Institute of Software Technology
Software Engineering

University of Stuttgart
Universitätsstraße 38
70569 Stuttgart

Master Thesis

Property-based Testing: Evaluating its Applicability and Effectiveness for AUTOSAR Basic Software

Aparna Bose

Course of Study: INFOTECH

Examiner: Prof. Dr. Stefan Wagner

Supervisor: M.Sc. Jonas Fritzsch

Dipl.-Inf. Hans-Jörg Reichle

Commenced: August 14, 2019

Completed: February 21, 2020

Abstract

Previous work has shown that Property-based Testing (PBT) can be successfully applied to testing synchronous software. For example, it has been demonstrated that PBT can be applied to testing cloud services, web services and telecoms software. But less research has been carried out to evaluate this approach to testing asynchronous code as in automotive software. In the work presented in this Master thesis, the data generation feature of PBT is exploited to test the functionality of a software module based on the AUTOSAR Adaptive Platform.

Properties are defined considering the system as a black-box targeting its functionality on an abstract level. First, we apply stateless properties to test a single functionality and thereby find the communication delay needed to incorporate in our testing at system level. Later, we implement a test infrastructure based on stateful properties using the Python tool Hypothesis for the demonstration of research based on PBT. The testing framework is interfaced with the runtime environment to integrate the former with the system being tested. The test inputs generated in this approach are evaluated for their effectiveness and efficiency in testing the software module under test. Finally, experts in the testing field have been interviewed to draw comparisons between PBT and traditional methods of testing.

Acknowledgement

It gives me great pleasure in acknowledging the contributions of all those without whom this thesis would not have been possible. First and foremost, I wish to thank Prof. Dr. Stefan Wagner for giving me an opportunity to do my thesis in the Department of Software Technology.

This work would not have been straightforward without the guidance of my supervisor Mr. Jonas Fritzsch who has shown me, by his example what a good researcher should be like. He has patiently corrected my writing and supported my research.

I would like to thank my supervisor at Vector Mr. Hans-Jörg Reichle, who took the time out to hear my ideas and guided me on the right path throughout the thesis with his knowledge and experience. I would also like to thank my manager at Vector Mr. Philipp Kallenberg for sharing his valuable suggestions and remarks which helped me progress in my work.

I am immensely grateful to Mr. Klaus Bergdolt, without whom this thesis would not have been possible at the right time in my life.

I would like to thank my fiancé Dr. Deepak Bos und my family for their immense support and encouragement throughout my thesis. Finally, my sincere thanks goes to my best friends Meenu Saratchandran, Aniket Bhovad and Akshay Kulkarni who stood by me during the low tides of my studies.

Contents

1 Introduction **15**
1.1 Motivation 16
1.2 Scope of Work 16
1.3 Research Questions 17
1.4 Research Methodology 17
1.5 Thesis Outline 18

2 Fundamentals **19**
2.1 Testing 19
2.2 Test Levels in the V-Model 21
2.3 Property-based Testing (PBT) 22
2.3.1 Hypothesis Framework 23
2.3.2 Stateless Properties 23
2.3.3 Stateful Properties 24
2.3.4 Application of stateless and stateful properties 25

3 System Under Test **27**
3.1 AUTOSAR 27
3.1.1 AUTOSAR Classic 27
3.1.2 AUTOSAR Adaptive 27
3.2 Service-Oriented Architecture 28
3.3 Architectural Overview of Adaptive Platform 29
3.4 Communication Management (ara::com) 32
3.5 Service Discovery 34
3.5.1 Application View 35
3.5.2 Bus View 36

4 Specifications and Test Design **37**
4.1 AUTOSAR Specifications for ara::com 37
4.2 Comparison of Test Strategies 37
4.2.1 Example-based Testing 38
4.2.2 PBT 38
4.3 Deriving Properties from Specifications 39

5 Implementation **41**
5.1 Block Diagram of SUT 42
5.2 Integration of Hypothesis with the Adaptive Platform 43

5.3 Stateless Property 44
5.4 Service Test Model 45
5.5 State diagram of a Service 46
5.6 Stateful Property 47

6 Results and Evaluation **51**
6.1 Effectiveness 51
6.1.1 Issue found during testing 51
6.1.2 Concurrency 52
6.1.3 Statistics of Test Inputs 53
6.1.4 Analysis of Test inputs 54
6.1.5 Fault Injection 55
6.1.6 Coverage 55
6.2 Efficiency 57
6.3 Interview Study 58

7 Threats to Validity **61**

8 Related Work **63**

9 Conclusion **65**

10 Future Scope **67**

Bibliography **69**

List of Figures

2.1 Manual and Automated testing [17] 20
2.2 Optimal Software Testing [19] 21
2.3 Test levels in the V-model [21] 22
2.4 Regular unit tests for sum function 24
2.5 Stateless property 24
2.6 Example of a test run in Hypothesis 25

3.1 Service-Oriented Communication [43] 29
3.2 Adaptive Platform Architecture [9] 30
3.3 SOC via Communication Management [49] 32
3.4 Communication on Adaptive Platform 33
3.5 SOME/IP Protocol [54] 34
3.6 Local and SOME/IP Service Discovery [51] 35

4.1 Example-based testing 38

5.1 Block Diagram of the SUT 41
5.2 Setup of the PBT test system for SD 42
5.3 Renesas R-Car ECU running Adaptive Applications [61] 43
5.4 Stateless property to find the delay needed after an asynchronous API call 45
5.5 Test model representing a service. 46
5.6 Implemented model of service 47
5.7 Stateful property for testing Service Discovery 48
5.8 An example test generated by Hypothesis for SD 50

6.1 Minimal example for the port mismatch error detected 51
6.2 Concurrency of service statemachine models 52
6.3 Number of test inputs generated by Hypothesis in 100 examples with step count=50 53
6.4 An example generated by Hypothesis with an interface call without a *Check()* 54
6.5 Minimal example for an injected fault into *StartOfferService()*. 55
6.6 Transition Coverage of Service 1 in a test run of 100 examples 56
6.7 Transition Coverage of Service 2 in a test run of 100 examples 56
6.8 Transition Coverage of Service 3 in a test run of 100 examples 57

List of Tables

4.1 Test Specification for local and remote Service Discovery [11] 37

5.1 Boundary Values of time obtained with stateless property. 44

6.1 Combinations of states of services from a test run of 100 examples. 53
6.2 Statistics of test cases covered for all services in a run of 100 examples. . . 54
6.3 Execution and wait times in the test runs of SD. 58
6.4 Participant data. 58

List of Acronyms

AA Adaptive Application

AP Adaptive Platform

API Application Programming Interface

ARA AUTOSAR Runtime for Adaptive Applications

AUTOSAR Automotive Open System Architecture

BSW Basic Software

C Checked

CM Communication Management

ECU Electronic Control Unit

FSM Finite State Machine

ID Identifier

MLoC Million Lines of Code

NC Not Checked

NO Not Offered

O Offered

OEM Original Equipment Manufacturer

OTA Over-the-air

PBT Property-based Testing

SD Service Discovery

SOA Service-Oriented Architecture

SOC Service-Oriented Communication

SOME/IP Scalable service-Oriented MiddlewarE over IP

SUT System Under Test

1 Introduction

Ever since the first software has been introduced 30 years ago in automotive electronic modules [1], the automotive industry has evolved into software-intensive applications from mechanically intensive ones [2, 3]. A modern car has 80 to 120 embedded microcontrollers with software running upon and these units are referred to as Electronic Control Units (ECUs). It's only a matter of time when an automobile can be rightly described as 'a computer on wheels' [4].

More than 80% of the innovations in an automobile is contributed by software, making it an integral part of the automobile industry [1]. The remark of the famous automotive entrepreneur Robert Bosch [4], "Without exception, our aim must be to improve the current status; and instead of being satisfied with what has been achieved, we must always strive to do our job even better", has already been proven true with the industry growing into an era of new-generation cars. With the vast expansion in the functionality of automobiles, the complexity of software has also increased in the order of 100 Million Lines of Code (MLoC). This number is huge as compared to that of the social media giant "Facebook" which is only around 60 MLoC [2, 4, 5]. According to Haghighatkhah et al., one could argue that modern cars run on code as much as on fuel [3].

The growing complexity of software in automobiles has led to the formation of AUTOSAR (Automotive Open System Architecture) Consortium by a group of OEMs (Original Equipment Manufacturers) and Tier 1 companies in 2003 [6]. The software used in Electronic Control Units (ECUs) is thereby standardized to tackle the complexity of its architecture. This has resulted in simplified development processes, modularity and reuse of software components as well as the integration of ECUs from different suppliers into a single OEM product [7]. Deeply embedded ECUs which do not need dynamic linking of services are based on the AUTOSAR Classic Platform. The new standard released by AUTOSAR in 2017 is referred to as the Adaptive Platform. The aim of the Adaptive AUTOSAR is to develop ECUs with high computing power and secure communication-based applications such as driver assistance, autonomous driving, communication with cloud services, software updates Over-the-air (OTA) etc. [8]. Neither of these platforms would replace one another but instead would interact with each other and other external systems to form an integrated system [9].

1.1 Motivation

The specifications released by AUTOSAR for the Basic Software (BSW) modules are interpreted by the suppliers to develop software components in the ECUs. The suppliers test the developed ECUs to check if their software implementation satisfies their interpretation of AUTOSAR specification [7]. The software components will be delivered by the suppliers to the Tier 1 companies who would then develop applications on top of the basic software modules. At each level of software component development and integration, these are thoroughly verified and validated to ensure that the implementation meets functional expectations. Testing is, therefore, an integral part of assessing the quality of the software before it is delivered to the customer [10].

For testing at the system level, test specifications are released by AUTOSAR in the document “System Tests of Adaptive Platform” [11]. These specifications aim mainly at validating the requirement items mentioned in the requirement documents of each software module [12]. The test specifications cannot deliver intensive testing of each software module, rather can aid in extensive testing of all the modules of the Adaptive Platform.

Property-based Testing (PBT) is a new randomized approach of testing which became popular after a tool named QuickCheck [10]. It involves representing the functional specification in the form of properties. These properties are then tested with a large number of automatically generated test cases. At the end of testing, the test inputs for which the properties do not hold are identified [13]. The failing test inputs are then simplified by the property-based testing framework through a shrinking process. Not only do these shrunk test sequences help debug the defective code faster, but also are these saved to ensure that the problem has been successfully fixed during consecutive tests [7, 14].

PBT has been widely used in testing synchronous functions [P6]. However, PBT has not been evaluated in the aspect of testing asynchronous software like automotive software. Asynchronous software differs from synchronous software in that the former takes time to output the response to the requests without blocking the caller and this factor attributes to challenges in its testing [15].

This thesis aims to evaluate the effectiveness and efficiency of the property-based approach for testing the Basic Software modules in ECUs. As a proof of concept, conventional and property-based test approaches are compared for the “Communication Management” BSW module of AUTOSAR. Communication Management is a significant functional cluster of BSW which is responsible for the communication between different applications of Adaptive Platform [16]. For this purpose, a test environment based on property-based approach is designed and implemented.

1.2 Scope of Work

The main contribution of the thesis is the design of a test infrastructure based on property-based approach and investigation whether this approach is feasible for verifying the func-

tionality of AUTOSAR Basic Software. For the study, one of the functional clusters of ECU Basic Software based on Adaptive Platform, namely, Communication Management (CM) is chosen. The property-based approach is evaluated for testing the software module at a higher level, also referred to as system testing, where CM Modules of two ECUs are involved.

1.3 Research Questions

This master-thesis addresses following key questions and tries to answer them:

RQ1: How can properties be derived from system specifications and requirements of the AUTOSAR Basic Software?

RQ2: When can a module be regarded as "sufficiently" tested using property-based testing? What are appropriate variables, metrics and values in this context?

RQ3: How effective is property-based testing for AUTOSAR software? When can it extend or replace conventional testing methods and when is it inappropriate?

1.4 Research Methodology

The Thesis is carried out in the following steps to answer the corresponding research questions:

How can properties be derived from system specifications and requirements of AUTOSAR Basic Software? (RQ1)

1. Extensive research on literature is done to reveal projects wherein property-based testing has already been used. This step gives a better understanding of the property-based testing approach in general. Additionally, how properties have been derived in these existing projects is analysed. Further, this serves as a guide to think in terms of properties during the next step.
2. Examination of specification, requirement and other technical documents released by AUTOSAR for the module "Communication Management" of Adaptive Platform are done. This step is necessary to get familiar with the functionality of the module to identify relevant properties for testing.

When can a module be regarded as "sufficiently" tested using property-based testing? What are appropriate variables, metrics and values in this context? (RQ2)

3. Design and implementation of a property-based test infrastructure for the module is performed for the qualitative analysis of the identified properties.

4. Evaluation and visualisation of the test results along with in-depth research on the existing adequacy criteria [5] for random and black-box testing aid in formulating a test adequacy criterion for property-based test approach.

How effective is property-based testing for AUTOSAR software? When can it extend or replace conventional testing methods and when is it inappropriate? (RQ3)

5. Assessment of response of the System under Test (SUT) to the test inputs helps to figure out the "effectiveness" aspect of this testing approach. For outlining the comparison to conventional testing, experts in the testing field are interviewed. This step helps further to identify the metrics and variables of RQ2 that could be taken into consideration. This step is also critical to understand the areas where PBT test approach could leave a significant impact on testing along with those where there are shortcomings.

1.5 Thesis Outline

This master thesis is structured as follows:

Chapter 1 - Introduction: The motivation of the thesis is explained with a basic introduction to property-based testing and AUTOSAR. The research questions are formulated, and the methodology of research has been outlined.

Chapter 2 - Fundamentals: The essential concepts and technologies that are necessary to understand the thesis are discussed.

Chapter 3 - System Under Test: The system which is tested in this thesis is described.

Chapter 4 - Specifications and Test Design: The existing test method is explained with the available test specification. The design approach based on our approach is also specified.

Chapter 5 - Implementation: The implementation of the design is discussed in this chapter.

Chapter 6 - Results and Evaluation: The results of the test approach are explained along with the interview study conducted during this work.

Chapter 7 - Threats to Validity: The potential threats to the findings and results of this thesis are presented.

Chapter 8 - Related Work: The various research work related to property-based testing and testing of AUTOSAR software modules has been identified.

Chapter 9 - Conclusion: A summary of the work with respect to the research questions is presented in this chapter.

Chapter 10 - Future Scope: An outline for future work is discussed in this chapter.

2 Fundamentals

In this chapter, we discuss the various terms and technologies necessary to understand the concepts introduced in this thesis, which are testing, the underlying V-model.

2.1 Testing

Any software, or rather any product must deliver what it promises. For embedded software like automotive software, factors like timing and safety impose strict constraints in addition to the correctness of the software. The inevitable task of ensuring these requirements is through testing the software or the product thoroughly during its development or after it has been implemented. Testing is the process of verifying whether the functionality of a product or software conforms to its specifications. Testing software accounts for 50% of its development [17]. There are three major steps in testing which are the following [17]:

1. **Preparation of test specification** - This step requires good knowledge of the system being tested like the functionalities to be tested, expected reaction or outputs etc. as it involves the design of the test process.
2. **Implementation of test cases** - The test cases are derived from test specifications and comprise of test sequences which are applied on the system being tested.
3. **Analysis of test results** - This step involves test report generation and is essential to determine whether the test outputs correspond to those listed in the test specification.

Depending on how the testing is performed, it can be manual and automatic. Manual testing involves writing test cases and executing them one by one without using test automation tools [17, 18]. On the other hand, automated testing includes an external tool or software for test execution and test report generation [19], and is ideal when repetitions of tests are needed for a system [17]. According to Meyer [20], manual tests and automatic tests are good at depth and breadth respectively. Additionally, an effective testing process comprises of both manual as well as automatic test cases [20].

According to Figure 2.1, the initial time required for creating and launching the automatic tests is more in comparison to manual tests. The number of manual test runs increases gradually with respect to time. If the software has to be tested only once, then it is better to go for manual testing as it is not worth spending the initial efforts of setting up the automated environment for a single test. But if the software development involves fast

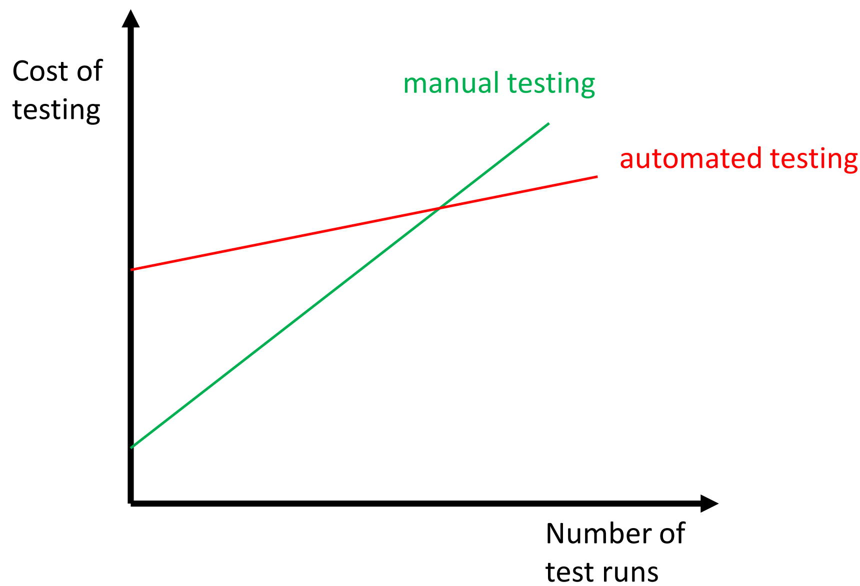


Figure 2.1: Manual and Automated testing [17]

changing versions each of which has to be tested separately, then automated testing is a much better choice than manual testing [21].

Figure 2.2 illustrates the significance of depth of testing with respect to the software quality. Severe reduction of tests can cause to miss many bugs and testing more than required can lead to increased testing costs as well. It is beneficial to run an optimal amount of tests which lead to effective testing with favourable costs [19].

There are different approaches to testing software. Some of these which are relevant for this thesis are explained below:

Black box Testing - In this type of testing, knowledge of the implementation details of the system being tested is not required. Test cases are created with reference to the functionality derived from an abstract model of the system or its specification. Test inputs are provided to the system and the outputs of testing are compared with those corresponding to the system specification [17, 14].

White box Testing - The internal structure of the system is taken into consideration for this type of testing. Test cases are produced in such a way so that all the statements, functions or branches of the source code are executed [17]. In simpler terms, this means that all the desired internal system states are covered atleast once [14].

Gray box Testing - This type of testing is the combination of black box and white box tests [17]. Here, the system functionality as well as the logical path of the code is considered during testing.

Model based Testing - A model is the functional behaviour of the system in an abstract form [14]. Here, test cases are generated from the model and tested on the system [22]. Modelling languages like UML are used to define the test model and the model is traced

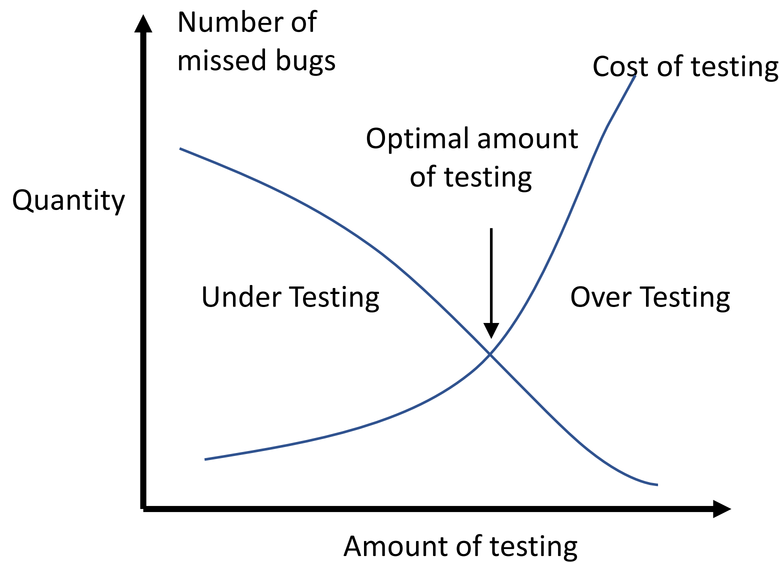


Figure 2.2: Optimal Software Testing [19]

back to the requirements to make sure that testing is performed in the right direction [23].

Random Testing - In random testing, the test inputs are selected arbitrarily or at random from the test input domain. This type of testing can be combined with other test methods to find subtle errors and is cost-effective [22].

2.2 Test Levels in the V-Model

Based on the scope of testing, test levels are defined in the V-model [21, 24] which serves as the reference model in the development and testing of automotive software [24]. The various test levels [25, 14] are shown in Figure 2.3 and their description is as follows:

Unit Testing - A software comprises of small components or modules referred to as units. Each individual unit of the software is hence tested against its specification in unit testing [24].

Integration testing - This step follows unit testing of all the software modules. Here, a group of units and its combined functionality involving their interactions are tested [26].

System Testing - In this step, the software system is tested as one single block after integrating all the units [26] and the external functionality is checked against its specifications. This is done prior to acceptance testing [24].

Acceptance testing - This is performed for the end-user or customer of the system where it is verified whether the software behaves according to their requirements [26, 24]. After this, the software is sent to production [14].

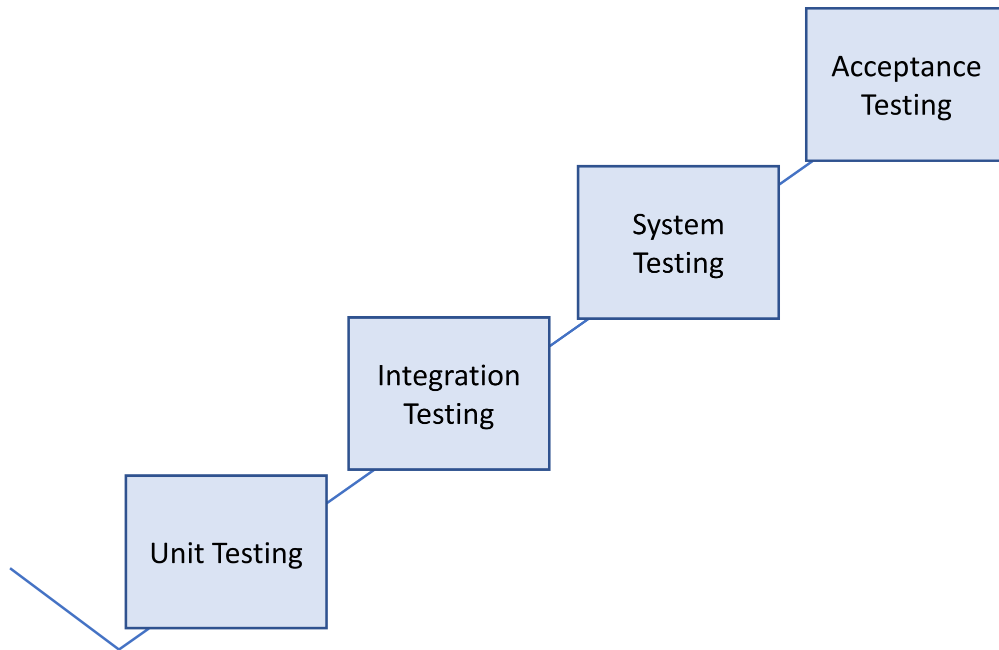


Figure 2.3: Test levels in the V-model [21]

2.3 Property-based Testing (PBT)

Conventional approaches of testing involves engineers writing test cases manually with test inputs, executing them one by one and verifying the outputs against expected results. This cumbersome, time consuming and repetitive act of manually writing tests for a system [27] has opened the way for another testing approach which is all about the word "properties". A property is the abstract behaviour of a system which should always hold. It can simply be the relationship between the inputs and outputs of a system [14], or invariants that should satisfy universally for a System Under Test [28]. A property can also be an executable formal specification of a software [29]. In PBT, we define properties and use a framework to generate test inputs which will be executed on the SUT to see if the property fails for any of the test inputs generated. Test inputs are created with the help of generators which are part of any PBT framework. We specify the input data type, number of test inputs and range of test data for the generators to create the test inputs for us [29]. With PBT, we perform a formal verification or reasoning of the system being tested as compared to the traditional methods of testing [30].

A PBT test tool reports test cases for which the property fails and these failing test cases are minimized by the tool through a process called shrinking. This test input simplifies debugging the cause of failure [14, 28] and is the smallest test input that continues to fail [27]. Without the shrinking feature of PBT, it would be difficult to manually extract information about a failing test case [14]. The failing test input is referred to as minimal test case or falsifying example. These failing inputs are also saved in a database of the tool and reused inorder to ensure that a previous failure has been successfully fixed and not reported again [14].

For testing a system via PBT, the first step is to write a property which serves as the specification for the generation of a large number of test data using a PBT tool. In PBT, the properties we write can be said to be powerful [28]. Also, it is assumed that the functionality which is of test interest is captured into a property as only this is being validated in testing [31]. Properties can be simple like algebraic functions or could also be complex like state machine models [28, 13, 32]. There are two categories of properties which are stateless and stateful properties which are discussed in the sections below.

PBT is a form of random testing [29, 13, 10, 33, 32, 14] which could often help to uncover subtle corner cases that would be difficult to be found otherwise through manual testing methods [29]. PBT can be seen to be derived from the general idea of model based testing which uses models to generate large number of test data and sometimes these test data can be directly executed on these models as well. Similarly, properties in PBT serves the same functionality of models as these are the source of test data generation as well as test oracle [28, 32].

2.3.1 Hypothesis Framework

Although PBT became popular with the tool QuickCheck which was developed for Haskell programs, a number of PBT frameworks or tools have been developed for other programming languages as well. Hypothesis is a modern framework or tool which is available for PBT in the Python programming language. It provides an extensive set of libraries which can be easily integrated into the existing testing workflow without needing to have any familiarity with Haskell programming language [34]. For the reason that Hypothesis is a stable, powerful and its easy integration into existing test suites, we choose Hypothesis for testing the system in this thesis.

Each test input generated by Hypothesis is referred to as an example and we can set the number of required examples during each test run with the help of a *settings* object. By default, the number of examples is 100. This default value is carefully chosen by the developers of the framework where the total running time of tests can be balanced against the chance of missing a bug. Hypothesis provides constructs using which we can easily define inputs required to test the System Under Test [34]. Hypothesis can also be easily integrated into Pytest which is an existing test framework to write tests ranging from small unit tests to complex functional tests [35]. We write the test cases on Hypothesis which randomly generates the test inputs with the help of the pytest suite.

2.3.2 Stateless Properties

Stateless properties can be perceived as abstractions of regular unit tests which are example-based inputs applied for testing. An example of unit test for testing a sum function is given in Figure 2.4 which consists of multiple invokes of sum function with different values of integer inputs. The outputs of the sum function are tested against the expected results for each set of input values. Such a unit test case can be abstracted and

```
def test_sum():
    assert sum(2, 5) == 7
    assert sum(10, 10) == 20
    assert sum(-2, 2) == 0
```

Figure 2.4: Regular unit tests for sum function

defined as a stateless property as shown in Figure 2.5. Here, the datatypes of inputs for the sum function are defined and these inputs are generated at random by the generator of the PBT tool Hypothesis which is used in our work. The implementation of sum function is tested with these generated integer inputs against the expected results. A set of 10 examples generated by Hypothesis for testing the sum function is given in Figure 2.6. Hypothesis also gives the statistics of the passing examples and run time of the test in the end.

Stateless properties are used to model simple synchronous functions which do not involve any change of states with the test inputs. These are ideal for testing algebraic and data operations like sum, reverse, sort etc. where inputs can be easily generated at random by the generators of the PBT tool.

```
from  hypothesis.strategies import lists, integers

@settings(max_examples=10)
@given(integers(), integers())

def test_sum(number1, number2):
    assert sum(number1, number2) == number1 + number2
```

Figure 2.5: Stateless property

2.3.3 Stateful Properties

Stateful properties are properties used to write test cases when the SUT can be modelled as a state machine. Unlike stateless properties, stateful properties consist a collection of functions with each performing various operations needed for testing the SUT. These functions and their respective functionalities are the following:

- *Setup* - This function in a stateful property sets up the SUT before each test run. It is called once in the beginning of each test run. The initial state of the SUT can be attained with the help of this function.
- *Input Commands* - Each stateful property has one or more commands which represent the transitions of the state machine being tested. These are the actual test

```
Trying example: test_sum(number1=0, number2=0)
Trying example: test_sum(number1=732581178, number_2=-29735)
Trying example: test_sum(number1=-22, number2=6759)
Trying example: test_sum(number1=1806, number2=72)
Trying example: test_sum(number1=-25, number2=799656799)
Trying example: test_sum(number1=-10, number2=99)
Trying example: test_sum(number1=20, number2=-17241)
Trying example: test_sum(number1=64, number2=-8532699149773325137)
Trying example: test_sum(number1=-12367, number2=-33)
Trying example: test_sum(number1=-8055875548987401112, number2=20872)
============================ Hypothesis Statistics ==================

sum.py::test_sum:

  - 10 passing examples, 0 failing examples, 0 invalid examples
  - Typical runtimes: < 1ms
  - Fraction of time spent in data generation: ~ 0%
  - Stopped because settings.max_examples=10

========================== 1 passed in 0.04 seconds =================
```

Figure 2.6: Example of a test run in Hypothesis

inputs which are randomly generated by the PBT tool. With these commands, the SUT changes its state from the initial state to other states.

- *Preconditions* - Each stateful property can have preconditions. An input command can trigger a transition to another state only when a particular condition is satisfied.
- *Postconditions* - Post conditions refer to functions which check the invariants that should hold true for the SUT.
- *Tear Down* - This function is called at the end of each test run which involves a cleanup.

In this thesis, stateful properties are used for testing the Service Discovery functionality at system level. This is explained in detail in the next chapter.

2.3.4 Application of stateless and stateful properties

Depending on the test requirements, the choice of properties varies for each software specification. At unit and component levels of testing, we validate single testable functions and components that are isolated [36]. As such, stateless properties are a great fit for these lower levels of testing, where test cases are required to generate a plethora of datatypes and their combinations.

When we deal with integration and system tests, we consider two or more components that are integrated and their functionality as a whole is tested. At these higher levels of testing, the System Under Test need to be modelled as a Finite State Machine (FSM)

with different inputs transitioning it into different states and outputs. As such, stateful properties are more appropriate than stateless properties. PBT tools like Hypothesis offer constructs to define stateful properties for testing the SUT in this case. Modelling the SUT with stateful properties offer a great advantage over manually writing all possible combinations of inputs and states which would be cumbersome [36].

Regardless of whether we define stateful or stateless properties, PBT tool helps us create a vast number of test inputs which exercise majority of the source code of the SUT [36].

3 System Under Test

The object under consideration whose functionality is tested is usually referred to as the System under Test (SUT) [37]. In this chapter, the underlying architecture of the Adaptive Platform (AP) and system which is being tested in this thesis are explained. To do so, we explain the concept of SOA and then discuss the Adaptive Platform by their functional clusters followed by the Service Discovery functionality of the Communication Management module.

3.1 AUTOSAR

AUTOSAR stands for Automotive Open System Architecture which is an automotive standard formulated by a group of OEMs and Tier 1 Suppliers. To tackle the growing software complexity in automobiles which could not be handled by traditional development processes and enable reuse of software components between different platforms and suppliers, software is abstracted from hardware with the introduction of the AUTOSAR standard in 2003 [38].

3.1.1 AUTOSAR Classic

The first standard released by AUTOSAR in 2005 is now referred to as AUTOSAR Classic [39]. This standard was developed for the deeply embedded ECUs which access the sensors and actuators directly and have stringent real-time requirements [8]. AUTOSAR Classic is based on signal-oriented communication.

3.1.2 AUTOSAR Adaptive

The second standard defined by AUTOSAR in 2017 is referred to as AUTOSAR Adaptive. This standard was released to address the requirements of high-performance ECUs in future vehicles needed for advanced applications like autonomous driving [39]. With this platform, vehicles which are perceived until now as closed systems get transformed into systems networked with their environment [39]. The architecture of the Adaptive Platform is designed in such a way that it is capable of processing large amounts of data from all domains with low latency [8, 40]. For this reason, the Adaptive Platform is based

on Service-Oriented Communication (SOC) which provides a great flexibility to its design. SOC is explained in detail in Section 3.2. AUTOSAR Adaptive supports SOME/IP (Scalable service-Oriented MiddlewarE over IP) as the middleware protocol [39]. Ethernet is the communication medium chosen for the Adaptive Platform as it provides seamless integration of IP-based protocols with the applications and back-end infrastructure [40, 39]. Additionally, the requirements of high bandwidth demanded by high payloads of data is also met by the Ethernet [39].

Both AUTOSAR Classic and Adaptive Platforms are not meant to substitute one another, rather they are complementary development platforms [39]. Deeply embedded systems like power train and chassis will be realized by Classic ECUs, and applications requiring high computing power will be based on Adaptive ECUs [41]. Future vehicles would consist of ECUs belonging to both these platforms [8].

3.2 Service-Oriented Architecture

The software architecture of the Adaptive Platform is based on Service-Oriented Architecture (SOA) unlike the signal-based communication paradigm in the AUTOSAR Classic Standard where signals are broadcasted between ECUs [8, 42]. As the Adaptive Platform involves advanced applications like autonomous driving, with Ethernet as the communication medium, it calls for more sophisticated protocols capable of delivering higher payload demands [41].

SOA implements Service-Oriented Communication (SOC) where services are exchanged between multiple applications on the communication system [41]. SOC facilitates the scope of new services independent of vendors, products or technologies without much change in the underlying software architecture. As such, communication paths can be established dynamically at run time as required by the Adaptive Platform [16]. The fundamental blocks of SOA and their functionalities are as follows:

Service Provider refers to a server who offers or supplies the services.

Service Consumer refers to a client who utilizes the services provided by the Service Providers.

Service Registry is also referred to as Service Broker and it enables the services to be advertised, searched and discovered [43]. The Service Providers must register their services at the Service Registry which can then be found by the Service Consumers. This process by which the services are discoverable is called Service Discovery [44]. Service Registry thereby links the Service Providers and the Service Consumers in SOC as depicted in Figure 3.1.

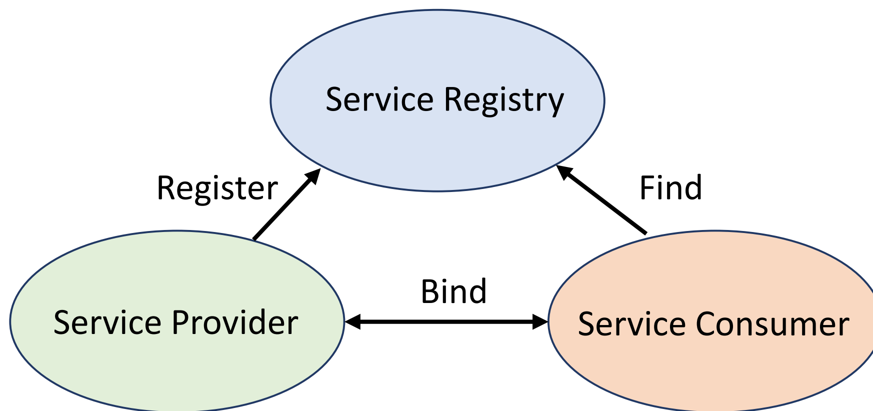


Figure 3.1: Service-Oriented Communication [43]

3.3 Architectural Overview of Adaptive Platform

The modules of the platform, also referred to as functional clusters, form the AUTOSAR Runtime for Adaptive Applications (ARA). The functional clusters provide the interfaces for Adaptive Applications and these are of two types; Adaptive Foundation and Adaptive Services. Adaptive Foundation consists of APIs (Application Programming Interfaces) which are the basic blocks providing the functionalities of the Adaptive Platform (AP). Adaptive Services are the standard services of the AP [45, 46]. In Adaptive Platform, ARA links the clients to the services dynamically at runtime.

Each Adaptive ECU contains a few instances of functional clusters belonging to Adaptive Foundation. Adaptive Services can be distributed across the automotive network [6, 46].

Figure 3.2 shows the architectural structure of the Adaptive Platform. The significance of each of the functional clusters belonging to Adaptive Foundation is as follows:

- **Operating System Interface**

  Operating System Interface denotes the functionalities of the hosting operating system (POSIX PSE51). It initialises the System on ECU startup and hands over the control to the Execution Management. It manages resources, performs run-time scheduling and inter-process communication for Adaptive Applications.

- **Execution Management**

  Execution Management is responsible for the initialization of the Adaptive Platform as well as the startup/shut down of the applications in ECUs. It manages the resources necessary for running applications and works in conjunction with the Operating System [45, 47].

- **Communication Management**

  The functional clusters - Communication Management (CM) and REST are communication stacks that establish communication between Adaptive Applications (AAs).

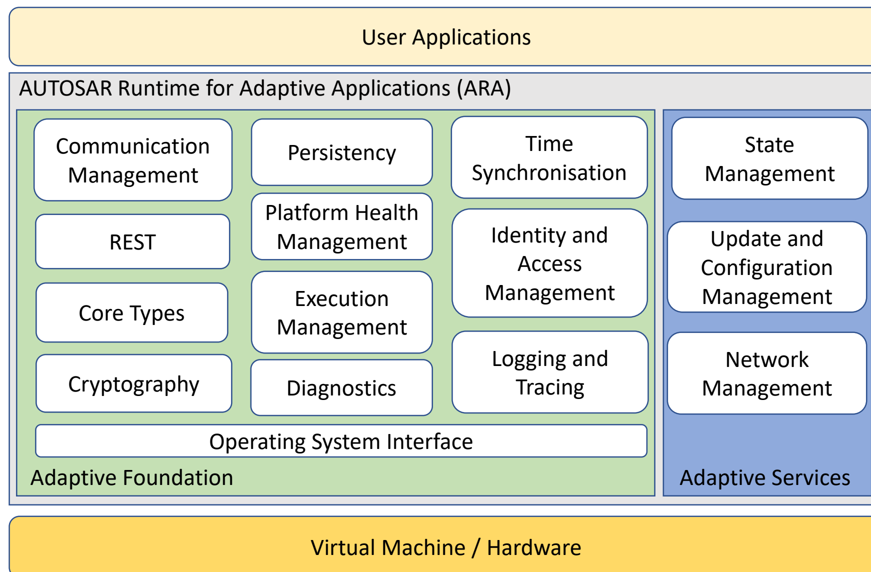


Figure 3.2: Adaptive Platform Architecture [9]

CM offers Service-Oriented Communication for AAs and abstracts the underlying network protocol (SOME/IP) or bus (Ethernet, CAN) used for communication [9, 48].

- **REST**

  REST is a framework which originated from the web-based stateless APIs to build RESTful APIs for Adaptive Applications. AAs can offer and request RESTful services with the help of this functional cluster [9].

- **Peristency**

  Persistency allows the other functional clusters to store information in the non-volatile memory of an ECU. It also provides interfaces to access the stored data later [9].

- **Time Synchronization**

  Time Synchronization offers a mechanism through which events across the system are correlated, tracked and triggered at an accurate point in time. It also offers interfaces to AAs for retrieving timing information synchronised with other applications or ECUs [9].

- **Logging and Tracing**

  Logging and Tracing provides APIs for AAs to record or log events with severity levels such as debugging, informative, warning, error etc. [48]. Logs are written to a file on the system, or a serial console, or simply forwarded on the communication bus [9].

- **Cryptography**

  Cryptography provides APIs for performing cryptographic operations and secure key management [9].

- **Identity and Access Management**

  Identity and Access Management manages permissions so that AAs can be granted or denied access to the other functional clusters [48].

- **Platform Health Management**

  Platform Health Management allows to supervise applications to be able to check whether they are running frequently or rarely, within minimum and maximum time limits etc. It also allows to configure certain recovery actions in cases of failure of supervised applications [9].

- **Diagnostics**

  Diagnostics offers APIs for communication and events with regard to in-vehicle or remote diagnostics [9].

- **Core Types**

  Core Types offers interfaces which provide certain functionalities which are accessed by multiple functional clusters [9].

Following are the functional clusters belonging to Adaptive Services:

- **State Management**

  State Management is responsible for handling the operational state of the Adaptive Platform. It handles incoming events and prioritizes them to trigger Execution Management to set the internal states [9].

- **Update and Configuration Management**

  Update and Configuration Management provides the flexibility to the Adaptive Platform to update the software components and configuration through over-the-air (OTA) updates. It is responsible for installing, removing and keeping a record of a software component on the Adaptive Platform [9].

- **Network Management**

  Network Management deals with managing the state of the communication bus and keeping the connected ECUs active with periodic messages [9].

3.4 Communication Management (ara::com)

One of the functional clusters of Adaptive Foundation - Communication Management, also referred to as ara::com, is being tested in this thesis. Ara::com is the communication middleware which realizes the Service-Oriented Communication of the Adaptive Platform. It is responsible for communication between applications local to an ECU and remote applications which reside in other ECUs. This also includes SOC with Adaptive Services [42, 46].

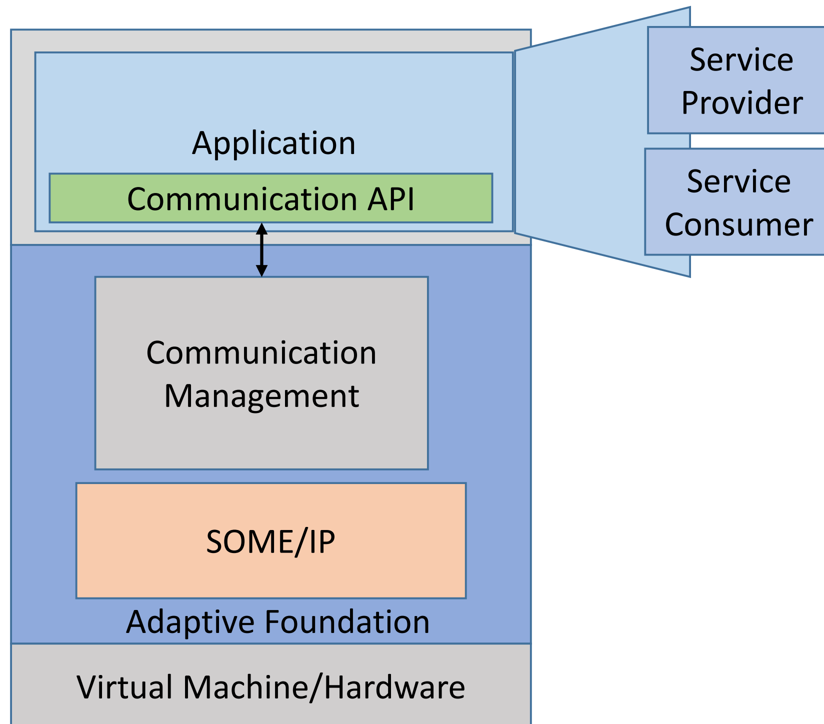
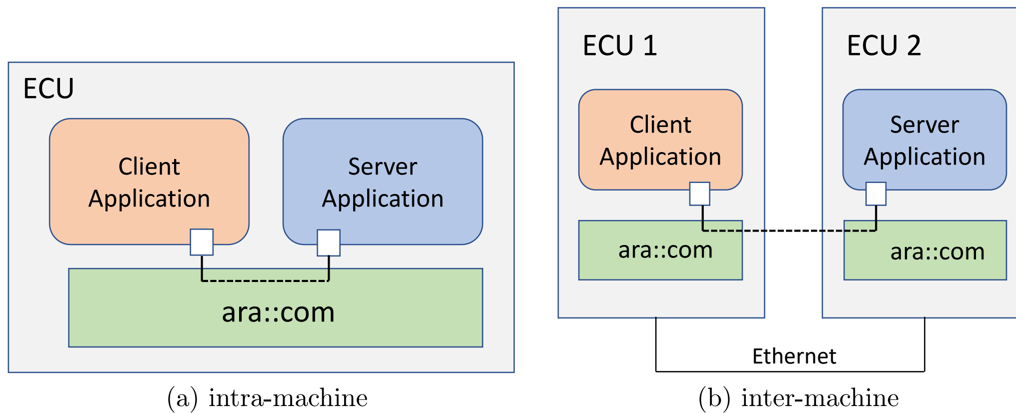


Figure 3.3: SOC via Communication Management [49]

Adaptive Applications (AAs) consists of one or more processes with each process involving one or more threads [50]. Each Adaptive Application can provide services to other Adaptive Applications. A service refers to a functionality offered by an Adaptive Application which can be requested by other applications on the Adaptive Platform [42]. Services are independent of the underlying software platform (AUTOSAR Classic/Adaptive or non-AUTOSAR). Applications providing services are called Service Providers or server applications and those requesting services are called Service Consumers or client applications (see also Section 3.2). This is illustrated in Figure 3.3.

Client and server applications could be located in the same ECU or remotely in different ECUs. Ara::com provides the interfaces for Service-Oriented Communication in both of these cases. If the client and the server reside in the same machine, the communication is said to be intra-machine. If the communication between client and server happens remotely, it is inter-machine communication. These are shown in Figure 3.4. For intra-machine communication, ara::com is the direct interface [9] whereas for inter-machine



(a) intra-machine (b) inter-machine

Figure 3.4: Communication on Adaptive Platform

communication, ara::com routes the data over an Ethernet network which is also referred to as over the wire [51].

Each Adaptive Application providing services registers each of its services at the Service Registry which is part of ara::com. Each client application can find the registered services by querying the Service Registry. This mechanism to offer and find services, thereby connecting the communication partner applications, is abstracted by ara::com. Ara::com connects the clients and servers dynamically at runtime [40, 39].

SOC through ara::com is accomplished with the bus protocol SOME/IP [42, 8] (as shown in Figure 3.5 for inter-machine). SOME/IP stands for Scalable service-Oriented MiddlewarE over IP and is an automotive communication protocol based on client/server architecture [46]. However, there is no dependency for SOC on any communication protocol [52]. SOME/IP network protocol could be implemented on different operating systems and even on embedded systems without any operating system (like non-AUTOSAR platforms) [53].

Service-Oriented Communication over SOME/IP consists of remote procedure calls and event notifications. Serialization of the data is also handled by SOME/IP [53]. Unlike the classical approach where all the data is broadcasted over the bus, clients request a particular service only when it is needed. After requesting a service, a client can subscribe to events and call methods on the server [16].

A service is a combination of zero or multiple events, methods, and fields [53]. Events refer to data which is sent from servers to clients either cyclically or on an update, after the client has subscribed to it. Methods are the remote procedure calls that can be made by a client on the server application. Fields are a combination of events and methods. Clients can set the value of a field on the server side as well as they can get the present value of a field from the server. Further, clients get notified whenever the value of a field changes after subscribing to it.

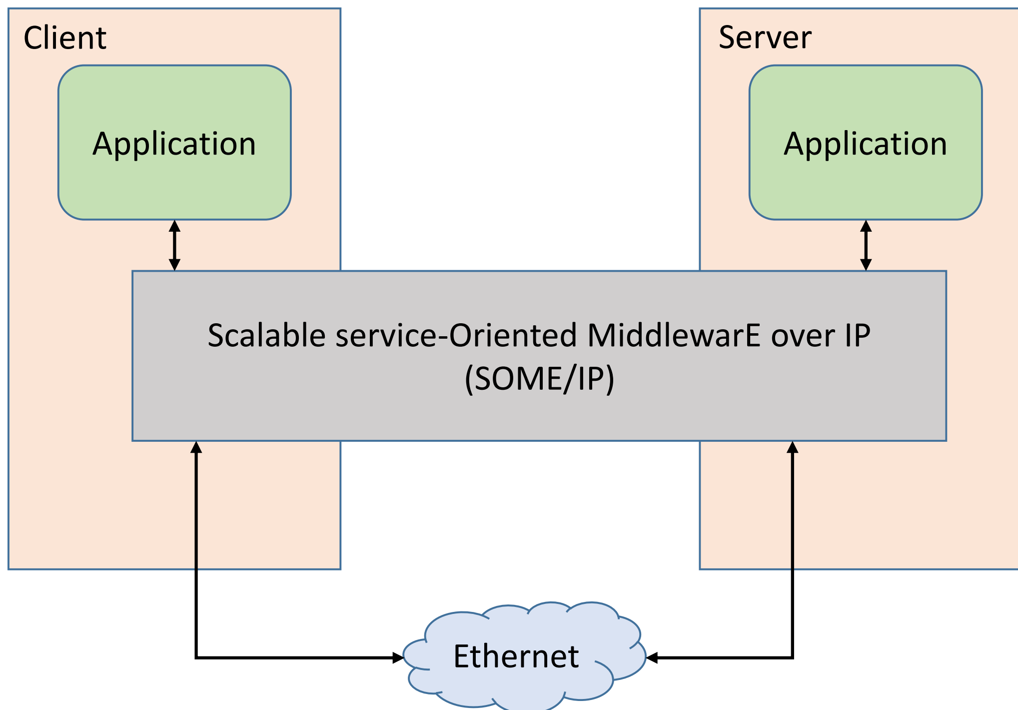


Figure 3.5: SOME/IP Protocol [54]

3.5 Service Discovery

Service Discovery (SD) as already mentioned in 3.2 is implemented by ara::com to enable Service-Oriented Communication on the Adaptive Platform. The main task of SD is to manage the availability of services on the network [55]. Service Registry is a part of ara::com which acts as a brokering instance in SOC.

With Service Discovery, different applications on ECUs can offer services and find available services within the vehicle network. An application can be simultaneously a client as well as a server. When the application offers a service to other applications on the vehicle network, it becomes a server of that service. When the application uses a service offered by another application residing locally or remotely on the network, it becomes a client to that service [55].

After the startup of the vehicle network from the initial powered down state, the applications are started by the Execution Management [56]. Service Discovery decides whether the communication is established internally or externally [52]. This is dependent on the location of the Service Provider on the network. Finding services is enabled through SOME/IP Service Discovery and is divided into Local and Remote SD. Local SD occurs for services running in the same ECU (intra-machine) and Remote SD is for the services over the vehicle network (inter-machine) [57]. SD can be viewed at the application level as well as the network or bus level which are explained in the sections below.

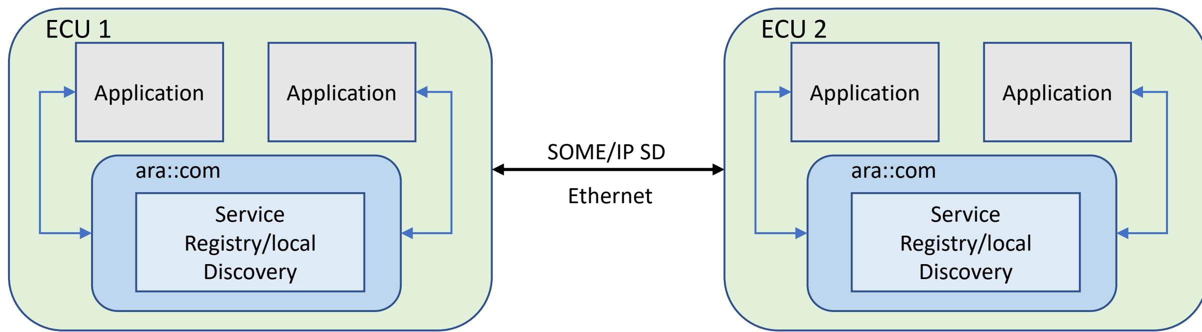


Figure 3.6: Local and SOME/IP Service Discovery [51]

3.5.1 Application View

Both local and remote SD are initiated by the interface calls implemented by ara::com at the application level. These calls are made by the AAs to start the SOC. The functionality of Service Discovery as seen by the applications is tested in this thesis. This is explained in this section.

Each service is identified with a Service Identifier (Service ID) [58]. Ara::com provides the interfaces so that the applications can offer and find services through Service Discovery [59]. Following are the interfaces implemented by ara::com which are used by the applications:

StartOfferService() - Any Adaptive Application (AA) can call this method when it wants to provide a service to other applications [51]. After this method has been called on a service by an application, the service can be requested by all other applications on the network.

StopOfferService() - Any AA which wants to stop providing a service which has been previously offered can call this method. After calling this method on a service by a server application, the service can no longer be requested by the client applications.

StartFindService() - Any AA which is a Service Consumer calls this method once in an attempt to query the availability of a certain service. This interface call registers a handler in ara:::com which gets called upon detection of a matching service [52]. If the service has been previously offered, it returns back a handle from the Service Registry of ara::com with the status that the service is available along with the location of the Service Provider. If the service has not been previously offered and the local Service Registry does not know the current state of the requested service, the client sends a SOME/IP-SD *FindService* message on the bus. This is necessary to notify the local Service Registry of the client application as soon as the service is offered by any server application. After calling the StartFindService once by the client, a handler is triggered each time the status of the service changes, i.e. when it becomes available and unavailable, unless and until the client calls the method *StopFindService()* on the same service.

StopFindService() - As mentioned before, to disable the triggering of callbacks as part of monitoring the availability of a service with *StartFindService()*, this method is called by

the client application. As a result, the client is no longer notified of the changes in the status of the service until it explicitly calls *StartFindService()* again.

3.5.2 Bus View

The interface calls at the application level need to be communicated at the bus level whereby applications residing in different ECUs can be connected. This is done through Service Discovery (SD) specified by the SOME/IP protocol. SOME/IP-SD enables the mechanism by which the Service Registries of the remote applications are synchronized.

As soon as the ECUs have completed local SD, SOME/IP-SD messages are sent to all other connected ECUs over the Ethernet which contain one or more *FindService* and *OfferService* entries.

SOME/IP-SD *OfferService* messages are sent cyclically and contain information about the services provided by server applications. Upon reception of this message by the Service Registries of other machines, the client applications residing on these machines can request these services. When a client triggers a *StartFindService()* for a service on the application layer and the local Service Registry cannot resolve the state of the service, SOME/IP-SD *FindService* messages are sent over the bus. Upon reception of these, the remote machines check for a match if the service has been offered locally. If a match is found, the machine sends a SOME/IP-SD *OfferService* message for the service over the bus [57].

After the startup of the SOME/IP-SD processes, the clients and servers pass through the following three phases:

Initial Wait - This is the first phase the clients and the servers enter after the initial down state. This phase is required so that the possibility that the system could be jammed by initial bursts of data is reduced [56]. Additionally, this time could be utilised for the local SD and the system can begin to send SOME/IP-SD messages on the network [57].

Repetition - After the initial wait phase, applications begin to send SOME/IP-SD messages with *FindService* and *OfferService* entries over the network. These messages are sent in regular intervals [60].

Main - This phase starts after the Repetition phase during which SD stabilizes the system by sending no more *FindService* entries, rather respond to the incoming *FindService* messages. This is done by sending SOME/IP-SD messages cyclically with *OfferService* entries containing the location of the Service Provider in regular intervals [57, 60].

In this chapter, we discussed the System Under Test in detail including its architecture and functional view of SD at the application and bus levels. In the next chapter, we describe the test specification for ara::com released by AUTOSAR and how conventional methods of testing are carried out. Lastly, we mention how we derive properties for SD to be used in our test approach.

4 Specifications and Test Design

In this chapter, we discuss the test specification released for ara::com by AUTOSAR followed by existing methods of testing based on this specification. Then we describe how we derive properties for our test infrastructure.

4.1 AUTOSAR Specifications for ara::com

AUTOSAR has released specification and requirement documents for ara::com module which specify the functional requirements for the developers of ara::com module. For performing tests at system level, AUTOSAR has released “System Tests of Adaptive Platform" which consists of test specifications for each of the modules of the Adaptive Platform. An excerpt from the test specification for local and remote SD consisting of steps of tests and their corresponding pass criteria is shown in table 4.1.

Table 4.1: Test Specification for local and remote Service Discovery [11]

Number	Test Step	Pass Criteria
1	Offer service [SERVICE3]	
2	Offer service [SERVICE1]	
3	Request service [SERVICE3]	Service 3 is available
4	Stop Offering service [SERVICE3]	Service 3 is not available
5	Request service [SERVICE1]	Service 1 is available

4.2 Comparison of Test Strategies

In this section, we would compare the conventional method of testing the ara::com module based on the given test specification to that of our test approach.

4.2.1 Example-based Testing

Conventional methods of testing the ara::com module at system level involve testing it with well-crafted inputs and verifying whether it behaves as expected. This form of testing is referred to as example-based testing as the SUT is tested against the expected behaviour with manually written examples.

Test case is written with reference to each step in the test specification as given in table 4.1 and the corresponding pass criteria are checked to determine if any of the test steps failed. Figure 4.1 gives an idea of an example-based test for the above test specification of ara::com. From figure, the methods are performed for the steps of the test specification followed by assertions to check whether the corresponding pass criteria are obtained. In example-based testing, the test generation is manual which means we write each individual test case manually but the test execution can be automated.

```
StartOfferService(service=3)
StartOfferService(service=1)
StartFindService(service=3)
assert Service3.available() == True
StopOfferService(service=3)
assert Service3.available() == False
StartFindService(service=1)
assert Service1.available() == True
```

Figure 4.1: Example-based testing

4.2.2 PBT

As we discussed PBT in detail in 2.3, the initial effort of formulating the properties requires a good understanding of the software. We define properties which refer to the abstract behaviour of the system being tested. We input the properties to a PBT tool to generate the test cases which are then used to test the SUT. A huge number of test cases are generated randomly by the PBT tool which tries to falsify the defined property. If the framework finds any failing test case, it shrinks the test case and outputs the same as a minimal test case. This minimal test case can be used to debug the issue or failure which has been found. PBT offers a test mechanism where we can generate extensive test cases based on the properties that we define. Unlike example-based testing which consists of test cases with each verifying a single functionality, PBT tool can generate a large number of test inputs based on a single property. In PBT approach, the test generation and test execution are both automated as the framework automatically generates the test inputs from the defined property.

4.3 Deriving Properties from Specifications

In order to formulate a property, we need to understand the functionality of the SUT. After attaining the system knowledge, we need to find a feature in terms of the SUT's functionality such that it should always hold true. In our case, as we test Service Discovery, we define the properties - *A service is always available after it has been offered and a service is unavailable after it has been stopped.* This is universal for the services in SD and should always hold true. As such, this feature of the services qualifies to be a property in our PBT testing approach. The PBT framework Hypothesis would then try to violate this property with the test inputs that it randomly generates. The test implementation details for the property that we defined are described in section 5.6.

In comparison to example-based testing, we require more efforts in the beginning of our test approach in understanding the system in terms of its abstract functionality. This is inevitable as this step determines the property's quality which would represent the SUT and based on which the test inputs are generated.

In this chapter, we discussed how the traditional testing method differs from PBT approach and how we derive properties for the SUT. In the next chapter, we describe how we implement the test infrastructure based on PBT approach for the properties that we defined.

5 Implementation

In this chapter, we describe our implementation of the test infrastructure based on PBT. We start with the block diagram of the system being tested in this thesis and then describe how we integrate the Hypothesis testing framework with the former. After that, we explain how we apply stateless properties to determine the communication delay due to the asynchronous behaviour of the API calls on the SUT. Later, we provide the state diagrams of the test model of a service required for PBT test cases along with that of the actual test implementation of a service. Finally, we describe how we use stateful properties to carry out the test implementation of Service Discovery.

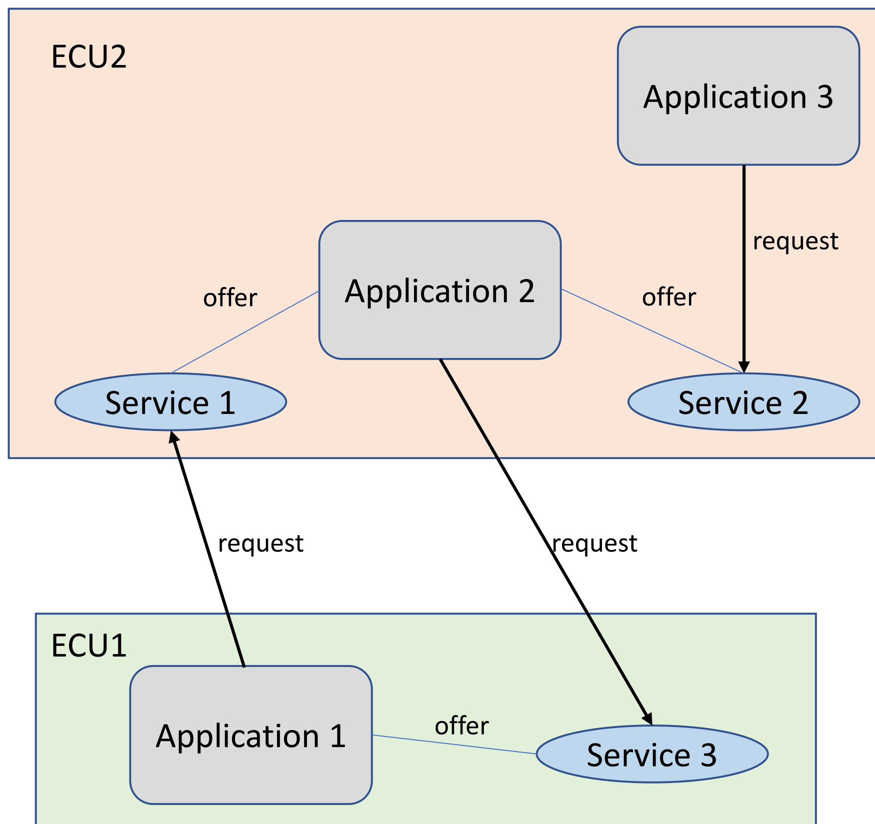


Figure 5.1: Block Diagram of the SUT

5.1 Block Diagram of SUT

We explained the theoretical aspect of Service Discovery and how it works on the application level in Section 3.5.1. In this section, an overview of the system which is being tested in our work is provided. As we already outlined in chapter 3.5.1, we test whether the applications can offer and find services with the SD implemented by the ara::com modules. The system being tested in this thesis consists of two ECUs each having its ara::com modules. ECU 1 hosts Application 1 which offers Service 3. ECU 2 hosts two applications - Application 2 and Application 3. Application 2 offers Service 1 and Service 2. Service 3 is requested by Application 2 and Service 1 is requested by Application 1. Application 3 requests Service 2 and offers none. The block diagram of the SUT consisting of ECUs with applications is shown in figure 5.1.

We trigger the applications to offer services with *StartOfferService()* and stop offering them with *StopOfferService()*. The requesting applications can find the required services with *StartFindService()*. An API call is made with the corresponding Service ID which uniquely identifies each service.

In the next section, we explain how we integrate the framework Hypothesis which we use for testing in this thesis with the SUT. This will route the communication from the test cases that generated by the Hypothesis to the ECUs and vice-versa.

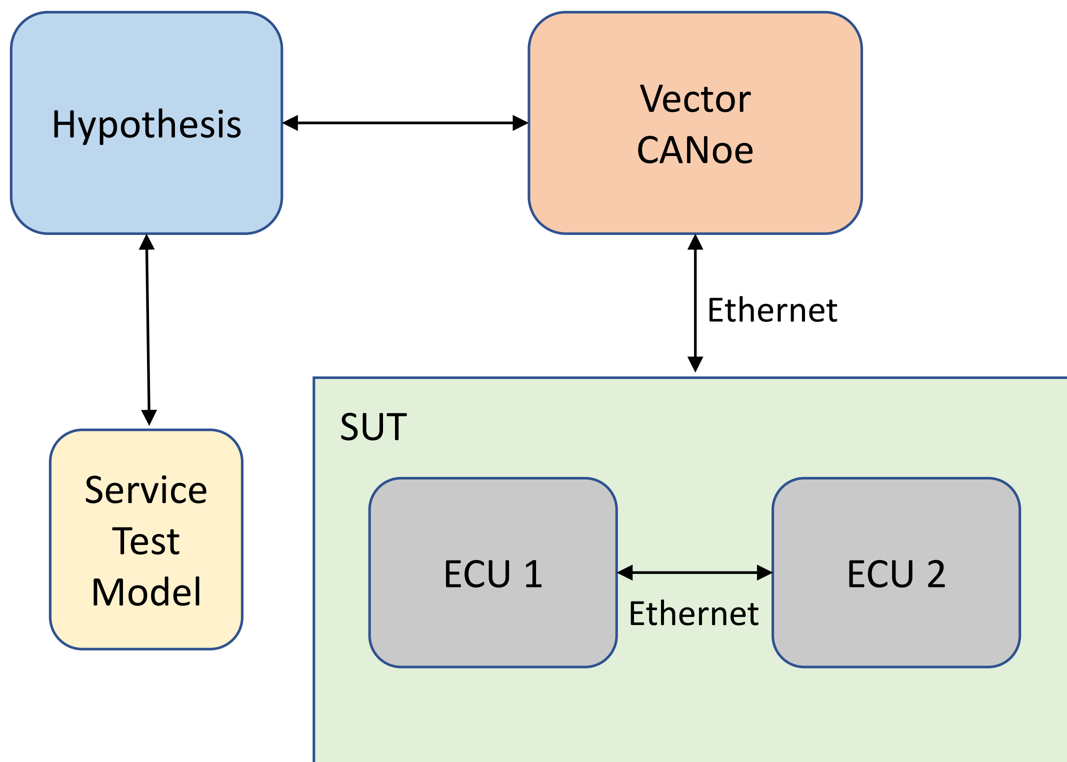


Figure 5.2: Setup of the PBT test system for SD

5.2 Integration of Hypothesis with the Adaptive Platform

The PBT framework Hypothesis running on PC is connected to the Adaptive ECUs through the Vector tool CANoe which acts as a runtime environment. An interface is implemented for routing the inputs to the ECUs under test so that SD can be tested. The test inputs generated by Hypothesis are communicated through CANoe by setting variables on the latter which triggers the corresponding methods on the SUT through the Ethernet. These test inputs are simulatenously run on the service test model as well, so that the expected behaviour can be known for testing. The response from the SUT is sent back to CANoe which sets variables on the latter. The CANoe variables are read in our test cases on Hypothesis for comparing the outputs of the SUT with the expected results. The integrated test setup is shown in Figure 5.2.

The Adaptive Applications are hosted on Renesas R-Car hardware as shown in figure 5.3 which serves as the ECU system being tested in our work. This hardware provides a

Figure 5.3: Renesas R-Car ECU running Adaptive Applications [61]

higher computing performance required for applications on the Adaptive Platform. R-Car series also conform to the functional safety standard ISO 26262 of the automotive industry [61].

5.3 Stateless Property

In this section, we describe how we use stateless properties to determine the delay in communication between the testing framework Hypothesis and the SUT. As we deal with testing asynchronous software, the results will not be available to us immediately after performing a test operation on the SUT. For testing, we need to know the minimum time required to make sure that the result is available for verification after performing an API call on the SUT. For this purpose, a stateless property is applied to generate test cases in order to find this boundary value of time.

There is a non-determinism associated with real time systems which involves variances in timing-related properties [62]. Therefore, we can find the minimum value range of waiting time for which the output validation fails. The stateless property is shown in Figure 5.4. The interface call *StartFindService()* for Service 1 is made in the beginning of the test cases in the *Setup* method so that the event handler is triggered as soon as Service 1 is offered. Consequently, Service 1 becomes available to Application 1 requesting it.

The test method contains an interface call *StartOfferService()* for Service 1. We want to find the minimum value of time t that is required so that we can successfully assert that Service 1 is available. The value of t is generated at random by the Hypothesis strategy, the range of which is set using a *minimum value* and *maximum value*. The Hypothesis strategy is set to start from a minimum value of 0 to 500ms. As it starts the test run from 0ms when the test run does not wait at all after calling *StartOfferService()*, the service is not available and therefore assertion succeeds. Hypothesis would then input higher values for t thereby trying to falsify the property i.e., to fail the assertion. For example, suppose we wait for 500ms after performing *StartOfferService()*, there is sufficient time for the communication to return the response value from the SUT back to Hypothesis and therefore the service becomes available.

Table 5.1: Boundary Values of time obtained with stateless property.

Number of examples	Maximum time for all examples (s)	Minimal Value of t (ms)
13	37.23	272
73	110.33s	250
11	35.32	250
11	36.07	250

```
class MyTest(unittest.TestCase):

    @classmethod
    def setUp(cls):
        StartFindService(1)


    @given(t= strategy.floats(minimum value=0, maximum value=0.5))
    def test(self, t):
        StartOfferService(1)
        time.sleep(t)
        try:
            assert Service1.Available() == False
        finally:
            StopOfferService()
            time.sleep(1)
```

Figure 5.4: Stateless property to find the delay needed after an asynchronous API call

The results of the test run to determine the boundary value of time is shown in Table 6.3. From the table, the minimum value of time obtained after shrinking by Hypothesis is 272ms during the first test run and 250ms in the later runs. Even with these values, the assertions passed in some examples and failed in others. From the test results, the average waiting time that we need to include after an asynchronous operation is 255.5ms. In order to ensure that we provide sufficient delay considering any timing variance that can arise due to real-time behaviour of the SUT, we propose a waiting time of 300ms before we check the output from the SUT following an API call on it.

5.4 Service Test Model

As we deal with black-box testing, we don't have access to the source code of the SUT. As such, we cannot know which state a service would be in, so that we can assert that it is available or not available in our test case. For formulating expected results in our testing, we propose to use a model for our testing. As we are testing the SD and the outputs of the SUT which we test is related to the availability of services, the test model represents a service as depicted in Figure 5.5. A service has two states “Not Offered"(NO) and “Offered" (O) and two types of inputs (*StartOffer(), StopOffer()*) transitioning them from one state to another. The test inputs generated using our PBT test cases will be executed on this service test model so that it can be used as a reference for the expected results in our test assertions.

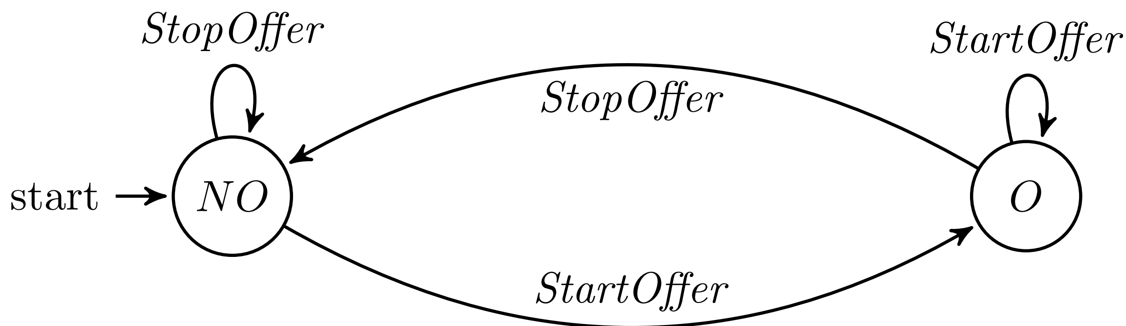


Figure 5.5: Test model representing a service.

5.5 State diagram of a Service

We discussed the service test model above where services have two states NO and O. When a service is in Not Offered state, it is not available to the client application requesting it and therefore its availability is asserted against *False*. When a service is in Offered state, it becomes available to the client and therefore, its availability is asserted against *True*. In this section, we describe the model of a service as implemented by our PBT test case.

In 5.3, we have seen that there is a minimum communication delay of 300ms involved after a test input operation on the SUT and we can perform an assertion only after a waiting period. These assertions testing the availability of the services are included in a *Check()* operation in our test case. In our test case implementation, we have an additional criterion to define the state of a service which is based on the *Check()* operation. This criterion provides two other aspects to service states -"Not Checked" (NC) and "Checked" (C). When a service is in NO state and its availability is verified with the *Check()* as *False*, the service is in Not Offered/Checked (NO/C) state. Similarly, when a service is in Offered state and its availability is verified with *Check()*s as *True*, then the service is in Offered/Checked (O/C) state. These *Check()* are performed only after the minimum delay of 300ms. The state diagram of the test implementation of a service is shown in Figure 5.6.

Initially, a service is not offered and not checked as well and therefore is in Not Offered/Not Checked (NO/NC) state. With *StartOffer()*, they can go to Offered/Not Checked (O/NC) state and again after 300ms, a *Check()* can transition them to Offered/Checked (O/C) state. A *StopOffer()* can make them go back to Not Offered/Not Checked (NO/NC) state which is the initial state.

With the delay needed for the asynchronous operations in our testing, we have four states for a service - NO/NC, NO/C, O/NC and O/C. In the next section, we describe how we apply stateful property to model the service state machine for our test approach.

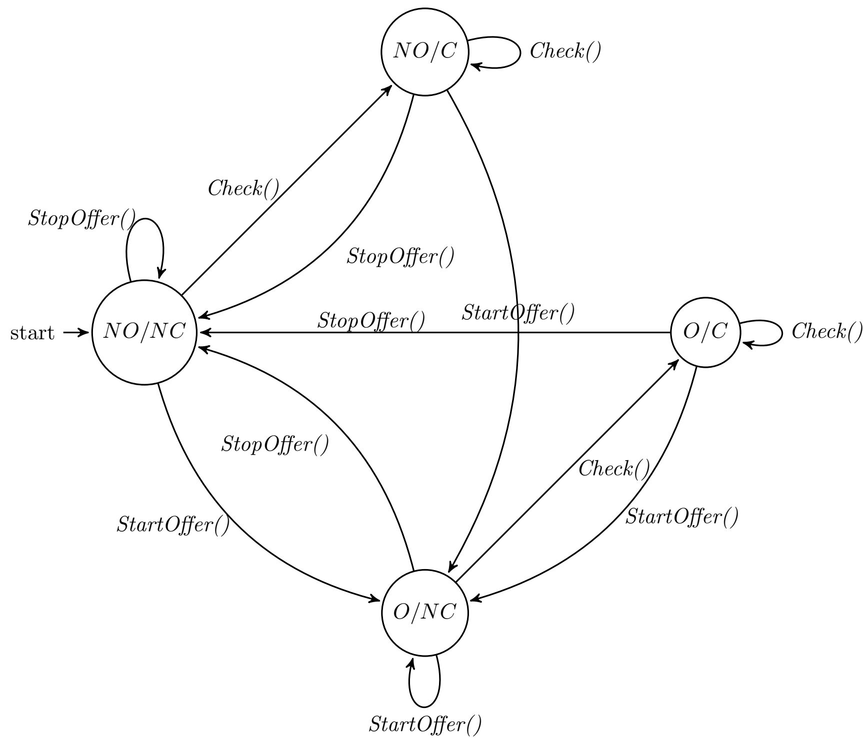


Figure 5.6: Implemented model of service

5.6 Stateful Property

PBT in its basic form fits in unit testing and can by means of stateful properties be used in testing integrated components [30]. Testing the functionality of Service Discovery involves dealing with services having two states - Not Offered (NO) and Offered (O), which attributes to the characteristics of stateful systems. For testing such stateful systems, stateful properties are more appropriate and hence we prefer these over stateless properties.

For performing PBT at the sytem level, we consider the ara::com modules in the two ECUs as black box whose functionality of Service Discovery is viewed as if they were a single component. The major challenge in testing our system is that we are dealing with a system which has aynchronous behaviour. That means we have to wait some time until the system sends back a response to our request. This is unlike any synchronous functions which are single test functions like sort and reverse for lists where PBT is commonly demonstrated. As discussed in 5.3, we have found the wait time to be atleast 300ms before we can successfully validate the result of an operation.

We use the construct *RuleBasedStateMachine* offered by the Hypothesis framework for stateful testing as shown in Figure 5.7. We can perform the startup of the application on the ECUs at the beginning of the tests using an *initialize* operation. *Initialize* is generated once at the beginning of each example by Hypothesis. In this initialisation operation itself, we are triggering the applications to perform the *StartFindService()* for all the services once

```
from hypothesis.stateful import RuleBasedStateMachine

class test_servicediscovery(RuleBasedStateMachine):

    @initialize(target=None)
    def initialize_ECU(self):
        Starts up applications
        Calls StartFindService() on SUT

    @rule(service=st.integers(min_value=1, max_value=3))
    def StartOffer(self, service):
        Initiates StartOffer on service test model
        Calls StartOfferService() on SUT

    @rule(service=st.integers(min_value=1, max_value=3))
    def StopOffer(self, service):
        Initiates StopOffer on service test model
        Calls StopOfferService() on SUT

    @rule(t=st.floats(min_value=0.3, max_value=0.5))
    def wait():
        time.sleep(t)

    def check(self):
        assert service_status(SUT) == service_status(servicemodel)

    def teardown(self):
        Closes up applications
```

Figure 5.7: Stateful property for testing Service Discovery

in the beginning. This will trigger callbacks according to the changes in the availability of services i.e., when offered and not offered. To test the SD functionality, we define two operations *StartOffer()* and *StopOffer()* to trigger the corresponding *StartOfferService()* and *StopOfferService()* on the SUT respectively. The operations are called with an integer value corresponding to Service ID of the three Services 1, 2 and 3. The integer values are generated by a construct called rule which passes these values to the *StartOffer()* and *StopOffer()* functions. These two operations are generated at random by Hypothesis which are the test stimuli of our testing.

In testing, functions to verify whether an operation has performed correctly are as important as the test inputs. For this reason, we define the operation *Check()* which is performed after each *StartOffer()* and *StopOffer()* command. A *Check()* after a *Startoffer()* will verify whether the service has become available and a *Check()* with the previous command *StopOffer()* will verify whether the status of the service has become unavailable. To verify that there is no dependency between the services, we perform the *Check()* operation for all services after each command. This helps us to ensure that a command performed for a service does not change the state of another service.

For these verifications, the *Check()* operation contains assertions. The response from the SUT has to be asserted against the expected results. The expected results for the *Check()* will be obtained from the service test model as described in Section 5.4. Even though the *Check()* is triggered after each command, sometimes the execution is faster, such that 300ms has not passed since the previous command so that we can successfully check the results. To provide this delay which is needed for the *Check()*, we have implemented a *wait()* rule for generating the waiting time required for our testing purpose. We define a range of values from 300 to 500ms for this rule. This allows us to wait for the asynchronous operations to respond and verify the results only after the communication delay.

In short, we have the property defined as "A service is available to a client after it has been offered by a server application and a service is not available to a client after it has been stopped by a server application". This property should always hold true for Service Discovery. Hypothesis will repeatedly throw various sequences of *StartOffer()* and *StopOffer()* operations, until it finds that the property has been violated or a limit of examples is reached without having the property violated. An example consisting of a set of the stateful operations generated by Hypothesis for testing SD is shown in Figure 5.8.

To summarize, we discussed how we implemented the test setup for SD with stateful properties and how we applied stateless property to determine the communication delay in our approach. In the next chapter, we evaluate our approach to determine its effectiveness and efficiency. Later, we discuss the findings of the interview study conducted to draw comparisons between PBT and existing methods of testing.

```
Trying example: run_state_machine(factory=test_servicediscovery)
state = test_servicediscovery()
state.initialize_ECU()
state.StopOffer(service=2)
state.wait(t=0.3216752675645543)
state.StopOffer(service=3)
state.StartOffer(service=2)
state.StopOffer(service=2)
state.StartOffer(service=1)
state.wait(t=0.24927510296699018)
state.StartOffer(service=1)
state.wait(t=0.07147243285476228)
state.wait(t=0.19881836315522916)
state.StopOffer(service=1)
state.StopOffer(service=2)
state.wait(t=0.24808102214860947)
state.wait(t=0.15580801691972143)
state.StartOffer(service=2)
state.StartOffer(service=2)
state.StartOffer(service=1)
state.StopOffer(service=2)
state.wait(t=0.3870595349351187)
state.wait(t=0.35939414379939355)
state.StartOffer(service=2)
state.wait(t=0.27149823194099937)
state.StopOffer(service=1)
state.wait(t=0.34343588454274887)
state.StartOffer(service=2)
state.wait(t=0.2822209382959738)
state.StopOffer(service=1)
state.StopOffer(service=3)
state.StopOffer(service=1)
state.wait(t=0.35138062760339156)
state.StopOffer(service=2)
state.StartOffer(service=1)
state.StartOffer(service=3)
state.wait(t=0.04939774471219347)
state.StopOffer(service=1)
state.wait(t=0.332831603726529 8)
state.StopOffer(service=3)
state.StartOffer(service=1)
state.StopOffer(service=2)
state.wait(t=0.11377908760715622)
state.StopOffer(service=1)
state.StartOffer(service=2)
state.teardown()
```

Figure 5.8: An example test generated by Hypothesis for SD

6 Results and Evaluation

In this chapter, we discuss the results gained during test executions of our implementation for the Service Discovery of ara::com module. We evaluate the effectiveness and efficiency of our approach. We mention the issue in configuration file identified, the method of fault injection to evaluate the minimal example obtained during testing and then we analyse the test inputs with respect to some of the test cases that we would otherwise perform manual testing with. We describe the findings of the interview study conducted during this thesis for comparing our approach to existing test methods.

6.1 Effectiveness

In this section, we describe how we evaluated the effectiveness of our approach for testing Service Discovery. According to Eldh et al., effectiveness of a test suite can be described in terms of coverage and the number of faults identified with the test approach [63]. First, we describe an issue which was found and reported by the Hypothesis during our test run. As the services are independent of one another, we determine whether we can evaluate the concurrency of these services at any point during our test run. Then, we use the approach of finding state and transition coverages of our test implementation for the three services. Later, we analyse the test inputs generated by Hypothesis for our testing.

6.1.1 Issue found during testing

The PBT test run by the Hypothesis found an issue during the test run which was due to a mismatch error in the port configuration files used for the applications in SD. The

```
Falsifying example: run_state_machine(factory=test_servicediscovery,
                                      data=data(...))
state = test_servicediscovery()
state.initialize_ECU()
state.StartOfferService(service=1)
state.StartOfferService(service=3)
state.StopOfferService(service=3)
state.wait(t=0.4)
state.teardown()
```

Figure 6.1: Minimal example for the port mismatch error detected

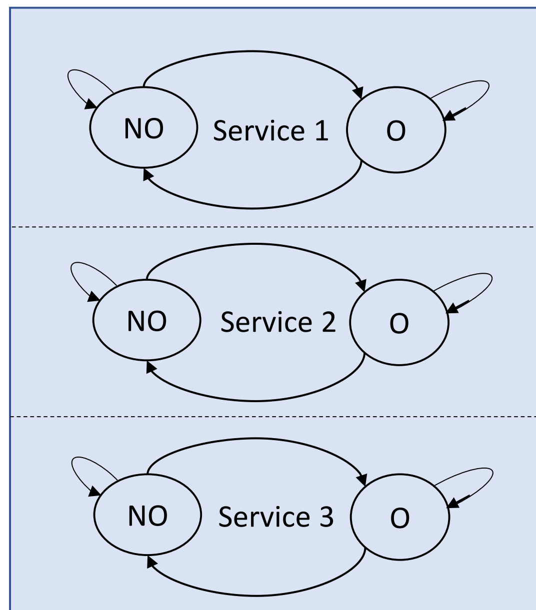


Figure 6.2: Concurrency of service statemachine models

falsifying example obtained during the test is shown in figure 6.1. The Services 1 and 3 were initially available with the corresponding *StartOfferService* methods. Later, when the Service 3 was stopped with a *StopOfferService*, Service 1 became unavailable along with Service 3 which led to an assertion error. After investigating the error that Hypothesis reported continuously during the test runs, it was found that the Service 1 and 3 used the same TCP/IP ports for communication which led to this failure. This mismatch in the configuration of applications were reported to the developers and rectified.

This shows that our approach not only detect errors but also outputs an easier minimal example for the errors that it detects which helps us to debug them easier.

6.1.2 Concurrency

The services - Service 1, Service 2 and Service 3 are offered and requested independently of one another. This implies that each service is a concurrently occurring state model as in Figure 6.2. In a test run of 100 examples (default number of examples as set by the Hypothesis Framework), all the possible combinations of the states (Not Offered, Offered) have been achieved concurrently during the test run. From the results in Table 6.1 showing the tested combinations, it is evident that the services are concurrent and therefore independent from each other.

With PBT, we can scale up to a large number of concurrently running services whose combinations of all the possible states are otherwise time-consuming to be written manually for testing.

Table 6.1: Combinations of states of services from a test run of 100 examples.

Service 1	Service 2	Service 3	Verified
Not Offered	Not Offered	Not Offered	✓
Not Offered	Not Offered	Offered	✓
Not Offered	Offered	Not Offered	✓
Not Offered	Offered	Offered	✓
Offered	Not Offered	Not Offered	✓
Offered	Not Offered	Offered	✓
Offered	Offered	Not Offered	✓
Offered	Offered	Offered	✓

6.1.3 Statistics of Test Inputs

In this section, we analyse the number of commands - *StartOffer()*, *StopOffer()* and *Check()* generated by Hypothesis in a set of 100 examples as depicted in Figure 6.3. During the run, the step count of the examples is set to an average of 50. Step count is the number of steps in each example during the run [hypothesis site]. A higher step count is important so that sufficient wait times are attained in the test examples so that we receive the asynchronous response and *Check()* can be made. From Figure 6.4 which shows an example generated by Hypothesis, *StartOfferService()* command for service 1 is generated but there is no *Check()* as there is no waiting made for the asynchronous response. Although the statistics look promising, the large number of randomly generated

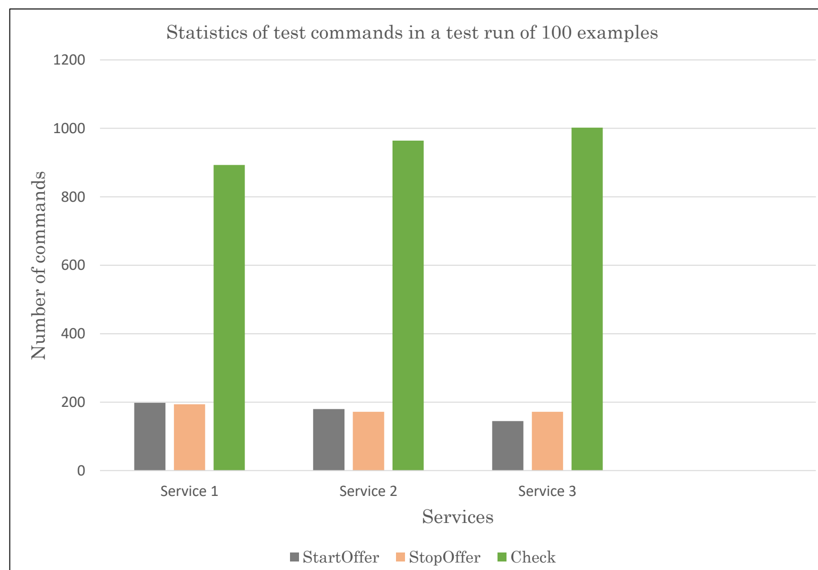


Figure 6.3: Number of test inputs generated by Hypothesis in 100 examples with step count=50

commands should not give a false feeling of security after testing when the step count is low and such small examples can exist without a *Check()*. For this reason, an analysis of the randomly generated test inputs is needed to clarify whether these satisfy our testing requirements.

```
state = test_servicediscovery()
state.initialize_ECU()
state.StartOfferService(service=1)
state.teardown()
```

Figure 6.4: An example generated by Hypothesis with an interface call without a *Check()*

Finally, we checked how many times Hypothesis checks the availability of a service after its next state changes from its previous one. In other words, we counted the statistics for the verification made when a service transitioned from “Not Offered" to “Offered" state and vice-versa. This count was found to be 52, 42 and 48 times for Services 1, 2 and 3 respectively in a test run of 100 examples generated by Hypothesis. This count ensures that we performed checks each time when the services changed their states and not when the services were in a same state (NO or O) throughout the test run.

6.1.4 Analysis of Test inputs

In this section, we analyse whether the test inputs generated by Hypothesis in a run of 100 examples cover some of the test cases derived from the AUTOSAR system test specification [11] and other test cases which we have in mind. The four test cases considered are as follows:

- T1 - A service which is not available can be offered.
- T2 - A service which is available can be stopped.
- T3 - Calling *StartOffer()* on an already offered service has no impact.
- T4 - Calling *StopOffer()* on an already stopped service has no impact.

Table 6.2: Statistics of test cases covered for all services in a run of 100 examples.

Test cases	Transitions covered	Service 1	Service 2	Service 3
T1	NO/NC→NO/C→O/NC→O/C	9	11	7
T2	O/NC→O/C→NO/NC →NO/C	5	8	3
T3	O/NC→O/C→O/NC→O/C	4	6	10
T4	NO/NC→NO/C→NO/NC→NO/C	2	8	7

```
Falsifying example: run_state_machine(factory=test_servicediscovery,
                                      data=data(...))
state = test_servicediscovery()
state.initialize_ECU()
state.StartOfferService(service=1)
state.wait(t=0.4)
state.teardown()
```

Figure 6.5: Minimal example for an injected fault into *StartOfferService()*.

The number of hits corresponding to these test cases along with the transitions covered in a test run of 100 examples generated by Hypothesis are given in table 6.2. All the test cases which we have considered, have been covered during the test run which shows that our approach is effective in testing SD.

Analysing the test inputs which are randomly generated is essential to understand the quality of these tests and the property as well. This step would not then give us a false sense of security after we generate thousands of test cases randomly using the PBT framework.

6.1.5 Fault Injection

In order to confirm whether the shrinking feature of Hypothesis can produce the right falsifying example, we injected a fault so that the *StartOfferService()* for service 1 is not communicated to the SUT. As a result, the check fails during assertion of availability of Service1. The error due to the injected fault is detected by Hypothesis after generating a total of 42 examples including the ones that were shrunk. Our PBT framework has rightly minimized the failing examples to the one listed in Figure 6.5. In the falsifying example, Hypothesis has made a *StartOfferService()* call, waited for 400ms so that the check could be successfully made before throwing an **Assertion Error**. The test data with 42 examples took 133.09s to run until it stopped to give the falsifying example.

6.1.6 Coverage

We apply black-box testing methods like PBT when the source code of the SUT is not available. As such, code coverage cannot be used as a criterion to measure the effectiveness of testing. In white-box testing techniques, test cases are formulated with the goal of executing all the statements, functions and branches of the code.

In state-transition testing which is another black-box testing method [64, 65], state and transition coverages are used as coverage criteria. In this manner, it is ensured during testing that all the critical points in the state-space of SUT are covered [66]. As we deal with a stateful system in our work, it makes sense to verify whether we have exercised all the transitions and states of our stateful system.

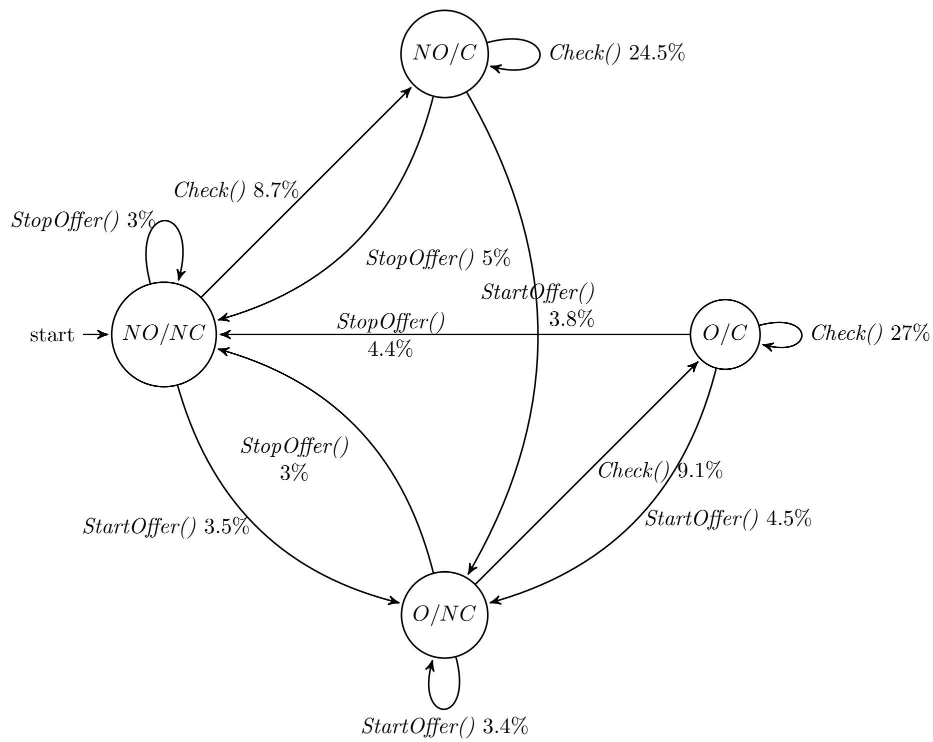


Figure 6.6: Transition Coverage of Service 1 in a test run of 100 examples

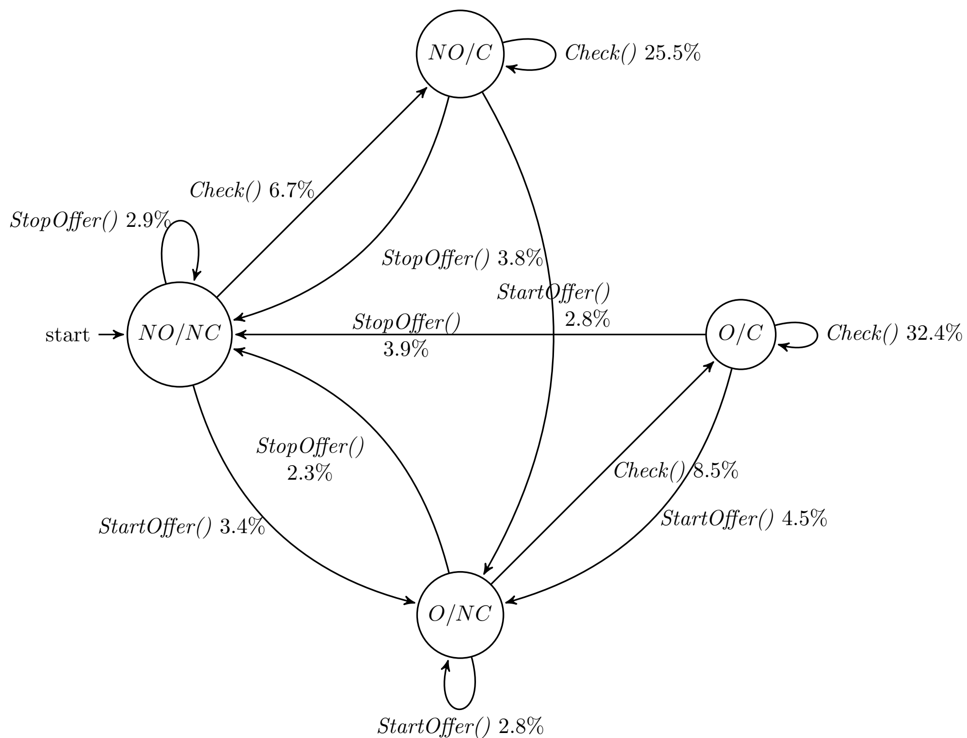


Figure 6.7: Transition Coverage of Service 2 in a test run of 100 examples

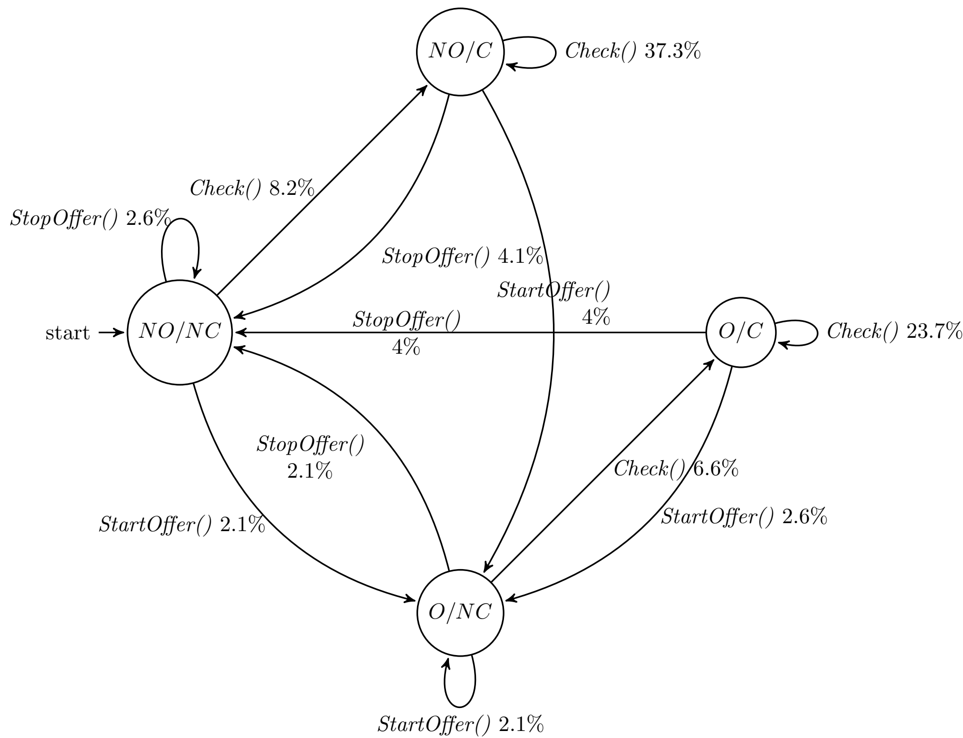


Figure 6.8: Transition Coverage of Service 3 in a test run of 100 examples

For this purpose, we have measured all transition coverages in a test run of 100 examples for Service 1, Service 2 and Service 3 in Figures 6.6, 6.7, and 6.8 respectively. We collected the statistics by analysing the percentage of each transition to the total number of all transitions. The transition coverage diagrams shows that all the states and transitions are covered atleast once during testing. *Check()* transitions are the most frequent transitions, accounting for atleast 30% when the services are in "Not Offered" and "Offered" states. Analysing the state-space coverage help us to understand the degree of risk remaining after testing [66]. Verifying with *Check()* whether the services are available when in "Offered" state and not available when in "Not Offered" state reduces the risk substantially after testing. Our results show that we have covered all the states and transitions for all the services atleast once which are the requirements for such coverage criteria.

6.2 Efficiency

Efficiency refers to the practicality of a testing approach. According to Eldh et al., efficiency should be calculated considering the time and effort spent for test creation along with the aspect of test execution [63]. In this regard, effort involved in writing each test case is high in manual testing whereas in the case of PBT, it is higher initially while defining the properties. From table 6.3, 30% of the test execution time is attributed to the waiting time due to the asynchronous behaviour of our software. This communication delay is involved in any black-box testing method that we would perform at system level as

Table 6.3: Execution and wait times in the test runs of SD.

Number of examples	Total execution time (s)	Wait time (s)
5	44.96	16.59
10	112.07	47.89
100	798.92	276.69
500	4054.03	1494.42

long as communication is asynchronous. The waiting time for our PBT test run increases by a factor of 8 as the number of examples increases.

6.3 Interview Study

We conducted an interview study with 6 participants who are experts in testing field and collected their feedback and opinions about our testing approach. We chose semi-structured, open-ended, individual interviews with experts in testing field [67]. This format allowed us to get a broad overview over their opinion about PBT. We further investigated the factors for comparison of our testing method with traditional methods of testing from the answers of the interviewees. The interview guide was designed based on the research questions formulated earlier.

The participants were employees who have atleast 4 years of experience in an automotive product-based organization with more than 2000 employees. The length of each interview ranged from 45 to 60 minutes and the interviews were recorded and transcribed for analysis.

Table 6.4: Participant data.

Participant ID	Area of responsibility	Experience in years
P1	Test Infrastructure	>35
P2	Product test suite	>4
P3	Product test suite	>14
P4	Component test	>12
P5	Component test	>12

P1 argued that verifying the concurrency of services is an important factor to evaluate the combined test cases generated by our approach more than the state and transtion coverages for each service considered individually. P1 reported that test stimuli can be generated arbitrarily using our approach compared to traditional testing method where only a given functionality is tested with each test case. P1 talked about the possibility

of having extensive test cases generated using our approach although the timing factor of asynchronous software could reduce the efficiency at the system level. According to P1, although defining properties would be a tedious task, this step would help understand the system better.

P2 had the opinion that a large number of tests can be generated easily with our approach and the random factor seems useful in identifying issues that one might not have thought of with static test cases. Nevertheless, P2 claimed that testers know the possibility of types of bugs that could exist in certain test scenarios from their experience. As such, test cases do not always demand for random factors present in our approach. P2 suggested a method whereby we could analyse the statistics of the test cases generated using PBT approach. According to P2, such an analysis could confirm whether the test cases in our mind have been covered with PBT. A tool development based on this approach mentioned by P2 is discussed in the previous work in section 8. We have performed the analysis mentioned by P2 for our test cases in section 6.1.4. Additionally, P2 reported that the test coverage criterion in our case would be a mathematical problem to find out whether all the possible combinations of test inputs are covered using our approach. P2 made a remark that this could make PBT a better fit in the area of product testing where easy generation of more number of tests could be beneficial. P2 also claimed that a greater amount testing is performed at the component level with the aim to fix as many bugs as possible, than an the product level.

P3 reported that it is a difficult task to say when we have sufficiently tested, although state and transition coverages are important metrics like we used in our approach. P3 suggested that requirements should be the primary specifications for formulating properties and pointed out that the first application of PBT requires good knowledge of the system. P3 also mentioned that our approach could be a good extension to conventional methods of testing to find out how the systems respond to random scenarios other than well defined ones.

P4 and P5 are experienced in development and testing at the unit or component level. They described that they have used testing methods very similar to our approach to generate various combinations of datatypes for functions. P4 argued that after the V-model processes of requirements, design and unit tests, there would be a slight chance of finding an error or bug even with our approach.

P4 and P5 stressed the factor of randomness in our approach that “why should testers choose random testing methods over systematic ones?". They would have welcomed our approach if there was a systematic method of defining properties. P4 mentioned that the shrinking feature of PBT tool is exciting but claimed that finding an error or a bug would be a mere coincidence. P4 and P5 suggested a possibility of generating parameters with PBT tool which can be exported and used as inputs in existing static unit test cases.

P5 asked whether test cases like T3 and T4 which we discussed in section 6.1.4 have been covered using our approach. An analysis of such test cases through which the existence of known issues or bugs could be verified, helps us judge the quality of test inputs generated by our approach. P5 mentioned that it would be an extra task to model the SUT like

in our approach to get the expected results when we do not have access to the source code in black-box testing methods. Additionally, P5 reported that traceability of the test cases to the requirements is important for the customers. In our case, this would mean that we should ensure that the test inputs which are generated randomly in our approach correspond to the requirements.

None of the interviewees had the opinion that our approach has the potential to replace the conventional methods of testing considering that the latter is more systematic and not random. P4 and P5 added that our approach could be used in stress testing at the product testing level.

From the interview study, we conclude that developing a systematic approach to testing systems with PBT could convince test experts about its applicability to a great extent. Inspite of the randomness involved in test methods like PBT, a qualitative analysis of the test inputs generated with these approaches can help build reliability.

To summarize, we identified an issue in configuration of ports with our PBT approach which shows the effectiveness in testing the functionality of SD. Although, time is needed in the initial stage of understanding the SUT, the automatic generation along with the shrinking features of PBT add up to efficiency of our test approach. Additionally, it is beneficial to analyse the test inputs which are generated randomly so that we can judge the quality of the properties that we define.

7 Threats to Validity

Here, we report the potential threats to the validity of our work. We have carried out an experiment for the functionality of Service Discovery of Communication Management module of the Adaptive Platform. The results presented in this paper are not applicable to the other modules of the Adaptive Platform. The boundary value of time which we obtained as 300ms was the result of around 10 repeated test runs on Hypothesis and this could vary depending on the software implementation and runtime environment. For the results, we have considered only a few test cases derived from the test specification to check whether these are covered by our approach. This could be analysed for a wide variety of test cases that we would otherwise perform system testing with. Regarding the interviews conducted, it is possible that the experts interpreted the questions in a different way than what was intended by the interviewer. The goal of this interview study was to obtain an insight into the test experts' opinion and viewpoints about our approach. Our study involving 5 participants could limit the generalizability of the results obtained during our work. For a more thorough research on the results, a larger sample of interviewees could be considered [67].

8 Related Work

In this chapter, we discuss the research work that is already available which is relevant to this thesis. We describe the work that has been previously carried out for testing AUTOSAR with the QuickCheck PBT tool. Additionally, we mention other studies where PBT has been applied for testing software.

AUTOSAR has released a set of requirements or specifications for each of the modules of the Basic Software, which would then be interpreted by the suppliers who develop them [6]. These interpretations may not necessarily and precisely correspond to the specifications released by AUTOSAR, as the latter tend to be more abstract. This may also be due to the implementation freedom granted to the ECU vendors that the AUTOSAR specification remains ambiguous [68].

Svensson et al. have discovered 227 defects in ECU vendor's Basic software code through a research on random testing of 20 modules of Classic Platform, of which 180 defects were due to ambiguities and could be resolved by revisiting the AUTOSAR standard. Moreover, it is interesting to note that they have also found a significant error in the AUTOSAR standard specification itself. The specification stated that a function returns a pointer whereas in the requirement document it was given that it copies a value to a memory location, which has led to the conflict [7].

Hughes emphasize on the importance of testing as well as the fact that writing thousands of test cases manually can be time-consuming and cumbersome. Hence, the author proposes the idea "Don't write tests, generate them" as an alternative solution [33].

With property-based testing using the QuickCheck tool we derive properties from formal specifications which would lead to a better understanding of our project. This even helps to point out errors, if there are any, in the specification [10]. Moreover, corner cases could be figured out which may not otherwise be found by writing manual test cases [29].

Arts et al. have proved the efficiency of PBT on QuickCheck models in finding out software defects in already well-tested software implementations of CAN modules. Additionally, these models could also be represented by fewer lines of code as compared to the test cases written in TTCN (Testing and Test Control Notation), which were defined by AUTOSAR [7]. A large number of test sequences when randomly generated can help the testers identify the areas not noted before. Additionally, the process of shrinking the failing test input to a minimal one by the property-based testing tool makes it more comprehensible enabling us to rework on the errors in lesser time and fewer efforts [7, 14].

According to Raina, property-based testing is a form of black-box testing where functionality is tested using the provision of inputs followed by evaluation of outputs. Here, the

implementation of the functionality isn't given a high importance and hence need not be known to the testers [14].

Aichernig et al. discuss another PBT Tool FsCheck for web services by deriving properties from XML based business specifications [32]. The benefit of generating more random test cases with a PBT tool PropEr for an embedded application is known, where a set of 100 test cases pointed out a failure whereas a set of 10 test cases gave out none [14]. This fact clearly states the advantage of PBT wherein a large number of test cases are automatically generated to falsify a property [69] as compared to manually written test cases, where such a huge number of test cases rely upon the testers' patience, effort and time.

Schumi et al. argue that it is also important to confirm that the System Under Test (SUT) corresponds to the specification, as programmers can later change the implementation code without changing the business specification [32].

Arts discusses that Quviq QuickCheck has been used to test Project Fifo which is a cloud management system [31] which gave rise to timing error due to an asynchronous API call. This was also fixed by adding a wait before triggering the next operation after the asynchronous operation. This timing aspect is similar to our work on stateless property where we find the minimum waiting time required after an asynchronous software operation.

Gerdes et al. have used to derive minimal examples from formal specifications of software using QuickCheck so that the users can understand the behaviour of APIs better. They conducted an experiment with students to evaluate these examples which showed that they could understand the programs better. They argue that random examples are meaningless considering their randomness but also admit the fact that this characteristic gives rise to finding unexpected corner cases [70].

It is possible to integrate external generators for test cases with a PBT tool so that the strategies of both can be combined [13]. It can give a better coverage of the test system than with a single PBT tool although the computation times could be higher for the test generation by the external generators.

Although PBT tool can easily generate thousands of test inputs within a short time, analysing whether these inputs include a test case which we have in mind would be difficult. Hughes et al. have developed a tool which take a property and a unit test as inputs to check whether the latter could be generated by the defined property. They performed a case study with six unit tests derived from acceptance test suite of AUTOSAR and their corresponding QuickCheck models based on CAN stack. The results showed that five of these unit tests could be generated by the property models with an error in the last unit test itself. Such method or tools can help the testers to judge the property's quality in PBT [28].

Generality of an approach refers to the ability of any testing approach to handle diverse language constructs and its use in other domains [67]. Considering the plethora of work done previously for PBT, we find that the aspect of generality is good for our test approach.

9 Conclusion

In this master thesis, we proposed a black-box testing methodology based on PBT and evaluated its applicability for AUTOSAR basic software. These findings are to some extent transferable to other asynchronous embedded software.

Embedded systems comprise of state machines to a large extent and manually implementing automated test cases for all the possible combinations of the states and transitions is cumbersome. In this regard, stateful properties have the potential to generate a large number of test inputs for independent and parallel instances of the same statemachine. Our test infrastructure is based on stateful properties for the functionality of Service Discovery with the application of stateless properties to determine the timing characteristic of asynchronous software which answers RQ1.

In order to formulate a test adequacy criterion (RQ2), determining whether the generated test cases cover all the states and transitions atleast once would be beneficial. Analysing whether these include the test inputs which would otherwise be used to perform static test cases helps judge the property's quality. It has been inferred though from the interview study that the test adequacy for customers means that the individual test cases trace back to their requirements. In that case, with our random approach, it would be an extra task to analyse the large number of test cases generated at random and link them to the corresponding software requirements.

For PBT, the tremendous effort lies in abstracting the SUT into properties which demands for system knowledge from the testing engineers. Our results show that PBT is effective in testing Service Discovery and our approach can be scaled up easily to a higher number of services. The results of our interview study conducted shows that the experts rely on traditional methods of testing much more strongly than random testing methods like PBT (RQ3). They find our approach relatively new with no prior experiences and are therefore unable to assess its applicability. The "randomness" factor of PBT obtained from the interview study coincides with the previous work done by Gerdes et al. [70], although it can be beneficial in finding corner cases. Moreover, our approach demands for new skills from the testing engineers for defining properties. Although PBT can aid in automating test designs and test generation in comparison to manual test designs, it can potentially give rise to acceptance problems among the testers for these reasons.

To summarize, PBT can be an ideal candidate for black-box testing of complex systems. This approach can be used in addition to the traditional testing methods to reveal any corner cases. Analysing the property's quality using the generated test cases can build trust among the testers to use this method to test their systems.

10 Future Scope

Our approach is effective in testing the functionality of Service Discovery of Communication Management module of the Adaptive Platform. However, our work is just an elementary research on the application of PBT in testing software components based on the AUTOSAR standard. The developed test system can be extended to other modules of the Adaptive Platform exploring the possibilities of application of stateless and stateful properties. As pointed out during the interview study conducted during this work, research can be carried out to check whether PBT can generate parameters which can serve as inputs to static test cases. Another future work could be developing a systematic process to assist testers in defining properties for their software.

Bibliography

[1] Manfred Broy, Ingolf H Kruger, Alexander Pretschner, and Christian Salzmann. "Engineering automotive software". In: *Proceedings of the IEEE* 95.2 (2007), pp. 356–373.

[2] Lucia Lo Bello, Riccardo Mariani, Saad Mubeen, and Sergio Saponara. "Recent advances and trends in on-board embedded and networked automotive systems". In: *IEEE Transactions on Industrial Informatics* 15.2 (2018), pp. 1038–1051.

[3] Alireza Haghighatkhah, Markku Oivo, Ahmad Banijamali, and Pasi Kuvaja. "Improving the state of automotive software engineering". In: *IEEE Software* 34.5 (2017), pp. 82–86.

[4] Christof Ebert and John Favaro. "Automotive software". In: *IEEE Software* 3 (2017), pp. 33–39.

[5] David McCandless, Pearl Doughty-White, and Miriam Quick. *Codebases Millions of lines of code*. https://informationisbeautiful.net/visualizations/million-lines-of-code/. Accessed: 2019-12-04. 2015.

[6] *AUTOSAR Consortium:Automotive Open System Architecture, standard documents*. https://autosar.org/. Accessed: 2019-12-04.

[7] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. "Testing AUTOSAR software with QuickCheck". In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2015, pp. 1–4.

[8] Tischer Mirko. *The Computing Center in the Vehicle AUTOSAR Adaptive*. https://assets.vector.com/cms/content/know-how/_technical-articles/AUTOSAR/AUTOSAR_Adaptive_ElektronikAutomotive_201809_PressArticle_EN.pdf. Accessed: 2019-12-04.

[9] *Explanation of Adaptive Platform Design, 2017, AUTOSAR Consortium, standard documents*. https://autosar.org/. Accessed: 2019-12-04.

[10] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Acm sigplan notices* 46.4 (2011), pp. 53–64.

[11] *System Tests of Adaptive Platform*. https://www.autosar.org/standards/adaptive-platform/adaptive-platform-1903/. Accessed: 2019-12-04. 2019.

[12] *Requirements on Communication Management*. https://www.autosar.org/standards/adaptive-platform/adaptive-platform-1903/. Accessed: 2019-12-04. 2019.

[13] Bernhard K Aichernig, Silvio Marcovic, and Richard Schumi. "Property-based testing with external test-case generators". In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2017, pp. 337–346.

[14] Shivani Raina. *QuickCheck-Style Testing of Embedded Software using the PropEr Framework*. 2012.

[15] Percy Pari Salas and Padmanabhan Krishnan. "Automated software testing of asynchronous systems". In: *Electronic notes in theoretical computer science* 253.2 (2009), pp. 3–19.

[16] Mostafa Massoud. "Evaluation of an Adaptive AUTOSAR System in Context of Functional Safety Environments". In: (2017).

[17] Marcin Bajer, Marek Szlagor, and Marek Wrzesniak. "Embedded software testing in research environment. A practical guide for non-experts". In: *2015 4th Mediterranean Conference on Embedded Computing (MECO)*. IEEE. 2015, pp. 100–105.

[18] Sangeetha Yalamanchili and K Sitha Kumari. "Comparison of manual and automatic testing using genetic algorithm for information handling system". In: *2016 International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*. IEEE. 2016, pp. 1795–1799.

[19] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. "Software testing techniques: A literature review". In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. IEEE. 2016, pp. 177–182.

[20] Bertrand Meyer. "Seven principles of software testing". In: *Computer* 41.8 (2008), pp. 99–101.

[21] Abel Marrero Perez and Stefan Kaiser. "Integrating test levels for embedded systems". In: *2009 Testing: Academic and Industrial Conference-Practice and Research Techniques*. IEEE. 2009, pp. 184–193.

[22] Oskar Ingemarsson and Sebastian Weddmark Olsson. "Evaluation of validity of verification methods: Automating functional safety with QuickCheck". MA thesis. 2015.

[23] Ina Schieferdecker. "Model-based testing". In: *IEEE software* 1 (2012), pp. 14–18.

[24] Ali Mili and Fairouz Tchier. *Software testing: Concepts and operations*. John Wiley & Sons, 2015.

[25] "ISO/IEC/IEEE International Standard - Software and systems engineering – Software testing –Part 2:Test processes". In: *ISO/IEC/IEEE 29119-2:2013(E)* (2013), pp. 1–68. ISSN: null. DOI: 10.1109/IEEESTD.2013.6588543.

[26] Hanmeet Kaur Brar and Puneet Jai Kaur. "Differentiating integration testing and unit testing". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. IEEE. 2015, pp. 796–798.

[27] Joel Martin and David Levine. "Property-based testing of browser rendering engines with a consensus oracle". In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE. 2018, pp. 424–429.

[28] Alex Gerdes, John Hughes, Nick Smallbone, and Meng Wang. "Linking unit tests and properties". In: *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*. 2015, pp. 19–26.

[29] Yusuke Wada and Shigeru Kusakabe. "Performance evaluation of a testing framework using QuickCheck and Hadoop". In: *Information and Media Technologies* 7.2 (2012), pp. 694–700.

[30] André Santos, Alcino Cunha, and Nuno Macedo. "Property-based testing for the robot operating system". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2018, pp. 56–62.

[31] George Fink and Matt Bishop. "Property-based testing: a new approach to testing for assurance". In: *ACM SIGSOFT Software Engineering Notes* 22.4 (1997), pp. 74–80.

[32] Bernhard K Aichernig and Richard Schumi. "Property-based testing with FsCheck by deriving properties from business rule models". In: *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2016, pp. 219–228.

[33] John Hughes. "Experiences with QuickCheck: testing the hard stuff and staying sane". In: *A List of Successes That Can Change the World*. Springer, 2016, pp. 169–186.

[34] *Getting Started with Hypothesis*. https://hypothesis.works/articles/getting-started-with-hypothesis/. Accessed: 2020-02-05.

[35] *Pytest Framework*. https://docs.pytest.org/en/latest/. Accessed: 2020-02-10.

[36] *Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do*. Pragmatic Bookshelf, 2019.

[37] *ISTQB Glossary*. https://glossary.istqb.org/en/search/. Accessed: 2020-10-07.

[38] H Fennel, L Lundh, J Leflour, JL Maté, and K Nishikawa. "AUTOSAR–Challenges and Achievements 2005". In: ().

[39] Dr. Simon Frohn and Fabian Rees. *From Signal to Service*. https://assets.vector.com/cms/content/know-how/_technical-articles/Ethernet_AUTOSAR_Adaptive_Elektronik_Automotive_201803_PressArticle_EN.pdf. Accessed: 2020-01-07.

[40] Dr. Markus Oertel and Dr. Bastian Zimmer. *E/E Architectures with AUTOSAR Adaptive*. https://assets.vector.com/cms/content/know-how/_technical-articles/AUTOSAR/AUTOSAR_Adaptive_Architecture_ATZ_201905_PressArticle_EN.pdf. Accessed: 2020-01-07.

[41] Lorenz Slansky and AUTOSAR Chairman. *AUTOSAR for Intelligent Vehicles*.

[42] Simon Fürst and Markus Bechter. "AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE. 2016, pp. 215–217.

[43] Rahamatullah Khondoker, Bernd Reuther, Dennis Schwerdel, Abbas Siddiqui, and Paul Müller. "Describing and selecting communication services in a service oriented network architecture". In: *2010 ITU-T Kaleidoscope: Beyond the Internet?-Innovations for Future Networks and Services*. IEEE. 2010, pp. 1–8.

[44] *A Service Oriented Architecture for Ambient IntelligenceChoreography and Secure Service Discovery*. https://www.academia.edu/27381202/A_Service_Oriented_Architecture_for_Ambient_Intelligence_Choreography_and_Secure_Service_Discovery. Accessed: 2020-01-16.

[45] Gaurav Singh, Narayan Kamath, and RK Sharma. "Implementing Adaptive AUTOSAR Diagnostic Manager with Classic Diagnostics as APIs". In: *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE. 2018, pp. 894–898.

[46] Jelena Jovičić, Mila Kotur, Milan Z Bjelica, and Ištvan Papp. "Visualizing Functional Verification in Adaptive AUTOSAR". In: *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE. 2018, pp. 1–4.

[47] Mia Stepanović, Jelena Jovičić, Goran Stupar, and Marko Kovačević. "Application lifecycle management in automotive: Adaptive AUTOSAR example". In: *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE. 2018, pp. 1–4.

[48] Johannes Dorfner. *AUTOSAR – Part 2: Adaptive Platform*. https://www.methodpark.de/blog/autosar-part-2-adaptive-platform/. Accessed: 2020-01-07.

[49] Dr.-Ing. Thomas Scharnhorst. *AUTOSAR proofs to be THE automotive software platform for intelligent mobility*. https://www.autosar.org/fileadmin/user_upload/2017_ELIV_AUTOSAR_proofs_to_be_THE_automotive_software_platform_for_intelligent_mobility.pdf. Accessed: 2020-01-07.

[50] Holger Blasum and Sergey Tverdyshev. "Classic and Adaptive AUTOSAR in MILS terms." In: *MILS@ DSN*. 2018.

[51] *Explanation of ara::com API*. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_EXP_ARAComAPI.pdf. Accessed: 2020-01-20.

[52] *Specification of Communication Management*. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_SWS_CommunicationManagement.pdf. Accessed: 2020-01-20.

[53] *SOME/IP Protocol Specification*. https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-0/AUTOSAR_PRS_SOMEIPProtocol.pdf. Accessed: 2020-01-08.

[54] Abhijith Ajith Kumar. “Analysis and Implementation of Various Interoperability Techniques for ADAS Development”. Technische University of Chemnitz. MA thesis. 2018.

[55] *Specification of Service Discovery*. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_ServiceDiscovery.pdf. Accessed: 2020-01-22.

[56] Jan R Seyler, Thilo Streichert, Michael Glaß, Nicolas Navet, and Jürgen Teich. “Formal analysis of the startup delay of SOME/IP service discovery”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium. 2015, pp. 49–54.

[57] Jochen Kreissl. “Absicherung der SOME/IP Kommunikation bei Adaptive AUTOSAR”. MA thesis. 2017.

[58] *Example for a Serialization Protocol (SOME/IP)*. http://some-ip.com/papers/cache/AUTOSAR_TR_SomeIpExample_4.2.1.pdf. Accessed: 2020-01-20.

[59] *Requirements on Communication Management*. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-03/AUTOSAR_RS_CommunicationManagement.pdf. Accessed: 2020-01-22.

[60] *SOME/IP Service Discovery Protocol Specification*. https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-2/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf. Accessed: 2020-01-22.

[61] *R-Car*. https://www.renesas.com/us/en/products/automotive/automotive-lsis/r-car.html. Accessed: 2020-02-09.

[62] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. *Automated system testing of real-time embedded systems based on environment models*. Tech. rep. Technical Report 2011-19, Simula Research Laboratory, 2011.

[63] Sigrid Eldh, Hans Hansson, Sasikumar Punnekkat, Anders Pettersson, and Daniel Sundmark. “A framework for comparing efficiency, effectiveness and applicability of software testing techniques”. In: *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*. IEEE. 2006, pp. 159–170.

[64] Cassia de Souza Carvalho and Tatsuhiro Tsuchiya. “Coverage criteria for state transition testing and model checker-based test case generation”. In: *2014 Second International Symposium on Computing and Networking*. IEEE. 2014, pp. 596–598.

[65] “ISO/IEC/IEEE International Standard - Software and systems engineering–Software testing–Part 4: Test techniques”. In: *ISO/IEC/IEEE 29119-4:2015* (2015), pp. 1–149. ISSN: null. DOI: 10.1109/IEEESTD.2015.7346375.

[66] Itzel Dominguez Mendoza, D Richard Kuhn, Raghu N Kacker, and Yu Lei. “CCM: A tool for measuring combinatorial coverage of system state space”. In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 291–291.

[67] Dora Dzvonyar and Bernd Bruegge. “Team Composition and Team Factors in Software Engineering: An Interview Study of Project-Based Organizations”. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2018, pp. 561–570.

[68] Thomas Arts and Mohammad Reza Mousavi. “Automatic consequence analysis of automotive standards (AUTO-CAAS)”. In: *Proceedings of the First International Workshop on Automotive Software Architecture*. 2015, pp. 35–38.

[69] Bernhard K Aichernig and Richard Schumi. “Property-based testing of web services by deriving properties from business-rule models”. In: *Software & Systems Modeling* 18.2 (2019), pp. 889–911.

[70] Alex Gerdes, John Hughes, Nicholas Smallbone, Stefan Hanenberg, Sebastian Ivarsson, and Meng Wang. “Understanding formal specifications through good examples”. In: *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*. 2018, pp. 13–24.

[71] Michael W Whalen, Ajitha Rajan, Mats PE Heimdahl, and Steven P Miller. “Coverage metrics for requirements-based testing”. In: *Proceedings of the 2006 international symposium on Software testing and analysis*. 2006, pp. 25–36.

[72] *Testing Asynchronous APIs with QuickCheck*. `http://www.erlang-factory.com/static/upload/media/1461230674757746pbterlangfactorypptx.pdf`. Accessed: 2020-02-05.