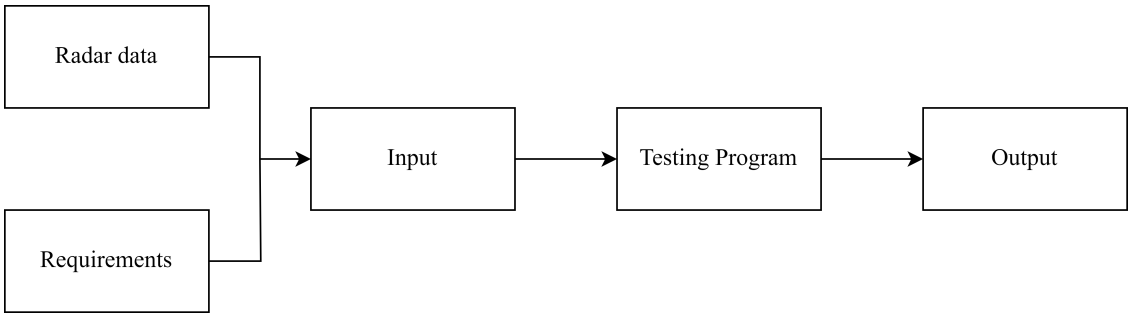

Code-based requirements for software verification

Master's thesis in Systems, Control and Mechatronics

ALBIN OLSSON

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Code-based requirements for software verification

ALBIN OLSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2024

Code-based requirements for software verification
ALBIN OLSSON

© ALBIN OLSSON, 2024.

Supervisor: David Johansson, Saab AB
Examiner: Martin Fabian, Automation

Master's Thesis 2024
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Overview of the implemented testing software.

Typeset in LaTeX
Printed by TeknologTryck
Gothenburg, Sweden 2024

Code-based requirements for software verification
ALBIN OLSSON
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Saab AB is currently investigating code-based requirements for software verification. This is to enhance the quality and reliability of their products, ensuring they meet industry standards and best practices. The purpose of this master's thesis is to explore the potential of property-based testing in code-based requirements for hardware-near-software verification. The research aims to assess how well property-based testing methods can be applied to the unique verification challenges that arise in systems closely integrated with hardware. More specifically, the thesis will focus on developing a program to use this type of property-based testing and see if it can be effectively adapted to test code-based requirements for data that interacts directly with hardware. By conducting discussions with professionals in the field, as well as implementing the code to obtain the verification process a new program written in Python and used for hardware-near-software verification that will be adapted by Saab AB is presented in the thesis.

Keywords: Code-based verification, Software verification, Programming, Testing.

Acknowledgements

Firstly, I would like to show my gratitude to Chalmers University of Technology for offering an environment that promotes learning and research, and for the opportunity to undertake this exciting master's thesis in software verification.

My appreciation goes to my examiner, Professor Martin Fabian, whose guidance, support, and patience have been instrumental throughout this journey. Your insightful feedback and academic standards have pushed me to reach my goals with this master's thesis.

I also wish to express my sincere thanks to my supervisor, David Johansson, for his invaluable advice, technical guidance, and timely feedback, which significantly contributed to the successful completion of this thesis.

Lastly, I wish to express my gratitude to my family for their unwavering support, encouragement, and belief in my abilities. Your unconditional love and motivation have been amazing throughout this journey.

Albin Olsson, Gothenburg, Aug 2023

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem statement . . . 1
- 1.3 Scope and Limitations . . . 2
- 1.4 Overview of the thesis structure . . . 2

2 Theory **3**
- 2.1 Hardware-near software . . . 3
  - 2.1.1 Embedded systems . . . 3
- 2.2 Radar . . . 3
  - 2.2.1 Data output . . . 4
  - 2.2.2 Communication errors in radar systems . . . 4
- 2.3 Requirement specifications . . . 6
  - 2.3.1 Significance of linking requirements to code through testing . . . 6
  - 2.3.2 Ensuring system functionality and quality . . . 6
  - 2.3.3 Requirement traceability . . . 6
- 2.4 Functional programming . . . 6
- 2.5 Data generation . . . 7
  - 2.5.1 Ground truth . . . 7
  - 2.5.2 Tampered data . . . 7
- 2.6 Testing data within hardware-near software systems . . . 8

3 Method and Implementation **9**
- 3.1 Program overview . . . 9
- 3.2 Radar Data . . . 9
  - 3.2.1 Data generation . . . 11
    - 3.2.1.1 Techniques for generating ground truth input data . . . 11
    - 3.2.1.2 Techniques for generating tampered input data . . . 11
- 3.3 Requirements . . . 11
  - 3.3.1 description . . . 12
  - 3.3.2 messages . . . 12
  - 3.3.3 parameters . . . 13
  - 3.3.4 correct_sequences . . . 13
  - 3.3.5 incorrect_sequences . . . 13
  - 3.3.6 invalid_sequences . . . 13
  - 3.3.7 get_desc() . . . 13

3.3.8 analysis() . . . 13
3.3.9 Example of requirement . . . 14
3.4 Input . . . 14
3.5 Testing program . . . 14
3.5.1 Functions for checking data . . . 14
3.5.2 Support functions . . . 15
3.5.3 ReqParent . . . 15
3.5.4 Testing requirements . . . 15
3.5.4.1 test_requirements() . . . 16
3.5.4.2 get_req_desc() . . . 16
3.5.4.3 run_requirements() . . . 16
3.5.4.4 get_tested_req_count() . . . 16
3.5.5 Validating testing program . . . 16
3.6 Output . . . 17

4 Analysis **19**
4.1 Solution overview . . . 19
4.2 Advantages of code-based requirements . . . 24
4.3 Strengths and weaknesses of the implemented program . . . 24
4.4 Societal, ethical and environmental analysis . . . 25
4.4.1 Societal impact . . . 25
4.4.2 Ethical impact . . . 26
4.4.3 Environmental impact . . . 26

5 Conclusion **27**
5.1 Conclusion . . . 27
5.2 Future work . . . 27

Bibliography **29**

A Tables over simple functions **I**

1

Introduction

1.1 Background

As technology continues to advance, the complexity of hardware and software systems is rapidly increasing. The reliability of these systems is of utmost importance to ensure the systems' safety and performance. To achieve this, verification of the systems becomes essential. Verification is the process of checking whether a system satisfies its specified requirements or not. There are various verification techniques available, and one of them is property-based testing.

Property-based testing is a technique that uses properties or specifications of a system to define test cases. Variations of property-based testing are widely used in software verification. This thesis, however, aims at testing data closely connected to hardware, rather than code, to verify the functionality of a system. The use of property-based testing can aid in discovering potential bugs and help ensure that hardware-near software systems are functioning correctly.

The purpose of this master's thesis is to develop a program that can be used to verify hardware-near-software systems by using property-based testing. The program will focus on code-based requirements and explore the challenges and limitations of applying property-based testing in this context.

To summarize, this master's thesis aims to develop a verification tool that can improve the reliability of hardware-near software systems using property-based requirements. The result of the thesis is a program that tests data from a system using predefined properties. The data is acquired from a system during testing.

1.2 Problem statement

The underlying issue this thesis aims to improve is the difficulty of connecting requirements to test cases in an effective way. The current method for connecting requirements to test cases is often carried out manually which makes it time-consuming and can lead to errors.

The problem statement of this thesis is to develop and implement a program for connecting code-based requirements to test cases that are automated and efficient.

The implemented program should be able to handle code-based requirements and should also be easy to use and maintain by employees at Saab AB.

1.3 Scope and Limitations

This study focuses on code-based testing in the context of software development, and specifically on the problem of connecting data that is collected from a system to test cases.

The scope of the study is limited to the development of a program that can process requirements automatically and connect them to test cases. The limitations of the study include the use of a specific data structure and the focus on a single organization, Saab AB. Also, no graphical interface will be created during this thesis. Furthermore, no external packages for Python will be installed. This is due to the need to maintain a consistent and controlled environment, ensuring that the software remains stable, avoids potential compatibility issues, and adheres to stringent security protocols. Using only the standard library minimizes external dependencies and reduces potential vulnerabilities that third-party packages might introduce.

1.4 Overview of the thesis structure

The outline of the thesis is as follows. Chapter 2 provides the theory necessary to understand the upcoming chapters. Important theoretical concepts that are described are hardware-near software, radar, requirement specifications, functional programming, data generation, and testing of hardware-near-software systems. Chapter 3 provides a detailed system overview where each part of the methodology is explained. Chapter 4 builds upon Chapter 3 and in this chapter the implemented solution together with an analysis of the applied solution where strengths and weaknesses are provided in an analysis. Here, the impacts on societal, ethical, and environmental aspects are also presented. The last chapter of the report is Chapter 5, which gives conclusions of the thesis and proposes future work that can be done within the subject.

2

Theory

In this chapter, the theory behind the thesis work is presented. The topics of hardware-near software, radar, requirement specification, programming, data generation, and testing are introduced.

2.1 Hardware-near software

In the realm of software development, hardware-near software operates at a fundamental level, interacting directly with the system hardware. Unlike high-level applications, which interact with operating systems or other software systems to perform tasks, hardware-near software interacts directly with the hardware itself.

Examples of hardware-near software include firmware, device drivers, real-time operating systems, and embedded systems. These software pieces are important to a system's function because they manage how the system interacts with its hardware components [1]. This thesis focuses on testing the data from embedded systems mainly in a radar application, using high-level programming languages.

2.1.1 Embedded systems

Embedded systems are dedicated systems designed to perform a specific task, usually in real time. Examples range from micro-controllers in a vehicle's engine management system to the software controlling a smart home's heating system [2]. An embedded system is a system of computer hardware that includes embedded software. For this thesis, the data from an embedded system will be tested using predefined properties using a program that takes the data that should be verified and the requirements on that data as input.

2.2 Radar

Radar, which stands for "Radio Detection and Ranging" [3], is a technology used to detect and track objects in its surrounding environment. The radar emits radio waves and analyzes their reflections on objects. A schematic overview of how a radar works is given in Fig. 2.1.

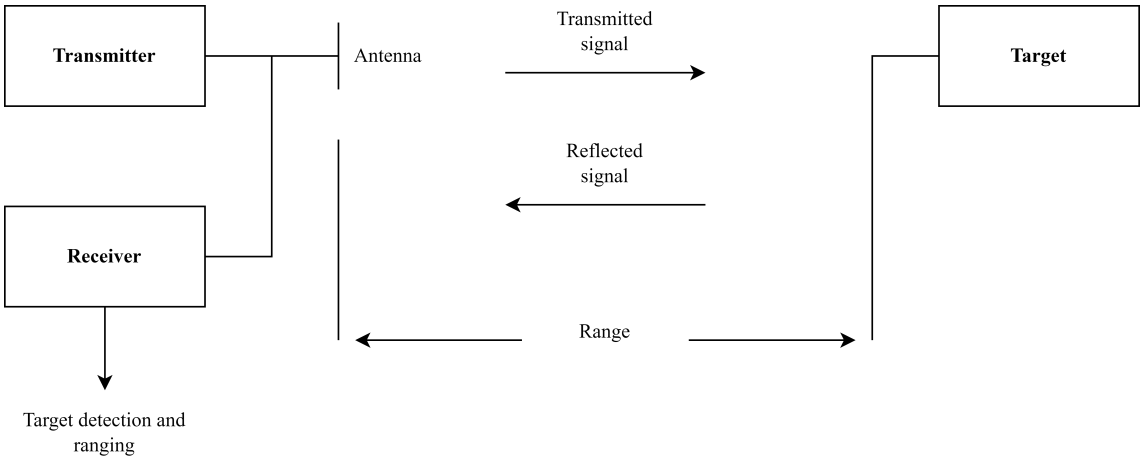

Figure 2.1: Schematic overview of the working principle of radar.

The radar system generates high-frequency electromagnetic waves. The waves are produced by an antenna and sent out into the air. The electromagnetic waves, often referred to as radar signals or radar pulses, propagate through the air at the speed of light. When the waves encounter an object, a portion of the energy is reflected back toward the radar system. The reflective behavior of an object depends on its size, shape, and composition. The radar system has a dedicated antenna that receives the reflected waves, converting them into electrical signals. The receiver amplifies and processes these signals for further analysis. Various signal processing techniques are applied to the received signals to extract useful information. This includes filtering out unwanted noise, amplifying weak signals, and separating different targets if they overlap. The radar system analyzes the processed signals to determine objects' presence, location, and properties. This involves detecting the time it takes for the signals to travel to the object and back (round-trip time) and analyzing the Doppler shift in frequency caused by moving objects. The radar system can present the information in a human-readable format, such as a display screen, audio output, or as structured data. The operator can interpret the data to identify and track objects and measure their distance, speed, direction, and sometimes even their shape [4].

2.2.1 Data output

Data is extracted from the radar module in the form of bits and is parsed to structured numerical and texted data prior to reaching the testing program to be developed in this thesis. It is often presented as numerical values, such as timestamps, sequential numbers, range, and velocity.

2.2.2 Communication errors in radar systems

Hardware systems often consist of various interconnected components that communicate with each other to function as a cohesive unit. In the realm of hardware systems, communication between these components is crucial for the system's seamless functioning. However, there can be instances where communication errors may

occur, affecting the overall system's performance and reliability. This section explores some common scenarios where such communication errors can occur.

Signal timing issues - Communication in hardware systems often rely on accurate and timely signal transmission. Discrepancies in signal timings can lead to data loss or corruption, which can cause significant communication errors. Examples include clock skew, where the clock signal arrives at different components at different times, or synchronization issues between sender and receiver components [5].

Physical connection problems - Issues with the physical infrastructure connecting different components can also lead to communication errors. These issues can range from loose connections and broken wires to overheating issues or damage due to environmental conditions. These physical problems can disrupt the communication lines between components, leading to communication failures.

Electromagnetic interference (EMI) - Hardware systems are susceptible to interference from external electromagnetic sources, which can distort the signal while it is being transmitted, causing communication errors. Even systems within close proximity can cause interference with each other, impacting the communication process [6].

Software or firmware bugs - Communication between hardware components is often controlled by low-level software or firmware. Any bugs or glitches in this code can lead to improper communication, resulting in errors. This can also include issues with device drivers, which facilitate communication between the operating system and hardware components [7].

Inadequate power supply - If a component does not receive an adequate power supply, it may not function correctly, leading to errors in transmitting or receiving data. Fluctuations in power supply can also cause temporary communication disruptions [8].

Hardware incompatibility - In some cases, communication errors can occur when hardware components from different manufacturers or different technology generations are interconnected. These components may not follow the same communication protocols or may have different data transmission speeds, leading to communication issues [9].

In hardware-near software development, it is crucial to understand these potential error scenarios to design robust systems. Proper testing strategies should be in place to identify these issues so that they can be rectified to ensure smooth communication within the system and overall system integrity. Measures such as redundancy, error detection and correction techniques, and shielding against electromagnetic interference can be used to mitigate these risks.

2.3 Requirement specifications

The development of many hardware systems starts with establishing its requirement specifications. These are well-structured, detailed descriptions of what the system should do and achieve.

Requirement specifications are typically drawn up at the beginning of the development process, providing a clear guide for what the developers should create. These specifications can be functional (describing what the system should do) or non-functional (detailing constraints or qualities the system should possess). This thesis incorporated functional requirements. In many cases, these requirements are designed to be *testable*, meaning that each requirement should be falsifiable through some form of testing [10].

2.3.1 Significance of linking requirements to code through testing

When testing a hardware system, it is essential to verify that the system meets the specified requirements. This process can be done through a specific software testing program. This thesis aims at developing a program that tests the data in a way that finds where requirements are not met.

2.3.2 Ensuring system functionality and quality

Connecting each requirement to a test ensures that every aspect of the system can be verified and validated. This is not limited to individual software or hardware components but extends to their interactions, overall system behavior, and whether the system meets the expectations outlined in the requirement specifications. This comprehensive testing and validation ensure the overall quality of the system, paving the way for a reliable, robust, and efficient system [11].

2.3.3 Requirement traceability

The connection between requirement and testing provides traceability from a system requirement to its validation. It creates a transparent and traceable path from requirement specification to system test. This helps in project management, keeping track of what requirements have been tested, which have passed, and which have failed. Furthermore, connecting requirements to testing through code assists in maintaining an organized record for future references or audits [12].

2.4 Functional programming

Functional programming is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative type of programming style that emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.

For the current study, Python's capability to incorporate functional programming will be leveraged. Python allows for the clear and concise expression of algorithms, which improves the readability and maintainability of the code. Its wide range of libraries and tools makes it a popular choice for software development, and its support for functional programming enables a flexible, reliable, and efficient implementation of the program [13].

In the implementation phase, we will use functional programming techniques such as pure functions, function composition, and map, filter, and reduce operations for processing requirements and generating test cases.

2.5 Data generation

The program developed in this thesis is meant to test data from gathered log files. However, there are instances where gathering actual data may not be feasible. For example when a system or its functions are still under development and lack real log data, or when access to system logs is restricted due to security or privacy concerns, generating synthetic log data offers a viable solution. This approach provides insights into potential system interactions, aids in anticipating challenges, and ensures continued analysis without compromising system integrity or confidentiality. In order to test the operational capabilities of the program, data was generated in a systematic manner that enabled review and analysis.

2.5.1 Ground truth

Generating ground truth data for functional software testing involves defining expected outcomes based on the software's specifications. In some cases, it might require manual calculation or the use of another verified system to generate the expected output [14]. In the case of the study, data was generated in a way that would mimic the expected behavior of the system when it works as intended.

2.5.2 Tampered data

Random tampering with ground truth data essentially involves introducing controlled disruptions or 'noise' into the data. This serves multiple purposes in the context of software testing [14].

- Robustness testing: By introducing controlled deviations to the ground truth data, developers can evaluate how well the software handles inaccuracies or unexpected data. This is particularly relevant for software systems that are deployed in real-world environments, where the data may not always be perfect or predictable.
- Fault tolerance evaluation: Random tampering can help assess the software's ability to continue functioning correctly in the face of erroneous data. This is crucial in mission-critical applications where maintaining operation, even under non-ideal circumstances, is of paramount importance.

- Error detection and recovery: The introduction of random errors into the ground truth data can also serve to test the software's ability to detect and recover from such errors. It can help in identifying whether the program can rectify the error or gracefully degrade its performance rather than failing entirely.

2.6 Testing data within hardware-near software systems

Testing is a vital aspect of software development, directly impacting the quality, reliability, and robustness of the end product. By identifying potential issues early in the development process, testing saves both time and resources, while aiding in the identification of discrepancies between the final software and the intended design and functional requirements [14].

The general idea of the program is to go through a set of parameters and make sure that they follow certain properties and patterns. The requirements are set in a way that they should catch all possible errors that are of significance to the functionality of the system.

3

Method and Implementation

In this chapter, the process of creating the testing program is presented. Firstly, a program overview is given in Section 3.1, with the following sections going into detail to explain how the modules in the overview are constructed.

3.1 Program overview

In Fig. 3.1 the backbone of the construction process is shown in a schematic overview. As can be seen, the input to the program consists of partly radar data and partly of requirements for that radar data. This input data is transmitted to the testing program developed during this thesis. The testing program checks if the input data follows the requirements and outputs data with information about how well the input meets the requirements.

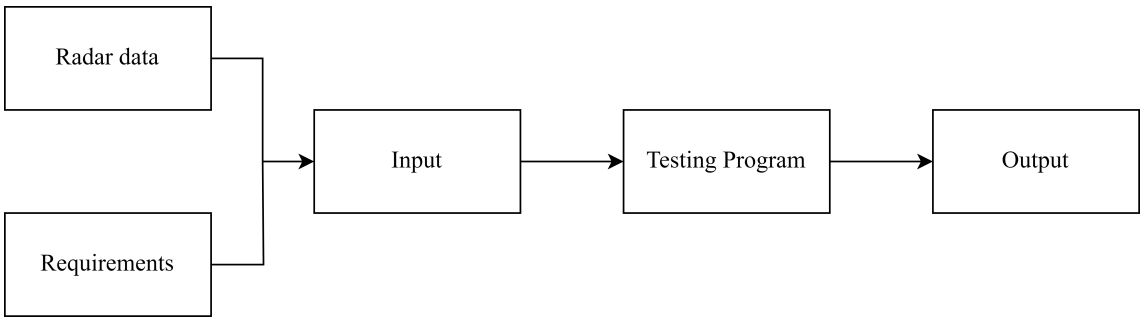

Figure 3.1: Schematic overview of the inputs to the testing program.

In the following subsections, the blocks in Fig. 3.1 will be described in detail and how they are built up in the program.

3.2 Radar Data

The structure of the radar data in Fig. 3.1 was given in advance by Saab AB. What the radar data further consists of is illustrated in Fig. 3.2.

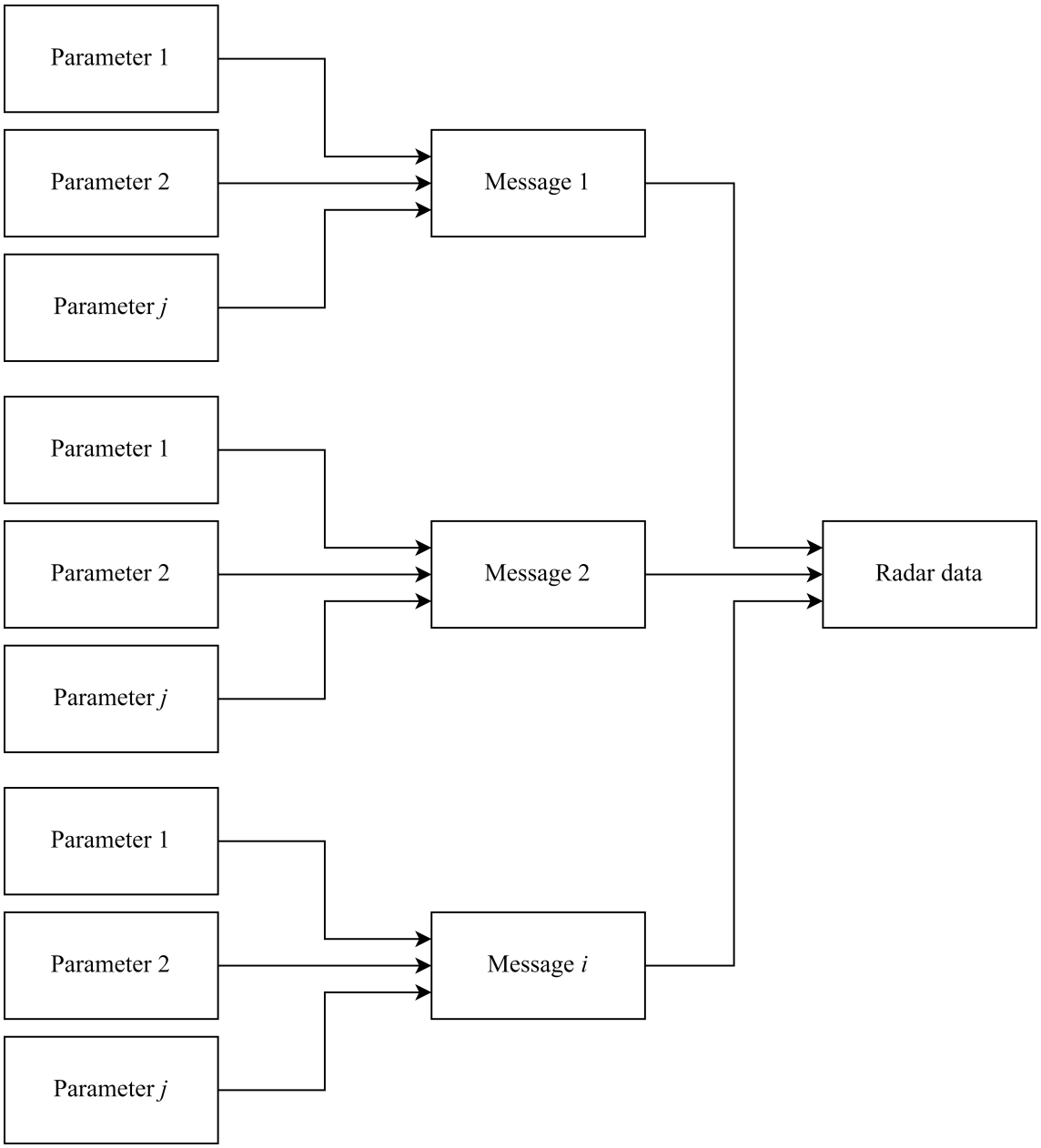

Figure 3.2: Schematic overview of the radar data structure.

The data is grouped into sets of what is referred to as *messages*, one example of a message is the video from the radar module. Messages can be seen as a bundle of parameters that are connected in some way. The parameters of the video contain information about the received radio waves describing the movement of an observed object through space as well as functional parameters necessary for connecting messages to one another, such as indices and timestamps. In Fig. 3.2 the total number of messages is N, denoted as *Message* i, with $i \in [1, 2, \ldots, N]$. The number of parameters for *Message* i is M_i, denoted as *Parameter* j, with $j \in [1, \ldots, M_N]$.

3.2.1 Data generation

In order to test the operational capabilities of the testing program, data was generated in a controlled and systematic manner that enabled review and analysis. This section provides information about how that was done.

3.2.1.1 Techniques for generating ground truth input data

The technique to generate data was by creating data for each parameter that follows the established guidelines of what is expected behavior of the system. The data should also be structured in a predefined way that is currently being used by Saab, which is shown in fig. 3.2

3.2.1.2 Techniques for generating tampered input data

In order to test the functionality of the testing program, data had to be manipulated in a way that mimics real-world problems. This was done by randomly changing, swapping, and deleting data from the ground truth data. All introduced errors were saved in a separate file that can be imported and used as input in order to validate the functionality of the program.

3.3 Requirements

One of the most crucial steps is to gather and define the necessary requirements for the data in the system under test. The requirements-gathering process was conducted collaboratively with employees at Saab. Frequent meetings and discussions were held with employees at Saab, leveraging their expertise and knowledge in software verification. The goal was to anticipate and define requirements that could catch all possible errors to ensure the testing process's efficiency, reliability, and robustness. The requirements should be written in a way that they catch errors, as well as define what is expected behavior so that the program can find if the data does not meet the requirements. After identification, the potential errors were transformed into specific requirements.

The requirements are defined in a clear, concise, and unambiguous manner, making them suitable for testing. It was ensured that each requirement was testable, meaning that it was possible to create a test case that could verify whether the requirement was met or not. The central aim of the requirements was to identify and encompass potential errors that might be present within the data. The structure of the requirements is described in Fig. 3.3.

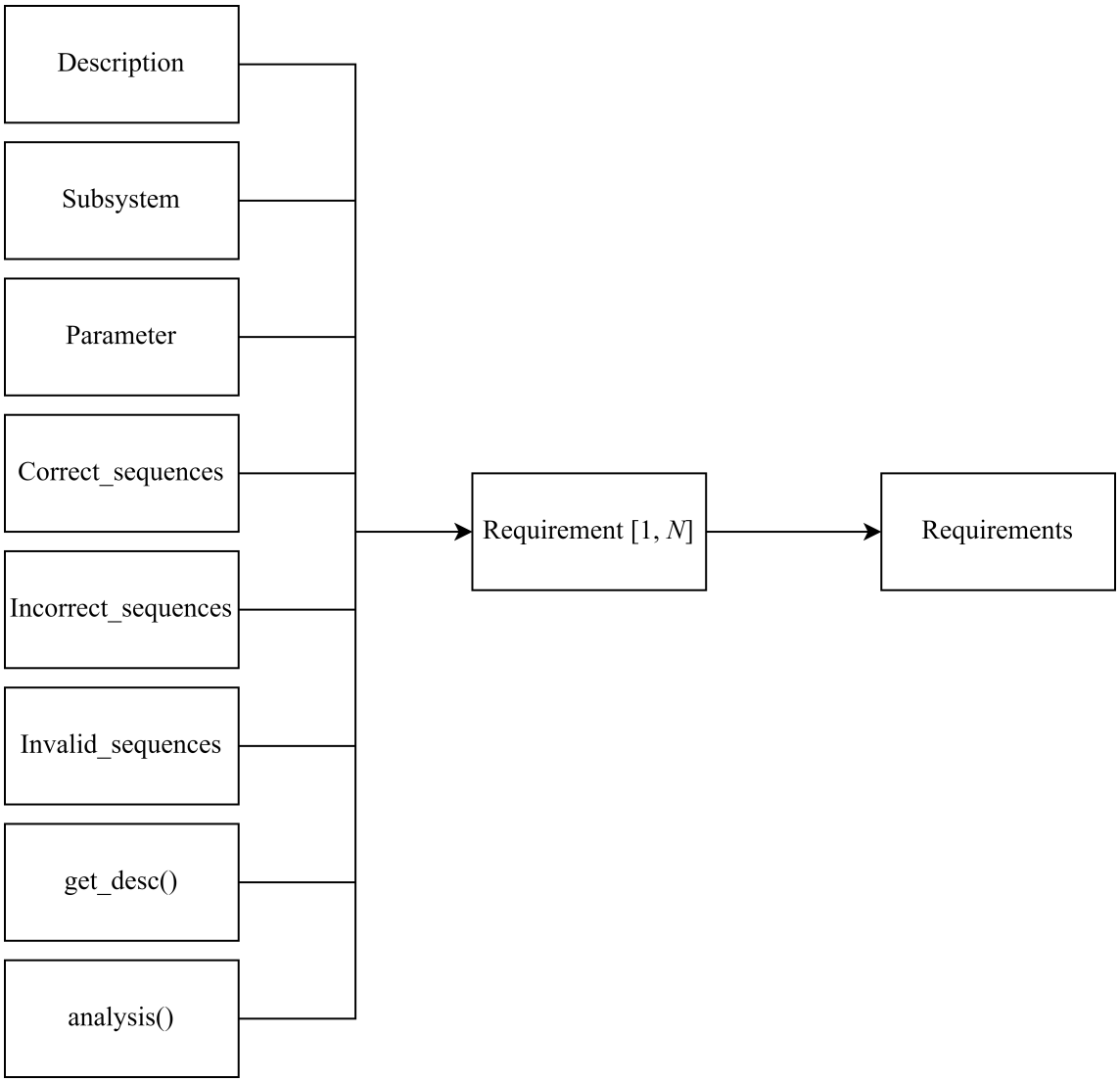

Figure 3.3: Schematic overview of requirement structure.

The **Requirements** consists of an arbitrary number of **Requirement N**, where N, denoted as *Requirement* i, with $i \in [1, 2, \ldots, N]$. Each **Requirement** consists of variables and functions that are described in the subsections 3.3.1-3.3.8.

3.3.1 description

The variable description is a string that clarifies what the requirement specifies. It is used for printing out which requirements are being tested.

3.3.2 messages

This variable defines which message or messages that the requirement should hold for. It consists of a list of strings and can be multiple messages simultaneously.

3.3.3 parameters

This variable defines which parameter or parameters the requirement should hold for. It consists of a list of strings and can therefore be multiple parameters at once.

3.3.4 correct_sequences

This variable consists of a list of sequences that should pass the requirement. This has the same predefined structure as the data in order to be able to loop through all combinations of messages and parameters given above. Each list entry is a dictionary of a dictionary of a list containing the sequence that should be tested. The sequence is structured as a dictionary of a dictionary of a list since it has to match the given radar data structure. The dictionary keys can either be written directly as the message and parameter to be tested, or they can be written as "_message_" and "_parameter_". If the latter is chosen it will go through all possible combinations of the variables message and parameters described in sections 3.3.2 and 3.3.3.

Each entry into the list should fulfill the requirement according to the function analysis(). This variable is used to ensure that the requirement is correctly written. correct_sequences, incorrect_sequences and invalid_sequences are written and used to prevent false positives and false negatives from the requirements. These are meant to be checked before the real data is tested against the requirements.

3.3.5 incorrect_sequences

This variable is built just like the correct_sequences of Section 3.3.4. The only difference is that the entry into the list should not fulfill the analysis. This variable is used to ensure that the requirement is correctly written.

3.3.6 invalid_sequences

This variable is optional and can be used to enter invalid sequences. It can be written in the same form as previous sequences. A sequence will be deemed invalid during the `test_req()`3.5.4 stage, described in subsection 3.5.3, if it is not tested. One reason for not being tested could be that the message or parameter assigned to be affected by the requirement is not in the sequence.

3.3.7 get_desc()

This function is used for getting the description of the requirement as a string.

3.3.8 analysis()

This function is the main part of the requirement. It takes the data, messages to test, parameters to test, and the index of the list specified by the message and parameter as input. Note that each parameter consists of a list. It returns either True for a passed test or a string stating in which way the requirement is not fulfilled if not True.

```python
class Req3(ReqParent):

    description = 'ANT EXI nr should only appear once'

    messages = ['ant']

    values = ['exi_nr']

    correct_sequences = [
        {"_MESSAGE_": {'_VALUE_': [1, 2, 3, 4, 5, 6]}},
        {"_MESSAGE_": {'_VALUE_': [0, 1, 2, 3, 6, 100]}},
        ]

    incorrect_sequences = [
        {'_MESSAGE_': {'_VALUE_': [1, 2, 2, 4, 5, 6]}},
        {'_MESSAGE_': {'_VALUE_': [0, 0, 3, 4, 5]}},
        {'_MESSAGE_': {'_VALUE_': [1, 2, 3, 4, 2, 7]}},
        ]

    def analysis(self, data, msg, value, N):
        data_to_check = data[msg][value][N]
        in_list = data[msg][value][0:N]
        return CheckSpec.check_not_in_list(data_to_check, in_list, 'ant exi numbers')
```

Figure 3.4: Example of requirement

3.3.9 Example of requirement

To clarify how this looks like when it is implemented, a figure of an implemented requirement is shown here.

In this example, the analysis uses a library, `CheckSpec`. The function `check_not_in_list()` returns either `True`, or a string stating the reason for not fulfilling the requirement.

3.4 Input

The input to the program consists of a file with all the requirements and also the data that should be tested according to the requirements.

3.5 Testing program

The testing program is the focus of the thesis work. It is a program that has been developed to test the input described in Section 3.4 and provide structured outputs that tell if the data breaks the requirements. This is done by executing the `analyis()` on the data at all the instances described by `messages` and `parameter`

3.5.1 Functions for checking data

Some functions were created for re-usability in the requirements. These are listed in the table in Appendix A.1.

3.5.2 Support functions

Another set of functions was defined for structure and re-usability. These are listed in the table in Appendix A.2.

3.5.3 ReqParent

A parent class, `ReqParent` was created and serves as a foundational blueprint from which the child classes (`Req1`, `Req2`, and so on) can inherit properties and methods. This is extended in the derived classes. The `ReqParent` class contains several variables and methods. The variables are found in Table 3.1.

Table 3.1: Table over variables in ReqParent.

Variable name	Description
`correct_sequences`	`correct_sequences` initiated to an empty list
`incorrect_sequences`	`incorrect_sequences` initiated to an empty list
`invalid_sequences`	`invalid_sequences` initiated to an empty list
`warning`	`warning` initiated to False
`temp`	`temp` initiated to an empty list

Furthermore, `ReqParent` contains several methods that will be inherited by all requirements which are described below:

- `get_desc()` - This method is used for getting the description of the requirement as a string. This can be overwritten in the child class.
- `get_sequence_length()` - A method for getting the length of the sequence that should be tested against the requirement.
- `prereq()` - Method for catching potential errors in the data that prevent the test from being run.
- `error_handler()` - A method that saves the found error in a systematic way.
- `setup()` - Method for filtering edge-cases before `analysis()` is executed.
- `test_req()` - A method that verifies the `correct_sequence`, `incorrect_sequence` and `invalid_sequence` against the requirement. When this function is executed, it will print out if the requirement gave an appropriate response to the sequences and how many of each sequence group was tested. If it did not, the output will return which sequence that did not pass the test as expected.

3.5.4 Testing requirements

This section and its subsections describe the main functional part of the testing program. It consists of a series of functions that should be run from the main file. They test all the data against all the requirements and provide prints describing the testing procedures and the results.

3.5.4.1 test_requirements()

The first thing to loop over is `test_req()` in `ReqParent` to control if the requirements correctly respond to the sequences `correct_sequences`, `incorrect_sequences` and optionally the `invalid_sequences`. This function is used to check if the requirements are correctly written before we input any real data. It provides a print of which requirements passed the tests and how many sequences were tested. If a requirement does not pass the test it provides information about what sequence did not pass and why.

3.5.4.2 get_req_desc()

This function provides a print of the description of all requirements currently tested in a readable manner. This is done by looping over `get_desc()` in all requirements.

3.5.4.3 run_requirements()

This function will go through the analysis of every requirement shown in Fig. 3.3. It goes through all combinations of `subsystem` and `parameter` and tests all corresponding values and perform the analysis once per index up to the length of the sequence. The length of the sequence is found using `get_sequence_length()` in `ReqParent`.

3.5.4.4 get_tested_req_count()

As mentioned each parameter is a list of elements or a list of a list of elements. Each element of the first list is tested once using the `analysis()` in the requirement. This function prints out how many times each requirement has been evaluated. It prints out the number of passed tests out of the total number of elements for the parameter and prints it out so that it is easily understandable. It also provides a summary of how many requirements passed the tests, how many requirements passed with a warning, and how many failed.

3.5.5 Validating testing program

To validate and make sure that the testing program runs as expected, a set of data was created to mimic real-world data. This was done using two functions, namely:

- `generate_correct_data()` - This is a function developed with guidelines from Saab that should mimic real-world data. The generated data is in the same form that is used in testing and with no inserted errors.

- `change_specific_data()` - This function takes the data, and changes a specified parameter to a given value.

It also contains `destroy_data()`, which is a function that changes the given data by randomly applying the following functions with a given probability:

Table 3.2: A table over inner functions in `destroy_data()`.

Function name	Description
`remove_message()`	Removes the message.
`duplicate_message()`	Duplicates the message.
`swap_message()`	Swaps the message with next message.
`set_element_to_zero()`	Sets element to 0.
`set_element_to_random()`	Sets element to a random integer.
`change_element()`	Change element with another element.
`set_list_to_zeros()`	Set all elements in list to zeros.
`increase_list_size()`	add a random integer to the end of a list.
`decrease_list_size()`	Remove the last element in a list.
`randomize_list()`	Randomize all elements in a list.
`do_nothing()`	Do nothing.

3.6 Output

When the testing program is executed, it gives us results in the form of prints. What these prints look like and what they mean is described in Section 4.1.

The importance of the printed messages are:

- **Finding Problems** - The messages can tell if something went wrong and give clues about why it happened. This helps to fix any issues or locate a faulty subsystem.
- **Verifying Test Results** - the messages are compared to the expected outcomes to ensure consistency. If there is a mismatch, it indicates an issue that needs attention.
- **Documentation of Test** - The messages serve as a record of the test's execution, offering valuable insights for reference and analysis.
- **Improving the Tests** - If the same issues appear frequently in the messages, it might mean that what is being tested has to be re-evaluated, or that there is something wrong with the hardware.

4

Analysis

This chapter goes through the implemented solution and evaluates the performance of the implemented solution for connecting requirements to test cases in an analysis.

4.1 Solution overview

The solution overview is presented in figures 4.5, and 4.6. Fig. 4.5 shows the structure of the input to the implemented testing program, whose structure is shown in Fig. 4.6. The components of each requirement are coded by the end user of the program and can be an arbitrary number of requirements. The data with its messages and its parameters are given from running the physical radar system, exporting and parsing data, and are imported into the testing program. When running the program, the data should meet all the requirements without raising errors.

The first function that should be executed after creating the requirements is `test_requirements()` which is further explained in Section 3.5.4.1. This ensures that the requirements are correctly written. The output will state if all requirements are correctly written or what is wrong as illustrated in Fig. 4.1

```
Outcome for test of sequences per requirement:
Req1: Number of tested sequences: (Correct: 4 Incorrect: 4 Invalid: 2). Passed the tests.
Req2: Number of tested sequences: (Correct: 8 Incorrect: 6 Invalid: 0). Passed the tests.
Req3: Number of tested sequences: (Correct: 2 Incorrect: 3 Invalid: 0). Passed the tests.
Req5: test not passed for correct sequence 0: {'video': {'exi_nr': [1, 1, 1, 1, 2, 2, 2], 'pri_nr': [0, 1, 2, 3, 1, 1, 2]}}
Req6: Number of tested sequences: (Correct: 2 Incorrect: 3 Invalid: 0). Passed the tests.
Req7: Number of tested sequences: (Correct: 2 Incorrect: 3 Invalid: 0). Passed the tests.
Req8: Number of tested sequences: (Correct: 2 Incorrect: 2 Invalid: 0). Passed the tests.
Req9: Number of tested sequences: (Correct: 2 Incorrect: 3 Invalid: 0). Passed the tests.
Req11: Number of tested sequences: (Correct: 2 Incorrect: 2 Invalid: 0). Passed the tests.
Req18: Number of tested sequences: (Correct: 4 Incorrect: 3 Invalid: 0). Passed the tests.
Req19: Number of tested sequences: (Correct: 8 Incorrect: 12 Invalid: 0). Passed the tests.
Req21: Number of tested sequences: (Correct: 2 Incorrect: 3 Invalid: 0). Passed the tests.
Req25: Number of tested sequences: (Correct: 1 Incorrect: 1 Invalid: 0). Passed the tests.
```

Figure 4.1: Output from `test_requirements()`.

After testing the functionality of the requirements, it is then possible to print out what the requirements are testing according to their written descriptions as illustrated in Fig. 4.2.

```
/code_based_test/main.py
These requirements will be tested:
sc:
        exi_nr:
                Req1: sc exi_nr should iterate with 1 for each index
                Req9: sc exi_nr should be between 0 and 65536
                Req18: For every exi in sc there should exist only one ant message with same exi
        nof_pri:
                Req6: SC nr of PRI should be within 1-10
        nof_samples:
                Req7: SC nr of samples should be within 1-10
        PRI_length:
                Req25: sc PRI_length should be between 0 and 10 for all subexi
102:
        exi_nr:
                Req1: 102 exi_nr should iterate with 1 for each index
ant:
        exi_nr:
                Req2: exi_nr for ant at given index should have corresponding exi_nr in SC at any index
                Req3: ANT EXI nr should only appear once
        status:
                Req8: Status for ANT should be True if nr of PRI for SC is out of bounds at corresponding EXI
video:
        exi_nr:
                Req2: exi_nr for video at given index should have corresponding exi_nr in SC at any index
        pri_nr:
                Req5: VIDEO PRI nr should be 0 for new EXI numbers
                Req11: PRI nr should iterate with one while EXI is same
        i:
                Req19: video i should not be all zeros
        q:
                Req19: video q should not be all zeros
101:
        test_value:
                Req21: 101 test_value should iterate with 1 for each index
```

Figure 4.2: Output from `get_req_desc()`.

After running that function it is possible to run `get_tested_req_count`, the output of which is a summary of what requirements have been tested and how many times each has passed. This is illustrated in Fig. 4.3

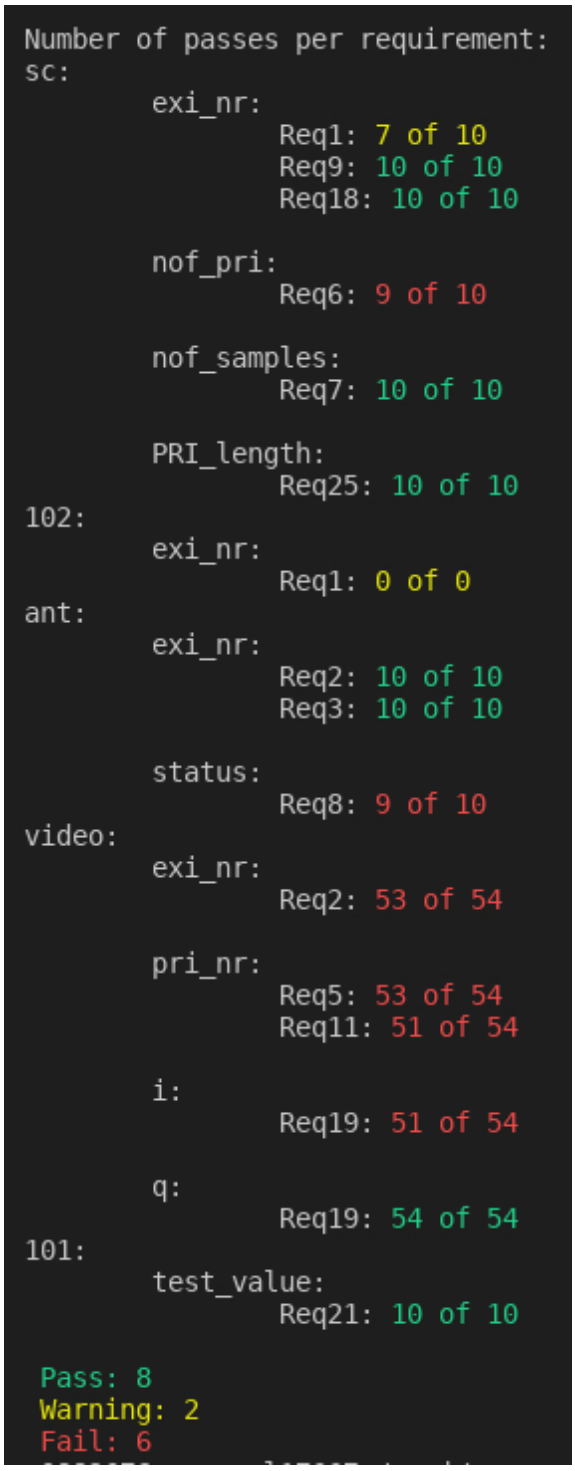

Figure 4.3: Output from `get_tested_req_count()`.

The function `run_requirements` can also be run after `test_requirements` is executed. This function prints a mote technical and detailed description of which requirements are not met and where,

```
Error by EXI:
{0: {'sc': [],
     'ant': [],
     'video': [20, 58, 63],
     'found': {'Req2': {'video': {'exi_nr': {'desc': ['0 not in sc EXI '
                                                      'numbers'],
                                             'index': [[20, 58, 63]],
                                             'subcount': [3],
                                             'count': 3}}},
               'Req5': {'video': {'pri_nr': {'desc': ['5 != 0',
                                                      '1 != 0',
                                                      '6 != 0'],
                                             'index': [[20], [58], [63]],
                                             'subcount': [1, 1, 1],
                                             'count': 3}}}},
     'inserted': {'video': ['video exi_nr changed to 0 from 25942',
                            ['video exi_nr changed to 0 from 25948'],
                            ['video exi_nr changed to 0 from 25948']]}},
 25942: {'sc': [2],
         'ant': [2],
         'video': [15, 16, 17, 18, 19, 21, 22],
         'found': {'Req5': {'video': {'pri_nr': {'desc': ['6 != 0'],
                                                 'index': [[21]],
                                                 'subcount': [1],
                                                 'count': 1}}}},
 25943: {'sc': [3],
         'ant': [3],
         'video': [23, 24, 25, 26, 27, 28, 29, 30, 31],
         'found': {'Req7': {'sc': {'nof_samples': {'desc': ['11 not within '
                                                            '[1-10]'],
                                                   'index': [[3]],
                                                   'subcount': [1],
                                                   'count': 1}}},
                   'Req5': {'video': {'pri_nr': {'desc': ['1 != 0'],
                                                 'index': [[23]],
                                                 'subcount': [1],
                                                 'count': 1}}},
                   'Req11': {'video': {'pri_nr': {'desc': ['0 != 2', '2 != 1'],
                                                  'index': [[24], [25]],
                                                  'subcount': [1, 1],
                                                  'count': 2}}}},
         'inserted': {'sc': ['sc nof_samples changed to 11 from 10'],
                      'video': ['video message swapped with next',
                                ['video q randomized to [143, 199, -205, -66, '
                                 '-16, 184, 233, -37, 156, 6] from [140, -24, '
                                 '153, -49, -208, -229, -144, -107, 99, '
                                 '96]']]}},
```

Figure 4.4: Output from `run_requirements`.

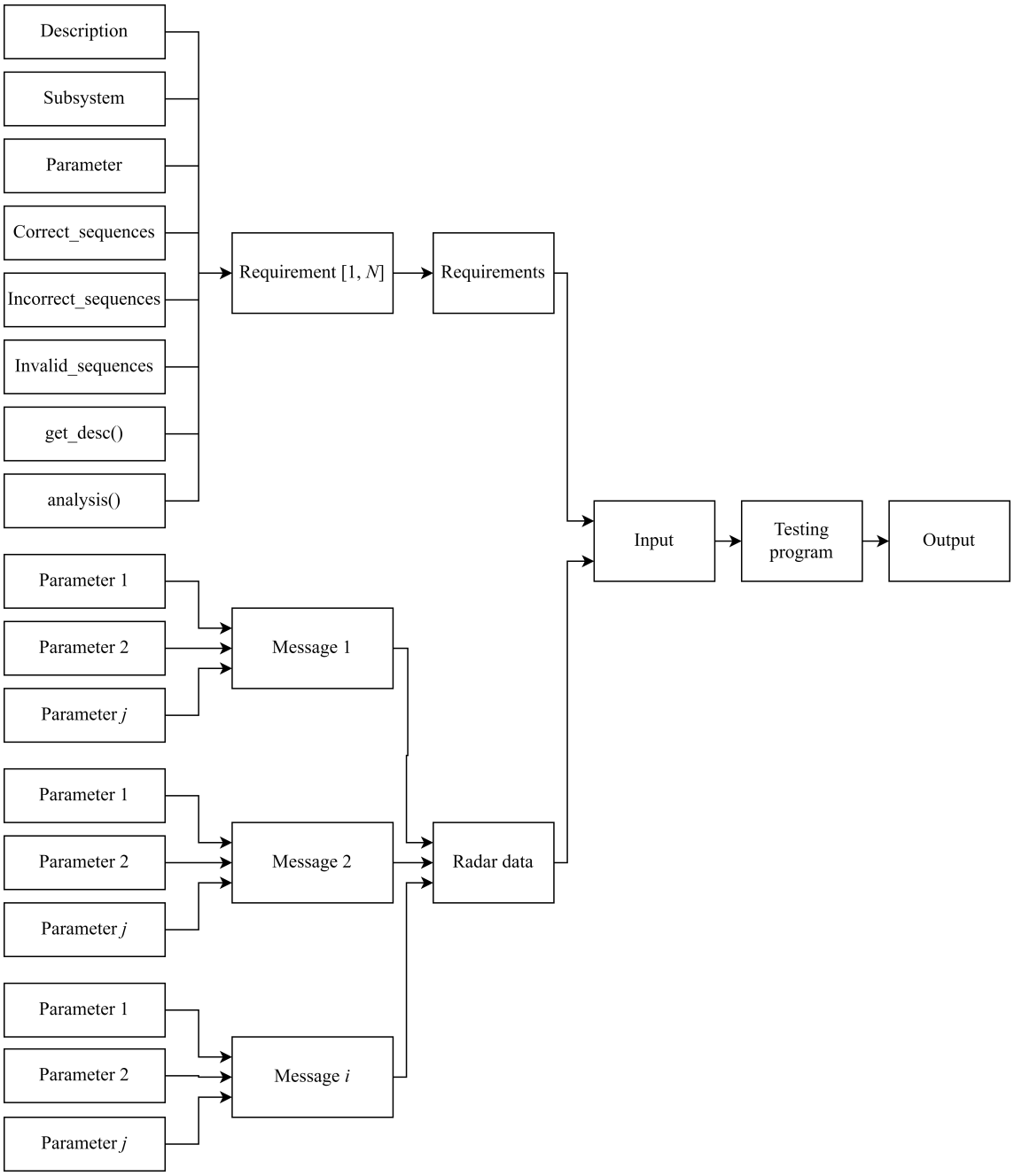

Figure 4.5: Implemented solution overview.

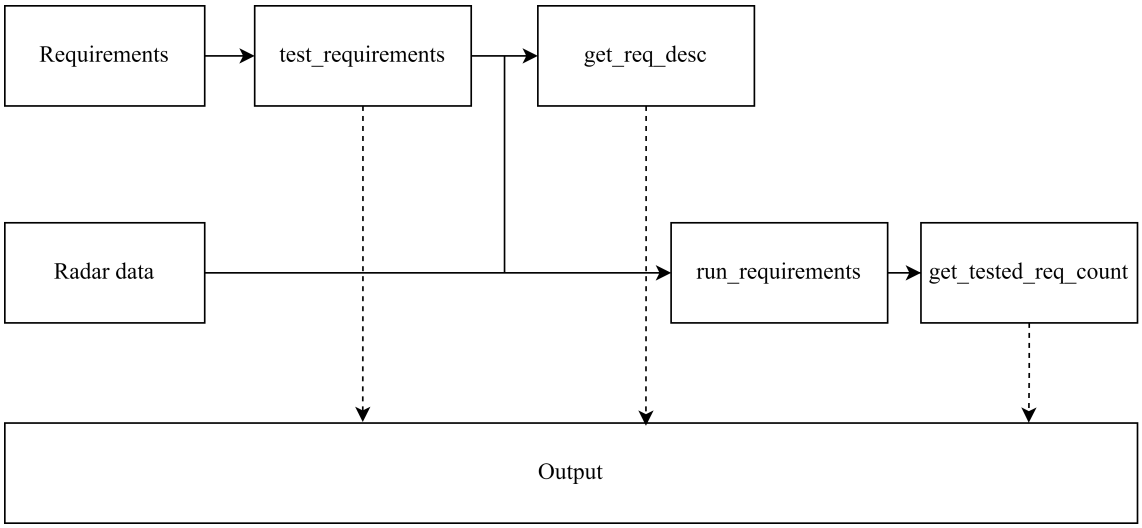

Figure 4.6: Testing software overview.

4.2 Advantages of code-based requirements

The objective was to establish a closer relationship between requirements and test cases. When a requirement changes, the test cases associated with that requirement will be modified accordingly. This reduces the likelihood of divergence between test cases and requirements.

By representing requirements in a code-like format, there are several advantages. They can be integrated into traditional test cases, allow for concise summaries, and provide a revision history of the requirement. Consequently, they can be reviewed similarly to how code review is conducted.

With the previous solution at Saab, which was mostly manual work, it was often problematic that when a requirement changed, one or more test cases also needed to be modified to match the new requirement. However, with the developed approach where requirements are directly linked, changes automatically propagate through the testing chain. This significantly reduces the risk of requirements and test cases diverging due to overlooked updates.

Also, test cases were often designed in a manner where a specific known execution was required, and the analysis would then be adjusted to match this execution. In contrast, the developed program operates on the premise that the analysis should remain consistent regardless of the input data used.

4.3 Strengths and weaknesses of the implemented program

In traditional practices, connecting software requirements to test cases was mostly a manual effort. Although functional, this approach occasionally lacked speed and precision. The introduction of the new software aims to address these limitations.

A new method, termed "code-based requirements for software verification," has been adopted. This method aspires to enhance the reliability of system testing and offers a streamlined adjustment process. With this setup, changes in software requirements can automatically adjust associated tests without extensive manual intervention.

One notable advantage of this approach is its adaptability. Previously, alterations in software requirements necessitated manual adjustments in testing methods. With the current system, changes in requirements prompt automatic updates in corresponding tests, thanks to the direct linkage between the two processes.

Recording requirements in this manner also facilitates easy tracking of modifications and expedites review processes.

A backside of this approach to writing requirements is that the process of actually coding the requirements directly takes longer time to do, as opposed to simply writing into PDF format.

4.4 Societal, ethical and environmental analysis

With the implementation of code-based requirements societal, ethical, and environmental aspects, both beneficial but also detrimental, appear and have to be analyzed.

4.4.1 Societal impact

Collaboration and Communication

Code-based requirements encourage collaboration and open communication between developers, product managers, and stakeholders. As the requirements are directly embedded in the code, all team members have a clearer understanding of the functionality being developed, reducing misunderstandings and promoting better teamwork.

Transparency and accountability

Code-based requirements offer transparency as the requirements are visible to all team members. This transparency fosters accountability, making it easier to track changes and understand the reasons behind them. It can also help in identifying any potential security or privacy concerns.

Inclusive development contra increased technical complexity

When requirements are integrated into the code base, it becomes easier for developers to consider and address accessibility and inclusion concerns from the early stages of development. This leads to software that is more usable and caters to a wider range of users. However, the implementation also leads to an increased technical complexity that in turn makes it harder for non-developer stakeholders to understand and participate in the development process. This could also lead to

reduced collaborations and stand in the way of effective communication between team members.

Support for rapidly changing requirements

In today's fast-paced world, requirements can change frequently. Code-based updates allow developers to adapt to these changes swiftly, reducing the time lag between identifying new needs and implementing solutions.

4.4.2 Ethical impact

Integrity and trustworthiness

Code-based requirements contribute to the overall integrity and trustworthiness of the program. As requirements are directly traceable and connected to the implementation, it helps ensure that the program meets the intended functionality and avoids potential manipulation or misinterpretation of requirements.

Security and privacy

By incorporating requirements directly into the code, developers can ensure that security and privacy considerations are implemented properly. This approach makes it easier to identify potential vulnerabilities early in the development process, reducing the risk of data breaches and other security issues. However storing requirements directly in the codebase might also expose sensitive information or security-related details to unintended audiences, increasing the risk of security breaches or privacy violations.

User-centric design

Code-based requirement updates enable developers to take a user-centric approach to development. Directly embedding requirements allows for quicker feedback and iterations based on user testing and feedback, leading to programs that better meet users' needs and expectations.

4.4.3 Environmental impact

Efficient development process

Code-based requirements streamline the development process, enabling faster iterations and deployment. This efficiency can lead to reduced development time, resulting in a lower overall environmental impact.

Increased program maintenance complexity

Without proper documentation or separate requirement files, maintaining the code can become more challenging, potentially leading to longer maintenance cycles and higher resource usage during updates.

5

Conclusion

The summary of this master's thesis and recommendations for future works are presented in this chapter.

5.1 Conclusion

The previous method to connect requirements to test cases has successfully been switched out to a more efficient solution where testing using code-based requirements is implemented. Defining the requirements in code makes the system more reliable, and up to date with new requirements that easily can be adapted into the code that defines those requirements.

The solution works by testing data with respect to the requirements in the proposed environment. The code was written in Python which is a high-level programming language, this makes it suitable for a fast-changing environment where new requirements can easily be added to the list of requirements without having to implement and test them manually.

The program was created so that it takes the requirements written in code and the data that those requirements should apply to and runs them through the main part of the program. The main part of the program is responsible for running the requirements in the correct places in the data and clearly printing the result of each requirement.

When analyzing the societal, ethical, and environmental aspects of the implementation there are more benefits than drawbacks, such as the support for rapidly changing requirements, obtaining an efficient development process, and the robustness of the system. However, it is important to always consider the privacy and security of the platform where the requirements are placed so that they are stored securely.

5.2 Future work

As we look to future improvements and refinements of the testing software, one primary consideration is the incorporation of a feature that allows for the seamless exporting of requirements to a well-formatted PDF document. This capability is particularly pertinent for a range of stakeholders, including system managers and

representatives from associated subsystems. These groups often emphasize the importance of having documentation that is readily accessible and easy to read. By offering a polished PDF version of the requirements, we can cater to those who may not have direct access to, or familiarity with, a development environment to execute the code. Furthermore, presenting requirements in a PDF format can enhance overall readability and accessibility, making it easier for a broader audience to engage with the content.

The next phase in the program's adaptation entails the development of a graphical interface. Preliminary efforts towards this objective have been initiated, with a beta version currently in operation. This will help developers to faster learn how to use the program and it will also help make it easier to operate with visual clues.

References

[1] Arthur M. Langer. *Guide to Software Development. [electronic resource] : Designing and Managing the Life Cycle*. Springer London, 2016. ISBN: 9781447167990. URL: https://search.ebscohost.com/login.aspx?direct=true&db=cat07472a&AN=clec.SPRINGERLINK9781447167990&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[2] A.K. Ganguly. *Embedded Systems : Design, Programming and Applications*. Alpha Science International, 2014. ISBN: 9781783320462. URL: https://search.ebscohost.com/login.aspx?direct=true&db=cat07472a&AN=clec.EBC5288028&site=eds-live&scope=site&authtype=guest&custid=s3911979&groupid=main&profile=eds.

[3] Ahman Emmanuel Onoja, Abdusalaam Maryam Oluwadamilola, and Lukman Adewale Ajao. "Embedded system based radio detection and ranging (RADAR) system using Arduino and ultra-sonic sensor". In: *American Journal of Embedded Systems and Applications* 5.1 (2017), pp. 7–12.

[4] JM Headrick and JF Thomason. "Applications of high-frequency radar". In: *Radio Science* 33.4 (1998), pp. 1045–1054.

[5] Stephen H Hall. *High-speed digital system design—A handbook of interconnect theory and design practices*. 2000.

[6] Kenneth L Kaiser. *Electromagnetic compatibility handbook*. CRC press, 2004.

[7] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools*. " O'Reilly Media, Inc.", 2006.

[8] Vishram S Pandit, Woong Hwan Ryu, and Myoung Joon Choi. *Power Integrity for I/O Interfaces: With Signal Integrity/Power Integrity Co-Design, Portable Documents*. Pearson Education, 2010.

[9] Radia Perlman. *Interconnections: bridges, routers, switches, and internetworking protocols*. Addison-Wesley Professional, 2000.

[10] Bob Lightsey. "Systems engineering fundamentals". In: *Defense acquisition univ ft belvoir va* (2001).

[11] Jeffrey O Grady. *System validation and verification*. Vol. 12. CRC Press, 1997.

[12] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012.

[13] Steven Lott. *Functional python programming*. Packt Publishing Ltd, 2015.

[14] Glenford J Myers et al. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

A

Tables over simple functions

Table A.1: Table over functions for checking data.

Function name	Input	Returns
`check_equal`	expected, actual	True if 'actual' equals 'expected'; otherwise, returns a descriptive error string
`check_not_in_list`	in_list, value	True if 'value' is not in 'in_list'; otherwise, returns a descriptive error string
`check_not_true_or_false_in_list`	in_list	True if True or False is not in 'in_list'; otherwise, returns a descriptive error string
`check_value_in_list`	in_list, value	True if 'value' is in 'in_list'; otherwise, returns a descriptive error string
`check_less_than`	actual, upper_bound	True if 'actual' is less than 'upper_bound'; otherwise, returns a descriptive error string
`check_greater_than`	actual, lower_bound	True if 'actual' is greater than 'lower_bound'; otherwise, returns a descriptive error string
`check_less_or_equal`	actual, upper_bound	True if 'actual' is less or equal to 'upper_bound'; otherwise, returns a descriptive error string
`check_greater_or_equal`	actual, lower_bound	True if 'actual' is at least 'lower_bound'; otherwise, provides an error description.
`check_list_values_in_range`	in_list, upper_bound, lower_bound	True if all 'in_list' elements are between 'lower_bound' and 'upper_bound'; else, gives an error description
`check_list_values_equal`	in_list, expected	True if all elements in 'in_list' is equal to 'expected'; otherwise, returns a descriptive error string

Table A.2: Table over functions for structure and readability

Function name	Input	Description
get_first_index_by_value	data, msg, value,	Returns the index of the first occurrence of 'value' in data[msg][value]
get_indices_by_value		Return a list of indices where 'value' appears in data[msg][value]
logical_xor		Returns the logical XOR for 2 booleans
get_corresponding_value		
get_value_by_index		Retrieve the value from 'lst' at the specified 'index'.
random_sign		Returns 1 or -1 with a 50% chance each.
merge_dicts		Merge multiple dictionaries into a single dictionary.
order_dicts_by_keys		Sort a list of dictionaries based on their keys in ascending order.
get_next_dict_value		
get_first_value_by_exi		Returns True if all elements in 'in_list' is equal to 'expected'; otherwise, returns a descriptive error string
pretty_print		Uses the built in function pprint to print data in a readable manner
defaultdict_to_dict		Converts defaultdict to dict
custom_sort		A function for custom sorting when using both integers and strings
deep_defaultdict		A function creating a defaultdict of arbitrary depth
sort_value_by_key		Function for sorting values in a dict by its key.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY