Model-Based Testing of Internet of Things Protocols

Xavier Manuel van Dommelen

xavier_vd@outlook.com

December 9, 2021, 51 pages

Supervisor at Axini: Dr. Ir. Machiel van der Bijl, vdbijl@axini.com
Host organization: Axini
https://www.axini.com/nl/
Supervisor at the UvA: Professor Andy D. Pimentel, a.d.pimentel@uva.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

Internet of Things (IoT), systems that have obtained the ability to connect and interact with each other through a network. These systems are currently being deployed all over the world and generally implement the same accepted communication protocols. However, manufacturers of these IoT systems are struggling to extensively test the conformance of these protocols. As a result, it becomes possible that systems are released on the market that do not conform to the standard of these protocols. Preventing these flawed systems is important because conformance mistakes can influence the interoperability between systems or even introduce security vulnerabilities.

To overcome the challenges from the IoT protocol testing domain, research has started looking into different methods. One of these methods is model-based testing. However, research thus far has mainly been looking into if it is possible to utilize model-based testing and not at why should we use model-based testing. For this reason, the goal of our research is to demonstrate to what extent model-based testing is a viable option for the IoT protocol testing domain.

In our work, we highlight that model-based testing should be able to overcome 4 out of the 5 challenges from the IoT protocol testing domain based on existing literature. From these findings, we then subtract 5 expectations on how model-based testing will perform in this domain. To obtain further insight, we create and experiment with a proof of concept that is capable of partially testing a widely-used IoT communication protocol, the Bluetooth Low Energy (BLE) protocol, using a model-based testing approach. Using our proof of concept, we are then able to demonstrate that we can find flaws in the official BLE specifications as well as finding conformance mistakes on a certified BLE system. Finally, by performing three experiments using our proposed testing environment, we are capable of confirming that 3 out of our 5 expectations also hold in the IoT protocol testing domain.

Contents

1 Introduction **4**
- 1.1 Research Questions . . . 5
- 1.2 Outline . . . 5
- 1.3 Contributions . . . 5

2 Background **6**
- 2.1 Internet of Things Protocol Context . . . 6
- 2.2 Internet of Things Protocol Testing . . . 7
  - 2.2.1 Conformance Testing . . . 7
  - 2.2.2 Interoperability Testing . . . 7
- 2.3 Model-Based Testing . . . 7
  - 2.3.1 Goal . . . 7
  - 2.3.2 Concept . . . 8
  - 2.3.3 Modeling Language . . . 8
  - 2.3.4 Test Generation . . . 9
  - 2.3.5 Supporting Infrastructure . . . 10

3 Related Work **11**
- 3.1 Different Directions . . . 11
- 3.2 Applied Model-Based Testing in the IoT Testing Domain . . . 11
- 3.3 Summary . . . 12

4 Combining Knowledge Domains of Model-Based Testing and IoT Protocol Testing **13**
- 4.1 IoT Protocol Testing Challenges . . . 13
  - 4.1.1 Low Budget . . . 13
  - 4.1.2 Changing Standards . . . 13
  - 4.1.3 Protocol Configurations . . . 14
  - 4.1.4 Platform Variance . . . 14
  - 4.1.5 Distributed Hardware . . . 14
  - 4.1.6 Discussion . . . 15
- 4.2 Model-Based Testing . . . 16
  - 4.2.1 Advantages . . . 16
  - 4.2.2 Disadvantages . . . 17
  - 4.2.3 Discussion . . . 17
- 4.3 Discussion . . . 18
  - 4.3.1 Combining Knowledge Domains . . . 18
  - 4.3.2 Threats to Validity . . . 19

5 Implementing Bluetooth Low Energy Protocol Within AMP **21**
- 5.1 Axini Modelling Platform . . . 21
  - 5.1.1 Motivation . . . 21
- 5.2 Bluetooth Low Energy Protocol . . . 21
  - 5.2.1 Motivation . . . 21
  - 5.2.2 BLE Specification Version . . . 22
  - 5.2.3 Host Control Interface . . . 22
  - 5.2.4 Layer Coverage . . . 23

5.3 Model . . . 23
5.3.1 Link Layer Coverage . . . 23
5.3.2 Model Overview . . . 24
5.3.3 Constraints . . . 24
5.3.4 Macros . . . 25
5.3.5 Model Configurations . . . 26
5.3.6 Findings . . . 27
5.3.7 Limitations . . . 27
5.4 Adapter . . . 28
5.5 Test Architecture . . . 28

6 Testing BLE using AMP **30**
6.1 Assumption . . . 30
6.2 Experimental Setups . . . 30
6.2.1 System Under Test . . . 30
6.2.2 Test Criteria . . . 31
6.2.3 Conformance Experiment . . . 32
6.2.4 Quantity Experiment . . . 32
6.2.5 Model Assumption Experiment . . . 32
6.2.6 Experiment Overview . . . 33
6.3 Test Results . . . 33
6.3.1 Conformance Experiment . . . 33
6.3.2 Quantity Experiment . . . 35
6.3.3 Model Assumption Experiment . . . 36

7 Discussion **37**
7.1 Implementing BLE Within AMP . . . 37
7.2 Testing BLE using AMP . . . 37
7.2.1 Threats to Validity . . . 38
7.3 Meeting with the Creator of Bluetooth . . . 39

8 Conclusion **40**
8.1 Findings . . . 40
8.1.1 Research Question 1 . . . 40
8.1.2 Research Question 2 . . . 41
8.1.3 Overall . . . 41
8.2 Limitations and Threats to Validity . . . 42
8.3 Future work . . . 42

Bibliography **44**

Appendix A Specification Flaws **47**
A.1 UnderSpecification . . . 47
A.1.1 Link Layer Protocol . . . 47
A.1.2 Error Precedence . . . 48
A.1.3 Reserved for Future Usage Parameters . . . 48
A.1.4 *Ignored* Key Word . . . 48
A.2 Inconsistent Specification . . . 48
A.2.1 Contradictory Specifications . . . 48
A.2.2 Set Scanning Two Resulting State Versions . . . 48

Appendix B Conformance Errors **50**
B.1 Assumption Related Errors . . . 50
B.1.1 Self Loops . . . 50
B.1.2 Reserved for Future Usage Parameters . . . 50
B.1.3 Set Scanning Two Resulting State Versions . . . 51

Chapter 1

Introduction

In the past decade, the term *Internet of Things (IoT)* has become well known. IoT generally refers to everyday objects that have obtained the ability to connect and interact with each other through a network [1]. Over the years, the number of these IoT devices has grown tremendously, reaching an approximate amount of 9.9 billion devices in 2021 [2].

Along with this growth, new IoT devices are being developed that often implement the same widely accepted communication protocols [3]. Examples would be Bluetooth Low Energy [4] and Zigbee [5]. Because these systems utilize the same communication protocols, it is important that these protocols are implemented according to their specifications. When the implementation deviates from the specifications, the functionality to interact with other systems implementing the same communication protocol could be jeopardized.

In addition to managing communication between systems, these protocols are often also intended to introduce an additional layer of security to the systems [3, 6]. To achieve this, it is also important that such protocols are implemented according to their specifications. This is because doing this incorrectly could introduce security vulnerabilities otherwise [7].

Currently, manufacturers face several challenges that prevent them from extensively testing their IoT devices and thus protocols [8]. For this reason, research has started looking into different testing approaches to overcome these challenges. One of these approaches is the evolving testing methodology of *model-based testing* [9]. An argument that is used to support the use of this approach is that it is often suggested that model-based testing yields better and cheaper tests than other strategies [10].

Research on model-based testing for IoT protocol testing looks mainly into proof of concepts and investigates individual challenges [11–13]. As a result, it is difficult to evaluate to what extent model-based testing is capable of resolving the problems in this domain as a whole. Such an evaluation is needed to compare testing approaches to determine the most optimal approach. For this reason, our research tries to investigate to what extent model-based testing is able to resolve these challenges and what other influences this approach brings.

To this end, we will look into the current challenges in the IoT protocol testing domain and the advantages and disadvantages of using a model-based testing approach according to literature. By combining these two knowledge domains we will obtain expectations on how model-based testing would fit in the IoT protocol testing domain. Furthermore, we will create and use a proof of concept that is capable of testing a part of the Bluetooth Low Energy protocol using a model-based testing approach. Based on our experience and findings we then investigate if our expectations also hold in the IoT protocol testing domain.

1.1 Research Questions

The goal of our research is to investigate to what extent model-based testing is a viable option to resolve the current issues with testing IoT protocols. To achieve this goal, our research will seek to answer the following questions:

RQ 1) What problems could model-based testing resolve in the context of testing IoT protocols according to literature?

RQ 2) To what extent can a deployed model-based testing approach assist in solving the problems relating to the testing of IoT protocols in practice?

- **RQ 2.1)** How can we test an IoT protocol using a model-based testing approach?
- **RQ 2.2)** How does an applied model-based testing approach for testing IoT protocols compare to our findings in RQ1?

1.2 Outline

Manufacturers are currently struggling to extensively test IoT protocols that are implemented within their systems [8]. To investigate to what extent model-based testing can assist within this domain we first provide the reader with the essential background knowledge in Chapter 2. In this chapter, we discuss the context of IoT protocols, what kind of testing is relevant to these protocols, and what model-based testing is.

In Chapter 3, we highlight related research that has been done on model-based testing and IoT protocols. We then emphasize what knowledge is missing within these papers to further position our research within this domain.

The purpose of Chapter 4 is to highlight what the current challenges of testing IoT protocols are and how model-based testing would fit within this domain according to literature. For this purpose, the chapter discusses the current challenges of testing IoT protocols and the promised benefits and disadvantages of model-based testing individually. A discussion then pairs the current challenges of IoT with our found expectations of using a model-based testing approach. These expectations are further studied in the remaining chapters.

Chapter 5 discusses how we create a model-based testing platform that is capable of testing Bluetooth Low Energy and reflects our experience back to our findings from Chapter 4. The resulting platform is used in Chapter 6 wherein we perform additional experiments to further evaluate our findings from Chapter 4.

The results from Chapter 5 and Chapter 6 are discussed in Chapter 7. Additionally, in this chapter, we discuss our meeting with the creator of Bluetooth to whom we presented our work. Finally, we come to a conclusion in Chapter 8.

1.3 Contributions

Our research makes the following contributions:

1. An analysis on how model-based testing fits in the IoT protocol testing domain according to literature.
2. An overview of encountered challenges and possible solutions when developing a testing environment that applies model-based testing to test an IoT protocol.
3. Experiments showing to what extent a deployed model-based testing approach can assist in solving the problems relating to the testing of IoT protocols in practice.
4. An overall evaluation to what extent model-based testing can resolve the challenges in the IoT protocol testing domain.

Chapter 2

Background

This chapter will present the necessary background information for this research. First, we will give background on the Internet of Things (IoT) protocols in general. This will give us the context of the evolution and nature of IoT protocols for when we discuss the IoT protocol challenges. Then we will look into relevant testing methodologies based on the found characteristics that we will refer to in the next chapters. Finally, we will give a more in-depth overview of what model-based testing is and which technologies are generally required to create a testing environment using this approach.

2.1 Internet of Things Protocol Context

As mentioned in the introduction, most of the time, IoT refers to everyday objects that have obtained the ability to connect and interact with each other through a network [1]. The work of Elnashar [14] discusses the evolution of IoT and its use within the real world. They mention that the challenges relating to IoT devices can be categorized into two main categories; challenges relating to unlicensed networks that aim for short-range communication and challenges relating to cellular licensed networks.

For our research, we are interested in the unlicensed category, because the number of different manufactures is significantly larger within the unlicensed domain compared to the cellular domain that consists solely of tech giants [14]. Having more manufacturers requires more developers to follow the same protocol specifications. When there are more parties working on implementing the same specifications on different systems it becomes more likely that mistakes, and thus errors, are introduced. In this scenario, a generic testing environment becomes more important.

The work of Vorakulpipat et al. [6] discusses the evolution and importance of information security within the IoT domain. In this work, they mention that an IoT device should handle the security on the system itself, for example how it handles memory access and network security. They continue to explain that the security of IoT devices has gone through two generations. The first generation of IoT devices did not have a standard. This resulted in people making their own protocols which as a consequence made it difficult to control all the devices and evaluate their security. In the second and current generation, new protocols were introduced. However, in contrast to the first generation, some of these protocols became an accepted community standard. Developers began to utilize these standards due to their easy access, simple implementation, and the established confidence of security by the community.

In the end, for our context, we decided that the IoT systems will consist of unlicensed systems that use short-range communication to communicate with each other. This decision will scope our selection of possible IoT systems and thus protocols to investigate later on. Furthermore, we found that the systems within this IoT domain are developed by a large number of different parties [15, 16]. This makes a generic testing environment more relevant. Finally, we found that because these IoT protocols are also used to ensure software security that an extensive testing environment that checks for a correct implementation becomes crucial.

2.2 Internet of Things Protocol Testing

In the previous subsection, we discussed the evolution of IoT and its nature. Because IoT systems are unique by their nature certain testing methodologies become more important. The upcoming paragraphs will introduce these testing methodologies and link them back to the previously described nature.

2.2.1 Conformance Testing

We mentioned that IoT manufacturers usually make use of the same community accepted protocols for their systems to ensure security [3, 6]. Previous work has demonstrated that when a manufacturer does not correctly implement their protocols, it can lead to heavy security breaches [7]. As a result, it is crucial to develop systems that follow the specifications correctly. This consequence makes the conformance testing methodology important.

In the case of conformance testing, the tester evaluates to what extent the implementation of a standard follows its specification [17–19]. This method should not only evaluate the default behavior but also edge cases where the specification allows the user to make their own decisions. An example would be the possibility to set different configuration combinations as a user.

A note should be made when using conformance testing in the context of testing IoT communication protocols. Because we only evaluate the correctness of the implementation we do not guarantee that two systems will be able to communicate. We only guarantee that the implementation follows the defined specification.

2.2.2 Interoperability Testing

By definition, IoT systems are systems that are capable of communicating with each other as discussed in Section 2.1. This characteristic makes *interoperability testing* an important testing methodology for IoT [8, 20, 21]. In this methodology, we determine if a target system implementing a standard communication protocol can interact with other products implementing that same protocol. This testing method does not check if the implementation follows the standard but creates test cases based on the assumption that the two systems implement the same protocol.

Furthermore, interoperability testing is generally seen as a complementary approach to conformance testing [20]. This is even more relevant because in our use case we have several manufacturers that can implement the same protocol slightly differently. To still test if these systems can connect with each other we need to not only test for conformance, but also for interoperability.

2.3 Model-Based Testing

Before we can discuss model-based testing as a solution it is important to establish what model-based testing is. To accomplish this, we will discuss the goal and overall idea of model-based testing after which we will discuss the different components that are required for a model-based testing implementation.

2.3.1 Goal

In order to say anything about model-based testing, we need to take a step back and look at software testing in general. This is because software testing encompasses model-based testing and we want to define certain characteristics of software testing that are of importance for model-based testing. Software testing is frequently perceived as an essential but tedious part of the development process [22]. Additionally, this part of the process often takes a substantial amount of development time [23], making an efficient approach more beneficial. In this process, a developer tries to evaluate if a System Under Test (SUT) works as specified according to the specifications.

Such an evaluation can be done in four steps[24, 25]: specification interpretation, test creation, test execution, and test result evaluation. These steps can be done by hand but can also be (partially) automated. Using test scripts that are made by a tester that has a clear understanding of the specification, the steps to execute the tests and evaluate the test results can be automated. Model-based testing takes this automation a step further by also introducing automatic test generation, leaving only the specification interpretation as a non-automated step. This additional layer of automation as well as the

model-based testing approach on its own bring certain benefits and disadvantages which we will discuss in more depth in Section 4.2.

2.3.2 Concept

Model-based testing is an evolving technique for automatically generating a suite of test cases from requirements for a SUT [9, 26]. This is done by formally defining a model that conforms to the specifications of the SUT. As a result, the model describes what kind of input and output the SUT can expect and uses this formal definition to generate test cases to evaluate the correctness of the SUT. A schematic overview of this technique is displayed in Figure 2.1.

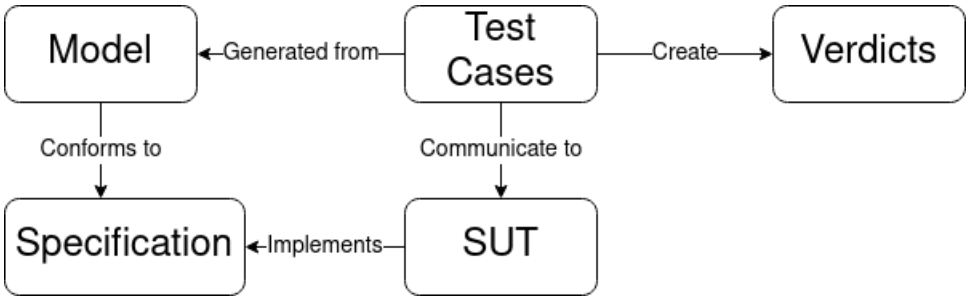

Figure 2.1: Model-based testing pipeline [25]

This approach is a form of black-box testing [27]. With black-box testing, a tester looks at the SUT as a *black box* because the tester only monitors the input and output. Therefore, model-based testing does not evaluate internal aspects but rather tests the input and output conformance of the system [28, 29]. In other words, with black-box testing and thus model-based testing we do not look at the "how" but rather at the conformance of the implementation.

Finally, an implementation of a testing environment using a model-based testing approach generally requires three key technologies [9]; Modeling Language, Test Generation, and a Supporting Infrastructure. These technologies will be described in the remainder of this chapter and can be used as reference when we discuss our implementation in Chapter 5.

2.3.3 Modeling Language

As described previously, model-based testing requires a formal model on how the SUT should behave. This is where the first key technology, namely the modeling language, comes into play. Currently, there are several semantic rules from different theories a modeling language could follow to define a formal model [26]. Examples of such theories are: Finite State Machine (FSM), Labelled Transition System (LTS) [29], Unified Modeling Language (UML) [30], and Symbolic Transition System (STS) [31].

From this list, the rules from the LTS theory are often applied to the modeling language that is used within model-based testing tools to model telecommunication and protocol systems [26]. Furthermore, STS is an extension of the LTS theory that introduces a notion of data within the model making it an interesting choice.

As we will explain in Section 5.1, we will experiment with a model-based testing tool that applies the STS theory for the creation of the models. For this reason, we will henceforth focus on explaining the LTS and STS theory.

Labelled Transition System

Since LTS is fundamental for STS we will briefly describe it. With LTS, a SUT can be modeled using transitions between states where each transition is complemented with labels [29]. These labels describe observable input and output of the SUT which can be used to describe the transitions between different states of the SUT. Putting this together, a model can be created consisting of states where each transition describes an action or response of the SUT. An action that is executed by the SUT is also known as a stimulus, while the output is known as a response.

Symbolic Transition System

A problem with LTS and many transition systems is that complex data structures with a large or even infinite data domain can result in an explosion of the state space [31]. This is because each value within the data domain could result in a different state. An example would be an action on a SUT that accepts a value higher than 0 but lower than 5. In this case, 4 transitions would be needed to describe the same action but with different data values.

STS resolves this issue by introducing the concept of data allowing a user to define internal data constraints within a label of a transition. These constraints could for example define a range of possible data inputs for a specific label. Furthermore, the data variables can be updated during transitions allowing you to manage internal variables more precisely. As a result, different states can be grouped together where data variables determine the differences, lowering the possible number of states.

2.3.4 Test Generation

The second technology of model-based testing is related to the steps and algorithms that are required for the generation of tests. Since our tests should be based on the defined behavior of the model the verdicts of the tests are also already defined. This is because the model describes the allowed input and expected output behavior on which a verdict should be based. As a result, a test generation algorithm could come up with an infinite number of tests for which a verdict can be made based on the received output [10, 32].

Test Criteria

Due to economical reasons, it is often not possible to generate an infinite amount of tests with a model-based testing environment. For this reason, a manufacturer should try to reduce the generated tests to a limited number that is as low as possible [10]. However, a manufacturer should also strive to guarantee a certain degree of quality. To achieve this, manufacturers need to find a way to generate a *good* test set. In this context good tests are cheap to derive, execute, evaluate and are capable of finding *frequent* and *crucial* failures.

To limit the amount of generated tests while still trying to guarantee the quality, a manufacturer defines test generation criteria [10, 32]. Examples of such criteria are Transition Coverage and Data Coverage [26].

In this context, Transition Coverage refers to the percentage of the tested and evaluated transitions within a model. Data Coverage refers to the percentage of the tested input domains for every transition.

IOCO Theory

A commonly used theory for model-based test generation is the Input-Output Conformance (IOCO) theory by Tretmans [29]. The IOCO theory allows a developer to formally check if a set of actions, also known as a *trace*, on the SUT conform to the specifications as defined in an LTS. This theory also comes in a different form that uses a distinct representation of the traces in case data variables and parameters are involved. By using the different representation, the IOCO theory can also accept an STS as valid input to test for conformance. This version of the theory is also known as the Symbolic IOCO (SIOCO) theory[27].

Because the SIOCO theory does not alter or extend the IOCO theory, the same definition holds[29]. The definition states that a SUT SIOCO-conforms to a model, an STS, if and only if:

$\forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$

Where

1. *Straces* is the set of all finite traces
2. *s* is the specification as described within an STS
3. *i* is the implementation which is also known as the SUT
4. *i* **after** σ is a set composed of all states that are reachable from *s* after the trace σ
5. *out(s)* is the set of all possible output transitions in *s*

Using this definition of conformance, we are now capable of formally generating test cases for a SUT that conform to the specifications as described within an STS.

2.3.5 Supporting Infrastructure

The final technology describes the tools that support the infrastructure for the tests. In our context, this refers to an adapter that allows the SUT and the model-based testing environment to communicate with each other. The communication between these two systems consists out of two main steps:

1. The first step is the translation between a label and an action of the SUT. For example, if there is a state transition with the label to open a door then this label should be parsed into the corresponding command on the SUT that opens a door. Of course, the opposite should also be possible; for example, translating an output of the SUT into a label that is defined in the model.
2. The second step consists of handling the communication between the two systems. After an action is created for the SUT by the model-based testing platform, this action needs to be sent to the adapter that in return will pass it to the SUT after translating the received label. Again, the opposite should also be possible.

Chapter 3

Related Work

Currently, there are several papers published that discuss model-based testing within the IoT protocol testing domain. In this chapter, we highlight these papers and discuss how our work differentiates itself from previous work.

3.1 Different Directions

Our work is focused on investigating to what extent model-based testing can be applied to test the IoT protocol conformance on IoT systems. However, model-based testing can also be applied to test other aspects of the IoT protocol domain.

For example, model-based testing can be used to evaluate the robustness of an IoT communication protocol. The work of Yoneyama et al. [33] utilizes model-based testing to test the robustness of the COAP protocol by modeling network faults. By modeling network faults they obtain a simulation environment to see how the protocol behaves in different scenarios such that they can evaluate its robustness.

Furthermore, the work of Aziz et al. [34] demonstrates that by formally modeling the MQ Telemetry Transport protocol, an IoT protocol, and analyzing the result, they can evaluate the correctness of the protocol. This research comes closer to what we discuss in Section 5.3 but differs in the sense that they focus on the correctness of the protocol, while we discuss the completeness of the protocol specifications. An example would be that they discuss how an attacker could abuse the protocol based on the specifications while we focus on underspecifications that could result in implementation differences.

Then Malik et al. [35] use model-based testing as a tool to demonstrate that we can automatically test IoT protocols on systems remotely. In their work, they briefly describe why they make use of model-based testing but their main topic is the framework to overcome the challenge of remotely testing IoT systems and their protocols. Our work differs by focusing on the first step of their research, namely why use model-based testing in the first place.

3.2 Applied Model-Based Testing in the IoT Testing Domain

In the case study of Tappler et al. [11], it is shown how models for a model-based testing approach can be automatically created through active automated learning. Furthermore, this work demonstrates that by using their automatically generated models they are capable of finding implementation mistakes that go against the protocol specifications. For their experiments, they focus on creating and testing an IoT protocol, namely the MQTT communication protocol. In the end, their research shows a proof of concept that is capable of automatically creating models that can be used within model-based testing to find conformance errors. However, this research does not draw a complete image about to what extent this approach would be beneficial and effective within this domain. This is because, in their work, they mainly discuss the found conformance errors and do not relate their approach back to the different challenges within the IoT protocol testing domain.

Ahmad et al. [12] investigate the possibility to use a model-based testing approach to test IoT systems in their entirety. In addition to just testing the system, they discuss a framework that enables sharing models between developers as a service. For their use case, this is important because they focus on the universal IoT system FIWARE. This allows them to distribute models across all developers that make use of this system. Overall, their research shows that it is possible to use model-based testing to test an IoT system with a service-oriented solution. First, our work separates itself from their work by focusing on the possibility to use model-based testing to test an IoT protocol, and not the entire system. Secondly, their work is still limited in their evaluation. In their research, they discuss some challenges to test IoT systems and how model-based testing would fit in this domain. However, they do not discuss and test additional limitations or benefits from the model-based testing approach.

The work of Inçki et al. [13] presents a model-based testing implementation in which they could perform interoperability tests to evaluate the IoT communication protocol COAP. However, they do not present any experiments that make use of their presented approach. Consequently, we are not able to evaluate the benefits or disadvantages of this approach. Furthermore, in contrast to our research, they do not give an in-depth explanation and reflection on why they use model-based testing. This makes it more difficult to decide if this method would be sufficient to be a solution for this domain. Nonetheless, the fact that they were able to create an implementation to test interoperability for an IoT communication protocol suggests that it would also be possible for other IoT protocols which we can use as an argument within our research.

3.3 Summary

First, we found that there are different research paths that can be looked into when investigating model-based testing in the IoT protocol testing domain. Examples would be testing the robustness or correctness of such protocols. Our work sets itself apart from these papers by focusing on testing the conformance of IoT protocols on the implementation of SUTs. Secondly, we found that previous research that also looks into this direction quickly decides that model-based testing is an appropriate approach and mainly discusses the findings of their approach. As a result, it is difficult to evaluate if their approaches would actually resolve all the problems within the field of testing IoT protocols. In this context, our research tries to resolve this knowledge gap by investigating what needs to be resolved first and then connect our experiments to evaluate these findings. Additionally, we investigate the benefits and disadvantages of model-based testing that fall outside of these findings to draw a complete image.

Chapter 4

Combining Knowledge Domains of Model-Based Testing and IoT Protocol Testing

In this chapter, we will investigate what the current challenges are for testing the implementation of IoT protocols and how Model-Based Testing would fit in this picture.

4.1 IoT Protocol Testing Challenges

4.1.1 Low Budget

Several papers [8, 11, 20, 36] discuss the challenge surrounding the subject of keeping the testing costs low for IoT products.

The work of Bures et al. [8] explains that, due to large competitiveness in the IoT market, it is necessary for the price of IoT products to go down to stay relevant. As a consequence, there would be less and less budget for extensive testing. This makes it difficult for manufacturers to extensively test their IoT systems.

Furthermore, in the research from Marinissen et al. [36] they mention the influence of the low price of the IoT systems. They discuss that manufacturers often make the choice to run older technologies that have been proven to be reliable before, with the goal to save costs on an extensive testing environment. They mention that if a new extensive testing framework would be introduced, that the price of these cheap IoT systems could double or even triple. This statement would fall in line with the idea that the increased costs of a testing framework could jeopardize the sales of a manufacturer as mentioned previously by Bures et al. [8].

In the end, budgeting forms a challenge for obtaining an extensive testing environment. To overcome this challenge a low-cost testing environment is necessary that is also capable of testing extensively.

4.1.2 Changing Standards

The work of Taivalsaari et al. [37] mentions the dynamic nature of IoT systems and their protocols. They mention that currently, the speed at which protocols are receiving an update has increased. This increased speed creates a scenario in which manufacturers need to keep updating their testing environment to accommodate to the new updated standards. This scenario brings a challenge to the conformance testing environment [20] which needs now to be updated more frequently.

The work of Ziegler et al. [21] continues on this topic by discussing the consequences from a manufacturer standpoint. In their work, they reason that creating additional tests requires additional time and that this time loss can result in a scenario in which competitors can release their product earlier. As a consequence, a manufacturer can lose clients to other manufactures that promote that they already have

implemented the new specifications because they decided to not implement extensive tests for these new specifications.

Because of this, an adaptable testing environment where we can easily change and add tests to accommodate for the updated protocol specifications is requested [20].

4.1.3 Protocol Configurations

Another challenge that is discussed in literature is the large number of configuration combinations that can be used within an IoT protocol implementation [8]. To completely test for interoperability we would need to test all possible configuration combinations between two systems. As a result, a lot of additional testing coverage is introduced that should be looked into when testing the interoperability of IoT systems.

The work of Kim et al. [20] also mentions that these large number of configurations raise a challenge for conformance testing. They mention that traditional conformance testing is labor-intensive. When more test cases are introduced, such as when more configurations are introduced, the amount of labor will grow. This can then slow down rapid development.

A possible solution to deal with this challenge is discussed in literature [8, 20]. This solution would be to have an automated testing environment that could test each of these combinations automatically to lower the human labor for testing such configurations.

4.1.4 Platform Variance

In the research of Bures et al. [8], they mention the problem in which IoT systems are generally not updated over time. They argue that this is because IoT systems are often of low cost and have a low energy consumption. This has as a consequence that IoT systems in a real-world scenario often need to communicate with other IoT systems that implement the same protocol but run different protocol versions.

We also mentioned in Subsection 4.1.1 that manufacturers often decide to implement old technologies on low-cost IoT systems such that they do not have to implement an extensive testing environment [36]. This would have the same effect as Bures et al. [8] previously described in which different systems run different versions of the same technology in a real-world scenario.

Consequently, to be able to test the interoperability between two systems that run different protocol versions we require a testing environment that implements both protocol versions. This scales quickly when you want to accommodate for all the different protocol versions in a world where new protocol versions get introduced more frequently as discussed in Subsection 4.1.2.

The main challenge that arises from this situation is that in a perfect world we would want to interoperability test systems in combination with all the available protocol versions that would be supported. However, this would mean that we would require a testing environment that describes each of these protocols independently creating a lot of additional work. In this situation, it becomes important that a testing environment implementing certain protocol versions can be updated quickly to a different protocol version. This should allow a manufacturer to become more at ease to stay up to date with the latest protocol versions as well as reverting back to older versions for testing purposes.

4.1.5 Distributed Hardware

The work of Kim et al. [20] mentions the location dependency of traditional interoperability testing. This dependency would introduce more human labor because to perform these tests custom reference equipment would be needed. Managing and getting this equipment to one location would then result in a significant delay of time which as a result would make interoperability testing more difficult.

Research from Rosenkranz et al. [38] gives more detail on how this location dependency comes to be. They discuss that manufacturers could decide to change their implementation from the protocol specifications to create optimizations within their implementation. However, when changes are introduced interoperability is not guaranteed anymore with other systems that have a different implementation such as a default implementation that follows the specifications. Because of this, a manufacturer would like to use interoperability testing to test their implementation against all other possible implementations. To

do this the manufacturer would require the reference equipment as discussed in the previous paragraph thus creating the location dependency.

As a result, the challenge is introduced in which we need to overcome the traditional costs of getting the reference equipment in one location for interoperability testing. The research from Kim et al. [20] suggests that a testing framework that allows for remotely testing interoperability could resolve this challenge.

4.1.6 Discussion

Overall, we are able to find different areas of interest when looking for possible challenges in the context of testing IoT protocols. Looking at our found challenges, one challenge, in particular, can be seen as their root cause. This challenge is the low budget challenge that we discuss in Subsection 4.1.1. The other challenges are mostly related to problems that need to be overcome because they otherwise would result in additional labor and therefore require more time and thus money influencing the difficulty of this original challenge.

When we put this together we get the overarching challenge of needing a low-cost testing environment with the following sub-challenges making this more difficult:

1. A quickly changing environment which requires frequent updates to the test cases.
2. A large number of protocol configurations that introduce additional work to obtain complete coverage for conformance- and interoperability testing.
3. The platform variance that requires us to take all the supported protocol versions into account when testing for interoperability.
4. The hardware distribution that introduces more preparation work to get the systems physically in place.

Using these challenges as a basis we can reason about what kind of testing approach and thus environment would be optimal. Based on the first and third sub-challenge, a method that would allow us to easily update the entire testing environment after a specification change would be desirable. This would allow the manufacturer to stay up to date with quickly changing specifications and make it easier to create test cases for all the different protocol versions. The second challenge is mostly related to the large number of test cases that need to be created to obtain a complete coverage that takes the different configuration possibilities into account. A testing approach that could lower the amount of work in creating these test cases would be desirable in this situation. The fourth challenge can not be resolved by a specific testing method since it is caused by the necessity of getting physical systems in place. To tackle this challenge a new approach would be required. An example would be a system that allows us to connect to these systems remotely, removing the required physical aspect.

Threats to Validity A few disclaimers should be made when discussing our findings. First, our findings, and thus proposed challenges, are based on a small sample of literature. Most of the literature that we were able to find briefly describe these different challenges but are not fully dedicated on researching these individual challenges. Because of this it is possible that there are different undiscovered challenges that could further explain our found challenges or show a new direction of challenges entirely. As a result, it is possible that there are other unseen problems. However, our found challenges still contain arguments and a structured reasoning of why they would be relevant, making them a good starting point.

Secondly, our goal was to investigate the general challenges that a manufacturer can encounter when testing IoT protocols. As a consequence, our found challenges are also described for a common scenario making it possible that they are not applicable for every case. It is possible that specific protocols encounter different and even more impacting challenges that should be prioritized before looking at our found challenges.

Finally, our found sub-challenges do not contain any form of priority. Based on our literature we were not able to make any kind of assessment on which sub-challenge would be more important. As a result, based on our findings the reader should evaluate each challenge equally important and draw a conclusion for themselves. In this context, a possible argument could be made that some of these challenges require a similar solution making them arguable more important because one solution could resolve multiple issues.

4.2 Model-Based Testing

In Section 2.3 we discussed the principles of model-based testing. The upcoming subsections will further discuss model-based testing and investigate the potential benefits and disadvantages of using a model-based testing approach.

4.2.1 Advantages

Flexibility

When using model-based testing an assumption that is often made is that the testing environment is flexible to adapt to changing requirements and specifications [10, 26].

An explanation for this assumption would be the fact that with model-based testing you make use of a model that describes the SUT on a high abstract level. It is argued that making changes in this higher level would be easier than manually changing individual low-level test cases when requirements are changed [26].

Pretschner et al. [10] clarify this reasoning further by discussing the fact that after the high-level model is changed, the new low-level test cases that would be automatically generated are also changed. As a result, time could be saved compared to a manual approach. Because of this a testing environment that makes use of model-based testing would be easier to reuse and extend when requirements or specifications change.

Specifications Understanding

It is mentioned that during the creation of the model for a model-based testing environment you are able to get a better understanding of the specifications [10, 26].

Pretschner et al. [10] explain that when a person is forced to thoroughly think about the specifications and thus the system that they will naturally get a better understanding of the system. It is then discussed by Utting et al. [26] that this enforced behavior, as a result, will also create a scenario in which specification- and requirement flaws will be noticed more often.

As such, the creation step in the model-based testing approach allows a developer to get a deep understanding of the specifications. Additionally, during this process, they become more likely to find flaws in the specifications which then can improve the specifications overall.

Cost Savings over Time

According to early literature [10, 26] another assumption that is often made with model-based testing is that it results in a lower cost testing environment when used for a longer time period. However, according to their research, no empirical study has been done yet that proves that this assumption is true.

The work of Apfelbaum et al. [39] argues that this is the case due to the reusability and the previously described flexibility of the model-based testing approach. They discuss that once a model has been created that it can be easily extended when additional requirements and specifications are introduced. This would save time and thus money as the project continues. Furthermore, they mention that these models are a good way of passing knowledge. They discuss that new developers often have difficulties understanding what the SUT should be doing and not just what it is doing. By having an abstract model that reflects this knowledge it will be possible to preserve this knowledge and update it by changing the model.

In 2015, Binder et al. [40] released a paper in which they investigated the perceived usefulness of model-based testing by their practitioners. To obtain this information they created a survey with 32 questions and asked 100 model-based testing practitioners about their opinion. The results showed that 2/3 of the respondents found that the promised cost reduction of using model-based testing was partially or completely satisfied. Of course, this experiment only queried a small number of participants leaving room to find different results when performed on a larger scale. However, finding these results still supports the cost reduction assumption.

In the end, we found literature that discusses and supports the assumption that model-based testing will reduce the costs more and more for creating and maintaining a testing environment as time passes.

Conformance Testing

Another assumption that is often made is that model-based testing results in better tests compared to the manually created tests [26, 40].

In the research of Pretschner et al. [10] they clarify that model-based testing performs specifically well in finding specification errors when compared to other testing approaches. Other research from Pretschner [41] clarifies this further by mentioning that this is due to the large number of tests that can be generated using the model. In their experiment, they demonstrate that when they generate the same number of tests as manually written tests that they are able to find a similar number of errors. However, when they allow the test environment to generate more test cases, thus covering more parts of the specification, the model-based testing approach is capable of finding more errors. Furthermore, because these tests are automatically generated we do not lose time obtaining these additional tests making them priceless.

In short, according to our found literature model-based testing does not necessarily guarantee better quality of tests overall. Nevertheless, it can shine when looking for specification errors when generating a large number of tests to evaluate the conformance of the implementation.

4.2.2 Disadvantages

Initial Investment

In contrast to the long-term cost savings, it is mentioned that model-based testing results in high initial costs [10]. These costs are mostly related to the creation of a correct model. This is because the model needs to be created carefully such that it exactly reflects the specification. As a result, it becomes a time-consuming task and therefore expensive.

Furthermore, it is discussed by Dias [42] that building a model is a time-consuming step because a certain level of expertise is first required by the developer. A developer would require knowledge about the modeling language, testing coverage criteria, generated output format, and supporting tools that would allow the model-based testing platform to interact with the SUT. The survey from Binder et al. [40] further investigates this statement and found that an average of 80 hours is required before becoming a proficient model-based testing user. Furthermore, after becoming proficient with the model-based testing environment a developer would still require to become knowledgeable with the specifications. After which, a delicate process starts during which a correct model needs to be created.

In short, before a developer is able to start with the creation of a model a developer requires to obtain a certain level of expertise. After obtaining this knowledge the time-consuming step of creating a model starts which in total results in a large initial investment when going for a model-based testing approach. Finally, it is important to note that this large initial investment from the modeling step is relative to the other steps within the model-based testing approach and not necessarily to other testing approaches.

Late Specification Flaw Awareness

We mentioned earlier in Subsection 4.2.1 that model-based testing allows a person to get a better understanding of the specifications of a SUT. However, the work of Utting et al. [26] mentions that omissions in the specifications are often found only after the model-based testing environment is implemented. As a consequence, there would be a larger delay before the testing phase to investigate specification flaws could start. This disadvantage is therefore correlated to the previous disadvantage of a large initial investment since the initial time to start with the evaluation of the specifications would also be increased.

4.2.3 Discussion

Overall, previous research demonstrates that model-based testing shines mostly due to its abstract modeling layer and its level of automation. The model that forms an abstract layer of the behavior of the SUT allows for easy flexibility in an ever-changing specification environment. Furthermore, creating such a model allows developers to get a thorough understanding of the behavior of the SUT as well as becoming more likely to find specification flaws in this process. Finally, the capability of automatically generating test cases based on the specification for no additional costs makes it a good approach for conformance testing.

However, as we described earlier, getting to this stage requires a relatively large investment. The initial time that it will take to get a first version of the model to generate the tests will require the most time. Therefore, it will become the most expensive step. Furthermore, because this is a time-consuming step it will take longer before the specifications and the system can be evaluated based on the model that has been created.

Threats to Validity Most of our found (dis)advantages are based on assumptions discussed in previous literature. It is mentioned that currently there is no empirical evidence that can confirm if these assumptions are true or false. Nevertheless, these assumptions are based on logical reasoning and experiences making them more likely to be true. As a result, because these are assumptions we should still make a note that these (dis)advantages might not be correct and might not apply to all cases.

Another note that should be made is about the perspective in which we discuss model-based testing. In the previous subsections, we discuss model-based testing on its own and do not compare it to other testing approaches. As a result, we get an overview that demonstrates what aspects of model-based testing is likely to do well and where it is likely to perform worse. This approach has as a consequence that our research can not be used on its own to evaluate if this testing method is better than other methods. If we want to achieve this, more research should be performed looking into these aspects. However, this additional research falls outside the scope of our project and we will leave it for future research.

Furthermore, due to the scope of our project we were not able to look at all the available literature about model-based testing. As a consequence, it is possible that there are other (dis)advantages that we did not cover in our work. It is also possible that some of our found (dis)advantages are part of a bigger, more generic (dis)advantage. Therefore it is important that the reader takes this into account before making a decision on using model-based testing based on our presented information.

Finally, just as our IoT testing challenges our found (dis)advantages are not ordered. As a result, it might be possible that some of our presented (dis)advantages are more applicable and more noticeable than others. However, because we are limited by the scope of our project we are not able to further investigate this and we will leave this to future research.

4.3 Discussion

4.3.1 Combining Knowledge Domains

When we look at our found challenges related to testing IoT protocols we found that the main challenge that we want to overcome is the challenge of obtaining an extensive testing environment at a low cost. When we combine this with our found knowledge about model-based testing we see some opportunities.

According to our found literature, model-based testing is likely to save costs when used for a long period of time. However, it has the drawback that it requires a large initial investment in the modeling step. This drawback can be overcome when the model of IoT protocols is made public. This could be in a free form where the community works on having a functional model or in a regulated form where the organization that created the protocol presents the community with a model. This approach would be possible because multiple parties require to implement and test the same IoT protocol implementation as discussed in Section 2.1. As a result, these parties would normally need to create the same model but if this model is already available this step can be skipped. Once the main hurdle of the initial costs is overcome the creation of the remaining testing environment should be relatively cheap.

The scenario in which the models are publicly available for the manufacturers could also make it possible to deal with the platform variance sub-challenge. By combining models of different protocol versions it should be possible to easily test two different implementations that implement the same protocol but different versions. However, this is mostly speculation and the actual difficulty and effectiveness of combining such models remain to be seen in the future.

When we dive further into the sub-challenges that were presented, we see that it is generally difficult to keep the testing environment up to date with the frequent updates that are brought to the IoT protocol specifications. This challenge could be overcome using the model, abstraction layer, from the model-based testing approach. Using the model we would only need to make minor changes on a higher abstraction level to automatically generate the new test cases to test a new specification implementation.

As a result, developers would only be required to perform small changes on an existing model to obtain test cases that conform to the new updates saving time and thus costs.

Furthermore, the automatic test generation would also be a solution for the sub-challenge where it would be a lot of work to create test cases for all the different possible configuration combinations. As a result, we would be able to obtain a high level of coverage for no additional costs after the model has been created.

We should also note that model-based testing is not a direct solution for the challenge related to the distributed hardware. To tackle this challenge a new approach should be investigated that would allow us to lower the cost of getting the different hardware systems in place before applying a testing method. In this context, the work of Ahmed et al. [12] demonstrates that using a custom framework, it is possible to remotely test an IoT protocol using model-based testing as its method. This work shows us that model-based testing could be used as a method once such a framework is in place.

Aside from the benefits related to the challenges of testing IoT protocols, the testing method of model-based testing brings some additional benefits.

The first benefit is the improved chance of finding specification flaws. We discussed in Section 2.1 that many manufacturers follow the same specifications for their implementation. This requires the specifications to be as complete as possible if we want the manufacturers to end up with the same implementation. Because of this, it is important that the specifications do not contain flaws or underspecification, leaving unintentional freedom to a developer. In the scenario in which a model would be created, it would become easier to find such flaws and resolve them. This would then have the consequence that the specifications would become better for all the related parties.

Finally, because different manufacturers implement the same protocols, it is important that their implementations conform to the specification to ensure that the resulting systems can communicate with each other. As we saw earlier, model-based testing is capable of obtaining a large coverage due to its automated nature. Consequently, using a model-based testing approach we should be able to ensure conformance of the SUT.

4.3.2 Threats to Validity

Most of our previously drawn conclusions are based on the assumption that the development cost of the testing environment becomes acceptable after removing the modeling step. Even when the majority of the development costs can be dropped it could be possible that the remaining development costs for setting up the adapter would be too expensive for some cases. Furthermore, our found literature did not mention an exact time distribution that would allow us to determine how much time would be saved when leaving the model development out of the pipeline. However, using the experience from our experiment in Chapter 5 we are able to get an idea on one case but this would just be one sample making it not absolute. We suggest that future research should look into a more in-depth empirical study on how the time is distributed when using a model-based testing approach to come up with more concrete numbers.

As we mentioned earlier, our literature research did not discuss any precedence between the presented topics. For this reason, it could be possible that some of our drawn conclusions are more or less applicable than other conclusions that we made. However, even without the precedence, our findings can be seen as a starting point describing which aspects of model-based testing might be effective in the IoT protocol testing domain.

We should also discuss some points relating to our research approach in its entirety. Due to the time constraints of our project we were not able to thoroughly investigate all the papers on both topics. As a result, our findings are based on a small sample of literature. Furthermore, it must be noted that the available literature that specifically discusses the problems in the IoT protocol testing domain is still limited. Additionally, the lack of empirical studies that look into model-based testing forced us to base most of our conclusions on assumptions and logical reasoning as was done in prior research. As a result, it could be possible that some of our drawn conclusions could be proven to be different in the future.

It is also important to reflect back on our original goal and how this influenced the content of our research. Our original goal was to combine the IoT protocol testing challenges at the manufacturer's side with the model-based testing knowledge domain. Because of this, we looked at the challenges of how

we could evaluate and test an implementation of IoT protocols. As a result, this work can not be used to map model-based testing to the IoT testing challenges domain when you want to test and evaluate the correctness and effectiveness of the protocol itself.

Finally, because our focus is on mapping model-based testing into the IoT protocol testing domain our research can not be used on its own to determine the best testing approach. Our work could be combined with other research that evaluates other testing approaches to determine if model-based testing would be the best fit. However, we leave this step also to future research.

Chapter 5

Implementing Bluetooth Low Energy Protocol Within AMP

The goal of this chapter is to describe how a model-based testing approach could be applied to create a testing environment for an IoT protocol. For this purpose, we will create a testing environment using the Axini Modeling Platform (AMP) that is capable of testing a part of the Bluetooth Low Energy (BLE) protocol. The upcoming subsections will describe our most important design decisions, our personal experience for each of the components of the resulting testing platform, and an overview of the overall architecture of this testing platform.

5.1 Axini Modelling Platform

As mentioned at the beginning of this chapter we are going to use AMP as our testing tool. This tool has been created by Axini[1]. Additionally, we mentioned in Section 2.3 that a model-based testing approach generally requires three different key technologies.

For the first technology to describe the model, AMP makes use of the Axini Modeling Language (AML). For each language construct within AML, a formal translation to a (part of an) STS is defined. Therefore, each process defined in an AML model is translated to a complete STS. Because of this translation, the STS theory can be applied for test generation using any model written in AML.

Secondly, AMP applies the SIOCO theory to create a formal translation between the model and actions on the SUT. This translation is then used as the underlying theory for the test generation once a model has been defined.

Finally, AMP supplies its testers with a *plugin-adapter* framework. This framework acts as the middle man between the adapter and AMP to manage the communication between these two parties.

5.1.1 Motivation

Overall, the decision to utilize AMP as our testing tool is mostly based on our affiliation with Axini as our host organization. However, their toolchain applies formal theories such as STS and SIOCO making it a powerful tool to use for our use case. Furthermore, this tool has been proven to be applicable in practice giving us more insurance that we can obtain a functional proof of concept during our research.

5.2 Bluetooth Low Energy Protocol

5.2.1 Motivation

We discussed in Section 2.1 that IoT systems generally utilize the same community accepted protocols. When we want to investigate the testing possibilities of model-based testing on IoT systems it would make sense to investigate one of these accepted protocols. This is because these protocols are already

[1] https://www.axini.com/

available on a wide scale of devices as well as that they are known to be implemented by a large number of manufacturers.

To investigate what kind of protocol we want to implement for our proof of concept we looked at the work of Al-Sarawi et al. [3]. They discuss that IoT systems need to deal with several constraints such as processing capability, storage volume, short power life, and radio range. Because of this, they reason that these systems require a communication protocol to efficiently take these constraints into account. In their work, they then discuss several widely accepted communication protocols. From their list of communication protocols, we decided to test the BLE protocol because it was the most accessible to us.

5.2.2 BLE Specification Version

Now that we have decided on a protocol, it is important to decide on a specification version that we would like to model and test. Currently, the Bluetooth protocol and therefore also the BLE protocol are managed by Bluetooth SIG[2]. As of writing this thesis, Bluetooth SIG has published version 5.2 [43] of the Bluetooth specifications. Naturally, we would like to model this specification version to provide the community with the most recent model. However, we found that we do not have easy access to products that implement this protocol version due to its recent release in 2019.

In the situation where we would decide to still model the newest version, we would end up in a scenario where we could not test a system using our model. As a consequence, we would not be able to evaluate our resulting testing environment, limiting our findings. For this reason, we decided to model the protocol using an older specification version. Using an older version limits this model's use case for new products that want to use the most recent protocol specifications. However, this decision allows us to test a wide range of different systems that are already on the market when using these older specifications. Furthermore, when we encounter underspecification in the older version we could cross-check the specifications of these newer versions to possibly determine the intended behavior. In Subsection A.1.4 we will give an example of how we cross-checked the different specification versions and which uncertainties we resolved in our model.

In the end, we decided to use version 4.2 [4] of the Bluetooth specification for our model. As previously discussed implementing an older version has its benefits but does not suggest an exact version that we should model. We could argue that we should implement the version that is currently being used the most in practice but we were not able to find an overview discussing this. As a result, we went for the version that was the most accessible for us to use, which was version 4.2.

5.2.3 Host Control Interface

When looking into the Bluetooth Core Specification version 4.2 [4] we found that you can access a Bluetooth Controller's capabilities through the Host Controller Interface (HCI). As demonstrated in Figure 5.1, the HCI layer allows a user to command the lower software layers and thus the different Bluetooth controllers.

[2] `https://www.bluetooth.com/`

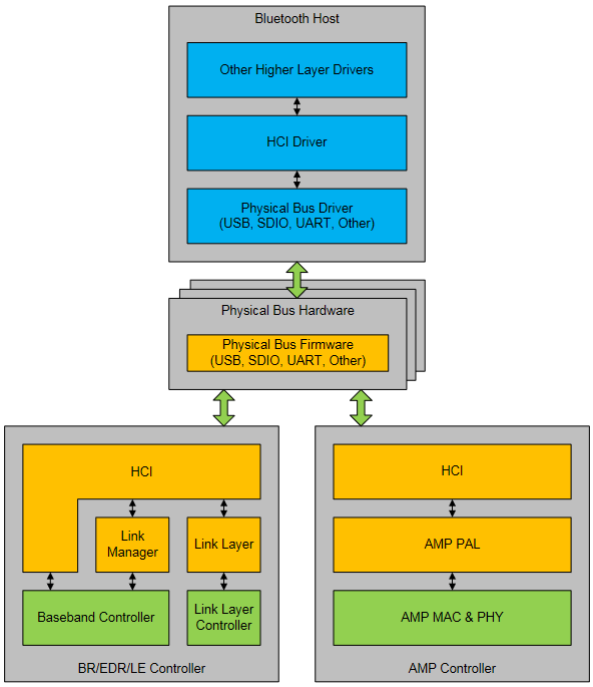

Figure 5.1: Overview of the lower software layers of the Bluetooth Core Specification version 4.2 [4]

For our use case, we are interested in testing the Bluetooth Low Energy controller. This would be a specific controller from the left path in Figure 5.1. The HCI functions as an API to perform different actions on the different lower level software layers. As such we can send commands that would only be executed on the Low Energy Controller allowing us to test this controller and thus the implementation of the BLE protocol.

5.2.4 Layer Coverage

The BLE protocol can be separated into different layers and as such are also separated in the specifications. Each of these layers contributes their own functionality and have their own set of requirements. For the scope of our research, it would be too much work to model each of these layers. As a result, we decided to model just a part of one of these layers, namely the Link Layer.

The Link Layer describes the steps that two systems implementing the BLE protocol should take to obtain and sustain a connection. Consequently, if a manufacturer would make mistakes while implementing this layer it would directly influence the interoperability. Because interoperability is an important factor for IoT systems, as discussed in Subsection 2.2.2, we decided to model this specific layer.

5.3 Model

We will now describe the different steps and decisions that we have taken to create the model to test a part of the Link Layer from the BLE protocol.

5.3.1 Link Layer Coverage

Due to the limited scope of our project, we also made the decision to only model the Link Layer partially. The system's behavior related to the Link Layer on the Low Energy controller can be represented as a state machine. This state machine is displayed in Figure 5.2. Furthermore, the states within this figure that are accessible within our model are marked green.

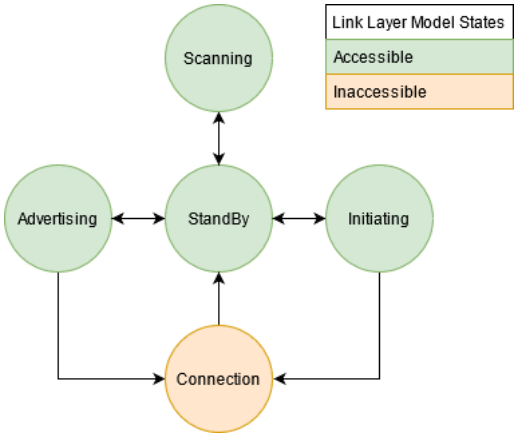

Figure 5.2: State diagram of the Link Layer state machine on the Low Energy Controller according to the Bluetooth Core Specification version 4.2 [4]

As the previous figure shows, we decided not to model the *Connection* state, marked orange. This state would require the most care and introduce the most additional complexity. The additional complexity within the *Connection* state comes forth in the form of modeling the behavior of two systems concurrently and how they would interact. Modeling the behavior from this state would be an interesting direction because it would allow us to directly evaluate interoperability between systems. However, due to time constraints, we leave this to future work.

Instead of trying to model two systems at the same time, we decided to focus on modeling the remaining states and focus on testing the different configuration options. Trying to see if we can model the different possible configurations within these states is important. This is because we found this to be one of the testing challenges as discussed in Subsection 4.1.3.

5.3.2 Model Overview

For the creation of our model, we used the state machine from Figure 5.2 as our starting point. We decided to utilize the same states within our model and search through the specifications to look for the corresponding HCI commands for the basic transitions.

Using the HCI command descriptions as a foundation, we concluded that the following HCI commands would be most applicable to reflect the state transitions:

- HCI_LE_Set_Advertise_Enable) Handles the transitions between the *StandBy* and *Advertising* state.
- HCI_LE_Set_Scan_Enable) Handles the transitions between the *StandBy* and *Scan* state.
- HCI_LE_Create_Connection) Handles the transition from the *StandBy* to the *Initiating* State.
- HCI_LE_Create_Connection_Cancel Handles the transition from the *Initiating* state to the *StandBy* state.

Then to model the different configurations we decided to scope this down to only simulate the configuration options for the *Scanning* and *Advertising* state. In the model, this will be represented using transitions that go towards the same state after successfully changing the configurations. The HCI commands that resemble these transitions are:

- HCI_LE_Set_Advertising_Parameters
- HCI_LE_Set_Scan_Parameters

5.3.3 Constraints

It is important to note that each of the commands and thus transitions within our model has its own restrictions of when and how it should be used. An example would be the allowed input parameters or the states in which the command may be used. Furthermore, the specifications also describe what errors

should be thrown in certain situations. To describe the input domain for each of these cases we make use of the constraints from the STS theory.

The constraints of simple commands with a few restrictions are generally shorter and therefore less complex. An example would be the *HCI_LE_Set_Advertise_Enable* command which for a successful call could get a constraint as shown in Listing 5.1.

```
Advertising_Enable == 1
```

Listing 5.1: Example constraint for a successful call of the *HCI_LE_Set_Advertise_Enable* command

The previous constraint indicates that for this specific transition the *Advertising_Enable* parameter should be set to 1. However, there are also a lot of HCI commands that require more parameters and restrictions resulting in more complex constraints. An example would be when we want to define the input domain for a successful call of the *HCI_LE_Set_Advertising_Parameters* command. The beginning of this constraint is displayed in Listing 5.2.

```
((Advertising_Interval_Max >= param_advertising_interval_min && Advertising_Interval_Max
    <= param_advertising_interval_max &&  Advertising_Interval_Min >=
    param_advertising_interval_min && Advertising_Interval_Min <=
    param_advertising_interval_max && Advertising_Interval_Min <=
    Advertising_Interval_Max && (Advertising_Type == param_advertising_type0 ||
    Advertising_Type == param_advertising_type4)) ...
```

Listing 5.2: Beginning of the constraint for a successful call of the *HCI_LE_Set_Advertising_Parameters* command

As a result of the additional parameters and the more extensive restrictions, the constraints can become rather large and complicated when defining them. However, defining the constraint for a specific transition only needs to be done once after which the testing environment can generate test cases using the entire defined input domain.

Furthermore, we decided to use constraints to also evaluate the output of the SUT. Normally, a tester would define labels for the output but in our use case, these would be integers indicating the status of the executed command. We could decide to define a label for each of these status values. However, this would mean that each of these labels would also require to be defined in the adapter. Since we want to make the required work of developing an adapter as low as possible, to lower the burden for manufacturers we decided to only use one output label named *status*. This *status* label will contain a data value indicating the status code of the executed command. This code will then be used within a constraint in the model to see if the returned status code matches our expected output.

5.3.4 Macros

A scenario that we encountered often while creating our model is that we needed to perform the same kind of transition from different states of the model. Examples would be when we want to change the configurations or check for correct error behavior that could occur in different states. Normally, this would mean that we need to define similar transitions with their corresponding constraint multiple times in different places. If this approach would be taken, our model would become more difficult to maintain and adapt when changes are introduced because the changes should be applied to each occurrence. This is not desired in an environment where changes to protocol specifications are released frequently [37]. As such it is desired that a modeling language has a way to overcome this issue such that the model can reduce duplication making it easier to maintain.

In the context of the modeling language (AML) used within our chosen tool (AMP) we can overcome this issue using macros. With macros, we can define a series of actions and reuse them in different places. Using this, we can define the similar parts within a macro and perform a macro call in the different states. When changes are introduced we now only need to update the macro definition and the changes will be applied to the entire model where this macro is called.

5.3.5 Model Configurations

In addition to just modeling the behavior of the SUT, we added different configuration options for the model. These options allow a tester to manage to what extent the model will be used for the generation of test cases. The list of supported configurations is discussed in Table 5.1.

ID	Configuration	Data Type	Motivation
1	error_paths	Boolean	Scenario Simulation
2	error_self_loop_paths	Boolean	Assumption due to underspecification
3	error_future_param_paths	Boolean	Assumption due to underspecification
4	error_validation_strength	Integer	Assumption due to underspecification
5	scan_between_duplicates	Boolean	Assumption due to underspecification
6	force_link_layer_transitions	Boolean	Assumption due to underspecification

Table 5.1: Model configuration options

Motivation

As can be seen in Table 5.1 there are two different reasons why we implemented the different configuration options. The first reason is to allow a tester to model specific SUT behavior. For example, when a tester is interested to focus only on correct interactions with the SUT and not interested (yet) in testing incorrect behavior. To allow this we added a configuration to enable and disable transitions that would result in an error.

The second reason is because of underspecification. Often, the specifications do not state specifically what the intended behavior of the SUT should be. To still be able to complete the model we have made some assumptions based on other information that is available within the specifications. Nonetheless, a tester might not agree with our reasoning and therefore we have implemented configurations to configure to what extent these assumptions are enforced when testing the SUT.

Usage

Most of our model configurations can be used by setting a Boolean indicating if the related transitions should be looked at. Only the configuration with ID 4 requires an integer which indicates the strength that should be used. We will now further explain each of the model configurations from Table 5.1:

- Configuration 1 allows you to enable or disable transitions that would result in an error code.
- Configuration 2 allows you to enable or disable transitions that could change the state but instead would result in the same state.
- Configuration 3 allows you to enable or disable transitions that would result in an error because parameter values would be used that are reserved for future usage.
- Configuration 4 accepts five different strength values:
  - With a strength set to 0 all error codes are allowed when an error code is expected.
  - With a strength set to 1, only the error codes that are specifically mentioned in the specifications need to correspond to any of the expected errors if multiple errors could be thrown. Otherwise all error codes are accepted.
  - With a strength set to 2 we have the same situation as with strength 1 however we apply our assumption on which error has a precedence when multiple errors could be thrown thus only allowing only one error code.
  - When the strength is set to 3 we only accept one or more of the expected errors. There is no precedence check.
  - Finally, when the strength is set to 4 we only accept the error codes with the highest precedence according to our assumptions.
- Configuration 5 allows you to enable or disable transitions that would move between the two possible scanning states in which *Filter_Duplicates* is enabled or disabled.

- Configuration 6 allows you to enable or disable transitions that would check if transitions that would not be possible according to the link layer specification result in the correct error code.

5.3.6 Findings

For the implementation of our model, we made several scoping decisions with the result that the Link Layer state machine became our starting point. In our experience finding this starting point was quite difficult. This was because the BLE protocol is extensive, over 2500 pages, and does not directly mention a starting point on where to model the SUT's behavior. However, after finding the starting point and defining what we wanted to model, the process of modeling became relatively straightforward. It was straightforward because as we mentioned, the specifications are quite extensive and explain clearly how commands should be used under good circumstances.

Although we have a large amount of information from the specifications available to us we still encountered many situations in which we did not have enough information to determine the intended behavior for many situations. The most common underspecification that we encountered is related to how and when specific errors should be thrown and specifically when such errors are related to the Link Layer restrictions. We will discuss our found individual underspecifications in more depth in Section A.1.

Furthermore, during the development of our model, we were able to find specification descriptions that seemed in contradiction to each other. One example would be the usage of the *HCI_LE_Set_Advertising_Parameters* command in which two parameters are recommended to be set to different values but then the default values are set to the same value. This and other examples are discussed in more depth in Section A.2.

All these findings demonstrate that the BLE protocol specifications still have room to be further improved. Furthermore, these findings show us that if we are encountering these issues that it is also likely that manufacturers need to deal with these flaws. As a result, it becomes more likely that manufacturers unintentionally end up with different protocol implementations. These might influence the interoperability between systems as a consequence. The fact that we are able to find these issues in the modeling step, shows us that not only the resulting testing platform might be useful but that the modeling step can also be used to refine the specifications.

Finally, based on our experience we want to reconfirm the statement that most of the development time goes into developing the model. We mentioned in Subsection 5.3.3 that the constraints for certain transitions can become rather complicated. Furthermore, each command that we want to model could result in multiple transitions with their own input domain. As a result, a model developer needs to think thoroughly about the possible transitions of a command and define an appropriate input domain accordingly. This together becomes a time-consuming task making the development of the model the biggest aspect when setting up a model-based testing environment.

5.3.7 Limitations

The first few limitations of our model have to do with our original approach and lack of time to restructure after we noticed the limitation. These limitations have to do with the input domain within the model for several transitions.

For our model, we decided originally that we only want to define the parameters that are used to execute the command. Normally, to execute an HCI command you require some additional values alongside the parameters of the command itself. These are the OpCode Group Field, Opcode Command Field, and the expected resulting event code. With our current approach, these values should be defined within the adapter creating additional work for a manufacturer. We recommend putting these values as well in the model to save unnecessary work for the manufacturers in the future.

Furthermore, the specifications define a memory size for the values that the HCI commands accept. Currently, we limit the input domain to always fit these memory sizes. However, in a real-world scenario, it is possible that larger values are also given to the command. For this reason, we recommend adding such a feature in the future when modeling the BLE protocol.

Our model also does not keep track of an event stack. There are some HCI commands within the BLE protocol that create additional events after they have been successfully executed. At the moment our model evaluates only the direct responses of the commands and lacks any way to evaluate these additional

events that could be generated at any moment in time. For the future, we recommend keeping track of these events asynchronously and evaluate if these events fall in line with the expected behavior.

Finally, our current approach makes it difficult to model all the different error combinations. At present, if we want to model the input domain for a transition that could result in multiple errors we define the input domain individually. This does not scale well because we need to think about the possible combinations as well as think about the allowed input domains. We recommend looking into a more flexible approach that would allow a model developer to combine input domains of individual errors and generate the combinations by themselves. We believe that such an approach is possible but due to the scope of our project, we will leave this to future work.

5.4 Adapter

We mentioned in Subsection 2.3.5 that the goal of the adapter is to manage the communication between the testing environment and the SUT. For our proof of concept in which we make use of AMP, there is already an integrated framework we can use to manage the aforementioned communication. Using this framework a developer only needs to define the communication between the adapter and the SUT. In other words, a developer needs to define the translations between the model labels and the in- and output of the SUT.

Defining these translations is not much work because we already know which commands should be called based on the label. However, the step of executing a command after receiving a label became a rather tedious task. We found that the documentation about how to program and execute HCI command calls is rather scarce. While searching for documentation we found one book by Huang et al. [44] that discusses that there is almost no documentation on how to program Bluetooth and makes an attempt on explaining this. As a result, obtaining the knowledge on how to actually execute the commands also became a time-consuming task. We expect that manufacturers of BLE systems will not encounter these issues because they would have more knowledge on how to run commands on their own systems. However, in the scenario in which we want to have a model available to the public and have the community also test their own systems and models, this can become an obstacle. It is possible that such documentation is more available for other IoT protocols but we will leave further investigation towards such goals to future research.

In the end, we decided to use PyBluez[3] to interact with the HCI. This decision was made using the reasoning that Python was easier to manage compared to the original library Bluez[4] that uses the C language. In Bluez you are required to manage memory allocation while with PyBluez this is taken out of the developer's hands. However, we mentioned earlier that we made the decision to not model behavior outside of the memory ranges defined in the specifications. Modeling this additional behavior and applying it within the adapter would be more straightforward in Bluez. This suggests that when such an extension would be implemented that an adapter based on Bluez could maybe be developed more easily. Finally, we will leave further research into the most suitable library for future work.

5.5 Test Architecture

After combining the previously discussed components, we obtain the overall test architecture as shown in Figure 5.3.

[3] https://github.com/pybluez/pybluez
[4] http://www.bluez.org/

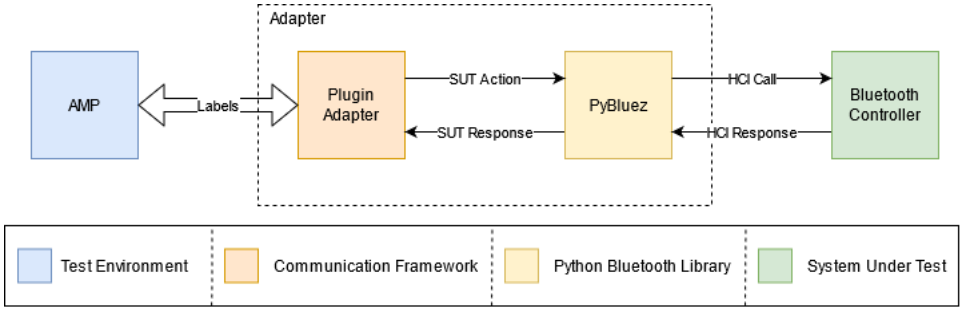

Figure 5.3: Overview of our resulting test architecture that is capable of partially testing the BLE protocol given a SUT

As this figure shows, we have AMP as our testing environment that manages the test generation based on a given BLE model. This testing environment can communicate with the adapter that consists out of two sub-components. The first sub-component, the Plugin Adapter, contains the logic to handle the communication with AMP. Additionally, this sub-component is able to translate the labels into actions on the SUT and vice versa. The second sub-component, PyBluez, is capable of accessing the HCI on the SUT to call the wanted action or to return the SUT's response. Finally, any SUT that contains an implementation of the BLE protocol can be connected to this testing environment.

Chapter 6

Testing BLE using AMP

We will now discuss how we utilized our developed testing environment to partially test the BLE protocol on a SUT.

6.1 Assumption

A fundamental assumption we make for our tests is that we assume that the tool-chain is implemented correctly. This means that we assume that the model, adapter, and the testing environment (AMP) are all implemented correctly and that none of the found mistakes are caused by potential flaws in these components.

Normally, when a bug is found using a model-based testing approach the tester investigates the cause individually and takes appropriate action. However, for our research, we want to demonstrate that we can extend our testing environment to generate and evaluate many aspects of the BLE protocol. As such, we do not want to focus on every bug that is found within all of the resulting test cases and rather look at the fact that we are capable of generating these test cases.

Finally, as a result of our assumption, it is possible that a sample of our found bugs is not caused by an incorrect SUT but rather by a flaw within our toolchain. Nevertheless, we still expect that a larger percentage of bugs that we find are caused by an incorrect SUT. An argument towards this expectation is that our testing environment, AMP, is already successfully being used in real-world scenarios for different clients from Axini, giving us more confidence in the testing environment. Furthermore, AMP is also continuously being tested by Axini, which as a result should lower possible bugs within the testing environment. However, to still demonstrate that we are capable of finding bugs within the SUT, we will work out some examples of found bugs through the use of our model-based testing approach.

6.2 Experimental Setups

6.2.1 System Under Test

The SUT that we will use for every experiment is an *Intel Dual Band Wireless-AC 8265* which supports Bluetooth version 4.2. This SUT has received a certificate[1] from the official Bluetooth SIG organization indicating that they have correctly implemented Bluetooth version 4.2 and thus also have correctly implemented the BLE protocol for this version.

The certificate that was given to the SUT is from 2016-06-26. This means that every mistake we are able to find in the implementation was undetected in the testing environment as it was implemented at that specific moment in time.

[1] `https://launchstudio.bluetooth.com/ListingDetails/3524`

6.2.2 Test Criteria

AMP allows a tester to easily change test generation configurations based on the wanted testing criteria. An overview of these configurations is given in Figure 6.1.

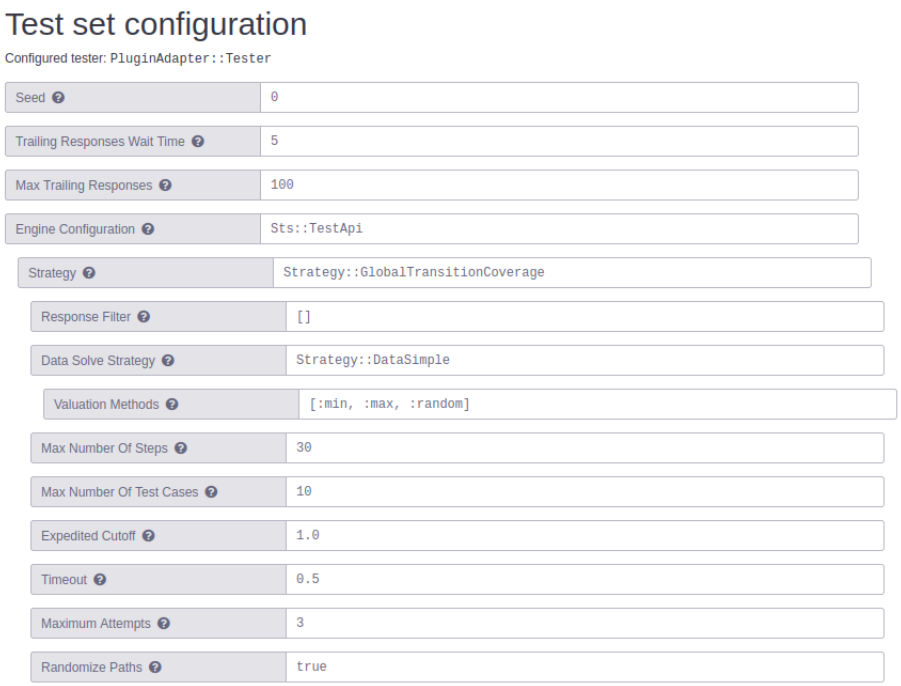

Figure 6.1: Overview of test generation configurations within AMP

For our research, we are interested in the testing criteria that describe the *Transition Coverage* and the *Data Coverage*. An explanation of these terms can be found in Section 2.3.4. We will quickly discuss why these criteria are relevant and which configurations we are going to use to influence them during our experiments.

Transition Coverage

One of the IoT protocol testing challenges that we discussed in Subsection 4.1.1 is the challenge of having a low budget for an extensive testing environment. As a result, it could be possible that such testing environments do not test all the possible actions for all situations that are mentioned in the specification. For this reason, we find it important to demonstrate that we can obtain a 100% Transition Coverage using our testing environment.

Data Coverage

Another IoT protocol testing challenge that we found is the challenge to test every possible combination of the protocol configurations. Using our model, we are able to define each of these combinations using constraints to define the corresponding input domains. Furthermore, to test each combination we need to generate all of these values and combinations using these input domains. When a high Data Coverage is acquired, we have a high coverage of these input domains. This shows that we are able to test most of these combinations. Thus, this demonstrates that we can overcome this challenge.

Test Generation Configurations

To influence the previous two coverage values we can make use of the *Max Number of Test Cases* and *Max Number of Steps* configurations.

The *Max Number of Test Cases* configuration gives an indication of how many unique test cases the testing environment will generate and evaluate. Each test case starts in the initial state as defined in the model after which it can perform *steps* to traverse the model. For each of these steps, also known

as transitions, it will generate a corresponding SUT action. Such an action brings the SUT into a new state after which the response of the SUT is evaluated.

By increasing the number of test cases, we can influence the Transition and Data Coverage. This is because by testing from the starting state more frequently we are able to traverse more paths and thus obtain a higher coverage.

The *Max Number of Steps* configuration sets the maximum number of steps that can be performed within one test case. By increasing this number we could traverse more of the model within one test case. Because all the states within our model are always accessible from any other state using a finite number of steps, we could obtain a 100% Transition Coverage for each test case.

For our experiments, we decided to use a *Max Number of Steps* set to 30. We found that when using this value, we would generally obtain a 100% Transition Coverage for each test case. However, this value was found empirically. Because of this, there may be more optimal values. Optimizing these values falls outside the scope of our research and therefore we will leave it to future work.

6.2.3 Conformance Experiment

In Section 4.2.1 we discussed the benefit of using a model-based testing approach to find conformance mistakes more easily. For this reason, our first experiment aims to show that we are able to find conformance mistakes in the SUT using our testing environment.

For our experiment, we will automatically generate and evaluate 20 test cases. After this step, we will manually categorize the found errors and give a detailed example of one of these errors.

Finally, because our goal is to demonstrate that we can find conformance mistakes, we will set the configurations of the model, such that it does not enforce our own assumptions about the specifications. The reason for this is that we want the resulting test cases and thus evaluations to become non-negotiable.

6.2.4 Quantity Experiment

Our second experiment aims to demonstrate that we can achieve a high Data Coverage using our testing environment to test a BLE protocol implementation. However, our chosen testing platform, AMP, does not have a mechanism to analyze the exact Data Coverage. For this reason, we first need to reason on how we can obtain a high Data Coverage.

In Section 2.3.3, we explained that the input domain for the parameters of a transition can be defined using constraints. With the *Valuation Methods* configuration from Figure 6.1 we can influence how the parameters for each transition will be generated based on this input domain.

By default, AMP tests the edges of the resulting input domain. This means that the first parameters that are generated will use the minimum and maximum values of the input domain. After these parameters have been tested and more tests are generated for the same transition, random values will be used from the input domain.

Putting this together, we conclude that, when using a higher *Max Number of Test Cases* configuration we can obtain a higher Data Coverage. When we have more test cases that result in the same transitions, the test generation will create a larger variety of tests using different values thus obtaining a higher Data Coverage.

For this reason, our experiment will focus on demonstrating that we can run our testing environment with a high value for the *Max Number of Test Cases* configuration. In theory, we can run our testing environment with any arbitrary number for *Max Number of Test Cases* but we will limit ourselves to 1000 test cases. Furthermore, just like the first experiment, we will set the model configurations such that the generated test cases do not enforce any of our assumptions about the specifications.

6.2.5 Model Assumption Experiment

For our third experiment, we want to further investigate our found underspecifications. Within our model, we created flags to enable and disable assumptions that are related to these underspecifications. An explanation about each of these underspecifications can be found in Section A.1.

In this experiment, we want to check whether we can find "mistakes" when we force our underspecification related assumptions to be the correct behavior. In the case that we are able to find "implementation mistakes" related to these assumptions, it will show us that manufacturers can have different interpretations surrounding this subject. Such results could give us more confirmation that our found "underspecifications" are actually underspecifications.

In short, for this experiment, we will automatically generate and evaluate 20 test cases for which we will enable all of our assumption flags in the model configurations. We will then manually go through these results and look for errors related to these flags to see for which assumptions we are able to find an implementation mistake.

6.2.6 Experiment Overview

An overview of our three experiments and their configurations can be found in Table 6.1.

	Max Number of Test Cases	Max Number of Steps	error_paths	error_self_loop_paths	error_future_param_paths	error_validation_strength	scan_between_duplicates	force_link_layer_transitions
Conformance Experiment	20	30	False	False	False	1	False	False
Quantity Experiment	1000	30	False	False	False	1	False	False
Model Assumption Experiment	20	30	True	True	True	5	True	True

Table 6.1: Different experiment configuration setups for our defined BLE model

6.3 Test Results

6.3.1 Conformance Experiment

Running the testing environment with the configurations from the *Conformance Experiment* row, as discussed in Table 6.1, we obtain the results that are displayed in Figure 6.2.

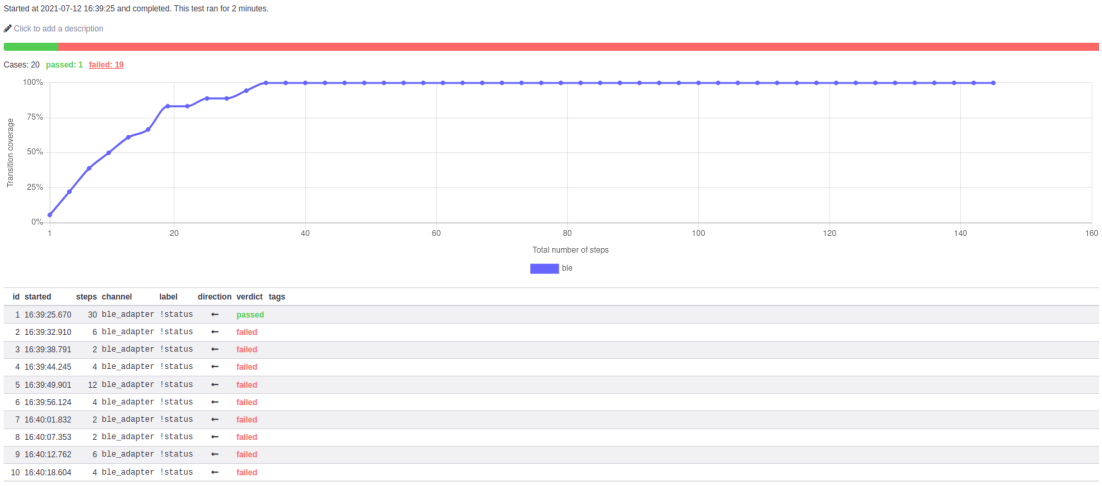

Figure 6.2: Partial overview of the resulting test cases and their evaluations using the configurations as discussed for the *Conformance Experiment*

The results from Figure 6.2 show us that we can obtain a Transition Coverage of 100% as well as to automatically find 19 test cases where the SUT does not conform to our model. Normally, a tester would manually investigate these test cases and decide which to resolve first. Then after resolving a problem, the test environment can be run again to repeat the process. This would be the normal approach, because multiple test cases may be caused by the same implementation mistake. However, the normal approach is not viable for us. This is because our goal is to show that we can find conformance errors in contrast to debugging the entire SUT. To compensate for this, we will give a categorized overview of our found errors in Table 6.2.

Test Case ID(s)	Failed in State	Failed on Label	Expected Output	Received Output
2,4,5,7,15,17,18,20	Scan	setAdvertisingParams	0 (success)	18 (invalid parameters)
3,11,13,14	StandBy	setScanParams	0 (success)	18 (invalid parameters)
8,19	StandBy	createConnection	0 (success)	18 (invalid parameters)
9,12	StandBy	setAdvertisingParams	0 (success)	18 (invalid parameters)
10	Advertise	setScanParams	0 (success)	18 (limited parameters)
16	StandBy	createConnection	0 (success)	13 (limited resources)

Table 6.2: Overview of the failed test cases and their cause using results from the *Conformance Experiment*

This overview shows us that we are capable of finding 6 different conformance mistakes, when we only look at the current state in which the error occurs, the expected input, and the expected output. Furthermore, we see that almost all failed cases are caused by an inaccurate error response about invalid parameters. Nonetheless, it can still be possible that some of these error categories are caused by the same problem or that errors within the same category have a different problem due to different actions that were taken earlier.

When we look into one of these failed test cases more closely, we can obtain more detailed information about the transitions that were taken and where the conformance error occurs. We will use the second test case from this experiment as an example. This example is shown in Figure 6.3.

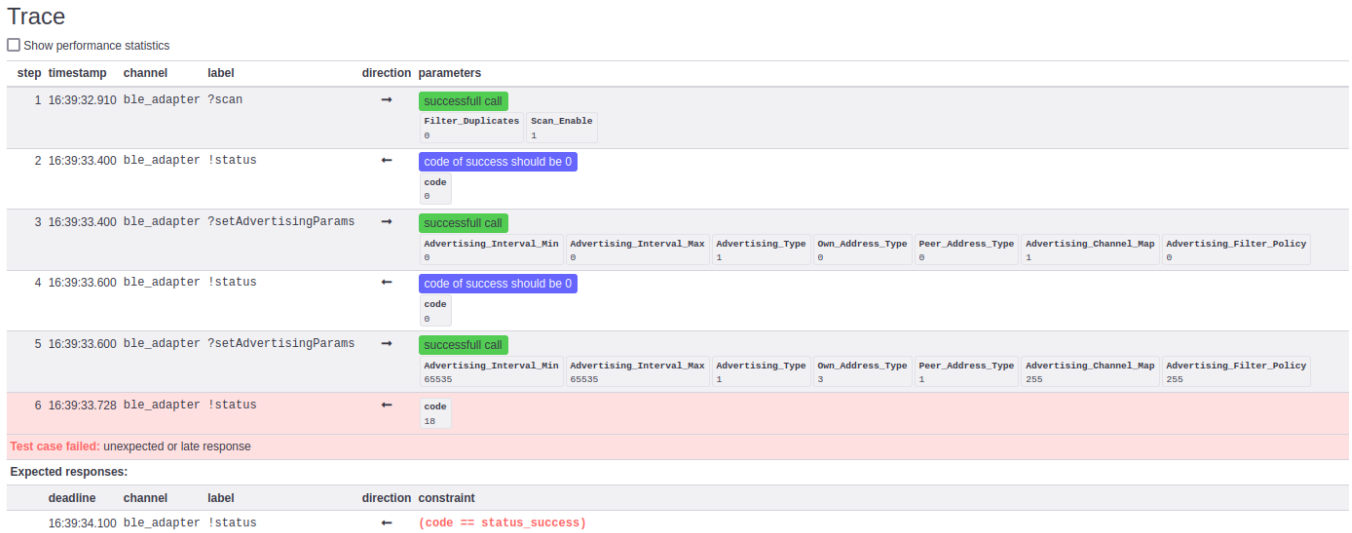

Figure 6.3: Overview of an automatically generated test case using the configurations as discussed for the *Conformance Experiment*

Before talking about the content, we should explain how Figure 6.3 should be interpreted. The figure represents a test case, in this scenario test case 2, where each test case consists of steps, transitions, that are represented as a row. Each row is marked with a label that represents an action on the SUT or a response of the SUT. The arrows (direction) represent this flow. An arrow pointing to the right, $\longrightarrow$,

indicates an action or command that is represented by the label that is performed on the SUT while a left arrow, ⟵, indicates the expected response from the SUT. Then the parameters column contains the data that is passed alongside the label, in addition to a note that is displayed in the colored boxes.

If we now investigate this test case more closely, we see that the test case was able to perform some successful transitions before failing. It was able to enable scanning with the first step after which it received the expected output. Then it tried to set the advertising parameters two times where only the first attempt was successful. On the second attempt, the SUT produced output with code 18 after executing the command while an output of 0 was expected. At this stage, a conformance error has been found and it would be up to the manufacturer to look further into the SUT code to determine why this occurs and provide the users with an appropriate solution.

6.3.2 Quantity Experiment

For the second experiment, we ran the testing environment with the configurations for the *Quantity Experiment* as discussed in Table 6.1. The results of this experiment are displayed in Figure 6.4.

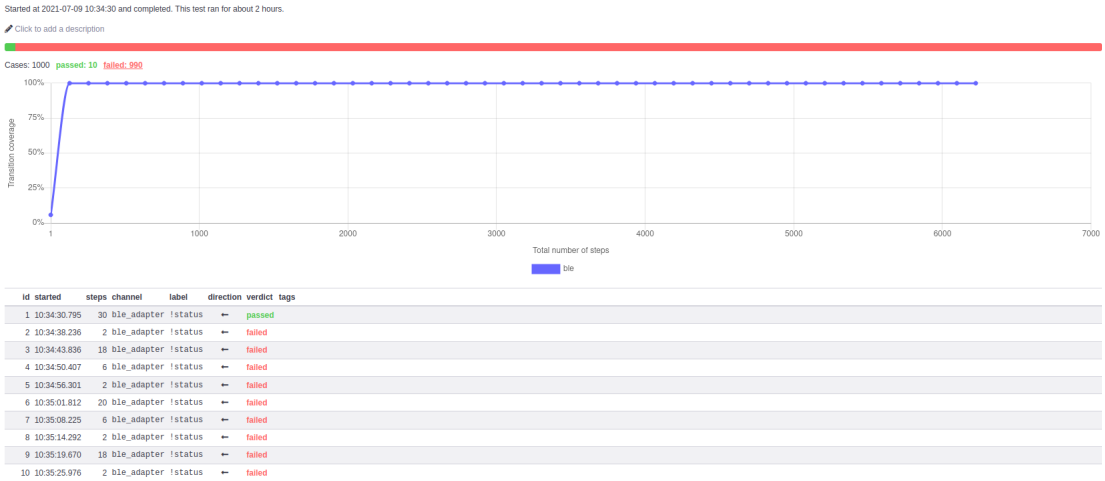

Figure 6.4: Partial overview of the resulting test cases and their evaluations using the configurations as discussed for the *Quantity Experiment*

Figure 6.4 shows us that using our testing environment we are able to generate and evaluate 1000 test cases on the SUT. From these test cases, 99% failed which indicates that the SUT does not conform to the defined behavior from the model. Additionally, this does not necessarily mean that the SUT contains this many implementation mistakes. It is possible that one mistake influences all of these failed test cases. However, we are now interested in the possibility to generate many test cases. For this reason, we will discuss this topic further in Section 7.2.

Furthermore, Figure 6.4 shows that the entire process takes about 2 hours. Because our experiment is not focused on performance we will leave more exact measurements to future work. Finally, our results show us that we are able to achieve a 100% transition coverage based on the generated test cases.

When we manually look at more detailed information on the test cases, we are able to see that the test generation makes use of random values as mentioned previously. An example that reflects this from our experiment can be seen by looking into the differences from Figure 6.6 and Figure 6.5.

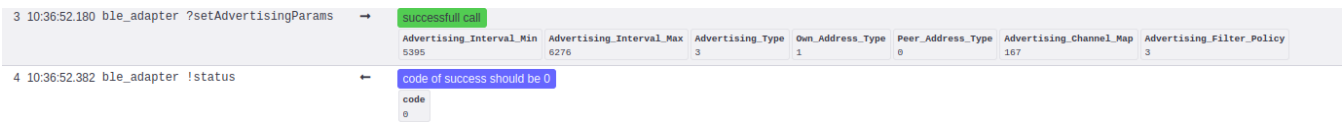

Figure 6.5: Successful call of the *HCILESetAdvertisingParameters* command using parameters from the input domain as defined within our BLE model (step 3) with the expected response (step 4) from test case 1

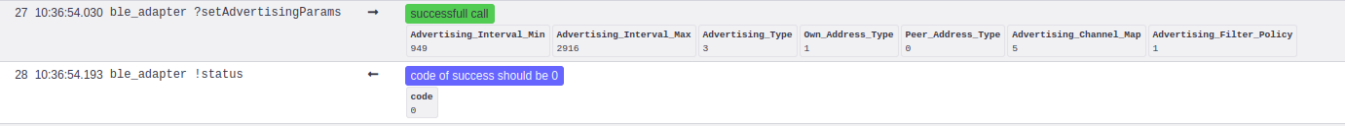

Figure 6.6: Successful call of the *HCILESetAdvertisingParameters* command using parameters from the input domain as defined within our BLE model (step 27) with the expected response (step 28) from test case 1

These two figures demonstrate that different parameter combinations can be generated for the same transition, in this case for the *setAdvertisingParams* transition.

6.3.3 Model Assumption Experiment

Applying the *Model Assumption Experiment* configurations from Table 6.1 we obtain the results as shown in Figure 6.7.

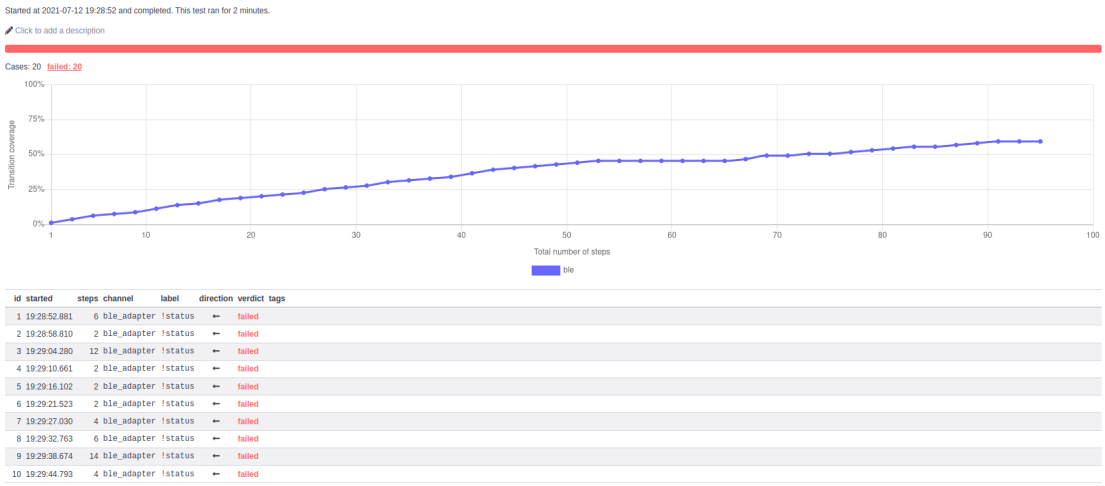

Figure 6.7: Partial overview of the resulting test cases and their evaluations using the configurations as discussed for the *Model Assumption Experiment*

These results show that using our current configurations, we are only able to achieve a Transition Coverage of 59.49%. This is mostly due to the fact that this time, we are not able to obtain a test case that is able to perform the maximum number of steps, 30, before finding an implementation mistake. In other words, during this experiment, we are not able to obtain a 100% Transition Coverage and therefore not able to test every transition within the Model. However, we still decide to continue and investigate if we could find implementation mistakes relating to our assumptions about the specifications. After manually going through the 20 test cases we find that some test cases fail due to some of our enforced assumptions. This categorization is displayed in Table 6.3. Furthermore, one example for each of these failed test case categories can be found in Section B.1.

Assumption Configuration	Number of Failed Test Case(s)
error_self_loop_paths	6
error_future_param_paths	13
error_validation_strength	0
scan_between_duplicates	1
force_link_layer_transitions	0

Table 6.3: Categorised implementation errors related to different underspecification assumptions using the configurations as discussed for the *Model Assumption Experiment*

Chapter 7

Discussion

In this chapter, we will discuss our findings and the results of our experiments from Chapter 5 and Chapter 6. Finally, we will discuss a meeting with the creator of the Bluetooth protocol in which we presented our work and asked for his opinion.

7.1 Implementing BLE Within AMP

During the development of the BLE model, we encountered several obstacles that needed to be overcome. We expect that when modeling similar communication protocols such as Zigbee [5], similar situations will be encountered. Our experience describes how we dealt with most of these challenges, such that our solutions can be used as a starting point to overcome similar situations.

Furthermore, our work distribution between the adapter and model falls in line with our expectations from Section 4.2.2. This expectation says that a model-based testing approach requires a large initial investment which is often caused by the development of the model. Because we experienced the same situation, we can confirm that for our use case, the BLE protocol, this expectation holds. Therefore, we can also expect that when a model of the BLE protocol would become available to the community, that a significant amount of time can be saved when setting up a model-based testing environment. This is because in this scenario the most time-consuming step would be removed from the model-based testing method. Our expectation is that this assumption will also hold for other IoT protocols.

While creating the model, we were also able to find several flaws in the specifications. This falls in line with our findings from Section 4.2.1. In this section, it is suggested that when a tester creates a model that the tester is more likely to find specification flaws. As such, we can conclude that this assumption also applies to modeling the BLE protocol. Furthermore, because it also applies to the BLE protocol, it becomes more likely to also apply to other IoT protocols.

Finally, due to time constraints, we were forced to scale our model to only a small part of the BLE protocol. Implementing a model that would cover the entire protocol would require more work and introduce more complexity. However, we believe that it is possible to achieve a complete model based on our current progress. Extrapolating this thought, we should also be capable to completely model other IoT protocols.

7.2 Testing BLE using AMP

After we created the necessary components to run a model-based testing environment to test the BLE protocol we proposed three experiments to evaluate the resulting platform and thus the model-based testing approach.

During our *Conformance Experiment*, we were able to show that our resulting testing environment is capable of finding specification errors. Furthermore, we demonstrated how a tester could obtain further knowledge to replicate the specification error such that they can start the debugging process. By showing that we are capable of finding specification errors, we demonstrate that the promised benefit of finding

specification errors also applies to testing the BLE protocol. This finding can then be extrapolated to also be an indicator that this assumption will hold for other IoT protocols.

Our second experiment, the *Quantity Experiment*, showed us that we are able to generate an arbitrary number of test cases up to 1000. By showing that this is possible, we indirectly demonstrate that we can automatically obtain a high Data Coverage. By demonstrating this using a model-based testing approach we show that we can overcome the IoT testing challenge of testing all the different protocol configurations from Subsection 4.1.3.

Furthermore, when we combine the results of these two experiments we argue that we can find all different conformance errors when we obtain a Data Coverage of 100%. This is because the first experiment demonstrated that we can find the conformance errors within our model while the second experiment suggests that we can obtain a 100% Data Coverage.

Within these two experiments, we also found a large percentage of failing test cases. We want to emphasize that this does not necessarily mean that there are this many bugs within the SUT. As mentioned earlier, a test case fails once it receives a response from the SUT that does not conform to the model. Because a test case takes multiple steps, it is possible that different test cases pass through the same transition which could in both cases result in an error. Furthermore, it could be possible that one bug influences different transitions resulting in more failed test cases. For this reason, it is recommended to look into the failed test cases individually and not as a group, as we did during our experiments. However, because our goal is to demonstrate that we are capable of achieving certain assumptions and not resolving all the problems within the SUT our approach is sufficient.

During our third experiment, *Model Assumption Experiment*, we were able to find implementation mistakes for most of our enforced assumptions. Finding these mistakes shows that manufacturers can have different interpretations on what the correct implementation should be regarding the underspecification about these assumptions. As a result, we demonstrated that the topics related to underspecification which we found during our modeling step do not only apply to us but also to manufacturers. Therefore, the assumption that we can find specification flaws during the modeling step also holds for modeling the BLE protocol. Extrapolating this finding we could argue that this assumption, therefore, is also more likely to hold for other IoT protocols.

Finally, all found conformance implementation mistakes are errors that were not found earlier when the SUT received its certificate for correctly implementing BLE. This could be seen as an indication that our proposed testing environment is a better environment than the testing environment that was used at that moment in time. However, this reasoning is limited due to the fact that the testing environment in question is more than 5 years old as of writing this research. It could be possible that the current testing environment is better or still worse, than our proposed testing environment but this remains to be seen.

7.2.1 Threats to Validity

The first thing we should discuss are the testing criteria that we used during our experiments. For our experiments we deemed two different coverage measurements to be important; Transition Coverage and Data Coverage. However, it could be possible that by using other coverage measurements and therefore different test generation configurations we could obtain a better or even more extensive foundation for the use of model-based testing to test IoT protocols. Looking into other coverage values falls outside the scope of our research and we will leave this to future work.

Another obstacle that we mentioned in Subsection 6.2.4 is the absence of a method to measure the Data Coverage within our experiments. We reasoned that this could be made up by increasing the number of test cases but this approach is still theoretical. For this reason, the conclusions drawn from this theoretical reasoning should not yet be seen as absolute facts and should be a priority to further investigate in future work.

Furthermore, we demonstrated that we could generate 1000 test cases which took about 2 hours. However, our argument is that we can automatically generate and evaluate test cases at no cost, but we do not investigate the costs of performing this. It could be possible that such a time span becomes unmanageable once a complete model is implemented and more tests need to be generated and ran. Nevertheless, we believe that this scenario is unlikely because running the testing environment comes at no cost except for the time that the SUT is used and the resulting electricity costs. Because IoT systems often have a

low power consumption [8] we believe that the electricity costs should not form an issue. In this case, only the elapsed time of using the SUT could form a problem. We recommend future research to further look into this aspect.

Another potential problem that we encountered during the evaluation of our results is that we are not able to directly see in which state the model is when it finds a conformance error. Having such information is important because it would make it easier for testers to replicate errors. For our small step size, this did not become a problem because the state could easily be determined by hand but this could become a problem when the step size becomes significantly larger. We found that this problem could easily be resolved by adding this information into the notes that are passed alongside each label within our testing environment. If future research decides to replicate our work we encourage them to take this into account and make it possible to easily track the current state of the SUT.

Finally, in our third experiment, we were able to find a subset of conformance errors related to our enforced assumptions. We want to clarify that this does not mean that the remaining assumptions are implemented correctly. The results from this experiment showed that we were not able to obtain a 100% Transition Coverage and we did not try to obtain a high Data Coverage. As a result, it is still possible that the other assumptions are implemented differently making them also potential relevant topics to look further into.

7.3 Meeting with the Creator of Bluetooth

During our research we had the opportunity to talk with the creator of Bluetooth, Dr. Ir. Jaap C. Haartsen. In this meeting, we were able to present our work and obtain his opinion about our research thus far.

In this meeting, he showed interest in our approach and highlighted the current problems in the IoT Bluetooth domain. He mentioned how current IoT manufacturers are struggling to achieve interoperability with other manufacturers that also implement systems that use Bluetooth. This is mostly caused by some manufacturers with a larger market share, slightly changing their implementation with respect to the official specifications.

In this context, our work would not directly be applicable because we focus on the implementation directly on the Bluetooth chip and not on higher abstract layers that utilize the functionality of the chip. However, by enforcing the lower levels to conform to the specifications we also influence how the system will function on higher levels. Therefore our suggested approach would already function to some extent as a tool to support interoperability. Furthermore, in this meeting, we discussed the possibility to create an abstract model describing how systems implementing Bluetooth would communicate with each other on these higher levels. Such a model could then be used to directly evaluate the interoperability between two systems using a model-based testing approach.

Overall, we were able to obtain an opinion about our work directly from the creator of the protocol that we were trying to test. In this meeting, it became clear that our approach was interesting enough to further look into and that it also has opportunities to be utilized in more than just our presented way.

Chapter 8

Conclusion

Currently, IoT manufacturers are having difficulties developing an extensive testing environment to completely evaluate their IoT protocol implementations. These protocols are fundamental for providing interoperability and security with other systems. For this reason, it is important to investigate potential solutions to overcome these difficulties and obtain a functional testing environment.

A potential solution that has been partially looked into is the model-based testing approach. However, current research has mainly been focusing on whether it is possible to utilize model-based testing and not necessarily on why we should use model-based testing. In this context, the goal of our research is to evaluate to what extent model-based testing can be used to overcome the challenges within the IoT protocol testing domain.

8.1 Findings

To determine whether model-based testing would be an appropriate pick we proposed the following research questions:

RQ 1) What problems could model-based testing resolve in the context of testing IoT protocols according to literature?

RQ 2) To what extent can a deployed model-based testing approach assist in solving the problems relating to the testing of IoT protocols in practice?

- **RQ 2.1)** How can we test an IoT protocol using a model-based testing approach?
- **RQ 2.2)** How does an applied model-based testing approach for testing IoT protocols compare to our findings in RQ1?

8.1.1 Research Question 1

To answer the first question we combined literature describing the different challenges of testing IoT protocols and the literature describing the potential benefits and disadvantages of model-based testing. Our research demonstrated that model-based testing possesses the traits to provide an extensive testing environment that accommodates for the low development cost challenge, if the models of the protocols are made publicly available. Furthermore, we found that model-based testing should be able to keep up with the quickly evolving protocol specifications that would normally make it difficult to keep a testing environment up to date. Finally, our findings suggest that model-based testing should also be able to cope with the large number of different test cases that are introduced by the different protocol configuration options.

Overall, the literature shows us that model-based testing should almost be able to resolve every challenge that we found in the IoT protocol testing domain. This is under the assumption that when models of IoT protocols are made publicly available that the time saved by removing the modeling step would result in a testing environment that is acceptable for IoT manufacturers. We also found that model-based testing is not a direct solution to the challenge of getting the required hardware in place which results in an

unwanted price overhead. However, this challenge can be resolved by using a different approach that is not related to the utilized testing method.

8.1.2 Research Question 2

Our findings from research question 1 showed us that model-based testing should almost be able to resolve every challenge in the IoT protocol testing domain. To verify if these findings are also applicable in practice, we implemented and experimented with a proof of concept that is capable of testing the BLE protocol using a model-based testing approach.

Research Question 2.1

In Chapter 5 we demonstrated how we can create a platform that tests an IoT protocol which in our case tests the BLE protocol. In this chapter we found potential obstacles and solutions of developing such a platform based on our experience. We then explained and demonstrated how we can use this testing environment to test the BLE protocol using three different experiments in Chapter 6. In the end, we are able to create a proof of concept that is capable of partially testing the BLE protocol using a model-based testing approach. Furthermore, we are able to see that some assumptions that are made about model-based testing are also applicable to our specific case. Based on our findings we expect that it should be possible to also utilize the model-based testing approach to test other IoT protocols but that obtaining these models will require a large amount of work.

Research Question 2.2

During the development of the model-based testing components that are required for the testing environment, we were capable of confirming some expectations that we made for RQ1. First, we were able to demonstrate that we are capable of finding specification flaws during the development of the model. This is one of the promised benefits of using a model-based testing approach. Furthermore, we were able to reconfirm that the modeling step is the most extensive step in the model-based testing approach when modeling the BLE protocol. This second finding is important because this is fundamental for potentially obtaining a low-cost testing environment when we would remove the modeling step by making the IoT protocol models publicly available.

Then, using the three different experiments from Chapter 6 we showed how some of the expectations from RQ1 are also true in practice when testing BLE. The first two experiments demonstrated that we are capable of obtaining an extensive testing environment that can automatically find conformance errors within a SUT. This finding can be directly related to our expectation that model-based testing can resolve the challenge of testing every possible protocol configuration option. Finally, by checking if the specification flaws also occur on the SUT and by finding examples of these errors, we demonstrated that our found specification flaws are relevant to further look into. As a result, this finding further supports the assumption that we are capable of finding relevant specification flaws in the modeling step of an IoT protocol.

8.1.3 Overall

Through our research, we were able to show what the current problems are within the IoT protocol testing domain and how model-based testing would fit within this domain according to the literature. Then, using this as a cornerstone we were able to create and discuss a proof of concept that is capable of partially testing the BLE protocol and utilizes a model-based testing approach. Additionally, by utilizing the resulting testing environment we were able to show that some of our found expectations from the literature are applicable when testing the BLE protocol using this proof of concept.

In the end, according to the literature we are likely to resolve almost any major challenge from the IoT protocol testing domain by using a model-based testing approach if the models for these protocols are made publicly available. Furthermore, in our research, we were able to show that some of the expectations from the literature are also true in practice when testing the BLE protocol. Because of this, it is more likely that these specific expectations will also be true for other IoT protocols. For this reason, we conclude that model-based testing is likely to be a successful approach to resolve the current challenges within the IoT protocol domain.

8.2 Limitations and Threats to Validity

First, one of the limitations of our research is that we are not able to confirm every assumption about model-based testing that can be applied to the IoT protocol testing domain. As a result, there are still challenges from the IoT protocol testing domain, that according to literature, should be resolvable using a model-based testing approach. However, these are not yet confirmed in practice. Further research can adopt a similar methodology to our research to confirm the remaining assumptions.

An example of such an assumption is that model-based testing would be able to cope with the high speed at which IoT protocols receive updates. This could for example be investigated by creating a model using the specifications from a deprecated version after which the model is adapted to conform to later versions. By looking into the additional work of applying the changes, we should be able to come to a conclusion related to this topic. We will leave such an approach to future work.

Secondly, an assumption that we often make during our research is that testing the BLE protocol would be similar to testing other IoT protocols. As a result, we are not able to tell for certain that our approach would work or have the same benefits and flaws when performed on different IoT protocols. Investigating this fell outside of the scope of our project and we will leave this to future work.

Thirdly, our fundamental assumption to make model-based testing viable for the IoT protocol testing domain is the cost savings by removing the modeling step. On that account, investigating this should obtain the highest priority for future work. This is because if we are not able to lower the development costs of the testing environment to at least the same price of what currently is being used in practice then it is less likely to be used. However, the remaining benefits of using model-based testing might still make it viable to utilize but this remains to be seen.

Finally, investigating the scaling of the testing environment forms a limitation within our work. Currently, due to time constraints, we decided to model a limited version of the BLE protocol. While running our testing environment with our model, we did not run into any performance problems. However, it could be possible that performance becomes an issue once the model becomes larger. For this reason, we suggest that once a complete model has been created of an IoT protocol that research starts looking into the performance aspects of the testing environment.

8.3 Future work

Within our work, we focused on discovering to what extent model-based testing would be a solution to the challenges in the IoT protocol testing domain. However, it would be interesting to compare our findings to other testing approaches to see if these other approaches would offer more or fewer benefits when compared to our findings. Performing this additional step would assist manufacturers to obtain a better idea of which testing approach to use.

Another aspect that we did not look into during our research is the possibility of directly testing interoperability when using model-based testing. Currently, our model is capable of modeling the protocol aspects up to the creation of a connection with a different system but does not test the connection itself. It would be interesting to see whether it is possible to model the behavior of two systems in one model that would create and establish a connection. If this is possible, it should also be possible to utilize model-based testing to automatically generate test cases to demonstrate that two different SUTs are capable of communicating with each other.

Finally, we did not perform an in-depth analysis about what kind of conformance errors we could find on the SUT using the model-based testing approach. It might be possible that by using a model-based testing approach a tester is capable of finding implementation flaws that directly jeopardize interoperability or the security of the SUT. Such an in-depth analysis could be looked into by using our proof of concept that utilizes the scoped model or by creating a new implementation using a model-based testing approach. Looking into this will be left to future research.

Acknowledgements

For some people, it just might have been some months working on a master project without giving it a second thought. However, this project was worked on during unusual times, namely during the peak of Corona. This scenario introduced, at least for me, many obstacles that made it more difficult to keep focus and more importantly keep the morale up to keep going forward. In these hard times, some people assisted me in overcoming these hardships such that I could bring this research to a good end.

In this context, I would like to first thank my supervisors; Machiel and Andy. They assisted me with navigating through the entire project and bringing structure to places where I got overwhelmed. Then, I would like to thank Tobias, another person that performed his master project at the same time as me at Axini. During both our projects you could say that he was my partner in crime. What I try to say with this, is that we were encountering similar challenges and that we were in a similar situation together. Having such a person to communicate with allowed me to sustain my sanity and sometimes even come up with solutions. For this, I am very grateful.

Of course, I would also like to thank Axini itself for allowing me to perform my master project at their organization. Furthermore, I want to thank the Axini team for giving me additional feedback, assistance, and for allowing me to feel like a part of the team. Finally, I want to express my gratitude towards my friends, specifically the Tuturu group (you guys know who you are), my parents, and my sister. Without your listening ears and mental support, I would not have been able to come as far as I have.

Bibliography

[1] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International journal of communication systems*, vol. 25, no. 9, p. 1101, 2012.

[2] S. Inc. (2020). "Internet of things (iot) active device connections installed base worldwide from 2015 to 2025*," [Online]. Available: `https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/` (visited on 12/14/2020).

[3] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of things (iot) communication protocols," in *2017 8th International conference on information technology (ICIT)*, IEEE, 2017, pp. 685–690.

[4] Bluetooth SIG, *Core specifcation 4.2*, `https://www.bluetooth.com/specifications/specs/core-specification-4-2/`, Accessed: 2021-06-28, 2014.

[5] S. C. Ergen, "Zigbee/ieee 802.15. 4 summary," *UC Berkeley, September*, vol. 10, no. 17, p. 11, 2004.

[6] C. Vorakulpipat, E. Rattanalerdnusorn, P. Thaenkaew, and H. D. Hai, "Recent challenges, trends, and concerns related to iot security: An evolutionary study," in *2018 20th International Conference on Advanced Communication Technology (ICACT)*, IEEE, 2018, pp. 405–410.

[7] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 195–212.

[8] M. Bures, T. Cerny, and B. S. Ahmed, "Internet of things: Current challenges in the quality assurance and testing methods," in *International Conference on Information Science and Applications*, Springer, 2018, pp. 625–634.

[9] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st international conference on Software engineering*, 1999, pp. 285–294.

[10] A. Pretschner, "Model-based testing in practice," in *International Symposium on Formal Methods*, Springer, 2005, pp. 537–541.

[11] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing iot communication via active automata learning," in *2017 IEEE International conference on software testing, verifcation and validation (ICST)*, IEEE, 2017, pp. 276–287.

[12] A. Ahmad, F. Bouquet, E. Fourneret, F. Le Gall, and B. Legeard, "Model-based testing as a service for iot platforms," in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2016, pp. 727–742.

[13] K. Inçki and I. Ari, "Observing interoperability of iot systems through model-based testing," in *Interoperability, Safety and Security in IoT*, Springer, 2017, pp. 60–66.

[14] A. Elnashar, "Iot evolution towards a super-connected world," *arXiv preprint arXiv:1907.02589*, 2019.

[15] J. Saleem, M. Hammoudeh, U. Raza, B. Adebisi, and R. Ande, "Iot standardisation: Challenges, perspectives and solution," in *Proceedings of the 2nd international conference on future networks and distributed systems*, 2018, pp. 1–9.

[16] J. Hwang, A. Aziz, N. Sung, A. Ahmad, F. Le Gall, and J. Song, "Autocon-iot: Automated and scalable online conformance testing for iot applications," *IEEE Access*, vol. 8, pp. 43 111–43 121, 2020.

[17] D. Rayner, "Osi conformance testing," *Computer Networks and ISDN Systems*, vol. 14, no. 1, pp. 79–98, 1987.

[18] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Formal Methods in System Design*, vol. 34, no. 3, pp. 238–304, 2009.

[19] ——, "Black-box conformance testing for real-time systems," in *International SPIN Workshop on Model Checking of Software*, Springer, 2004, pp. 109–126.

[20] H. Kim, A. Ahmad, J. Hwang, H. Baqa, F. Le Gall, M. A. R. Ortega, and J. Song, "Iot-taas: Towards a prospective iot testing framework," *IEEE Access*, vol. 6, pp. 15 480–15 493, 2018.

[21] S. Ziegler, S. Fdida, C. Viho, and T. Watteyne, "F-interop–online platform of interoperability and performance tests for the internet of things," in *Interoperability, Safety and Security in IoT*, Springer, 2016, pp. 49–55.

[22] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 85–103.

[23] T. B. Glenford J. Myers Corey Sandler, *The art of software testing*. Wiley Online Library, 2011, vol. 3.

[24] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer, "Gast: Generic automated software testing," in *Symposium on Implementation and Application of Functional Languages*, Springer, 2002, pp. 84–100.

[25] S. Janssen, "Transforming source code into symbolic transition systems for practical model-based testing," 2017.

[26] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[27] L. Frantzen, J. Tretmans, and T. A. Willemse, "A symbolic framework for model-based testing," in *Formal approaches to software testing and runtime verification*, Springer, 2006, pp. 40–54.

[28] W. Bohlken, M. van der Bijl, and A.-M. Oprescu, "Model-based fuzzing using symbolic transition systems.," in *SATToSE*, 2020.

[29] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, Springer, 2008, pp. 1–38.

[30] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting, and E. Torreborre, "Model-based testing from uml models," *INFORMATIK 2006{Informatik für Menschen{Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik eV (GI)*, 2006.

[31] L. Frantzen, J. Tretmans, and T. A. Willemse, "Test generation based on symbolic specifications," in *International Workshop on Formal Approaches to Software Testing*, Springer, 2004, pp. 1–15.

[32] I. Schieferdecker, "Model-based testing," *IEEE software*, vol. 29, no. 1, p. 14, 2012.

[33] J. Yoneyama, C. Artho, Y. Tanabe, and M. Hagiya, "Model-based network fault injection for iot protocols," in *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, SCITEPRESS-Science and Technology Publications, Lda, 2019, pp. 201–209.

[34] B. Aziz, "A formal model and analysis of an iot protocol," *Ad Hoc Networks*, vol. 36, pp. 49–57, 2016.

[35] B. H. Malik, M. Khalid, M. Maryam, M. Nauman, S. Yousaf, M. Mehmood, and H. Saleem, "Iot testing-as-a-service: A new dimension of automation," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 5, 2019.

[36] E. J. Marinissen, Y. Zorian, M. Konijnenburg, C.-T. Huang, P.-H. Hsieh, P. Cockburn, J. Delvaux, V. Rožić, B. Yang, D. Singelée, *et al.*, "Iot: Source of test challenges," in *2016 21th IEEE European Test Symposium (ETS)*, IEEE, 2016, pp. 1–10.

[37] A. Taivalsaari and T. Mikkonen, "A roadmap to the programmable world: Software challenges in the iot era," *IEEE software*, vol. 34, no. 1, pp. 72–80, 2017.

[38] P. Rosenkranz, M. Wählisch, E. Baccelli, and L. Ortmann, "A distributed test system architecture for open-source iot software," in *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems*, 2015, pp. 43–48.

[39] L. Apfelbaum and J. Doyle, "Model based testing," in *Software quality week conference*, 1997, pp. 296–300.

[40] R. V. Binder, B. Legeard, and A. Kramer, "Model-based testing: Where does it stand?" *Communications of the ACM*, vol. 58, no. 2, pp. 52–56, 2015.

[41] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One evaluation of model-based testing and its automation," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 392–401.

[42] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, 2007, pp. 31–36.

[43] Bluetooth SIG, *Core specifcation 5.2*, `https://www.bluetooth.com/specifications/specs/core-specification-5-2/`, Accessed: 2021-06-28, 2019.

[44] A. S. Huang and L. Rudolph, *Bluetooth essentials for programmers*. Cambridge University Press, 2007.

Appendix A

Specification Flaws

In this chapter, we will elaborate on some specification flaws that we found during our research. The different specification flaws that we discuss in this chapter are all related to the Bluetooth Core Specification version 4.2 [4].

A.1 UnderSpecification

A.1.1 Link Layer Protocol

From the specifications, we are aware that the HCI commands allow you to execute functions on the lower layers such as the Link Layer. Additionally, the specifications about each of these layers describe restrictions that should be met. Therefore we expect that the HCI commands that interact with these lower layers also follow these restrictions. However, the HCI command descriptions do not directly relate to these lower layers and sometimes no restrictions are defined that we would expect based on the lower layer descriptions. As a result, we encounter underspecification that forces us to make assumptions about the intended behavior of the SUT.

Valid Transitions

An example of such underspecification is related to valid transitions. Based on the state diagram from Figure 5.2 that represents the Link Layer we expect that certain HCI commands are only allowed in certain states. For example, the command to enable advertising should only be callable from the standby state. Yet, this is not mentioned in the HCI command descriptions.

If we would follow the restrictions of the Link Layer protocol we would assume that an error should be thrown when such a command is not allowed. The most logical error to us would be the *COMMAND DISALLOWED (0x0C)* error. This error indicates that a command should not be executed because it finds itself in a state where this command is not allowed. As such this assumption is applied to our model.

Self-Loops

The Link Layer state machine from Figure 5.2 also does not reflect transitions to itself, self-loops. This could indicate that commands that could change the state of the SUT should not be allowed to bring the SUT towards the same state. An example would be to disable scanning when it is already disabled.

Our initial thought would be to throw the *COMMAND DISALLOWED (0x0C)* error. However, these commands would still be valid when different parameters would be used so this error code can not be used. Another idea would be to throw the *INVALID HCI COMMAND PARAMETERS (0x12)* error but this error is also not applicable when looking at the use cases described in the specifications. As a result, there is no specific error that describes this situation. Consequently, we are forced to assume that the *UNSPECIFIED ERROR (0X1F)* error is thrown.

A.1.2 Error Precedence

While investigating the HCI command error chapter from the specifications we found that there is nothing written about error precedence when multiple errors could occur. The specifications hint that this would be implementation dependent but we argue that this should still be defined within the specifications or at least to some extent.

From our perspective, the *COMMAND DISALLOWED (0x0C)* error would have the highest precedence. This is because when a command is not allowed the SUT should not evaluate the parameters of the command call. An example would be when a command is disallowed and attackers try to pass on invalid parameters to abuse an implementation mistake. In this scenario, we should only evaluate the command call itself before evaluating and processing the passed parameters. As a result, defining error precedence might influence the security of such protocols and should therefore be discussed in the specifications.

A.1.3 Reserved for Future Usage Parameters

Generally, the specifications give a description of the HCI commands overall as well as their individual parameters. For each of these parameters, they define the valid input domain and in some places, they define a range of values that are reserved for future usage.

The specifications do not directly state how the system should behave when faced with a parameter that is reserved for future usage. We assume that since these parameters are defined for the input domain that when such a parameter is given it should still be considered as a valid parameter. Therefore the *INVALID HCI COMMAND PARAMETERS (0x12)* error is not applicable. When looking through the error list we found the *UNSUPPORTED FEATURE OR PARAMETER VALUE (0x11)* error. This error seems the most applicable to our use case because these parameters are not supported yet.

A.1.4 *Ignored* Key Word

While reading through the HCI command chapter from the specifications we came across the key word *ignored*. This key word was used to describe that certain parameters should be *ignored* when the SUT finds itself in a specific situation. However, in the specifications of BLE version 4.2 [4] it is not elaborated what this exactly means. Therefore we need to make an assumption on what the intended behavior of *ignoring* a parameter is. When we looked at the latest specification version, 5.2 [43], we found that an additional paragraph was added with the sole purpose of elaborating this key word. As such we decided to follow this description and apply it within our model.

A.2 Inconsistent Specification

A.2.1 Contradictory Specifications

The specification about the *HCI_LE_Set_Advertising_Parameters* command contains a contradiction. In the description of the command, they mention that the configuration values of *Advertising_Interval_Min* and *Advertising_Interval_Max* are allowed to be the same value but that for the best usage they should not be set to the same value. However, when reading the individual parameter descriptions of this command they mention that the default values of these configurations are the same. This is a contradiction because they discuss that they want the users to use different configuration values for the most optimal behavior but at the same time, they set the configuration values to the same by default. This contradiction is also still described in the latest, version 5.2 [43] specifications. We recommend that later specifications will change this default value to encourage the best possible usage.

A.2.2 Set Scanning Two Resulting State Versions

With the *HCI_LE_Set_Scan_Enable* command a developer is able to disable or enable scanning. However, this command takes two parameters. One of these is the *Filter_Duplicates* parameter that influences how the output of the discovered devices is represented to the user. You could argue that this is a configuration and therefore should be taken into account in the *HCI_LE_Set_Scan_Parameters* command. Because this is not the case, we can also reason that there are two possible scanning states using the *HCI_LE_Set_Scan_Enable* command. As a result, we need information about when we are allowed to use this command to switch this specific configuration. However, this information is not available within the

specifications and therefore we again need to make assumptions on what the intended behavior is. We assume that the system is able to change this configuration while scanning because it is related to how the data is displayed and not to how the SUT performs the scanning action itself.

Appendix B

Conformance Errors

B.1 Assumption Related Errors

In Subsection 6.3.3 we discuss how we utilized a SUT to evaluate how it would behave when we enforce our assumptions about the specifications. In our experiment, we encountered test cases that would fail for some of these assumptions. We will give one example in the form of a test case for each of these failed assumption categories.

B.1.1 Self Loops

When we enforce the assumption that we discussed in Section A.1.1 about self-loops onto the SUT we encounter that the SUT behaves differently. A test case showing this is demonstrated in Figure B.1. This test case shows that when we try to disable scanning while it is already disabled that the SUT will throw an error with value 12 while we expect an error of code 31.

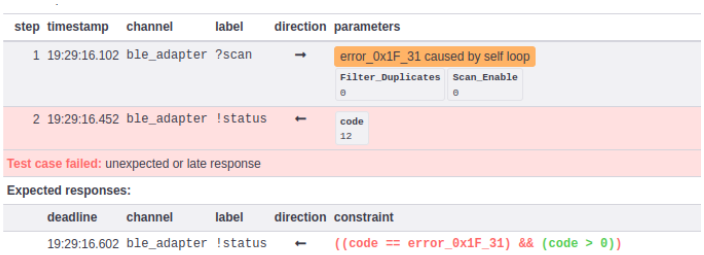

Figure B.1: Test case that shows that the SUT does not conform to our expected behavior related to our *self loops* assumption

B.1.2 Reserved for Future Usage Parameters

When we enforce the assumption that we discussed in Subsection A.1.3 about the usage of reserved for future parameter values onto the SUT we encounter that the SUT behaves differently. A test case showing this is demonstrated in Figure B.2. This test case shows that the SUT is first successful in enabling and disabling advertising but that it fails during the third attempt. In the third attempt, the SUT executes the command with a value reserved for future usage and therefore expects an error code of 17 but instead receives the error code 18.

Figure B.2: Test case that shows that the SUT does not conform to our expected behavior related to our *reserved for future usage parameter* assumption

B.1.3 Set Scanning Two Resulting State Versions

When we enforce the assumption that we discussed in Subsection A.2.2 about the transition between the two possible scanning states onto the SUT we encounter that the SUT behaves differently. A test case showing this is demonstrated in Figure B.3. This test case shows that it is able to successfully enable scanning with the *Filter_Duplicates* configuration set to 0 (false). The model expects that the SUT is able to change this configuration by calling the command again using a different value. However, when this is done on the SUT, step 3, the SUT responds with an error code 12 while we expected that it would be successful (code 0).

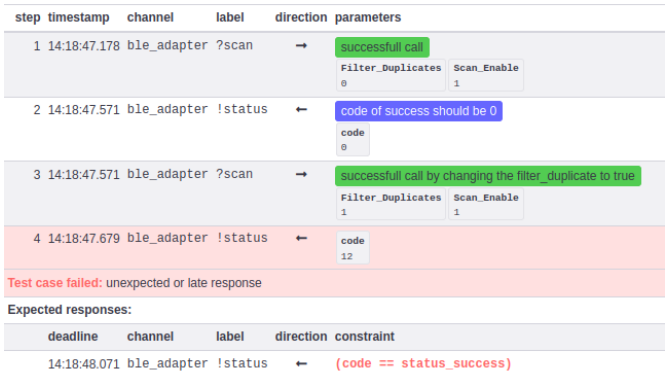

Figure B.3: Test case that shows that the SUT does not conform to our expected behavior related to our reserved for future usage parameter assumption