set of files that is put under version control by Git is called a *repository*. A repository can be stored locally or remotely. Usually, repositories are used for managing source code in software projects, but version control can be used on any set of files [3].

To work safely on a set of files, developers *branch* their changes into individual lines of development. This way, work can be done in parallel while keeping the original line of development free from unwanted changes. When completing work on a branch, developers have to *merge* their changes back into the original branch. This process is governed by the *branching strategy* in place. One common example is to work on specific `feature` branches, that are branched and merged from a `development` branch, which is regularly merged with a more stable `main` or `master` branch [3].

Adding changes to a repository happens stepwise. First is *staging* the changes, second is *committing* them to the repository. The staging area functions as a middle step between

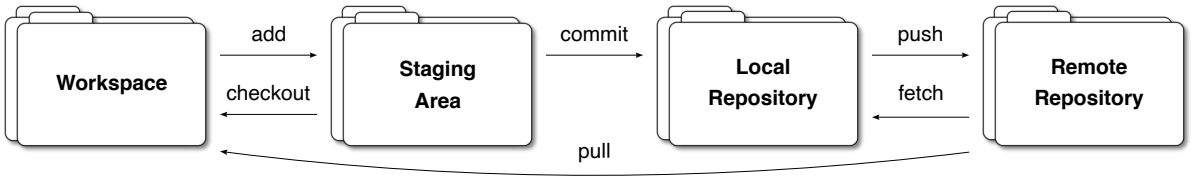

Figure 2.2.: Basic Git Concepts

working directory and repository. It allows for having control over how changes are grouped into commits. With a commit, changes are recorded to the repository history. A commit is the captured state of a set of files at one point in time, and a branch is a reference to the latest in a series of commits [3].

Finally, when working with a remote repository, committed changes need to be *pushed* in order for all other team members to have access to them. Fetching and pulling inversely is used to update a local repository to the state of a remote [3]. The basic concepts are visualized in figure 2.2.

2.2.2. Repository Management

There are several providers offering platforms for remote repositories, with GitLab being the one used within the context of this thesis. It builds upon the functionality of Git with collaboration tools such as merge requests and issue tracking as well as automation with pipelines. Its core principles are introduced in the following.

In GitLab, users are organized in *groups*. Groups allow for managing settings as well as reviewing development activity across several projects. A *project* is a container for a Git repository in GitLab. Within a project, *issues* are atomic units that organize the work to be done. They can be used to track tasks, report bugs, request features and more. They provide a space for discussion among team members and organize responsibility by assignment. GitLab also introduces *merge requests* as an additional component available when merging changes from one branch into another. They display the gathered edits from all commits in the source branch, making the resulting differences in the target branch transparent. Reviewers are assigned and suggestions discussed within the context of a merge request. Upon completion of review, changes are approved before they are merged into the target branch. When creating a merge request early, it can function as a space that monitors code quality:

Results of tests and scans on the committed changes can be displayed to indicate the impact of merging [3].

2.2.3. Continuous Integration

By today's standards, collaborative software development is done incrementally, where the product is developed and verified in small functional units, and iteratively, where the product is improved continuously in repeating cycles. Agile software development is a set of principles formalizing these in comparison to traditional development more "lightweight" practices. GitLab is geared towards agile development. Much power of GitLab comes from its so-called *CI/CD pipelines* and the associated fundamentals are introduced in the following [3].

Continuous integration (CI) refers to the act of continuously implementing and verifying changes to a code base. Its purpose is to ensure that changes integrate well with the existing code base and arising problems are spotted early on. A typical scenario for this would be to execute a standardized set of tests, checks or scans on the files affected by the committed changes [3].

Subsequently, *continuous delivery (CD)* refers to automatically and periodically transferring code to the right environment. In conventional software development, an *environment* describes how a machine storing and executing software is configured and what tools are available to it. Depending on the project, there might be environments for development, testing, integration, production and more [3].

A *pipeline* is a series of actions performed automatically on files in a GitLab project. It is usually triggered whenever changes are committed to the repository and can be configured to manipulate any files inside the project. A GitLab project can only have one pipeline, which can however be configured to perform different kinds of actions on different parts of the code base [3].

Pipelines consist of *stages*. They are used to group tasks that are related to each other, so a conventional configuration would include a build, a test and a deploy stage. The tasks performed within a stage are called *jobs*. Just as a user would pass several commands to a computer in order to perform a task, jobs contain predefined commands that are executed according to the pipeline configuration. If not specified otherwise, jobs inside a stage are executed in parallel, while each stage is executed sequentially [3].

The processes executing the pipeline commands are called *runners*. As jobs typically require a specific environment, they are not run by the main GitLab applications. Runners need to be registered with the GitLab instance using the similarly named GitLab Runner application that has to be installed on the machine that is intended to run the process. A runner then receives the commands from the GitLab instance based on the pipeline configuration in the GitLab project, executes them and reports the results back to the GitLab instance [11].

Every job is executed in its own working directory on the machine that executes the runner. Once a runner receives a job, it creates the associated working directory and fetches the commit that triggered the pipeline, so the relevant files are available in the correct version required for the job. The commands are then executed inside the working directory and artifacts are stored according to the pipeline configuration. This way, multiple jobs in the same stage per default have the same basis available to them. The outcomes are isolated, reproducible without conflict and comparable among each other [3].

The *executor* is the environment in which a runner executes a job. It is specified upon registration. It is possible to register several runners on one machine and assign different executors to them. The simplest form is the shell executor, which runs jobs in a shell session of the machine it is running on. The job's commands are interpreted the same as commands typed in a command line terminal by a user [11]. GitLab offers several further executor types better suited for project scalability, these are however out of the scope of this thesis.

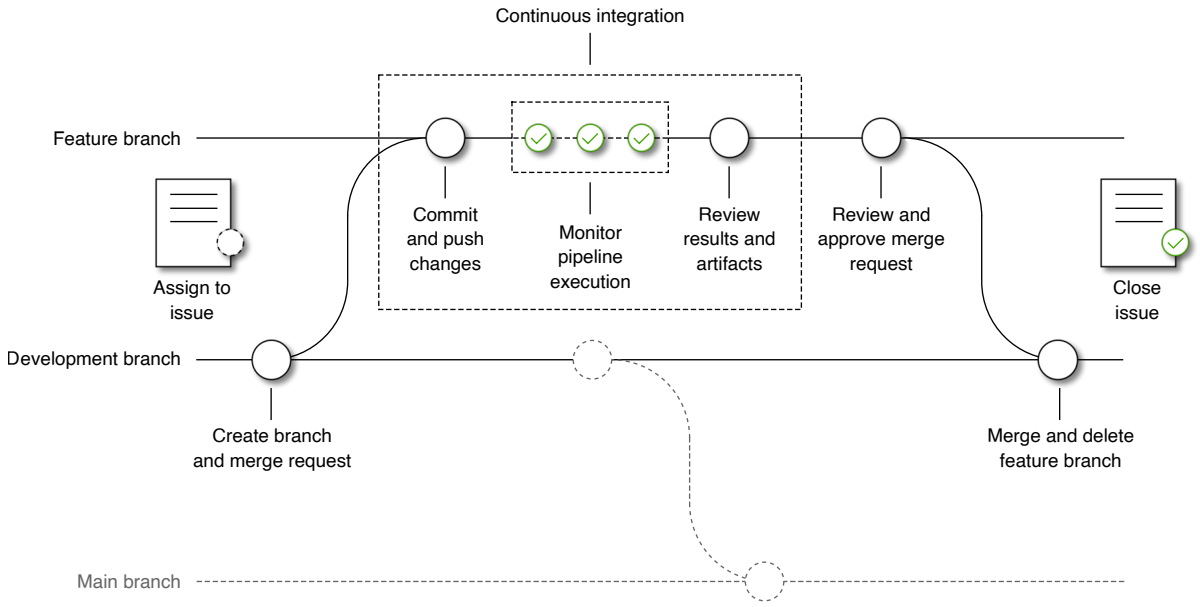

Figure 2.3.: Software Management

Figure 2.3 presents an attempt to visualize how the introduced concepts relate to each other. Integration happens *continuously* and verification, as opposed to more traditional workflows, is an iterative aspect of development.

2.3. Software Testing

In software development, verification is intended to evaluate the *correctness* of a development artifact by comparison with a reference artifact considered as complete and correct [12]. Alongside validation, which evaluates whether the system is fit and effective for its intended purpose, it is a central activity in any major software project.

There are both static and dynamic verification methods. *Static verification* intends to discover software faults directly. Typically this is done either by code review or by using specialized tools and referred to as *static analysis*. *Dynamic verification* intends to uncover failures in a piece of software by executing it under controlled conditions and is mostly referred to as *software testing* [2].

2.3.1. Terminology

In the context of software testing, certain terminology is commonly used but not in all cases uniformly defined. For the thesis at hand, the foundation syllabus and glossary defined by the International Software Testing Qualifications Board (ISTQB) are used [2].

The tested piece of software is referred to as *test object* and a discrepancy between expected and shown behavior is referred to as *defect* or *failure*. Failures in turn are the result of *faults* or *bugs* introduced in the software. Software does not fail in the way that hardware does: generally speaking, faults are introduced in a system by errors or mistakes made during development. They are present in a system from the beginning until their removal [2].

The *test basis* serves as the definition of the expected behavior of the the test object. It usually consists of a specification and related documentation, but the expertise and experience of the tester can also be counted towards it. A component or subsystem as test object usually needs to be broken down into separate testable items. These *test items* can for example be functions or methods of the test object [2].

A *test case* is a description of how a test object is tested in terms of prerequisites, required input, necessary procedure and expected output. A test case usually has one elemental objective, which typically is the verification against one requirement. *Test suites* are sets of test cases [2].

Test Levels

Defects are identified best at the level of abstraction on which they occur. For this reason, different testing levels have been established. Due to their varying scopes, they might require different techniques and tools.

Component tests verify the low level building blocks of a system architecture in isolation from the rest of the system. A component can itself have lower level building blocks, however the interaction with other system components is not investigated. Component tests typically verify the complete and correct implementation of functionality defined in the test object's specification by checking input/output behavior. The ISTQB syllabus considers *module*, *unit* or *class tests* as types of component tests [2].

Integration testing aims at finding faults in the interfaces and the interaction between multiple tested components. As test basis now software architecture, system design and especially interface specifications have to be considered. The types of failures addressed by integration testing relate to for example data consistency, dependency issues or error propagation. Ideally, integration testing is done incrementally and adhering to an integration strategy. This could be top-down or bottom-up in the system architecture or by the availability of the individual components [2].

System testing is intended to check that the complete, integrated system fulfills its requirements. Many functions and system attributes result from the interaction of the system's components and can only be verified once the system is fully integrated. At this stage, the system is verified from the customer's point of view and incompletely or unsuitably implemented requirements revealed. If requirements are vague or missing, system testing can show where clarification is required [2].

Result Cases

Test results are categorized depending on whether a reported deviation actually constitutes a defect. With a true positive or true negative result, the test case correctly identified a deviation or its absence. False positive or negative results imply that the test case falsely interprets intended behavior as undesirable or vice versa. This relation is shown in table 2.1.

Table 2.1.: Result Cases in Software Verification

Result evaluation	Test object correct	Test object faulty
"Failed"	False negative	True positive
"Passed"	True negative	False positive

2.3.2. Testing Techniques

The techniques applied in dynamic software testing can roughly be broken down in black-box and white-box techniques. Choosing the appropriate technique depends on the situation [2].

Black-Box Testing

In black-box testing, the inner attributes of the test object are unknown. Therefore, the following test techniques focus on observing output behavior with only inputs and preconditions as variables in test case creation.

In *equivalence partition testing*, equivalence partitions or classes group inputs together that are expected to produce the same output of a test object. Testing one member of one equivalence class is assumed to be representative in output behavior for the entire class. The entire input value range is divided into these classes, also accounting for invalid inputs. Representatives of the created classes are then selected and tested. Analyzing the test object's behavior at boundary values is important to reveal ambiguities in the specification [2].

Other testing techniques like *decision table testing* or *pair-wise testing* are helpful when individual combinations of inputs cause different output behavior of the test object. It is the goal

of these techniques to test every possible combination of inputs in a structured way, which is most useful when the inputs signify logical conditions [2].

White-Box Testing

The goal of white-box testing techniques is the successful execution of every part of the test object's code. In contrast to black-box testing, it is required that the internal composition of the test object is available for test design [2].

In *statement testing*, the statements in a test object's code are sought to be executed. A statement is any single operation or instruction inside the tested code, and with every statement executed without defect the test object is assumed to function as intended. The ratio of executed to total statements is called *statement coverage* [2].

In *decision testing*, the decisions following conditional statements in the test object's code are evaluated. Following such a statement, the control flow of the test object splits in multiple outcomes. Testing then attempts to execute each outcome at least once. *Decision coverage* is the ratio of executed to total decision outcomes. The technique is more comprehensive than statement testing but also usually requires more test cases. Full decision coverage guarantees full statement coverage, but not the other way around [2].

Condition testing addresses decisions that are made based on multiple conditions. In branch condition testing the outcomes of these atomic conditions are evaluated individually. In branch condition combination testing, the goal is that the combinations of these logical conditions are tested. As this can become quite comprehensive, modified condition decision testing addresses only those combinations of conditions that change the result of that decision [2].

For illustration, consider the following example.

```
1 function compute(x, y, z):
2     result = 0
3
4     if (x > 0) and (y > 0) then
5         result = x
6     else
7         result = y
8     end if
```

```
9
10      if (x > 2) or (z < 0) then
11          result = result + 1
12      end if
13
14      return result
15  end function
```

To reach full statement coverage of this function, two test cases are required. Test 1 with e.g. `x=3, y=3, z=3` would evaluate both `if`-statements to true. Test 2 with `x=-1, y=2, z=-5` would additionally cover the missing `else`-case. The test suite with tests 1 and 2 would however not reach full decision coverage, which is revealed by considering the control flow graph of the test object, which is depicted in figure 2.4. Adding test 3 with e.g. `x=1, y=1, z=5` to the test suite would cover the missing false-evaluation of the second `if`-statement. To achieve full condition coverage, yet another test case is needed, as so far not every atomic condition has been evaluated to both true and false. Test 4 with e.g. `x=3, y=0, z=10` would evaluate the missing condition `(y > 0)` and allow the test suite to achieve full condition coverage.

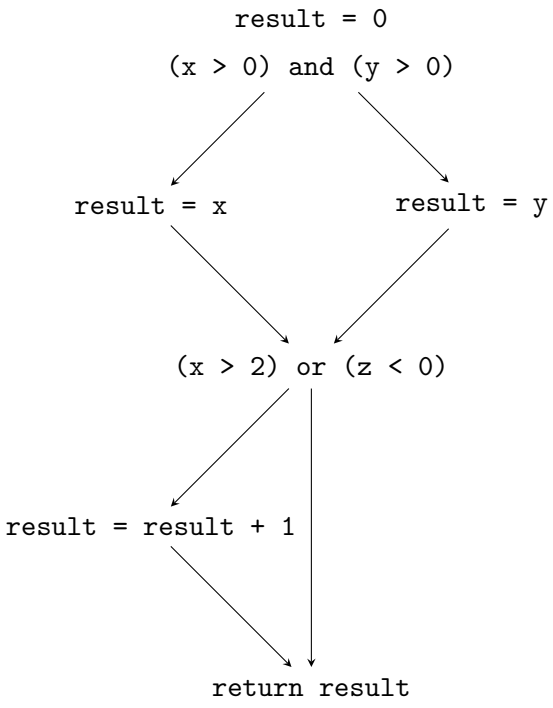

Figure 2.4.: Control Flow Graph for the Example

2.4. Static Analysis

Independent of the testing technique used, dynamic testing is in a realistic scenario not exhaustive. Even with extensive testing efforts, at least some faults in any system must be assumed to remain unrevealed. The goal of static verification is to discover faults in software directly without executing it. Conventionally this is done by experienced programmers reviewing the code in question. Software tools are increasingly capable of performing static analysis with varying scopes and objectives [2].

2.4.1. Terminology

Static analysis tools derive information from the analyzed code similarly to code compilers [13]. Code compilers translate a source program into instructions readable by the machine that needs to execute them. Depending on the scope of the analysis, parts of this process are also executed by a static analysis tool, which is why it is necessary to understand the background [14].

Code compilation is divided in at least two phases, the front-end and the back-end. In the *front-end phase*, a compiler performs three types of analyses [14].

- Lexical analysis identifies the individual units the characters of a program constitute.
- Syntax analysis determines whether this provided sequence is a permissible statement according to the rules of the programming language.
- Semantic analysis determines the meaning of the syntactically correct statement.

In the *back-end phase*, the program synthesis is performed. This is typically done by translating the information provided from the front-end into an intermediate language, which is the basis for code optimization. The intermediate program is subsequently translated into machine code [14].

The front-end is sensitive to the programming language, the back-end to the processor architecture. When compilers perform optimization independent of processor architectures, this is done in an additional phase referred to as *middle-end*.

Static analysis tools make use of the same techniques as a compiler front-end does. The gathered information is however not processed into an executable output but directly presented

to the user. Existing tools vary in their scope from mere syntactic to comprehensive semantic analysis.

Not every programming language is compiled before execution. Such *interpreted* languages are executed command-by-command without an explicit compile step. For these languages, static analysis does not distinguish run-time errors explicitly and can be performed dynamically during edit-time [14]. The fundamentals of the analysis however remain the same and are presented in the following.

Lexical Analysis

Lexical analysis provides the basis for all static analysis. The characters of the source program are read one by one and categorized into *tokens*. These are strings of characters with an identifiable meaning, such as the operators, keywords, delimiters or identifiers specified by the programming language in use [14].

The following example aims to illustrate this. Consider the following statement.

```
1 if (x > 10) x = x + 1;
```

This statement would be labeled by a compiler into a stream of tokens that could look like the following.

```
1 operator("if") separator("(") identifier("x") operator(">")
2 integer("10") separator(")") identifier("x") operator("=")
3 identifier("x") operator("+") integer("1") separator(";")
```

The actual naming of these tokens would differ depending on the compilation or static analysis tool used. The result however always is a categorized set of characters for each statement.

Syntax Analysis

Syntax analysis checks if the provided tokens conform to the programming language's rules on forming valid expressions, but without considering the language's meaning and the program behavior. This is usually done by building an intermediate representation of the code called *parse tree*. The tree represents how a given expression is syntactically composed of terms and tokens. Syntax analysis is also called hierarchical analysis or parsing [14].

Considering again the example statement, such a parse tree in a simple form might look like the following.

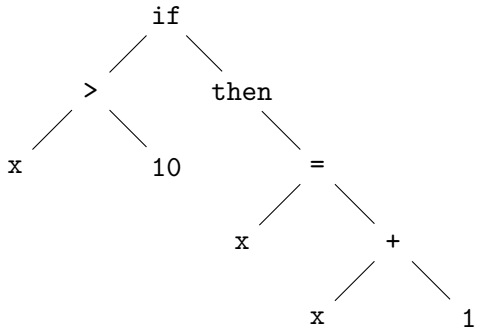

Figure 2.5.: Parse Tree for the Exemplary Statement

The example illustrates how syntax analysis recognizes hierarchical information. In order to execute the statement correctly, the compiler or analyzer needs to understand which part of the statement is dependent upon which other part of it. The syntax of a programming language defines that an `if` statement consist of a condition to evaluate and a statement to execute if that conditions is fulfilled. Similarly, an assignment is defined by its operator, which in turn defines the hierarchy of the following operations, and so on.

Semantic Analysis

Semantic analysis concludes the front-end phase of the compilation process. The concrete semantics of a program are not only determined by the semantic rules of the programming language in use but also the context of the program itself. With this information, the compiler determines whether the statement's execution is permissible [14].

Considering again the example, along the parse tree, a compiler would likely have to determine whether

- `x` is a valid integer variable and the operation `x + 1` is permissible,
- there is no type mismatch and the assignment `x = x + 1` is permissible,
- the evaluation `x > 10` conforms to the `if` operation and produces a boolean result,
- the `if` operation is permissible and the assignment `x = x + 1` conforms to it.

The scope of semantic analysis differs widely and most static analysis tools far exceed these basic semantic checks. The approach however is the same: preconfigured code patterns are identified along the parse tree and flagged if they are found. The capability of a static analysis tools is directly determined by both the quality and the quantity of the checks it includes. The aim of this thesis is to better illustrate the current possibilities of static analysis in chapter 4.

While most are not, some semantic analysis techniques are *mathematically sound*. Such techniques employ mathematical logic in order to prove properties of the semantics of a program. They are able to determine whether a statement is semantically permissible for every possible execution in the program context [4], [13]. The remaining sections of this chapter provide an introduction into how this is achieved.

2.4.2. Abstract Interpretation

The use of abstract interpretation in the context of computer program analysis began in 1976 with the first publication on the subject by Patrick and Radhia Cousot. Today, it is used as a general theory to approximate the possible semantics of a computer program. Its goal is to simplify a problem by abstraction and to infer a more general or partial solution to the problem from that abstraction. Apart from its use in program language development, its main application lies in static program analysis for software verification [4].

The concept of abstract interpretation is rather universal. A general mathematical example is the rule of signs:

"A negative minus zero is negative, a positive [minus zero] positive; zero [minus zero] is zero. When a positive is to be subtracted from a negative or a negative from a positive, then it is to be added" [4].

Given the set of all integers $\mathbb{Z}$, the sign can be understood as a property of each element of $\mathbb{Z}$. An abstract interpretation of the sign rule is performed by an abstraction function $\alpha_{\pm}$ which provides the abstraction of the sign property for a given set of integers S as follows [4].

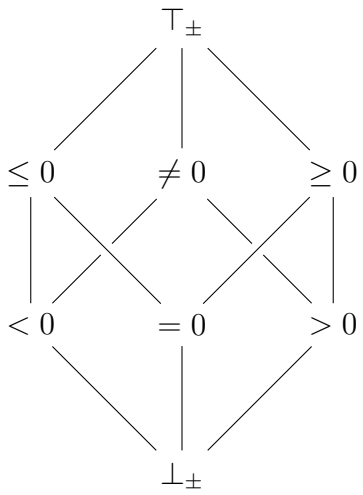

Figure 2.6.: Hasse Diagram of the Abstract Domain for $(\alpha_\pm, \gamma_\pm)$ (after [4])

Table 2.2.: Evaluation of the Abstraction Function $\alpha_\pm(S)$ for $S \subseteq \mathbb{Z}$

S	$\alpha_\pm(S)$
$S = \{\} = \emptyset$	$\bot_\pm$
$S \subseteq \{..., -3, -2, -1\}$	< 0
$S \subseteq \{0\}$	$= 0$
$S \subseteq \{1, 2, 3, ...\}$	> 0
$S \subseteq \{..., -2, -1, 0\}$	≤ 0
$S \subseteq \{..., -2, -1, 1, 2, ...\}$	$\neq 0$
$S \subseteq \{0, 1, 2, ...\}$	≥ 0
$S \subseteq \{..., -2, -1, 0, 1, 2, ...\}$	$\top_\pm$

The concretization function $\gamma_\pm$ provides the meaning of signs as sets of integers, so $\gamma_\pm(\bot_\pm) = \emptyset$, $\gamma_\pm(< 0) = \{..., -3, -2, -1\}$ and so on. Thus, a less precise abstraction of S is obtained, which allows to keep some information about S without having to retain the precise values of its members [4].

The pair of functions $(\alpha_\pm, \gamma_\pm)$ defines the *abstract domain*, which determines what levels of precision can be achieved and how they are hierarchically ordered. Hasse diagrams can be helpful in representing these relations, which for the above example is depicted in figure 2.6 [4].

In the abstract domain, the element "$= 0$" for example is more precise than "≥ 0": through the concretization function, the former evaluates to a smaller set than the latter. Likewise, "> 0" is more precise than "≥ 0" and represented accordingly in the diagram. "$= 0$" and "> 0" are incomparable, as there exists no shared subset between the two. "$\top_{\pm}$" can evaluate to every element of $\mathbb{Z}$ and thus is the least informative element. "$\bot_{\pm}$" is the contradictory element and evaluates to the empty set $\emptyset$.

Every operation performed in the concrete domain has a representation in the abstract domain. A subtraction of two integers $n \geq 0, m < 0$ under the rule of signs would according to [4] be executed in the abstract as

$$\alpha_{\pm}(\{n - m \mid n \in \gamma_{\pm}(\geq 0) \text{ and } m \in \gamma_{\pm}(< 0)\}) \tag{2.8}$$

$$= \alpha_{\pm}(\{n - m \mid n \in \{0, 1, 2, ...\} \text{ and } m \in \{..., -3, -2, -1\}\}) \tag{2.9}$$

$$= \alpha_{\pm}(\{1, 2, 3, ...\}) > 0 \tag{2.10}$$

The relations in the abstract domain allow to retain the best possible abstraction, for each operation, through entire computer programs. For every operation, an abstraction can be computed similarly to above. Following the control flow of a program, an analysis then is able to associate a level of information for each operation in accordance with figure 2.6. In this example it is trivial to see that a single operation $n - m$ where $n \geq 0$ and $m < 0$ always evaluates to > 0. For more complex operations in real computer programs with more complicated control flow, this can become more useful. While one operation might evaluate to > 0 and another to < 0, a later combination of both results is less informative but still evaluates to $\neq 0$ (the shared upper bound of both). Such less precise levels of information can still be helpful in determining properties of a program, in this case the absence of the possibility of a division by zero for example. Consider the following example, that arbitrarily transforms some input. The abstraction is performed statement by statement and noted to the right.

```
1 function transform(input) {              input ∈ (−∞, +∞)
2     x = input;                           x ∈ (−∞, +∞)
3     y = 42;                              y ∈ [42, 42]
4
5     if (x > 10) {                        x > 10 ⟹ x ∈ [11, +∞)
6         x = x * x;                       x ∈ [121, +∞)
7     } else {                             x ≤ 10 ⟹ x ∈ (−∞, 10]
```

```
8           x = x + y;                          x ∈ (−∞, 52]
9       }                                       x ∈ (−∞, +∞)
10
11      while (x < 100) {
12          x = x + 13;                         x ≥ 100 ⟹ x ∈ [100, +∞)
13      }
14
15      return x - y;                           x − 42 ≥ 58 ⟹ return ∈ [58, +∞)
16  }
```

The initial input has no specified bounds, so here `input` $\in (-\infty, +\infty)$. The following `if` condition results in branching control flow. If the condition is fulfilled, the abstraction becomes `x` $\in [121, +\infty)$. If not, it becomes `x` $\in (-\infty, 52]$ after the statement. The best possible abstraction after the control flow merges thus steps back to `x` $\in (-\infty, +\infty)$. This illustrates how the stepwise calculation of the best abstraction with the help of the abstract domain can still be useful for parts of the control flow, even though the constraints might later collapse into a coarser category.

Through the `while` loop and following operations, the abstraction becomes more informative with `x` ≥ 100 so finally `return` $\in [58, +\infty)$. Any additional operations before the `return` statement as well as further uses of the function's returned value could safely be abstracted to > 0 for every execution of the program.

In general, if an analysis by abstract interpretation concludes a property to hold, it will always hold in every execution of the program. More concretely, when such a property is the absence of a specific kind of defect, this means that the analysis does not provide false negatives (see table 2.1), no defect is missed [13].

To remain computable, sound techniques however sacrifice mathematical completeness: If the property is true in the abstraction, the analysis won't be able to always prove it in the concrete. There will be true properties that the analysis fails to prove [4]. More concretely, this means that false positives are still possible [13].

How useful abstract interpretation as an analysis method is depends entirely on the abstract domain. The worst-case scenario is that the analysis can only provide inconclusive abstraction results (in the example $\top_{\pm}$) due to an unprecise abstract domain [15].

A desirable outcome would be an abstraction that allows to exclude the type of defect targeted by the analysis. This refinement is theoretically always possible, but might not be computable by a machine. Finding efficiently computable and still precise abstractions is very difficult in practice [4].

2.4.3. Model Checking

Another formal verification method relevant in this context is model checking. The technique requires a finite state transition model that describes the behavior of a control system. By systematically exploring all possible execution paths, it can be shown that a given system model satisfies given properties [5].

Finite state machine (FSM) are useful system descriptions in almost any application domain. They allow to model complex system behavior precisely and without ambiguity, by clearly describing the relations between system inputs, outputs and conditions. Following [12], a FSM is defined as a tuple $M = (Q, q, \Sigma, h, F)$ where

- $Q \neq \varnothing$ is the finite *state space*, the set of states the FSM can be in,
- $q \in Q$ is the *initial state*, the state in which the FSM always starts,
- $\Sigma \neq \varnothing$ is the finite *input alphabet*, the set of symbols that the FSM is able to read,
- $h \subseteq Q \times \Sigma \times Q$ is the *transition relation* that describes which state follows which combination of previous state and given input,
- $F \in Q$ is the set of *final states*, where the FSM terminates.

Properties in this context are characteristics that a FSM exhibits, usually of qualitative nature. They must be formulated in an unambiguous syntax, which in the context of model checking is based on *propositional logic*, which is introduced in the following based on reference [5].

Propositions can be any factual statement. They are atomic if they are singular and can not be broken down further. A finite set of atomic propositions is declared AP, of which single elements are denoted by latin letters $a, b, \ldots$. Conventional logical operators indicate whether a propositions holds or not. For atomic propositions $a, b \in AP$ conjunction would be expressed as

$$a \wedge b \tag{2.11}$$

which holds if and only if both propositions a and b hold. Negation would be denoted as

$$\neg\, a \tag{2.12}$$

which holds if and only if a does not hold. Other operators are derived: disjunction can be expressed as

$$\neg(\neg\, a \wedge b) = a \vee b \tag{2.13}$$

and implications as

$$\neg\, a \vee b = a \rightarrow b \tag{2.14}$$

just to name the most important relations.

To be useful for the analysis of state transition systems, propositional logic must be extended by *temporal* modalities. Two elemental operators are introduced: $\Diamond$ indicates "eventually", i.e. "eventually in the future", and $\Box$ indicates "always", i.e. "now and forever in the future". Based on propositional logic, two types of semantics exist that allow the specification of system properties [5].

Linear temporal logic (LTL) is based on the notion that time in a transition system proceeds linearly, i.e. that at each moment in time there is a single succeeding computation. To express corresponding properties, two additional modalities are introduced: $\bigcirc$ indicates "next", i.e.

$$\bigcirc\, a \tag{2.15}$$

holds when at the next computation step a holds; and $\cup$ indicates "until", i.e.

$$a \cup b \tag{2.16}$$

holds when at some future moment b holds and a holds until that moment [5].

Computation tree logic (CTL) is based on the notion of branching time, where the time in a transition system may split into alternative courses. Some properties can only be expressed in LTL and some others only in CTL. Because with CTL at every moment there may be *several* possible future computations, it is possible to formulate properties that only pertain to *some* computations originating from a specific state and not all. For this, two additional path qualifiers are introduced: $\exists$ indicates "exists", i.e.

$$\exists \Diamond\, a \tag{2.17}$$

holds when there is at least one execution path along which eventually a is fulfilled; while $\forall$ indicates “for all”, i.e.

$$\forall \Diamond a \tag{2.18}$$

holds when all execution paths eventually fulfill a [5].

For illustration, consider the FSM shown in figure 2.7. It is an example from reference [16] that checks whether an input contains the sequence $0, 1$ or the sequence $1, 0$ by traversing either the state q_1 or q_2 respectively. The state q_3 is an accepting state where the automaton accepts the input sequence and terminates execution.

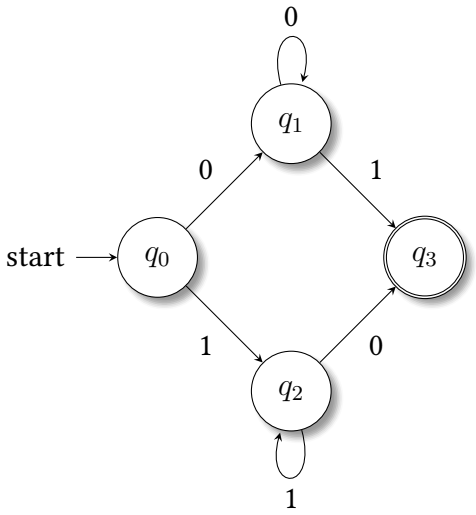

Figure 2.7.: Example FSM (modified from [16])

With appropriate properties an implementation of this FSM could be verified by a model checking tool. By defining the set of atomic propositions for the individual states $AP = q_0, q_1, q_2, q_3$ these can be formulated. For example, if the required sequence is not provided, the FSM must loop infinitely. This could be described by

$$\exists \Box q_1 \tag{2.19}$$

which implies that there exists at least one path along which the current state always is q_1 (or q_2 respectively). That every state has an outgoing transition could be described by

$$\forall \Box \exists \bigcirc true \tag{2.20}$$

which means that on all paths there always exists one path for which a next step exists. Further, the accepting state must be reachable from every state. This could be described by

$$\forall \Box \exists \Diamond q_3 \tag{2.21}$$

which means that on all paths there always exists one path that eventually reaches q_3. That this state is in fact accepting could be described by

$$\forall \Box (q_3 \rightarrow \forall \bigcirc q_3) \tag{2.22}$$

which means that for all paths being in state q_3 always implies that the next state is also q_3. That the corresponding sequence of states for a correct input actually exists can be described by

$$\exists \Diamond (q0 \wedge \exists \bigcirc (q1 \wedge \exists \bigcirc q3)) \tag{2.23}$$

which implies that there exists a path that eventually follows the sequence q_0, q_1, q_3 (or q_0, q_2, q_3 respectively), which is here specified with nested conjunctions. Extending AP with propositions for explicit input values would allow to further add properties that more directly verify the existence or absence of system conditions.

A model checking tool typically creates the system model itself after being provided a suitable specification. The properties have to be provided in a suitable syntax too, against which the tool then verifies the model automatically by systematically laying out and exploring all possible execution paths through the model [5].

The considerations so far should have made clear that model checking allows to verify quite elaborate characteristics. The technique is however appropriate for logic-intensive applications and less suited for data-intensive applications, as that data typically ranges over infinite domains [5].

Nevertheless, MathWorks' model checking tool Simulink Design Verifier attempts to be compatible with any control system application. It implements Stalmarck's proof procedure for propositional logic, which enables it to perform well in industry-scale scenarios [17], [18]. How exactly the tool converts a Simulink model into a compatible state transition model is not publicly available. It however has to forgo many of the strengths of model checking in the process. The tool is introduced in chapter 4 and further discussed in chapter 6.

3. Motivation and Goal

At the time of writing, the space industry is experiencing a strong shift towards the development of reusable launch vehicles. As such, the DLR is involved in the project Cooperative Action Leading to Launcher Innovation in Stage Toss-back Operations (CALLISTO) as well as pursuing the Reusability Flight Experiment (ReFEx) initiative. The Institute of Space Systems is among other things contributing with the development, verification and validation of the GNC subsystems for the two spacecraft.

The subsystem components work together conceptually as illustrated in figure 3.1. This strongly simplified architectural overview was adapted from [7] to better show the resemblance to the standard control loop presented in figure 2.1. Both the guidance and the control function are realized within the Guidance and Control Computer (GCC) or On-Board Computer (OBC) respectively while the navigation function is assumed by the Hybrid Navigation System (HNS). As opposed to highly integrated systems, such a distributed architecture is more versatile and can be adapted to different projects with less effort. By clearly defining the interfaces to other functions, development and verification can be more focused on the particular function [7].

The GNC algorithms are developed in MATLAB/Simulink using the software management principles outlined in chapter 2. There are similarities in the development processes across

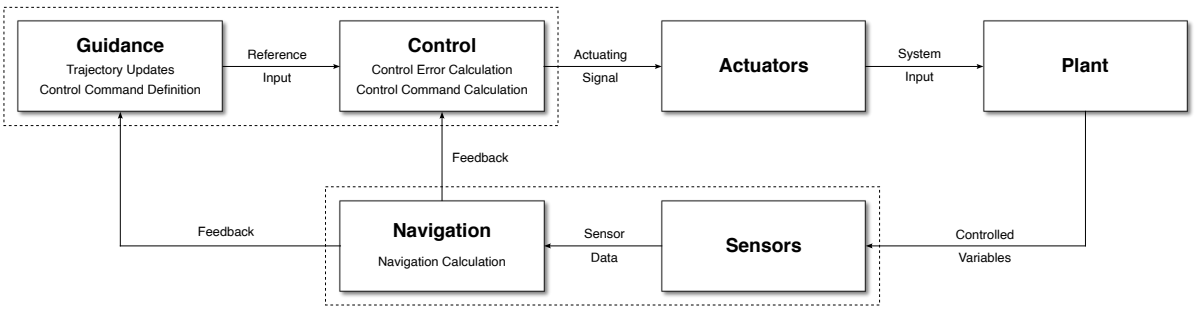

Figure 3.1.: Simplified Functional GNC Architecture (modified from [7])

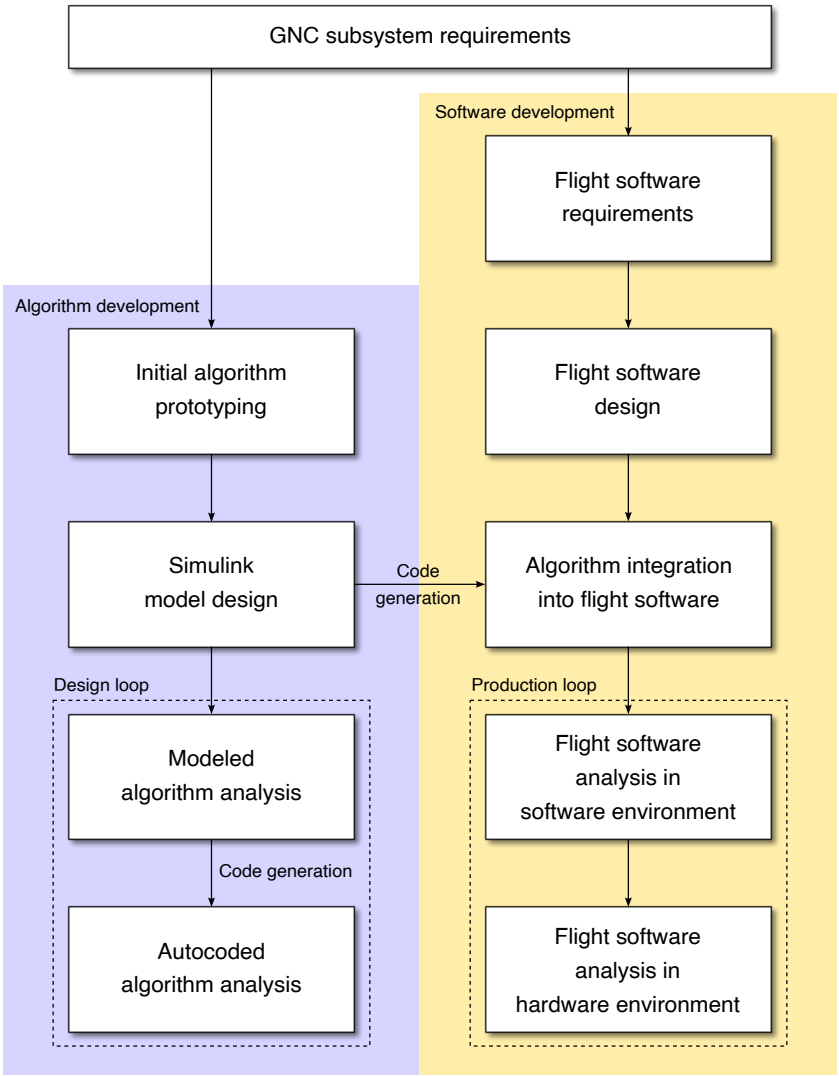

Figure 3.2.: Spacecraft GNC Development Process (modified from [19], [20])

agencies and projects. Authors at the National Aeronautics and Space Administration (NASA) NASA Engineering & Safety Center (NESC) note how the automatic generation of code from models has shifted development processes to be more parallel. For projects such as the space shuttle, GNC development was linearly proceeding from GNC subsystem requirements to algorithm design and via derived requirements to flight software development. For the Orion Multi-Purpose Crew Vehicle (MPCV), the GNC and flight software development teams are working side-by-side in the production of the software artifacts that lead to the onboard flight code. MATLAB/Simulink modeling tools are used to auto-generate the GNC algorithmic flight software as C++ code [19], [20]. An overview of this approach is presented in figure 3.2.

At the DLR Institute of Space Systems, the development and verification process is similar. There are different repositories in use depending on the project affiliation, function and stage in the GNC development process of the software they contain. The individual repositories are subject to a quality assurance process based on the fundamentals of software management and testing outlined in chapter 2. Unit testing is done predominantly using the MATLAB unit testing framework and GitLab CI, while system testing is done by running extensive simulation campaigns.

This verification approach generally resembles conventional functional verification, where the test objects are verified against their functional requirements. What this thesis aims to add is an evaluation of verification tools that so far are not integrated with the overall development process. The goal is to include techniques originating from theoretical computer science in the areas of program analysis and formal verification. Among others, these consider the formal term of *program correctness*: A program is considered to be correct when it produces a correct output for *every* acceptable input [21]. This notion succinctly captures the motivation behind the thesis at hand. Rather than analyzing model performance or simulation parameters, the considerations that follow are more closely related to systematically targeting the detection of software faults and increasing confidence in correct software execution throughout the development process. This process is strongly influenced by the tools that are used, which requires the compatibility of the proposed verification techniques to be considered.

The research question for this thesis thus is: *"How suitable are MathWorks verification tools for the automated verification of GNC software with respect to correctness?"* This question delineates the intention behind this thesis explicitly:

- The considerations are limited to verification tools provided by MathWorks. If suitable, recommendations for alternatives are given in the end.
- The verification techniques shall be suitable for automation. This mainly pertains to software execution but can also involve automated results evaluation.
- The scope of verification are GNC systems. The specifications involved are more data-intensive than in other application domains.
- The verification goal is software correctness as opposed to system performance.

By answering this question, the goal is not only to give an initial evaluation, but also to develop a basic framework and workflow around the verification tools that applies the lessons learned during the investigation.

4. Methods and Tools

With a good understanding of their theoretical foundation, the verification tools within the scope of this thesis can be introduced. In order to better understand practical aspects and limitations, a model of a more or less arbitrarily chosen example system is introduced. This model has similar characteristics to the actual models in development at the institute with a lower complexity. It serves as the basis for evaluating the tools and discussing their recommended use.

4.1. Thrust Controller Simulation

After some preliminary literature research, thrust control of chemical propulsion systems was selected as the application domain for an exemplary simulation. The associated control problem is presented in figure 4.1, which has been adapted from reference [22] and adjusted to the arrangement of a standard control loop. It corresponds to a conventional pressure-fed bipropellant propulsion system, where the supply pressure inside their respective tanks supplies fuel and oxidizer through feedlines, control valves and injector elements into a main combustion chamber. Without elaborating on the specifics of rocket engine design, it is important here to know that the combustion chamber requires fuel and oxidizer in a specific mixture ratio

$$r_m = \frac{\dot{m}_o}{\dot{m}_f} \tag{4.1}$$

of oxidizer and fuel mass flow. It sets the combustion temperature and therefore the engine's performance and material temperature, while the total mass flow corresponds to the desired thrust of the engine [22].

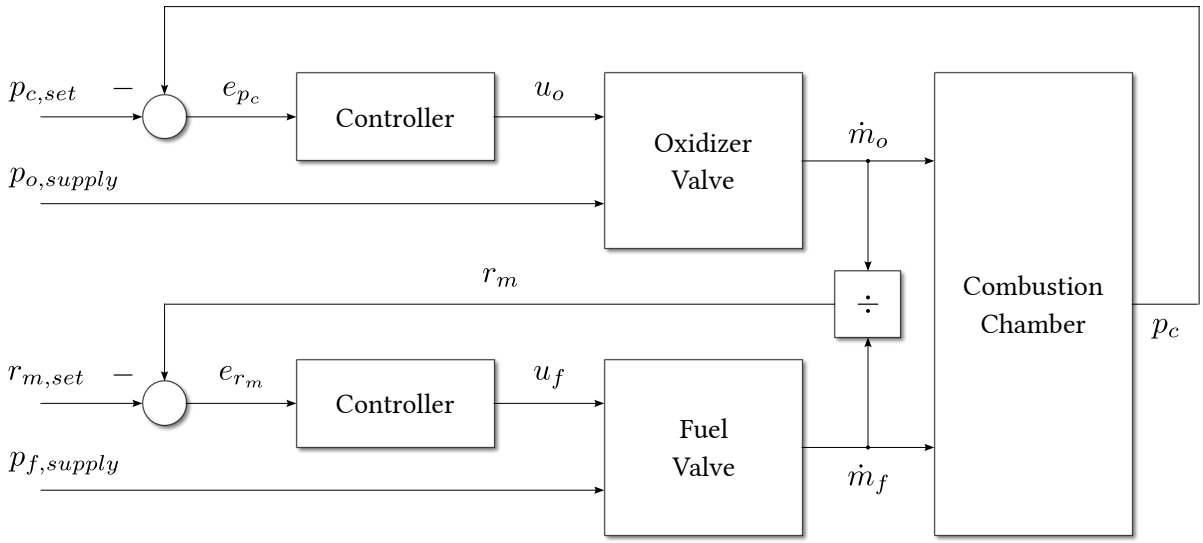

Figure 4.1.: Thrust Control (modified from [22])

There are two actuators in such a configuration, namely the fuel and the oxidizer valve, which need to be controlled. The thrust of the engine corresponds to the chamber pressure P_c, which is the primary controlled variable and usually associated with the propellant with the higher mass flow. The mixture ratio r_m is determined by dividing oxidizer and fuel mass flow, which is fed back into the mixture ratio control loop governing the fuel mass flow [22].

These two control loops interact dynamically, so keeping both controlled variables near their set points presents a significant challenge. Experience shows that the mixture ratio control loop should be tuned to react faster to set point deviations than the chamber pressure control, which keeps the engine temperatures at the design conditions [22].

Combustion Chamber

With the information from [22] and [23], it is possible to derive a simplified linear time-invariant description of the combustion chamber dynamics as follows.

For a combustion chamber with volume V_c, the total mass flow into the chamber equals the sum of the outflow through the nozzle plus the rate of change of mass stored in the chamber [23].

$$\dot{m}_t(t) = \dot{m}_{\text{out}}(t) + \frac{d}{dt}\Big(\rho_c(t)\, V_c\Big) \tag{4.2}$$

The combustion is assumed to be isothermal and the combustion temperature T_c to be constant. With the specific gas constant R the density $\rho_c(t)$ becomes

$$\rho_c(t) = \frac{P_c(t)}{R\,T_c}. \tag{4.3}$$

Further, the flow is assumed to be choked, which is a simplification that allows the mass outflow $\dot{m}_{\text{out}}(t)$ for a given throat area and exhaust velocity to be approximated with

$$\dot{m}_{\text{out}}(t) = \frac{A_t}{c^*} P_c(t). \tag{4.4}$$

The throat area A_t is given by the dimensions while the characteristic velocity c^* is determined by the characteristics of the combustion itself. Substituting yields

$$\dot{m}_t(t) = \frac{A_t}{c^*} P_c(t) + \frac{d}{dt}\left(\frac{P_c(t)}{R\,T_c} V_c\right) = \frac{A_t}{c^*} P_c(t) + \frac{V_c}{R\,T_c}\frac{dP_c(t)}{dt}. \tag{4.5}$$

In order to derive the transfer function, a Laplace transformation is performed.

$$\dot{m}_t(s) = \frac{A_t}{c^*} P_c(s) + \frac{V_c}{R\,T_c} s P_c(s) = P_c(s)\left(\frac{A_t}{c^*} + \frac{V_c}{R\,T_c} s\right). \tag{4.6}$$

Rearranging yields

$$\frac{P_c(s)}{\dot{m}_t(s)} = \frac{1}{\frac{A_t}{c^*} + \frac{V_c}{R\,T_c} s} = \frac{\frac{c^*}{A_t}}{1 + \left(\frac{V_c\ c^*}{R\,T_c\ A_t}\right) s}. \tag{4.7}$$

This represents a transfer function in standard form

$$G(s) = \frac{P_c(s)}{\dot{m}_T(s)} = \frac{K}{1 + \tau s} \tag{4.8}$$

with $K = \dfrac{c^*}{A_t}$ and $\tau = \dfrac{V_c\ c^*}{R\,T_c\ A_t}$.

In order to arrive at the state-space representation introduced in section 2.1, first a suitable state variable has to be chosen, which here is the chamber pressure P_c.

$$x(t) = P_c(t) \tag{4.9}$$

Rearranging equation 4.5 to

$$\frac{V_c}{R\,T_c}\frac{d}{dt} P_c(t) = \dot{m}_T(t) - \frac{A_t}{c^*} P_c(t) \tag{4.10}$$

enables to isolate $\frac{d}{dt}P_c(t)$ as

$$\frac{d}{dt}P_c(t) = \frac{R\,T_c}{V_c}\dot{m}_T(t) - \frac{R\,T_c\,A_t}{V_c\,c^*}P_c(t). \tag{4.11}$$

The input to the system is the total mass flow $\dot{m}_t(t)$:

$$u(t) = \dot{m}_t(t). \tag{4.12}$$

So the state equation becomes

$$\dot{x}(t) = -\left(\frac{R\,T_c\,A_t}{V_c\,c^*}\right)x(t) + \left(\frac{R\,T_c}{V_c}\right)u(t) \tag{4.13}$$

with $A = \dfrac{R\,T_c\,A_t}{V_c\,c^*}$ and $B = \dfrac{R\,T_c}{V_c}$. The output equation simply is

$$y(t) = x(t) \tag{4.14}$$

so $C = 1$ and $D = 0$. This represents a linear, time-invariant system assuming a combustion at a constant temperature. Suitable values in accordance with [24] for an exemplary propulsion system with storable propellants are presented in table 4.1.

Table 4.1.: Combustion Values for an Exemplary Propulsion System

Parameter	Value
Specific gas constant R	$300\,\mathrm{J\,kg^{-1}\,K^{-1}}$
Chamber temperature T_c	$3200\,\mathrm{K}$
Characteristic velocity c^*	$1700\,\mathrm{m\,s^{-1}}$
Throat area A_t	$1 \cdot 10^{-4}\,\mathrm{m^2}$
Chamber volume V_c	$5 \cdot 10^{-3}\,\mathrm{m^3}$

Valves

While references [22], [23] do not provide details on the dynamic behavior of the actuators, reference [25] notes that their dynamics can not be neglected with respect to the combustion dynamics themselves. The valves are therefore approximated with an assumed first-order lag between the command $u(t)$ they receive and their actual opening $x_v(t)$. This is characterized

by the valve's response time τ_v, so the valves dynamic behavior is described by the differential equation

$$\tau_v \frac{dx_v}{dt} + x_v(t) = u(t) \tag{4.15}$$

which corresponds to the already introduced standard form in the frequency domain

$$x_v(s) = \frac{1}{\tau_v s + 1} u(s). \tag{4.16}$$

In order to avoid large differences in the orders of magnitude in the simulation, the valve 's opening position $x_v(t)$ is normalized to the range [0,1] and mapped linearly to the valve's actual output mass flow $\dot{m}(t)$:

$$\dot{m}(t) = \dot{m}_{\max} x_v(t). \tag{4.17}$$

The valve's transfer function from command $u(t)$ to mass flow $\dot{m}(t)$ then becomes

$$\frac{\dot{m}(s)}{u(s)} = \frac{\dot{m}_{\max}}{\tau_v s + 1}. \tag{4.18}$$

To arrive at the state-space representation for the valves, as state the opening of the valve is chosen:

$$x(t) = x_v(t). \tag{4.19}$$

Equation 4.15 representing the valve dynamics can be rearranged to obtain the state equation

$$\dot{x}(t) = -\frac{1}{\tau_v} x(t) + \frac{1}{\tau_v} u(t) \tag{4.20}$$

so $A = -\frac{1}{\tau_v}$ and $B = \frac{1}{\tau_v}$. The output of the valve is the mass flow through it, so the output equation is

$$y(t) = \dot{m}_{\max} x(t) \tag{4.21}$$

so $C = \dot{m}_{\max}$ and $D = 0$.

Realistic values for the valve's characteristics can be obtained from suppliers of propulsion systems and are listed in table 4.2 in accordance with [26].

Table 4.2.: Estimate Values for Exemplary Propulsion Valves

Parameter	Value
Response time τ_v	$0.1\,\mathrm{s}$
Maximum mass flow $\dot{m}_{max}$	$0.1\,\mathrm{kg\,s^{-1}}$

Simulation Model

For the implementation in Simulink, the configuration presented in figure 4.1 is rearranged to allow for one consolidated controller subsystem. The actuator and plant dynamics are modeled using state-space blocks. The resulting model is depicted in figure 4.2, the content of the controller subsystem in figure 4.3.

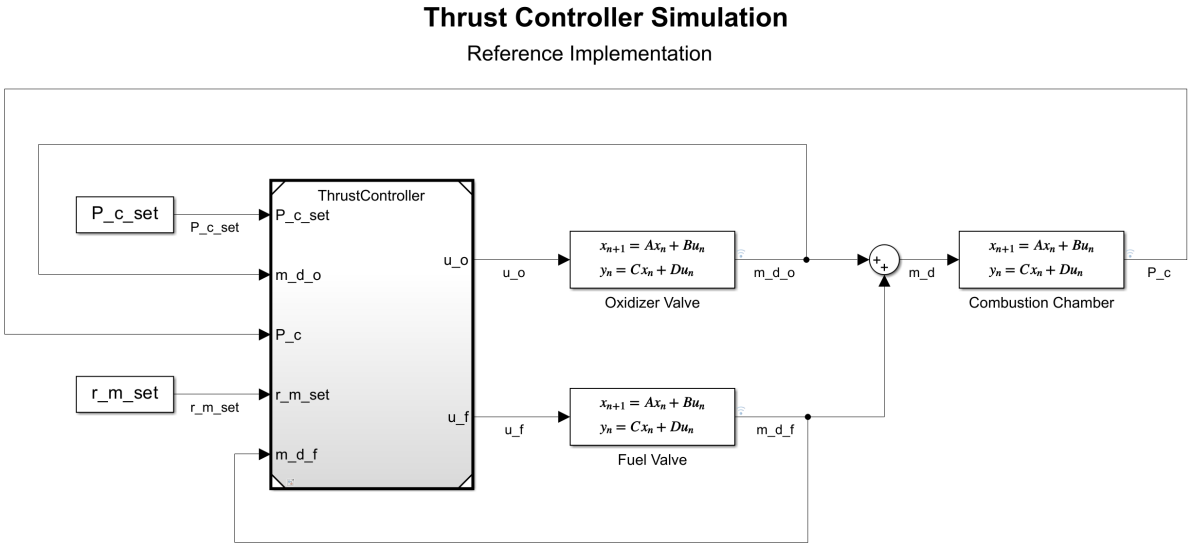

Figure 4.2.: Thrust Control Simulation

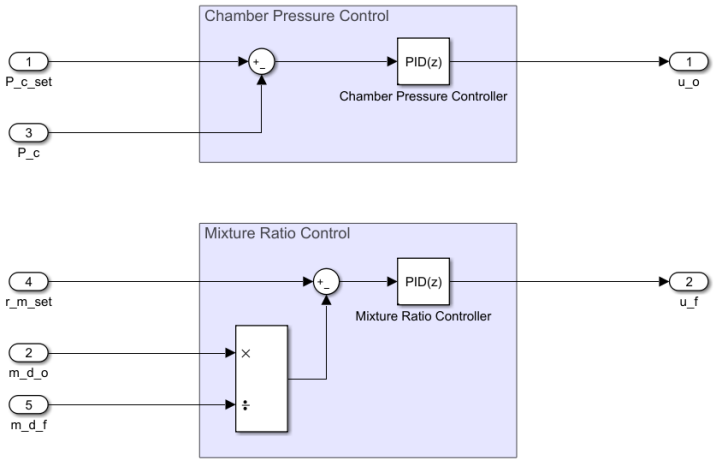

Figure 4.3.: Thrust Controller

For simulation, the controller receives the objective of achieving a chamber pressure set point of $P_{c,set} = 1\,\mathrm{MPa}$ while the mass flows of oxidizer and fuel flow converge toward a mixture ratio set point of $r_{m,set} = 1.5$. Ideally, there should be no overshoot in the chamber pressure and a moderately fast settling of the process variables. The Control System Toolbox is used to tune the controller gains for this scenario, which leads to the dynamics shown in figure 4.4. The desired chamber pressure is achieved in just under 2 seconds, while fuel and oxidizer mass flow settle at the corresponding levels and ratio.

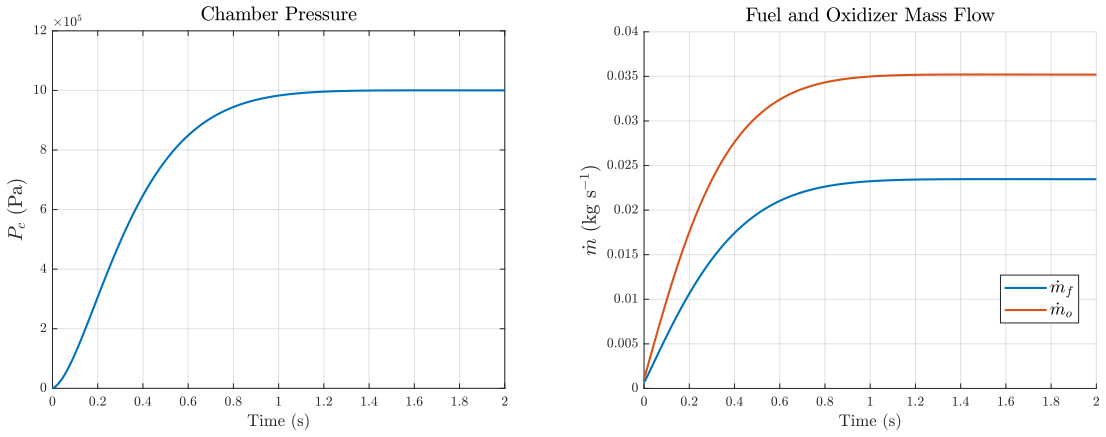

Figure 4.4.: Controller Simulation Results

4.1.1. Controller Function

To develop a more expressive example, the controller functionality is implemented programmatically inside a MATLAB function. In the resulting variant of the simulation model, the new function `thrustControlFunction` is called by a MATLAB Function block inside the Controller subsystem. It contains the sample time and controller gains as hard-coded values and uses persistent variables to store and accumulate the integrator states over the duration of a simulation.

Inside the function, first the control output for the oxidizer valve is calculated for the chamber pressure control loop. Then, the control output for the fuel valve for the mixture ratio control loop is calculated. An optional guard against division by zero is introduced here for evaluation purposes later. The function can be found in appendix A.1.

4.1.2. Finite State Machine

In order to obtain an implementation for the evaluation of model checking, an additional variant of the controller implementing a FSM is required. Figure 4.5 depicts the arbitrary control logic that is used.

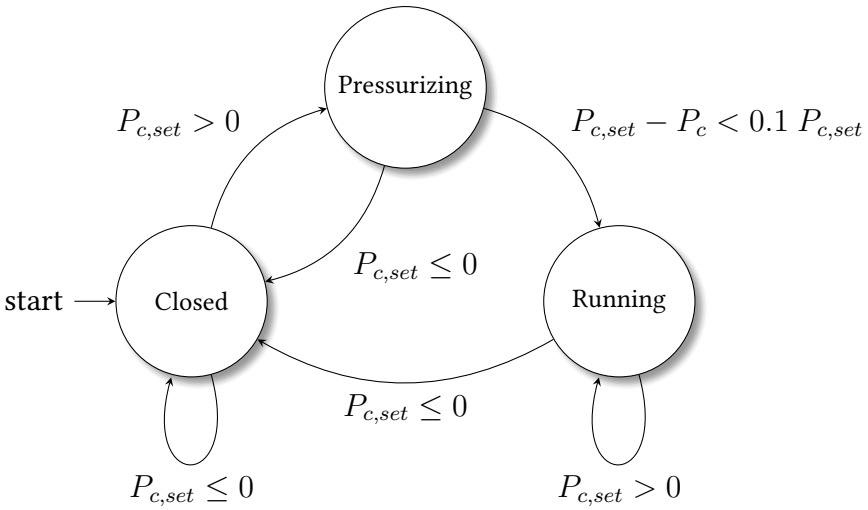

Figure 4.5.: Controller State Machine

The controller sets the two control outputs u_o and u_f depending on the actual and set point chamber pressure P_c and $P_{c,set}$. Initially, the control outputs are set to $u_o = u_f = 0$ so the

valves remain closed. As soon as a positive $P_{c,set}$ is commanded, the valves open to a fixed position by setting u_o and u_f to a constant value. After P_c is close to $P_{c,set}$, the controller calculates the outputs with the function `computeControl`, which performs the same calculations as `thrustControlFunction` but without initializing the persistent variables.

The states are defined as enumerators inside the class `ThrustControllerStates`. The class inherits from the `Simulink.IntEnumType` class. This is required for enumerations that are used in Simulink and are intended for code generation. The enumerators have an enumerated name and an underlying integer which is used internally and in the generated code [10].

```
classdef ThrustControllerStates < Simulink.IntEnumType
    enumeration
        Closed        (0)
        Pressurizing  (1)
        Running       (2)
    end
end
```

The MATLAB Function block inside the Controller subsystem of this new simulation variant now calls the function `thrustControlFSM`. This function defines a new persistent variable `currentState` that holds the enumerator for the current state of the controller. The FSM itself is then implemented as a switch case statement. The conditions for the transitions are programmed with nested `if` statements as shown in the following excerpt.

```
% Switch for current state
switch currentState
    case ThrustControllerStates.Closed
        if P_c_set > 0
            % Transition Closed to Pressurizing
            [u_o, u_f, currentState, intC, intMR] = ...
                updateClosedToPressurizing(P_c_set, P_c, r_m_set, ...
                m_d_o, m_d_f, intC, intMR);
        else
            % Remain in Closed
            [u_o, u_f, currentState, intC, intMR] = ...
                remainClosed(currentState, intC, intMR);
        end
    ...
end
```

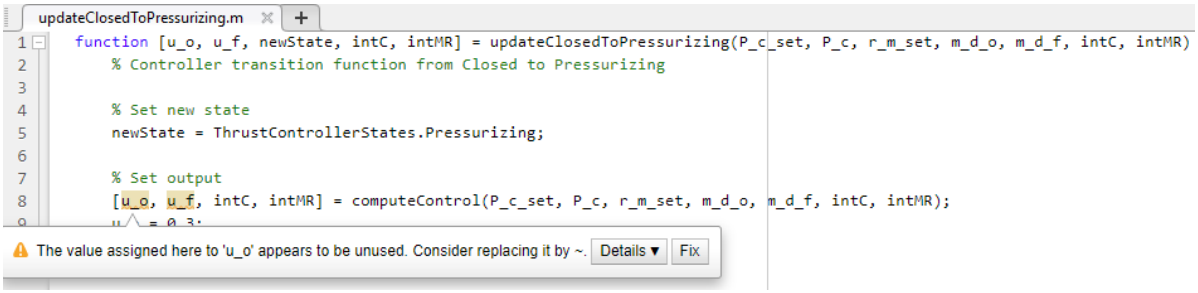

Figure 4.6.: Code Analyzer Issue

The transitions themselves are embedded in dedicated transition functions that increment the integrators and set the desired control outputs. All associated functions can be found in appendix A.1.

4.2. Verification

The simulation is a representative example for the actual systems that need to be verified using the methods that follow. Here, the controller subsystem represents the product that would be verified and delivered. The goal of this section is to provide a basic example on how to employ each verification tool within that scope. The focus is to gain a basic understanding and run the verification programmatically, so it can be automated in section 4.3.

4.2.1. Static Analysis of MATLAB Code

MATLAB originally started out as an interpreted language: It didn't need to be compiled before execution and was executed command-by-command by its own interpreter. Today, the language still does not require a dedicated compilation step, but is compiled *just-in-time*, i.e. as it is executed, by its execution engine. This increases performance over interpretation [27].

Static analysis in MATLAB is run automatically during edit-time by the Code Analyzer as shown in figure 4.6. As soon as one of its checks identifies an issue in the written code, the line is flagged in the editor interface with a message describing the issue.

Programmatic access to the Code Analyzer is provided by the `checkcode` function. It runs the available static analysis checks on the files specified when calling the function and returns

the results in a *structure array*, containing the messages and their locations [6]. Structure arrays are data types subject not only to MATLAB that are useful for grouping related data. The `checkcode` function provides options to return more information, such as the message identifications as shown in the following listing.

```
% Define file
files = 'script.m';

% Run analysis
results = checkcode(files, '-struct');
```

A Code Analyzer message can be suppressed manually by writing a `%#ok` directive in the concerned line of code. The analysis can be configured to ignore these suppression. Further configuration of the analysis can be done in the Code Analyzer user interface and reused programmatically with a configuration file [6].

As the analysis already is available to a developer during edit-time, it is prudent to simply employ the `checkcode` function as a safeguard to verify whether the files committed to a repository in fact do not raise any messages. Refer to chapter 5 for the technical details of how this is implemented.

In newer releases of MATLAB, the issues found by the Code Analyzer are stored in a `codeIssues` object. Notably, this object can be used with a `fix` function to fix certain issues programmatically and a `export` function to export the results in a standardized format [6]. This is out of the scope for this thesis, but should be considered for future applications.

Checks

In order to understand the scope of the Code Analyzer in comparison to the other static analysis tools investigated in the remainder of this chapter, a review of its checks is performed based on the product's documentation [6]. A summary is presented in table 4.3.

Of special interest here are the checks related to code generation compatibility. For them, the analysis has to be explicitly activated by adding the `%#codegen` directive to the file. Other than that, the checks aim at achieving correct and efficient MATLAB code but do not have the same scope as a comprehensive coding standard that for example targets robustness or security of code employed in embedded systems.

Table 4.3.: MATLAB Code Analyzer Checks

Group	Checks	Verification objective
Incomplete analysis	18	Reason why an analysis could not be started or completed, for example due to too high complexity, too deep nesting or invalid files
Syntax errors	50	Valid use of characters, missing characters, valid use of operators and valid character sequences
Language specification errors	155	Valid use of the MATLAB language, for example in defining attributes, declaring functions, defining classes, using certain language-specific constructs
Bugs	36	Various cases of syntactically detectable dead logic, correct use of operands, availability of variables
Custom checks	19	Various coding constructs that can be configured to be flagged as disallowed
Compatibility considerations	>200	Usage of functions, methods and properties that have been or will be removed or replaced in MATLAB and its toolboxes
Good practices	103	Recommendations for coding constructs that can be inefficient, unnecessary, too complicated or can lead to unwanted or unexpected behavior
Unset variables	7	Missing or not executed variable definitions
Unused constructions	21	Flagging of functions, arguments, operations etc. that might not be used anywhere
Suggested improvements	>200	Recommendations for using outdated functions, methods or properties
Readability	35	Possible simplifications for the use of certain unnecessary or more complicated statements
Formatting	7	Unnecessary or recommended use of commas, parentheses, semicolons
Performance	44	Recommendations for the use of certain statements that unnecessarily decrease execution performance

Table 4.4.: MATLAB Code Analyzer Checks (continued)

Group	Checks	Verification objective
Code generation	20	Usage of functions, statements and other constructs supported for code generation
Deployment	10	Compatibility relating to packaging and deploying MATLAB programs as standalone applications
System objects	9	Valid specification relating to MATLAB system objects used in system blocks within Simulink
Unsupported features	13	Flagging of functions that are not supported officially
Behavior changes	>200	Flagging of instances where MATLAB handles or will handle statements differently in another release

4.2.2. Static Analysis in Simulink

Simulink offers a static analysis method for models with the Model Advisor. Its capabilities fall into the scope of both syntactic and non-sound semantic analysis techniques introduced in section 2.4. Its functionality is integrated into the user interface and meant to assist during the modeling process. Just like other static analysis tools, it runs a variety of checks with a predefined objective [10].

Simulink Check extends this functionality with around 300 additional checks with varying scope. After discussing how they can be run, they are presented based on their classification in the product.

MATLAB creates an instance of a `Simulink.ModelAdvisor` object for each model that is opened in the current session. Its `getModelAdvisor` method returns a handle to the object for the model or subsystem that was specified. It offers object functions to select individual or groups of checks, run checks, read results and create reports among others [10].

Simulink Check opens up the application programming interface (API) to the Model Advisor, allowing for the model analysis to be run using the `ModelAdvisor.run` method. It receives a cell array of check names and systems as input arguments and returns one `ModelAdvisor`

`.SystemResult` object per system that was analyzed that in turn contains a `ModelAdvisor.CheckResult` object for each check that was run [28]. The following listing already presents sufficient code to run a Model Advisor analysis programmatically.

```
% Define checks and model
checkIDs = 'mathworks.design.UnconnectedLinesPorts';
systems = 'Controller';

% Run analysis
results = ModelAdvisor.run(systems,checkIDs);
```

Design Checks

The checks available just without Simulink Check have the `mathworks.design` prefix. They target common problems with several Simulink features, however also do not represent a comprehensive standard.

The analyzed aspects mostly go beyond the complexity of the demonstration example. They however also do not allow for a comprehensive review with a specific goal but rather represent a selection of common issues and good practices in specific cases. These checks in general are expected not to fail when run and are therefore incorporated as another safeguard that ensures there are no fundamentals flaws introduced in a Simulink model.

Table 4.5.: Model Advisor Checks

Group	Verification objective
Modeling elements	Identify common issues with standard Simulink modeling elements such as merge, outport or integrator blocks as well as unconnected lines and ports
Model referencing	Identify incorrect configuration settings with respect to including a Simulink model inside another as a *model block*
Model file integrity	Identify problems with character encoding and nondefault model properties
Unit inconsistencies	Identify disallowed, undefined, mismatching or ambiguous unit specifications and conversions
Library links	Identify common problems with the usage of blocks that are specified in a library model
Simplified initialization	Identify incorrect configuration settings with respect to *simplified initialization*, which aims to make model initialization more predictable and consistent
Model upgrades	Access to the Upgrade Advisor feature to identify the applicability of features introduced in newer releases of Simulink
Bus usage	Identify common problems with *buses*, which are commonly used to group signals and parameters
Code generation efficiency	Identify model configuration settings that can result in a lower efficiency of code generated from the model
Data transfer efficiency	Identify settings and modeling techniques that can result in low data transfer efficiency

Advisory Checks

The MathWorks Advisory Board (MAB) provides an extensive set of guidelines for the development of control algorithms with MATLAB/Simulink and their use in embedded systems. They originate from the MathWorks Automotive Advisory Board (MAAB) established in 1998

by Ford, Daimler Benz, and Toyota. Later, both the MAAB and Japan MathWorks Automotive Advisory Board (JMAAB) developed their own standards, which are now incorporated into one global MAB standard [28].

The standard has a much wider scope than the default Model Advisor checks discussed before. It appears suitable as a general guideline for modeling with a variety of rather basic rules. The MAB structures the guidelines as shown in table 4.6.

High Integrity System Modeling Checks

High Integrity Systems Modeling (HISM) is a concept with which MathWorks refers to set of standards that apply to software engineering in different domains. Internally, the associated checks have the prefix `mathworks.hism`. They incorporate rules from the following standards and domains [28].

- Radio Technical Commission for Aeronautics (RTCA) documents DO-178C and DO-331 for the aerospace and defense domain
- International Organization for Standardization (ISO) standard 26262 for the automotive domain
- European Standard (EN) 50128 and 50657 for the rail and transportation domain
- International Electrotechnical Commission (IEC) standard 61508 for the industrial automation and robotics domain
- ISO 13485 and IEC 62304 for medical devices

Each HISM check is referenced to a rule in one or several of these standards. Vice versa, the individual standards have rules with overlapping rationales, which apparently is the subset of rules covered by the HISM checks. The common goal of these checks is the robustness of code generated from the model that the checks are run against. Table 4.7 contains an overview of how the checks are organized and what their objectives are.

The Motor Industry Software Reliability Association (MISRA) standards take up a special position in this context. These are sets of rules developed specifically with safety, security, and reliability of embedded system software in mind [29]. While some HISM checks additionally refer to MISRA standards, further secure coding and MISRA compliance checks are available with Embedded Coder and covered separately further below.

Table 4.6.: MAB Checks

Scope	Group	Checks	Verification objective
Naming conventions	–	20	Character usage, lengths, and naming rules for folders, files, subsystems, blocks, signals, parameters and bus names
Simulink guidelines	Configuration parameters	4	Correct setting of configuration parameters with respect to Boolean data, integer rounding, incorrect calculations and model diagnostics
	Diagram appearance	19	Recommended appearance of the model including layout settings, fonts, element sizes, positioning, descriptions, signal connections, signal flow, naming consistency, subsystem structure and the use of prohibited blocks
	Signals	12	Recommended usage of buses, signal names, labels and block parameters with respect to signals as well as type and sample time settings
	Conditional subsystems	6	Recommended block layout and usage with respect to conditional subsystems and their relations and settings
	Operation blocks	16	Correct usage of logical, relational and numerical operation blocks as well as lookup tables, saturation, integrator, delay and type conversion block
	Other blocks	16	Correct setting and positioning of inports and outports, correct usage of switch, data store memory and variant subsystem blocks
MATLAB guidelines	Appearance	2	Identify number of nested statements and presence of a function header in MATLAB function blocks
	Data and operations	3	Recommended usage of shared data, enumerations and inputs/outputs with respect to MATLAB function blocks
	Usage	5	Identify limits on lines of code and levels of called functions, recommended usage of strings, switch-case-statements and comments in MATLAB function blocks

Table 4.7.: HISM Checks

Scope	Group	Checks	Verification objective
Simulink block considerations	Naming conventions	2	Character usage in model file and element names
	Math operations	9	Recommended usage of absolute, remainder, reciprocal, square root, logarithm, product, assignment and gain blocks
	Ports and subsystems	11	Recommended settings for while iterator, for iterator, if and switch case blocks as well as inport and outport specifications
	Signal routing	5	Correct usage of data store memory, merge and signal routing blocks as well as consistent vector and signal indexing
	Logic and bit operations	4	Unambiguous usage of relational operator, logical operator and bit-wise operation blocks
	Lookup table blocks	3	Recommended settings regarding lookup table blocks, tunable parameters and bit-shift operations
Stateflow chart considerations	–		*Out of scope for this thesis*
MATLAB considerations	MATLAB functions	3	Complexity and length of MATLAB functions used in models as well as the use of strong typing at function interfaces
	MATLAB code	8	Code Analyzer checks as well as detection of several undesirable code patterns

Table 4.8.: HISM Checks (continued)

Scope	Group	Checks	Verification objective
Configuration parameter considerations	Solver	3	Simulation time, solver and tasking options required for production code generation
	Math and data types	2	Logic signal and lifespan settings for production code generation
	Diagnostics	16	Enabling error messages for or disabling of configuration parameters related to code robustness
	Hardware implementation	1	Matching target configuration parameters between production hardware and test hardware
	Model referencing	1	Settings related to consistency of model and its references
	Code generation	9	Code generation settings that increase robustness and verifiability of generated code
Requirements considerations			*Out of scope for this thesis*
MISRA C compliance considerations	Modeling style	4	Naming rules compliant with the MISRA C standard
	Block usage	3	Exclusion of blocks not recommended for MISRA C compliance
	Configuration settings	1	Various configuration parameters that are required for MISRA C compliance

Code Generation Checks

MathWorks provides a set of modeling guidelines for models that are intended for code generation for embedded systems with the Embedded Coder toolbox [29]. These guidelines are structured in four categories.

- *Blocks:* Use of certain fixed-point operations, precalculation of signals and absence of redundant blocks with the goal of higher code efficiency
- *Modeling patterns:* Use of certain signal elements and block placements with respect to subsystem for more efficient memory usage
- *Configuration parameters:* Prioritization of code generation objectives for higher code efficiency
- *Component deployment:* Identify settings and modeling techniques that can result in low data transfer efficiency

These guidelines have a rather specific and limited scope in comparison to the standards introduce before. They appear to target code efficiency but do not seem to cover robustness of the generated code. In the Model Advisor, these guidelines are implemented as checks with a `mathworks.codegen` prefix. These are 29 checks that identify inefficient or potentially ambiguous operations, verify hardware settings, identify non-recommended blocks and configuration parameters, identify correct compilation settings, and verify various settings in model elements that are required or recommended for production code generation [29].

The scope of these checks suggests to employ them in conjunction with, but before any standards compliance checks. The rationale would be, that a model needs to be suited and modeled efficiently for code generation, before the compliance with a much more comprehensive standard is verified, even if only partial. Reviewing the associated check results should take this prioritization into consideration.

Secure Coding Checks

For embedded systems, there are several coding guidelines addressing security concerns specifically. They aim at reducing coding constructs that are vulnerable to exploitation in embedded software. Model Advisor checks for the following guidelines are available for Embedded Coder [29].

- The Software Engineering Institute (SEI) Computer Emergency Response Team (CERT) C standard is a set of guidelines for secure coding practices for the C languages. They "are a work in progress and reflect the current thinking of the secure coding community" [30].
- The Common Weakness Enumeration (CWE) is an extensive, community-developed list of software and hardware weakness types that under certain circumstances can become security vulnerabilities in embedded software and hardware [31].
- ISO/IEC TS 19761 is the formal ISO standard for secure coding in C [29].

Additionally, the MISRA standards define a "safe subset" of the C languages that also protects against safety and security vulnerabilities, but also further increases robustness and reliability of embedded systems software [32]. The Model Advisor checks associated with the secure coding standards are grouped with the prefix `mathworks.security` and overlap with the MISRA compliance checks with the prefix `mathworks.misra`.

While code security is not the goal of this thesis, the associated standards generally have the robustness and reliability of embedded software in mind and as such should be part of a verification process with respect to correctness.

It is important to note that compliance with a standard can not be fully evaluated with the model as test object. Running the introduced checks rather "increases the likelihood" of generating code compliant with the standards [28]. For better results, the use of additional verification tools is necessary.

4.2.3. Model Checking in Simulink

Simulink Design Verifier is a model checker that is integrated into the MATLAB/Simulink user interface. It uses a proof system that is based on Stålmarck's proof procedure for propositional logic [17]. Understanding the theoretical details of this procedure is not required in order to operate the tool, but it is helpful in understanding its limitations.

There are three distinct *modes* that the tool offers. They all represent different aspects of model checking in general.

- *Design error detection:* In this mode, the analysis can be configured to find one or several predefined error types in a model or prove their absence. If the analysis is able to find a design error, it provides a *counterexample*, an exemplary signal trace that causes the error [33].
- *Test generation:* Here, the model checker's ability to find counterexamples is used systematically to define new test cases for simulation-based testing that achieve missing coverage. The coverage objective has to be defined and the coverage data recorded prior to the analysis [33].
- *Property proving:* In Simulink, properties are modeled as requirements in the model. For this purpose, the tool offers proof objective and assumption blocks and functions. The properties are then proven or falsified by the analysis [33].

For this thesis, design error detection shall be the focus of evaluation. The central issue with Simulink Design Verifier is the compatibility of the model with the analysis technique. Due to the nature of model checking, a model that is simulated correctly in Simulink is not necessarily suitable for an analysis. For this reason, the other two modes that require more upfront effort are not further pursued here.

The compatibility check is the first step in a Design Verifier analysis. There is a subset of Simulink blocks that is supported, and it is checked whether the analyzed model contains unsupported blocks. Blocks in Simulink's Discrete, Math Operations, Logic Operations, Sinks, and Sources Libraries are supported for the most part. Notable exclusions are the State-space, Sine Wave, Square Root and Signal Editor blocks. Blocks from the Continuous Library are not supported. Limitations exist when using enabled, triggered and variant subsystems as well as for user-defined functions. For MATLAB function blocks, certain operations like calls to external C functions as well as most toolbox functions are not supported. S-Functions or C/C++ code containing for example continuous states, zero-crossing functions and infinite or not representable objects are also not supported [33].

Incompatibilities are handled automatically by the analysis with a technique called *stubbing*. Unsupported blocks are ignored by the analysis and the block output is assumed to be able to take on any value. In some cases, this might lead to an analysis that is still conclusive. Another way to approach incompatibilities is by defining replacement rules for the analysis. These are written in custom MATLAB-files using special syntax. For example, arithmetic operations on a signal with an expected signal range could be replaced with a precalculated lookup table of the results [33].

If a model is compatible, the analysis proceeds to generate a *model representation* with several approximations. Floating-point values in signals or parameters are converted to rational numbers in some cases. This prevents numerical errors from affecting the result of the analysis. In lookup tables, the interpolation is set to linear to increase analysis performance. Further, `while`-loop iterations are reduced if no constant bound can be found, so they always exit [33].

Simulink Design Verifier is run programmatically with the `sldvrun` function. It accepts a model or subsystem name and a design verification options object that specifies the analysis. As shown in the following listing, this `options` object is returned by the `sldvoptions` function and contains all analysis settings as parameters with their default values. These are subsequently modified for the individual analysis. Here, also the replacement rules or coverage objective and data files mentioned earlier would be passed to the respective parameter. The `sldvrun` function returns a status code `status` of the analysis result, the report and data file names `files` created by the analysis and an error or warning information message `msg`.

```
% Define model and create options object
system = 'Controller';
options = sldvoptions;

% Configure analysis
options.Mode = 'DesignErrorDetection';
options.DetectDivisionByZero = 'on';
options.SaveReport = 'on';

% Run analysis
[status, files, msg] = sldvrun(system, options);
```

The following are the detectable design errors explained in more detail, based on the tool's documentation [33].

- *Dead logic:* Model elements remain inactive for an entire simulation.
- *Out of bound array access:* The model tries to access an array element with an invalid index.
- *Division by zero:* The denominator of a division operation becomes zero during a simulation.

- *Integer overflow:* An operation on an integer signal exceeds its representable range.
- *Non-finite and* `NaN` *floating-point values:* A floating-point signal becomes infinite or not representable during a simulation.
- *Subnormal floating point values:* A floating-point signal becomes too small to be accurately represented.
- *Specified minimum and maximum value violations:* The minimum and maximum values on signals and outports throughout a model are exceeded during a simulation.
- *Data store access violations:* A data store memory block receives an unintended sequence of read and write operations during a simulation.
- *Specified block input range violations:* The minimum and maximum values on block input signals in a model are exceeded during a simulation.

Both the plain function implementation as well as the controller FSM are as expected compatible with the analysis. Exemplary cases of dead logic or division by zero are correctly identified as shown in figure 4.7. The associated signal trace is provided too. Both the analysis output as well as the analysis report point to the location of the error, but only at the top level. It appears as though the analysis does not fully consider nested MATLAB functions and as such *is not able to point to the precise location of an error.*

It should further be noted that in spite of the existing approximations, inconclusive analyses are already produced in the context of the small-scale demonstration example. As noted in section 2.4, model checking is not suited for data-intensive applications, which here results in frequent analysis timeouts. For use in a project context, it should prior be evaluated if it is feasible to specify minimum and maximum values at *every* root-level inport block of the analyzed model.

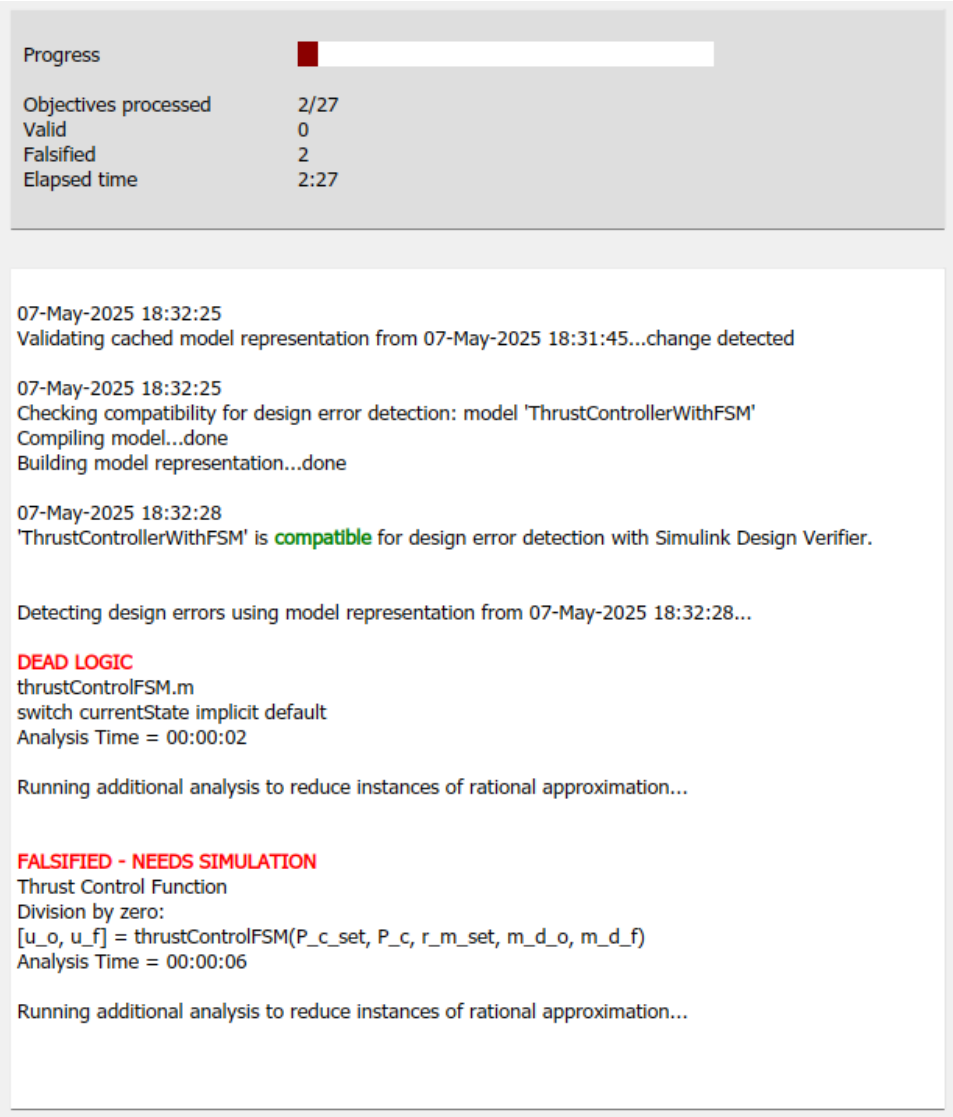

Figure 4.7.: Design Verifier Analysis

4.2.4. Testing in Simulink

Simulink Test is MathWorks' toolset for simulation-based testing. It is of interest here because it conveniently allows to isolate and test any part of a Simulink model with a *test harness*. A clear graphical and a comprehensive programmatic interface help with test authoring, test execution and test management in alignment with the fundamentals of software testing discussed in chapter 2 [34].

Simulink Test exists next to the MATLAB unit testing framework ("MATLAB Unit Test"), from which it can use several features but should be distinguished. When large parts of a model's functionality are implemented in MATLAB code, it is prudent to develop tests in the related testing framework. It provides several tools to author, execute, evaluate and automate tests of MATLAB code. Tests can be written as simple scripts, as functions or as classes that inherit from the `matlab.unittest.TestCase` *superclass* to leverage the full capabilities of the framework. Tests are executed by *test runners*, the fundamental API of MATLAB Unit Test, which is supplemented by *plugins* that enable individual evaluation and reporting features [6].

A good testing strategy should define exactly when which testing tool shall be used. A clear use case with added value for Simulink Test would be integration testing – in conjunction with unit testing implemented with MATLAB Unit Test. Two testing methods are of interest here and are demonstrated further below.

Preparation

Independent of the used testing method, there are two prerequisites to be taken care of. First is creating a test harness, which is realized in Simulink via the subsystem context menu. As subsystem blocks can contain arbitrary levels of further subsystems, this essentially means that tests can be authored at any level in the model with little effort. The use of test harnesses is not unique to Simulink, but can refer to any part of a program that links a testing framework to a component under test and enables the execution and evaluation of a test suite. Simulink Test however reduces the effort in creating and maintaining them when compared to conventional programming languages. The test harness for the Thrust Controller subsystem is shown in figure 4.8.

Test harnesses are per default saved in the same file as the associated Simulink model with changes in the model being updated to the harness when opening. It should be noted that this proved unreliable in cases where changes in signal names needed to be updated in harnesses of model files that were copied before. A harness can alternatively be saved externally as an own model file and linked with a `xml` file that is created by Simulink Test. This approach worked more reliably in the context of the demonstration example and could be adopted in larger projects for better traceability.

Next, the inputs to the component under test have to be defined. Using Simulink's Signal Editor block, inputs can be authored manually. Additionally, Simulink Test's Test Sequence

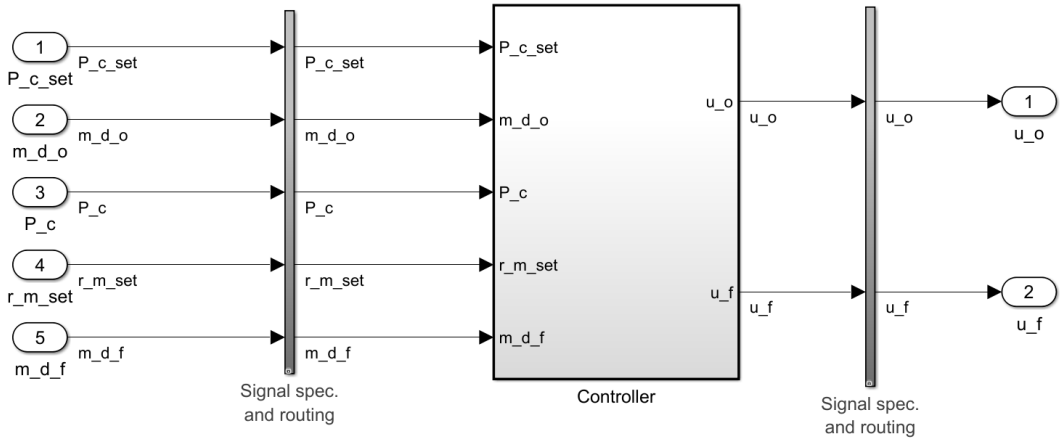

Figure 4.8.: Controller Test Harness

block allows to author sequential tests that react to a simulation with steps or transitions written with MATLAB. For external inputs, there are three additional options [34].

- MATLAB scripts: Input parameters are defined in scripts using the MATLAB language. At the time of writing, this is the least documented approach and therefore not further pursued here.
- Excel spreadsheets: Input parameter values are defined for each simulation time step in an Excel spreadsheet. At the time of writing, this is the most comprehensively documented approach. Templates with the required layout can be generated when authoring a test case. As logged simulation data can be exported to Excel spreadsheets with a similar layout, this approach promises to be the most versatile.
- MATLAB data files: Input parameters are stored in MATLAB's binary `mat` file format. Manual editing of parameter values is possible with the Signal Editor but less convenient than in a conventional spreadsheet format. As `.mat` is the default export file format for logged simulation data, this approach however promises to be best suited for data-driven simulation models such as those investigated here.

The following demonstration makes use of MATLAB data files, which contain the logged data from the already developed simulations. As signal logging via the Simulink user interface proved to be unreliable when mapping to a test harness, logging is done with a short script.

The `sim` function runs a simulation and returns a `Simulink.SimulationOutput` object, which contains all data associated to the simulation. Logged data is stored in the `logsout` property and can be saved to a `mat` file from there.

```
% Run the simulation and log the signals enabled for logging
simOut = sim('DiscreteThrustControl');

% Retrieve logged data from simulation output
logs = simOut.get('logsout');

% Save dataset to MAT file
save('inputData.mat', 'logs');
```

Baseline Testing

In baseline testing, simulation output data is compared to *baseline data*. This data is obtained by simulating a component with prior created test harness and input data. The test itself then verifies whether the component under test produces the same output within a defined margin [34].

The Test Manager is the user interface for Simulink Test. As graphical interface its test case templates guide through the test authoring phase, but it offers all required functionality programmatically too. The baseline test case is shown in figure 4.9. After setting the Simulink model, the associated harness model, the input data as well as its mapping, the baseline data can be captured from here. If baseline data is already present, it is set here. Either way, the output tolerance in absolute and/or relative values as well as the leading and lagging tolerance with respect to time must be defined here for a functioning test case. Coverage settings are set at the test file level but can be changed here for the individual test case.

This concludes the required settings for a simple baseline test case. It can now be executed after new changes to the Controller subsystem. If not specified to overwrite, model configuration parameters such as the simulation time should be carried over from the simulation the test harness is associated to. This however was found to work unreliably too, so in doubt the applicable overrides should be used.

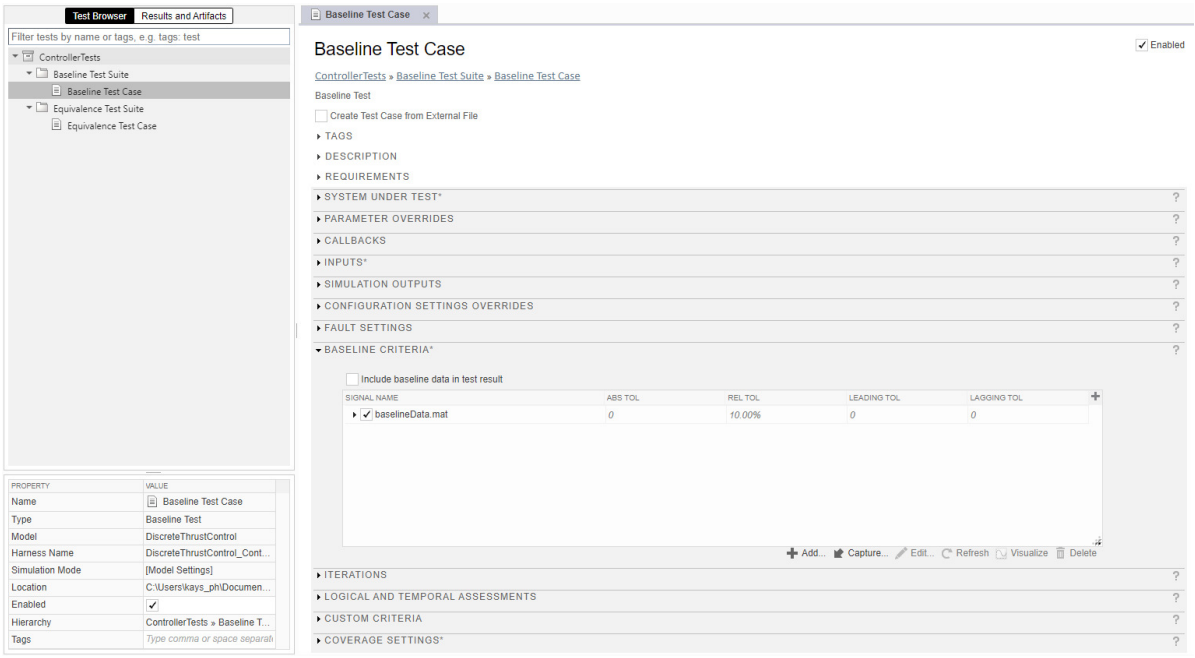

Figure 4.9.: Baseline Test Case in the Simulink Test Manager

The programmatic execution of the test file is demonstrated in the following listing. The `sltest.testmanager` namespace contains all functions related to test execution and reporting, where this is just a minimal example. Apart from that, there are more functions related to test authoring, test harnesses, test sequences and assessments [34]. Such comprehensive programmatic support might be helpful in scaling a testing strategy over many components, different variants, versions etc.

```
% Open the test file
testFile = 'TestFile.mldatx';
sltest.testmanager.load(testFile);

% Run all tests
results = sltest.testmanager.run;

% Generate report from results data
sltest.testmanager.report(results, 'TestReport.pdf');

% View simulation output data
sltest.testmanager.view;
```

Equivalence Testing

Equivalence or back-to-back testing verifies whether two simulations produce the same output within a defined tolerance [34]. The associated test case is shown in figure 4.10. The same input data is now used on two harnesses of two different simulations. Here, the FSM implementation of the controller subsystem is tested for equivalence against the plain function implementation. Instead of baseline data, now equivalence criteria and the associated tolerances must be defined.

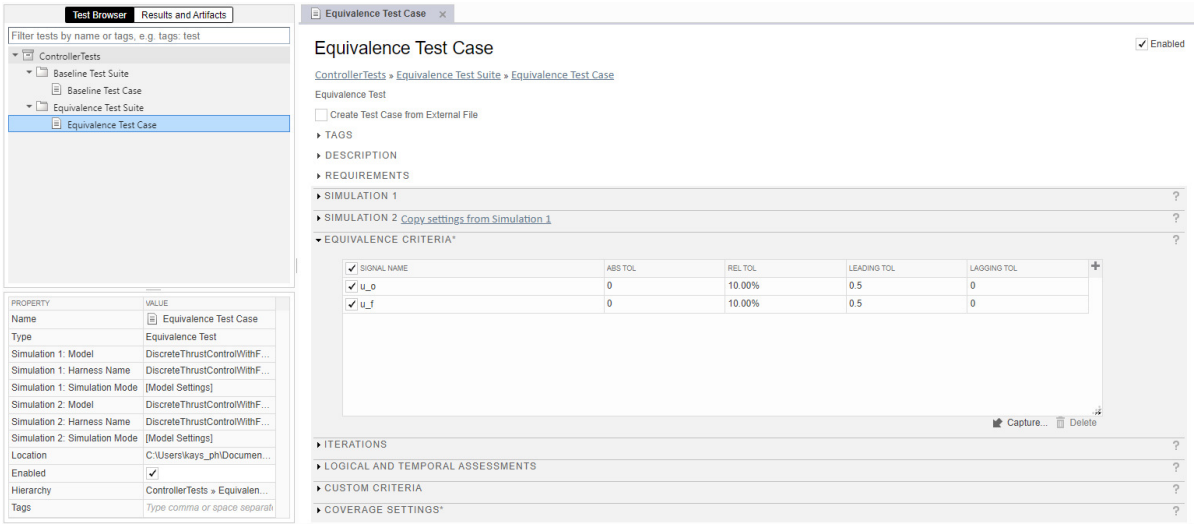

Figure 4.10.: Equivalence Test Case in the Simulink Test Manager

The test execution is identical to before. For either test case, the results are displayed in the test Manager with a visualization of the specified tolerances. The results for u_o in the equivalence test case are shown as an example in figure 4.11. The tolerance was defined to account for the fact that the FSM implementation commands a fixed output from the beginnig of the test case. The transition to the state “Running” however is triggered too late for the tolerance and as such the test fails. Notable is the associated detailed coverage results collection shown in figure 4.12.

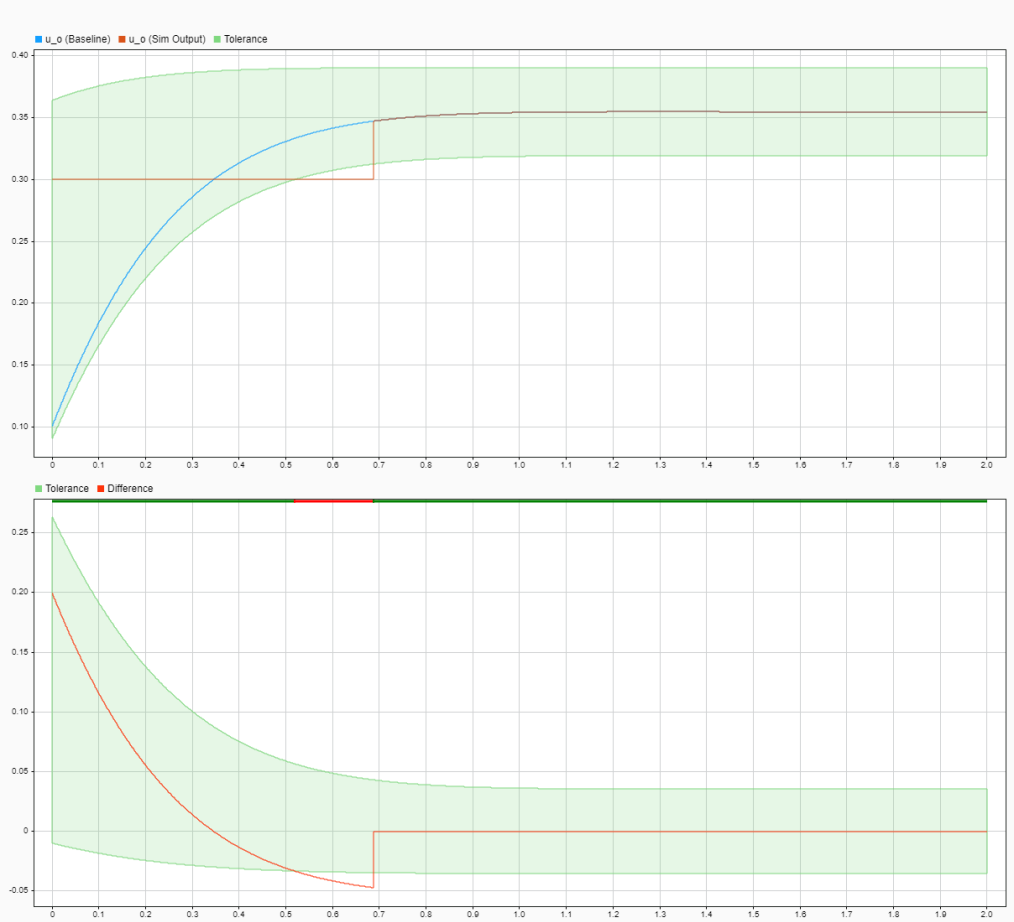

Figure 4.11.: Equivalence Test Results for u_o

ANALYZED MODEL	REPORT	COMPLEXITY	DECISION	EXECUTION
computeControl		3	100%	--
DiscreteThrustControl/Controller		1	--	100%
DiscreteThrustControlWithFSM/Controller		3	100%	100%
DiscreteThrustControlWithFunction/Controller		3	100%	100%
remainClosed		2	0%	--
remainPressurizing		2	100%	--
remainRunning		2	100%	--
thrustControlFSM		10	73%	--
thrustControlFunction		4	100%	--
updateClosedToPressurizing		2	100%	--
updatePressurizingToClosed		2	0%	--
updatePressurizingToRunning		2	100%	--
updateRunningToClosed		2	0%	--

Figure 4.12.: Equivalence Test Coverage Results

SIL Testing

Apart from model-in-the-loop (MIL) testing, equivalence testing with Simulink Test also supports software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing. In SIL testing, the output of code generated from the component under test is verified. In PIL testing, this code is run on the target processor after a previous connectivity configuration. These tests are authored by specifying the associated verification mode in the test harness properties. This essentially presents an extension of the SIL and hardware-in-the-loop (HIL) simulation modes that are already part of Embedded Coders functionality. Lastly, also HIL testing is supported. Here the code is executed on the standalone hardware including its own input/output connectivity, for which the product Simulink Real-Time is required [34].

4.2.5. Code Generation

While the details of automated code generation are beyond the scope of this thesis, it must nevertheless be part of the verification process in a basic form. There are three related code generation products of interest here. MATLAB Coder provides the ability to generate C and C++ code from MATLAB code, while Simulink Coder provides the same capability for Simulink models. Apart from code optimization and targeting capabilities, Embedded Coder also contains additional verification functionality [29].

Code generation from a Simulink model can be understood as a process of four to five steps, of which not all require interaction with the user. A visualization of the process with information from references [35] is shown in figure 4.13.

1. The model is configured for code generation through its configuration parameters. Parameters for example relating to the language specification, compilation toolchain and optimization can be set here.

2. The coder software is invoked using its `slbuild` function to generate C++ code. This concludes the interaction with the user and internally triggers the next step.

3. The coder software generates a model description file. This file contains a description of the model's execution semantics in a high-level language.

4. The software's Target Language Compiler converts this intermediate description into the target-specific code. For this it uses its own function library as well as different target files. The system target file for example defines the required compilation toolchain and software settings associated with the intended execution environment.

The last step would be the basis for generating an actual executable if that were required. The Target Language Compiler creates a *Makefile* based on the target files and an already existing template. Makefiles contain instructions for the build process of software, i.e. how source code has to be compiled and linked to form a correct executable (cf. section 2.4). The coder software can be configured to automatically execute this file after the Target Language Compiler has finished its tasks [35].

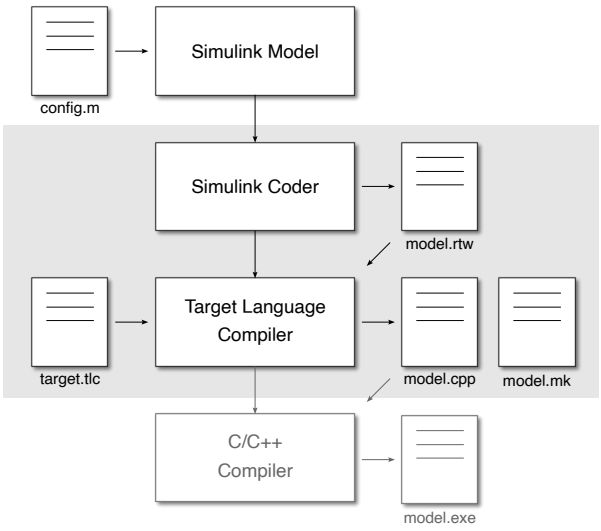

Figure 4.13.: The Code Generation Process (modified from [35])

The configuration parameters of a model can be saved to and restored from a MATLAB data file. This allows for flexibility and portability in the configuration, but makes traceability of individual parameter settings cumbersome. Therefore, for this thesis, a provided configuration set is translated into instructions to set the configuration parameters programmatically. This is done wit MATLAB's `set_param` and `get_param` functions, that set and obtain the configuration parameters of a model. To obtain only the parameters of interest, the configuration set is compared to a default set. It is important to set the system target file from generic real-time (GRT) ro embedded real-time (ERT), as this changes the availability of certain parameters.

```
% Load the specified configuration set
data = load('CodeGen.mat');
configSet = data.('CodeGen_cfg');

% Create default configuration set for comparison
defaultConfigSet = Simulink.ConfigSet;
set_param(defaultConfigSet, 'SystemTargetFile', 'ert.tlc');

% Retrieve all parameter names from loaded configuration set
loadedParameters = get_param(configSet, 'ObjectParameters');
paramNames = fieldnames(loadedParameters);

% Retrieve parameters from default configuration set
defaultParameters = get_param(defaultConfigSet, 'ObjectParameters');
nonDefaultParams = {};
```

The obtained parameters can then be written explicitly in a configuration script or function. Even if already set in the model, this step ensures traceability in the code generation process. The code generation itself is triggered with the `slbuild` function.

```
% Load model
load_system('Controller');

% Set parameters for C++ code generation
set_param(model, 'SystemTargetFile', 'ert.tlc');    % System target file
set_param(model, 'TargetLang', 'C++');              % Code generation
    language
set_param(model, 'GenCodeOnly', 'on');              % Do not execute makefile

% Generate code
slbuild(model);
```

4.2.6. Static Analysis of Generated Code

The Polyspace product family serves as the static analysis toolset for this thesis. It is marketed by MathWorks but technically remains separate from the MATLAB/Simulink product environment. There are two products of interest here. Polyspace Bug Finder performs the

bulk of both syntactic and semantic static analysis with hundreds of checks for a wide scope of issues. Additionally, compliance of the analyzed code to industry coding standards can be checked [36]. Polyspace Code Prover complements this with sound semantic analysis using abstract interpretation. The analysis thus has a less wide scope but achieves a higher degree of certainty [37]. Both products can analyze code written in C, C++ and Ada, while C++ shall be the focus here. Other products like Polyspace Access are out of scope for this thesis. Overall, Polyspace is found to offer the following options to be run.

- *Graphical user interface:* Polyspace analyses and results are organized in projects. These can be configured from the graphical user interface. Scope of the analysis, files to be included and any customizations can be defined here. Analysis results are stored in a proprietary file format per default. For this thesis, the graphical user interface is mainly useful in reviewing results for confirmation.
- *Integration:* To simplify the workflow for verification of code generated from models, Polyspace can be integrated with MATLAB/Simulink. There, an API allows to configure and run analyses programmatically. As this represents the use case investigated here, this approach is further investigated below.
- *Command line:* Polyspace can also be fully configured and run programmatically from the command line. This approach should be considered in case of issues with the aforementioned approach.

The integration is performed with a dedicated setup function. This function accepts installation directories differing from the default and further options. Integration across release versions is possible but results in limited functionality [36].

```
% Integrate Polyspace with Matlab/Simulink
polyspacesetup('install');
```

After successful integration, there appear to be three approaches to run a Polyspace analysis from MATLAB programmatically that exist independently of each other. The syntactically easiest is using the dedicated `pslinkrun` function. It accepts a model or system name and a configuration object created by the `pslinkoptions` function. This function in turn accepts one of three Simulink object types, which defines the configuration options of the analysis. This way, a default configuration for either a generic code generator, a Simulink model or a S-function can be obtained. The options object has around 20 properties relating to the analysis itself, results export, additional files, data ranges and specifics to the code generator [36].

These represent only a simplified subset of the full configuration potential of a Polyspace analysis. An analysis is run as shown below assuming that code has been generated prior to the analysis.

```
% Load Simulink model
model = 'Controller';
load_system(model);

% Create configuration object
options = pslinkoptions(model);

% Configure analysis
options.VerificationMode = 'BugFinder';
options.VerificationSettings = 'PrjConfig';

% Run analysis
[polyspaceFolder, resultsFolder] = pslinkrun(model,options);
```

Note that the `VerificationSettings` property for example does not allow for the inclusion of C++ specific standards in the analysis. Apart from the default `'PrjConfig'` option, there are only options to include standards specific to C. This approach therefore is insufficient for the verification purposes of this thesis.

A more flexible approach is using the dedicated `polyspaceBugFinder` and `polyspaceCodeProver` functions. These are more versatile: With the proprietary `.psprj` project file as input argument they open the project, and with the analysis results files `.psbf` and `.pscp` respectively they open the results in Polyspace. With an options object as an input argument they run the respective analysis. This options object is an instance of the `polyspace.BugFinderOptions` class and can be configured similarly to the approach before [36]. In the following example, the sources, include folders and results directory are set manually.

```
% Create options object
options = polyspace.BugFinderOptions;

% Set source file, include folders and results directory manually
options.Sources = {fullfile(pwd, 'sources', 'source.cpp')};
options.EnvironmentSettings.IncludeFolders = {fullfile(pwd, 'sources')};
options.ResultsDir = fullfile(pwd, 'results');

```

```
% Run specified analysis
polyspaceBugFinder(options);

% Open analysis results
polyspaceBugFinder('-results-dir', options.ResultsDir);
```

At the time of writing there is insufficient documentation on the `polyspace.BugFinderOptions` class to fully understand its use. A better documented approach is using the `polyspace.Project` class. Instances of this class have a `Configuration` property to customize the analysis, a `run` method to execute the analysis and a `Results` property that contains the results as a `polyspace.BugFinderResults` or `polyspace.CodeProverResults` object respectively [36].

The `polyspace.Project.Configuration` property itself has 133 properties that in part apply to both or either of the analysis types. Reviewing the documentation, they seem to be structured using intermediate properties to group settings that are related [36].

- The general configuration properties contain settings related to the analysis environment, constraints on variables, report generation, multitasking and target compiler information. They are grouped in the intermediate properties `EnvironmentSettings`, `Multitasking`, `TargetCompiler` etc.
- The configuration properties for a Bug Finder analysis mainly relate to the scope of checks, coding standards to be included and metrics to collect. They are grouped in the intermediate properties `BugFinderAnalysis` and `CodingRulesCodeMetrics`.
- The configuration properties for a Code Prover analysis specify various verification assumptions and precision aspects of the analysis and are grouped in intermediate properties such as `ChecksAssumption`, `CodeProverVerification`, `Precision` etc.

When the analyzed code was generated from a Simulink model, it is possible to automate the configuration using the `polyspace.ModelLinkOptions` class. An instance of this class can be associated with the model that the code is generated from. Thereby, the Simulink *model configuration parameters* are used to determine a subset of the Polyspace *configuration object properties* [36]. This for example sets the target compiler, source and include files or whether a `main` function is included in the code. Remaining configuration properties take their default values and can be modified afterwards. The model must be loaded in Simulink and code must have been generated from it. The association command then uses Embedded Coder and enables the generation of a `linksData.xml` file. The presence of this file in the

result directory allows to trace locations of issues in the code back to the corresponding location in the model via hyperlinks in the Polyspace user interface.

Alternatively, the paths to the code can be defined manually analogous to the last listing above. This results in missing code-to-model traceability when the code was generated from a model and no `linksData.xml` file is present in the result directory. This way however the same programmatic approach can be extended to the analysis of handwritten code.

The correct configuration can and should be validated using the `.log` file that is created with every analysis, where every analysis setting is listed explicitly. This is important as in some cases Code Prover analyses were found to produce inconsistent results, which could be traced to incorrect manual configurations. The automated configuration promises to be less error-prone and is therefore preferred. The following listing includes a Polyspace analysis run with the `polyspace.Project` class.

```
% Create Polyspace project
project = polyspace.Project;

% Create configuration object associated with model
configuration = polyspace.ModelLinkOptions(model);

% Associate project Configuration property with this configuration object
project.Configuration = configuration;

% Specify some additional settings
project.Configuration.BugFinderAnalysis.ChecksUsingSystemInputValues = true
project.Configuration.BugFinderAnalysis.SystemInputsFrom = 'all';

% Run analysis
status = run(project, 'bugFinder');

% Obtain results and summary
results = project.Results;
summary = getSummary(results, 'defects');
```

A full Polyspace analysis covers four verification aspects, which are explained in more detail below.

Defects

What MathWorks refers to as *defect checking* is executed by Polyspace Bug Finder and represents the part of the analysis with the widest scope. In the terminology introduced in chapter 2, it includes both syntactic and non-sound semantic static analysis of C/C++/Ada code. As such, the checks are not able to track the control flow of a program as well as sound semantic analysis methods might be. Due to several assumptions, the analysis is still surprisingly expressive.

Depending on how global variables are defined, the analysis can conservatively assume an initialization according to standards and language definitions. For volatile variables, i.e. variables that might change at any time without an explicit write operation, similar assumptions are made. A priori, there are also no assumptions for the values of inputs to functions. Errors caused by operations with unbounded variables per default can not be caught by a Bug Finder analysis. To be flagged as a defect, a variable used in an faulty operation needs to be bound by an `assertion` or `if` statement in the program [36].

These assumptions are intended to lower the rate of false positive defect findings but conversely might lead to some false negatives. For several checks, Bug Finder therefore offers the option to run a more exhaustive analysis where all values of variables or function inputs are considered. The listing above includes the necessary commands for this as an example. MathWorks makes no claim about the reliability of these extended analyses other than that they are still not as exhaustive as a sound formal analysis [36].

The preconfigured defect checks can be divided in 15 groups with different verification objectives. A review of those has been documented in table 4.9. Even given the limitations, it becomes apparent how just the defect analysis alone can already provide a substantial degree of confidence in software quality.

The aforementioned `Configuration` property of the `polyspace.Project` class has an intermediate `BugFinderAnalysis` property that applies to running Bug Finder analyses. Its `CheckersPreset` property accepts a `'default'` setting that applies a predefined set of checks, while `'all'` applies all available defect checks and `'custom'` enables to configure the checks to be run. In that case, a defect options object needs to be passed to the the `CheckersList` property. This object is instantiated with the `polyspace.DefectsOptions` class, in which all defects are listed as Boolean properties and need to be set to `true` to be enabled [36]. For an

analysis with custom checks, the following code would need to be added to the analysis run with the `polyspace.Project` class.

```
% Create defects options object
defects = polyspace.DefectsOptions;

% Enable some arbitrary checks
defects.INT_ZERO_DIV = true;
defects.INT_OVFL = true;
defects.BITWISE_NEG = true;

% Extend configuration
project.Configuration.BugFinderAnalysis.CheckersPreset = 'custom';
project.Configuration.BugFinderAnalysis.CheckersList = defects;
```

Coding Standards

The second verification aspect of Polyspace is the compliance of the analyzed code with internationally recognized coding standards, which is also covered by Polyspace Bug Finder. While the software includes checks for a variety of standards, the remarks here focus on those applicable to C++.

C++ was created in 1979 as "C with Classes" by Bjarne Stroustrup to provide improved program organization capabilities yet keep the efficiency and flexibility of the language C. After its first commercial release in 1985 it was officially standardized in 1998 with ISO/IEC 14882:1998. It received a minor revision with ISO/IEC 14882:2003. The thus specified language is referred to as C++03 [38].

In 2005, the MISRA C++ Working Group was established after MISRA C had become the predominant coding standard for safety-critical system programming with C. Its objective was to formulate a single, generic set of guidelines for the use of C++ in safety-critical systems that are understandable to the majority of programmers. Various existing guidelines were gathered, reviewed and extended to produce a subset of the programming language that is safe to use in critical applications. The resulting standard MISRA C++:2008 is applicable specifically to C++03 [39].

In 2011, C++ received a major modernization, followed by another minor revision in 2014. Facing these substantial changes as well as a need to use them in safety-critical systems,

Table 4.9.: Polyspace Defect Checks

Group	Checks	Verification objective
Numerical defects	21	Find faults in numerical operations on integer and floating point data types such as overflow, division by zero, precision loss or negative shift operations
Static memory	17	Find faults relating to memory allocation at compile-time, such as out of bound access, null pointer or buffer overflow
Dynamic memory	8	Find faults relating to memory allocation at run-time, such as invalid or mismatched allocations, deallocations and deletions
Data flow	14	Find faults relating to the flow of information, such as dead code, infinite loops, uncalled functions and other unused code constructs
Resource management	5	Find faults related to file handling, such as mismatched read and write or open and close operations
Programming errors	75	Find errors resulting from wrong syntax. They contain logical errors, incorrect assertion and error handling, type and declaration mismatches, string and character handling errors and data handling issues. Defects are classified in *high*, *medium* and *low impact* categories.
Object-oriented defects	17	Find errors and unsafe operations resulting from incorrect class usage, inheritance, encapsulation and related assignments
Exceptions	6	Find issues and errors in exception handling
Concurrency	24	Find faults related to multitasking, such as missing or incorrect usage of data access synchronization
Security	38	Find security weaknesses in file access, privilege handling, standard function usage, database queries and more
Cryptography	39	Find weaknesses in the use of cryptographic routines
Tainted data	17	Find instances of unvalidated data usage from unsecure sources such as external inputs and volatile objects
Good practice	37	Find issues and faults that might indicate logical errors, vulnerability or maintainability issues. These issues relate to readability, hard-coding, duplications, macros, bad memory management or the use of forbidden constructs
Performance	46	Find issues that negatively impact performance such as inadvertent operations or inefficient function or variable usage

Automotive Open System Architecture (AUTOSAR) updated the MISRA C++:2008 standard. AUTOSAR C++14 specifies obsolete rules, minor improvents on existing rules as well as additional rules. It applies to both C++11 and C++14 by detailing which features introduced in either version may be used or shall not be used [40].

Polyspace Bug Finder includes checks for the majority of rules specified by these standards. They are activated and configured via the intermediate `CodingRulesCodeMetrics` property of the `polyspace.Project.Configuration` property. For the example of MISRA C++:2008, the property `EnableMisraCpp` activates the checks while the property `MisraCppSubset` specifies the scope. Apart from the default `'required-rules'`, `'all-rules'` as well as other standard-specific subsets can be activated. Alternatively, the property accepts a coding rules options object instantiated with the `polyspace.CodingRulesOptions` class, which allows to create a custom list of coding rules [36]. To customize coding standard checks, the following code would need to be included in an analysis run with the `polyspace.Project` class. Note that `EnableCheckersSelectionByFile` has to be enabled, since the analysis internally uses an XML file to enable the coding rule checkers.

```
% Create coding rules options object
rules = polyspace.CodingRulesOptions('misraCpp');

% Disable some arbitrary check
rules.Section_1_General.rule_1_0_1 = false;

% Extend configuration
project.Configuration.CodingRulesCodeMetrics.EnableMisraCpp = true;
project.Configuration.CodingRulesCodeMetrics.MisraCppSubset = rules
project.Configuration.CodingRulesCodeMetrics.EnableCheckersSelectionByFile =
    true;
```

Apart from the two introduced standards, Bug Finder also includes the Joint Strike Fighter Air Vehicle (JSF AV) C++ as well as the SEI CERT C++ standards. With respect to C, there are several further standards included, which however all fall beyond the scope of this thesis [36].

It should be noted that C++17 received further refinements and simplifications with ISO/IEC 14882:2017. To account for these changes as well as the increased use of automatic tools in development and verification, MISRA has released the MISRA C++:2023 standard. It incorporates the AUTOSAR guidelines and provides better decidability for its rules, specifically so

that static analysis can achieve greater coverage [41]. The guideline is included in Polyspace releases 2024a and later [36].

Code Metrics

Lastly, Polyspace Bug Finder is able to collect statistical data about the analyzed program [36]. These are trifold:

- *Project metrics* relate to size and number of function calls in a project.
- *File metrics* relate to comment density and interdependency of functions in a file.
- *Function metrics* relate to comment density, complexity and used variables.

These metrics alone do not present decidable criteria for the evaluation of software quality. However, the Hersteller Initiative Software (HIS) has developed recommended upper levels for these metrics which can be used as criteria for analysis checks. Metrics collection is activated with the `CodeMetrics` and threshold checking with the `Guidelines` properties within the `Configuration.CodingRulesCodeMetrics` property. Alternatively, custom thresholds are provided via an XML file [36].

Run-Time Errors

What MathWorks refers to as *run-time error detection* corresponds to sound semantic analysis executed by Polyspace Code Prover using abstract interpretation. In principle, the tool is operated similar to Bug Finder, with a number of predefined checks that are evaluated autonomously. However, as the analysis method generates a much more elaborate representation of the semantics of the analyzed program, it is not surprising that there are differences in the details of configuration.

Generally, the analysis is set up within the `CodeProverVerification` intermediate property of the `polyspace.Project.Configuration` property. For example, the classes and methods that are of interest for the analysis can be specified and characterized more closely. Also, the tool needs to be instructed whether it shall use a `main` function within the analyzed program, ignore it, or generate one if none is provided. In case of the latter, further customization with respect to variable initialization and function calls is possible. The checks and associated assumptions are controlled individually via the `ChecksAssumption`

Table 4.10.: Polyspace Run-Time Error Checks

Group	Checks	Verification objective
Data flow	8	Find errors relating to the flow of information such as uncalled or unreachable functions and code or uninitialized pointers or variables
Numerical errors	5	Find errors in arithmetic operations such as division by zero, overflow, subnormal results and invalid shift operations
Static memory	3	Find errors relating to memory allocation at compile-time such as out of bounds array access
Control flow	2	Find control flow errors such as non-terminating loops
C++ specific errors	5	Find errors related to invalid C++ specific operations, incorrect object oriented programming or uncaught exceptions
Further checks	8	Find errors specific to C/C++ standard libraries and AUTOSAR libraries

property. The Code Prover analysis can be further customized using the `Precision`, `Scaling` and `VerificationAssumption` properties [37].

While these properties all have default values, they are just as important as the general analysis settings with respect to environment, target compiler etc. Setting up the analysis requires good knowledge about the program that is analyzed. Other properties are only described vaguely by MathWorks as e.g. "certain verification approximations" or generic "precision level" [37]. Their impact would rather need to be evaluated experimentally. Within the scope of this thesis, it is therefore preferred to obtain the analysis settings from the model configuration parameters using the `polyspace.ModelLinkOptions` class introduced above.

An overview of the available checks in Polyspace Code Prover is presented in table 4.10. They have a more narrow scope than the sets of checks introduced so far, but as described in section 2.4 correspond to a much more rigorous analysis.

4.3. Automation

It has been shown so far that there already exists extensive support for the programmatic use of MathWorks verification software. This is a main prerequisite for effectively automating the introduced tools. Another cornerstone is setting up the remote repository to support the

automated execution of tasks. Here, this is done by first setting up a GitLab runner that is available to the example project and then configuring a CI/CD pipeline for this project that contains the desired verification steps.

For the evaluation purposes here, it is sufficient to set up a local computer as runner using the `shell` executor, by which MATLAB is executed in batch mode. The general idea of file-based results exporting and evaluation for CI that is used hereafter has been taken from an example on Simulink integration with the CI/CD software Jenkins [42].

4.3.1. Code Analyzer

Instead of solely relying on the command line output returned by the `checkcode` function, a better evaluation of Code Analyzer results in the context of a GitLab pipeline would be preferable. GitLab natively supports the processing of static analysis results with its Code Quality feature. For that, the results must be provided in a JavaScript Object Notation (JSON) file formatted according to a variation of the standardized Code Climate report format. The file format JSON is typically used for the transfer of structured data, and Code Climate requires a single JSON array in which every object corresponds to an issue found in the code [11]. The general structure is shown in the listing below.

```
{
  "description": "This is a description of a check.",
  "check_name": "Name of check",
  "severity": "minor",
  "location": {
    "path": "directory/file.m",
    "lines": {
      "begin": 42
    }
  }
}
```

The Code Analyzer does not provide exporting capabilities, but the creation of a file in this format can be accomplished fully within MATLAB. A structure array that represents this format can be created in MATLAB and populated with the analysis results provided by `checkcode`. MATLAB's `jsonencode` can be used to write the structure array to a JSON file, which is then made available to GitLab as an artifact.

After writing the file, the pipeline could terminate as failed in case Code Analyzer issues have been found. This is easily implemented, as GitLab automatically evaluates exit codes returned by applications run in a pipeline. For exit codes other than `0`, the job in which it was returned per default terminates as failed [11]. MATLAB can deliberately be terminated with an exit code using the `quit` or `exit` functions. This terminates the session, which then is evaluated by GitLab and terminates the associated job as failed.

4.3.2. Model Advisor

Better capabilities to evaluate the Model Advisor check results would be desirable too. To that end, the display of check results in the GitLab graphical user interface (GUI) and the deliberate termination of a pipeline job in case of a failed check are introduced.

GitLab natively has the ability to process test results that are specified in Extensible Markup Language (XML) files following the JUnit format specification. JUnit is the testing framework for the programming language Java, but its specifications meanwhile have been adopted more broadly [43]. As the Model Advisor check results are already obtained as `checkResults` object inside MATLAB, it is convenient to further process them there. MATLAB provides an interface for operating on `xml` files with the `com.mathworks.xml.XMLUtils` class. It grants access to the programming language Java's API for `xml` processing [6]. Part of this in turn is the Document Object Model (DOM) API, which provides the basis for all document operations in this context. A document is represented as a tree structure, where the document object itself is the root and associated objects are nodes. Foundational objects are *elements*, which in turn might have *attributes*. The API provides the required methods to manipulate all of those objects [44].

In practice, this means that a Java XML document object is created with the `createDocument` method. The root element of this document is referenced to a variable, which allows to create and append further elements using the `createElement` and `appendChild` methods. With further operations and associated methods such as the setting of attributes, stepwise the document is created. The document object is then written to an actual XML file with MATLAB's `xmlwrite` function. The following listing contains a simple example.

```
1 document = com.mathworks.xml.XMLUtils.createDocument('root');
2 root = document.getDocumentElement;
3 child = document.createElement('child');
4 child.appendChild(document.createTextNode('text'));
```

```
root.appendChild(child);
xmlwrite(document)
```

Using this class, the challenge now is to write the content of the `checkResult` object obtained from a Model Advisor run into the JUnit XML structure [43]. It is shown in its very basic form in the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite name="TestSuite" tests="2" failures="1" errors="0">
    <testcase name="Test Case 1"/>
    <testcase name="Test Case 2">
        <failure message="Test failed"/>
    </testcase>
</testsuite>
```

After writing the file, the pipeline should also terminate as failed if a Model Advisor check fails. This can be implemented with the same approach as discussed in section 4.3.1.

4.3.3. Simulink Test

To make Simulink Test result output compatible with CI systems, test files can be run with MATLAB Unit Test. The implementation therefore resembles the approach described in section 4.2: From a Simulink Test test file, a test suite and a test runner are created, which is then customized with the required plugins. These can originate from either Simulink Test with the `sltest.plugins` or MATLAB Unit Test with the `matlab.unittest` namespace. The following listing is a minimal example and includes and adds the `TestManagerResultsPlugin` which is required to make Test Manager results available to MATLAB Unit Test.

```
% Import necessary classes
import matlab.unittest.TestRunner
import matlab.unittest.TestSuite
import sltest.plugins.TestManagerResultsPlugin

% Create test suite and test runner
suite = testsuite('SimulinkTest.mldatx');
runner = TestRunner.withNoPlugins;

```

```
% Add Test Manager Results plugin
tmrPlugin = TestManagerResultsPlugin;
runner.addPlugin(tmrPlugin);

% Run test suite
results = runner.run(suite);
```

The goal then is to find a way that provides meaningful results in CI systems. This should entail test reports that can be processed by GitLab such as JUnit XML files as well as human-readable formats. Further, the achieved test coverage should at least be collected and ideally also be exported and displayed as a result.

4.3.4. Polyspace

Due to limitations mainly in the reporting features of MATLAB's `polyspace.Project` class, it was decided to execute Polyspace in the pipeline using shell commands. At the cost of having to find a solution for interfacing with the MATLAB/Simulink environment, access to the full configuration and reporting capabilites of Polyspace is provided this way. For code generated from a Simulink model, MATLAB's `polyspacePackNGo` function mostly automates this process. The function receives a model name as input argument and automatically extracts a Polyspace analysis configuration from it [10]. This requires the model configuration parameters to be set accordingly, which however already was a prerequisite for using the `slbuild` function. If `slbuild` was configured to pack the generated code in an archive file, `polyspacePackNGo` adds the analysis configuration to that same archive. When further configuration e.g. related to the analysis assumptions is required, `polyspacePackNGo` accepts the Polyspace options object introduced in section 4.2.6 as a second input argument [36].

Due to the many configuration options, Polyspace accepts an options file, a text file that substitutes providing the same information in an otherwise possibly very lengthy shell command. `polyspacePackNGo` saves the configuration in such an options file, which means that a Polyspace analysis can be executed without further adjustment in the environment that the generated code is deployed to. In practical terms, this requires to append the code generation commands from section 4.2.5 as shown in the following listing.

```
% Load Simulink model
model = 'Controller';
```

```
load_system(model);

% Set parameter for code generation
set_param(model, 'GenCodeOnly', 'on');
set_param(model, 'PackageGeneratedCodeAndArtifacts', 'on');

% Generate code and Polyspace options file
slbuild(model)
zipFile = polyspacePackNGo(model)
```

Afterwards, the generated archive can be deployed and unpacked. Then Polyspace can be executed in the pipeline using shell commands. The operation is completed with a few lines as shown in the following listing. The generated options file in the `polyspace` subdirectory of the archive contains all information so that Polyspace can be called from there. In the example, Polyspace Bug Finder is executed with additional checks for MISRA C++:2008 compliance and metrics collection.

```
7z x Controller.zip
cd Controller\polyspace
polyspace-bug-finder -options-file optionsFile.txt -misra-cpp -code-metrics
polyspace-results-export -format json-sarif
```

Last is a a command to export the analysis results in a JSON file in the standardized Static Analysis Results Interchange Format (SARIF) defined by the Organization for the Advancement of Structured Information Standards (OASIS). Unfortunately, GitLab does not recognize the SARIF standard yet. As discussed in section 4.3.1, GitLab however supports the processing of static analysis results with its Code Quality feature. The conversion to this format can also be accomplished within MATLAB. Its `jsondecode` function is able to parse the file exported by Polyspace. Then, a structure array can be created and populated just like before. MATLAB's `jsonencode` writes the file, which is then again made available to GitLab as an artifact for display in the GUI.

Lastly, the pipeline should also terminate deliberately if any undesirable check results are raised. This can be achieved in a similar fashion to before, by terminating MATLAB with an exit code using the `quit` or `exit` functions if issues with certain characteristics are found in the decoded JSON file.

5. Results

The thesis at hand yields results of two kinds. First is the basic framework around the verification tools that makes them immediately usable. It mostly consists of various wrapper functions that configure a tool and then execute it via its API. They have been written by making use of the respective documentation and the official MATLAB generative pre-trained transformer (GPT) large language model (LLM) [45]. They are concrete, measurable results and as such described in the remainder of this chapter. Second is the evaluation of strengths and limitations of the tools that result in a series of recommendations. These are separately discussed in chapter 6.

In order to make these remarks suitable as independent documentation, some repetition could not be avoided. Automation is based on a pipeline configuration that runs every tool in sequence and uploads corresponding artifacts for use in GitLab. It can be found in appendix A.3. All model functions and supplementary scripts are documented in appendices A.1 and A.4.

5.1. MATLAB Code Analyzer

The Code Analyzer analysis is implemented in the function `runCodeAnalyzer`. It looks for any MATLAB files in the directory using MATLAB's `dir` function and wildcard characters.

```
22 % Find all MATLAB files
23 mFiles = dir('**/*.m');
```

It runs static analysis with the Code Analyzer using its `checkcode` function and subsequently displays a message for each file with issues as well as for each issue within those files.

```
% Run Code Analyzer analysis on files and display message
issuesFound = false;
for k = 1:numel(mFiles)
    filePath = fullfile(mFiles(k).folder, mFiles(k).name);
    messages = checkcode(filePath, '-id', '-struct');
    if ~isempty(messages)
        % Display issues
        disp(['Code Analyzer: Issues found in: ', filePath]);
        for i = 1:numel(messages)
            lineNum = messages(i).line;
            colRange = messages(i).column;
            msgText = messages(i).message;
            msgID = messages(i).id;
            fprintf('  Line %d (Columns %d-%d): %s\n', lineNum, colRange(1),
    colRange(end), msgText);
```

The function writes each issue into a structure array, which is required to later pass the found issues to the function `writeToCodeQuality`. To fulfill the required Code Quality formatting, each issue receives a unique fingerprint which is realized with a counter. A Boolean flag is used to mark that issues were found.

```
            % Create unique fingerprint
            fingerprint = sprintf('%s_%d', msgID, resultCounter);
            resultCounter = resultCounter + 1;

            % Append issue details to issues array
            issue = struct( ...
                'description', msgText, ...
                'check_name', msgID, ...
                'fingerprint', fingerprint, ...
                'severity', 'minor', ...
                'location', struct( ...
                    'path', filePath, ...
                    'lines', struct('begin', lineNum) ...
                ) ...
            );
            issues = [issues; issue];
        end
```

```
65          issuesFound = true;
66      end
67  end
```

If no issues were found, the function displays an according message. If it did, it passes the found issues to the function `writeToCodeQuality`. Finally, it includes an option to exit with exit code 1 if any issues are found, or code 0 if none are found.

As the issues structure array already has the required formatting, `writeToCodeQuality` simply has to encode and write it to a JSON file.

```
15  % Convert the structure array to JSON text
16  jsonText = jsonencode(issues, 'PrettyPrint', true);
17
18  % Write JSON text to the specified file
19  fid = fopen(filename, 'w');
20  if fid == -1
21      error('Cannot open file %s for writing.', filename);
22  end
23  fwrite(fid, jsonText, 'char');
24  fclose(fid);
```

The full functions can be found in appendix A.2. Declaring the created JSON file as Code Quality report artifact in the pipeline configuration lets GitLab display the results after running the associated job as shown in figure 5.1.

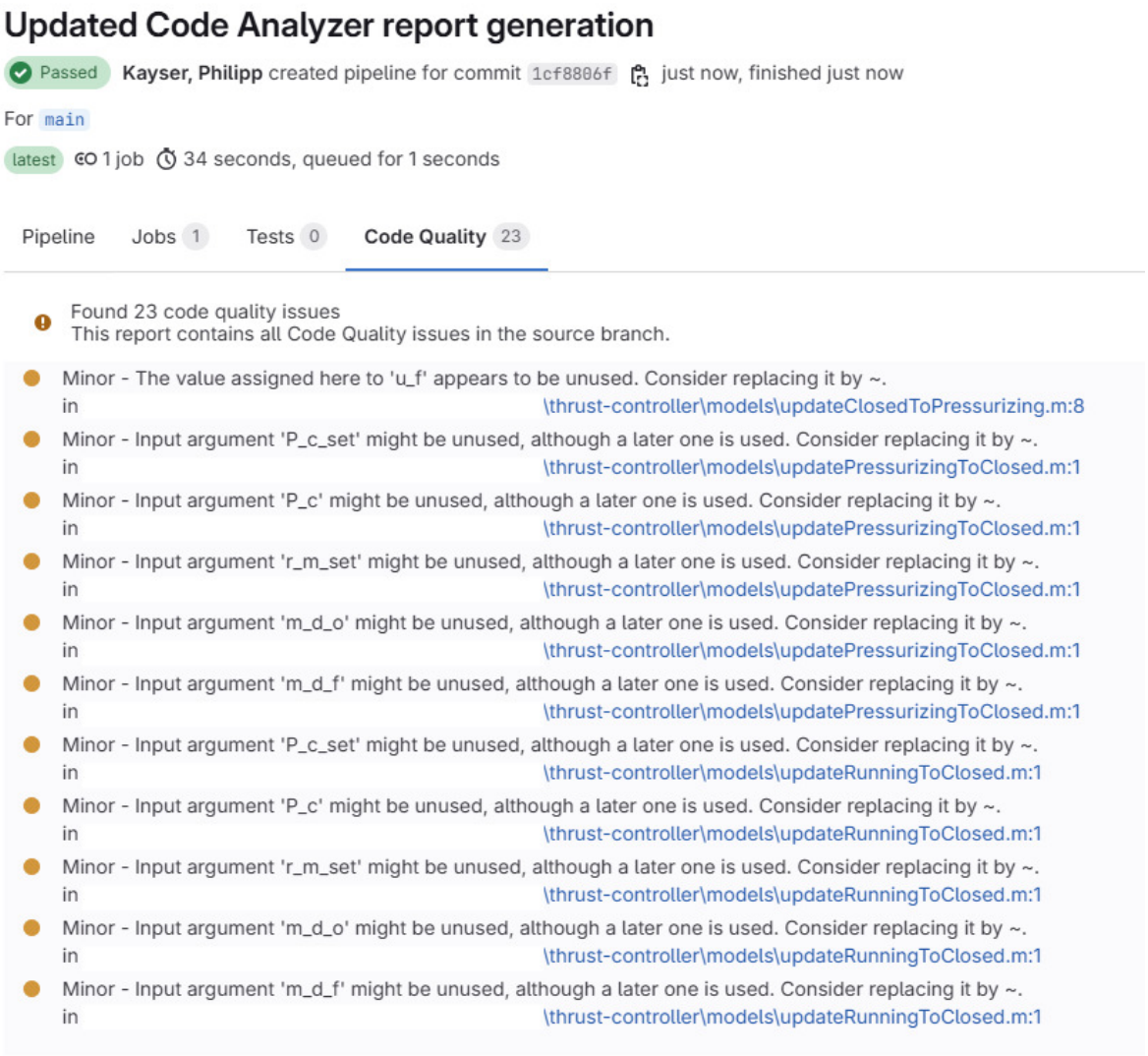

Figure 5.1.: Code Analyzer Results Displayed in GitLab

5.2. Simulink Model Advisor

The Model Advisor analysis is realized with three dependent functions. The function `getCheckIDs` returns check IDs for a Model Advisor analysis as a cell array. The cell arrays of all check IDs are defined with the name of the check as a comment, as shown in the following example.

```
30 % Define Simulink checks
31 designIDs = {
```

```
    'mathworks.design.UnconnectedLinesPorts', ...   % Identify unconnected
    lines, input ports, and output ports
    ...
    'mathworks.design.AmbiguousUnits', ...          % Identify ambiguous
    units in the model
};
```

This is repeated for the individual check ID groups that have been introduced in chapter 4. The cell arrays are returned based on what `option` keyword is passed to the function with a simple `switch case` statement.

```
% Determine which cell arrays to concatenate and return
switch lower(option)
    case 'all'
        checkIDs = [designIDs, maabIDs, hismIDs, codegenIDs, misraIDs];
    case 'design'
        checkIDs = designIDs;
    case 'advisory'
        checkIDs = maabIDs;
    case 'integrity'
        checkIDs = hismIDs;
    case 'codegen'
        checkIDs = codegenIDs;
    case 'misra'
        checkIDs = misraIDs;
    otherwise
        error('Invalid option.');
end
```

The function `runModelAdvisor` runs the given list of Model Advisor checks programmatically and generates a `HTML` report. It requires the name of the Simulink model to analyze and the cell array of Model Advisor check IDs to run provided by the function `getCheckIDs`. The analysis is run using the `ModelAdvisor.run` method.

```
% Define the report format and path
reportFormat = 'html';
reportName = 'ModelAdvisorReport';
xmlReportName = 'ModelAdvisorReport.xml';
xmlReportPath = fullfile(artifactDir, xmlReportName);
```

```
37
38 % Run Model Advisor checks
39 checkResult = ModelAdvisor.run(model, checkIDs , ...
40     'DisplayResults', 'Details', ...
41     'ReportFormat', reportFormat, ...
42     'ReportPath', artifactDir, ...
43     'ReportName', reportName);
```

The function calls the function `convertToXML` to generate a JUnit-compatible XML results file. It then also optionally exits with a non-zero code if any checks have failed via a fail flag array. This way, the function safely performs the evaluation after the results export.

```
48 % Determine exit code based on check results if activated
49 if autEval
50     % Extract SystemResult object
51     systemResult = checkResult{1};
52
53     % Retrieve array of individual check results and preallocate
54     checkObjs = systemResult.CheckResultObjs;
55     failFlags = false(1, length(checkObjs));
56
57     % Populate failFlags array
58     for i = 1:length(checkObjs)
59         failFlags(i) = strcmp(checkObjs(i).status, 'Fail');
60     end
61
62     % Display a message indicating completion
63     if any(failFlags)
64         disp('Model Advisor: Some checks failed. Exiting with error code
    1.');
65         exit(1);
66     else
67         disp('Model Advisor: All checks passed or warnings only. Exiting
    with error code 0.');
68         exit(0);
69     end
70 end
```

The function `convertToXML` converts the Model Advisor check results into a JUnit-compatible XML report. First, it obtains the results and initializes the data structure for the later export.

```
% Extract SystemResult object from check results cell array
systemResult = checkResult{1};

% Get array of individual check results from SystemResult
checkObjs = systemResult.CheckResultObjs;

% Initialize XML document
docNode = com.mathworks.xml.XMLUtils.createDocument('testsuites');
testsuites = docNode.getDocumentElement;

% Create a testsuite element with appropriate attributes
testsuite = docNode.createElement('testsuite');
testsuite.setAttribute('name', 'ModelAdvisorChecks');
testsuite.setAttribute('tests', num2str(length(checkObjs)));
testsuites.appendChild(testsuite);
```

It then writes the check results in this structure. The test case name is set using the `checkName` property. If a check has a warning or failure status a generic message is added, pointing to the Model Advisor report for details. This is done as the check results object does not contain this information in the release of MATLAB used for this thesis.

```
% Iterate over each check result
for i = 1:length(checkObjs)
    % Create a test case element for each check result
    testCase = docNode.createElement('testcase');
    % Use checkName for test case name
    testCase.setAttribute('name', checkObjs(i).checkName);

    % Define a generic message to be used for warnings and failures
    message = 'See Model Advisor report for details.';

    % If check failed, add a failure element with message
    if strcmp(checkObjs(i).status, 'Fail')
        failure = docNode.createElement('failure');
        failure.setAttribute('message', ['Check failed. ' message]);
```

```
45          testCase.appendChild(failure);
46      % If check returned a warning, add a systemOut element with message
47      elseif strcmp(checkObjs(i).status, 'Warn')
48          systemOut = docNode.createElement('system-out');
49          systemOut.appendChild(docNode.createTextNode(['Check returned a
    warning. ' message]));
50          testCase.appendChild(systemOut);
51      end
52
53      % Append test case element to testsuite
54      testsuite.appendChild(testCase);
55  end
```

The XML file itself is finally written using MATLAB's `xmlwrite` function. Again, the complete functions can be found in appendix A.2. Declaring the written file as JUnit report artifact in the pipeline configuration enables the results to be displayed like test cases in GitLab as shown in figure 5.2.

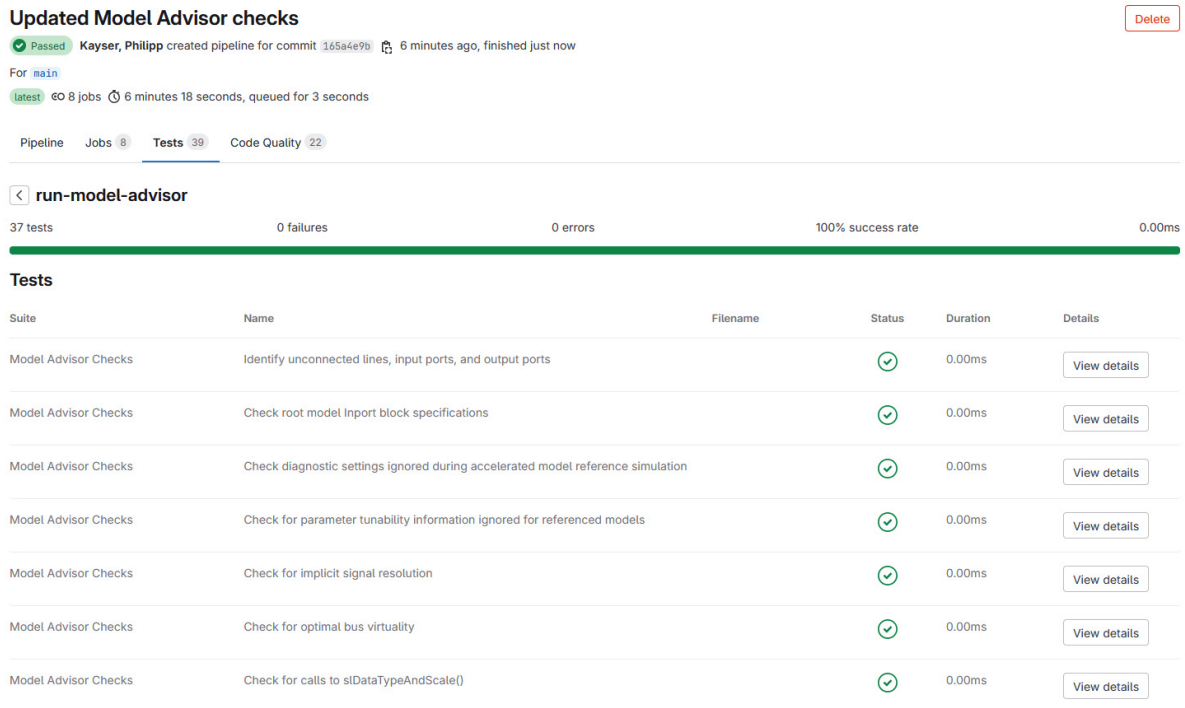

Figure 5.2.: Model Advisor Results Displayed in GitLab

5.3. Simulink Design Verifier

The function `runDesignVerifier` configures Simulink Design Verifier for design error detection, runs the analysis on the specified model and saves a report with data file. The configuration is done with the design verification options object `sldvOptions` and if required could be adjusted here.

```
% Configure Simulink Design Verifier for design error detection
sldvOptions = sldvoptions;
sldvOptions.Mode = 'DesignErrorDetection';
sldvOptions.DetectBlockInputRangeViolations = 'off';
sldvOptions.DetectDeadLogic = 'off';
sldvOptions.DetectDivisionByZero = 'on';
sldvOptions.DetectInfNaN = 'off';
sldvOptions.DetectIntegerOverflow = 'off';
sldvOptions.DetectOutOfBounds = 'off';
sldvOptions.DetectSubnormal = 'off';
sldvOptions.SaveReport = 'on';
```

The analysis is run with these options using the `sldvrun` function.

```
% Run Design Verifier analysis
[status, files, ~] = sldvrun(model, sldvOptions);
```

The returned character `status` is used to discern between suitable messages addressing the analysis results. Apart from the conventional error exit code, there is a case for an analysis timeout. The function displays an appropriate message for each case. Finally, a HTML report as well as an analysis data file is saved. The latter contains the raw model checking traces that the Design Verifier used.

5.4. Simulink Test

Testing with Simulink Test is implemented in the function `runTests`. In its final implementation, the function runs the defined tests using the MATLAB Unit Test framework. The test files are loaded with the Simulink Test Manager, from which MATLAB Unit Test creates a test suite and a test runner.

```
% Open the test file
fileName = 'ControllerTests.mldatx';
filePath = fullfile(pwd, 'tests', fileName);
sltest.testmanager.load(filePath);

% Create test suite from test file
import matlab.unittest.TestSuite
suite = testsuite(filePath);

% Create test runner
import matlab.unittest.TestRunner
runner = TestRunner.withNoPlugins;
```

As with any MATLAB Unit Test script or function, plugin classes are added that provide the required reporting capabilities. Here, a PDF and a XML report in JUnit format are created.

```
% Add plugin to produce MATLAB Test Report
import matlab.unittest.plugins.TestReportPlugin
pdfFile = fullfile(artifactDir, 'TestReport.pdf');
trp = TestReportPlugin.producingPDF(pdfFile);
addPlugin(runner,trp)

% Add plugin to add Test Manager results to Test Report
import sltest.plugins.TestManagerResultsPlugin
tmr = TestManagerResultsPlugin;
addPlugin(runner,tmr)

% Add plugin to create XML results file
import matlab.unittest.plugins.XMLPlugin
resfile = fullfile(artifactDir, 'TestResults.xml');
plugin = XMLPlugin.producingJUnitFormat(resfile);
addPlugin(runner,plugin)
```

Simulink Test plugins provide additional coverage collection and reporting capabilities. In this case, decision coverage is collected and exported to a report in the standardized Cobertura format.

```
% Set coverage metrics to collect
import sltest.plugins.coverage.CoverageMetrics
```

```
cmet = CoverageMetrics('Decision',true);

% Set coverage report properties
import sltest.plugins.coverage.ModelCoverageReport
import matlab.unittest.plugins.codecoverage.CoberturaFormat
rptfile = fullfile(artifactDir, 'TestCoverage.xml');
rpt = CoberturaFormat(rptfile);

% Create model coverage plugin
import sltest.plugins.ModelCoveragePlugin
mcp = ModelCoveragePlugin('Collecting',cmet,'Producing',rpt);
addPlugin(runner,mcp)
```

Finally, the test are run using the `run` method of the `matlab.unittest.TestRunner` class. The created artifacts are declared in the pipeline configuration and are accordingly displayed by GitLab as shown in figure 5.3.

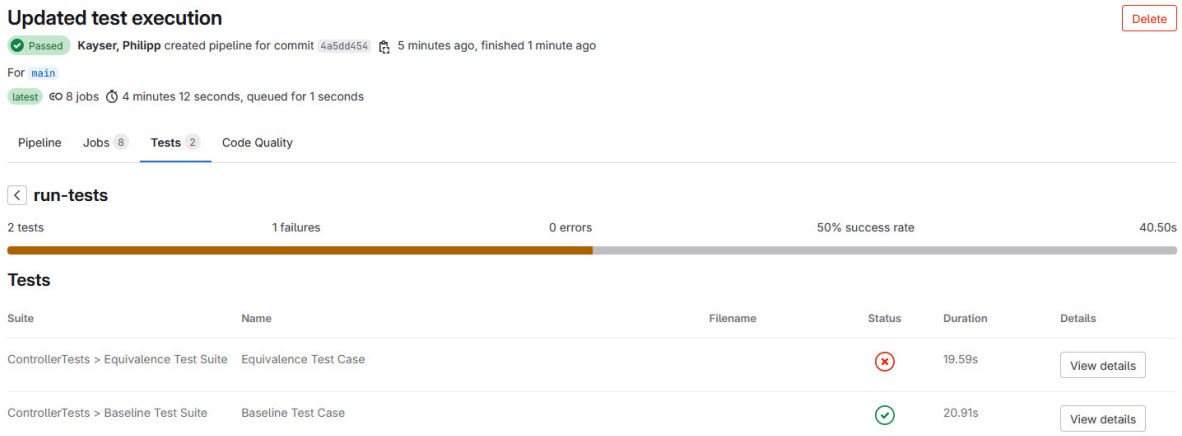

Figure 5.3.: Test Results Displayed in GitLab

5.5. Embedded Coder

The code generation process is encapsulated in two functions. The function `setConfiguration` sets the desired model configuration parameters. The configuration was obtained by running the script `getConfiguration`. For clarity, the parameter settings are commented as shown in the following excerpt.

```
% Set parameters for C++ code generation
set_param(model, 'SystemTargetFile', 'ert.tlc');            % System target
    file
set_param(model, 'TargetLang', 'C++');                      % Select code
    generation language
set_param(model, 'GenCodeOnly', 'on');                      % Do not execute
    makefile when generating code
set_param(model, 'TargetLangStandard', 'C++03 (ISO)');      % Language
    standard
set_param(model, 'PackageGeneratedCodeAndArtifacts', 'on'); % Automatically
    run packNGo after the build is complete
set_param(model, 'BuildConfiguration', 'Faster Runs');      % Choose a build
    configuration defined by the toolchain
```

After calling `setConfiguration`, the function `generateCode` invokes Embedded Coder and subsequently creates the Polyspace options file.

```
% Set configuration parameters
setConfiguration(model);

% Generate the code
slbuild(model);

% Generate and package Polyspace options files
polyspacePackNGo(model);
```

5.6. Polyspace

The dedicated functions `runPolyspaceBugFinder` and `runPolyspaceCodeProver` execute a Polyspace analysis from within MATLAB and can be found in appendix A.2. The aforementioned and more favorable execution with shell commands is described in the pipeline configuration.

For Polyspace Bug Finder, the following pipeline script applies. The generated code archive is unpacked, where the Polyspace configuration is found in the automatically created options file. From there, the analysis is started, which in this case includes all Bug Finder

checks and the required part of the MISRA C++:2008 rules. The options file contains relative paths pointing to all required functions. Afterwards, the results are exported to a SARIF formatted JSON file, which must be converted to the Code Quality format by the function `convertToCodeQuality`.

```
script:
  - 7z x ThrustController.zip
  - cd thrust-controller\polyspace
  - polyspace-bug-finder -options-file optionsFile.txt -checkers all
    -misra-cpp required-rules
  - polyspace-results-export -format json-sarif -output-name
    bugFinderResults.json
  - cd ..\..
  - matlab -wait -batch
    "convertToCodeQuality('thrust-controller\polyspace\bugFinderResults.json',
    'bugFinderCodeQuality.json')"
artifacts:
  reports:
    codequality:
      - bugFinderCodeQuality.json
```

The OASIS SARIF format is more complex than the Code Quality format. Additionally, the output files differ slightly in structure between Bug Finder and Code Prover, which leads to the function `convertToCodeQuality` being quite comprehensive. The general approach is however similar to before, where an appropriately formatted structure array is populated with the found issues and then written to a JSON file. Only this time, the results have to be decoded from another JSON file beforehand.

```
% Read and decode the input SARIF file
fid = fopen(inSarifFile, 'r');
raw = fread(fid, '*char')';
fclose(fid);
sarifData = jsondecode(raw);

% Initialize an empty struct array for the output
gitlabFindings = struct('description', {}, 'check_name', {}, 'fingerprint',
    {}, 'severity', {}, 'location', {});
```

In the SARIF format, results are structured in individual *runs*, which needs to be considered with an additional `for` loop. Further, the source JSON file contains a dedicated section listing the full names associated with the IDs of check rules. As this list is different depending on the results found in an analysis, it is dynamically stored in a MATLAB `Map` object to later obtain the rule names.

```
% Build a mapping from rule IDs to rule names
ruleMap = containers.Map;
if isfield(runData, 'tool') && isfield(runData.tool, 'driver') && ...
        isfield(runData.tool.driver, 'rules')
    for iRule = 1:numel(runData.tool.driver.rules)
        thisRule = runData.tool.driver.rules(iRule);
        if isfield(thisRule, 'id') && isfield(thisRule, 'name')
            ruleKey = strtrim(char(thisRule.id));
            ruleMap(ruleKey) = thisRule.name;
        end
    end
end
```

After obtaining the paths of the files with issues from the *artifacts* section in the source file, the actual results can be processed. It was found that decoding a Bug Finder JSON file yields a cell array, while a Code Prover JSON file does not. To be able to read results from both tools, this is checked with MATLAB's `iscell` function.

```
% Process results
for iRes = 1:numel(runData.results)
    if iscell(runData.results) % Bug Finder JSON
        res = runData.results{iRes};
    else % Code Prover JSON
        res = runData.results(iRes);
    end

    % Extract description message
    if isfield(res, 'message') && isfield(res.message, 'text')
        descriptionText = strtrim(char(res.message.text));
    else
        descriptionText = '(No message provided)';
    end

```

```
82      % Extract rule ID and look up rule name
83      if isfield(res, 'ruleId')
84          ruleId = strtrim(char(res.ruleId));
85          if isKey(ruleMap, ruleId)
86              checkName = ruleMap(ruleId);
87          else
88              checkName = ruleId;
89          end
90      else
91          ruleId = 'unknown_rule';
92          checkName = 'unknown_rule';
93      end
```

The same has to be considered for determining the file paths. With these prerequisites, the fields `description`, `check_name` and `location` for the target Code Quality JSON file can be populated. The field `fingerprint` is again realized with a counter, and for `severity` the helper function `mapSeverity` is used.

The function `mapSeverity` per default returns the severity `'info'`. For Bug Finder, the type of defect is used for the severity classification, which is stored in the `metaFamily` property. For Code Prover, the color of the issue is used instead.

```
31  % Use metaFamily mapping for Bug Finder
32  if ~isempty(metaFamily)
33      switch metaFamily
34          case 'Defect'
35              severity = 'major';
36          case 'Coding Rule'
37              severity = 'minor';
38      end
39
40  % Use color mapping for Code Prover
41  elseif ~isempty(color)
42      switch color
43          case 'RED'
44              severity = 'critical';
45          case 'GRAY'
46              severity = 'major';
```

```
47          case 'ORANGE'
48              severity = 'minor';
49      end
```

With the returned `severity` string, `convertToCodeQuality` can write the JSON output.

```
136 % Encode results to JSON
137 jsonOut = jsonencode(gitlabFindings, 'PrettyPrint', true);
138
139 % Write JSON output
140 fid = fopen(outGitlabFile, 'w');
141 fwrite(fid, jsonOut, 'char');
142 fclose(fid);
```

Optionally, the function can also quit with exit code 1 and terminate the job as failed if any found result has `'critical'` severity. Declaring the created file as Code Quality report artifact in the pipeline configuration lets GitLab process and display the findings as shown in figure 5.4.

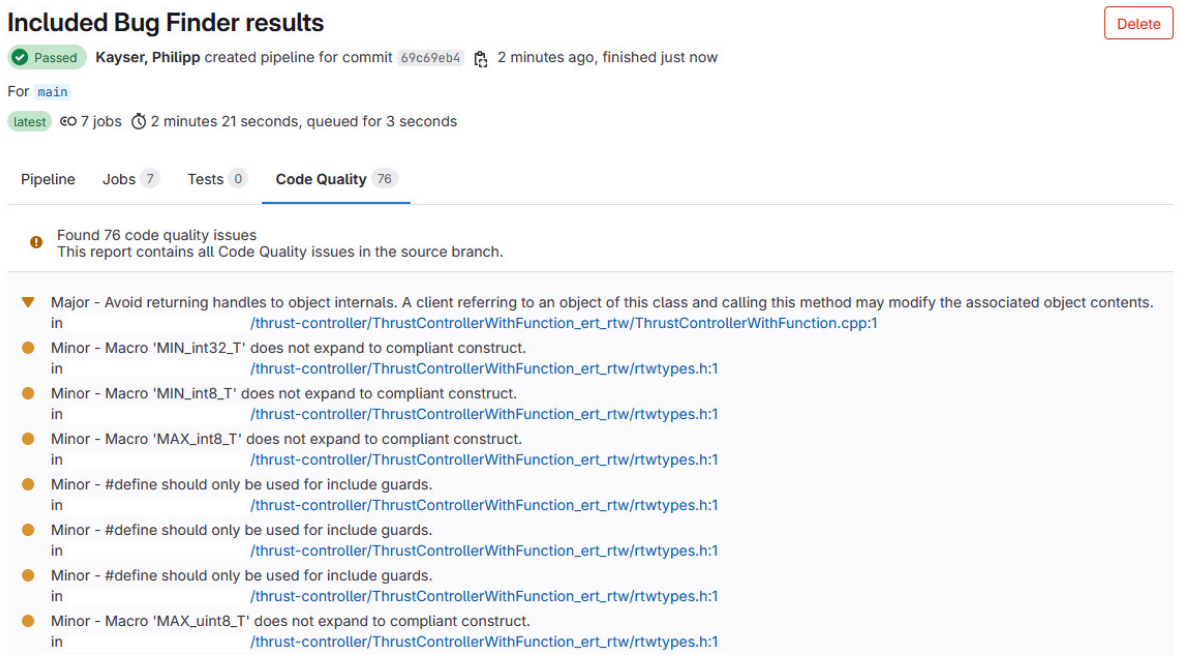

Figure 5.4.: Bug Finder Results Displayed in GitLab

For Polyspace Code Prover, the entire approach is similar. The results are displayed in GitLab as shown in figure 5.5.

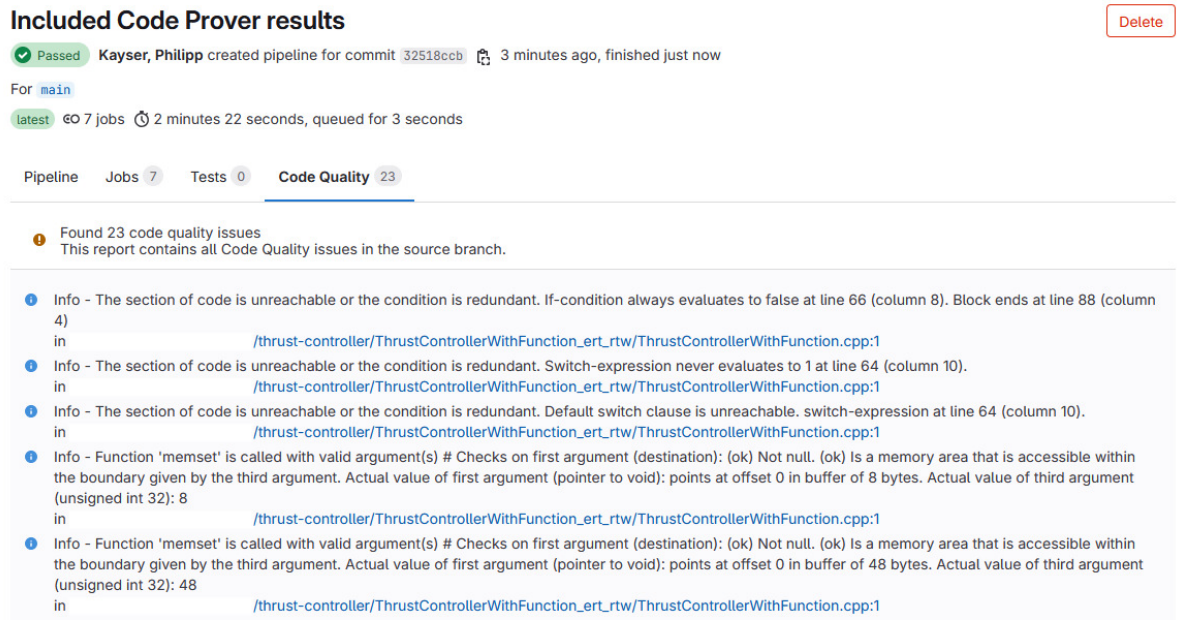

Figure 5.5.: Code Prover Results Displayed in GitLab

6. Discussion

Any attempt to automate software verification is only as good as the processes themselves are defined, accepted and run. To cite Fewster and Graham, "automating chaos just gives faster chaos" [2]. While it is beyond the scope of this thesis to consider an entire development and verification process, several contributions in that regard can however be noted.

In their online presence, MathWorks emphasize what can theoretically be achieved with their software verification products. It is however difficult to get an overview of the applicable prerequisites and constraints. The impression arises that an admirable level of software quality can be achieved swiftly and user-friendly. Throughout chapter 5, it has been shown that in fact, automation of every tool under consideration is possible without difficulty just by following the documentation. The addition of functionality as here in the context of results evaluation is possible due to MATLAB's wide scripting possibilities. It should be noted that, while comprehensive, MathWorks documentation oftentimes does not have the character of a cohesive user guide. This thesis aims to provide just that with chapters 4 and 5. With regard to the research question, the answer is different for each tool. More fine-grained critique depends on the technical details and must be given individually.

In chapter 2 it was shown that the expressiveness of static analysis completely depends on the preconfigured checks a tool offers. The Code Analyzer is the only option with respect to static analysis of MATLAB code. As was shown in table 4.3, the tool provides a basic level of software quality for general MATLAB applications but falls short when it comes to production code generation. The tool alone is generally not sufficient for software verification with respect to correctness. As it is easy to integrate in a development and verification process, the recommendation here is to apply it nevertheless as an additional early layer that improves code quality. Apart from the justification of individual issues, global configuration of the analysis can be done programmatically if needed, which promises to make the tool flexible and unobtrusive in its use [6].

With the Model Advisor, MathWorks provides static model analysis capabilities at a deep semantic level. Simulink Check is an inevitable addition to use the Model Advisor consistently in a verification process, both because of the programmatic access and the compliance checks the product provides. These checks serve as a benchmark for a basic level of model quality and the Model Advisor in that regard is again without any alternative. With respect to software correctness, the tool must inevitably be complemented by additional measures further downstream in the development and verification process. As can be seen in tables 4.5 to 4.8, the checks encompass breadth over depth. Various industry standards are included but not fully covered. Achieving full compliance to one of the standards would be associated with considerable manual work either way. Additionally, none of the standards pertain to spacecraft systems directly. NASA published a set of guidelines for modeling with MATLAB/Simulink that was developed and applied during the development of the Orion GNC algorithms and flight software [20]. The starting point were the MAB guidelines, additional rules and custom checks were added over time from learnings of the development process. The recommendation here is to follow the same procedure and to first apply the MAB guidelines and then to customize them with the Orion GNC MATLAB/Simulink Standards as template.

With respect to model checking, a general advantage is that the accurate modeling of a system can already lead to the discovery of ambiguities and inconsistencies in a specification. A verification with model checking on the other hand is only as good as the model of the system [5]. Simulink Design Verifier adds to that limitation in its attempt to make model checking compatible with any Simulink model. Compared to how versatile a verification with model checking theoretically can be, the preconfigured checks of the Design Verifier are rather limited. While custom properties can be modeled as assertions, the degree of elaboration that LTL and CTL formulae offer are superior. A case study conducted at the University of Konstanz demonstrated that this restriction considerably limits the tools usefulness in real-world scenarios. For the properties that could be modeled as assertions, the standalone model checker SPIN concluded the analysis orders of magnitude faster than Simulink Design Verifier. Scaling the analyzed model further amplified this discrepancy [46]. When applied to feedback control systems, compatibility issues with Simulink were found to make the analysis unpredictable which might lead to low acceptance of the tool among developers. This is only reinforced by the fact that static analysis tools provide the same error detection capabilities at source code level with much more detailed context. At this time, the tool can therefore not be regarded suitable for the verification of GNC systems. The recommendation here is to investigate if an applicable scenario can be found where root level signals of a

model can be consistently limited to a finite range and a Design Verifier analysis can be run robustly without timeout and compatibility issues. Based on the results of this thesis, it has to be expected that this might not be possible.

Simulink Test was first and foremost found to be an interface for functionality that is already provided by Simulink and Embedded Coder. In that regard, a main benefit of the tool is the Test Manager which makes structured test authoring and management easy. The added value of the tool towards verification however depends mostly on the quality of the tests written for it, which can be facilitated but not substituted by the tool. Nevertheless, Simulink Test can be considered a valuable supplement to MATLAB Unit Test mainly due to its aptness for integration in a CI workflow. In particular, the intuitive definition of tolerance bands, the ease of SIL testing and the coverage metrics collection were found to be promising features – with especially the latter being of importance in the verification for correctness. With respect to the research question, the Simulink Test can be regarded as suitable, however under the condition that it is used adequately. The recommendations here is to use it for automated integration and SIL testing in conjunction with unit testing with MATLAB Unit Test.

At the time of writing, there are more than 300 reported bugs related to incorrect code generation with Embedded Coder in the MATLAB/Simulink release used for this thesis [29]. In the development and verification process, these are added to the bugs inherent to the source code compiler (cf. figure 4.13). This fact undermines any verification results that was provided prior to code generation and emphasizes the importance of static analysis later in the process. As table 4.9 shows, a Polyspace Bug Finder analysis is rather comprehensive. In comparison, the Clang static analyzer for example features around 100 non-experimental checks and can check compliance with one coding standards (SEI CERT C with experimental checks) [47]. With around 190 checks, the static analysis tool cppcheck is more comprehensive but does not consider coding standards [48]. This comparison of course does not consider the quality of the checks, but semantic configuration options like the Polyspace tools offer are at the time of writing not documented for both. As the added value of a consistently applied Polyspace analysis stands out among all tools considered for this thesis, the recommendation here is to prioritize the integration of Polyspace Bug Finder into the development and verification process. With the methods presented in chapter 4 the configuration of the analysis can be created largely automated and reused for Polyspace Code Prover. In order not to obstruct the existing continuous integration workflows, it is recommended to run the default set of Polyspace Bug Finder defect checks automated in a pipeline, while coding guideline compliance checks are run asynchronously.

A static analysis tool is expected to ideally find all defects present in the analyzed code (i.e. minimize the amount of false negatives) without reporting issue that do not constitute actual errors (i.e. minimize the amount of false positives). While in this small-scale demonstration, the reproduction of either in the generated code was not possible, it is difficult to guarantee the same in industry-scale applications. Static analysis by abstract interpretation provides the added benefit of a mathematically sound analysis without any false negatives. It was shown that this is achieved for the sake of completeness and entirely depends on the expressiveness of the abstract domain (cf. chapter 2). With this in mind, sound static analysis tools like Polyspace Code Prover can be regarded as instrumental for software verification with respect to correctness in any safety-critical application. The recommendation here is to run Polyspace Code Prover asynchronously and complementary to Polyspace Bug Finder. The results of these less frequent but more rigorous analyses should then inform the justification of potential issues reported by Bug Finder's defect checks.

Evaluation in this context is non-trivial, which is why dedicated benchmark test suites for static analysis tools exist. The National Institute of Standards and Technology (NIST) Juliet Test Suite for C/C++ for example contains more than 60 000 test cases that cover around 1 600 types of defects. With these, tool providers can obtain evaluation metrics such as the *rate of true positives* – the ratio of true defects recognized by the tool to true defects in the code – and the *rate of false positives* – the ratio of false positives to defect-free statements and expressions in the code. Generally, a low false positive rate in conjunction with a high true positive rate is desirable. For sound analysis tools, a true positive rate of 1 is a functional requirement and the false positive rate is the primary metric [13]. It should be noted that MathWorks does not report these metrics for the Polyspace analysis tools, which is why a qualified answer regarding their evaluation with respect to other tools can not be given at this time. The recommendation here is to consider a detailed comparison with competing tools at a future point in time. The static analysis tool Astrée for example was evaluated to satisfy the rigorous Ockham Sound Analysis Criteria and is even used for the verification of the aforementioned Juliet Test Suite for C/C++ [49].

7. Conclusion and Outlook

In conclusion, it can be stated that the presented suite of tools is overall well suited for the verification of flight software in the context of GNC systems with respect to correctness. It was discussed in detail that this suite is neither complete nor that tools fulfill this statement individually. But reflecting on the introductory statement that software should be assumed to be faulty until demonstrated otherwise, MathWorks provides the means to do just that.

Comprehensive semantic analysis of source code in conjunction with abstract interpretation is the cornerstone of this verification approach. The tool suite does not substitute testing, but complement it. Only model checking was found to be unsuitable, all other tools employ techniques that directly or indirectly raise software quality. This comes at a significant pricing, so the investigation should not be seen as concluded here. In fact, comparable theses consider one tool in isolation and conduct appropriate case studies [46], [50].

Looking forward, Code Analyzer and Polyspace Bug Finder can already be implemented in project context. In parallel, use cases for Simulink Test can be identified and first test cases written. In the mid-term, a tailored configuration of static analysis checks should have been created, which can then be used to include an automated evaluation of analysis results. Polyspace Code Prover would be run asynchronously but regularly by designated developers. More test cases for Simulink Test would have been created by then and can be implemented in pipelines, possibly extending to SIL testing. In the long term, a more profound evaluation of all tools in project context should exist. These could for example be detailed case studies, should be more distinct and quantitative than this thesis and take into consideration that some of the tools have alternatives. Several examples that merit further investigation have been named in this thesis. In general, the fact that lowering the dependence on MathWorks products creates new possibilities can also have a positive impact on software quality.

Bibliography

[1] J.-L. Lions, *Ariane 501 Inquiry Board Report*, Jul. 1996.

[2] A. Spillner and T. Linz, *Software Testing Foundations, A Study Guide for the Certified Tester Exam*. Rocky Nook, 2021, ISBN: 9781681988535.

[3] C. Cowell, *Automating DevOps with GitLab CI/CD Pipelines*, N. Lotz and C. Timberlake, Eds. Packt Publishing, 2023, ISBN: 9781803242934.

[4] P. Cousot, "Abstract Interpretation: From 0, 1, to ∞," in *Challenges of Software Verification*. Springer Nature Singapore, 2023, pp. 1–18, ISBN: 9789811996016. DOI: 10.1007/978-981-19-9601-6_1.

[5] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, Massachusetts: The MIT Press, 2008, ISBN: 9781435643277.

[6] The MathWorks Inc., *Matlab Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: https://www.mathworks.com/help/simulink/index.html (Last accessed 04/18/2025).

[7] R. Schwarz, D. Kiehn, G. F. Trigo, *et al.*, "Overview of Flight Guidance, Navigation, and Control for the DLR Reusability Flight Experiment (ReFEx)," 2019. DOI: 10.13009/EUCASS2019-739.

[8] K. Ogata, *Modern Control Engineering*. Prentice-Hall, 2010, ISBN: 9780136156734.

[9] R. C. Dorf, *Moderne Regelungssysteme*, 10th ed., R. H. Bishop, Ed. München: Pearson Studium, 2007, ISBN: 9783863266233.

[10] The MathWorks Inc., *Simulink Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: https://www.mathworks.com/help/simulink/index.html (Last accessed 04/25/2025).

[11] GitLab Inc., *GitLab Documentation*, 2025. [Online]. Available: https://docs.gitlab.com/ (Last accessed 05/08/2025).

[12] J. Peleska and W.-l. Huang, *Test Automation: Foundations and Applications of Model-Based Testing*, Lecture Notes, Nov. 2021.

[13] J. Herter, D. Kästner, C. Mallon, and R. Wilhelm, "Benchmarking Static Code Analyzers," *Reliability Engineering & System Safety*, vol. 188, pp. 336–346, Aug. 2019. DOI: 10.1016/j.ress.2019.03.031.

[14] H. Herold, *Grundlagen der Informatik*, 4th ed., B. Lurz, M. Lurz, and J. Wohlrab, Eds. München: Pearson, 2023, ISBN: 9783863263515.

[15] R. Giacobazzi, I. Mastroeni, and E. Perantoni, "How Fitting is Your Abstract Domain?" In Springer Nature Switzerland, 2023, pp. 286–309, ISBN: 9783031442452. DOI: 10.1007/978-3-031-44245-2_14.

[16] T. Tantau, *The TikZ and PGF Packages*, 2025. [Online]. Available: https://pgf-tikz.github.io/pgf/pgfmanual.pdf (Last accessed 04/17/2025).

[17] W. A. Storm, "A Model Checking Example: Solving Sudoku Using Simulink Design Verifier," *Lockheed Martin Corporation*, 2009.

[18] M. Sheeran and G. Stålmarck, "A Tutorial on Stålmarck's Proof Procedure for Propositional Logic," in *Formal Methods in Computer-Aided Design*. Springer Berlin Heidelberg, 1998, pp. 82–99, ISBN: 9783540495192. DOI: 10.1007/3-540-49519-3_7.

[19] NASA Engineering & Safety Center, *Technical Update 2015*, 2015. [Online]. Available: https://www.nasa.gov/nesc/knowledge-products/technical-updates/ (Last accessed 04/28/2025).

[20] M. Jackson and J. Henry, "Orion GN&C Model Based Development: Experience and Lessons Learned," in *AIAA Guidance, Navigation, and Control Conference*, American Institute of Aeronautics and Astronautics, Aug. 2012. DOI: 10.2514/6.2012-5036.

[21] V. Hadzilacos, *Introduction to the Theory of Computation*, Lecture notes, 2007.

[22] C. F. Lorenzo and J. L. Musgrave, "Overview of Rocket Engine Control," in *AIP Conference Proceedings*, vol. 246, AIP, 1992, pp. 446–455. DOI: 10.1063/1.41807.

[23] Y. C. Lee, M. R. Gore, and C. C. Ross, "Stability and Control of Liquid Propellant Rocket Systems," *Journal of the American Rocket Society*, vol. 23, no. 2, pp. 75–81, Mar. 1953, ISSN: 1936-9964. DOI: 10.2514/8.4544.

[24] G. P. Sutton, *Rocket Propulsion Elements*, 9th ed., O. Biblarz, Ed. Hoboken, New Jersey: John Wiley & Sons Inc., 2017, 1 p., ISBN: 9781118753651.

[25] H. Coxinho, T. Raposo, and E. Moreno, "Mixture Ratio and Thrust Control of a Liquid-Propellant Rocket Engine," 2016.

[26] ArianeGroup GmbH, *Orbital Propulsion Fluidic Equipment*.

[27] L. Shure and D. Bergstein, *Run Code Faster With the New MATLAB Execution Engine*, Blog post, Feb. 2016. [Online]. Available: `https://blogs.mathworks.com/loren/2016/02/12/run-code-faster-with-the-new-matlab-execution-engine/` (Last accessed 04/22/2025).

[28] The MathWorks Inc., *Simulink Check Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/slcheck/index.html` (Last accessed 05/06/2025).

[29] The MathWorks Inc., *Embedded Coder Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/ecoder/` (Last accessed 05/01/2025).

[30] Software Engineering Institute, *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*, 2016.

[31] The MITRE Corporation, *Common Weakness Enumeration*, 2024.

[32] Motor Industry Software Reliability Association, *MISRA C:2012 Guidelines for the Use of the C Language in Critical Systems*, Nuneaton, UK, 2013.

[33] The MathWorks Inc., *Simulink Design Verifier Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/sldv/` (Last accessed 04/29/2025).

[34] The MathWorks Inc., *Simulink Test Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/sltest/` (Last accessed 04/20/2025).

[35] The MathWorks Inc., *Simulink Coder Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/rtw/` (Last accessed 04/21/2025).

[36] The MathWorks Inc., *Polyspace Bug Finder Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/bugfinder/index.html` (Last accessed 05/10/2025).

[37] The MathWorks Inc., *Polyspace Code Prover Documentation*, Natick, Massachusetts, United States, 2025. [Online]. Available: `https://www.mathworks.com/help/codeprover/index.html` (Last accessed 04/30/2025).

[38] B. Stroustrup, *A Tour of C++* (C++ In-Depth Series), Third edition. Boston: Addison-Wesley, 2023, 299 pp., ISBN: 0136816487.

[39] Motor Industry Software Reliability Association, *MISRA C++:2008 Guidelines for the Use of the C++ Language in Critical Systems*, Nuneaton, UK, 2008.

[40] Automotive Open System Architecture, *Guidelines for the Use of the C++14 Language in Critical and Safety-Related Systems*, 2018.

[41] Motor Industry Software Reliability Association, *MISRA C++:2023: Guidelines for the Use of C++17 in Critical Systems*, Nuneaton, UK, 2023.

[42] J. Frey, *Jenkins Simulink Model Advisor*, GitHub Repository, 2017. [Online]. Available: `https://github.com/dapperfu/Jenkins-Simulink-Model-Advisor` (Last accessed 04/23/2025).

[43] S. Bechtold, S. Brannen, J. Link, *et al.*, *JUnit 5 User Guide*, 2025. [Online]. Available: `https://junit.org/junit5/docs/current/user-guide/` (Last accessed 05/11/2025).

[44] Oracle, *Java Document Object Model Interface Documentation*, 2020. [Online]. Available: `https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/package-summary.html` (Last accessed 04/16/2025).

[45] The MathWorks Inc., *The Official MATLAB GPT by MathWorks*, 2025. [Online]. Available: `https://chatgpt.com/g/g-QFTjbeK3U-matlab` (Last accessed 05/05/2025).

[46] F. Leitner-Fischer and S. Leue, "Simulink Design Verifier vs. SPIN - A Comparative Case Study," in *Participant's Proceedings of FMICS 2008, ERCIM Working Group on Formal Methods for Industrial Critical Systems*, 2008.

[47] LLVM Contributors, *Clang Compiler User's Manual*, 2025. [Online]. Available: `https://clang.llvm.org/docs/UsersManual.html` (Last accessed 05/05/2025).

[48] D. Marjamäki, *Cppcheck - A Tool for Static C/C++ Code Analysis*, 2025. [Online]. Available: `https://sourceforge.net/p/cppcheck/wiki/Home/` (Last accessed 05/04/2025).

[49] P. E. Black and K. S. Walia, *SATE VI Ockham Sound Analysis Criteria*. May 2020. DOI: `10.6028/nist.ir.8304`.

[50] M. Liliegard and V. Nilsson, "Model-Based Testing with Simulink Design Verifier," M.S. thesis, Chalmers University Of Technology, Göteborg, Sweden, 2014.

A. Appendix

A.1. Model Functions

```
function [u_o, u_f] = thrustControlFunction(P_c_set, P_c, r_m_set, m_d_o, m_d_f)
    %#codegen
% Thrust controller function
%
% DESCRIPTION:
%   This function is an implementation of the thrust controller as a MATLAB
%   function. It is designed to be called at each simulation step. The
%   integral terms are stored in persistent variables so that they
%   accumulate across time steps.
%
% INPUTS:
%   P_c_set  - Desired chamber pressure
%   P_c      - Measured chamber pressure
%   r_m_set  - Desired mixture ratio
%   m_d_o    - Current oxidizer mass flow
%   m_d_f    - Current fuel mass flow
%
% OUTPUTS:
%   u_o      - Control output for oxidizer valve (chamber pressure loop)
%   u_f      - Control output for fuel valve (mixture ratio loop)

    % Sample time
    Ts = 0.001;

    % Controller gains
    KpC  = 1e-7;   % Proportional gain for chamber pressure
    KiC  = 1e-6;   % Integral gain for chamber pressure

    KpMR = -0.3;   % Proportional gain for mixture ratio
    KiMR = -3;     % Integral gain for mixture ratio
```

```
30
31      % Persistent states for the integrators
32      persistent intC  intMR
33
34      % Initialize the integrators on the first call
35      if isempty(intC)
36          intC  = 0;
37          intMR = 0;
38      end
39
40      %% Chamber Pressure Controller
41
42      % Error
43      eC = P_c_set - P_c;
44
45      % Integrator update
46      intC = intC + eC * Ts;
47
48      % PI output
49      u_o = KpC * eC + KiC * intC;
50
51      %% Mixture Ratio Controller
52
53      % Compute mixture ratio
54      if m_d_f == 0
55          r_m = 0;
56      else
57          r_m = m_d_o / m_d_f;
58      end
59
60      % Error
61      eMR = r_m_set - r_m;
62
63      % Integrator update
64      intMR = intMR + eMR * Ts;
65
66      % PI output
67      u_f = KpMR * eMR + KiMR * intMR;
68
69  end
```

```
1  classdef ThrustControllerStates < Simulink.IntEnumType
2  % Enumeration of controller states
```

```
    enumeration
        Closed        (0)   % Valves fully closed, no control
        Pressurizing  (1)   % Bringing chamber pressure up to desired level
        Running       (2)   % Normal operation, both loops active
    end
end
```

```
function [u_o, u_f] = thrustControlFSM(P_c_set, P_c, r_m_set, m_d_o, m_d_f) %#codegen
% Thrust controller state machine implementation
%
% DESCRIPTION:
%   This function represents a state machine implementation for a thrust
%   controller with persistent variables. Each transition is encapsulated
%   in a helper function.
%
% INPUTS:
%   P_c_set  - Desired chamber pressure
%   P_c      - Measured chamber pressure
%   r_m_set  - Desired mixture ratio
%   m_d_o    - Current oxidizer mass flow
%   m_d_f    - Current fuel mass flow
%
% OUTPUTS:
%   u_o      - Control output for oxidizer valve (chamber pressure loop)
%   u_f      - Control output for fuel valve (mixture ratio loop)

    % Persistent variables for the FSM
    persistent currentState intC intMR
    if isempty(currentState)
        currentState = ThrustControllerStates.Closed;
        intC = 0;
        intMR = 0;
    end

    % Initialize outputs
    u_o = 0;
    u_f = 0;

    % Switch for current state
    switch currentState

        case ThrustControllerStates.Closed
            if P_c_set > 0
```

```
37                % Transition Closed to Pressurizing
38                [u_o, u_f, currentState, intC, intMR] = ...
39                    updateClosedToPressurizing(P_c_set, P_c, r_m_set, ...
40                    m_d_o, m_d_f, intC, intMR);
41            else
42                % Remain in Closed
43                [u_o, u_f, currentState, intC, intMR] = ...
44                    remainClosed(currentState, intC, intMR);
45            end
46
47        case ThrustControllerStates.Pressurizing
48            if P_c_set <= 0
49                % Transition Pressurizing to Closed
50                [u_o, u_f, currentState, intC, intMR] = ...
51                    updatePressurizingToClosed(P_c_set, P_c, r_m_set, ...
52                    m_d_o, m_d_f, intC, intMR);
53            elseif abs(P_c_set - P_c) < 0.1 * P_c_set
54                % Transition Pressurizing to Running
55                [u_o, u_f, currentState, intC, intMR] = ...
56                    updatePressurizingToRunning(P_c_set, P_c, r_m_set, ...
57                    m_d_o, m_d_f, intC, intMR);
58            else
59                % Remain in Pressurizing
60                [u_o, u_f, currentState, intC, intMR] = ...
61                    remainPressurizing(P_c_set, P_c, r_m_set, m_d_o, ...
62                    m_d_f, currentState, intC, intMR);
63            end
64
65        case ThrustControllerStates.Running
66            if P_c_set <= 0
67                % Transition Running to Closed
68                [u_o, u_f, currentState, intC, intMR] = ...
69                    updateRunningToClosed(P_c_set, P_c, r_m_set, m_d_o, ...
70                    m_d_f, intC, intMR);
71            else
72                % Remain in Running
73                [u_o, u_f, currentState, intC, intMR] = ...
74                    remainRunning(P_c_set, P_c, r_m_set, m_d_o, m_d_f, ...
75                    currentState, intC, intMR);
76            end
77    end
78 end
```

```
function [u_o, u_f, newState, intC, intMR] = remainClosed(oldState, intC, intMR)
    %#codegen
    % Controller transition function from Closed to Closed

    % Set new state
    newState = oldState;

    % Set output
    u_o = 0;
    u_f = 0;
end
```

```
function [u_o, u_f, newState, intC, intMR] = updateClosedToPressurizing(P_c_set,
    P_c, r_m_set, m_d_o, m_d_f, intC, intMR) %#codegen
    % Controller transition function from Closed to Pressurizing

    % Set new state
    newState = ThrustControllerStates.Pressurizing;

    % Set output
    [u_o, u_f, intC, intMR] = computeControl(P_c_set, P_c, r_m_set, m_d_o, m_d_f,
    intC, intMR);
    u_o = 0.3;
    u_f = 0.2;
end
```

```
function [u_o, u_f, newState, intC, intMR] = updatePressurizingToClosed(P_c_set,
    P_c, r_m_set, m_d_o, m_d_f, intC, intMR) %#codegen
    % Controller transition function from Pressurizing to Closed

    % Set new state
    newState = ThrustControllerStates.Closed;

    % Reset integrators
    % intC = 0; intMR = 0;

    % Set output
    u_o = 0;
    u_f = 0;
end
```

```
function [u_o, u_f, newState, intC, intMR] = remainPressurizing(P_c_set, P_c,
    r_m_set, m_d_o, m_d_f, oldState, intC, intMR) %#codegen
```

```
    % Controller transition function from Pressurizing to Pressurizing

    % Set new state
    newState = oldState;

    % Set output
    [u_o, u_f, intC, intMR] = computeControl(P_c_set, P_c, r_m_set, m_d_o, m_d_f,
    intC, intMR);
    u_o = 0.3;
    u_f = 0.2;
end
```

```
function [u_o, u_f, newState, intC, intMR] = updatePressurizingToRunning(P_c_set,
    P_c, r_m_set, m_d_o, m_d_f, intC, intMR) %#codegen
    % Controller transition function from Pressurizing to Running

    % Set new state
    newState = ThrustControllerStates.Running;

    % Set output
    [u_o, u_f, intC, intMR] = computeControl(P_c_set, P_c, r_m_set, m_d_o, m_d_f,
    intC, intMR);
end
```

```
function [u_o, u_f, newState, intC, intMR] = remainRunning(P_c_set, P_c, r_m_set,
    m_d_o, m_d_f, oldState, intC, intMR) %#codegen
    % Controller transition function from Running to Running

    % Set new state
    newState = oldState;

    % Set output
    [u_o, u_f, intC, intMR] = computeControl(P_c_set, P_c, r_m_set, m_d_o, m_d_f,
    intC, intMR);
end
```

```
function [u_o, u_f, newState, intC, intMR] = updateRunningToClosed(P_c_set, P_c,
    r_m_set, m_d_o, m_d_f, intC, intMR) %#codegen
    % Controller transition function from Running to Closed

    % Set new state
    newState = ThrustControllerStates.Closed;

```

```
    % Set output
    u_o = 0;
    u_f = 0;
end
```

```
function [u_o, u_f, intC, intMR] = computeControl(P_c_set, P_c, r_m_set, m_d_o,
    m_d_f, intC, intMR) %#codegen
% Control computation function

    % Gains and sample time
    Ts   = 0.001;
    KpC  = 1e-7;   KiC  = 1e-6;
    KpMR = -0.3;   KiMR = -3;

    % Chamber pressure
    eC   = P_c_set - P_c;
    intC = intC + eC * Ts;
    u_o  = KpC * eC + KiC * intC;

    % Mixture ratio
    if m_d_f == 0
        r_m = 0;
    else
        r_m = m_d_o / m_d_f;
    end
    eMR   = r_m_set - r_m;
    intMR = intMR + eMR * Ts;
    u_f   = KpMR * eMR + KiMR * intMR;
end
```

A.2. Verification Functions

```
function runCodeAnalyzer(autoEval)
    % Run Code Analyzer on MATLAB files in the directory
    %
    % DESCRIPTION
    %   This function looks for any MATLAB files in the directory and
    %   checks each for Code Analyzer issues. For each file with issues, it
    %   displays a message and passes the found issues to the function
    %   writeToCodeQuality. Optionally, it exits with exit code 1 if any
    %   issues were found.
    %
```

```
    % INPUTS
    %   autoEval - Option to activate automatic evaluation
    %
    % OUTPUTS
    %   none

    % Define function arguments and defaults
    arguments
        autoEval (1,1) logical = false
    end

    % Find all MATLAB files
    mFiles = dir('**/*.m');
    if isempty(mFiles)
        disp('Code Analyzer: No MATLAB files found.');
        exit(0);
    end

    % Initialize array to collect all issues
    issues = [];
    resultCounter = 1;

    % Run Code Analyzer analysis on files and display message
    issuesFound = false;
    for k = 1:numel(mFiles)
        filePath = fullfile(mFiles(k).folder, mFiles(k).name);
        messages = checkcode(filePath, '-id', '-struct');
        if ~isempty(messages)
            % Display issues
            disp(['Code Analyzer: Issues found in: ', filePath]);
            for i = 1:numel(messages)
                lineNum = messages(i).line;
                colRange = messages(i).column;
                msgText = messages(i).message;
                msgID = messages(i).id;
                fprintf('  Line %d (Columns %d-%d): %s\n', lineNum, colRange(1),
    colRange(end), msgText);

                % Create unique fingerprint
                fingerprint = sprintf('%s_%d', msgID, resultCounter);
                resultCounter = resultCounter + 1;

```

```
                % Append issue details to issues array
                issue = struct( ...
                    'description', msgText, ...
                    'check_name', msgID, ...
                    'fingerprint', fingerprint, ...
                    'severity', 'minor', ...
                    'location', struct( ...
                        'path', filePath, ...
                        'lines', struct('begin', lineNum) ...
                    ) ...
                );
                issues = [issues; issue];
            end
            issuesFound = true;
        end
    end

    if ~issuesFound
        disp('Code Analyzer: No issues found.');
    else
        writeToCodeQuality(issues, 'codeAnalyzerCodeQuality.json');
    end

    % Exit with non-zero exit code if issues were found
    if autoEval
        if issuesFound
            exit(1);
        else
            exit(0);
        end
    end
end
```

```
function writeToCodeQuality(issues, filename)
    % Write Code Analyzer results to Code Quality JSON file
    %
    % DESCRIPTION
    %   This function accepts a structure array of Code Analyzer issues
    %   and writes it to a JSON file.
    %
    % INPUTS
    %   issues      - structure array of Code Analyzer issues
    %   filename    - name of the created results file
```

```
    %
    % OUTPUTS
    %   none

    % Convert the structure array to JSON text
    jsonText = jsonencode(issues, 'PrettyPrint', true);

    % Write JSON text to the specified file
    fid = fopen(filename, 'w');
    if fid == -1
        error('Cannot open file %s for writing.', filename);
    end
    fwrite(fid, jsonText, 'char');
    fclose(fid);
end
```

```
function loadModel(model)
    % Load model
    %
    % DESCRIPTION
    %   This helper function adds the model directory to the MATLAB path
    %   and loads the specified model.
    %
    % INPUTS
    %   model   - name of the model to load
    %
    % OUTPUTS
    %   none

    % Add path
    addpath('models')

    % Load model
    load_system(model);
end
```

```
function checkIDs = getCheckIDs(option)
    % Get Model Advisor check IDs
    %
    % DESCRIPTION
    %   This function returns check IDs for a Model Advisor analysis as a
    %   cell array based on the option provided.
    %
```

```
8   % INPUTS
9   %   option          - one of the following:
10  %       all         - returns all available check IDs concatenated into one cell
    array
11  %       design      - returns the base Simulink checks
12  %       advisory    - returns the MathWorks Advisory Board guideline checks
13  %       integrity   - returns the High Integrity Systems Modeling guidelines
    checks
14  %       codegen     - returns the code generation checks
15  %       misra       - returns the MISRA compliance checks
16  %
17  % OUTPUTS
18  %   checkIDs        - cell array of check IDs
19  %
20  % NOTES
21  %   To get all check IDs see doc Simulink.ModelAdvisor:
22  %   ma = Simulink.ModelAdvisor.getModelAdvisor(model)
23  %   checks = ma.getCheckAll
24
25  % Define function arguments
26  arguments
27      option (1,1) string
28  end
29
30  % Define Simulink checks
31  designIDs = {
32      'mathworks.design.UnconnectedLinesPorts', ...                   % Identify
    unconnected lines, input ports, and output ports
33      'mathworks.design.RootInportSpec', ...                          % Check root
    model Inport block specifications
34      'mathworks.design.ModelRefSIMConfigCompliance', ...             % Check
    diagnostic settings ignored during accelerated model reference simulation
35      'mathworks.design.ParamTunabilityIgnored', ...                  % Check for
    parameter tunability information ignored for referenced models
36      'mathworks.design.ImplicitSignalResolution', ...                % Check for
    implicit signal resolution
37      'mathworks.design.OptBusVirtuality', ...                        % Check for
    optimal bus virtuality
38      'mathworks.design.CallslDataTypeAndScale', ...                  % Check for
    calls to slDataTypeAndScale()
39      'mathworks.design.DiscreteTimeIntegratorInitCondition', ...     % Check for
    Discrete-Time Integrator blocks with initial condition uncertainty
```

```
40      'mathworks.design.DisabledLibLinks', ...                    % Identify
    disabled library links
41      'mathworks.design.ParameterizedLibLinks', ...               % Identify
    parameterized library links
42      'mathworks.design.UnresolvedLibLinks', ...                  % Identify
    unresolved library links
43      'mathworks.design.CSStoVSSConvert', ...                     % Identify
    configurable subsystem blocks in the model for converting to variant subsystem
    blocks.
44      'mathworks.design.CheckForProperFcnCallUsage', ...          % Check
    usage of function-call connections
45      'mathworks.design.CheckMaskDisplayImageFormat', ...         % Check and
    update mask image display commands with unnecessary imread() function calls
46      'mathworks.design.CheckMaskRunInitFlag', ...                % Check and
    update mask to affirm icon drawing commands dependency on mask workspace
47      'mathworks.design.DiagnosticSFcn', ...                      % Runtime
    diagnostics for S-functions
48      'mathworks.design.DiagnosticDataStoreBlk', ...              % Check if
    Read/Write diagnostics are enabled for Data Store blocks
49      'mathworks.design.DataStoreMemoryBlkIssue', ...             % Check Data
    Store Memory blocks for multitasking, strong typing, and shadowing issues
50      'mathworks.design.SLXModelProperties', ...                  % Check
    Model History properties
51      'mathworks.design.SFuncAnalyzer', ...                       % Check
    S-functions in the model
52      'mathworks.design.CheckVirtualBusAcrossModelReferenceArgs', ... % Check for
    large number of function arguments from virtual bus across model reference
    boundary
53      'mathworks.design.ReplaceZOHDelayByRTB', ...                % Check
    Delay, Unit Delay and Zero-Order Hold blocks for rate transition
54      'mathworks.design.BusTreatedAsVector', ...                  % Check bus
    signals treated as vectors
55      'mathworks.design.DelayedFcnCallSubsys', ...                % Check for
    potentially delayed function-call block return values
56      'mathworks.design.OutputSignalSampleTime', ...              % Identify
    block output signals with continuous sample time and non-floating point data type
57      'mathworks.design.MergeBlkUsage', ...                       % Check
    usage of Merge blocks
58      'mathworks.design.InitParamOutportMergeBlk', ...            % Check
    usage of Outport blocks
59      'mathworks.design.DiscreteBlock', ...                       % Check
    usage of Discrete-Time Integrator blocks
```

```
60          'mathworks.design.ModelLevelMessages', ...                          % Check
    model settings for migration to simplified initialization mode
61          'mathworks.design.NonContSigDerivPort', ...                         % Check for
    non-continuous signals driving derivative ports
62          'mathworks.design.DataStoreBlkSampleTime', ...                      % Check data
    store block sample times for modeling errors
63          'mathworks.design.OrderingDataStoreAccess', ...                     % Check for
    potential ordering issues involving data store access
64          'mathworks.design.UnitMismatches', ...                              % Identify
    unit mismatches in the model
65          'mathworks.design.AutoUnitConversions', ...                         % Identify
    automatic unit conversions in the model
66          'mathworks.design.DisallowedUnitSystems', ...                       % Identify
    disallowed unit systems in the model
67          'mathworks.design.UndefinedUnits', ...                              % Identify
    undefined units in the model
68          'mathworks.design.AmbiguousUnits', ...                              % Identify
    ambiguous units in the model
69      };
70
71      % Excluded checks
72      % 'mathworks.design.StowawayDoubles', ...                               % Identify
    questionable operations for strict single-precision design (requires Coder
    license)
73      % 'mathworks.design.OptimizationSettings', ...                          % Check
    optimization settings (requires Coder license)
74      % 'mathworks.design.MismatchedBusParams', ...                           % Check
    structure parameter usage with bus signals (requires Coder license)
75      % 'mathworks.design.ReplaceEnvironmentControllerBlk', ...              % Identify
    Environment Controller blocks to be replaced with Variant Source blocks
    (relevant since R2021b)
76
77      % Define MAB compliance checks
78      mabIDs = {
79          'mathworks.maab.hd_0001', ...       % Check for prohibited sink blocks
80          'mathworks.maab.db_0142', ...       % Check whether block names appear below
    blocks
81          'mathworks.maab.db_0143', ...       % Check for mixing basic blocks and
    subsystems
82          'mathworks.maab.db_0110', ...       % Check usage of tunable parameters in
    blocks
```

```
83      'mathworks.maab.jc_0061', ...       % Check the display attributes of block
    names
84      'mathworks.maab.jc_0081', ...       % Check display for port blocks
85      'mathworks.maab.jc_0131', ...       % Check usage of Relational Operator
    blocks
86      'mathworks.maab.db_0140', ...       % Check for nondefault block attributes
87      'mathworks.maab.na_0008', ...       % Check signal line labels
88      'mathworks.maab.na_0009', ...       % Check for propagated signal labels
89      'mathworks.maab.na_0003', ...       % Check logical expressions in If blocks
90      'mathworks.maab.na_0004', ...       % Check for Simulink diagrams using
    nonstandard display attributes
91      'mathworks.maab.na_0034', ...       % Check input and output settings of
    MATLAB Functions
92      'mathworks.maab.na_0024', ...       % Check MATLAB code for global variables
93      'mathworks.maab.na_0039', ...       % Check use of Simulink in Stateflow
    charts
94      'mathworks.maab.na_0036', ...       % Check use of default variants
95      'mathworks.maab.na_0037', ...       % Check use of single variable variant
    conditionals
96      'mathworks.maab.na_0019', ...       % Check usage of restricted variable
    names
97      'mathworks.maab.na_0021', ...       % Check usage of character vector inside
    MATLAB Function block
98      'mathworks.maab.na_0022', ...       % Check usage of recommended patterns
    for Switch/Case statements
99      'mathworks.maab.na_0017', ...       % Check the number of function calls in
    MATLAB Function blocks
100     'mathworks.maab.himl_0003', ...     % Check MATLAB Function metrics (the two
    individual checks don't work in R2020a)
101     'mathworks.maab.jc_0011', ...       % Check Implement logic signals as
    Boolean data (vs. double)
102     'mathworks.maab.jc_0021', ...       % Check model diagnostic parameters
103     'mathworks.maab.na_0011', ...       % Check scope of From and Goto blocks
104     'mathworks.maab.jc_0141', ...       % Check usage of Switch blocks
105     'mathworks.maab.na_0031', ...       % Check usage of enumerated values
106     'mathworks.jmaab.jc_0627', ...      % Check usage of Discrete-Time
    Integrator block
107     'mathworks.jmaab.jc_0653', ...      % Check for avoiding algebraic loops
    between subsystems
108     'mathworks.jmaab.na_0020', ...      % Check for missing ports in Variant
    Subsystems
109     'mathworks.jmaab.jc_0624', ...      % Check for cascaded Unit Delay blocks
```

```
110         'mathworks.jmaab.ar_0001', ...      % Check file names
111         'mathworks.jmaab.ar_0002', ...      % Check folder names
112         'mathworks.jmaab.jc_0211', ...      % Check port block names
113         'mathworks.jmaab.jc_0201', ...      % Check subsystem names
114         'mathworks.jmaab.jc_0231', ...      % Check character usage in block names
115         'mathworks.jmaab.jc_0008', ...      % Check definition of signal labels
116         'mathworks.jmaab.jc_0009', ...      % Check Signal name propagation
117         'mathworks.jmaab.jc_0642', ...      % Check Signed Integer Division Rounding
    mode
118         'mathworks.jmaab.jc_0659', ...      % Check usage of Merge block
119         'mathworks.jmaab.jc_0110', ...      % Check block orientation
120         'mathworks.jmaab.jc_0222', ...      % Check character usage in signal names
    and bus names
121         'mathworks.jmaab.jc_0241', ...      % Check length of model file name
122         'mathworks.jmaab.jc_0242', ...      % Check length of folder name at every
    level of model path
123         'mathworks.jmaab.jc_0243', ...      % Check length of subsystem names
124         'mathworks.jmaab.jc_0244', ...      % Check length of Inport and Outport
    names
125         'mathworks.jmaab.jc_0245', ...      % Check length of signal and bus names
126         'mathworks.jmaab.jc_0247', ...      % Check length of block names
127         'mathworks.jmaab.jc_0604', ...      % Check if blocks are shaded in the model
128         'mathworks.jmaab.jc_0610', ...      % Check operator order of Product blocks
129         'mathworks.jmaab.jc_0621', ...      % Check icon shape of Logical Operator
    blocks
130         'mathworks.jmaab.jc_0645', ...      % Check if tunable block parameters are
    defined as named constants
131         'mathworks.jmaab.jc_0656', ...      % Check default/else case in Switch Case
    blocks and If blocks
132         'mathworks.jmaab.jc_0626', ...      % Check usage of Lookup Tables
133         'mathworks.jmaab.jc_0622', ...      % Check for parentheses in Fcn block
    expressions
134         'mathworks.jmaab.jc_0791', ...      % Check duplication of Simulink Data
    names
135         'mathworks.jmaab.jc_0603', ...      % Check Model Description
136         'mathworks.jmaab.jc_0806', ...      % Check diagnostic settings for
    incorrect calculation results
137         'mathworks.jmaab.jc_0651', ...      % Check output data type of operation
    blocks
138         'mathworks.jmaab.jc_0602', ...      % Check for consistency in model element
    names
139         'mathworks.jmaab.jc_0641', ...      % Check for sample time setting
```

```
140         'mathworks.jmaab.jc_0121', ...      % Check usage of Sum blocks
141         'mathworks.jmaab.db_0112', ...      % Check Indexing Mode
142         'mathworks.jmaab.db_0097', ...      % Check position of signal labels
143         'mathworks.jmaab.db_0042', ...      % Check position of Inport and Outport
    blocks
144         'mathworks.jmaab.jc_0161', ...      % Check for usage of Data Store Memory
    blocks
145         'mathworks.maab.db_0141', ...       % Check signal flow in model
146         'mathworks.jmaab.db_0146', ...      % Check position of conditional blocks
    and iterator blocks
147         'mathworks.jmaab.db_0032', ...      % Check signal line connections
148         'mathworks.maab.db_0081', ...       % Check for unconnected signal lines and
    blocks
149         'mathworks.jmaab.jc_0630', ...      % Check settings for data ports in
    Multiport Switch blocks
150         'mathworks.jmaab.jc_0650', ...      % Check input and output datatype for
    Switch blocks
151         'mathworks.jmaab.jc_0643', ...      % Check usage of fixed-point data type
    with non-zero bias
152         'mathworks.jmaab.jc_0611', ...      % Check signs of input signals in
    product blocks
153         'mathworks.jmaab.jc_0644', ...      % Check type setting by data objects
154         'mathworks.jmaab.jc_0628', ...      % Check usage of the Saturation blocks
155         'mathworks.jmaab.jc_0232', ...      % Check character usage in parameter
    names
156         'mathworks.jmaab.jc_0246', ...      % Check length of parameter names
157         'mathworks.jmaab.jc_0640', ...      % Check undefined initial output for
    conditional subsystems
158         'mathworks.jmaab.jc_0800', ...      % Check comparison of floating point
    types in Simulink
159         'mathworks.jmaab.jc_0792', ...      % Check unused data in Simulink Model
160         'mathworks.jmaab.na_0002', ...      % Check fundamental logical and
    numerical operations
161         'mathworks.jmaab.na_0010', ...      % Check usage of vector and bus signals
162         'mathworks.jmaab.jc_0171', ...      % Check connections between structural
    subsystems
163     };
164
165     % Excluded checks
166     % 'mathworks.maab.na_0016', ...   % Check lines of code in MATLAB Functions
    (might be replaced by 'mathworks.maab.himl_0003')
```

```
167     % 'mathworks.maab.na_0018', ...   % Check nested conditions in MATLAB Functions
        (might be replaced by 'mathworks.maab.himl_0003')
168     % 'mathworks.jmaab.jc_0623', ...  % Check usage of Memory and Unit Delay blocks
        (might cause issue)
169     % 'mathworks.jmaab.jc_0801', ...  % Check for use of C-style comment symbols
        (might cause issue)
170
171     % Define High Integrity System Modeling checks
172     hismIDs = {
173         'mathworks.hism.himl_0001', ...     % Check usage of standardized MATLAB
        function headers (might cause issue)
174         'mathworks.hism.himl_0002', ...     % Check for MATLAB Function interfaces
        with inherited properties (might cause issue)
175         'mathworks.hism.himl_0003', ...     % Check MATLAB Function metrics (might
        cause issue)
176         'mathworks.hism.himl_0004', ...     % Check MATLAB Code Analyzer messages
        (might cause issue)
177         'mathworks.hism.himl_0006', ...     % Check if/elseif/else patterns in
        MATLAB Function blocks (might cause issue)
178         'mathworks.hism.himl_0007', ...     % Check switch statements in MATLAB
        Function blocks (might cause issue)
179         'mathworks.hism.himl_0012', ...     % Check MATLAB functions not supported
        for code generation (might cause issue)
180         'mathworks.hism.hisl_0006', ...     % Check usage of While Iterator blocks
181         'mathworks.hism.hisl_0007', ...     % Check usage of For and While Iterator
        subsystems
182         'mathworks.hism.hisl_0020', ...     % Check for blocks not recommended for
        C/C++ production code deployment
183         'mathworks.hism.hisl_0021', ...     % Check for inconsistent vector indexing
        methods
184         'mathworks.hism.hisl_0023', ...     % Check usage of variant blocks
185         'mathworks.hism.hisl_0024', ...     % Check for root Inports with missing
        properties
186         'mathworks.hism.hisl_0031', ...     % Check model file name
187         'mathworks.hism.hisl_0033', ...     % Check usage of lookup table blocks
188         'mathworks.hism.hisl_0040', ...     % Check safety-related solver settings
        for simulation time
189         'mathworks.hism.hisl_0066', ...     % Check usage of Gain blocks
190         'mathworks.hism.hisl_0071', ...     % Check safety-related settings for
        hardware implementation
191         'mathworks.hism.hisl_0072', ...     % Check for parameter tunability ignored
        for referenced models
```

```
192         'mathworks.hism.hisl_0013', ...     % Check safety-related diagnostic
    settings for data store memory
193         'mathworks.hism.hisl_0036', ...     % Check safety-related diagnostic
    settings for saving
194         'mathworks.hism.hisl_0037', ...     % Check safety-related model referencing
    settings
195         'mathworks.hism.hisl_0041', ...     % Check safety-related solver settings
    for solver options
196         'mathworks.hism.hisl_0042', ...     % Check safety-related solver settings
    for tasking and sample-time
197         'mathworks.hism.hisl_0043', ...     % Check safety-related diagnostic
    settings for solvers
198         'mathworks.hism.hisl_0044', ...     % Check safety-related diagnostic
    settings for sample time
199         'mathworks.hism.hisl_0045', ...     % Check safety-related optimization
    settings for logic signals
200         'mathworks.hism.hisl_0046', ...     % Check safety-related block reduction
    optimization settings
201         'mathworks.hism.hisl_0048', ...     % Check safety-related optimization
    settings for application lifespan
202         'mathworks.hism.hisl_0052', ...     % Check safety-related optimization
    settings for data initialization
203         'mathworks.hism.hisl_0053', ...     % Check safety-related optimization
    settings for data type conversions
204         'mathworks.hism.hisl_0054', ...     % Check safety-related optimization
    settings for division arithmetic exceptions
205         'mathworks.hism.hisl_0056', ...     % Check safety-related optimization
    settings for specified minimum and maximum values
206         'mathworks.hism.hisl_0038', ...     % Check safety-related code generation
    settings for comments
207         'mathworks.hism.hisl_0039', ...     % Check safety-related code generation
    interface settings
208         'mathworks.hism.hisl_0047', ...     % Check safety-related code generation
    settings for code style
209         'mathworks.hism.hisl_0049', ...     % Check safety-related code generation
    identifier settings
210         'mathworks.hism.hisl_0301', ...     % Check safety-related diagnostic
    settings for compatibility
211         'mathworks.hism.hisl_0302', ...     % Check safety-related diagnostic
    settings for parameters
212         'mathworks.hism.hisl_0303', ...     % Check safety-related diagnostic
    settings for Merge blocks
```

```
213         'mathworks.hism.hisl_0304', ...     % Check safety-related diagnostic
    settings for model initialization
214         'mathworks.hism.hisl_0305', ...     % Check safety-related diagnostic
    settings for data used for debugging
215         'mathworks.hism.hisl_0306', ...     % Check safety-related diagnostic
    settings for signal connectivity
216         'mathworks.hism.hisl_0307', ...     % Check safety-related diagnostic
    settings for bus connectivity
217         'mathworks.hism.hisl_0308', ...     % Check safety-related diagnostic
    settings that apply to function-call connectivity
218         'mathworks.hism.hisl_0309', ...     % Check safety-related diagnostic
    settings for type conversions
219         'mathworks.hism.hisl_0310', ...     % Check safety-related diagnostic
    settings for model referencing
220         'mathworks.hism.hisl_0314', ...     % Check safety-related diagnostic
    settings for signal data
221         'mathworks.hism.hisl_0074', ...     % Check safety-related diagnostic
    settings for variants
222         'mathworks.hism.himl_0011', ...     % Check type and size of condition
    expressions
223         'mathworks.hism.himl_0008', ...     % Check usage of relational operators in
    MATLAB Function blocks (might cause issue)
224         'mathworks.hism.himl_0010', ...     % Check usage of logical operators and
    functions in MATLAB Function blocks (might cause issue)
225         'mathworks.hism.himl_0013', ...     % Metrics for generated code complexity
    (might cause issue)
226         'mathworks.hism.hisl_0001', ...     % Check usage of Abs blocks
227         'mathworks.hism.hisl_0008', ...     % Check usage of For Iterator blocks
228         'mathworks.hism.hisl_0010', ...     % Check usage of If blocks and If Action
    Subsystem blocks
229         'mathworks.hism.hisl_0011', ...     % Check usage of Switch Case blocks and
    Switch Case Action Subsystem blocks
230         'mathworks.hism.hisl_0012', ...     % Check usage of conditionally executed
    subsystems
231         'mathworks.hism.hisl_0016', ...     % Check relational comparisons on
    floating-point signals
232         'mathworks.hism.hisl_0017', ...     % Check usage of Relational Operator
    blocks
233         'mathworks.hism.hisl_0018', ...     % Check usage of Logical Operator blocks
234         'mathworks.hism.hisl_0019', ...     % Check usage of bitwise operations
235         'mathworks.hism.hisl_0015', ...     % Check usage of Merge blocks
```

```
236         'mathworks.hism.hisl_0022', ...     % Check data types for blocks with index
    signals
237         'mathworks.hism.hisl_0025', ...     % Check for root Inports with missing
    range definitions
238         'mathworks.hism.hisl_0026', ...     % Check for root Outports with missing
    range definitions
239         'mathworks.hism.hisl_0029', ...     % Check usage of Assignment blocks
240         'mathworks.hism.hisl_0032', ...     % Check model object names
241         'mathworks.hism.hisl_0034', ...     % Check usage of Signal Routing blocks
242         'mathworks.hism.hisl_0063', ...     % Check for length of user-defined
    object names
243         'mathworks.hism.hisl_0102', ...     % Check data type of loop control
    variables
244         'mathworks.hism.hisl_0073', ...     % Check usage of bit-shift operations
245         'mathworks.hism.hisl_0028', ...     % Check usage of Reciprocal Sqrt blocks
    (might cause issue)
246         'mathworks.hism.hisl_0067' ...      % Check for divide-by-zero calculations
    (might cause issue)
247     };
248
249     % Define code generation checks
250     codegenIDs = {
251         'mathworks.codegen.PCGSupport', ...                                 % Check
    for blocks not recommended for C/C++ production code deployment
252         'mathworks.codegen.EfficientTunableParamExpr', ...                  % Check
    configuration parameters for generation of inefficient saturation code
253         'mathworks.codegen.LUTRangeCheckCode', ...                          %
    Identify lookup table blocks that generate expensive out-of-range checking code
254         'mathworks.codegen.LogicBlockUseNonBooleanOutput', ...              % Check
    output types of logic blocks
255         'mathworks.codegen.HWImplementation', ...                           % Check
    the hardware implementation
256         'mathworks.codegen.SWEnvironmentSpec', ...                          %
    Identify questionable software environment specifications
257         'mathworks.codegen.CodeInstrumentation', ...                        %
    Identify questionable code instrumentation (data I/O)
258         'mathworks.codegen.UseRowMajorAlgorithm', ...                       %
    Identify blocks generating inefficient algorithms
259         'mathworks.codegen.QuestionableSubsysSetting', ...                  %
    Identify questionable subsystem settings
260         'mathworks.codegen.RowMajorCodeGenSupport', ...                     % Check
    for blocks not supported for row-major code generation
```

```
261         'mathworks.codegen.RowMajorUnsetSFunction', ...                         %
    Identify TLC S-Functions with unset array layout
262         'mathworks.codegen.BlockSpecificQuestionableFxptOperations', ...    %
    Identify blocks that generate expensive fixed-point and saturation code
263         'mathworks.codegen.QuestionableFxptOperations', ...                 %
    Identify questionable fixed-point operations
264         'mathworks.codegen.ExpensiveSaturationRoundingCode', ...            %
    Identify blocks that generate expensive rounding code
265         'mathworks.codegen.BlockNames', ...                                 % Check
    block names
266         'mathworks.codegen.cgsl_0101', ...                                  %
    Identify blocks using one-based indexing
267         'mathworks.codegen.SolverCodeGen', ...                              % Check
    solver for code generation
268         'mathworks.codegen.codeGenSupport', ...                             % Check
    for blocks not supported by code generation
269         'mathworks.codegen.MdlrefConfigMismatch', ...                       % Check
    for model reference configuration mismatch
270         'mathworks.codegen.ModelRefRTWConfigCompliance', ...                % Check
    for code generation identifier formats used for model reference
271         'mathworks.codegen.SubsysCodeReuse', ...                            % Check
    reuse of subsystem code
272         'mathworks.codegen.SampleTimesTaskingMode', ...                     % Check
    sample times and tasking mode
273         'mathworks.codegen.ConstraintsTunableParam', ...                    % Check
    for blocks that have constraints on tunable parameters
274         'mathworks.codegen.QuestionableBlks', ...
275         'mathworks.codegen.CodeGenSanity', ...
276         'mathworks.codegen.checkEnableMemcpy', ...
277         'mathworks.codegen.toolchainInfoUpgradeAdvisor.check', ...          % Check
    and update model to use toolchain approach to build generated code
278         'mathworks.codegen.codertarget.check', ...                          % Check
    and update embedded target model to use ert.tlc system target file
279         'mathworks.design.datastoresimrtwcmp', ...                          % Check
    for relative execution order change for Data Store Read and Data Store Write
    blocks
280     };
281
282     % Define MISRA compliance checks
283     misraIDs = {
284         'mathworks.misra.CodeGenSettings', ...                      % Check
    configuration parameters for MISRA C:2012
```

```
285         'mathworks.misra.BlkSupport', ...                       % Check for
    blocks not recommended for MISRA C:2012
286         'mathworks.misra.BlockNames', ...                       % Check for
    unsupported block names
287         'mathworks.misra.AssignmentBlocks', ...                 % Check usage of
    Assignment blocks
288         'mathworks.misra.SwitchDefault', ...                    % Check for
    switch case expressions without a default case
289         'mathworks.misra.AutosarReceiverInterface', ...         % Check for
    missing error ports for AUTOSAR receiver interfaces
290         'mathworks.misra.BusElementNames', ...                  % Check bus
    object names that are used as bus element names
291         'mathworks.misra.ModelFunctionInterface', ...           % Check for
    missing const qualifiers in model functions
292         'mathworks.misra.CompliantCGIRConstructions', ...       % Check for
    bitwise operations on signed integers
293         'mathworks.misra.RecursionCompliance', ...              % Check for
    recursive function calls
294         'mathworks.misra.CompareFloatEquality', ...             % Check for
    equality and inequality operations on floating-point values
295         'mathworks.misra.IntegerWordLengths', ...               % Check integer
    word length
296         'mathworks.security.CodeGenSettings', ...               % Check
    configuration parameters for secure coding standards
297         'mathworks.security.BlockSupport', ...                  % Check for
    blocks not recommended for secure coding standards
298     };
299
300     % Determine which cell arrays to concatenate and return
301     switch lower(option)
302         case 'all'
303             checkIDs = [designIDs, maabIDs, hismIDs, codegenIDs, misraIDs];
304         case 'design'
305             checkIDs = designIDs;
306         case 'advisory'
307             checkIDs = mabIDs;
308         case 'integrity'
309             checkIDs = hismIDs;
310         case 'codegen'
311             checkIDs = codegenIDs;
312         case 'misra'
313             checkIDs = misraIDs;
```

```
        otherwise
            error('Invalid option.');
    end
end
```

```
function runModelAdvisor(model, checkIDs, autoEval, artifactDir)
    % Run Model Advisor checks on the specified model
    %
    % DESCRIPTION
    %   This function runs a specified list of Model Advisor checks
    %   programmatically. It generates an HTML report and a JUnit-compatible
    %   XML results file. It exits with a non-zero code if any checks have
    %   failed and automatic evaluation has been activated.
    %
    % INPUTS
    %   model       - Name of the Simulink model to analyze
    %   checkIDs    - Cell array of Model Advisor check IDs to run
    %   artifactDir - Directory to save the generated report
    %   autoEval    - Option to activate automatic evaluation
    %
    % OUTPUTS
    %   none

    % Define function arguments and defaults
    arguments
        model
        checkIDs
        autoEval (1,1) logical = false
        artifactDir (1,1) string = fullfile(pwd, 'artifacts')
    end

    % Create artifact directory if necessary
    if ~exist(artifactDir, 'dir')
        mkdir(artifactDir);
    end

    % Define the report format and path
    reportFormat = 'html';
    reportName = 'ModelAdvisorReport';
    xmlReportName = 'ModelAdvisorReport.xml';
    xmlReportPath = fullfile(artifactDir, xmlReportName);

    % Run Model Advisor checks
```

```
    checkResult = ModelAdvisor.run(model, checkIDs, ...
        'DisplayResults', 'Details', ...
        'ReportFormat', reportFormat, ...
        'ReportPath', artifactDir, ...
        'ReportName', reportName);

    % Convert results to JUnit XML format
    convertToXML(checkResult, xmlReportPath);

    % Determine exit code based on check results if activated
    if autoEval
        % Extract SystemResult object
        systemResult = checkResult{1};

        % Retrieve array of individual check results and preallocate
        checkObjs = systemResult.CheckResultObjs;
        failFlags = false(1, length(checkObjs));

        % Populate failFlags array
        for i = 1:length(checkObjs)
            failFlags(i) = strcmp(checkObjs(i).status, 'Fail');
        end

        % Display a message indicating completion
        if any(failFlags)
            disp('Model Advisor: Some checks failed. Exiting with error code.');
            exit(1);
        else
            disp('Model Advisor: All checks passed or warnings only.');
            exit(0);
        end
    end
end

function convertToXML(checkResult, xmlReportPath)
    % Convert check results to JUnit formatted XML file
    %
    % DESCRIPTION:
    %   This function converts the Model Advisor check results into a
    %   JUnit-compatible XML report.
    %
    % INPUTS:
    %   checkResult   - Cell array containing one SystemResult object
```

```
10      %   xmlReportPath - Full path where the XML report is to be saved
11      %
12      % OUTPUTS:
13      %   none
14
15      % Extract SystemResult object from check results cell array
16      systemResult = checkResult{1};
17
18      % Get array of individual check results from SystemResult
19      checkObjs = systemResult.CheckResultObjs;
20
21      % Initialize XML document
22      docNode = com.mathworks.xml.XMLUtils.createDocument('testsuites');
23      testsuites = docNode.getDocumentElement;
24
25      % Create a testsuite element with appropriate attributes
26      testsuite = docNode.createElement('testsuite');
27      testsuite.setAttribute('name', 'Model Advisor Checks');
28      testsuite.setAttribute('tests', num2str(length(checkObjs)));
29      testsuites.appendChild(testsuite);
30
31      % Iterate over each check result
32      for i = 1:length(checkObjs)
33
34          % Create test case element and set attributes
35          testCase = docNode.createElement('testcase');
36          testCase.setAttribute('classname', 'Model Advisor Checks');
37          testCase.setAttribute('name', checkObjs(i).checkName);
38          message = 'See Model Advisor report for details.';
39
40          % If check failed, add a failure element with message
41          if strcmp(checkObjs(i).status, 'Fail')
42              failure = docNode.createElement('failure');
43              failure.setAttribute('message', ['Check failed. ' message]);
44              testCase.appendChild(failure);
45
46          % If check returned a warning, add a systemOut element with message
47          elseif strcmp(checkObjs(i).status, 'Warn')
48              systemOut = docNode.createElement('system-out');
49              systemOut.appendChild(docNode.createTextNode(['Check returned a warning.
    ' message]));
50              testCase.appendChild(systemOut);
```

```
        end

        % Append test case element to testsuite
        testsuite.appendChild(testCase);
    end

    % Write XML document to file
    xmlwrite(xmlReportPath, docNode);
end
```

```
function metricIDs = getMetricIDs(option)
    % Get metric IDs
    %
    % DESCRIPTION
    %   This function returns metric IDs as a cell array based on the
    %   option provided.
    %
    % INPUTS
    %   option              - one of the following:
    %       all             - returns all available metric IDs
    %       size            - returns only size metrics
    %       architecture    - returns only architecture metrics
    %       compliance      - returns only compliance metrics
    %       readability     - returns only readability metrics
    %
    % OUTPUTS
    %   metricsList - cell array of metric IDs

    % Define function arguments
    arguments
        option (1,1) string
    end

    % Define size metrics
    sizeMetrics = {'mathworks.metrics.SimulinkBlockCount',...   % Calculates the
    number of blocks in the model
        'mathworks.metrics.SubSystemCount',...                  % Calculates the
    number of subsystems in the model
        'mathworks.metrics.LibraryLinkCount',...                % Calculates the
    number of library-linked blocks in the model
        'mathworks.metrics.MatlabLOCCount',...                  % Calculates the
    number of effective lines of MATLAB code
```

```
29      'mathworks.metrics.SubSystemDepth',...                  % Calculates the
    subsystem depth of the model
30      'mathworks.metrics.IOCount',...                         % Calculates the
    number of inputs and outputs in your model
31      'mathworks.metrics.ExplicitIOCount',...                 % Calculates the
    number of inputs and outputs in your model
32      'mathworks.metrics.FileCount',...                       % Calculates the
    number of model and library files
33      'mathworks.metrics.MatlabFunctionCount',...             % Calculates the
    number of MATLAB Function blocks in your model
34      'mathworks.metrics.ModelFileCount',...                  % Calculates the
    number of model files
35      'mathworks.metrics.ParameterCount'};                    % Calculates the
    number of instances of data objects that parameterize the behavior of a model
36
37  % Define architecture metrics
38  architectureMetrics = {'mathworks.metrics.CyclomaticComplexity',...     %
    Calculates the cyclomatic complexity of the model
39      'mathworks.metrics.CloneContent',...                                %
    Calculates the fraction of total number of subcomponents that are clones
40      'mathworks.metrics.CloneDetection',...                              %
    Calculates the number of clones in components across the model hierarchy
41      'mathworks.metrics.LibraryContent'};                                %
    Calculates the fraction of total number of components that are linked library
    blocks
42
43  % Define compliance metrics
44  complianceMetrics = {'mathworks.metrics.MatlabCodeAnalyzerWarnings',... %
    Determines warnings for MATLAB code blocks in your model
45      'mathworks.metrics.DiagnosticWarningsCount',...                     %
    Calculates the number of diagnostic warnings reported
46      'mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178',...      %
    Returns the fraction of checks the model passes from Model Advisor
    DO-178C/DO-331 Standards
47      'mathworks.metrics.ModelAdvisorCheckCompliance.maab'};              %
    Returns the fraction of checks the model passes from Model Advisor MAB Standard
48
49  % Define readability metrics
50  readabilityMetrics = {'mathworks.metrics.DescriptiveBlockNames',...     %
    Determines nondescriptive Inport, Outport, and Subsystem block names
51      'mathworks.metrics.LayerSeparation'};                               %
    Calculates the data and structure layer separation
```

```
52
53      % Determine which cell arrays to concatenate
54      switch lower(option)
55          case 'all'
56              metricIDs = [sizeMetrics, architectureMetrics, complianceMetrics,
        readabilityMetrics];
57          case 'size'
58              metricIDs = sizeMetrics;
59          case 'architecture'
60              metricIDs = architectureMetrics;
61          case 'compliance'
62              metricIDs = complianceMetrics;
63          case 'readability'
64              metricIDs = readabilityMetrics;
65          otherwise
66              error('Invalid option.');
67      end
68  end
```

```
1   function collectModelMetrics(model, metricIDs, artifactDir)
2       % Collect model metrics of the the specified model
3       %
4       % DESCRIPTION
5       %   This function collects metrics of a model specified in a list of
6       %   metric IDs.
7       %
8       % INPUTS
9       %   model       - Name of the Simulink model to analyze
10      %   metricsList - Cell array of model metrics to calculate
11      %   artifactDir - Directory to save the generated report
12      %
13      % OUTPUTS
14      %   none
15
16      % Define function arguments and defaults
17      arguments
18          model
19          metricIDs
20          artifactDir (1,1) string = fullfile(pwd, 'artifacts')
21      end
22
23      % Create artifact directory if necessary
24      if ~exist(artifactDir, 'dir')
```

```
        mkdir(artifactDir);
    end

    % Initialize the metric engine
    metricEngine = slmetric.Engine();
    setAnalysisRoot(metricEngine, 'Root', model);

    % Execute the metrics collection
    execute(metricEngine, metricIDs);

    % Retrieve and display model metrics
    res = getMetrics(metricEngine, metricIDs);
    metricData = {'MetricID', 'ComponentPath', 'Value'};

    % Loop through each metric and display the results
    cnt = 1;
    for n = 1:length(res)
        if res(n).Status == 0
            results = res(n).Results;
            for m = 1:length(results)
                disp(['MetricID: ', results(m).MetricID]);
                disp(['  ComponentPath: ', results(m).ComponentPath]);
                disp(['  Value: ', num2str(results(m).Value)]);
                metricData{cnt+1, 1} = results(m).MetricID;
                metricData{cnt+1, 2} = results(m).ComponentPath;
                metricData{cnt+1, 3} = results(m).Value;
                cnt = cnt + 1;
            end
        else
            disp(['No results for: ', res(n).MetricID]);
        end
        disp(' ');
    end

    % Export the metrics to an XML file
    filename = 'MetricResults.xml';
    exportMetrics(metricEngine, filename, artifactDir);

    disp('Simulink Check: Model Metrics collection completed.');
end

function runDesignVerifier(model, artifactDir)
    % Run Simulink Design Verifier in Design Error Detection mode
```

```
3     %
4     % DESCRIPTION
5     %   This function configures Simulink Design Verifier for design error
6     %   detection, runs the analysis on the specified model and saves a report
7     %   with data file.
8     %
9     % INPUTS
10    %   model       - Name of the Simulink model to analyze
11    %   artifactDir - Directory to save the generated report
12    %
13    % OUTPUTS
14    %   none
15
16    % Define function arguments and defaults
17    arguments
18        model
19        artifactDir (1,1) string = fullfile(pwd, 'artifacts')
20    end
21
22    % Create artifact directory if necessary
23    if ~exist(artifactDir, 'dir')
24        mkdir(artifactDir);
25    end
26
27    % Configure Simulink Design Verifier for design error detection
28    sldvOptions = sldvoptions;
29    sldvOptions.Mode = 'DesignErrorDetection';
30    sldvOptions.DetectBlockInputRangeViolations = 'off';
31    sldvOptions.DetectDeadLogic = 'off';
32    sldvOptions.DetectDivisionByZero = 'on';
33    sldvOptions.DetectInfNaN = 'off';
34    sldvOptions.DetectIntegerOverflow = 'off';
35    sldvOptions.DetectOutOfBounds = 'off';
36    sldvOptions.DetectSubnormal = 'off';
37    sldvOptions.SaveReport = 'on';
38
39    % Run Design Verifier analysis
40    [status, files, ~] = sldvrun(model, sldvOptions);
41
42    % Handle the status of the analysis and save results
43    switch status
44        case -1
```

```
            disp('Design Verifier: Analysis exceeded the maximum processing time.');
        case 0
            disp('Design Verifier: Error occurred during design error detection.');
        case 1
            disp('Design Verifier: Design error detection completed successfully.');
            % Save the generated report
            copyfile(files.Report, fullfile(artifactDir,
    'DesignVerifierReport.html'));
            % Save the data file
            copyfile(files.DataFile, fullfile(artifactDir,
    'DesignVerifierData.sldv'));
    end
end
```

```
function runTests(artifactDir)
% Run tests with Simulink Test
%
% DESCRIPTION
    %   This function runs predefined test cases with Simulink Test.
    %
    % INPUTS
    %   artifactDir - Directory to save the generated report
    %
    % OUTPUTS
    %   none

    % Define function arguments and defaults
    arguments
        artifactDir (1,1) string = fullfile(pwd, 'artifacts')
    end

    % Create artifact directory if necessary
    if ~exist(artifactDir, 'dir')
        mkdir(artifactDir);
    end

    % Open the test file
    fileName = 'ControllerTests.mldatx';
    filePath = fullfile(pwd, 'tests', fileName);
    sltest.testmanager.load(filePath);

    % Create test suite from test file
    import matlab.unittest.TestSuite
```

```
30      suite = testsuite(filePath);
31
32      % Create test runner
33      import matlab.unittest.TestRunner
34      runner = TestRunner.withNoPlugins;
35
36      % Add plugin to produce MATLAB Test Report
37      import matlab.unittest.plugins.TestReportPlugin
38      pdfFile = fullfile(artifactDir, 'TestReport.pdf');
39      trp = TestReportPlugin.producingPDF(pdfFile);
40      addPlugin(runner,trp)
41
42      % Add plugin to add Test Manager results to Test Report
43      import sltest.plugins.TestManagerResultsPlugin
44      tmr = TestManagerResultsPlugin;
45      addPlugin(runner,tmr)
46
47      % Add plugin to create XML results file
48      import matlab.unittest.plugins.XMLPlugin
49      resfile = fullfile(artifactDir, 'TestResults.xml');
50      plugin = XMLPlugin.producingJUnitFormat(resfile);
51      addPlugin(runner,plugin)
52
53      % Set coverage metrics to collect
54      import sltest.plugins.coverage.CoverageMetrics
55      cmet = CoverageMetrics('Decision',true);
56
57      % Set coverage report properties
58      import sltest.plugins.coverage.ModelCoverageReport
59      import matlab.unittest.plugins.codecoverage.CoberturaFormat
60      rptfile = fullfile(artifactDir, 'TestCoverage.xml');
61      rpt = CoberturaFormat(rptfile);
62
63      % Create model coverage plugin
64      import sltest.plugins.ModelCoveragePlugin
65      mcp = ModelCoveragePlugin('Collecting',cmet,'Producing',rpt);
66      addPlugin(runner,mcp)
67
68      % Run the test
69      result = run(runner,suite);
70
71  end
```

```
function setConfiguration(model)
    % Set configuration for C++ code generation
    %
    % DESCRIPTION
    %   This function configures a specified Simulink model for C++ code
    %   generation with Embedded Coder.
    %
    % INPUTS
    %   model - Name of the Simulink model to configure
    %
    % OUTPUTS
    %   none
    %
    % NOTE
    %   The configuration is taken from the configuration parameters obtained
    %   from the file 'CodeGen.mat' using the script 'getParameters.m'.

    % Set parameters for C++ code generation
    set_param(model, 'SystemTargetFile', 'ert.tlc');            % System target file
    set_param(model, 'TargetLang', 'C++');                      % Select code
    generation language
    set_param(model, 'GenCodeOnly', 'on');                      % Do not execute
    makefile when generating code
    set_param(model, 'TargetLangStandard', 'C++03 (ISO)');      % Language standard
    set_param(model, 'PackageGeneratedCodeAndArtifacts', 'on'); % Automatically run
    packNGo after the build is complete
    set_param(model, 'BuildConfiguration', 'Faster Runs');      % Choose a build
    configuration defined by the toolchain

    % Optimization
    set_param(model, 'InstructionSetExtensions', 'None');       % Leverage target
    hardware instruction set extensions
    set_param(model, 'InlineInvariantSignals', 'on');           % Precompute and
    inline the values of invariant signals
    set_param(model, 'EfficientFloat2IntCast', 'on');           % Remove code from
    floating-point to integer conversions that wraps out-of-range values

    % Report
    set_param(model, 'GenerateReport', 'on');                   % Create code
    generation report
    set_param(model, 'LaunchReport', 'on');                     % Open report
    automatically
```

```
34
35      % Comments
36      set_param(model, 'ReqsInCode', 'on');                       % Insert entered
        requirements into the generated code as a comment
37      set_param(model, 'MATLABFcnDesc', 'on');                    % Insert MATLAB user
        comments into the generated code as comments
38
39      % Identifiers
40      set_param(model, 'MangleLength', 4);                        % Minimum mangle
        length
41
42      % Interface
43      set_param(model, 'SupportComplex', 'off');                  % Do not support
        complex numbers
44      set_param(model, 'SupportAbsoluteTime', 'off');             % Do not support
        absolute time
45      set_param(model, 'SupportContinuousTime', 'off');           % Do not support
        continuous time
46
47      % Code Style
48      set_param(model, 'ParenthesesLevel', 'Maximum');            % Specify the level
        of parenthesization in the code
49      set_param(model, 'PreserveExpressionOrder', 'on');          % Preserve operand
        order in expression
50      set_param(model, 'EnableSignedLeftShifts', 'off');          % Replace
        multiplications by powers of two with signed bitwise shifts
51      set_param(model, 'EnableSignedRightShifts', 'off');         % Allow right shifts
        on signed integers
52      set_param(model, 'CastingMode', 'Standards');               % Set casting mode
53  end
```

```
1   function generateCode(model)
2       % Generate C++ Code from Simulink model
3       %
4       % DESCRIPTION
5       %   This function configures a specified Simulink model for C++ code
6       %   generation with Embedded Coder, generates code from that model and
7       %   extends the archive with a Polyspace options file.
8       %
9       % INPUTS
10      %   model - Name of the Simulink model to configure and generate code from
11      %
12      % OUTPUTS
```

```
    %   none
    % Set configuration parameters
    setConfiguration(model);
    % Generate the code
    slbuild(model);
    % Generate and package Polyspace options files
    polyspacePackNGo(model);
    % Display a message confirming code generation completion
    disp('Embedded Coder: C++ code generation completed.');
end
```

```
function integrated = isPolyspaceIntegrated()
    % Check if Polyspace is integrated with MATLAB
    %
    % DESCRIPTION
    %   This helper function checks for Polyspace integration by verifying
    %   the existence of Polyspace functions within MATLAB.
    %
    % INPUTS
    %   none
    %
    % OUTPUTS
    %   integrated  - true if integrated, otherwise false

    integrated = exist('pslinkoptions', 'file') && exist('pslinkrun', 'file');
    if integrated
        disp('Polyspace is integrated with MATLAB.');
    else
        disp('Polyspace is not integrated with MATLAB.');
    end
end
```

```
function runPolyspaceBugFinder(model, artifactDir)
    % Configure and run Polyspace Bug Finder analysis
    %
    % DESCRIPTION
    %   This function configures Polyspace to run a Bug Finder analysis on
    %   a specified model. It uses a polyspace.Project object for
    %   configuration. There are different options available for defect and
```

```
8   %   compliance checking.
9   %
10  % INPUTS
11  %   model       - Name of the Simulink model to analyze
12  %   artifactDir - Directory to save the generated report
13  %
14  % OUTPUTS
15  %   none
16
17  % Define function arguments and defaults
18  arguments
19      model
20      artifactDir (1,1) string = fullfile(pwd, 'artifacts', 'bugFinder')
21  end
22
23  % Create artifact directory if necessary
24  if ~exist(artifactDir, 'dir')
25      mkdir(artifactDir);
26  end
27
28  % Create Polyspace Project object
29  proj = polyspace.Project;
30
31  % Associate project configuration with model specific information
32  proj.Configuration = polyspace.ModelLinkOptions(model);
33
34  % Configure the analysis
35  proj.Configuration.ResultsDir = artifactDir;
36
37  % Extend defect checking
38  proj.Configuration.BugFinderAnalysis.CheckersPreset = 'all';
39
40  % Enable code metric calculation
41  proj.Configuration.CodingRulesCodeMetrics.CodeMetrics = true;
42
43  % Enable Guideline checkers
44  proj.Configuration.CodingRulesCodeMetrics.EnableGuidelines = true;
45  proj.Configuration.CodingRulesCodeMetrics.Guidelines = 'all';
46
47  % Enable and extend MISRA C++:2008 checking
48  proj.Configuration.CodingRulesCodeMetrics.EnableMisraCpp = true;
49  proj.Configuration.CodingRulesCodeMetrics.MisraCppSubset = 'all-rules';
```

```
50
51      % Configure report generation
52      proj.Configuration.MergedReporting.EnableReportGeneration = true;
53      proj.Configuration.MergedReporting.ReportOutputFormat = 'HTML';
54      proj.Configuration.MergedReporting.BugFinderReportTemplate = 'BugFinder';
55
56      % Run analysis
57      bfStatus = run(proj, 'bugFinder');
58
59      % Open results
60      resultsFile = fullfile(artifactDir,'ps_results.psbf');
61      polyspaceBugFinder(resultsFile)
62  end
```

```
1   function runPolyspaceCodeProver(model, artifactDir)
2       % Configure and run Polyspace Code Prover analysis
3       %
4       % DESCRIPTION
5       %   This function configures Polyspace to run a Code Prover analysis on
6       %   a specified model. It uses a polyspace.Project object for
7       %   configuration.
8       %
9       % INPUTS
10      %   model       - Name of the Simulink model to analyze
11      %   artifactDir - Directory to save the generated report
12      %
13      % OUTPUTS
14      %   none
15
16      % Define function arguments and defaults
17      arguments
18          model
19          artifactDir (1,1) string = fullfile(pwd, 'artifacts', 'codeProver')
20      end
21
22      % Create artifact directory if necessary
23      if ~exist(artifactDir, 'dir')
24          mkdir(artifactDir);
25      end
26
27      % Create Polyspace Project object
28      proj = polyspace.Project;
29
```

```
    % Associate project configuration with model specific information
    proj.Configuration = polyspace.ModelLinkOptions(model);

    % Configure the analysis
    proj.Configuration.ResultsDir = artifactDir;

    % Configure report generation
    proj.Configuration.MergedReporting.EnableReportGeneration = true;
    proj.Configuration.MergedReporting.ReportOutputFormat = 'HTML';
    proj.Configuration.MergedReporting.CodeProverReportTemplate = 'Developer';

    % Run analysis
    cpStatus = run(proj, 'codeProver');

    % Open results
    resultsFile = fullfile(artifactDir,'ps_results.pscp');
    polyspaceCodeProver(resultsFile)
end
```

```
function generatePolyspaceReport(artifactDir)
    % Generate report from Polyspace analysis results
    %
    % DESCRIPTION
    %   This function uses the more customizable system command to generate
    %   a Polyspace report from a Polyspace results file. A Polyspace
    %   analysis has to be run beforehand to obtain that results file.
    %
    % INPUTS
    %   artifactDir - Directory to save the analysis report
    %
    % OUTPUTS
    %   none

    % Define parts of the command
    templatePath = 'C:\Program
    Files\Polyspace\R2021b\toolbox\polyspace\psrptgen\templates\bug_finder\BugFinder.rpt';
    outputName =
    'C:\Users\kays_ph\Documents\MATLAB\ThrustController\artifacts\PolyspaceReport.html';
    format = 'html';

    % Concatenate the full command
    command = ['polyspace-report-generator -template "', templatePath, ...
               '" -output-name "', outputName, ...
```

```
                '" -results-dir "', artifactDir, ...
                '" -format ', format];

    % Execute the command
    [status, cmdout] = system(command);

    % Check the status and display appropriate message
    if status == 0
        fprintf('Polyspace report generated successfully.\n');
    else
        fprintf('Error generating Polyspace report.\n');
        fprintf('Command output:\n%s\n', cmdout);
    end
end
```

```
function convertToCodeQuality(inSarifFile, outGitlabFile, autoEval)
    % Convert a Polyspace SARIF JSON file to a GitLab Code Quality
    % compliant JSON file.
    %
    % DESCRIPTION
    %   This function reads a Polyspace SARIF JSON file and writes out a
    %   JSON file compliant with GitLab Code Quality.
    %
    % INPUTS
    %   inSarifFile     - Directory and file name of a OASIS SARIF JSON
    %   file exported by Polyspace Bug Finder or Code Prover
    %   outGitlabFile   - Intended directory and file name of the created
    %   Code Quality JSON file
    %   autoEval        - Option to activate automatic evaluation
    %
    % OUTPUTS
    %   none

    % Define function arguments and defaults
    arguments
        inSarifFile
        outGitlabFile
        autoEval (1,1) logical = false
    end

    % Read and decode the input SARIF file
    fid = fopen(inSarifFile, 'r');
    raw = fread(fid, '*char')';
```

```
    fclose(fid);
    sarifData = jsondecode(raw);

    % Initialize an empty struct array for the output
    gitlabFindings = struct('description', {}, 'check_name', {}, 'fingerprint', {},
    ...
                            'severity', {}, 'location', {});

    % Counter for unique fingerprint generation
    resultCounter = 1;

    % Process each run in the SARIF file
    for iRun = 1:numel(sarifData.runs)
        runData = sarifData.runs(iRun);

        % Build a mapping from rule IDs to rule names
        ruleMap = containers.Map;
        if isfield(runData, 'tool') && isfield(runData.tool, 'driver') && ...
                isfield(runData.tool.driver, 'rules')
            for iRule = 1:numel(runData.tool.driver.rules)
                thisRule = runData.tool.driver.rules(iRule);
                if isfield(thisRule, 'id') && isfield(thisRule, 'name')
                    ruleKey = strtrim(char(thisRule.id));
                    ruleMap(ruleKey) = thisRule.name;
                end
            end
        end

        % Get the artifact list for file paths
        artifacts = [];
        if isfield(runData, 'artifacts')
            artifacts = runData.artifacts;
        end

        % If no results, skip this run
        if ~isfield(runData, 'results')
            continue;
        end

        % Process results
        for iRes = 1:numel(runData.results)
            if iscell(runData.results) % Bug Finder JSON
```

```
            res = runData.results{iRes};
        else % Code Prover JSON
            res = runData.results(iRes);
        end

        % Extract description message
        if isfield(res, 'message') && isfield(res.message, 'text')
            descriptionText = strtrim(char(res.message.text));
        else
            descriptionText = '(No message provided)';
        end

        % Extract rule ID and look up rule name
        if isfield(res, 'ruleId')
            ruleId = strtrim(char(res.ruleId));
            if isKey(ruleMap, ruleId)
                checkName = ruleMap(ruleId);
            else
                checkName = ruleId;
            end
        else
            ruleId = 'unknown_rule';
            checkName = 'unknown_rule';
        end

        % Map severity of analysis results
        severity = mapSeverity(res);

        % Determine file path from first location
        filePath = '(no file)';
        if isfield(res, 'locations') && ~isempty(res.locations)
            if iscell(res.locations)
                loc = res.locations{1};
            else
                loc = res.locations(1);
            end
            if isfield(loc, 'physicalLocation') && ...
               isfield(loc.physicalLocation, 'artifactLocation') && ...
               isfield(loc.physicalLocation.artifactLocation, 'index')
                artIndex = loc.physicalLocation.artifactLocation.index + 1;
                if ~isempty(artifacts) && artIndex <= numel(artifacts)
                    artifactUri = artifacts(artIndex).location.uri;
```

```
                        % Remove any leading "file:/" or similar prefix
                        filePath = regexprep(artifactUri, '^file:/+', '');
                    end
                end
            end

            % Create unique fingerprint
            fingerprint = sprintf('%s_%d', ruleId, resultCounter);
            resultCounter = resultCounter + 1;

            % Build GitLab Code Quality finding
            newFinding = struct( ...
                'description', descriptionText, ...
                'check_name',  checkName, ...
                'fingerprint', fingerprint, ...
                'severity',    severity, ...
                'location', struct('path', filePath, 'lines', struct('begin', 1)) ...
            );

            % Append new finding to output struct array
            gitlabFindings(end+1) = newFinding;
        end
    end

    % Encode results to JSON
    jsonOut = jsonencode(gitlabFindings, 'PrettyPrint', true);

    % Write JSON output
    fid = fopen(outGitlabFile, 'w');
    fwrite(fid, jsonOut, 'char');
    fclose(fid);

    fprintf('Wrote GitLab Code Quality results to: %s\n', outGitlabFile);

    % Quit with exit code 1 if any result has critical severity
    if autoEval
        anyCritical = any(strcmp({gitlabFindings.severity}, 'critical'));
        if anyCritical
            exit(1);
        end
    end
end
```

```
function severity = mapSeverity(res)
    % Map the severity of an analysis result
    %
    % DESCRIPTION
    %   This function maps the severity of a static analysis result
    %   depending on the analysis performed.
    %
    % INPUTS
    %   res         - Result
    %
    % OUTPUTS
    %   severity    - Severity

    % Default severity
    severity = 'info';

    % Initialize empty strings
    metaFamily = '';
    color = '';

    % Obtain properties
    if isfield(res, 'properties')
        if isfield(res.properties, 'metaFamily')
            metaFamily = strtrim(char(res.properties.metaFamily));
        end
        if isfield(res.properties, 'color')
            color = strtrim(char(res.properties.color));
        end
    end

    % Use metaFamily mapping for Bug Finder
    if ~isempty(metaFamily)
        switch metaFamily
            case 'Defect'
                severity = 'major';
            case 'Coding Rule'
                severity = 'minor';
        end

    % Use color mapping for Code Prover
    elseif ~isempty(color)
        switch color
```

```
            case 'RED'
                severity = 'critical';
            case 'GRAY'
                severity = 'major';
            case 'ORANGE'
                severity = 'minor';
        end

    % Otherwise fallback to level mapping
    elseif isfield(res, 'level')
        switch lower(res.level)
            case 'error'
                severity = 'major';
            case 'warning'
                severity = 'minor';
        end
    end
end
```

A.3. Pipeline Configuration

```
# This pipeline includes all verification jobs for the Thrust Controller example
    repository.
# It defines seven stages and runs seven jobs using the defined templates.

include:
  - local: 'jobs/code-analyzer.yml'
  - local: 'jobs/model-advisor.yml'
  - local: 'jobs/design-verifier.yml'
  - local: 'jobs/test.yml'
  - local: 'jobs/code.yml'
  - local: 'jobs/bug-finder.yml'
  - local: 'jobs/code-prover.yml'

default:
  tags:
    - matlab
    - windows

stages:
  - code-analyzer
  - model-advisor
```

```
  - design-verifier
  - test
  - code
  - bug-finder
  - code-prover

variables:
  MODEL: "ThrustController"

run-code-analyzer:
  extends: .run-code-analyzer
  stage: code-analyzer

run-model-advisor:
  extends: .run-model-advisor
  stage: model-advisor

run-design-verifier:
  extends: .run-design-verifier
  stage: design-verifier

run-tests:
  extends: .run-tests
  stage: test

run-coder:
  extends: .run-coder
  stage: code

run-bug-finder:
  extends: .run-bug-finder
  stage: bug-finder

run-code-prover:
  extends: .run-code-prover
  stage: code-prover
```

```
# This job runs static analysis of MATLAB files with the dedicated function
    runCodeAnalyzer,
# which calls writeToCodeQuality to display the findings in the GitLab UI.

.run-code-analyzer:
  script:
```

```
    - echo "Executing Code Analyzer function ..."
    - matlab -wait -batch "runCodeAnalyzer();"
  artifacts:
    reports:
      codequality:
        - codeAnalyzerCodeQuality.json
```

```
# This job runs static analysis of Simulink models with the Model Advisor with a
    dedicated function.
#
# The job expects the following variables to be defined:
#   - $MODEL

.run-model-advisor:
  script:
    - matlab -wait -batch "loadModel('models/$MODEL'); checkIDs =
    getCheckIDs('design'); runModelAdvisor('$MODEL', checkIDs);"
  artifacts:
    when: always
    paths:
        - artifacts
    reports:
        junit: artifacts/ModelAdvisorReport.xml
```

```
# This job performs model checking in Simulink with a dedicated function.
#
# The job expects the following variables to be defined:
#   - $MODEL

.run-design-verifier:
  script:
    - matlab -wait -batch "loadModel('models/$MODEL'); runDesignVerifier('$MODEL');"
  artifacts:
    when: always
    paths:
      - artifacts
```

```
# This job executes tests with Simulink Test with a dedicated function.
#
# The job expects the following variables to be defined:
#   - $MODEL

.run-tests:
```

```
  script:
    - matlab -wait -batch "loadModel('models/$MODEL'); runTests();"
  artifacts:
    when: always
    paths:
      - artifacts
    reports:
      junit: artifacts\TestResults.xml
      coverage_report:
        coverage_format: cobertura
        path: artifacts\TestCoverage.xml
```

```
# This job executes code generation for a specified model with a dedicated function.
# It saves the generated archive as artifact to be available for following stages.
#
# The job expects the following variable to be defined:
#   - $MODEL

.run-coder:
  script:
    - matlab -wait -batch "loadModel('models/$MODEL'); generateCode('$MODEL');"
  artifacts:
    paths:
      - ThrustControllerWithFunction.zip
```

```
# This job executes a Polyspace Bug Finder analysis with a generated code archive
    that has a Polyspace options file.
# It exports and converts the results to be available in GitLab with a dedicated
    function.
#
# Environmental variables:
#   - $CI_PROJECT_NAME
#
# File names have to be adjusted manually.

.run-bug-finder:
  script:
    - 7z x ThrustController.zip
    - cd $CI_PROJECT_NAME\polyspace
    - polyspace-bug-finder -options-file optionsFile.txt -misra-cpp required-rules
    - polyspace-results-export -format json-sarif -output-name bugFinderResults.json
    - cd ..\..
```

```
    - matlab -wait -batch
    "convertToCodeQuality('$CI_PROJECT_NAME\polyspace\bugFinderResults.json',
    'bugFinderCodeQuality.json')"
  artifacts:
    reports:
      codequality:
        - bugFinderCodeQuality.json

# This job executes a Polyspace Bug Finder analysis with a generated code archive
    that has a Polyspace options file.
# It exports and converts the results to be available in GitLab with a dedicated
    function.
#
# Environmental variables:
#   - $CI_PROJECT_NAME
#
# File names have to be adjusted manually.

.run-code-prover:
  script:
    - 7z x ThrustController.zip
    - cd $CI_PROJECT_NAME\polyspace
    - polyspace-code-prover -options-file optionsFile.txt
    - polyspace-results-export -format json-sarif -output-name codeProverResults.json
    - cd ..\..
    - matlab -wait -batch
    "convertToCodeQuality('$CI_PROJECT_NAME\polyspace\codeProverResults.json',
    'codeProverCodeQuality.json')"
  artifacts:
    reports:
      codequality:
        - codeProverCodeQuality.json
```

A.4. Supplementary Scripts

```
% runVerification.m
%
% DESCRIPTION
%   This script defines model and directories and subsequently runs all
%   functions defined for the automated verification and code generation
%   for the Thrust Control simulation.

```

```
close all
clear
clc

%% Load model
model = 'DiscreteThrustControl';
loadModel(model);

%% Check Model Compliance with Model Advisor and Simulink Check
checkIDs = getCheckIDs('design');
runModelAdvisor(model, checkIDs);

%% Collect Model metrics with Simulink Check
metricIDs = getMetricIDs('all');
collectModelMetrics(model, metricIDs);

%% Run Design Error Detection with Simulink Design Verifier
runDesignVerifier(model);

%% Run Tests with Simulink Test
runTests();

%% Generate C++ Code with Embedded Coder
generateCode(model);

%% Run Polyspace Bug Finder Analysis
if isPolyspaceIntegrated()
    runPolyspaceBugFinder(model);
end

%% Run Polyspace Code Prover Analysis
if isPolyspaceIntegrated()
    runPolyspaceCodeProver(model);
end

%% Close the model
close_system(model, 0);
disp('Script completed all tasks.');
```

```
% getParameters.m
%
% DESCRIPTION
%   This script obtains the simulation parameters for the Thrust Control
```

```
%   simulation using functionality of the Control Systems Toolbox.

close all
clear
clc

%% Define Thrust Chamber as Plant

% Define parameters
R       = 300;          % Specific gas constant in J/kg-K
T_c     = 3200;         % Chamber temperature in K
c_s     = 1700;         % Characteristic velocity in m/s
A_t     = 1.0e-4;       % Nozzle throat area in m^2
V_c     = 5.0e-3;       % Chamber volume in m^3

% Compute transfer function variables
K_p = c_s / A_t;
tau_p = (V_c * c_s) / (R * T_c * A_t);

% Create transfer function object
G_p = tf(K_p, [tau_p, 1]);

% Create stace-space model object
S_p = ss(G_p);

% Convert to discrete time and compare
S_pd = c2d(S_p,0.001,'foh');
step(S_p,'-',S_pd,'--');

%% Define Valve as Actuator

% Define parameters
m_d_max = 0.1;          % Maximum mass flow in kg/s
tau_v   = 0.1;          % Valve response time in s

% Create transfer function object
G_a = tf(m_d_max, [tau_v, 1]);

% Create space-space model object
S_a = ss(G_a);

% Convert to discrete time and compare
```

```
S_ad = c2d(S_a,0.001,'foh');
step(S_a,'-',S_ad,'--');

%% Define Kalman Filter for Plant

% Provide a model sys that has an input for the noise w.
% sys is not the same as Plant, because Plant takes the input un = u + w.
sys = S_pd*[1 1];

% Specify the noise covariances. Assume both noise sources have unit
% covariance and are not correlated (N = 0)
Q = 1;
R = 1;
N = 0;

% Design the filter
[kalmf,L,P] = kalman(sys,Q,R,N);

%% Define parameters explicitly

% Simulation parameters
P_c_set = 1e6;          % Chamber pressure set point
r_m_set = 1.5;          % Mixture ratio set point
m_d_0 = 1e-3;           % Nonzero initial actuator state

% Plant parameters
S_pd.A  = 0.9888;       % Plant state transition matrix
S_pd.B  = 16.2002;      % Plant control input matrix
S_pd.C  = 1.1719e+04;   % Plant measurement matrix
S_pd.D  = 9.5640e+04;   % Plant feedthrough matrix

% Actuator parameters
S_ad.A  = 0.9900;       % Actuator state transition matrix
S_ad.B  = 9.9006e-04;   % Actuator control input matrix
S_ad.C  = 1;            % Plant measurement matrix
S_ad.D  = 4.9834e-04;   % Plant feedthrough matrix
```

```
% getConfiguration.m
%
% DESCRIPTION
%   This script obtains and lists all configuration parameters of a model
%   that are not set to their default values.

```

```
% Clear workspace
clear
clc

% Load the specified configuration set
data = load('CodeGen.mat');
configSet = data.('CodeGen_cfg');

% Create default configuration set for comparison
defaultConfigSet = Simulink.ConfigSet;
set_param(defaultConfigSet, 'SystemTargetFile', 'ert.tlc');

% Retrieve all parameter names from loaded configuration set
loadedParameters = get_param(configSet, 'ObjectParameters');
paramNames = fieldnames(loadedParameters);

% Retrieve parameters from default configuration set
defaultParameters = get_param(defaultConfigSet, 'ObjectParameters');
nonDefaultParams = {};

% Compare parameters between loaded and default configuration set
for n = 1:length(paramNames)
    paramName = paramNames{n};

    % Check if parameter exists in both configurations
    if isfield(defaultParameters, paramName)
        currentValue = get_param(configSet, paramName);
        defaultValue = get_param(defaultConfigSet, paramName);

        % Compare values
        if ~isequal(currentValue, defaultValue)
            nonDefaultParams{end+1, 1} = paramName; %#ok
            nonDefaultParams{end, 2} = currentValue;
            nonDefaultParams{end, 3} = defaultValue;
        end
    else
        warning('Parameter "%s" not found in default configuration set.', paramName);
    end
end

% Display non-default parameters
disp('Configuration parameters not set to their default values:');
```

```
disp(table(nonDefaultParams(:,1), nonDefaultParams(:,2), nonDefaultParams(:,3), ...
    'VariableNames', {'ParameterName', 'CurrentValue', 'DefaultValue'}));
```

```
% getInputs.m
%
% DESCRIPTION
%   This script obtains input data from a simulation in a MAT file for
%   the signals that have been enabled for logging.

% Run the simulation and log the signals enabled for logging
simOut = sim('DiscreteThrustControl');

% Retrieve logged data from simulation output
logs = simOut.get('logsout');

% Save dataset to MAT file
save('inputData.mat', 'logs');

% Visualize the logged input signal for confirmation
inputSignal = logs.getElement('P_c').Values;
figure;
plot(inputSignal.Time, inputSignal.Data);
xlabel('Time (s)');
ylabel('Signal Value');
title('Logged Input Signal');

disp('Script completed all tasks.');
```

```
% createTests.m
%
% DESCRIPTION
%   This script creates arbitrary test cases for testing with Simulink Test.

% Create the test file, test suite, and test case structure
tf = sltest.testmanager.TestFile('controllerBaselineTest');
ts = createTestSuite(tf,'Baseline Test Suite');
tc = createTestCase(ts,'baseline','Baseline Test Case');

% Remove the default test suite
tsDel = getTestSuiteByName(tf,'New Test Suite 1');
remove(tsDel);

% Assign the system under test to the test case
```

```
setProperty(tc,'Model','DiscreteThrustControlWithFunction');

% Capture the baseline criteria
baseline = captureBaselineCriteria(tc,'baselineData.mat',true);

% Set the baseline criteria tolerance for one signal
sc = getSignalCriteria(baseline);
sc(1).AbsTol = 0.1;

% Turn on coverage settings at test-file level
cov = getCoverageSettings(tf);
cov.RecordCoverage = true;

% Enable MCDC and signal range coverage metrics
cov.MetricSettings = 'mr';
```

Declaration of Authorship

I hereby affirm that I have written the present work independently and have used no sources or aids other than those indicated. All parts of my work that have been taken from other works, either verbatim or in terms of meaning, have been marked as such, indicating the source. The same applies to drawings, sketches, pictorial representations and sources from the internet, including AI-based applications or tools. The work has not yet been submitted in the same or a similar form as a final examination paper.

I have used AI-based applications and/or tools and documented them in the appendix "Use of AI-Based Applications".

Date	Signature

Declaration of Publication

I agree that my thesis may be viewed by third parties in the university archive for academic purposes.

I agree that my thesis may be viewed by third parties for academic purposes in the university archive after 30 years (in accordance with §7 para. 2 BremArchivG).

Date	Signature

Declaration of Consent

Submitted papers can be checked for plagiarism using qualified software in accordance with § 18 of the General Section of the Bachelor's or Master's Degree Examination Regulations of the University of Bremen. For the purpose of checking for plagiarism, the upload to the server is done using the plagiarism software currently used by the University of Bremen.

I agree that the work I have submitted and written will be stored permanently on the external server of the plagiarism software currently used by the University of Bremen, in a library belonging to the institution (accessed only by the University of Bremen), for the above-mentioned purpose.

Consent to the permanent storage of the text is voluntary. Consent can be withdrawn at any time by making a declaration to this effect to the University of Bremen, with effect for the future. Further information on the checking of written work using plagiarism software can be found in the data protection and usage concept. This can be found on the University of Bremen website.

With my signature, I confirm that I have read and understood the above explanations and confirm the accuracy of the information provided.

Date	Signature

Use of AI-Based Applications

All prompts were submitted to the official MATLAB GPT by MathWorks. Where suitable, follow-up queries were submitted after the listed prompts.

Number	Prompt	Comment
1	Explain the basics of Git and GitLab.	Good first overview. More helpful for Git as GitLab's own documentation is very comprehensive.
2	Explain to me how I can perform static analysis of Matlab code.	Good introduction into the legacy MLint and more recent Code Analyzer and codeIssues capabiltites.
3	Explain to me what Simulink Check is and how to use it.	Overview of product. Reading the documentation was then found to be more helpful.
4	Explain the following message: The model includes floating-point arithmetic. Simulink Design Verifier approximates floating-point arithmetic with rational number arithmetic.	initial understanding of Simulink Design Verifier.
5	Brainstorm ideas for a system to be modeled in Simulink. The model shall be used to demonstrate the Simulink model-based verification workflow using Simulink Design Verifier, Simulink Check, Simulink Test and Simulink Coverage in an illustrative way.	Initial ideas for the example project. After that, further literature research was required regarding the suitability.

6	Explain to me how model checking can be applied to control systems. I know that model checking can be done for models that employ logical conditions, switching (e.g. as state machine), but I don't understand how that can be applied to a feedback control system.	Generic answer with a lot of uncertainty. In the end not included in thesis as there is no certainty here.
7	Walk me through the process of discretizing and building the model outlined in the figure in the attachment in Simulink.	GPTs are currently still not very helpful in modeling.
8	Explain to me how to model a rocket engine fuel valve, that has pressure and area as an input and mass flow as an output in Simulink as a discrete transfer function.	Another more concrete attempt.
9	Explain to me how i can a) enforce custom coding guidelines and b) trace back errors from code to model in a model-based software engineering workflow using Matlab Embedded Coder and Polyspace products.	Initial understanding of these products. Again, reading the documentation was more helpful.
10	Explain the attached paper to me.	Explanation of Stålmarck's proof method for propositional logic. Was decided to be out of scope for this thesis.
11	Show me how to implement the transfer function (first equation on page 2) from this paper in Simulink.	Modeling the contents of the paper *Overview of Rocket Engine Control*.
12	Generate a Matlab script, that for a model file performs verification with the attached Matlab/Simulink verification products.	Initial draft of `runVerification.m`.
13	Encapsulate the functionality of the attached scripts in functions.	Initial draft of verification functions.

14	Write a Matlab script that opens a configSet file stored in a .mat file and lists all configuration parameters that are not set to their default value.	Initial draft of `getConfiguration.m`.
15	Describe how to model this closed loop rocket engine control in open loop. Most important to me is what would be the required inputs and outputs.	Better understanding but in the end discarded.
16	Walk me through the differences between MISRA C++ 2008 and 2023.	Very detailed and comprehensive but cumbersome to verify.
17	Name all changes from C++03 to C++11.	Again very detailed but not easily verifiable.
18	Change the following function so that instead of the valve area, it accepts the “opening” as a value from 0 to 1.	Normalization after unfavorable numeric results.
19	Describe how Git flow, GitHub flow and GitLab flow work.	Overview of branching strategies.
20	What steps are necessary in order to create a GitLab runner.	Again, documentation was more helpful after getting an overview.
21	Explain to me how the results of a Polyspace analysis are typically processed in an automated way.	GPT was not aware of export functionality, but remarks about other tools were helpful.
22	Compare the CodeClimate report format to the SARIF format.	Information from the GitLab documentation.
23	How can I verify the toolbox requirements of a Matlab command?	Not always trivial.
24	Can a GitLab CI/CD pipeline be configured to pass or fail depending on specific command line output?	Helpful explanation about exit codes.

25	In a GitLab CI/CD pipeline, how can I save and reuse artifacts from one job to another?	Pointer to documentation and artifact default behavior.
26	Look into the following repository: https://github.com/dapperfu/Jenkins-Simulink-Model-Advisor and explain what it does.	Initial idea of results reporting in GitLab.
27	The attached file contains a list of Model Advisor checks. Group the individual check IDs into cell arrays in a useful way.	GPT was not able to reliably perform this task.
28	Provide an example function for programmatically running Model Advisor checks and exporting them as XML.	Initial draft of Model Advisor automation.
29	Explain statement, decision and condition coverage with a simple control flow graph.	Helpful explanation but example was not very suitable.
30	Consider the attached papers and everything else you know about chemical rocket engine control. Is it possible to describe an attitude thruster for a spacecraft as a linear time-invariant system?	Helpful to some degree.
31	Brainstorm ways to combine a basic PID controller and a simple state machine in a control loop.	Initial idea of gain scheduling with a FSM.
32	How are pressure-fed spacecraft propulsion systems throttled? What is controlled and how?	Not entirely accurate summary.
33	Verify that both example files are indeed structured the same. List differences that might affect analysis of the files with Matlab.	Troubleshooting the JSON file conversion for Polyspace results.
34	Develop a Matlab function that converts results files provided by Polyspace analyses in the OASIS SARIF JSON format to a GitLab Code Quality compliant JSON file.	Initial draft that had to be troubleshooted extensively.

35	The attached image tries to illustrate what happens during the analysis phase of compilation, i.e. lexical analysis and syntax analysis. Find a more illustrative example with a shorter statement.	Shorter example provided.
36	The attached image is an excerpt of an explanation of abstract interpretation. I am not sure what the benefit of using such a Hasse diagram is. Try to find a way to explain abstract interpretation how it is done by static analysis tools.	Helpful in addition to Cousot's literature.
37	Append the main function so that Matlab is quit with an exit code 1 when there is at least one result with severity 'critical'.	Inclusion of `autoEval` functionality.
38	Simulink Design Verifier to the my best knowledge is a Model Checker integrated into Simulink. As far as I know, model checking is only applicable to finite automata, however the Toolbox in principle seems to be compatible also with, say, a standard closed-loop control loop with a PID controller. Explain to me in detail how this is possible from a theoretical point of view.	Another attempt that resulted in vague information.
39	The following excerpt is from an introduction in abstract interpretation for static analysis. Explain what it means.	Better explanation of the background of soundness in proof theory.
40	Assume a standard control loop including the conventional Controller, Actuator and Plant blocks. When I want to implement a very basic Kalman filter, how do I have to connect it correctly?	Overview of Kalman filter integration, in the end excluded.
41	Transform the attached in SI units.	Processing of values from *Rocket Propulsion Elements*.

42	When K is the nominator in a transfer function of a LTI system, is then always C=K in its state space representation?	Confirmation after the derived state-space representation seemed quite simple.
43	Compare cppcheck, Polyspace and Astrée.	A rather generic overview, with more details found in the documentation.
44	The following is a list of Simulink checks relating to compliance. Try to group and summarize them according to their meaning.	Another unsuccessful attempt.
45	Write Matlab code that performs the exact same task as the depicted subsystem.	Initial function implementation.
46	In the Kalman Filter Simulink block, what needs to be specified in the System Model section of its properties?	Understanding of Kalman filter in Simulink, later excluded.
47	Explain what MathWorks means with HISM. Is HISM itself a standard?	Understanding of Simulink modeling guidelines.
48	I want to use equivalence testing in Simulink Test to verify the performance/accuracy of a Kalman Filter in a conventional feedback control loop. What signals would I need to compare?	Understanding of Kalman filter in Simulink, later excluded.
49	Matlab code does not need to be compiled to be run, right? So, from knowing that static analysis is somewhat similar to compilation, what are the differences in Matlab?	Not quite accurate overview of static analysis for interpreted languages.
50	Explain in detail the Simulink model configuration and configuration parameters.	Good overview of model configuration behavior.
51	Explain OOP in Matlab.	Good overview of how Matlab handles object-oriented programming.

52	Summarize chapter 2 of this paper.	Another summary of *A Tutorial on Stålmarck's Proof Procedure for Propositional Logic.*
53	What is a level-2 Matlab S-function?	Overview of S-functions.
54	What exactly is a Design Verifier model representation?	Another vague answer.
55	Explain static and dynamic memory allocation in C/C++.	Good explanation of the question.
56	How exactly are numeric data types handled in Matlab?	Overview of typing in Matlab.
57	Consider the following Matlab function. Instead of using one control function, I want to implement this control behavior as a state machine with the attached logic.	Initial version of the FSM implementation.
58	What are the phases in the testing process according to ISTQB?	Reminder of the testing process.
59	Give an overview of C++ versions and their individual features/changes. Add some historic context where suitable.	Good overview of the question.
60	Show me how to declare optional input arguments to a Matlab function with a default value.	Overview of argument handling functionality in Matlab.
61	With the goal of code generation, is it possible to implement a controller in a Simulink simulation model and then generate code for the subsystem, or do I have to save it as individual model and reference it in the simulation?	Understanding the relation of model referencing and code generation.
62	I have two version of Matlab installed. When starting in batch mode, how do i specify the version?	Help in handling automated execution.

63	I have my machine set up as a GitLab runner. When I run Matlab from the command line, it works as expected, but when I run the exact same command in the corresponding GitLab pipeline, there is a license error. Is this to be expected?	Fruther help in handling automated execution.
64	How can I set variables from within a Matlab function to be available from the Workspace?	Use of the function `assignin` that is not advisable.
65	Is there an equivalent to Clang-format for Matlab code?	Reference to third-party open source tools.
66	I have a GitLab pipeline comprised of several stages. Show me how to encapsulate each of its job in a separate yml file.	Explanation of CI/CD components.
67	What options do I have for queuing GitLab pipeline executions (based on the availability of a license that software run in the pipeline needs)?	
68	Suppose my repository is located at something/my-repo and called “My Repo”, what GitLab CI variable will output “my-repo” instead of “My Repo”?	Understanding GitLab variables.
69	In software development, what is an integration or an release build?	Better understanding of conventional software development.
70	Explain the context of `com.mathworks.xml.XMLUtils`.	Overview of the Matlab Java XML processing API.
71	Is there something similar to the classes `plugins.codecoverage.CoverageResult` and `coverage.result.CoverageSummary` in earlier releases of Matlab?	Explanation when which functionality was added.
72	How can I programmatically access the coverage results for display in merge requests using Matlab/Simulink R2021b?	Explanation of missing functionality.