Service Testing for the Internet of Things

Eike Steffen Reetz

Faculty of Engineering and Physical Sciences
University of Surrey

This dissertation is submitted for the degree of

Doctor of Philosophy

January 2016
Acknowledgements

Working towards a PhD degree has been one of the most exciting and challenging periods of my life. During this time I have been accompanied by a lot of people who supported me either directly or indirectly.

First, I would like to thank my supervisors at the University of Surrey, Klaus Moessner and Payam Barnaghi, who provided me the opportunity to conduct research in a deeply interesting field. Their encouraging feedback and our inspiring discussions did help to advance my study progress. The same applies to my advisor at the University of Applied Sciences Osnabrück (UASO) Ralf Tönjes. Being the head of the research group for Mobile Communication at UASO – my employer –, I owe him cordial thanks for making one of my dreams come true and supporting my career by offering freedom for my personal research while sharing his valuable experiences openly. I am thankful for his guidance, the trust and delegated responsibilities that I have been provided. A special thank you is directed to Daniel Kümper, Michael Knappmeyer, and Marten Fischer for our productive collaboration and their assistance in every department.

In this context, I am very grateful for the possibility to meet kind colleagues and make new friends all over the world during numerous business trips to conferences and project meetings. The last couple of years have significantly broadened my horizon in a way I had not expected before.

The research group at UASO provided me a very pleasant environment day after day. I thank my colleagues, i.e. Daniel Bretschneider, Daniel Hölker, Nils Hohmann, Günter Hüdepohl, Thorben Iggena, and Frank Nordemann, especially for their help in the review process.

Finally I would like to thank my girlfriend Christin, my parents and my friends for their patience and for believing in me.
Abstract

Services that represent sensor and actuator nodes, together with service orchestration, aid in overcoming the heterogeneous structure of the Internet of Things (IoT). Interconnecting different sensor and actuator nodes and exposing them as services is a complex topic which is even more demanding for testing. Further effort is needed to enable common and efficient methodologies for testing IoT-based services. IoT-based services differ from web services since they usually interact with the physical environment via sensor and actuator nodes. This changes how testing can be performed. An open research question is thereby how to apply Model-Based Testing (MBT) approaches for facilitating scalable and efficient test automation.

This thesis introduces a novel test framework to facilitate functional evaluation of IoT-based services based on MBT methodologies. The concept separates the service logic from connected sensor and actuator nodes in a sandbox environment. Furthermore, a new IoT service behaviour model is designed for representing relevant characteristics of IoT-based services and ensuring the automated emulation of sensor nodes. The IoT-behaviour model proves to be automatically transformable into executable Test Cases (TCs). As a proof of concept, the automated test approach is prototypically implemented as a novel test tool. The execution of the TCs reveals, that crucial failures, such as unexpected messages, data types, or data values, can be detected during test execution.

Deriving tests from a test model typically result in huge number of TCs, which cannot be executed within a reasonable time and with limited resources. To enhance the diversity of executed TCs, similarity investigation algorithms are proposed and validated. The results show that the proposed Diversity-based Steady State Genetic algorithm can outperform existing solutions up to 11.6 % with less computation time. With regard to verifying the failure detection rate, experiments show that the proposed Group Greedy algorithm can enhance the rate up to 29 %.
# Table of contents

<table>
<thead>
<tr>
<th>List of figures</th>
<th>xi</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of listings</td>
<td>xv</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>xvii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Motivation                                     | 1     |
1.2 Research Objectives and Questions              | 3     |
1.3 Research Methodology and Achievements          | 4     |
1.4 Thesis Organisation                            | 7     |

## 2 Background and Related Work

2.1 Internet of Things Based Services               | 9     |
2.1.1 IoT-based Service Concept                    | 10    |
2.1.2 Value of Testing of IoT-based Services       | 11    |
2.1.3 Example IoT-based Services                   | 13    |
2.1.4 Opportunities to Test IoT-based Services     | 14    |
2.2 Classification of Testing Approaches           | 15    |
2.3 Model-based Testing                            | 17    |
2.4 Testing with IoT Resources                     | 20    |
2.5 Test Automation                                | 21    |
2.6 Test Case Selection Optimisation               | 24    |
2.6.1 Test Case Selection through Similarity
                                              Investigations | 25    |
2.6.2 Research Objectives for Test Case Diversity
                                              Optimisation | 28    |
2.7 Summary and Discussion                         | 29    |
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3</strong> Research Scope</td>
<td>31</td>
</tr>
<tr>
<td>3.1 Research Objectives and Challenges</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Hypothesis</td>
<td>34</td>
</tr>
<tr>
<td>3.3 Assumptions and Restrictions</td>
<td>34</td>
</tr>
<tr>
<td>3.4 Scientific Contributions</td>
<td>35</td>
</tr>
<tr>
<td>3.4.1 Publications</td>
<td>35</td>
</tr>
<tr>
<td>3.4.2 Individual and Collaborative Research</td>
<td>37</td>
</tr>
<tr>
<td><strong>4</strong> Framework for Automated IoT-based Service Testing</td>
<td>41</td>
</tr>
<tr>
<td>4.1 Targets and Methodologies</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Test-Driven Lifecycle Management</td>
<td>42</td>
</tr>
<tr>
<td>4.3 Service Composition and Provisioning Architecture</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Overall Test Architecture</td>
<td>46</td>
</tr>
<tr>
<td>4.5 Concepts for Sandbox Testing</td>
<td>46</td>
</tr>
<tr>
<td>4.5.1 Concept for Atomic IoT-based Service Testing</td>
<td>47</td>
</tr>
<tr>
<td>4.5.2 Concept for Composite IoT-based Service Testing</td>
<td>51</td>
</tr>
<tr>
<td>4.6 IoT-based Service Annotation and Model Derivation Opportunities</td>
<td>53</td>
</tr>
<tr>
<td>4.6.1 Model Derivation Opportunities for Atomic IoT-based Services</td>
<td>54</td>
</tr>
<tr>
<td>4.6.2 Testing Opportunities for Ontologies</td>
<td>59</td>
</tr>
<tr>
<td>4.6.3 Composite IoT-based Service Model Derivation Opportunities</td>
<td>61</td>
</tr>
<tr>
<td>4.7 Test Process Concept</td>
<td>62</td>
</tr>
<tr>
<td>4.8 Summary and Discussion</td>
<td>69</td>
</tr>
<tr>
<td><strong>5</strong> Test and Emulation Automation of AuST</td>
<td>71</td>
</tr>
<tr>
<td>5.1 Targets and Methodologies</td>
<td>71</td>
</tr>
<tr>
<td>5.2 Test Design Engine</td>
<td>72</td>
</tr>
<tr>
<td>5.3 Test Execution Engine</td>
<td>83</td>
</tr>
<tr>
<td>5.4 Run-Time Sandbox Environment</td>
<td>84</td>
</tr>
<tr>
<td>5.5 Summary and Discussion</td>
<td>85</td>
</tr>
<tr>
<td><strong>6</strong> Scalability through Test Case Diversity Optimisation of AuST</td>
<td>87</td>
</tr>
<tr>
<td>6.1 Objectives and Methodologies</td>
<td>87</td>
</tr>
<tr>
<td>6.2 Similarity Score Computation</td>
<td>88</td>
</tr>
<tr>
<td>6.2.1 Hamming Similarity Scoring</td>
<td>88</td>
</tr>
<tr>
<td>6.2.2 Hamming Similarity Scoring with Data Value integration</td>
<td>90</td>
</tr>
<tr>
<td>6.2.3 Levenshtein</td>
<td>93</td>
</tr>
<tr>
<td>6.3 Test Case Group Selection Optimisation</td>
<td>94</td>
</tr>
</tbody>
</table>
List of figures

2.1 Layers of the IoT ............................................ 9
2.2 IoT-based Service Concept .................................. 11
2.3 Model-Based Testing Concept ................................. 17
2.4 Test Cases Selection: Steps and Methodologies ........... 26

4.1 Test-Driven Lifecycle Management ............................ 44
4.2 IoT.test Architecture ......................................... 45
4.3 Known and Unknown Interfaces to Test Atomic Services .... 48
4.4 Atomic Service Adaptation for IoT Resource Emulation .... 49
4.5 Simplified Test Execution Engine ............................. 50
4.6 Composite Service Testing in a Sandbox Environment ....... 53
4.7 State Diagram .................................................. 59
4.8 Test Description Ontology ...................................... 60
4.9 Test Design Engine with External Interfaces and Main Components 62
4.10 State Behaviour Segment of Example Service One ........ 67

5.1 Main Model Objects for Logical Behaviour Description .... 73
5.2 Main Model Objects for Test Execution ....................... 74
5.3 AuST Editor to Visualise and Create IoT Behaviour Model (IoTBM) .. 76
5.4 Transformation Visualisation of IoTBM Paths ................. 80
5.5 Transformation Visualisation of MessageModelTemplate to Velocity ... 82
5.6 Test Execution Engine with External Interfaces and Main Components .. 83
5.7 Test Execution Visualisation .................................... 83
5.8 Runtime Sandbox Environment with External Interfaces and Main Components 85

6.1 Hamming Similarity Scoring Algorithm ....................... 89
6.2 Data Integration into Hamming Similarity Scoring Algorithm .... 91
6.3 Levenshtein Similarity Scoring Algorithm ...................... 93
6.4 Greedy Similarity Scoring ...................................... 95
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>Proposed Group Greedy Algorithm.</td>
<td>95</td>
</tr>
<tr>
<td>6.6</td>
<td>Random Search Test Case Group Selection.</td>
<td>97</td>
</tr>
<tr>
<td>6.7</td>
<td>Group Hill Climbing Algorithm.</td>
<td>98</td>
</tr>
<tr>
<td>6.8</td>
<td>Identification of the Next Generation in the Genetic Algorithm.</td>
<td>99</td>
</tr>
<tr>
<td>6.9</td>
<td>Generation of Childrens for the Genetic Algorithm.</td>
<td>100</td>
</tr>
<tr>
<td>6.10</td>
<td>Computation Time Complexity I</td>
<td>102</td>
</tr>
<tr>
<td>6.11</td>
<td>Computation Time Complexity II</td>
<td>103</td>
</tr>
<tr>
<td>7.1</td>
<td>Complete Initialisation Sequence for Camera Service.</td>
<td>109</td>
</tr>
<tr>
<td>7.2</td>
<td>IoT-Behaviour Model of Camera Service.</td>
<td>110</td>
</tr>
<tr>
<td>7.3</td>
<td>Test Execution With TTworkbench.</td>
<td>114</td>
</tr>
<tr>
<td>7.4</td>
<td>IoT Behaviour Model for Simple Composite Service.</td>
<td>115</td>
</tr>
<tr>
<td>7.5</td>
<td>IoT Behaviour Model for Complex Composite Service.</td>
<td>121</td>
</tr>
<tr>
<td>7.6</td>
<td>Permutation Table for Generic Services.</td>
<td>122</td>
</tr>
<tr>
<td>7.7</td>
<td>Example IoT-Behaviour Model.</td>
<td>124</td>
</tr>
<tr>
<td>7.8</td>
<td>Empiric Cumulative Density Function of the Similarity Scoring Algorithms.</td>
<td>125</td>
</tr>
<tr>
<td>7.9</td>
<td>Average Similarity Score with Random TCs Selection.</td>
<td>127</td>
</tr>
<tr>
<td>7.10</td>
<td>Average Similarity Score with Different Selection Algorithms.</td>
<td>129</td>
</tr>
<tr>
<td>7.11</td>
<td>Similarity Score for Generic Model.</td>
<td>133</td>
</tr>
<tr>
<td>7.12</td>
<td>Similarity Score with Different Selection Algorithms for Generic Model.</td>
<td>134</td>
</tr>
<tr>
<td>7.13</td>
<td>Failure Detection Rate of the Example Service with 132 TCs.</td>
<td>137</td>
</tr>
<tr>
<td>7.14</td>
<td>Failure Detection Rate of a Generic Service with 656 TCs.</td>
<td>137</td>
</tr>
<tr>
<td>7.15</td>
<td>Failure Detection Rate of Generic Service with 656 TCs.</td>
<td>139</td>
</tr>
</tbody>
</table>
List of tables

2.1 Behaviour Model Notations and their Strengths for IoT-Based Service Testing. 19
2.2 Test Automation Approaches. ................................. 23
3.1 Scientific Contributions. ...................................... 38
4.1 Concept Comparison. ......................................... 51
4.2 Ontology Entities. ............................................ 57
4.3 Transition Pre-States and Post-States ....................... 59
4.4 Abstract Transition Description of Example Service One. 67
7.1 TTCN-3 Translation of Model Object InitialState. .......... 111
7.2 TTCN-3 Translation of Model Object Event. ................. 111
7.3 TTCN-3 Translation of Model Object Action. ............... 112
7.4 Translation of Model Object NormalState. ................ 112
7.5 TTCN-3 Translation of Model Object EndState. ............ 113
7.6 Test Execution Time of AS. ................................ 115
7.7 Test Execution Time of Simple CS. ......................... 120
7.8 Test Execution Time of Complex CS. ....................... 121
7.9 Test Execution Time of the Test Process. .................. 123
7.10 Median Computation Time of the different Similarity Scoring Algorithms. 126
7.11 Median Gain between Different Algorithm Combinations and RS100. 130
7.12 Computation Time of Minimisation Algorithms with Hamming Similarity. 131
7.13 Computation Time of Different Minimisation Algorithms with Levenshtein. 132
7.14 Computation Time of Minimisation Algorithms for a Large Generic Model. 135
List of listings

4.1  Rule Description Example. ................................................. 56
7.1  TTCN-3 m_c_sys. ............................................................. 116
7.2  TTCN-3 m_p_HTTP. ......................................................... 117
7.3  TTCN-3 Example Test Case Part I. .................................... 118
7.4  TTCN-3 Example Test Case Part II. ................................. 119
# Nomenclature

## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Architecture Reference Model</td>
</tr>
<tr>
<td>AS</td>
<td>Atomic Service</td>
</tr>
<tr>
<td>AuST</td>
<td>Automated and Scalable Model-Based Testing Tool for IoT-based Services</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Model and Notation</td>
</tr>
<tr>
<td>CCTV</td>
<td>Closed Circuit Television</td>
</tr>
<tr>
<td>CoAP</td>
<td>Constrained Application Protocol</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial-off-the-Shelf</td>
</tr>
<tr>
<td>CPS</td>
<td>Cyber-Physical System</td>
</tr>
<tr>
<td>CS</td>
<td>Composite Service</td>
</tr>
<tr>
<td>ECDF</td>
<td>Empirical Cumulative Density Function</td>
</tr>
<tr>
<td>EFSM</td>
<td>Extended Finite State Machine</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>ETSI</td>
<td>European Telecommunications Standards Institute</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>GGr</td>
<td>Group Greedy</td>
</tr>
<tr>
<td>GHC</td>
<td>Group Hill Climbing</td>
</tr>
<tr>
<td>Gr</td>
<td>Greedy</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technologies</td>
</tr>
<tr>
<td>IOPE</td>
<td>Input, Output, Precondition, Effect</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IoT.est</td>
<td>Internet of Things Environment for Service Creation and Testing</td>
</tr>
<tr>
<td>IoTBM</td>
<td>IoT Behaviour Model</td>
</tr>
<tr>
<td>IQR</td>
<td>Inter-Quartile Range</td>
</tr>
<tr>
<td>ITU-T</td>
<td>ITU Telecommunication Standardization Sector</td>
</tr>
<tr>
<td>LTS</td>
<td>Labelled Transition Systems</td>
</tr>
<tr>
<td>MBT</td>
<td>Model-Based Testing</td>
</tr>
</tbody>
</table>
NFC .......... Near Field Communication
OWL .......... Web Ontology Language
OWL-S ....... Semantic Markup for Web Services
QoS ........... Quality of Service
REST ......... Representational State Transfer
RFID .......... Radio-Frequency Identification
RIF-PRD ........ Rule Interchange Format Production Rule Dialect
RS ............ Random Search
RS100 ........ Random Search with 100 Iterations
RS500 ........ Random Search with 500 Iterations
SCE .......... Service Composition Environment
SIP ............ Session Initiation Protocol
SLA .......... Service Level Agreement
SOA .......... Service Oriented Architecture
SQL .......... Structured Query Language
SSG .......... Diversity-based Steady State Genetic
SUT .......... System Under Test
TC ............ Test Case
TDE .......... Test Design Engine
TEE .......... Test Execution Engine
TT .......... Testing Technologies
TTCN-3 ........ Testing and Test Control Notation Version 3
UML .......... Unified Modeling Language
URL .......... Uniform Resource Locator
WADL ........ Web Application Description Language
WSDL ........ Web Service Description Language
WSN .......... Wireless Sensor Networks
XML .......... Extensible Markup Language
XSD .......... XML Schema Definition

Notations used in Equations

\( d \) .......... number of data values
\( j \) .......... number of populations
\( k \) .......... number of messages which involves data values
\( m \) .......... number of allowed symbols
\( n \) .......... number of test cases
\( N_A \) .......... number of actions
Nomenclature

\(N_E\) ............... number of events
\(N_{Paths}\) .......... number of created paths
\(N_p\) ............... size of the transition coverage set
\(N_{S_{max}}\) ........ maximal distinguishable states for a given number of events and actions
\(N_S\) ............... number of states
\(N_W\) ............... size of the \(W\) set
\(p\) ................. target number of test cases
Chapter 1

Introduction

1.1 Motivation

The Internet of Things (IoT) is becoming a valuable asset for realisations for various domains such as health care, transportation, disaster management, smart grid, smart cars, smart cities, smart homes, and smart factories (Industry 4.0). IoT is defined by the ITU Telecommunication Standardization Sector (ITU-T) as a global infrastructure to enable services by interconnecting things (physical or virtual) with Information and Communication Technologies (ICT) [1]. While computing processing power and communication technologies are becoming faster and cheaper, the number of things in the IoT is expected to increase up to more than 50 billion devices in 2020 [2].

The IoT is formed by a large number of sensor and actuator nodes providing heterogeneous and distributed resources from different owners. Today, implementations for the IoT are still limited to specific application domains and meet only isolated requirements of particular services. Those individual solutions hinder the penetration of IoT-based applications due to a comparatively time and cost extensive development process.

To overcome heterogeneity as well as sector and shared ownership boundaries (vertical boundaries), one promising approach is to apply a Service Oriented Architecture (SOA) to the domain of IoT [3]. According to the Open group [4], SOA can be defined as an architectural style that supports service-oriented development and the outcome of services.

A service is defined as a logical representation of a business activity with a specified outcome which is self-contained, may be composed of other services and is a “black-box” to consumers of the service. Concepts for service-based development, which can gather and derive data and information from sensor and actuator nodes can fulfil the need to dynamically design new types of services. Those IoT-based services interconnect the things with business processes, hence forming an application interwoven with the physical
environment. Interconnecting different sensor and actuator nodes, connected with different communication and architecture schemes, and exposing their features as services is a complex topic which is even more demanding for testing. To exploit IoT research it is necessary to have common and cost efficient methodologies for testing that ensure interoperable deployments [5].

The dependency on sensor and actuator nodes complicates the testing methodology. An obvious approach deploys the IoT-based service into the real physical environment. Alternatively, a separated but still realistic test environment, i.e. prototype testbed, can be designed with regard to the number of devices, network distribution, etc. Both approaches reveal major drawbacks. The utilisation of the real environment can critically affect the real world due to undesired actuations or unexpected side effects to coexisting services. On the other hand, a separate test environment requires the placement and configuration of sensor and actuator devices by the tester. This tends to become a complex challenge as soon as the number of devices and their heterogeneous functionality increases [6].

So far, only manually driven test approaches have been applied in the domain of IoT. Innovative Model-Based Testing (MBT) technologies enable (semi-) automating the test process in various domains such as object-oriented software, protocol testing, distributed systems and embedded systems [7]. MBT tries to derive a model out of a System Under Test (SUT), i.e. a system that is being tested for correct operation, and create test cases from this model (semi-) automatically [8]. While MBT allows for automating the process of testing, it is an open research question whether and how MBT techniques can be applied to IoT-based service testing. IoT-based services interact with the physical world and have to cope with problematic environments where input sensor data may be incomplete or erroneous. Testing IoT-based services differs from the typical evaluation of applications and services since their behaviour can often only be discovered through changes in the physical environment due to actuation. This changes the way a test needs to be designed as well as how the test environment needs to be developed. Also, IoT-based services demand for sophisticated methodologies that verify the execution and sensing behaviour at an early stage. This is particularly relevant for recommendation and actuating systems, e.g., a traffic control system.

The following IoT specific challenges need to be taken into consideration:

- Specific real world situations may not be available for testing and need to be emulated or should be generated in case of emergency scenarios.

- The time constraints of real world processes hinder fast testing.
Sensor devices are often resource constrained, e.g. battery powered, thus forbidding extensive testing.

IoT-based services that control actuator nodes need critical testing without/before affecting the real world.

1.2 Research Objectives and Questions

IoT-based services require a novel and domain specific test paradigm to overcome current drawbacks of manual testing since the interaction with sensor and actuator nodes change the way tests can be conducted. The main goal is to enhance the capabilities of testing whereby aspects for interaction with the physical environment are taken into account. The key objective is to ensure that an automated test derivation and execution process is established for simplifying testing. To achieve this objective the following research questions need to be addressed:

- What kind of domain specific aspects need to be taken into account when testing IoT-based services?
- How can IoT-based services be tested and what are their differences compared to web-service testing?
- How can an automated test process be flexible and extendible for different test targets and different SUTs?

Different owners of the distributed resources of the IoT and a varying number of possible sensor and actuator nodes result in a heterogeneous structure which can not be controlled by a single entity (such as a service provider). Therefore, it is important that developed IoT-based services can be tested without the need for interaction with external resources. This objective is motivated by the observation that the developed service may be based on resources that might not be controllable, e.g., when provided by a different entity or already used by other services whose execution can not be stopped. Furthermore, actuator nodes deployed in the real world (e.g., traffic management) can cause critical effects to the physical environment if they are utilised improperly. A so called sandbox environment is needed to verify if the SUT acts as expected in various situations before the IoT-based service is deployed in the physical world. A sandbox environment ensures that evaluation of the SUT can be conducted without any unintended effects from attached actuator nodes. Furthermore, it facilitates testing without any need to interact with external resources. As one requirement,
these external resources need to be emulated within the sandbox environment to make sure that the functional logic of the SUT can be evaluated holistically.

The most relevant research questions are as follows:

• What aspects can be tested without interacting with actuator and sensor nodes?
• Can emulation components be automatically instantiated and initialised?

Automation of testing can enhance the quality of the developed IoT-based services significantly. On the other hand, the methodology aims at guaranteeing that the test process can be executed with a reasonable amount of resources, including execution time. It is therefore a goal of the underlying work to ensure that the proposed test approach scales well.

The related focus of research comprises the following reflections:

• How can test automation be achieved in a scalable way?
• What are the bottlenecks of the test process in terms of scalability?
• Which techniques can improve the scalability?

A more detailed analysis of the research objectives is conducted in Chapter 3.

1.3 Research Methodology and Achievements

The achieved results and contributions to the state of the art are conducted based on the research methodology of analysis, conceptualisation as well as prototyping and empirical measurements:

**Methodology 1:** Analytical comparison of existing solutions for MBT are used to identify possible re-utilisation and adaptation for IoT-based service testing.

• As part of the analysis, different models are compared and assessed in terms of their capability to represent IoT-based services and to enable an automated test process. As a result, a new IoT service behaviour model for automated test case definition and execution is presented. It reflects the differences between the IoT-based services behaviour and web-services.
• The literature review of existing solutions indicates shortcomings with regard to testing IoT-based services in a productive or testbed environment. The comparison shows a high configuration effort. The current drawback is addressed with a test approach
that enables an early and simplified testing of IoT-based services by encapsulation and separation of service and sensor devices based on interface emulations. As a result IoT-based services can be tested automatically and simplified without the need to interconnect the IoT-based service with external resources during test execution. This separation between the service and external resources allows evaluating the service logic in a sandbox environment and can result in an accelerated testing and development, consequently.

- Investigation of MBT principles indicate that common approaches for deriving TCs from a behavioural model result in a huge number of TCs, which cannot be executed within a reasonable time and with reasonable computation costs [9, 10]. Therefore, it is eminently important to identify which TCs and which test data have to be selected for execution to ensure the best test coverage for the given time and resources. As a consequence, it is identified that the process of selecting test data and TCs, which have the highest probability to identify failures, is crucial for a successful application of model-based testing paradigms for IoT-based service testing. As a result this work follows a similarity investigation approach and identifies the most diverse TCs for test execution based on a group similarity between all selected TCs.

**Methodology 2:** Automated and early stage testing are conceptualised to derive a test architecture and a defined test process. The test framework and example services are prototypically implemented as a proof of concept.

- Concepts for sandbox testing based on emulation of external resources are designed referring to the results of the analytical methodology. Aspects of the IoT-based service behaviour and the needs for automatically deriving executable test cases lead to a framework for IoT-based service testing. The framework defines different architectural elements as well as a defined test process.

- As a proof of concept, the test framework is prototypically implemented. This novel test tool, called Automated and Scalable Model-Based Testing Tool for IoT-based Services (AuST), is designed and implemented. It enables testing of IoT-based services with MBT principles by separating the SUT from connected sensor and actuator devices. The chosen approach allows for a complete functional test coverage of the service logic by generating and emulating the sensor input values as required for the different TCs derived from the service behaviour model. It also facilitates testing the service logic behaviour at an early stage within the service lifecycle where elements of the service can be tested individually. This IoT-based service testing approach is
designed to be processed in an automated way, which would rarely be achievable with real world deployments of sensor and actuator devices. In addition, by integration of emulated sensor and actuator devices the time scale for test and evaluation can be accelerated.

- The problem of computing the similarity between TCs is reformulated to apply the Hamming distance and the Levenshtein distance algorithms. Afterwards, new optimisation algorithms based on Greedy, Hill Climbing, and Steady State Genetic are theoretically designed and prototypically implemented to optimise the test case group selection from a large set of TCs.

- Concepts for data type restrictions are retrieved based on guard dependency analyses and referring to test case and model analyses. The concept is prototypically implemented and as a result the data values are automatically generated prior to test execution for each test case individually. This methodology also improves the efficiency of integrating data values into the test case diversity optimisation.

- A complete test automation is provided that renders multiple model transformations from an IoTBM to a model for the script engine Velocity [11] and finally to Testing and Test Control Notation Version 3 (TTCN-3) [12] test cases. TTCN-3, being an European Telecommunications Standards Institute (ETSI) standard, is a test specification language, that supports black-box testing of distributed systems [13]. Due to its interfaces and the adaptation characteristics, the test language can be applied to different types of protocols and systems. The automation ensures that IoT-based services can be tested systematically without any need for human intervention during test design and hence supporting scalability and efficiency.

**Methodology 3:** Evaluation based on empirical measurements.

- Experiments based on example IoT-based services are conducted to proof the overall test process and the proposed approach of early stage testing based on emulation. As a result, different types of failures (including unexpected messages, data types, and data values) can be detected during test execution.

- Empirical test process evaluation is performed to evaluate the scalability of the proposed test process. As major outcome, the test compilation and execution, which have been performed with a commercially available tool called TTworkbench [14], are identified as the bottlenecks of the process. The obtained results lend weight to the
argument that for more advanced services it is not possible to execute all TCs within a reasonable time and without high computation costs.

- The evaluation of the test case diversity investigations are conducted based on various example services with different levels of complexity. The results strongly indicate that the proposed diversity-based Steady State Genetic algorithm can outperform existing solutions up to 11.6% with less computation time. Experiments to verify the failure detection rate show that proposed Group Greedy algorithm can enhance the failure detection rate up to 29%.

1.4 Thesis Organisation

The thesis is structured as follows. This chapter has provided an overview about the motivation of the conducted work, its objectives, the followed research methodology and related achievements.

Next, Chapter 2 examines the concept of IoT-based services, introduces testing approaches and discusses their applicability for IoT-based service testing. Major shortcomings of currently available manual testing approaches and high burdens of testbed configuration are discussed. MBT techniques for automated testing are analysed for adaptability and extendibility to simplify and automate testing of IoT-based services. Approaches for test case reduction are reviewed to address scalability issues and research directions are outlined.

Chapter 3 further defines the scope and objectives of the work. The main areas of contributions to publications and the EU FP7 project IoT.est [15] are outlined. Moreover, related collaborative research work is clarified.

In Chapter 4 new means for IoT-based service testing are characterised. It answers the question how MBT can be applied to the domain of IoT-based service testing without the need to interact with sensor and actuator nodes. Additionally, concepts for the proposed test-driven lifecycle management, the test architecture and the test process and its automation are provided. This results in a test framework for facilitating automated testing by means of MBT and IoT resource emulation.

Chapter 5 utilises the results of Chapter 4 and addresses the question how test automation can be realised scalable and adaptively for various types of services. As a proof of concept, the framework proposed in Chapter 4 is realised. The novel testing tool is called Automated and Scalable Model-Based Testing Tool for IoT-based Services (AuST).

In Chapter 6 the proposed algorithm for scalability improvements with TC selection is outlined based on diversity analysis. The author addresses open issues of currently known test
case diversity methodologies identified in chapter 2. Algorithms for similarity investigations are proposed and discussed. Various novel algorithms are compared to each other and their computation time complexity is discussed. The findings contribute to the question how the proposed algorithms scale and ensure reproducibility by the research community.

Chapter 7 presents the evaluation results of the implemented testing tool and the relevant test case reduction methodology. Example IoT-based services are utilised to prove its applicability. The performance of the test case selection is evaluated by discussing the corresponding results.

Chapter 8 finally concludes the thesis with a summary of achieved research contributions and future work directions are outlined.
Chapter 2

Background and Related Work

This chapter reviews related work in the field of model-based testing and IoT. Background knowledge and terminology definitions are introduced and will be reused in the subsequent chapters.

2.1 Internet of Things Based Services

Before explaining the concept of IoT-based services, common similarities between different IoT architectures are discussed within this section. Afterwards, the concept of atomic and composite IoT services is outlined, example services are explained and test opportunities are highlighted.

Architecture approaches for IoT have not been harmonised so far and standardisation is still, for most parts, in an early stage. Nevertheless, most of the proposed approaches follow a layered design where different tiers are designed to connect resource-constrained objects.
(e.g., sensor and actuator nodes) with a service interface by hiding the details and complexity of utilised technologies. Typically, objects and their functionalities are abstracted, a common set of components for device is provided and tools for creating IoT-based services are made available. Figure 2.1 shows the common layers of IoT systems (abstract architecture of [3, 16]) where the middleware represents a set of components to manage devices, entities (such as users or places) and context as well as providing means for service composition. The grey boxes indicate the typical realisation of the interfaces between the different layers. The vast number of objects as well as their heterogeneous and object specific communication often require utilisation of gateways for interconnection. The object abstraction layer typically exposes their abstracted functions as services for the re-utilisation within the middleware layer. In a similar way, the composed services are exposed through service interfaces to the application layer.

In recent years, a number of IoT architectures have been proposed. In related work the majority of researchers have designed an architecture which adopts the Wireless Sensor Networks (WSN) perspective (Sensei [17], IoT-A [18], SmartSantander [19], SmartThings [20], Bosch IoT Suite [21], IoT framework proposed by Cisco [22]). The IoT-A project provides an Architecture Reference Model (ARM) that includes i) an IoT Reference Model, ii) an IoT Reference Architecture, and iii) guidance from applications in different domains [23]. Various models such as the IoT Domain Model and the Information Model have been defined to reflect the domain specific information exchange. As one of the successors of IoT-A the IoT6 project leverage functionalities of IPv6 and re-use them in their architectural model. For example, discovery functionalities for services and resources with mDNS and DNS-SD [24] are utilised. From a high level perspective a common approach follows the service paradigm in order to overcome the heterogeneous nature of sensor and actuator nodes. As a consequence, the proposed test approach relies on a service concept that can encapsulate sensor and actuator nodes. The next subsection explains the underlying IoT-based service concept.

2.1.1 IoT-based Service Concept

IoT-based services utilise interfaces such as Constrained Application Protocol (CoAP) or Representational State Transfer (REST) [25] to encapsulate IoT resources for enhanced re-usability. Resources are software components that provide information about physical entities (discrete, identifiable part of the physical environment) or enable the control of devices [26]. As defined in [27], two types of services can be specified to ensure direct consumption and composition of IoT resources without dealing with heterogeneous interfaces:
i. An Atomic Service (AS) accessing \(1 - n\) IoT resources via its own individual interfaces and radio technologies. It enables access to these resources via standardised service interfaces. Here the AS is based on REST GET, POST, PUT, and DELETE request methods, whose invocation is defined in a Web Application Description Language (WADL) document [28]. The implemented AS can be deployed in a run-time environment for web services.

ii. A Composite Service (CS) enables a business process based composition of various AS and CS elements. It can also provide an interface to services that do not directly connect to IoT resources. It only uses AS and other CS to acquire sensor information and to control actuator nodes. The service logic can be described with the Business Process Model and Notation (BPMN) and a semantic description could be reused for composition and testing [29].

Figure 2.2 shows the schematic relation between AS, CS and IoT resources as defined within the previous enumeration. Note that each IoT-based service requires either a CS with at least one connection to an AS or one connection to a CS connected to at least one AS.

![Figure 2.2: IoT-based Service Concept.](image)

### 2.1.2 Value of Testing of IoT-based Services

IoT-based services tend to be complex due to their characteristics of heterogeneity, their distributed and mobile components and the integration of embedded devices. Therefore a variety of different failures can occur. Partly unavailable resources, wireless network connection losses and bandwidth limitations, dynamic interaction between devices with different capabilities (computation, network, memory, protocols) etc. can cause many
unexpected service behaviour if not tested precisely. From scenario perspective these failures can have catastrophically consequences if the service is not capable to react to these failures. Since the IoT-based services are connected with the real world via sensor and actor nodes they are able to change the state of the real world and this needs to be addressed by the test concept. Scenarios like disaster management, health-care or smart buildings can only be applied in a large-scale if a failure safe execution can be assured.

A combination of several system characteristics such as, vast number of distributed and resource limited sensor and actuator nodes, their heterogeneity and the consecutive interaction with the physical world demand for sophisticated testing methodologies. While there are other ubiquitous systems, such as embedded systems or Cyber-Physical System (CPS), IoT is more challenging to test due its size and its openness. IoT is intended to connect every kind of sensor and actuator nodes from different entities and make them accessible and usable for multiple purposes by multiple entities. This openness comes with some system limitations that differ to other more closed ubiquitous systems:

- Variation in resources: devices and components have varying characteristics in terms of storage, communication, processing, energy etc..

- Variation in latency and failure occurrence rate. Depending how the devices are connected to IoT, limited communication capabilities (e.g., Near Field Communication (NFC), wireless communication via mobile circular networks instead of wired), gateways and proxies tend to increase the latency and its jitter significantly compared to more closed systems such as embedded systems.

- Variation in data/information accuracy due to different entities involved and the immensely heterogeneity of devices.

- Incomplete understanding of how the application might affect the physical world. Due to the abstraction of sensor and actuator nodes, service developer is only capable to develop a service on a high level. Knowledge about the influences of actuation to sensor back-loops (like in control loop structures used in CPS [30]) only partly exists.

Note, that this is neither a complete list of characteristics of IoT nor is it the only distributed system that suffers from these issues. The characteristics have been selected to emphasise the need for more advanced testing capabilities. The characteristics are expected to be more demanding and challenging for IoT systems due to the rather incomplete system and application knowledge consequent upon their heterogeneity.
2.1 Internet of Things Based Services

2.1.3 Example IoT-based Services

This section presents three example services to illustrate the concept of atomic- and composite services. Later on, these example services are also used to explain and evaluate different aspects of the developed test methodology. The selection of these example services is driven by the need to provide rather simplified services that help to explain and verify the findings and contributions of the conducted research work. Note, IoT-based services are not limited to the described services and might have a higher complexity where, for example, even a single atomic service can have access to a large number of IoT resources.

Atomic Service Example: Camera Control Service  A camera control service is utilised to demonstrate how the followed test approach is applied to atomic service testing. The example service can control multiple Closed Circuit Television (CCTV) Cameras at different locations that are adjustable in their pan, zoom, and tilt via a REST interface. Another interface can be used to query the cameras position. Due to the characteristics of the service to adjust the position of the camera this atomic service possesses actuator and sensor characteristics. As a consequence, if the service is queried about a camera position, the response depends on previous interactions with the service. Therefore, the service is stateful.

Simple Composite Service Example: Window Open Service  The service logic is as follows: the service requests the current status of a window (‘Open’, ’Close’) every 60 seconds and each time alters the state of the window (e.g., close the window if it is open). Therefore, the service reacts to the current status of the window with different messages, which can be detected during test execution. The example service does not have any interface that can be directly tested. Instead, the service only communicates with one sensor and one actor service (atomic service).

Complex Composite Service Example: Temperature Control Service  To emphasise that the chosen approach can be applied to more complex services, a composite service with two different temperature sensors, one actuator to open a window, and one actuator to turn on an air condition is designed. Depending on the outside and inside temperature of a room, a window is opened or closed and the air conditioning is turned off or on.

Possible Failures  Some failures which may occur are based on:

- Resources (power supply, hardware failure, to many requests),
- Logic (functional or unhandled exceptions),
• Network (packet loss, bandwidth),

• Security (compromisation),

• Quality of information.

While security and network issues are related to any distributed-system, more advance and domain specific test methodologies are especially needed for identification of failures caused by resources, logic or the quality of information of IoT-based services. The reason is that the possibility and diversity of possible failures are higher than with other distributed systems (due to heterogeneity and shared ownership) and the potential effects of failure can be tremendous.

2.1.4 Opportunities to Test IoT-based Services

The meaning of software testing can be best described by defining the goal of testing; Myers et al. [31] describe software testing as a process which should ensure that the software performs as it was designed and verify that it does nothing unintended.

To the author’s best knowledge, only a very limited amount of related work exists on testing IoT-based applications and services. An overview of IoT related protocol testing is investigated by the PROBE-IT project [32]. Protocols such as Bluetooth, Zigbee, CoAP, NFC, RFID etc. are reviewed in terms of test profiles, existing tools and existing test specifications. As one of the outputs, interoperability testing of CoAP [33] is specified. Presenting one of the few IoT application testing approaches, Diaz et al. [34] try to reduce the testing effort and cost by systematically integrating gateways between the testing tool and the SUT.

In embedded systems, software test approaches have been designed to enable systematic testing. A good summary of recent activities is provided in [35]. Different to IoT, embedded systems are typically designed to achieve a dedicated goal, most commonly with real-time constraints, limited resources and dimensions with requirements to (re-) produce the system in large volumes [36]. Due to their dedicated function it is often possible to map parts of the physical environment by continuous time and discrete-event behaviour models [37]. While embedded systems and concepts for IoT share their ubiquitous behaviour due to integration of sensor and actor nodes, IoT approaches differ in terms of scale, openness and adaptability. While IoT actuator and sensor nodes might be already deployed, the full range of intentions of IoT-based applications can still be unknown. This observation can be compared with the development of the traditional internet where the utilisation of communication technology could not be safely predicted. IoT testing approaches need to be more adaptive and must not rely on closed system design concepts.
2.2 Classification of Testing Approaches

The systematic literature review of different testing tools such as ParTeG, Conforming, SpecExplorer and CertifyIt, conducted by [38], shows that these tools suffer from an incomplete definition of test adapters (except ParTeG) and stubs. While the so called test adapter is required to execute the created TCs, stubs are needed to emulate external components that are connected to the SUT. Further work is required to domain specific testing of IoT-based services that empower a faster and failure safe development of such applications.

As pointed out by Shafique and Labiche [38], there are many methodologies to model the SUT behaviour (Finite-state machines, Unified Modeling Language (UML), LOTOS, Markov chains, and Petri nets) with many tools supporting different aspects of model-based testing and its automation. Concerning challenges of model-based testing, Bertolino [39] claims that domain-specific test approaches are required to push the dream of 100% automation of the test process. MBT however, has not been applied to IoT-based services, along with the observation that further work is required to apply model-based testing for IoT-based services. The next sections will discuss general test concepts and verify their usability for IoT-based service testing.

2.2 Classification of Testing Approaches

Bourque’s and Dupuis’s definition [40] provides a detailed description about the meaning of software testing: 'Software testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usual infinite execution domain, against the expected behaviour’.

This definition includes the main characteristics of software testing. Different to statical analysis (e.g., code inspections) testing requires to execute the code either in the real or emulated environment which partly involves libraries, operating systems and network connections. Therefore, the possibility of detecting failures increases. Secondly, the infinite number of possible test cases need to be handled by the test process by effectively selecting test cases which provoke a different software execution behaviour. A key challenge in testing is the requirement of a deep understanding about how the software is supposed and expected to react to a certain test case.

The component which provides the expected behaviour is called oracle. While the problem statement of such a functionality is known for a long time [41], no overall satisfying solutions exist, especially for automation [39, 42]. However, model-based testing approaches can provide partial oracles based on the current test case. Since the source of information for the test case derivation and the oracle are the same, failures related to a wrong behaviour model are not detectable.
Testing IoT systems and services increases the requirement of understanding the desired behaviour. The behaviour of software needs to be verified in different environment situations since the physical environment interaction needs to be carefully tested. Consequently, the number of possible test cases also increases. Therefore, the selection of promising test cases as well as realistic and efficient modelling of the software and its environment behaviour is crucial.

Different categories of testing can be classified as proposed by Utting and Legeard [43]. They distinguish between the scale of the SUT (unit, module, integration, system), the characteristics being tested (functional, robustness, performance, usability) and the way test cases are derived (black box or white box). Functional testing aims at checking if the SUT produces correct output for a given input. Robustness tries to identify if the SUT can handle wrong inputs, unavailability of required external components, as well as network and hardware failures. Performance testing identifies if the SUT can fulfil specified Quality of Service (QoS) under heavy load situations. Usability testing verifies if the designed user interfaces cause problems due to wrong usage or misunderstandings by the user.

Bertolino classifies testing issues by formulating several questions as follows [39]:

- Why do we test? Are faults or usability to be evaluated?

- How are test cases selected? Is there a systematic approach, or a random execution, or some kind of ad-hoc testing mechanism?

- How much testing is enough? What is the stopping condition and what is the coverage of the testing?

- What is tested? A whole system or parts of it? Is the SUT part of a larger system?

- Where is the SUT executed, in the target context or in a simulated context?

- When is the SUT executed, at which stage of the product lifecycle?

Both classification approaches have identified the question how test cases are derived. Initially, approaches have been distinguished between white-box testing approaches, which have access to the source code and black-box testing, which is based on the software specification (although there are also some mixture approaches called Gray-box testing). Due to the vast number of approaches this classification has been replaced by a categorisation based on the used test methodology rather than the source of information (code or specification).
2.3 Model-based Testing

In recent years research MBT has raised interests from both industry and academy. The main idea is to use models derived from the system or the specification, and create test cases from this model. Afterwards the created test cases are executed and the reaction of the SUT is compared with the expected behaviour of the model. The leading idea is to derive test cases in an automated way if the model follows some machine interpretable notation [43]. Figure 2.3 shows the key relations between the SUT and the model. The model is a partial description of the SUT and is used to derive test cases. The test cases can then be run against the SUT. In case a failure is detected, the model and the SUT differ. Therefore, it can not be clear per se whether the detected failure is caused by a faulty SUT or a faulty model. In contrast to MBT some efforts have been invested to reason about the SUT behaviour from the visible system (output) behaviour instead of building an a priori model of the SUT. This approach can be categorised as a reverse-engineering approach and reveals some advantages related to dynamic programming analysis [44].

The roots of MBT have been introduced by Moore in the 1950s [45] and test automation in the 1970s by Chow [46]. However, most recently these theoretical backgrounds have been applied to different domains (Object-oriented software, reactive systems, distributed systems, embedded systems, protocol testing) although the concepts have not been widely applied to web services and Commercial-off-the-Shelf (COTS) [7] yet.

The main elements of mode-based testing can be described as: i) the mode describing the software behaviour, ii) the test algorithm to create the test cases, and iii) the tools which provide this process [47]. Within this thesis the focus is on describing the usefulness of common models to describe IoT-based services and highlight the missing functionality in...
order to create test cases out of the model. A good feature classification has been invented by Utting et al. [48] (partly adopted from [49]). The model is characterised by Subject, Redundancy, Characteristics, and Paradigm. In order to enable testing of IoT-based services we can describe these aspects as follows:

- **Model Subject**: paraphrases the intended behaviour model of the SUT and its environment. Traditionally, the environment can be considered as the model to restrict possible inputs or define stochastic models for typical user interaction with the SUT. However, for IoT-based services the model subject moreover requires modelling the interconnection with the real world through sensor and actuator nodes.

- **Redundancy**: One of the general long term goals of testing [39] is to share only one particular model for both testing and development. However, this target will probably not be achievable in the near future as far as IoT-based service testing is concerned. Therefore, the model we intend to derive will be limited to describe the testing perspective of the SUT and the relevant environmental behaviour.

- **Characteristics**: The characteristic describes if the model is discrete, continuous or a mixture of these two. For the sake of simplicity we assume the SUT to be discrete. Note that it is possible to describe continuous systems with discrete systems to avoid complex modulation approaches [50]. Deterministic properties of IoT-based services can not be guaranteed due to data jitter caused by detecting real physical values and time jitter generated by the sensor itself, involved components (e.g., gateways) and the underlying communication and network protocols. However, these data and time variations can easily be handled within the test result evaluation phase (e.g., the verdict description). Therefore, from the model perspective the SUT can be seen as deterministic. Timing becomes a more complex challenge if real-time systems were to be investigated.

- **Paradigm**: Notations to describe the model can be grouped into state based, transition based, history based, operational based, stochastic based and data flow based (cf. [48] for comparison). Table 2.1 shows some of the most common approaches and discusses their usefulness for IoT-based service behaviour modelling.

While behaviour modelling of IoT-based services seems to be feasible with many of the approaches introduced above (FSMs, labelled transition systems, UML State machines, Stream X-Machines) it is rather uncommon to model the environment (apart from user interaction) with classical model based testing approaches. Although Matlab has been widely
### 2.3 Model-based Testing

Table 2.1: Behaviour Model Notations and their Strengths for IoT-Based Service Testing.

<table>
<thead>
<tr>
<th>Modelling Notation</th>
<th>Description</th>
<th>Possible Extensions</th>
<th>Strength for IoT service testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite State Machine (FSM)</td>
<td>A FSM (in the domain of testing) is described by sets of input/output symbols, sets of states, state transition functions and output functions (Mealy Machine).</td>
<td>Data variables, timing, hierarchies of machines and parallelism between these machines.</td>
<td>• Well suited to model SUT behaviour</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Limited suitability for parallelisation</td>
</tr>
<tr>
<td>UML Models</td>
<td>Consists of events, states, extended states representing states with variables, guards (condition for action), actions and transitions.</td>
<td>Transition execution sequences.</td>
<td>• Well suited for visual creation and presentation of a behaviour model</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Well suited for SUT development and white-box testing</td>
</tr>
<tr>
<td>Stream X-Machine</td>
<td>Extended finite state machine for modelling also data variables in the memory. Transitions compute output values and update the memory based on the input and the current memory variables.</td>
<td>Time events and connections to other models.</td>
<td>• Well suited for model derivation due to specified mathematical and transition functions</td>
</tr>
<tr>
<td>Z Notation</td>
<td>Formal specification language to describing software systems. It consists of states, relationships as the system moves from one state to another, operations, relationships between inputs and outputs, allowed state changes.</td>
<td>—</td>
<td>• Well suited for theoretical and mathematical model analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Limited suitability for testing since the models are rather complex and not intuitively usable</td>
</tr>
<tr>
<td>Matlab Simulink</td>
<td>Parts of Matlab Simulink like block diagrams are rather modelling the control flow and are used to describe continuous systems.</td>
<td>Simulink state-flow is one extension to model transition-based state machines and state-charts.</td>
<td>• Well suited for process simulation and modelling of closed loop systems (among others)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Limited suitability for distributed and partly unknown systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Limited suitability to derive a behaviour models out of a SUT specification due to rather graphical oriented modelling</td>
</tr>
<tr>
<td>Markov chains</td>
<td>Stochastic model, commonly used to describe queueing networks.</td>
<td>—</td>
<td>• Well suited to describe user pattern or interactions of the SUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Limited suitability to describe the interaction of the SUT with the environment due to indeterministic behaviour</td>
</tr>
<tr>
<td>Labelled Transition System (LTS)</td>
<td>Introduced to describe data-intensive systems and hardware circuits. A formal description are used to describe the SUT and test generation algorithm can generate valid tests out of the specification.</td>
<td>Adoptions of FSM algorithm like state identification.</td>
<td>• Well suited to describe the behaviour model</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Well suited to derive behaviour model from SUT specification documents</td>
</tr>
<tr>
<td>Petri network</td>
<td>Describing executable processes, in parallel. Well suited for describing communication protocols and distributed systems.</td>
<td>—</td>
<td>• Limited suitability due to the scope of parallel processing.</td>
</tr>
</tbody>
</table>
utilised for such an approach, it has some major drawbacks if (semi-) automated model derivation from system specification is intended. While Stream-X and LTS have profound methodologies to derive the behaviour model out of a service specification, UML approaches are better to visualise and better to understand by a service or test developer. Overall, state based behaviour models such as Extended Finite State Machine (EFSM) are identified as good candidates for modelling the behaviour of an IoT-based service since they are capable to model a deterministic behaviour as well as providing visualisation capabilities and means to derive test cases out of the model. So far, existing models have not been applied to the IoT domain. Extensions are needed to ensure that the aspects such as the controlling characteristics (e.g., sensor and actutor node interaction) are covered.

2.4 Testing with IoT Resources

Real world interaction of services connected to Internet of Things (IoT) resources require sophisticated capabilities for testing. Functional tests have to assess that it is uncritical to deploy the service in a productive environment (e.g., traffic control system not working properly). Contrary to common service testing approaches, the strong interaction with the physical environment needs to be addressed by the test environment. Furthermore, the scalability and time efficiency of the approach needs to be estimated realistically. Previous testbed approaches handle IoT resources in different ways (e.g., WISEBED [70], Kansei [71] cf. Gluhak et al. [72] for a comprehensive comparison of testbed features):

- Physical IoT resources in a testbed
- Virtual IoT resources in a testbed
- A combination of physical and virtual IoT resources
- Gateways to generically connect physical IoT resources

In the first category real hardware is used for testing the behaviour of the system. Therefore, hardware-specific problems can be identified during the test execution. The main drawbacks are the lack of scalability and the limited ability to test specific situations (e.g., room temperature $< -20^\circ C$). Improved scalability as well as larger control of test situations can be achieved with virtual resources. Most approaches build a virtual machine that represents the IoT resource. The target platform can then be directly deployed in this virtual machine. Tests can therefore be run from the perspective of IoT resources as well as from connected services. Although, concepts combining virtual and physical resources in the
testbed have a great coverage of hardware, IoT resource, and software-related failures, they show limitations with regard to scalability. While it is comparably easy to handle one specific IoT resource and build a virtual equivalent, it remains unclear how hundreds of different IoT resources can be handled. In addition to this scalability issue, it is necessary to implement the virtual machines of the IoT resources as well as the test environment needs to know how to interact with the virtual resources (e.g., provide hardware interfaces like Bluetooth, ZigBee, and also have knowledge of partly proprietary protocols at the application layer). One approach of overcoming this limitation has been proposed by Diaz et al. [34]. The authors argue that a systematic implementation of gateways can lower the costs of testing by connecting the SUT and testing tool. As a result, complexity of interaction between the SUT and the IoT resource is hidden from the test framework but still needs to be modelled within the gateway.

Another open issue of using real IoT resources is the synchronisation between the SUT and the IoT resource. Knowledge about the required time duration for initialising the IoT resource after power-on or reset is required. Also, the power-on or reset capability needs to be either triggered manually by pushing a button or requires a strong debugging/controlling interface, capable of resetting the hardware by software.

Both solutions tend to be complex in case that more than one IoT resource is involved. By choosing a solution with virtual resources realising this functionality is easier, but there is still the need to synchronise the virtual IoT resource with the test framework. Due to the complex model, this reset process will a take a specific time – probably the same amount of time as required for utilising the physical hardware.

### 2.5 Test Automation

As defined by Utting [48], model-based testing can be interpreted as a process to automatically derive concrete and executable test cases from abstract formal models. In principle, formal models should be interpretable by machines to derive the concrete test cases. However, also manual test case derivation approaches still count as model-based testing attempts and only a minority of model-based testing approaches include an automated model derivation process from specifications (cf. [73]). Nevertheless, relevant research has been conducted to simplify the model creation process (e.g., automated data type generation [74]). The basic process of model-based testing can be separated into various design steps and described as follows:

**Step 1.** Create a model from a specification document to ideally cover all relevant behaviour aspects which are to be tested. For automation, a language for the specification with inference capabilities needs to be defined. While approaches based on Stream X-Machines
[58, 59] have presented notable methodologies (cf. [73]), conducting this step automatically is not common. However, recent trends indicate that the research efforts have increased significantly [75]. Current approaches focus mainly on Web Ontology Language (OWL) [76, 77] and Web Service Description Language (WSDL) [73, 78]. For IoT-based services the additional communication between AS and CS makes it more ambiguous to derive a service behaviour model out of a specification document and future work is needed to extend existing approaches for IoT-based service testing.

**Step 2.** Test case derivation needs to create test cases out of a behaviour model. The previous step has provided a consistent model (commonly created by manual graphical design [79–81]) and the model needs to be analysed to identify paths which can be traversed during test execution. For IoT-based services this derivation has never been applied systematically. Other software domains, e.g. protocol testing [82], provide an interesting origin for further enhancements.

**Step 3.** Test cases can be selected either based on abstract test cases or by creating only a predefined number of concrete test cases based on a dedicated test data generation process ([83]). Approaches based on abstracted test cases try to eliminate irrelevant abstract test cases and therefore limit the test data space before the data generation process is started. All approaches suffer from the large test space [10] and further improvements are required to handle the test space for IoT-based services in an automatic and diverse way.

**Step 4.** The test execution needs to control the SUT, as required for the current test case, and run the test cases. This process can either be realised with static test cases, produced before the test execution (offline), or can have a dynamic adoption component which generates the concrete test data values as a function from previous test results (online). The offline approach is rather common for functional testing while online testing has been utilised to identify crucial parametrisation of a SUT (e.g., Service Level Agreement (SLA) violations [84]).

**Step 5.** The test evaluation needs to validate if the test results are as expected. In theory, the test model describes the behaviour of the SUT and therefore all required information for validation is already present in the model. Nevertheless, the model is often limited in the concrete behaviour. Instead of describing the exact value it regularly describes only the expected data types [41, 85]. For IoT-based services data type definitions and there physical representation, need to be described.

Table 2.2 summarises promising approaches and concepts that may be re-used for IoT-based service testing according to the steps discussed above. For a complete review of using the test tools readers may refer to [86] and [87].
Table 2.2: Test Automation Approaches.

<table>
<thead>
<tr>
<th>Step</th>
<th>Approach</th>
<th>Description</th>
<th>Usability for IoT service testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>WSDL</td>
<td>WSDL descriptions enriched with semantics and rule annotations [73, 88].</td>
<td>Already available approaches do not provide enough capabilities to express the nature of IoT-based services. The data descriptions are rather focused on computation data types and do not take into account that data from real sensors represent a physical quantifier which has specific borders based on the current context (e.g., temperature in a room). An adequate degree of automation of model derivation has not been achieved so far.</td>
</tr>
<tr>
<td></td>
<td>UML</td>
<td>Many model-based testing approaches are based on UML descriptions (cf. [89, 90]). While developers are familiar with UML, and thus lowering the gap for utilisation, the descriptions are too incomplete to derive tests in an automated way.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>OWL</td>
<td>Wang et al. [91] derive a behaviour model from an OWL description. An information transformation algorithm is utilised to derive a Petri-net model. Another approach proposed in [92] is based on Input, Output, Precondition, Effect (IOPE) descriptions represented with Semantic Web Rule Language (SWRL) rules as well as OWL ontologies and enables the derivation of an EFSM.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>W-method, Wp</td>
<td>The first approaches for deriving test cases out of a FSM are based on the W-method providing state and transition coverage in one step and the Wp algorithm which reduces the number of required test cases by splitting state and transition coverage into two steps [93, 94].</td>
<td>The theoretical investigations allow for guaranteeing state and transition coverage, which is also applicable for IoT-based services. Nevertheless, full state and transition coverage tend to result in many test cases and further work is required to optimise the selection of test cases.</td>
</tr>
<tr>
<td></td>
<td>HSI, H</td>
<td>The HSI method reduces the number of required test cases, compared to the W-method, by harmonising the state identifiers. The H method enables an on the fly identification of these identifiers. [95]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UIOv, DS</td>
<td>The UIOv can be seen as a special case of the W-method and can be applied if an unique input output exists, for each state identifier, and thus shorten the test path [93]. Similar the DS method can only be applied if the characterisation set of the W-method consists of one single characterisation set for the specified FSM [95].</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Diversity-based</td>
<td>Diversity-based approaches try to select test cases based on their diversity. Either the test case itself is rated or the test data generation process is driven by diversity investigations. Approaches include search or cluster-based methods [83, 96].</td>
<td>Diversity algorithms have proven their efficiency for classical software testing. Further investigation is required to evaluate if the results can be transferred to the domain of IoT. Future utilisation of online adaptation of test data based on recent test results is an interesting approach for improvements.</td>
</tr>
<tr>
<td></td>
<td>History-based</td>
<td>History-based methodologies try to leverage statical correlations between previous test results. Hence, they have been mostly applied wrt. regression testing. However, relevant research tries to use on the fly evaluation of results to calculate critical input data for the next test case (cf. [97, 98]).</td>
<td></td>
</tr>
<tr>
<td>4 &amp; 5</td>
<td>Offline Testing</td>
<td>Offline testing tools like LTG, JUMBL, TAF, AETG, and TTworkbench ([14]) create test cases prior to executing the relevant tests. Tools can be found for functional, reliability, security, and interoperability testing [99].</td>
<td>Many of the testing tools are either commercial or closed source academic implementations and thus make it hard to evaluate the tools and the corresponding test results. To ensure that research results can be comprehended it is necessary to build the test cases based on a standardised language like TTCN-3 (supported by TTworkbench and Conformiq/Qtronic[100] and in [99]).</td>
</tr>
<tr>
<td></td>
<td>Online Testing</td>
<td>Tools like Spec Explorer, TGV or TorX are able to react instantly to the SUT behaviour during test execution. Some tools like Qtronic provide interfaces for external provisioning of online testing capabilities.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Oracle Automation</td>
<td>While tools like Conformiq/Qtronic and TAF use the model itself to create the test oracle, others rely on the users’ input (AETG, Jumble) or on defined input output relations (LTG, Spec Explorer).</td>
<td></td>
</tr>
</tbody>
</table>
2.6 Test Case Selection Optimisation

Automated test case generation is a major benefit of MBT techniques. The process of test derivation and execution with automatic generation algorithms can often produce a large number of test cases and may not be cost-effective [10]. To address this issue, the test developer can manually select test cases for execution but this can cause that important test cases, which detect failures, are not selected [101]. Concepts for reducing the number of TCs automatically are therefore an extensive research activity [10].

For MBT, methodologies for test case selection exist based on i) test execution history, ii) test coverage/requirements and iii) similarity investigations (for other approaches, especially for white-box testing, cf. [102]). TC reduction based on execution history is one of the widely utilised techniques for regression tests [97, 103]. The knowledge about which TCs failed in previous versions of the SUT, is taken into consideration for selecting the TCs that are executed during the current test procedure. Related research extended this approach by categorising SUTs and assuming a correlation between failures that occur at different SUTs of the same category [98]. Nevertheless, for this approach it is crucial to have a good history base. The assumption of correlation between failures in different versions of SUTs of the same categories has not been proven so far as being valid for all varieties of software (with the best knowledge of the author). One methodology to overcome a missing execution history is based on mutation testing. For example, Zhang et al. [104] combined mutation testing with test reduction and TC selection/prioritisation. While the results show effectiveness of the technique it is not possible to apply them for black-box testing.

A more straightforward technique is the reduction of target coverage [105]. By limiting the area of interest, the reduction of the number of TCs is possible. Other approaches follow the idea by utilising heuristics ([106–108]). However, the question about the relevance of elements still remains. Studies have shown that coverage/requirements-based reduction may decrease the possibility to detect failures [103, 109]. The MINTS tool combines coverage, history and cost data as basis for the test case selection [110]. It models the multi-dimensional minimisation problem as a binary Integer Linear Programming (ILP) and is able to apply different ILP solvers. The results indicate that this approach performs as good as classical heuristic techniques. However, the approach is only applicable for regression testing and white-box testing. It remains unclear how to proceed if no optimal solution exists (e.g., contradicting requirements). Other approaches try to use scenarios or TC weighting mechanisms by including the experts’ knowledge [111]. Nevertheless, some results indicate that the manual decision which TCs or test areas are important can even have a negative impact on the failure detection rate compared to random selection.
Recent trends in software engineering indicate that search-based optimisation techniques are a promising candidate for several software issues such as requirements, project planning, testing and re-engineering optimisations [112]. Proposals for test case selection (often called prioritisation within this context) include the utilisation of Greedy [113] and clustering [114][115] algorithms for white-box testing. The results indicate that cluster-based algorithms can outperform traditional coverage-based TC selection techniques by including human input. Early researchers adopting search-based approaches for model-based testing were Cartaxo et al. [116], who tried to select the less similar TCs while maximising the state or transition coverage of the test model with the remaining TCs. Hemmati et al. [96] extended this approach by utilising several similarity functions as well as minimisation algorithms and applied them to two larger SUTs. The findings indicate that the choice of technique significantly influences the performance. However, it remains open if the results can be transferred to other SUTs and whether there is a general setup which is always the best.

Cautinho et al. [117] focus on the comparison of different similarity functions (Similarity function [9], Levenshtein distance [118], Sellers algorithm [119], Jaccard index [120], Jaro distance [121], Jaro-Winkler distance [122]) for TCs derived for LTS and measure the influence to one reduction strategy inspired by Cartaxo et al. [9]. The results show that the performance of the chosen reduction strategy is significantly influenced by the chosen similarity function but no general best algorithm could be identified. Since only one reduction strategy is applied it remains unclear if the results can be transferred to other selection/reduction algorithms. A comparison to random selection is also not shown and therefore it is not possible to determine if there is any gain compared to a baseline approach.

The discussed studies of Cautinho et al. [117] and Hemmatie et al. [96] follow a promising approach to reduce the number of TCs efficiently and effectively. The realised research work within this thesis is influenced by the general methodology of those studies and therefore the next subsections highlight the overall strategy of the followed approach. Afterwards, major open research issues for test case reduction based on simplification investigations are outlined.

### 2.6.1 Test Case Selection through Similarity Investigations

Hemmati et al. [96] categorise their approach into three steps: i) encoding of test cases and test model ii) measure the similarity between test cases and iii) find the test cases with the largest diversity. Figure 2.4 illustrates the main test case selection steps and their key methodologies. Test cases are usually described with different types of information
and therefore the divergence comparison can be interpreted as a problem of identifying similarities over a multi-dimensional feature space.

![Diagram](image.png)

Figure 2.4: Test Cases Selection: Steps and Methodologies.

**Encoding:** During the encoding phase the information to be considered in the comparison process is preselected. Classical abstract test case descriptions (e.g., state based, transition based, event guard based) can be used to define the information included in the encoding [96]. While combinations of the descriptions provide the most extensive information, simple models can reduce the execution time of the whole test case selection process. In addition, if the tests aim at identifying specific failures (e.g., evaluating transition failures) the encoding needs to reflect these testing goals by removing irrelevant information.

**Similarity:** Two different categories of similarity calculation algorithms exist. The similarity can be calculated with set based algorithms, which do not take the order into account, or with sequence based functions. Sequence based function, like the Hamming distance, are widely used to estimate the error in a transmitted message based on the distance to allowed messages. Alternatively, they can be utilised to measure the distance between properties of a multidimensional feature space in data mining services [123]. To compute the Hamming distance for test cases, each test case is represented with a binary string; the number of identical bits between two test cases are counted and divided by the number of all possible elements. Since test cases can have different lengths, algorithms such as Jaccard Index [120], Gower-Legendre [124] and Sokal-Sneath [124] are supposed to be able to result in a better comparison. All three algorithms are based on a weighting between the intersection of A and B and the union of A and B (where A and B are two different strings). Another possibility is
using counting functions and identifying the number of identical elements in the input sets divided by the average length of inputs.

Sequence based functions take the order of the elements into account. This brings additional comparison capabilities if applied to test cases similarity investigations. The Levenshtein algorithm [125] is especially useful if the compared strings have gaps (e.g., both test cases include the same states but one test case has a loop at one point). Gaps can be used to increase the distance of two test cases or reward matches and penalise each mismatch. Other approaches such as the Needleman-Wunsch similarity function [126] follow basically Levenshtein’s approach but applies different weights (higher score for similarity and stronger penalise for mismatch). Other algorithms for local alignment like Smith-Waterman [127] can be used to find the best alignment for sub-sequences. Therefore, it would better fit when detecting sub test cases with similar characteristics.

**Test Case Selection:** While set based or sequence based functions can provide the similarity between two strings, the process of identifying test cases with the largest diversity is known to result in a NP-hard search problem. Therefore, a complete comparison between all test cases is not possible. Relevant approaches comprise i) Greedy minimisation ii) Clustering minimisation or Search minimisation (cf. [96]). Greedy based minimisation selects the test cases with the highest similarity score and removes the one with the shortest length from the database [96]. This approach assumes that longer test cases result in a higher possibility to detect test cases but does not take into account the circumstance that more time is required to execute the test cases.

Clustering techniques cannot be considered as a minimisation technique [96]. However, clusters can be formed based on the similarities and distances between test cases. Prominent clustering techniques comprise i) K-Mean clustering, which tries to minimise the average squared Euclid distance of objects to the cluster mean, and ii) the agglomerative hierarchical clustering, that starts with forming a cluster with the size of 1 and merges clusters until the desired number of clusters exist. For both clustering algorithms representative test cases can be selected from various clusters and assure the test cases are diverse. A complete review of clustering mechanism can be found in [128].

Search based minimisation techniques may run for a long time if no adequate stopping criteria is defined (cf. [83] to compare different search approaches). For test case selection, the search algorithm also needs to avoid the selection of duplicate test cases. As a baseline, random search can be used to identify the influence of the selected search methodology. For the application of test case selection, a uniform distribution without replacement will be efficient if we assume the failures to be uniformly distributed over the test case space.
In contrast, local search algorithms like Hill Climbing [129, 130] try to start at a random position and then look for neighbour solutions (i.e. smaller similarity). This approach is promising if the test case creation process is based on an algorithm which tends to create similar test cases in a sequence. The Simulated Annealing algorithm [129, 131], an alteration of the Hill Climbing, does not start from a random element and takes previous positions into account instead, specifically with a cool down function. Another popular approach for generating test cases utilises a Genetic algorithm [132, 133] and is applicable particularly for diversity searching. The Steady State Genetic algorithm [134, 135] selects the offspring which does not have a worse fitness value than their parents and starts the next generation. Meta-heuristic search techniques such as the Memetic algorithm can combine global and local search approaches by starting a global search to identify the starting point and then continue with local searches until a local optimum is reached.

2.6.2 Research Objectives for Test Case Diversity Optimisation

While the most recent empirical studies of Cautinho et al. [117] and Hemmati et al. [96] have contributed to the understanding of different similarity and selection techniques, several research gaps still remain open, including:

- The model utilised in both studies is rather abstract and does not take data value related failures into account. By integrating data values into the distance computation the target test coverage can be enhanced.

- The description of the algorithm is often incomplete, hence making it impossible to reproduce the evaluation results. For example, Hemmati et al. only describe the genetic inspired algorithms at an abstract level and do not provide enough detail of how to apply them to the problem of TC selection. Future work is needed to document the realisation of those algorithms.

- The computation time required to execute the different algorithms is not investigated in detail. Theoretical and empirical analyses are essential to provide a fair comparison between different algorithms.

- Both empirical studies use a depth-first algorithm to derive the TCs. Future work is needed to analyse if the results are similar with other algorithms such as the W-method. For example, the W-method ensures state and transition coverage and therefore the TCs are longer, compared to depth-first, but might lead to a higher similarity due to occurrence of state identification sequences.
• Misunderstanding of algorithms results in wrong measurements. For example, the Levenshtein algorithm is not correctly interpreted by Hemmati et al., since it does not follow the key idea of finding the smallest number of steps to change one TC to the other (delete, add, etc.). Therefore, future work is required to investigate the Levenshtein distance performance.

• The investigation of related work focuses on failure detection rate and on reducing the number of TCs. Future work is needed to further analyse the algorithms. Other interesting characteristics that should be taken into consideration are the average similarity between the selected TCs and the distribution of the similarity score values.

• A fair comparison between different algorithms is missing. So far the computation time is not taken into account. To make sure that the performance is not only caused by additional computation time, the different parameters of the algorithm need to be aligned as much as possible. For example, the stopping criteria can be adjusted to provide similar computation time of the different algorithms.

• The test case selection is performed based on a pairwise similarity score. Both studies optimise the selection stepwise based on the comparison between two TCs subsequently. However, the initial goal was to find a group of TCs that has the lowest similarity. Future work should therefore focus on a group oriented optimisation.

2.7 Summary and Discussion

The potential effects of deploying malfunctioning IoT-based services into the physical environment demand for advanced test capabilities. So far, only manually driven test approaches have been applied in the domain of IoT. High configuration effort for IoT resources such as sensor and actuator nodes hinder efficient testing approaches. Currently, the required effort for initialising and operating real or virtual testbeds limits the capabilities to test IoT-based services efficiently and effectively. Future work needs to aim at testing the logic of an IoT-based service without connecting the service to external resources such as sensor and actuator nodes. In contrast to earlier approaches, a simpler and more automated emulation methodology is suggested.

MBT has been utilised to test object oriented programming or web services. The outlined work has therefore investigated how the IoT service development can benefit from the capabilities of automated test derivation with MBT principles. Model notations like FSM, UML, Stream X-Machine, Z Notations, Matlab-Simulink, Markov chains, LTS and Petri
networks have been evaluated based on their applicability for IoT-based service testing. FSM and LTS have been identified as the best candidates for modelling IoT-based services. To the best knowledge of the author, none of the currently existing models can be applied directly to IoT-based services testing. Instead, extensions are required to ensure the service logic can be tested efficiently without the need to build and maintain complex testbeds. Future model approaches need to cover aspects for behaviour modelling of the SUT (e.g., controlling characteristics), provide means to emulate their connected resources (sensor and actuator nodes), and make sure that these models can be transferred into executable TCs.

The process of test automation has been discussed and methodologies for test model derivation, test case creation, test selection and test execution have been reviewed with regard to their applicability for IoT-based service testing. So far, automated testing has not been applied to IoT-based services. Future work is needed to optimise the automation degree of such a process for IoT-based service testing. As one of the open issues the complete automation of the test model derivation from of a system specification has been identified as a long term goal.

As one of the drawbacks of model-based testing, the number of test cases increases significantly with the complexity of the SUT. Required techniques for test case selection have been introduced and methodologies for similarity investigations have been reviewed. Recent work indicates that test case selection based on similarity investigation is a promising methodology but extensions are required to verify the performance of various algorithms. In conclusion, the proposed algorithms are often described incompletely, hence making it impossible to reproduce the evaluation results. The computation time required to execute the adopted algorithms has not been investigated in detail. Theoretical and empirical analyses are needed to provide a fair comparison between different algorithms. The optimisation goal of currently known algorithms has been identified as one of the major drawbacks since the optimisation is performed on the subsequent comparison between two TCs. Future work is proposed to focus on a group oriented optimisation. Also, the applicability of the similarity computation algorithms to the domain of IoT-based service testing and necessary adaptations will be discussed in the following chapters of this thesis.
Chapter 3

Research Scope

3.1 Research Objectives and Challenges

The research scope has already been briefly motivated and introduced in Section 1.2. Based on the concepts and background presented in the previous sections, this chapter provides a more detailed and narrowed description of the research challenges, objectives and scientific contributions and puts them in the context of the state of the art.

Objective I: Test Automation for IoT-based Services

Test automation by means of MBT requires designing a behaviour model that can be utilised to i) describe the behaviour of the SUT, ii) derive test cases and iii) make them executable [136]. Therefore, one key objective is to create such a model for IoT-based services. The service logic of the SUT needs to be automatically testable with regard to functional evaluation. The problem when creating such a model relies on finding a representation that ensures a certain degree of freedom (adaptability) to model different behaviours/reactions of the SUT and ensure that these different descriptions can still be automatically transformed into executable test cases.

As discussed in Chapter 2, several models exist to describe the behaviour of a SUT, but automatically employing these models to create executable TCs is still an open research question that requires further investigations. Compared to theoretical models, more domain specific information is required [39], such as protocol-, externally attached resource-, or deployment information. To ensure that the created TCs can be easily understood and potentially be extended the utilisation of the standardised TTCN-3 notation is intended. It allows using well established concepts for creating executable TCs from a defined test language. The objective is to automatically derive TCs that are TTCN-3 conform, which
can be interpreted with an already established testing tool. For this work, a commercially available tool called TTworkbench [14] is utilised but other tools such as the Conformiq Designer (former known as QTronic [100, 137]) could have been applied as well. When designing the behaviour model, the goal of transforming the test cases into TTCN-3 needs to be considered. Various behaviour model objects need to be mapped and/or inferred to TTCN-3 notation elements in order to be transformable. Different to theoretical model analysis, deriving abstract TCs and creating executable TCs manually is not sufficient.

Objective II: Automated Emulation of External Resources

The required service behaviour models differ from traditional evaluation models primarily due to an explicit description how the SUT is connected with sensor and actuator nodes. The goal is to utilise the model to separate the SUT from external resources (such as sensor and actuator nodes) by creating components for external resource emulation in an automated way. Therefore, the model needs elements for describing the communication between the external resources and the SUT including logical interpretation of data exchange (e.g., physical values such as temperature). So far, most of the MBT tools only provide test stubs for emulating components [34, 38].

For testing of composite IoT-based services, the focus is on ensuring that the behaviour model can be utilised to completely test the functionality of the SUT within the sandbox environment. Therefore, the emulated comments needs to respond as required by the specific TCs. E.g., if a SUT reacts to a temperature value below 25 °C with a heating actuation this behaviour can only be verified if the emulated sensor responds with a value below this threshold. The major challenges comprise identifying those thresholds and finding the dependency between data values originating from sensor and actuator nodes which are parts of the behaviour model. These input values need to be chosen according to the current TC since existing constraints may vary for each individual TC (particular parts of the behaviour are tested). The same applies to the output values, obviously. Note that creating a complex environment behaviour model which reflects the influences from actuator nodes to physical values is not intended and out of scope of this thesis. Concepts for connecting several emulation engines, however, are covered to a specific depth.

Objective III: Test Scalability through Test Case Diversity Optimisation

MBT principles facilitate the functional logic of the SUT to be tested holistically. However, one drawback need to be emphasised: the number of created TCs can often not be executed with reasonable resources and time. Therefore, it is aimed at identifying a relevant group
of TCs that reveal a higher possibility of detecting failures, contrary to randomly selecting a group of TCs out of all possible TCs. Based on the assumption that more diverse TCs do more likely result in failure detection, the existence and applicability of appropriate algorithms is a major research objective. These algorithms show a promising potential to perform better than random selection. The number of possible combinations of TCs that can be selected is considered to be an NP-Hard problem, cf. [138]. This leads to the assumption that a heuristic algorithm can optimise the selection in terms of required resource and needed computation time. For this reason, it is intended to transfer and apply search algorithms to the problem of test case group selection optimisation.

**Overall Objectives and Challenges**

In conclusion, the research objectives of the thesis comprise:

- Concept and realisation of a model-based testing approach for functional testing of IoT-based services.

- Automated test case derivation and execution at an early stage of the service lifecycle by resource emulation.

- Provide scalability for the test automation process by test case group selection optimisation.

The following challenges are emphasised:

- MBT has not been applied to IoT-based services yet and domain specific aspects, such as sensor and actuator nodes, need to be modelled.

- The complexity of test automation needs to be addressed with an adaptive test concept that provides a certain degree of freedom to derive test cases for different types of IoT-based services (atomic and composite) with different behaviours.

- Emulation automation is challenging due to data value generation based on test case specific constraints and interface derivation from the behaviour model.

- MBT tends to result in many test cases; to overcome this drawback, algorithms need to be investigated for optimising the test case selection and executing. Overall, resources and time need to be minimised.
3.2 Hypothesis

Considering the discussed related work and the outlined objectives the research hypothesis reads as follows:

- Simplified and automated testing of IoT-based services at an early stage of the service lifecycle can be enabled by a domain specific test approach that includes automated emulation of external resources into the automated test process.

- Scalability of MBT can be improved with diversity analysis. It can enhance the selection of test cases compared to random selection and achieve a higher failure detection rate. The failure detection rate can be increased without increasing overall test execution time.

3.3 Assumptions and Restrictions

Addressing the complex hypothesis requires to concentrate on the chosen research scope. The following assumptions and restrictions ensure novel findings without oversimplifying the research objectives.

- The approach for testing IoT-based services relies on the service concept provided in Section 2.1.1. It includes the assumption that the communication of the AS and CS is based on request/response messages as provided with REST.

- It is assumed that the SUT can be deployed and re-deployed from the test developer to ensure a known start at the beginning of the test execution.

- The test focuses on functional testing. The SUT is assumed to be able to answer requests within 10 s. If a SUT needs more than 10 s to answer, the behaviour is interpreted as failure. Therefore, this non-functional assumption is implicitly tested without describing this behaviour for each SUT individually.

- The model includes information about valid entities that can be utilised during test execution.

- The infrastructure, such as the runtime environment and underlying virtual and physical machines of the SUT are assumed to work properly. If a failure is detected it is assumed that it is caused by the SUT.
• The proposed emulation approach focuses on concepts for automated testing. Therefore, the goal is to ensure that the model can be tested completely. Provide an emulation model that describe a realistic behaviour of the real world is out of scope. Instead, the integration of external emulation tools is enabled.

3.4 Scientific Contributions

3.4.1 Publications

The findings and concepts which are presented within this thesis have been published at ten conferences, in one book chapter. The publications are grouped into three main categories: Test Framework, Scalability, and Evaluation. Main areas of contribution and the classification of publications are shown in Table 3.1 on page 38. The following subsections highlight the main objectives and findings within the different categories. A description of each publication and the author’s contribution can be found in Appendix A.

Test Framework

To prove the applicability of MBT for IoT-based Services a framework concept is investigated. As a starting point, a common service lifecycle is investigated and the integration of the test process is proposed. It ensures that the order of actions and related information exchange is specified. The service lifecycle includes service annotation steps that can on the one hand be used by service developers to reuse existing AS and on the other hand are also useful for defining the service behaviour models for testing. Based on the service lifecycle management emulation concepts, sandbox testing of ASs and CSs at an early stage is suggested. As a result, a test architecture, an evaluation process, and its realisation are designed and implemented. The following paragraphs highlight the major findings of those test framework aspects and where they are documented.

Lifecycle Management A specified lifecycle defines the required process from designing a service until monitoring the service, which is deployed in a productive environment. Different to previous lifecycle management approaches the foreseen steps are test and knowledge driven. Knowledge annotation is specified in different phases for the utilisation of various aspects such as service development, service deployment, and service monitoring. The service lifecycle is test driven due to re-utilisation of this knowledge with regard to the service behaviour modelling and an early integration of the test process before the service
is finally deployed in the productive environment. The results have been published in [15, 139–141] and are further discussed in Section 4.2.

**Annotation and Behaviour Model Derivation** In order to annotate services with information which can be utilised to derive the service behaviour model and emulation components in a (semi-) automated are discussed within this category. Existing service descriptions such as WADL are therefore extended with semantic annotations for interpretation of data types. Moreover, a concept for resource emulation that can be derived in an automated way with the employment of a rule language is proposed. Results have been published in [140, 142–144] and are specifically presented in Sections 4.6 and 4.5.1.

**Emulation** Concepts for emulating external resources of AS and CS are provided to enable the paradigm of early stage testing within a sandbox environment. The derivation of required emulation components is automated and included into the overall test process. A novel approach for functional testing of ASs and CSs is proposed. Results have been published in [15, 144–146] and explained in detail in Section 4.5.1 for AS, and Section 4.5.2 for CSs.

**Test Architecture and its Realisation** The test automation concept results in a test architecture that can derive and execute TCs in a sandbox environment. The proposed architecture and the related process are realised within a novel testing tool called Automated and Scalable Model-Based Testing Tool for IoT-based Services (AuST). AuST facilitates service logic testing at an early stage of the service lifecycle. Based on a new IoT-based service behaviour model, test cases are automatically created and executed. An automated emulation of sensor and actuator nodes ensures that the service logic can be tested before deploying the service into real world. AuST is also utilised to perform scalability investigations based on test case diversity optimisation. The results have been documented in [15, 139, 140, 145–148]. The overall test architecture is explained in Section 4.4 and its realisation is outlined in Chapter 5.

**Scalability**

Novel algorithms to optimise the selection of a group of TCs are proposed. The algorithms are discussed in terms of their time complexity and compared to a random search approach. While the chosen scalability improvements can be applied to any MBT approach, domain specific requirements for a better data value integration into the similarity investigations are taken into account. Proposed algorithms for test case selection prove that it is possible to reduce the similarity between selected TCs and on the same time require less computation time than random search. In addition, scalability is also addressed by the overall concept of
3.4 Scientific Contributions

external resource emulation. Concepts for automated emulation integration into ASs based on rules highlight further opportunities to achieve scalability. The results have been published in [144, 149]. Section 4.5.1 explains the concepts for scalable external resource emulation for ASs. Chapter 6 outlines the proposed algorithm for test case diversity optimisation and its computation time complexity.

Evaluation

The test architecture is prototypically implemented within the AuST logic. Different experiments with example services demonstrate the concepts of automated test derivation and execution. Furthermore, different aspects of the test case group optimisation are evaluated and general findings are identified. Evaluation results are documented in [140, 149]. Chapter 7 discusses the major findings in detail.

3.4.2 Individual and Collaborative Research

The documented research work has been conducted within the European FP7 project Internet of Things Environment for Service Creation and Testing (IoT.est). The IoT.est [15] project developed an IoT service creation environment whilst bridging the gap between various business services and the heterogeneity of networked sensors, actuators and objects. The approach employs semantic service descriptions to compose IoT services and derive corresponding functional tests semi-automatically. After the manual annotation of a service the service behaviour model is generated automatically. It can be altered manually before the automated generation of test cases begins. A consistent service concept is specified to enable this process. The designed concept of IoT-based service testing relies on the service realisation of the IoT.est project. As part of the project, components for annotating existing services and deploying those services in a runtime-environment have been developed by various project partners. The services and the runtime-environment are utilised to evaluate the applicability of the test concept. In cooperation with Portugal Telecom the project runtime-environment is extended to ensure that the services can be executed in a controlled manner. Therefore, functionalities such as starting/stopping the service and redirection of communication are implemented. This ensured that the service can be set to a known status (e.g., just started) and that the test execution can emulate the external resources by redirecting the communication.

The concept of testing IoT-based services has been designed in cooperation with Mr. Daniel Kümper, University of Applied Sciences Osnabrück. While his research scope focuses on the behaviour derivation from semantic service descriptions [142], the scope
Table 3.1: Scientific Contributions.

<table>
<thead>
<tr>
<th>Paper Title</th>
<th>Lifecycle Management</th>
<th>Annotation and Derivation</th>
<th>Emulation</th>
<th>Architecture and its Realisation</th>
<th>Test Scalability</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Testing of Context-Aware Applications [145]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Validation methodologies for the Internet of Things and its applications: a Survey [147] (submitted)</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Investigation of Opportunities for Test Case Selection Optimisation based on Similarity Computation and Search-Based Minimisation Algorithms [149]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>From Semantic IoT-Service Descriptions to Executable Test Cases - Information Flow of an Implemented Test Framework [139]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Test Framework for IoT-Based Services - A Knowledge Driven Approach [140]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Test-Enhanced Life Cycle for Composed IoT-Based Services [141]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Test Derivation for Semantically Described IoT Services [142]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Test-Enabled Architecture for IoT Service Creation and Provisioning [148]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>How to Test IoT Services before Deploying them into Real World [144]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Test Driven Life Cycle Management for Internet of Things based Services: a Semantic Approach [146]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>A Comprehensive Ontology for Knowledge Representation in the Internet of Things [143]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>A Test-driven Approach for Life Cycle Management of Internet of Things enabled Services [15]</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
of the author’s research concentrates on the later steps of the test process (TC derivation, resource emulation, test case diversity optimisation). Collaborative work has mainly been conducted with regard to service lifecycle management, service annotation (for testing) and data type description and restriction.

The realisation of the test concept benefits from a previously developed prototypical implementation of a testing tool at the University of Applied Sciences Osnabrück, which has mainly been developed by Mr. Marten Fischer and Mr. Rolf Lasch. Their tool tests communication protocols such as Session Initiation Protocol (SIP) based on FSM [82]. While the FSM is sufficient for describing communication protocols it does not have any elements for describing the communication between external resources such as IoT resources. The model also does not represent a concrete implementation but rather describes the capabilities of a communication protocol. In conclusion, alternative paths are allowed. For IoT-based service testing, a deterministic approach is followed where only one possible behaviour of the SUT is correct (based on current states, inputs, outputs, variables, etc). Another limitation is that the test case creation algorithm followed a depth-first search approach which tried to find all paths to the end state. As a consequence, loops are never traversed more than once during the test case creation process. Later on, only the shortest path to the end state is selected for creating test cases. This results in a rather limited number of test cases without a guarantee of state nor transition coverage. Major changes conducted by the author include:

- Extension of models to represent the communication between the SUT and external resources in order to enable emulation concepts.

- The finite state machine is extended to enable data types and data type restrictions based on guards and physical value restrictions. This includes a novel approach for data generation based on individual TCs and ensures the integration of data values in the test case diversity approach.

- Integration of the test case diversity analysis to secure a scalable test process.

- The path discovery has been modified to guarantee state and transition coverage to ensure a known test coverage.

- The process for creating executable test cases has been rewritten to enable resource emulation, automated (re-) deployment and the utilisation of an HTTP protocol adapter (called TTplugin) from Testing Technologie [150].

- Design and implementation of plug-ins for automated sandbox deployment of SUT and test execution visualisation.
Chapter 4

Framework for Automated IoT-based Service Testing

The question how MBT can be applied to the domain of IoT is answered within this chapter. It is explained how IoT-resource emulation can be integrated into the overall automated test process. The proposed test framework for IoT-based services covers aspects of the test integration into the service lifecycle management, IoT-resource and AS emulation, test automation and opportunities for automated model derivation from service description documents.

4.1 Targets and Methodologies

This chapter describes a framework that addresses IoT-based service characteristics during the process of test derivation and execution. The target of the framework is to provide means of deriving functional tests from a service behaviour model and making them executable in a sandbox environment. The framework comprises automated and domain specific aspects of the test derivation and execution process. As far as the AS emulation is concerned, the framework supports an automated approach where the emulated components can be integrated into the test execution and into the test case creation process, respectively. To address these issues, at first a test-driven lifecycle management approach is introduced to support the requirement of early stage testing. In line with the target to provide a concept of sandbox testing, the objective of IoT-resource and AS emulation is discussed and possible alternatives are analysed.

An overall test architecture and a test process are specified based on the previously defined requirements and objectives. Overall, the test process aligns the concepts of sandbox
testing and test automation. As part of the test process, a novel service behaviour model is introduced that reflects the aim of IoT-based service testing with automated emulation integration. Moreover, the model needs to be transformable into standardised test cases. This way, the created test cases can be executed by various tools. Transparency is improved and manual adaptations from test experts can be made, if desired.

The test data generation concept, as one of the steps within the test process, aims at computing sensor and actuator data values. Therefore, the service behaviour model needs to be able to restrict data types not only from software perspective (e.g., distinguish between double and int data types) but also take physical attributes (e.g., data values representing a temperature cannot have a value below \(-273.15\, ^\circ\text{C}\) into account. Furthermore, the data generation aids in ensuring that data values for the AS emulation concept comply to current test case goals. Therefore, each path is analysed individually and additional data value limitation coming from defined guards (e.g., test path where temperature > 20 \, ^\circ\text{C} results in heating switched off) are taken into account. This concept provides to test the service against the complete service behaviour model, which would not be easily achievable without any emulation of connected resources.

Moreover, opportunities are discussed for deriving the behaviour model (semi-) automatically from service descriptions documents, based on an example analysis. The goal is to highlight that methodologies exist, that can enable a (semi-) automated service behaviour model derivation for IoT-based services. However, the relevant approaches are out of thesis scope, as discussed in section 3.4.2.

### 4.2 Test-Driven Lifecycle Management

The concept of IoT-based services and service composition (Chapter 2.1.1) offers the ability to overcome heterogeneous structures of the IoT. With the assumption that a CS can be built upon existing ASs, owned by different entities, a common understanding has to be established – an understanding of how such services are designed, developed, and deployed. Therefore, a lifecycle management is defined to enforce the integration of a well structured evaluation and test methodology at an early phase of the development process.

In order to understand why lifecycle management is important, the paradigm shift of SOA needs to be kept in mind. A SOA enables an improved alignment of business and IT needs [151]. Due to the composition of services, logic can be separated from the implementation. Recent approaches tend to decide how to achieve quality of service, security, or combinations of functions at design time and thus reduce the flexibility of business processes. Another improvement is the reduction of the time to market since the decomposition of applications
into services increases sharing and re-usability of services. Different stakeholders, as well as IoT specific needs, such as the integration of sensor and actuator nodes, require the coordination and collaboration in terms of service design, execution and specifically testing. This results in the need of a common understanding of the service lifecycle process. In addition to these typical outcomes of a well defined lifecycle, a management process enables to (semi-) automatise the test execution based on the defined process and information that describes involved services. Knowledge about the functional and non-functional behaviour of the service is defined within the different phases of the lifecycle. The proposed approach takes advantage of well known phase models. In summary, the explicit integration of testing and testing-related knowledge exchange can significantly enhance the applicability for IoT-based services.

This work assumes that a common understanding of the service lifecycle is crucial in order to build successful IoT-based services [152, 153]. To ensure a knowledge driven service composition and testing approach the annotation process of the service becomes eminent. While previous lifecycle approaches like the classical V-Model or agile programming such as Extreme Programming have already considered techniques like test-first [154] and test-driven development, they do not explicitly describe the process of knowledge annotation. The proposed lifecycle approach overcomes these limitations by adding a clear view of knowledge representation and annotation as well as the process of test derivation from this knowledge.

The detailed steps of the test-driven lifecycle management are depicted in Figure 4.1. Its original purpose is to identify the business process requirements and goals and to categorise them into different lifecycle phases (short term and long term requirements). This categorisation assures a fast ability to demonstrate first results and helps to adjust requirements during the lifecycle process. The step Service Modelling, decomposes the business process and tries to identify possible service components, taking into account that already available services should be reused, if possible. As in all steps, requirements from previous steps are evaluated (in terms of feasibility) and new requirements are identified. Modelling goals ensures the proper identification of the required service components.

Contrary to the first steps, requiring rather manual actions, the Service Creation and Composition phase is supported with tools to discover and compose services. The outcome of this step is a deployable service, including a service and interface description. The next phase takes care of required meta data for service provisioning. This includes semantic descriptions of the service contract and the service run-time environment where the service is to be deployed. With this information, the Test Deviation phase can reason about the semantic descriptions in order to build test cases as well as identify how they should be executed.
4. Framework for Automated IoT-based Service Testing

Therefore, it is the goal to automatise and integrate the test design and execution into the service design time. As a benefit, there is a clear separation between developing and testing.

From the discovered information, further needs are detected within the Evaluation phase. For example, if the number of requests per minute fit to the expectations.

Service monitoring based on current behaviour can result in dynamic reselection of atomic services. In addition, the monitoring phase discovers if the service consumption behaves as expected, for example, if the number of requests per minute fit to the expectations. From the discovered information, further needs are detected within the Evaluation phase.

Contrary to classical approaches, the proposed lifecycle is driven by the semantic description of the service, the service run-time environment and the test environment itself. Therefore, it is the goal to automatise and integrate the test design and execution into the service design time. As a benefit, there is a clear separation between developing and testing.
4.3 Service Composition and Provisioning Architecture

i.e., the developer does not create the test cases explicitly, which might result in a non-optimal
test coverage. Moreover, testing is implicitly integrated since the test cases are automatically
built and executed in a controllable sandbox environment. This results in fast feedback and
rapid service improvement during design time.

4.3 Service Composition and Provisioning Architecture

The accomplished test automation has been realised within IoT.est (cf. Section 3.4.2) and the
high level architecture of the project is therefore illustrated in Figure 4.2. It shows the main
components which are needed to realise the proposed lifecycle for IoT-based services. In
order to enable the creation of composite services, it is assumed that every shape of an AS
can be utilised. It is not required to use specific tools or methodologies as long as the service
has a SOA interface. To make heterogeneous ASs usable within the service composition, they
need to be registered at the knowledge management with a specific semantic and interface
description. The knowledge management offers functionalities to register, discover and
manage services based on semantic and interface descriptions of the services [143].

Based on the information available at the knowledge management the behaviour model
for testing is semi-automatically derived. Afterwards, the tests can be created and executed. If
the tests passes, the service will be marked as tested and can be utilised by CSs. The CSs are
developed at the service composition environment. A human readable goal description of the
service guides the selection of adequate ASs or already developed CSs to be re-utilised. CSs
are created as business processes and can be deployed in the service runtime environment.
For testing a CS the service runtime environment is modified to be utilised as a test instance.
This service runtime environment is called sandbox environment and ensures a separation
between the SUT and external resources. Afterwards, the CS is registered in the knowledge
management. The stored information is the basis to derive the behaviour model for testing
purposed by the test design engine. As the next step, tests are created and executed with the
sandbox environment. Test results are attached to the service information and stored in the
knowledge management. If the tests are passed, the CS is deployed in the productive service
runtime environment. Functionalities to monitor and adopt the service ensure that the service
does not stop working even if one or more connected AS are not longer available. Based on
the knowledge management, alternative ASs are identified and connected to the CS.

4. Overall Test Architecture

Three main components have been designed for the test process to ensure a clear separation
between the different processing phases. The three main components can be described as:

- Test Design Engine (TDE) that is responsible to store the behaviour model of the
  SUT, to enable interaction with the test designer via a Graphical User Interface (GUI),
  analyse the behaviour model, and to derive executable TCs.

- Test Execution Engine (TEE) that executes the tests and ensures that the runtime
  sandbox environment is setup according to the current test case. It also provides
  emulated ASs and IoT-resources based on the specified tests.

- Runtime Sandbox environment that hosts the SUT and manipulates the communication
to ensure that a CS can be tested without the requirement to interact with the connected
  ASs and their links to sensor and actuator nodes.


To address the long term goal of test automation, the domain specific aspects need to be
taken into account. For IoT-based services, heterogeneity of sensor and actuator nodes
hinder an easy adaptation of automation mechanisms already defined. Also, different to other
domains, such as embedded systems, there is no single entity that can control all connected elements easily. The re-utilisation of sensor and actuator nodes already deployed result in shared resources that are not always available for testing purposes. Also, the effort of setting up all connected devices according to the current test objectives can be a challenging and error-prone task.

The goal of the chosen sandbox approach is to automate the process of AS and CS testing by isolating the SUT from connected resources. This enables testing of the service logic at an early stage of the lifecycle. At this early stage, neither the connected IoT-resources nor connected ASs have to be already deployed. In contrast, the service logic can be tested separately without any danger to create physical world changes due to undesired actuation effects. As another important issue, testing the service logic requires a setup of all possible situations that may take place in the physical world. For example, a very high temperature for a fire alarm service. It is obvious that many situations need to be simulated, either by manipulating the physical values, e.g., heating up the temperature, or by injecting emulated sensor and actuator values into the test process.

In difference to previous approaches, the proposed concept does not only ensure that the service logic can be tested completely; it furthermore automated this test derivation and execution process. As a consequence, a button up approach is followed, for facilitating emulation and integrating it into the overall test process seamlessly. As discussed in Section 2.1.1, two different categories of service are to be tested; ASs and CSs. While ASs are connected to IoT-resources, CSs rely on functionalities provided by connected ASs. Therefore, two different emulation concepts are defined to address these two types of IoT-based services.

### 4.5.1 Concept for Atomic IoT-based Service Testing

The author proposes a (semi-) automated process of test code insertion in order to support fast prototyping of IoT-based services with early stage testing. It is postulated that, if the logical interaction between the service and IoT resource is described in a machine readable way (e.g., semantically), the service knowledge can be utilised to build a generic resource emulation interface. This abstracts the heterogeneous interaction with the IoT resource and thus enables fast and scalable emulation of IoT resources from the logical perspective. For example, by including abstract classes or interfaces into the SUT itself, and by integrating a test controller which can select the actual class that should be utilised, the test environment is capable of controlling the interaction of the service with IoT/external resources (not limited to IoT resources) and can imperceptibly (from service logic perspective) replace IoT/external resources with controllable resource emulation components. The approach facilitates testing IoT-based services by inserting emulated IoT resource requests and responses at a logical
level. This approach can be utilised to test how the service acts when facing unexpected but well-formatted inputs and outputs originating from the IoT resources. Hence, it allows rapidly and efficiently emulating critical physical environment situations. The proposed solution of a resource emulation interface, directly controlled by the test framework, reveals a large potential to simplify and accelerate the testing process due to timing issues of the competing solutions.

This approach attempts to decrease the effort for emulating IoT resources by focussing on logical failures, which can occur in the IoT-based service. Therefore, it is suitable for testing the service connected to IoT resources but cannot be used to investigate the IoT resource itself. In contrast to classical virtual resource emulation techniques the followed approach foresees inserting small pieces of code into the SUT itself. This code insertion technique enables capturing and creating messages, which originated from the IoT resources or are directed to it, at a logical level. This way, a significant simplification of the required emulation responses and requests can be achieved. The difficulty of a complex communication interaction (from physical interfaces like IEEE 802.15.4 to communication protocols like 6LoWPAN) can be hidden from the software, hardware, and virtual resources. Moreover, the test framework does not require protocol specific knowledge.

To further explain the chosen methodology, Figure 4.3 illustrates the connections between the TEE, the Atomic IoT-based service and the connected IoT resource. The scope of the testing approach primarily evaluates the SOA interface. The derived behaviour model contains detailed information about interaction possibilities with this SOA interface. What makes it difficult to test is the resulting interaction between the SUT and the IoT resource. If the aim is to emulate the interaction between the IoT resources and the SUT, detailed knowledge about this interaction is required. This knowledge tends to be complex, and is challenging to be transformed into an emulation component. In order to simplify this task, the emulation concept constitutes as hypothesis that these complex interaction can be hidden from the emulated resources if the emulation can be achieved at a logical level.

![Diagram](image)

Figure 4.3: Known and Unknown Interfaces for Testing Atomic IoT-based Services.

In order to transfer this emulation issue to a logical level, the SUT needs to be extended. The dashed lines in the Figure 4.4 indicate adopted and extended components and interfaces
of the SUT, accordingly. The initial model consists of two interfaces: i) a service interface handling service requests and an ii) IoT/external resource interface capable of communicating with IoT/external resources. Without any test related changes the service provides a service logic. Moreover, some class(es) is(are) expected to provide access to the IoT/external resource interface. From the architectural point of view the SUT needs to be extended as follows: The IoT/external resource emulation interface and a test controller component take care of the current test status of the service (test mode, logging mode, or regular mode). In addition, each component that is connected to the IoT/external resource interface needs to be encapsulated (for example, by overloading classes and functions).

After the test cases have been prepared to be executed, the SUT is placed in the sandbox and the TEE starts to run the tests. Figure 4.5 shows a simplified version of the TEE and highlights the components required to utilise the IoT/external resource emulation interface, inserted into the SUT within test preparation process. The TEE provides three interfaces to communicate with the SUT: i) The Communication Interface which is connected to the SUT service interface and used to respond to the test request, ii) the Context Emulation Interface that is connected to the IoT/external Resource Emulation Interface, which receives and responds to resource requests as described within the test cases, and iii) a Log Interface to enable logging of messages transmitted from or to the IoT/external resource in case real resources are connected.

Figure 4.4: Atomic IoT-based Service adaptation for Simplified IoT Resource Emulation.
Figure 4.5: Simplified Test Execution Engine.

Comparision of Test Types, Test Concepts, and Required Efforts

The outlined approach of forming a controllable resource emulation interface is compared with the approach of physical and virtual IoT resources involved in the testing process within this section. Table 4.1 summarises the concept comparison. The different concepts are assessed with regard to their capabilities \( (\text{Cap}) \) and required efforts \( (\text{Eff}) \) to enable test cases of different test objectives (test types). For this comparison four different test types have been identified: Hardware test cases are capable of detecting if the unexpected behaviour of the SUT is based on hardware failures (e.g., flash storage failures, internal bus errors); Communication test cases investigate if the protocols involved in establishing a communication link are working as expected, while the Logical test cases cover possible failures resulting unexpected behaviour of SUT caused by logical failures (e.g., data exchange result in unexpected SUT reaction). Test cases for Scalability evaluate if the service is still working correctly in case the number of attached IoT resources are increased.

Hardware test cases can only be realised in cooperation with physical IoT resources but result in high efforts of either programming and designing a hardware debug interface or utilisation of hardware analysers. Communication test cases can be provided by either physical or virtual IoT resources. Both approaches are capable of evaluating if the communication is working as expected – although, due to the visualisation, it is much easier to investigate the behaviour.

While the test types Hardware and Communication are relevant to test the IoT resource functionality itself, these tests are not required to test the IoT-based service. Logical test cases can be realised with all three approaches compared above. However, the capabilities of physical IoT resources to process Logical test cases are limited due to the restricted possibility to manipulate the execution process.

High effort is needed to manipulate data values (e.g., temperature values) either by manipulating the environment (e.g., heating the room) or implementing a strong manipulation
interface. Virtual IoT resources have more potential for manipulation but still result in challenges as far as building the required manipulation interfaces is concerned. The presented approach of a resource emulation (Res. Emul.) interface reveals the best characteristics of this comparison. Due to the concentration of the logical level, the simplicity of the interface enables a highly flexible solution with low implementation effort. This advantage becomes also visible if Scalability is under investigation. While physical IoT resources are not easily available and manageable at a large scale, virtual IoT resources as well as the resource emulation interface concept expose high capabilities for scalability since hardware components are not involved. Compared to other solutions, the simple structure of the resource emulation interface decreases the required effort. In conclusion, the test types relevant for IoT-based service testing are realised with the proposed resource emulation interface.

### 4.5.2 Concept for Composite IoT-based Service Testing

Composite IoT-based services differ from the characteristics of ASs. While ASs provide access to sensor and actuator nodes functionalities through SOA interfaces, CSs rather concentrate on control and access elements of the physical environment via connected ASs. They do not necessarily provide a SOA interface that can be accessed from a user and this results in a limitation of what can be tested. The behaviour of a CS can often only be detected by the behaviour of the encapsulated actuator nodes (e.g., switching actuation). In order to test such a CS, at least four different approaches exist:

- Passively monitor communication between CSs and connected ASs and reason about this communication.
- Manipulate physical sensor values by changing the physical environment with actuators (e.g., heating up the temperature for temperature sensor).

<table>
<thead>
<tr>
<th>Test Type</th>
<th>Physical</th>
<th>Virtual</th>
<th>Res. Emul.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cap</td>
<td>Eff</td>
<td>Cap</td>
</tr>
<tr>
<td>Hardware</td>
<td>high</td>
<td>high</td>
<td>NA</td>
</tr>
<tr>
<td>Communication</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Logical</td>
<td>med</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Scalability</td>
<td>med</td>
<td>high</td>
<td>high</td>
</tr>
</tbody>
</table>
• Emulate sensor values by data injection into the AS or its IoT-resource.

• Emulate the connected AS entirely.

The first option, to passively monitor the behaviour of the interaction between the CS and the connected AS, hinders fast testing since it is unclear how much time is required to completely test the specified behaviour of the CS. The reason is that the CS can only react to the values detected by the encapsulated sensors and since they are not manipulated, it is rather unclear if all possible situations have been monitored. Especially situations that might not occur very often can not be tested easily. However, these situations need to be checked before deploying a CS into a real world physical environment. Besides, to benefit from the capabilities of model-based testing, it is the aim to control the test execution in a way that the actual test cases control what should be tested next, by actively stimulating the SUT. As a consequence, the capabilities of manipulating the physical values or the detected sensor values significantly enhance the potential of a model-based testing approach. As discussed in Section 4.5.1, changing the physical value through actuation is challenging and possibly requires a lot of resources. For physical value manipulation through actuation it is required to synchronise this process with the test case currently being executed. Moreover, changing physical values requires a specific amount of time that may depend on the physical value (e.g., time to heat up the room). Therefore, this approach can be used as a final check-up before deploying the CS in the real world – but would not be very resource and time efficient if applied at an early stage of the service lifecycle where the SUT might still evaluate. On the other hand, if timing behaviour is focused on, it might be required to perform the final testing with real AS and manipulated sensor detection (e.g., fire scenarios at an airport).

Injecting data values into the AS, or the connected IoT-resource, requires access to already deployed AS in a test mode. This might not be possible since other already deployed CS may also rely on these AS. Also, it is the aim of an IoT architecture that different entities participate, implicitly forming a heterogeneous network and ownership. Therefore, the utilised AS might be accessible but can not be set in a test mode from a CS developer or CS tester.

To overcome these drawbacks, the most isolated approach is to emulate the whole connected AS. Based on the assumption that the behaviour of the AS is known by the service developer (otherwise it can not be re-utilised), this knowledge can also be utilised to emulate those components. This emulation ensures a clear separation between SUT and connected resources. It can facilitate service testing at an early stage due to a safe execution, i.e. the physical world is not affected. The test phase can be started even if some or all of the required ASs have not been fully developed or are not accessible, e.g. due to lack of access rights
or a delayed development. Furthermore, testing can be significantly shortened due to the possibility to emulate the required AS behaviour faster than in real time and the possibility to test specific situations that might seldom occur in the real world.

Different to the concept for AS testing, CS testing in a sandbox environment might not require to change the CS itself. The reason is that the connected ASs are well known since their SOA interface is already described for CS development and AS testing. Therefore, the required encapsulation can be achieved by manipulating service endpoints (e.g., Uniform Resource Locator (URL)) either dynamically in the runtime environment during execution – in case this is feasible –, or by changing the CS itself before testing. It is required to control the communication between the CS and the connected ASs and redirects it during test execution. Figure 4.6 shows the main interfaces of the runtime environment and the required adaptation. While the CS itself does not necessarily provide a service interface that can be accessed by a user – and hence directly tested –, it provides at least one service interfaces to ASs. A test controller, either placed in the runtime itself, or in the SUT, is needed to encapsulate the communication between the CS and connected ASs and redirects the traffic to the TEE.

![Figure 4.6: Composite IoT-based Service Deployed in a Sandbox Runtime Environment During Test Execution.](image)

Due to the overall objective of an automated test process, the emulation itself needs to be automated as well. Based on the available knowledge about the connected AS it is proposed to consolidate this emulation into the test process by integrating the emulation directly into the created test case. This methodology simplifies the test execution synchronisation since no external component is required.

### 4.6 IoT-based Service Annotation and Model Derivation Opportunities

Service-oriented paradigms show considerable potential to be one of the key drivers for IoT enabled applications. In the future, the concept of re-using existing services for compos-
ing richer services and applications may constitute a novel ecosystem for large scale IoT deployments that are still missing today. Due to the heterogeneous nature of IoT devices and expected abstraction services, it is required to exchange complex information about possibly involved components and their interaction capabilities. From a testing perspective, this offers the opportunity to derive a behaviour model from the service descriptions (semi-) automatically. In the following, one possible approach to enable an automatic model derivation from service description for IoT-based services is presented. Following the goal to demonstrate that a rule based approach can be utilised, recent work by Ramolari et al. [73] is extended.

4.6.1 Model Derivation Opportunities for Atomic IoT-based Services

In the applied service design concept (cf. Chapter 2.1.1) the AS rely on a RESTful service design concept due to their simplicity and flexibility. A common description of such a service is based on WADL. For testing purposes, this description already provides useful information for the behaviour model derivation. Amongst others, the following elements are included:

- **Resource** elements that reflect different functionalities/interfaces of the service.
- **Param** elements that can describe inputs either as basic data type or as more complex reference to external XML Schema Definition (XSD) descriptions.
- **Method** elements that describe requests and response types (e.g., HTTP GET/POST).
- **Response** elements that specify the service response.

What is missing for IoT-based atomic service testing is a native capability to express a stateful service behaviour. While RESTful services interact statelessly by definition [155], a request might result in a state change of the service that is directly visible to all of its clients. In other words, a multi message client-server session is lacking. In the atomic service example outlined in Section 2.1.3 on page 13, the adjustment of the `pan`, `tilt` or `zoom` directly changes the camera adjustment. Any client that requests the camera position will receive the changed values. For testing, the connections between the corresponding interfaces need to be declared.

One possibility to overcome this limitation is to define the Input, Output, Precondition, Effect (IOPE) (other approaches are discussed in Section 2.5 and [75]) by utilising rules for describing the preconditions, that need to be fulfilled to invoke the service and the resulting changes after the invocation occurred (cf. [75, 156]). Ramollari et al. [73] propose utilising production rules. Production rules can be classified as reactive rules, which specify one or more actions that are executed if specific conditions are satisfied [157].
Besides of the problem of required model capabilities for stateful behaviour, the IoT-resource emulation concept, as discussed in Section 4.5.1, requires knowledge about how requests received over the RESTful interface result in communication with external/IoT resources. Following the goal to demonstrate that a rule based approach can be utilised, recent work by Ramolari et al. [73] is extended. Specifically, the communication between the AS and the IoT-resources is included. For demonstration simplification, the extension is realised based on the already utilised methodologies, i.e. based on stream X-Machine, and based on WSDL instead of WADL. However, the approach could have been realised with WADL and other forms of an EFSM as well.

**Service Behaviour Model Definition Based on Rules**

**Example: Stateful IoT Service**  The service under investigation performs elementary lookup functionalities. It allows for requesting values from embedded sensors via a service interface. The service interface offers one operation: `getValue` and provides two operations enabling communication with the IoT resource: i) `register` ii) `getResourceValue`. Furthermore, there is one internal timer event: `unregistered`. For simplicity, it is assumed that an IoT resource entity is created during the initialisation of the service and is marked as unregistered as long as the IoT resource does not use the register functionality. If an IoT resource has been registered previously, a consumer can request the IoT resource value (e.g. sensor data), by querying the service interface. If the IoT resource does not re-register itself during a specific time period, the IoT resource entity is marked as inactive and service consumers will no longer be able to request the IoT resource values. Listing 4.1 represents the encoded precondition and effect of the IoT-based service example. For clarity the Rule Interchange Format Production Rule Dialect (RIF-PRD) is depicted with an abstract syntax. An XML syntax with an associated semantics is available as well.

The presence of a semantic model that describes the service is assumed. The model specifies input, output and state-related attributes of the services. Table 4.2 lists entities that need to be described by an ontology. The description of preconditions and effects with RIF-PRD together with the semantic model for the input and output parameters can leverage building a model of the service interface. The next section explains how and why stream X-Machines are utilised to describe the model of the service.

**Example: Model Representation**  A streaming X-Machine is capable of representing the data and the control model of a system. It can be seen as an extension of the Finite State Machine with an attached complex data model, which is forming a memory model. In addition, transitions are represented by processing functions instead of input symbols.
For all \( ?resource \rightarrow ?status \rightarrow ?value \rightarrow ?request \)
And ( ?resource#Resource
\( ?resource[\text{hasStatus} \rightarrow ?status] \)
\( ?resource[\text{hasValue} \rightarrow ?value] \)
\( ?request[\text{#GetResouceValueRequest}] \)
If ( External (pred:matches(?status 'S2'))
Then Do (Retract (?request)
\( \text{Assert New (?response#ResourceValueResponse)} \))
\( \text{(* wsdl:operation register *)} \)
For all ?resource ?status ?lastseen ?request
And ( ?resource#Resource
\( ?resource[\text{hasStatus} \rightarrow ?status] \)
\( ?resource[\text{hasLastseen} \rightarrow ?lastseen] \)
\( ?request[\text{#GetRegisterResourceRequest}] \)
If Or (External (pred:matches(?status 'S1'))
\( \text{External (pred:matches(?status 'S2'))} \)
Then Do (Retract (?resource[\text{hasStatus} \rightarrow ?status])
\( \text{Assert (?resource[\text{hasStatus} \rightarrow 'S2']} \))
\( \text{Retract (?resource[\text{hasLastseen} \rightarrow ?lastseen])} \)
\( \text{Assert (?resource[\text{hasLastseen} \rightarrow nowDateTime])} \)
\( \text{Retract (?response)} \)
\( \text{(* wsdl:operation unregister *)} \)
For all ?resource ?status ?lastseen ?event
And ( ?resource#Resource
\( ?resource[\text{hasStatus} \rightarrow ?status] \)
\( ?resource[\text{hasLastseen} \rightarrow ?lastseen] \)
\( ?resource[\text{hasEvent} \rightarrow ?event] \)
\( ?event[\text{#unregisterEvent}] \)
If And (External (pred:matches(?status 'S2'))
\( \text{External (pred:dateTime \rightarrow greater \rightarrow or \rightarrow equal(?nowDateTime (?lastseen+360)))} \)
Then Do (Retract (?resource[\text{hasStatus} \rightarrow ?status])
\( \text{Assert (?resource[\text{hasStatus} \rightarrow 'S1']} \))
\( \text{Retract (?request)} \)
\( \text{(* wsdl:operation unregister *)} \)
\( \text{Retract (?response#UnregisterResponse)} \)
\( \text{Assert (?response[\text{hasMessage} \rightarrow 'Resource unregistered']}) \))

Listing 4.1: Rule Description Example.
The processing functions utilise input streams as well as read and write memory values by producing an output stream [158]. In the following sections it is briefly outlined how the rules shown in Listing 4.1 can be utilised to build a Stream X-machine representing the SUT.

State Variables The data model of the Streaming X-Machine can be utilised to represent the state variables of the IoT-based service example. State variables contain the attribute that they can influence during the input and output transformation process. Therefore, the state variables appear in the action part of the rules. The reserved words for \(?request\) (Input) and \(?response\) (Output) are ignored. In the example the identified variables are: \(\text{hasStatus}\) and \(\text{hasLastSeen}\). State variables are limited with regard to their range of validity. One limitation is defined by the Datatype properties (cf. Table 4.2) of the ontology (e.g., enumeration, nonNegativeInteger). On the other hand, state variables may also be restricted by the precondition sections of the production rules. In the service example the state variables have the following partition:

- \(\text{hasStatus}: S1, S2\)
- \(\text{hasLastseen}: \geq \text{nowDateTime} - 360, < \text{nowDateTime} - 360\)

Identifying Preliminary State The preliminary states constitute the summary of the partitions of the state variables. In our example this results in four pre-states, namely:

- \(S1 \geq M360, S1 < M360,\)
where $M360$ stands for $\text{nowDateTime} - 360$.

**Determining Input and Outputs** The next step is to identify the input and output functions. In the production rules the reserved words $\text{?request}$ and $\text{?response}$ allow for identifying these functions. In addition, our example has an event-based input, which is identified by the variable $\text{?event}$. This might not be a standard conform assumption but it is expected that event-based computation could be described based on extensions of RIF-PRD. It is out of scope to discuss a fully valid model for event-based transactions. The inputs are defined as follows, including events as input:

- $\text{GetResourceValueRequest}$
- $\text{GetRegisterRequest}$
- $\text{UnregisterEvent}$

The outputs comprise:

- $\text{ResourceValueResponse}$
- $\text{RegisterResourceResponse}$
- $\text{UnregisterResponse}$

**Determining Transition Pre-States** The input results in the execution of a processing function if the preconditions are satisfied during the pre-state phase. With mathematical deduction the valid pre-state values of the inputs are:

- $\text{GetResourceValueRequest}$: $S2 \geq 360, \ S2 < 360$
- $\text{GetRegisterRequest}$: $S1 \geq 360, \ S1 < 360, \ S2 \geq 360, \ S2 < 360$
- $\text{UnregisterEvent}$: $S2 < 360$

**State Merging** If one input is acceptable for more than one pre-state, the states can be merged. In our IoT-based service example the pre-states $S1 \geq 360, \ S1 < 360$ are both acceptable for any input. The resulting states are:

- $S1$: (hasStatus = S1, hasLastseen = *)
- $S2$: (hasStatus = S2, hasLastseen < nowDateTime - 360)
Table 4.3: Transition Pre-States and Post-States

<table>
<thead>
<tr>
<th>Input</th>
<th>Pre-State</th>
<th>Effect</th>
<th>Post-State</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetResourceValueRequest</td>
<td>S2</td>
<td>–</td>
<td>S2</td>
</tr>
<tr>
<td>GetRegisterRequest</td>
<td>S1, S2</td>
<td>hasLastSeen ← 0</td>
<td>S2</td>
</tr>
<tr>
<td>UnregisterEvent</td>
<td>S2</td>
<td>hasStatus ← S1</td>
<td>S1</td>
</tr>
</tbody>
</table>

Figure 4.7: State Diagram.

**Determining Transition Post-States**  
Similar to the approach of identifying the pre-states, post-states are identified by investigating the effect of the processing function execution. If more than one pre-state is acceptable for one input, but not for all inputs, two different processing functions are created (in the example GetRegisterRequest1 and GetRegisterRequest2). The resulting transitions are outlined in Table 4.3. Finally the resulting finite state machine of the Stream X-machine is shown in Figure 4.7.

The deduction of test cases from the identified stream X-Machine results in test cases for each processing function (cp. Figure 4.7). One of the major challenges involving physical IoT resources is the requirement to control the resources functionally but also timely. In order to execute the test case GetResourceValue the SUT needs to be put into the state 'S2'. Therefore, the following procedure is required: i) initialise SUT ii) try the input GetResourceValueRequest (failure expected) to assure that the current state is 'S1', iii) invoke GetRegisterRequest1, for state change, and iv) send input GetResourceValueRequest (expect a positive response). Based on different test coverage criteria the procedure could be different. This approach empowers a state and transition fault coverage [159].

### 4.6.2 Testing Opportunities for Ontologies

Different to integrating missing test information into the service description document itself, as discussed during the previous section, another approach is to utilise ontologies. The most
common definition of term ontology is being given by Gruber: 'An ontology is a formal explicit specification of a shared conceptualisation for a domain of interest' [160]. Due to their extendibility and linkability to already developed concepts, ontologies are a good candidate to structure and represent knowledge. Knowledge representation as defined by Bench-Capon 'is a set of syntactical and semantic conventions that makes it possible to describe things' [161]. Other approaches to structure and represent knowledge are frames, rule-based such as outlined in the previous section, Logic and Predicate Calculus, Model-based, cf. [161]. In the domain of IoT, the concepts of semantic web seem to fit perfectly. While semantics for pure web services have not prevailed so far, the IoT domain seems to be a much more suitable candidate, due to the fundamental need to exchange information about participating components. Within the IoT.est project [15] OWL-S is used as the primary language for IoT service description. Profile, Process and Service ontologies in the Semantic Markup for Web Services (OWL-S) [162] are used to describe the functionalities and processes for IoT services. The work employs concepts defined in the OWL-S [18] to specify the service test semantics, including inputs, outputs, preconditions and effects (named IOPE).

The description of the test model within a test ontology is one opportunity to structure the knowledge that is required for automated test case definition and execution. An ontology can also define different types of tests and potentially store test results. The Test Description Ontology enables the definition of re-usable test cases and the modelling of the test process.
The description ontology defines atomic tests and composite tests, distinguishing different types, e.g., functionality or reliability, and levels of tests, i.e. unit, interface, integration, or collaboration test. Service attribute values are constrained by a minimum (Min) and maximum (Max) value or a Value List and an optional Default Value. Moreover, Timers are defined to describe transition of states and events. Figure 4.8 shows a part of the test component of the description ontology. Description of the service test facilitates automated test generation for the SUT. Reasoning engines, e.g., rule based systems, can exploit the knowledge to derive the service behaviour model and constrain the test cases. To the author’s best knowledge, ontologies have not yet been applied for deriving an IoT-based service behaviour model. While the proposed Test Description Ontology is a first trial to structure test related information, it is identified that the required information for automated service behaviour model derivation exceeds the complexity of the targeted ontology. It is an open research question if ontologies can provide the required detail knowledge in a scalable and efficient way.

4.6.3 Composite IoT-based Service Model Derivation Opportunities

Composite IoT-based services are described based on the BPMN (cf. Section 2.1.1). The included business logic can be reasoned to create a service behaviour model. Elements such as Invocations, Receptions can be identified to model transitions and states. The involved ASs, required for executing in the composite service, are included as service endpoints and are referenced to the corresponding WADL documents. As a consequence, all information to derive the service behaviour model is already available within the describing documents of a composite service and the ASs connected to it. The data also includes all information needed to emulate the behaviour of the connected AS due to their link to the WADL description documents. As one of the drawbacks of automated service behaviour derivation, it might not be possible to detect if the service reacts unexpectedly if the BPMN already describes an unwanted service behaviour. In this case the tests would run against a wrong description and might not detect the failure. Due to the characteristics of model-based testing, the service behaviour can only be verified with the service behaviour model, and in case this model is already wrong, the test might not be able to detect failures. Therefore, automation model derivation might need to be extended by manual interventions from a test developer or needs to include other information sources such as high level goals (e.g., goal of service is to set a room to a stable temperature) to avoid strong dependencies between the service behaviour model and the BPMN document.
4.7 Test Process Concept

A defined test process guides the realisation of the developed AuST testing tool. Individual steps are identified and general needs and concepts are proposed. Figure 4.9 shows the main discover process steps as defined components. As part of the defined TDE the IoTBM represents the behaviour of the SUT that is under investigation. It allows to designing a behaviour model from the service perspective. The Path Finder then analyses the model and derives paths out of the model. A path contains of an ordered sequence of model elements that can be validated during test execution subsequently. The Test Data Generator analyses all found paths and generates data values for all data types, that are present in the elements of the paths. With the inclusion of data values into these paths, paths are called TCs. The data value generation includes data type restrictions as well as individual path constraints based on guards. The Test Data Generator is also utilised to realise the emulation of connected ASs. The Diversity Analyser optimises the selection of TCs based on test case similarity investigations (cf. Chapter 6). The Test Case Creator then reasons about the TCs elements and transforms them into executable TCs. Before test execution, the TCs are compiled with the Test Case Compiler. Afterwards, the compiled TCs are forwarded to the TEE. As part of the test execution, the SUT is deployed in a sandbox environment.

The following subsection will describe the main functionality and their novelties of the test process.

![Diagram of Test Design Engine with External Interfaces and Main Components](image)

**IoT-behaviour model** The automated test approach is built upon a behaviour model called IoT Behaviour Model (IoTBM) which covers the characteristics of the service logic of the SUT, enables automated emulation of connected components, as well as addressing the needs for creating executable TCs. A behaviour model is defined as direct or indirect detectable reaction of the SUT to given stimuli or internal events (such as timers) and allows creating
4.7 Test Process Concept

a model from the service perspective. As identified in Chapter 2, EFSM offer required capabilities to reflect the service logic of IoT-based services in a way that is easy to model and understand by test developers but can also be derived in an automated way (as discussed in Chapter 4.6). An EFSM model is therefore utilised to describe the behaviour of the IoT-based services. The goal of the proposed IoT-based service behaviour model is to prove that a model exists that can enable an automated test case creation process with the followed automated emulation concept. Other possible approaches, such as LTS or Stream-X machine, exist and might be applicable to describe IoT-based services. Further empirical analysis is required to make quantitative assertions which is the best model in terms of flexibility, scalability, and usability, etc.

The following characteristics of the service logic are needed within the IoT behaviour model to enable the automated test approach:

**States:** representing different logical points of the service and limiting the number of correct functionality.

**Events:** characterise starting of an activity, which might result in actions or a state change. Events can be either from the type timer or input message.

**Actions:** describe the reaction of the system to an event. It can be either a response message (Output) or can result in a request send to an external component (e.g., sensor node).

**Guards:** optional parameter describing a condition for a state change (e.g., if a threshold is reached).

**Memory:** ensures that variables (e.g., values requested from actor and sensor nodes) can be cached in the SUT model and can be used in a later state or action.

**Variable Types:** define the range of possible values and their initial values after initiation of the SUT. In addition, physical value data types (e.g., temperature values) are required to identify that there is interaction with the environment.

Further elements are needed to enable the discussed automated emulation concept:

**Communication Roles:** describe if the SUT currently acts as server or client. This is required to identify, if the SUT is expected to respond to a message or send a request message.

**Message Directions:** ensure that communication and their direction between CSs and ASs can be distinguished from other service interfaces. With this extension, components that should be emulated can be easily identified.
In contrast to most of the known EFSM models, which are rather abstract and can not be utilised to derive executable test cases without extensions, the proposed model includes the following extensions to ensure automated derivation and execution of the derived TCs:

**Message Types:** representing different known message types. This ensures that abstract Events and Actions can be transformed into protocol messages such as HTTP POST/Get request/response messages which can be send/received during test execution.

**URL:** describing the service endpoints to be tested.

The proposed behaviour model shows that EFSM models can be utilised to describe IoT-based service behaviour. The identified model extends known behaviour models to enable testing IoT-based services at an early stage. The major extensions include:

- Elements to model the control characteristics of IoT-based CSs (e.g., controlling actuator nodes with sensors). It describes among other, different communication roles and message directions. Different to reactive systems and web-services the only visible interaction of the IoT-based CS may be the interaction with connected sensor and actuator nodes. The described model ensures that these interactions can be tested.

- Elements to automate the process of test case creation. The aim is to emphasis an automated test case creation and execution process, including automated AS emulation, which is not possible with known models. In difference to the already established EFSM behaviour models the model not only covers logical aspects but also provides elements for the actual realisation. The behaviour model ensures that TTCN-3 elements can be created automatically and is therefore more applied than known behaviour models.

**Path finder** Analyses the model and derives paths from the defined behaviour model. A path consists of an ordered sequence of model elements that can be validated during test execution subsequently. In order to demonstrate the overall test automation process a rather simple methodology called W-method is selected that can guarantee full state and transition coverage. Other approaches including Wp, HSI (cf. Table 2.2) exist but have not been considered since it was not the goal of the work to select the best path finder methodology. In contrast, it was rather the scope to demonstrate the overall automation approach, which requires at least one path finder.

In order to apply the W-method to the behaviour model, described within the last subsection, four assumptions need to be fulfilled:
• IoTBM is *minimal*: no *State* exists that cannot be distinguished from another *State* based on a *separation sequence* (input sequence), which can distinguish every pair of *States* by the output reaction. In our model inputs can be expressed as *Events*: and outputs as *Actions*.

• IoTBM is *completely specified*: for every possible *Event* at each *State* there exists a known *Action* reaction.

• IoTBM is *strongly connected*: every *State* is reachable from each other *State* by one or more transitions. One possibility to overcome this requirement is to include a reset message into the model to set back the IoTBM to the initial *State* but have not been implemented within this test tool.

• Events do not change during testing: The IoTBM can respond to all allowed *Events* at any time.

For state and transition coverage it is needed to identify a characterisation set (also called W set), which is a set of input sequences that can be utilised to distinguish every pair of existing states in the model. The resulting paths are created by concatenating every sequence from the transition coverage set with every sequence in the characterisation set and apply them after the SUT is initiated (cf. [94]). The number of paths that are created from the W-method can be expressed for the integers for \( N_S \geq 2, N_E \geq 2, N_A \geq 2 \) as follows:

\[
N_{Paths} = (N_W \cdot (1 + N_E)) \cdot N_P
\]

with \( N_W = N_E \cdot N_A \) and \( N_P = N_S \cdot N_E \)

\[\Rightarrow N_{Paths} = N_E^2 \cdot N_A \cdot N_S \cdot (1 + N_A)\]

for \( N_{S_{\text{max}}} = N_A^{N_E} \) (Partial permutation with repetition)

\[
N_{Paths_{\text{Max}}} = N_E^2 \cdot N_A^{(1+N_E)} \cdot (1 + N_A)
\]

where \( N_{Paths} \) = the number of created paths, \( N_W \) = size of the W set, \( N_E \) = number of events, \( N_P \) = size of the transition coverage set, \( N_S \) = number of states, \( N_A \) = number of actions, \( N_{S_{\text{max}}} \) = maximal distinguishable states for a given number of events and actions, \( N_{TCS_{\text{Max}}} \) = number of test for \( N_S = N_{S_{\text{max}}} \).

The Equation 4.1 shows the dependencies of how many paths are created with the W-method based on the number of actions, events, and states defined in the IoTBM. The maximum number of allowed states, which results in a minimal IoTBM for a given set of actions and events can be expressed as a so called *partial permutation with repetition*. If the
maximum number of states is present in the model, the number of paths grows exponentially. Otherwise, the number of paths grows quadratic with the number of actions and events and linear with the number of states.

**Test Data Generator** Test Data Generator analyses all found paths and creates data values for all data types, that are present in the elements of the paths. With the inclusion of data values into these paths, paths are called TCs. Data values can be present within sent and received messages, as internal variables or as part of guard conditions. To ensure that data values can be created it is required to generate the data values according to:

- defined data types (e.g., integer, double, etc.),
- their constraints due to the representation of physical values (e.g., temperature $\geq -273.15^\circ C$), or sensor detection limitations (e.g., temperature sensor can only detect values between $-10^\circ C$ and $+60^\circ C$), and
- service logic dependencies between received/sent data values, internal operations and previously sended/received values.

The first two characteristics are statically defined within the IoT-behaviour model. Every occurrence of data values within messages or the service memory is defined with a data type and their restrictions. The third requirement can not be solved statically. Instead, it is needed to evaluate each path individually and identify possible constraints. To illustrate this, the flowing example shows a simple CS:

**Example 4.1:** A service that turns off a heater in a room as soon as a window in the room is open can be modelled with a **Guard** (window status == 'Open'). The response is stored in a **Variable** (with an **Action**) and this **Variable** is included into a **Guard**. The condition operation is set to **equal** and the condition to 'Open'.

The example service can be modelled with the behaviour shown in Figure 4.10 and Table 4.4 on the facing page. As an example, it is assumed that data values are created for the following path: $S1 \rightarrow T1 \rightarrow S1 \rightarrow T2 \rightarrow S2$. In case this path should be tested, it is at first expected that an internal timer (as part of transition $T1$) expires and as a reaction a request message is send to an external AS (window service). Depending on the answer of this atomic service the SUT acts differently. As part of example path, it is the goal to detect that the SUT sends a request to stop the heating (transition $T2$). Therefore, the guard statement 'windowStatus equals Open' need to be true. The guard statement contains tree
4.7 Test Process Concept

parts: i) variable (‘windowStatus’), ii) condition (‘equals’), and iii) a value (‘Open’). As a consequence, the involved variable ‘windowStatus’ is discovered backwards within the path. During the discovery, the variable is detected as part of the action ‘store response value in windowStatus’ (transition T2). Afterwards, the variable is linked to the event ‘response from window service’ (transition T2). With the identification of this message the generation of an adequate data value can be started. The concept of emulating external ASs enable the capability to send the required data values within the emulated message. Therefore, the next step is to combine the data value constraints from the data type (here the data type enumeration would be a good candidate), its restrictions (lets assume that only ‘Open’, and ‘Close’ are valid enumeration values), and the condition from the guard (‘Open’). For this particular case, the only valid data value is therefore ‘Open’.

![Figure 4.10: State Behaviour Segment of Example Service One.]

Table 4.4: Abstract Transition Description of Example Service One.

<table>
<thead>
<tr>
<th>TransitionNr.</th>
<th>Event</th>
<th>Action</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Timer</td>
<td>Send request to window service</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>Response from window service</td>
<td>Store response value in windowStatus</td>
<td>windowStatus equals ‘Open’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Send stop heating request</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>Response from window service</td>
<td>Store response value in windowStatus</td>
<td>windowStatus unequal ‘Open’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Send request to temperature sensor</td>
<td></td>
</tr>
</tbody>
</table>

The proposed approach is not limited to enumerations and can be applied to numerical values as well. Also multiple variables can be traced to generate the valid data values. In case that the relation from guards variables to previous operations or transmissions can not be resolved, it can be assumed that those paths can never be executed. This means that either the SUT contains functionalities that can never be reached (dead code) or the model does not represent the behaviour of the SUT. In both cases, the data generation process can identify this circumstance as a failure.

Compared to approaches that create the data values not until test execution (Fuzzy testing [12, 163, 164]), the proposed approach leads to known test values before test execution is started. A test developer is therefore capable to review and comprehend the automated results and adopted them if needed (e.g., due to experience etc.). Another important benefit of this
early data value generation is, that the created data values can be taken into account for test case diversity analysis, discussed during the next paragraph. It ensures that diversity of test cases can be also computed between the involved data values and this has the potential to enhance the test case selection performance.

**Diversity Analyser**  The common approaches to derive TCs from a behaviour model result in a huge number of TCs, which cannot be executed within a reasonable time without high time and computation costs [9]. Therefore, it is eminent to identify which TCs and which test data should be selected for execution to ensure the best target test coverage with the given time and resources. As a consequence, the process of selecting test data and TCs, which have the highest possibility to identify failures, is crucial for a successful application of model-based testing paradigms for IoT-based service testing. This work follows a similarity investigation approach and identifies the most diverse TCs for test execution based on a pairwise similarity between all TCs. This approach can improve the selection of TCs, if parts of the TCs have redundancies and the removal of these redundant TCs have a lower impact to the fault detection rate than randomly removing TCs. The selection of a group of TCs based on pairwise similarity scores result in a NP-hard problem [138] (set cover problem). Therefore, specific search-techniques are investigated for enhancing the performance compared to random search with similar or lower computation time effort. A detailed description of the diversity analysis can be found in Chapter 6 on page 87.

**Test Case Creator**  During the test case creation phase, abstract TCs are transformed into executable TCs. Each TC path is sequentially examined and required elements are created. The creation process includes the automated emulation of connected AS (for CS testing) and the definition of expected SUT behaviour reaction (e.g., message transmissions). The main challenge is to define this process in a way that every possible combination of behaviour results in valid test cases that can be executed. Also, a lot of syntactical details need to be taken into account and this makes the process hard to realise. The followed approach is therefore based on a two step approach: at first it is reasoned about the path elements, required elements are structured and syntactical details are avoided within this step. In the second step those objects are transformed into TTCN-3 defined syntax. The creation of executable test cases is driven by implementation needs and the demand to prove the applicability of the defined IoT-based behaviour model for automated model-based testing with automated emulation capabilities of external resources.
Test execution In order to demonstrate the validity of the proposed test derivation process the test execution is realised. For this cause, two components are defined and interconnected. The Test Execution Engine ensures that the test execution process can be executed and controlled. It is not in the scope of the work to create those test execution tool. Instead, an existing tool that can execute TTCN-3 is selected for the test execution and extended to support the automated test approach. While other test tools exists, which can execute TTCN-3, the TTworkbench from Testing Technologies [14] is selected due to their comprehensive functionality and their extendibility.

As part of the test execution, the SUT is deployed in a sandbox environment. The aim is to ensure that the SUT can be deployed in a controlled environment. The controlling includes the capability to start/stop and reset the SUT as required by the current test case. This is needed since the SUT needs to be set in a known state before each test case can be started. A second control mechanism ensures that ASs can be emulated during the test execution. Therefore, the registered endpoints (e.g., URLs) need to be manipulated. During test execution, the TEE can then set the SUT in the emulation mode where all communications to AS are redirected to the TEE.

4.8 Summary and Discussion

In this chapter, the concepts of model-based testing applied to IoT-based services have been introduced and discussed. At first, a test-driven lifecycle concept has been introduced that facilitates early stage testing. It consists of the different phases of a service lifecycle. Different to other service lifecycle approaches it includes a clear separation between service deployment and service testing and can be utilised to create a defined information flow, which can be utilised during the service composition and deployment as well as the test derivation and test execution phase.

The overall test architecture has been presented in context of the IoT service creation and provisioning architecture of the IoT.est project. The integrated concept shows how available service knowledge and functionality (e.g., service deployment) can be utilised by the test derivation and execution process. Also, the concept of sandbox deployment of the SUT is built upon the architecture. The concepts for sandbox testing include abstract IoT-resource emulation for AS testing and AS emulation for CS testing.

Within the analysis of AS testing, changes in the AS itself have been identified to be required for ensuring a test friendly behaviour. The analysis of existing concepts shows, that current solutions tend to be too complex to be integrated into an automated test approach. As a consequence, the integration of a test controller into the AS is proposed to switch
between test mode and productive mode. With this extension it is possible to emulate external resources in a simplified and abstract way. Instead of encapsulating the AS from the outside, where different physical interfaces such as IEEE 802.15.4 are utilised to communicate with IoT-resources, the encapsulation within the AS ensures that complex communication interaction can be hidden from the test process.

For testing CSs, it is investigated that the control characteristics need to be taken into account specifically. Instead of a user interface that can be tested directly, the IoT-based CS rather focus on the control aspect, by interconnecting with ASs. Subsequently, the tests concentrate on validating the communication between the CS and the connected ASs. The analysis indicates that emulating all the connected AS makes it possible to test the CS at an early stage. Moreover, this approach ensures that the service can be tested completely (full coverage of service behaviour model) and at the same time can be significantly shortened due to the possibility to emulate those AS behaviour faster than real time.

As a consequence of the knowledge driven service creation approach of IoT.est, the main research objective covers already available service description documents and the question whether they can aid in deriving the service behaviour model (semi-) automatically. Within this chapter, opportunities based on WADL descriptions of AS and BPMN descriptions of the CS have been revised. A possible extension with rules indicates that missing information, such as the precondition and effects, can be integrated into the existing AS description documents. Based on an example, it has been shown that RIF-PRD can be utilised to automatically derive the IoT service behaviour model.

The discussed test approach results in a defined test process that addresses, amongst others, the discussed concepts. As the starting point, a novel service behaviour model, called IoT-behaviour model, is designed. It facilitates the description of the IoT-based service behaviour in a way that it can be tested at an early stage. Therefore, elements for separating the SUT from the connected resources (e.g., ASs) are an integral part. Furthermore, model elements are attached to allow for deriving standardised and executable test cases. The AuST testing tool is implemented based on the discussed concept and will be presented in the following chapter.
Chapter 5

Test and Emulation Automation of AuST

This chapter highlights the realisation of the developed testing tool called AuST. As a proof of concept, the realisation of the test framework described in Chapter 4 is shown. The designed and developed testing tool is utilised to facilitate research experiments for IoT-based service testing and test case reduction analysis and therefore enables the validation of proposed concepts. The chapter covers the implementation efforts including novel concepts for test data generation and emulation integration.

5.1 Targets and Methodologies

The key objective of this chapter is to document how the implementation of the AuST tool is realised in an efficient and adoptable way in order to demonstrate the applicability of MBT to the domain of IoT-based services with secure testing in a sandbox environment.

The resulting implementation allows testing the functional behaviour of atomic and composite services without the need for manual intervention. It includes the realisation of the automated emulation concept and substantiates the abstract concepts to executable code, targeting efficiency and adaptability. Moreover, it also enables experiments with the proposed test case diversity approach and can ensure that the performance (computation time, similarity score minimisation, failure detection rate) of different aspects of the algorithm can be measured and evaluated. A detailed description of the test case reduction algorithms are discussed within the next chapter.

The realisation of an automated test process is a challenging task due to a variety of different aspects that need to be taken into account. The steps needed to transform a behaviour model into executable tasks rely on several aspects, addressing logical and syntactical needs. The only possibility to realise such a testing tool within the duration of a PhD study is to take advantage of already available efforts. As a basis, an internal testing tool from the
University of Applied Sciences is selected. The utilised tool aims at testing communication protocols such as SIP based on finite state machines. While this tool is sufficient to test communication protocols it does not reflect the needs for IoT-based service testing. Moreover, the missing documentation and prototypical characteristics resulted in significant adoption and re-development. In the following subsections it is not the goal to discuss all achieved changes but try to highlight what makes it special for the proposed IoT-based test concept.

The IoT specific test approach is realised based multi-modal model transformation approach where different technologies ensure an easy adaptation of the overall process. At first the IoT-behaviour model is realised with Eclipse Modeling Framework (EMF). It ensures that the model has explicitly described and can be easily extended and integrated into the process. It also enables simple integration into the GUI and this makes it easy to validate and adopt the created model instance if desired.

The IoT behaviour model is analysed with the W-method and individual paths are identified. This way, it can be assured that transition and state coverage can be guaranteed. The emulation of ASs for CS testing is taken into account during the data generation phase. According to dedicated paths the data value restrictions are identified with a backward path analysis.

Transformation of the test path to executable test cases is based on a two step approach. The first step analysis the test paths subsequently and structures the resulting test case from a logical point of view. It identifies required variables, functions, control flow, etc and attaches them to a Apache velocity model [11]. Velocity is a Java-based template engine that is originally designed to generate web pages or Structured Query Language (SQL) [11]. Syntactical details are considered in the second step where velocity objects are transformed into TTCN-3 code files. This way, it is possible to separate syntactical detail from test logic and therefore decrease its complexity significantly.

The test execution, including the automated deployment concludes the automated steps and is utilised as proof of concept.

5.2 Test Design Engine

IoT Behaviour Model To enable the targets of an automated test derivation and execution process the IoT Behaviour Model (IoTBM) contains elements to describe the service logic, used protocols and interfaces and describes the interactions with connected sensors and actuators via ASs. Aspects such as incoming/outgoing messages, event behaviour based on timer, data storage and dependencies are modelled. The network dependent features such as QoS have not been modelled. Due to the chosen explicit model descriptions, based on
EMF, future extensions can be realised with automated code generation techniques for EMF models. This eases an extension towards further testing targets such as QoS.

Figure 5.1 shows the main elements of the IoT behaviour model from the service logic perspective.

![Diagram of IoT behaviour model](image)

**Figure 5.1: Main Model Objects for Logical Behaviour Description.**

The **StateMachine** consists of **StateMachineObjects** which are either **States** or **Transitions**. While there can only be one **InitialState** defining the start-point of the SUT after deployment, more than one **EndState** can be modelled. Each **State** can have zero or more numbers of **Transitions**. Each Transition specifies a target State. One **State** can be connected to other **States** via **Transitions**. Each **Transition** can contain one **Event** (Timer or expected message expressed as **MessageModelTemplate**) and multiple reactions described with **Actions**. For automated creation of messages the **MessageModelTemplate** is closer to the actual realisation with TTCN-3 than abstract behaviour models. It already includes the structure how the message should look alike in TTCN-3 and this ensures an easy model transformation later on. As provided by the TTCN-3 language, pattern and wildcards [12] are allowed for messages, that should be received by TEE. Each **MessageModelTemplate** can contain multiple **Parameters**. A **Parameter** describes the connected data type and its restrictions. For example, a message sending a room temperature value could be expressed with a parameter indicating, that the value can only be between \(-10^\circ C\) and \(60^\circ C\) (e.g., due to sensor limitations). **Parameter** description allows that different data value realisations can be generated and inserted into the defined messages before TC creation. Another object that can be contained in a **Transition** is a **Guard**. It can be used to define specific conditions to distinguish different **Transitions** e.g., based on previous received data values. A **Guard**
consists of at least one ActionObject and a condition, which enables operations such as $>,<,\leq,\geq$. Logical OR and AND operators can connect different conditions of one or more ActionObjects. During execution the TEE 'follows' the transition where all guard conditions are evaluated as true.

**Example 5.1:** A service that turns off a heater in a room as soon as a window in the room is opened can be modelled with a Guard. An AS is used to encapsulate a sensor that can detect whether a window is open or not. The response is stored in a Variable (with an Action) and this Variable is included into a Guard. The condition operation is set to *equal* and the condition to ‘Open’.

To enable the targets of an automated test derivation and execution process the IoTBM contains elements to describe the service logic, used protocols and interfaces and describes the interactions with connected sensors and actuators via ASs. Different to other already established models it explicitly describes the connection between a CSs and connected ASs by separating the service logic from connected ASs. This separation and description ensures that the ASs can be automatically emulated and integrated into the test process. The provided model is also not only an abstract model but is designed to be transferable to standardised and executable test cases. Figure 5.2 shows the main elements required for the model transformation and utilisation of the model for automated emulation of AS behaviours.

**Figure 5.2:** Main Model Objects for Test Execution.

*Components* are defined based on their connections to *Ports*, *Variables* and *Timers* (not shown within the Figure). This connection model enables separate modelling of parts of the SUT as *Components* and ensures the utilisation of those elements in the correct context.
during test execution. The model is thereby driven by the goal to create executable TCs in TTCN-3 format. The logical separation of multiple components allows reusing already modelled components: for example, those that are used by other previously tested services.

A Port object describes the communication interface of the SUT from the service perspective. Role objects describe, if the interface is acting as a Client or as a Server. The interfaces for RESTful communication are defined using the MessageType objects. Different MessageTypes are predefined in TTCN-3 and this allows to directly link the abstract MessageType to an actual incoming or outgoing message matching detection and verification during test execution. MessageType defines the direction of communications. For reactive system design in, out and inout directions are provided. In order to support the encapsulation of the SUT from its connected ASs, the author defined additional direction types inFromAs (receiving message from AS) and outToAs (sending messages to AS) to identify that this MessageType defines a communication between the SUT and the connected ASs. During test case creation these defined MessageTypes are utilised to create the emulation of the AS.

Example 5.2: The heating service in example 5.1 needs to communicate with a window service to determine whether the windows is open or not. The MessageType is defined for the message. It should be sent to the window service with MessageType outToAs. The response of the window service is modelled with the MessageType set to inFromAs. The MessageModelTemplate that is sent to the window service is placed within an Action and the expected response MessageModelTemplate is located in an Event object.

By providing ActionObjects, the service action element can i) store or compute operations based on Variables, ii) send a message via defined Port, or iii) initiate a new Timer to provide recurring events. While the Variable object contains all information to describe the object as a data type, the linked Parameter can specify data type restrictions (e.g., based on the linked physical entity types such as temperature). It is used for data generation, which takes into account the restrictions based on the Parameter and Guards.

AuST Editor Views The defined model can be generated and changed within an editor. Three different types of editors have been developed:

- Graphical editor of the high level service logic (e.g., connecting different States with different Transitions)
- Graphical editor of the binding of Variables, Timers, and Port to components
Textual mode specify the details for ActionObjects and MessageModelTemplates and link them with the required StateMachineObjects.

Figure 5.3 shows the high level service logic within a graphical editor. In the left-hand side of Figure 5.3 shows a hierarchical structure of the model. It is structured with the StateMachineObjects and the ConnectivityView elements. The details are provided at the bottom of the figure. The right-hand side shows the graphical representation of the state machine. InitialState is represented with black circles and the NormalState is shown with boxes. Transitions are shown with arrows connecting those states. For compatibility reason, the EndState is not yet present in the model and will be automatically attached by the Path Finder.

Path Finder explores the IoTBM by starting with the InitialState object. Each Transition object element consist of a target state, that connects the different states of the IoTBM.
Path Finder can be configured to achieve transition coverage with a breadth-first method or transition and state-coverage based on the $W$ method [93].

The transition coverage is computed by identifying the $InitState$ and building a test tree based on a breadth-first visit of all transitions. All transitions in each state are inspected and if the transitions lead to an unvisited state a new branch path is created. Afterwards, the new branch end states are visited and their transitions are inspected recursive, until an $EndState$ is reached. Each new inspected transition results in a new test path. The number of found paths equals the number of transitions if transition coverage is selected. For transition and state coverage it is further needed to identify a characterisation set (also called $W$ set), which is a set of input sequences that can be utilised to distinguish every pair of existing states in the model. The resulting paths are created by concatenating every sequence from the transition coverage set with every sequence in the characterisation set and apply them after the SUT is initiated (cf. [93] for a detailed description).

**Example 5.3:** if an IoTBM consists of two actions, two events and five states it results in 120 paths (state and transition coverage), if it consists of four events, four actions, and ten states the number of paths is 3200 (as discussed in Section 4.7 and Equation 4.1). In contrast, the breadth-first algorithm creates $N_S \cdot N_E$ paths where $N_S =$ number of states and $N_E =$ number events but provides only transition coverage. The concluding number of paths are 10 for the first example and 40 for the second example.

**Test Data Generator**  
As introduced in Subsection 5.2, the model contains objects to describe $Variables$. Those $Variables$ are utilised to specify allowed data types in messages, express the memory of the SUT, and connect them with guard conditions. The $Parameter$ description enables specification of those variables to enable a data generation that takes the data type and its restrictions into account. Different objects are defined to describe the supported data types. The supported data types are classified as $BaseDataTypes$ (integer, double, string, boolean) and $ComplexDataTypes$ which can contain one or more base data type or complex data type (recursively to support complex types such as an XML structure). Each $BaseDataType$ can have a $BaseDataTypeRestriction$ that can describe restrictions based on $max value$, $min value$, $enumeration$, $length$ and $pattern$ that can be (partly) applied to the different $BaseDataTypes$ (only where applicable: e.g., a pattern cannot be applied to boolean). As an input, the Test Data Generator receives all paths generated by the Path Finder. Multiple copies of the paths are created dependent on the number of target data values realisations for the connected $Parameters$ per paths (e.g., ten data value realisations for each path).
For each path copy all Transitions are scanned (backwards, starting with the last Transition of the path) for Guards, that involve Variables that have been manipulated during previous submissions. The goal is to create test values, that are valid to pass the guard condition. It is therefore required to combine the restrictions of the Variable and the guard condition. This ensures that the guard condition is passed during test execution and ensures that every path of the model can be tested completely (as discussed in Section 4.7 and Example 5.4).

**Example 5.4:** A SUT that reacts differently to two different temperature value ranges where both temperature values are detected by the same sensor device. In both cases, the same message (in terms of structure) is received but with different data values. Since the SUT has two different reactions, depending on the data value, both cases need to be created with different data value ranges and tested individually to ensure that both cases have been verified.

If a Guard is detected, the variables connected to this Guard are stored to compare them with previous received MessageModelTemplates (from the perspective of the SUT). In case a MessageModelTemplate with the same Variable is detected, the data type restrictions are adjusted to the guard constraints. The previous restrictions defined for the Variable (in case there are any) are combined with the restrictions that are based on the current Guard and applied to the current path only. The combination is possible for max and min value and enumeration. Note, the author assumes, that the data value restrictions, stored in the model, are not contradicting the guard restriction (the possible data values to pass the guard are a subset of all allowed data values of the variable). For each variable the connected Parameter is then transferred to the data generation, here realised with a Drools engine [165]. Drools is an inference based rule engine (production rules) [166]. The Drools engine produces individual realisations of those data types and stores them in a TestValue object. These objects are then utilised to replace placeholders within the MessageModelTemplates with valid test data.

In each path it is checked if there are MessageModelTemplates where data values have not been generated based on the guards restriction procedure outlined above. For these MessageModelTemplates and their connected Variables and Parameters the data generation is performed only on the previous defined data types and restrictions. With the completion of this step, each path contains the individual realisation of the data values. All information to execute those paths are present. The output of this component will be called TCs since it satisfies all requirements for a TC.
5.2 Test Design Engine

**Diversity Analyser** follows a two step approach. At first, a pairwise similarity score between all TCs is computed. The similarity computation can be either performed based on Hamming distance or Levenshtein distance algorithm. The outcome is stored as a similarity score (one minus distance) in a two dimensional matrix, which comprises all pairwise comparisons. The second step is to use different optimisation algorithms such as Greedy, Genetic Search and Group Hill Climbing Search and as a baseline random-search to optimise the TCs group. The goal is to find a group of TCs, which have a low average similarity score between the selected TCs. Therefore, the different optimisation algorithms are applied individually to evaluate their performance. As a result, the Diversity Analyser output reduces the TCs to the identified group of TCs. The algorithms and the target number of TCs can be defined with the AuST tool. Chapter 6 explains the approach and novelties of different proposed algorithms in detail.

**Test Case Creator** is responsible to convert the TCs, received from the Diversity Analyser, stored in the IoTBM EMF format, to executable TCs in TTCN-3. TTCN-3 is a test specification language, that supports black-box testing of distributed systems and it is standardised at ETSI [13]. Due to their provided interfaces and their adaptability, the language can be applied to different types of protocols and systems. A two step approach is conducted where at first the IoTBM model is transformed to objects. The objects can be directly used by the template engine Velocity [11]. The actual generation of the TTCN-3 code is realised with the Velocity template engine. This enables the separation of syntactical details of the TTCN-3 language from the logic analysis and enhances the manageability of the test case creation process.

In the first phase of the transformation process, the test system is defined by creating a TTCN-3 system module, which includes all used Ports and their allowed MessageTypes. A configuration file link ports to TTCN-3 codecs and documents the location of external function plugins for test execution visualisation and automated deployment. For each Component, a TTCN-3 module is created, which includes the mapping of the Ports to the actual test system. Variables and Timers are defined for each Component.

Different test cases received from the Test Data Generator can be analysed individually. For each TCs received from the Test Data Generator a corresponding executable test case is created. During this process, different Velocity objects are created and joined together. To ensure a fundamental understanding of the key aspects, the process is discussed in a simplified manner. Figure 5.4 shows a high level view of the process for an individual test case as an UML 2.0 activity diagram, where the Pin element is used to express the output elements of the activities and the activity nodes. The process is based on the model objects.
defined in 5.2. Figure 5.4 shows these objects as input elements and guard conditions. The output elements are Velocity objects (marked with a capital ‘V’) which are used to generate the TTCN-3 code. Note, that these Velocity objects are rather an output of the activities then inputs for the next activity which means that the input elements of the actions are always the objects of the IoTBM. For a higher readability, the last action elements are not connected with the output of the action nodes.

Figure 5.4: Simplified Process Visualisation of the Transformation from IoTBM Paths to Velocity Objects.

The process, shown in Figure 5.4, starts by analysing each element of the current TCs. A TC is a sequence of StateMachineObjects and these objects can either be an instance of State or Transition. The first element of the test case is an InitialState and at first, a VTestCaseFunction is created. It represents a TTCN-3 function that is used to group different steps of the test case. Whenever a Transition execution is finished, the TTCN-3 function changes.

In case the StateMachineObject is an instance of Transition, the possible Actions objects and the Event object are handled.

At first, each Action object is investigated individually. Actions of type Port or Timer will cause the creation of a TTCN-3 Alt element, which can be expressed to allow different alternative reactions of the SUT. The Alt element is used to model the expected behaviour and default elements are used to stop the test execution if something unexpected happens. This means that the TC execution is stopped either if an unexpected message has been received by the TEE or if it takes too much time to receive a message (timeout). The TC verdict is set to ‘fail’ to indicate that an unexpected behaviour occurred. A verdict is returned by the test case after execution. In case that the Action object is a Timer, a VCommand object is used to start
5.2 Test Design Engine

or stop a Timer. This VCommand object is then added to the VAlt (either already existing or newly created). The Variable object is translated to a VAssign object, which can be utilised to make variable realisations in TTCN-3. For an Action object of type Port a TTCN-3 template is defined by creating a VDefineTemplate out of the MessageModelTemplate connected with the current Transition. Note that the IoTBM is created from the perspective of the SUT. Therefore, every MessageModelTemplate connected to an Action is expressed as an outgoing message. This means that for testing this message needs to be detected by the TEE. The MessageModelTemplate is used to compare the received message with the defined one. In order to express the behaviour of an IoT-based service, two different types of MessageDirection are distinguished within this process step. If the MessageDirection is ‘out’, a web service behaviour is modelled where the SUT can answer an incoming message Event (e.g., CoAP/HTTP GET/GETresponse) and therefore, only a receive message needs to be modelled. In case that the MessageDirection is ‘OutToAS’, this action indicates a request message to an AS. Since the test approach separates the CS from the connected AS this message needs to be redirected during test execution to the TEE. In addition, the TDE should make sure that the TEE can receive the message and can send an emulated response. For receiving a message of the MessageType ‘outToAS’ a VAltElement is created, which can be translated to an TTCN-3 ‘call’ function, that can be applied to a TTCN-3 port. Otherwise the TTCN-3 ‘getreply’ function element is created. After finishing all Action objects, the expected reaction of the SUT is modelled (e.g., state change) and a VTestCaseFunction defines which TTCN-3 function is called in case of an successful behaviour based on the detected Actions.

Note, in case that the Actions do not contain a Port object, the reaction cannot be detected directly. Therefore, the state might not be distinguishable from all other states and the W-method cannot be applied and only transition coverage can be applied.

After the inspection of all Action objects, the Event object is investigated. In case that the Event is a Timer, an VSimpleAltEvent element is created. This results in a waiting behaviour during test execution. After the Timer has timed out, the defined Actions are verified. If the Event is realised with a Port (Event is a message) it first needs to be distinguished if this Transition contains a Guard. A Guard results in the creation of an VIf object, that validates if the Guard condition(s) is(are) met after the Event has been occurred. If the Guard condition is not met, the test case execution stops. Similar to the Port handling of the Action object, a VDefineTemplate is created with the MessageModelTemplate and a template variable is assigned. A high level view of the performed steps to transform the MessageModelTemplate to Velocity objects is shown in Figure 5.5. In case, that an XML data structure is used for the message, inner templates are recursively created and structured as a VXmlSend. In
order to influence the inner elements of the XML structure variables are generated for each inner template element and assigned to a Variable with a \texttt{VAssign} object. Depending on the \texttt{MessageDirection} a request or reply operation is written for the TTCN-3 test case via an \texttt{VInvoke} or \texttt{VInvokeOnObject} object.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig5.5.png}
\caption{Simplified Process Visualisation of the Transformation from a MessageModelTemplate to Velocity Objects.}
\end{figure}

The creation of a TC is completed with the last \texttt{StateMachineObject (EndState)} by setting the test case verdict to 'pass'. The described procedures are repeated for every TC. A so called control sequence ensures, that all created TCs will be executed subsequently. Created TCs of two example services can be found at http://kat.ee.surrey.ac.uk/AuST/TestCases.html.

\textbf{Test Case Compiler} \hspace{1em} After creation of all TTCN-3 TCs, static message descriptions (such as HTTP-GET, etc.) in conjunction with external functions to control visualisation and automated deployment are copied to the test project. The \texttt{TThre} test compiler developed by Testing Technologies (TT) [14] is utilised to make these standard conform TCs executable. Note that this is just one possible realisation of the compilation process and it was not part of the research to optimise this step, instead it was used to demonstrate the automation of the whole process from a IoTBM to test execution.
5.3 Test Execution Engine

TEE executes the TCs and controls the run-time environment. The main components of the TEE are shown in Figure 5.6. A Test Execution Control Interface based on RESTful interface can be used to start the test execution for service defined by a service identifier. The Test Case Execution Engine is realised by using the TTworkbench developed by TT [14]. The TCs can be either executed in an automated way via script execution or by utilising the GUI of the TTworkbench. For visualisation of the automated execution an additional view based on node.js [167] is provided that shows the initial IoTBM and which transitions are currently tested. Figure 5.7 shows a Screen-shot of the test execution visualisation of the Simple Composite Service introduced in Section 2.1.3. Note that the EndState is not connected to the rest of the IoTBM since the service has no end and the EndState is only included for compatibility reasons. In case of a failure occurrence, the link colour is changed to red.

Figure 5.6: Test Execution Engine with External Interfaces and Main Components.

Figure 5.7: Test Execution Visualisation.
Atomic Service Testing  In difference to CS testing, AS testing is based on internal adaptations of the AS to ensure a test-friendly behaviour. As discussed in Section 4.5.1 the service needs to realise a test controller and a resource emulation interface. For this purpose, the interface is defined in a generic way that ensures an easy emulation based on the transferred message is possible. The defined structure of the HTTP request includes resources to distinguish different services, its features and an IoT-resources identifier to identify the IoT-resource that should be emulated. Within the body of the message a XML structure is utilised to define the structure of the expected answer. This means that individual parameter names and their data type (e.g., double, int, etc) are specified. The emulation component, within the TEE then answers the request with the same structure and includes individual data values for each parameter. Please note that the aim of the defined interface is to verify the overall concept. One of the limitations of the defined interfaces is that messages can only define required values based on data types, additional effort is required to include also value restriction based on physical types (e.g., meaningful temperature value ranges). Within this proof of concept the required limitations are realised within the emulation component. By adding information to dedicated services the values are further restricted. For the example atomic service in Section 2.1.3, this means that the *pan*, *tilt*, and *zoom* are limited to the specification of the camera.

Composite Service Testing  For composite service testing, the Run-Time Control Plugin connects the run-time environment with the test execution. It interconnects each TC execution start with the capability to reset the SUT to a defined start point. Based on TTCN-3 external function elements, an HTTP interface is used for communication between the run-time environment and the TEE. Due to the integration into the TTCN-3 test cases, this functionality is directly included into the test case execution process in an automated way.

5.4 Run-Time Sandbox Environment

The Run-Time Sandbox Environment is shown in Figure 5.8. It is realised based on a JBoss Switchyard application server [168]. Switchyard is capable to execute services based on BPMN. The run-time control interface is realised with HTTP. It enables that the SUT can be stopped and (re-) deployed by the *Execution Run-Time Control Engine*. This ensures that the SUT is in a defined state during test execution.

The separation between the SUT and the connected actor and sensor devices is realised by manipulating the endpoints of the connected AS. Each URL is changed to a dedicated TEE URL by the *Test Controller*. This ensures that the interface emulation of the ASs can
be performed by the TEE during test execution. The emulation itself is realised within the TTCN-3 test case as described in Section 5.2.

5.5 Summary and Discussion

This chapter covers the realisation aspects of the AuST testing tool. The realised functionality to automatically integrate AS emulation into the test case creation process enables the testing of CS at an early stage of the lifecycle. Different to related work approaches, the followed emulation concept is realised in a sandbox with capabilities to encapsulate and emulate IoT-resources and IoT-based ASs. As basis for the test derivation and execution process a IoT-behaviour model is realised, that can ensure: i) a clear separation between the SUT and its connected resources; ii) reflect the control characteristics of CS; and iii) can be automatically transformed into standard conform TTCN-3 test cases.

The applied multi-modal model transformation ensures a strict separation between logical and syntactical details. In this way the process is implemented in an adoptable and efficient way whereby utilisation of EMF ensures that also user interactions (e.g., model extension or verifications) are easily integrated. IoT related aspects for early stage testing are realised with individual data generation and automated emulation integration. Different to existing testing tools the implementation significantly simplifies (in terms of required work by test developer) the process of IoT service testing. The test automation is also extended to the test execution, whereby an existing run-time environment is extended to the needs of sandbox testing. The achieved work is based on an communication protocol testing tool developed at the University of Applied Sciences Osnabrück and the code is significantly changed to express the needs of the described framework and its concepts. The new development includes:
• Model extensions to represent the communication between the SUT and external resources to enable emulation concepts. A novel IoT-behaviour model is introduced.

• Within the new IoT-behaviour model, the finite state machine is extended to enable data types and data type restrictions based on guards and physical value restrictions. It includes a novel approach for data generation based on individual TCs and ensures the integration of data values in the test case diversity approach.

• Integration of the test case diversity analysis to secure a scalable test process.

• The path discovery is changed to guarantee state and transition coverage to ensure a known test coverage.

• The process to create executable test cases is rewritten to enable resource emulation, automated (re-) deployment and the utilisation an HTTP protocol adapter from Testing Technologies [150].

• Creation of plug-ins for automated sandbox deployment of SUT and test execution visualisation.

Among the aim to demonstrate the functionality of the proposed test framework, the implementation is also the basis to conduct the test case diversity analysis which is discussed in the next chapter. Evaluation results for the overall test automation and scalability through test case reduction are both performed with the AuST tool and are shown in Chapter 7.
Chapter 6

Scalability through Test Case Diversity Optimisation of AuST

This chapter discusses the test case reduction concept of the AuST testing tool. In particular, algorithms for test case similarity investigations and test case selections are proposed and analysed in terms of their functionality as well as their computation time complexity. Analytical investigations guide several extensions and novel designs of algorithms that aim at selecting TCs which have a low similarity.

6.1 Objectives and Methodologies

The overall purpose of the outlined algorithms are to find a set of test cases, that is selected from all available TCs, and have a higher diversity than known approaches. The general assumption is that more diverse test cases have a higher probability to detect failures [96]. The diversity assessment is based on a pair-wise similarity investigation between all available test cases.

At first, the distance algorithms Hamming and Levenshtein are adjusted to the problem of test case selection. Hemming similarity is extended with two novel approaches to include data values into the similarity computation. To overcome current limitations of Hemmatis’ et al. [96] Levenshtein interpretation, a new algorithm which is more close to the original meaning of Levenshtein is proposed.

Different to already known algorithms the proposed TC selection algorithms follows a group oriented optimisation approach. The novel algorithm Group Greedy, Group Hill Climbing and Diversity-based Steady State are therefore extending the state of the art. To ensure that the proposed algorithms can be compared to baseline approaches the Greedy
Test Case Diversity Optimisation algorithm proposed by Hemmati et al. [96] and a random search algorithm are also described within this chapter. The algorithms are outlined within the subsequent sections, including pseudo code and computation time complexity analysis with $O$ notation. In that way, the proposed solution can be validated and compared to other and already known approaches. The chapter discusses the algorithms and makes them reproducible by the research community. Afterwards, the computation time complexity of the different algorithms is graphically shown and compared.

6.2 Similarity Score Computation

The initial step for the TC diversity approach is to identify the similarity between TCs. At first, it is required to define which information is included in this similarity score computation. Results documented in [96] indicate that the best performance can be achieved by including all information present in the test model. To measure the distance between two objects in a multidimensional feature space, set-based similarity scoring mechanism can be used. In our case, a TC is not only a group of objects (e.g., states, transitions, input, and output), it is also a sequence, which could have a different order of these objects compared to other TCs. Sequence-based distance algorithm can therefore improve the test case similarity score computation. The result of the similarity score computation is a similarity score matrix, which contains the pairwise similarity score between all TCs. In the last step, the target number of TCs is selected based on the similarity score matrix. The goal is to find the group of TCs, which have the lowest average similarity score between the TCs of the selected group. Note that this is a NP-hard problem [138] and the optimum can therefore not be found in polynomial time. The maximum number of combination possibilities is reached with a group size of $k = n/2$, where $n$ is the total number of TCs and $k$ is the target group size. For example, with $n = 132$ and $k = 66$ the number of possible combinations is $\binom{n}{k} \approx 4 \cdot 10^{38}$.

6.2.1 Hamming Similarity Scoring

As a baseline, a Hamming-based algorithm is implemented according to the work discussed in [96]. The Hamming distance is an edit-based distance algorithm, it defines the minimum required operations to transform one string to another with editing operations (delete, insert, and substitute) for strings with the same length [169]. The general idea is to define a set of allowed symbols (comparable to an alphabet) and identify how many symbols are present. By sharing more symbols the TCs are more similar, which can be expressed as a high similarity, represented by the reciprocal of the applied Hamming distance.
6.2 Similarity Score Computation

Input: allowedSymbols

1  foreach testCase do
2     foreach allowedSymbol do
3         if testCase contains allowedSymbol then
4             occurrenceBit = 1
5         else
6             occurrenceBit = 0
7         end
8     Add occurrenceBit to bitOcurrenceStream
9     end
10  end
11  foreach bitOcurrenceStream do
12      simStream = pairwise XOR of bitOcurrenceStream
13      similarity = simStream / length of bitOcurrenceStream
14  end

Figure 6.1: Hamming Similarity Scoring Algorithm.

The pseudo code in Figure 6.1 shows the basic steps for this approach. In all discussed similarity scoring mechanisms, the algorithm starts with the identification of the allowed symbols. Depending on the encoding this could include input, output symbols, states, guards, etc. For simplicity of the presentation in this thesis, all experiments are conducted with allowed symbols for input, output and states. Guards have not been taken into account. If the algorithm extension uses guards, it would not change the algorithm but would result in a larger symbol space which causes in lower similarities between all TCs. For each TC, the occurrence of the individual allowed symbols is identified. Each occurrence is documented with an occurrenceBit = 1 and 0 if it is not present. This occurrenceBit is added to a bitOccurrenceStream. For each TC the bitOccurrenceStream is pair- and bitwise XOR compared to each bitOccurrenceStream from all other TCs. The resulting XOR distance is then used to count the bits, that are “1” and divide them by the length of bitOccurrenceStream. The result is stored as the pairwise similarity score. The Hamming Distance algorithm is limited to strings with the same length. Since TCs do not necessarily have the same length, the algorithm has been altered as proposed in [112]. The modified Hamming similarity scoring algorithm can handle strings with different length but it is set-based and does not take the occurrence order of the symbols into account.

The time complexity of the Hamming similarity scoring can be expressed as shown in Equation 6.1,

\[
\frac{n \cdot m + n \cdot \left( \frac{m \cdot (n - 1)}{2} + 1 \right)}{1 - 10} \in \mathcal{O}(n^2m) \tag{6.1}
\]
where $n$ is the number of TCs and $m$ is the number of allowed symbols. The numbers below the brackets show the associated line number of the pseudo code shown in Figure 6.1. In Figure 6.1, the lines 11-14 compare for each TC each allowed Symbol occurrence with all other TCs. Since the resulting similarity score between two TCs do not depend on the order of comparison, the computation can be stopped after $(n - 1)/2$ cycles, where $n$ is the number of TCs (mirror symmetry).

### 6.2.2 Hamming Similarity Scoring with Data Value integration

Within the last section, the Hamming similarity scoring computation is applied to an abstract test model. In other words, the individual data values have not been taken into account. An abstract test model, within this context, is therefore a test model which describes the behaviour of the SUT with defined ranges of data values. For each message, data types are defined, which specify expected data value restrictions.

From the application perspective, the restrictions can take place based on the application logic (e.g., cooling a room for a high temperature and heating it for a low temperature). Different realisations of those defined data values and their restrictions based on the current test case can be created during test derivation. These different realisations of abstract TCs are not distinguishable with the proposed Hamming similarity scoring computation. While data value generation can be already optimised (e.g., based on data space coverage) during data value generation for each data type occurrence individually, it is not possible to optimise them as a group together with other attributes of the test model. The optimised individual data values will no longer be optimal if a certain group of TCs has been selected for execution. Therefore, the integration of data values into the similarity investigation enables a better selection of the most divers TCs.

Figure 6.2 shows the pseudo code for data extension of the Hamming similarity scoring. In addition to the original detection of the Hamming similarity scoring, whether an allowed symbol is present or not, the Hamming data extension also includes the distance of one data value of one TC to each other generated data value of the same kind (same physical value type). To accomplish this, each data value of a test case is detected and compared to other data values of the same kind. To calculate the distance between two values, different algorithms can be utilised. As a baseline a linear distance computation has been applied.

Similarity investigations are based on a pairwise TC comparison and therefore, each distance value of each data value of one TC is subtracted (linear as a baseline) from the distance values of a second TC. After normalisation, a similarity between those two TCs for a specific data type can be expressed. Afterwards, the results are weighted to include the data similarity analysis into the overall Hamming similarity scoring. The weighting can be realised...
6.2 Similarity Score Computation

Input: Messages

1. simArray /* similarity of one data value to all others */

2. foreach TestCase do
3.     foreach Input and Output Message do
4.         foreach otherDataValue do
5.             simArray[TestCase][dataValue] = calculateSimilarity(DataValue, otherDataValue)
6.         end
7.     end
8. end

9. foreach TestCase do
10.     foreach dataValue do
11.         sumSim+= substr(simArray[OwnTestCase][DataValue], SimArray[OtherTestCase][DataValue])
12.     end
13. simMessage = sumSim / Number of Data Values
14. HammingValue[TestCase][Message] =
15.     w + (1 - w) · simMessage

Figure 6.2: Data Integration into Hamming Similarity Scoring Algorithm.

with two approaches: i) based on occurrence ratio of data types compared to all occurrence symbols and ii) changing the similarity score computed with the original Hamming algorithm by removing the symbol where the new algorithm can be applied (e.g., transitions with data types). The author calls the first approach Data-Enriched Hamming similarity scoring ($H_A$).

The second integration of data types is called Weighted Data-Enriched Hamming similarity scoring ($H_B$).

**Example 6.1:** The allowed symbols are $a, b, c, d$, where $c$ contains a data type and a data value respectively. Test case 1 (TC1) consists of the sequence $b$ and $c$ and test case 2 (TC2) is the sequence of the symbols $a$ and $c$. The initial Hamming similarity scoring is computed by creating a bitstring of the occurrences of the symbols where the initial Hamming similarity scoring (without data integration, that are named $H_O$) can be calculated as described in Equation 6.2, where '1' represents the presence of a symbol in the first two rows. For this example, the resulting $H_O$ is $\frac{1}{2}$. The influence of the two different data weighting approaches are demonstrated with an example where $H_D$ is set to 1.0 (complete similarity between all data value similarity scores).

Equation 6.3 shows the resulting combined Hamming similarity (here and after called $H_A$) for the first weighting approach. Where $\omega$ is the ratio of the number of data values divided by the number of allowed symbols. The results show, that $H_A$ can be higher than the original Hamming similarity score. The reason is, that the term $H_O$ already includes the symbol $c$ which is further analysed with the Hamming data algorithm. Therefore, the
first weighting algorithm can result in a higher similarity between TCs, that share the same symbols with data values. For the later TCs selection, this means, that it is more likely that these TCs will be removed from the set of selected TCs. In contrast, the second weighting approach guarantees, that the resulting similarity score is lower or equal than the original ($H_O$) one, by subtracting the symbols with data values from the comparison set. This changes the $H_O$ to $H_O' = \frac{1}{3}$. For the example, the resulting similarity score ($H_B$) is shown in Equation 6.4. The second approach to realise the weighting lowers the similarity between two TCs compared to the first weighting algorithm. Therefore, it is more likely that TCs, which share symbols with data values, will be selected for the test execution.

\[
abcd
\]

\[
\begin{align*}
TC_1 & \quad 0110 \\
TC_2 & \quad 1010 \\
XOR & \quad 0011 \\
\Rightarrow H_O & = \frac{1}{2} \\
H_A & = H_O \cdot (1 - \omega) + H_D \cdot \omega \\
H_A & = \frac{1}{2} \cdot \left(1 - \frac{1}{4}\right) + 1 \cdot \frac{1}{4} = \frac{5}{8} \\
H_B & = H_O' \cdot (1 - \omega) + H_D \cdot \omega \\
H_B & = \frac{1}{3} \cdot \left(1 - \frac{1}{4}\right) + 1 \cdot \frac{1}{4} = \frac{1}{2}
\end{align*}
\]  

Where $H_O$ is the Hamming similarity scoring, $H_A$ is the Data-Enriched Hamming similarity scoring, $H_B$ is the Weighted Data-enriched Hamming similarity scoring, $H_O'$ is the Hamming similarity scoring without symbols that include data values, and $H_D$ is the similarity scoring based on data value analysis.

Equation 6.5 shows the computation time complexity:

\[
C_{time} = n \cdot k \cdot (d - 1) + \underbrace{n \cdot k \cdot d \cdot (n - 1)}_{2-8} + \underbrace{\frac{k \cdot d}{2}}_{11} + \underbrace{\frac{2k}{9-10}}_{13} + \underbrace{\varepsilon \mathcal{O}(n^2kd)}_{13}
\]  

(6.5)
6.2 Similarity Score Computation

where \( n \) is the number of TCs, \( k \) is the number of messages which involves data value realisation, \( d \) is the number of data value realisations. The numbers below the brackets show the associated line number of the pseudo code shown in Figure 6.1.

### 6.2.3 Levenshtein

The realisation of the Levenshtein similarity computation is shown in Figure 6.3. Levenshtein

```plaintext
1  Input: SymbolsOfEachCase
2  foreach pair of testCases do
3      foreach symbolNum of FirstTestCase do
4          m[symbolNum, 0] = symbolNum
5      end
6      foreach symbolNum of SecondTestCase do
7          m[0, symbolNum] = symbolNum
8      end
9      foreach symbol of FirstTestCase do
10         foreach symbol of SecondTestCase do
11             if symbolFirst == symbolSecond then
12                 m[symbolFirst, symbolSec] = m[symbolFirst -1, symbolSec -1]
13             else
14                 m[symbolFirst, symbolSec] = Min (
15                     m[symbolFirst -1, symbolSec] + 1,
16                     m[symbolFirst, symbolSec -1] + 1,
17                     m[symbolFirst -1, symbolSec -1] + 1)
18             end
19         end
20  end
21  similarity = 1 - m[numOfSymbolsFirst, numOfSymbolsSec] /maxNumSymbols;
```

Figure 6.3: Levenshtein Similarity Scoring Based on Wagner-Fischer Algorithm [170].

is also an edit-based distance algorithm and is not limited to strings of the same length [118, 169]. Each edit operation (delete, etc.) results in an increasing distance between the compared strings. The Levenshtein algorithm is initiated by the identification of the symbols which are part of the current pair of TCs. A matrix is created where the first row and column contain an increasing sequence from one to the quantity of symbols with an increment of one. Based on the basic operations add, del, and modify the symbol sequence of the FirstTestCase is compared to the symbol sequence of the SecondTestCase. It is identified how many operation steps are required to alter the FirstTestCase and reach the second one. The algorithm is based on the Wagner-Fischer algorithm [118, 170] and the algorithm always prefers matches over insertions or deletions even if they provide a better score. The last computed matrix element contains the distance between the two TCs. For comparison this
distance is then converted to a similarity score and normalised to \( \text{maxNumSymbols} \). Different to the description of Hemmatie et al. \[96\] where only matches have been counted, this algorithm is able to identify the distance between two TCs as intended by the Levenshtein algorithm. Equation 6.6 shows the resulting computation time complexity:

\[
C_{\text{time}} = n \cdot \frac{m^2 \cdot (n-1)}{2} \epsilon \Theta(n^2m^2)
\]  

(6.6)

where \( n \) is the number of Test Cases and \( m \) number of allowed symbols.

### 6.3 Test Case Group Selection Optimisation

#### 6.3.1 Greedy

The Greedy algorithm is a heuristic problem solving approach where in each step an optimal solution is selected. It can be utilised to approximate a global optimum based on local optimums \[106, 112\]. For TCs reduction, the goal is to remove the worst TCs from all available TCs until the target amount of TCs is reached. As proposed by Hemmatie in \[96\], the comparison can be realised by applying a pairwise comparison between TCs. While the target number of TCs is not reached, the two most equal TCs are identified and the shorter TC is removed (assuming that longer TCs can find more failures). The resulting algorithm is shown in Figure 6.4.

Equation 6.7 shows the computation time complexity of the Greedy algorithm with \( \Theta \) Notation:

\[
C_{\text{time}} = \sum_{l=1}^{p} (n-l)
\]

\[
C_{\text{time}} = -\frac{p^2}{2} + pn - \frac{p}{2} \epsilon \Theta(n) \text{ for } n > p + 1
\]  

(6.7)

where \( n \) is the number of TCs and \( p \) is the target number of TCs. The while loop (line 3 in Figure 6.4) continues for \( p \) iterations in which the number of comparisons decreases in every iteration \((n-l)\). Therefore, the complexity can be expressed as being linear to the size of \( n \) with the \( \Theta \) notation.

A drawback of the Greedy algorithm is that the decision which TCs should be removed relies on a pairwise comparison of TCs \[112\]. This type of Greedy algorithm does not support the goal to optimise a group of TCs. To overcome this, a new Greedy algorithm called Group Greedy is proposed. Figure 6.5 shows the pseudo code of the Group Greedy algorithm. At first, the algorithm selects one TC as the first element of the target TC group.
6.3 Test Case Group Selection Optimisation

Input: simScs /* Similarity Scores */
Input: Test Cases (TCs)
Input: b /* Target number removed Test Cases */

numRemoved = 0
minScore = max allowed value
while numRemoved < b do
    foreach TC do
        tc1 = current TC
        foreach other TC do
            tc2 = other TC
            simScore = simScs[TC1][TC2]
            if minScore > simScore then
                minScore = simScore
                selectedTC = TC2
            end
        end
        removedTC = selTc(tc1, tc2, simScs)
        numRemoved++
    end
end

Figure 6.4: Greedy Similarity Scoring Based on the Proposal of Hemmatie et al. ([96]).

Input: simScs /* Similarity Scores */
Input: Test Cases (TCs)
Input: p /* Target number Test Cases */

selTCs /* list of selected test cases */
selTCs[1] = add first TC randomly
minScore = max allowed value
minSecScore = max allowed value
curScore = 0;
while selTcs.size < p do
    foreach unselected TC do
        foreach selected TC do
            curScore+= simScs[selTC][unSelTC]
        end
        if minScore > curScore then
            minScore = curScore
            tc1 = unSelTC
        else if minSecScore > curScore then
            minSecScore = curScore
            tc2 = unSelTC
        end
    end
    worseTC = selTc(tc1, tc2, simScs)
    selTCs = add better TC
    p++
end

Figure 6.5: Proposed Group Greedy Algorithm.
Then, for each unselected TC the total similarity score between the selected TCs and the unselected TCs is computed. The two unselected TCs with the lowest total similarity to the selected group are identified. In order to provide a similar flow to the first Greedy algorithm, the two TCs are then compared based on the length.

Equation 6.5 shows the computation time complexity:

\[
C_{\text{time}} = \sum_{l=3}^{p} (l-1) \cdot (n-l)
\]

\[
C_{\text{time}} = \frac{np^2}{2} - \frac{np}{2} - n - \frac{p^3}{3} + \frac{p}{3} + 2
\]

\[\in \mathcal{O}(n) \text{ for } n > p + 1\]

where \(n\) is the number of TCs and \(p\) is the target number of TCs. Due to the negative terms in the equation the computation time complexity mainly depends on \(n\).

### 6.3.2 Random

As the baseline, the random selection of a group of TCs has been implemented. As outlined in Figure 6.6, it starts with the input of the similarity score matrix, all TCs, the target number of TCs and the number of trials (\(\text{numTrials}\)). The output of the algorithm is the list of selected TCs for the execution. For each trial, the target number of TCs is selected out of all TCs. Afterwards, the summary similarity score (summary of all similarity scores between all selected TCs) is computed. For each iteration, the computed summary similarity is compared to the lowest previous summary similarity. After the defined number of trials, the group of TCs with the lowest summary similarity is selected for the test execution. This approach finds the best group of TCs, if the number of trials is infinitely. It is questioned, if other search algorithms can outperform this simple random methodology by identifying a better group of TCs within the same amount of resources (e.g., computation time).

Equation 6.9 shows the computation time complexity:

\[
C_{\text{time}} = m \cdot \left[ \frac{p + p \cdot \frac{p-1}{2}}{3} \right] \in \mathcal{O}(mp^2)
\]

with \(m\) is the number of repetitions, and \(p\) is the target number of selected TCs. The numbers under the braces indicates the line numbers in the Figure 6.6.
6.3 Test Case Group Selection Optimisation

6.3.3 Hill Climbing

Hill Climbing algorithm is based on the identification of local optima. Instead of searching for the best possible solution out of all groups of TCs, it can help to find good but not necessarily the best group of TCs. Hill Climbing searches for local optima by looking at neighbour elements. A neighbour is an element, which is structural close. From a start point, the algorithm tries to find a better solution by exchanging one element with a neighbour. The algorithm stops if no better neighbour is found (local optimum reached). Hill Climbing algorithm differs how the starting point is chosen, how neighbours are defined and how many elements can be reached with one move operation (numbers of direct neighbours) [96].

A new proposed Group Hill Climbing algorithm is shown in Figure 6.7. The first step is equal to the Random Search algorithm with one trial. Then, for each TC within this group it is investigated if a neighbour TC exists, which lowers the summary similarity between all TCs of the selected group ($\text{sumSim} < \text{lastSumSimilarity}$). The order of the created TCs thereby defines which TCs are neighbours. Therefore, the methodology how the TCs are created is expected to have influences on the performance of the identification of neighbours. To reduce the computation effort only the variable part of the summary similarity is computed. The variable part of this summary is the pairwise similarity score between the selected group of TCs (without the current TC) to the neighbours of the current TCs ($\text{rowSim}$ in Equation 6.7). The neighbour search stops when a neighbour with worse characteristics than previous
one is found. Therefore, the computation effort is not deterministic and depends on the initial set of TCs and the characteristics of the neighbours (e.g., if the next optimum is nearby).

**Input:** Similarity Scores

**Input:** Test Cases (TCs)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p /* Target number Test Cases */</td>
</tr>
<tr>
<td>2</td>
<td>selTCs /* list of selected test cases */</td>
</tr>
<tr>
<td>3</td>
<td>selTCs = randomly select ( p ) test cases out of all</td>
</tr>
</tbody>
</table>

3. **foreach** Selected TC do

   4. rowSim /* Current Row Similarity */
   5. rowSim = sum similarity of selTCs *
   6. tmpSelTCs /* temp. list of sel. TCs */
   7. tmpSelTCs = selTCs - currentTestCase *
   8. rTC /* Right TC of Current TC */
   9. rTC = currentTestCaseNumber + 1 *
   10. lTC /* Left TC of Current TC */
   11. lTC = currentTestCaseNumber - 1 *
   12. rRowSim /* Row Similarity with rTC */
   13. rRowSim = sum similarity of rTC to tmpSelTCs *
   14. lRowSim /* Row Similarity with lTC */
   15. lRowSim = sum similarity of lTC to tmpSelTCs *

16. **if** rowSim > rRowSim < leftRowSimilarity then

17. laRowSim /* Last Row Similarity */
18. laRowSim = rRowSim *
19. **while** rowSim > laRowSim do
20.    laRowSim = rowSim *
21.    rTC = rTC + 1 *
22.    rowSim = sum similarity of rTC to selTCs *
23. **end**
24. tmpSelTCs = selTCs + (rTC - 1)
25. **else if** rowSim > lRowSim < rRowSim then
26.    search left in the same way as before for the right sight *
27. **else**
28.    tmpSelTCs = selTCs + currentTestCase *
29. **end**
30. **end**

Figure 6.7: Group Hill Climbing Algorithm.

Equation 6.10 shows the computation time complexity with \( \mathcal{O} \) notation for the worst case where the algorithm needs the maximum steps (next neighbour is always better)

\[
C_{time} = \frac{p}{3} \cdot \frac{(n-p-1)}{5} \cdot \frac{p}{22} \mathcal{O}(n^3) \quad \text{for} \quad n > p + 1
\]  

(6.10)

where \( p \) is the target number of TCs and \( n \) is the number of TCs. The numbers below the braces indicate the line number in Figure 6.7.
6.3.4 Diversity-based Steady State Genetic Algorithm

Inspired by the evolution theory, genetic algorithm mimics natural evolution techniques such as mutation, selection, and crossover. A genetic algorithm utilise these techniques to apply heuristics search for optimisation problems. Within this section a new algorithm called Diversity-based Steady State Genetic algorithm is proposed. The new algorithm can be applied to the problem of test case group optimisation. Steady State thereby guarantees, that the next population is always better or equal than the previous one by deciding if either the children or the parents are transferred to the next population [171]. The algorithm relies on the basic steps of i) selection ii) crossover and iii) mutation.

```
Input: Similarity Scores
Input: Test Cases (TCs)
Input: p /* Target number Test Cases */
selTCs /* list of selected test cases */
selTCs = randomly select n test cases out of all
availTCs = all other test case
while newPopulation.size < p do
    ptc1 ← select first Parent
    ptc2 ← select second Parent
    childs = newChild(ptc1, ptc2)
    if mutation occurs then
        childs[1] = select random test case from availTCs
    end
    score[], tc[] ← calculate the score for the parent/childs by summing the similarity scores from all selTCs
    score1, tc1 ← select the best score/tc from score[],tc[]
    score2, tc2 ← select the second best score/tc from score[],tc[]
    if more then two TCs have the best score then
        tc1,tc2 ← handle equal score selection
    end
    add tc1 and tc2 to newPopulation
end
```

Figure 6.8: Identification of the Next Generation in the Genetic Algorithm.

Figure 6.8 shows the pseudo code for creation of the next generation within the proposed algorithm. At first, a random group of TCs is selected (line. 2). A new population is then stepwise created by selecting two parent TCs from all selected TCs. In this thesis, this is done with a tournament algorithm. The tournament algorithm randomly selects four potential parents from the selected TCs. The two TCs with the lowest similarity are then selected as parent TCs. The natural concept of crossover cannot be directly applied to a predefined group of TCs. Therefore, the crossover is declared as identifying TCs that have a low similarity to both parents. This ensures that the group can be changed significantly. The process of
generating new children is shown in Figure 6.9. Each pairwise joint similarity between all available TCs and both parents are compared with each other (line 4-15) and the two TCs with the lowest similarity to both TCs are selected as children. After the children have been selected, a random mutation of the first children can be realised based on a mutation occurrence probability (Figure 6.8 line 8-10). Finally, the summary similarity score of parents and children to the rest of the initially selected group are computed and the two TCs with the lowest summary similarity are transferred to the next population. This process is repeated until the target population size is reached.

Equation 6.11 shows the computation time complexity with $O$ notation:

$$C_{time} = j \cdot \left[ \frac{p}{2} \cdot (\underbrace{16}_{sel. \ parents} + \underbrace{2(n - p)}_{crossover} + \underbrace{4p}_{sel. \ best}) \right]$$

$$\in O(jpn) \text{ for } n > 3p \text{ and } O(jp^2) \text{ for } n \leq 3p$$

where $j$ is the number of populations, $p$ is the target number of TCs and $n$ is the number of TCs.
6.4 Computation Time Complexity Analysis

The individual analysis of theoretical computation time complexity of the different algorithms reveals that the complexity mainly depends on the number of TCs \((n)\) and the number of selected TCs \((p)\). Based on the identified computation time complexity two Figures comparing the different test case group selection algorithms are shown in this section. The value ranges are applied as specified within the last sections (e.g., \(n > p + 1\)). To provide a large data range a double logarithm representation is chosen. The following abbreviations are used: Greedy \((Gr)\), Group Greedy \((GGr)\), Random Search \((RS)\), Group Hill Climbing \((GHC)\), Diversity-based Steady State Genetic \((SSG)\). Additionally specified parameters are shown within the legend. The symbols correspond to the ones introduced within the previous sections.

Figure 6.10 shows the dependency of the computation time complexity as a function of the number of test cases where the selected number of test cases is determined. The figure shows that most algorithms linearly dependent on the number of test cases. One exception is \(RS\) with no dependency on the number of test cases. \(GHC\) shows the highest computation time complexity but the formula only specifies the worst case where the algorithm needs the maximum number of steps. The figure also shows that \(GGr\) results in a rather large computation time complexity due to the high dependency on the number of selected test cases.

Figure 6.11 shows the computation time complexity as a function of the number of selected test cases where the number of test cases is determined. The figure shows that \(RS, GHC,\) and \(GGr\) quadratically dependent on the number of selected TCs. All other algorithms have a lower dependency on the number of selected TCs where \(Gr\) shows the lowest dependency.

6.5 Summary and Discussion

The overall purpose of the outlined algorithms is to find a set of TCs, that is selected from all available TCs, and to increase the diversity compared to baseline approaches. The diversity has been assessed based on a pair-wise similarity investigation between all available test cases.

To ensure that the proposed algorithms can be understood and implemented by the research community a detailed description, including pseudo code, has been outlined, hence facilitating to reproduce the evaluation results compared to previous approaches such as the high level description used by Hemmati. The comparison of the proposed algorithms has also
Figure 6.10: Computation Time Complexity as a Function of the Number of Test Cases
Figure 6.11: Computation Time Complexity as a Function of the Number of Selected Test Cases
included the evaluation of the computation time. Theoretical analyses have been conducted to take the complexity of the algorithms into account when evaluating the algorithm performance (e.g., reducing of the average similarity score or increasing the failure detection rate).

As a starting point, the distance algorithms Hamming and Levenshtein have been adjusted to the problem of pair-wise test case similarity investigation. Hamming similarity has been extended to include two novel approaches for considering data values when calculating the similarity scores. The extension allows for the consideration of generated test data for the similarity investigation, therefore increasing the coverage of possible test targets to data related failures. While data value related failures are important to detect for many application domains, it is even more essential to identify such failures for IoT-based services. The reason is that both detection of physical values and actuation based on those sensed values constitute two key objectives of IoT-based services.

As discussed in Section 2.6.2 misunderstanding of algorithms partly results in wrong realisations. Since Hemmatis’ interpretation of Levenshtein does not follow the key idea of finding the smallest number of steps to change one TCs to the other (delete, add, etc.) as intended by Levenshtein, a new Levensthein algorithm has been proposed.

In contrast to earlier approaches, the proposed TC selection algorithms follow a group oriented optimisation approach. Instead of an iterative pair-wise comparison of the similarity score, the group of selected TCs is taken into account. Therefore, the novel algorithms Group Greedy, Group Hill Climbing and Diversity-based Steady State Genetic extend existing solutions in terms of the optimisation goal and its methodology. In order to ensure the proposed algorithms being comparable to baseline approaches, random search and a Greedy algorithm proposed by Hemmati et al. have also been presented with pseudo code and computation time complexity.

The theoretical computation time complexity analysis shows that all outlined similarity score computation algorithms depend quadratically on the number of possible TCs. The reason is, that all similarity score algorithms compare each TC with each other. Future work could create only parts of the comparison matrix depending on the need of the selected group optimisation technique. For example, the Group Hill Climbing algorithm requires only similarity score values to the neighbour TCs. While the computation time of the Levenshtein algorithm grows quadratically (all others are linear) with the number of allowed symbols, the additional performance gain, i.e. the enhancement of the similarity score, could outperform this drawback. Selection algorithms do mostly linearly depend on the number of TCs (for number of TCs $\gg$ number of selected TCs) but has an additional quadratic (Group Greedy, Random Search, Group Hill Climbing) dependency to the number of selected TCs.
Nevertheless, the goal is to select a rather small number of selected TCs from all available TCs and therefore all proposed algorithms scale linearly.

The following chapter will measure the performance of the proposed algorithms with various example services. The evaluation includes the investigation if the proposed algorithms can outperform existing solutions such as random search and Greedy. Also, scalability investigations are conducted to verify the theoretical analysis of the algorithm.
Chapter 7

Evaluation and Experimentation Results of AuST

This chapter presents and discusses the evaluation results of the performed research work. It consists of experiments validating the overall test process with the resulting standardised TTCN-3 and experiments to validate the proposed algorithms of the test case selection methodology.

7.1 Targets and Methodologies

The target of the chapter is to evaluate the designed and developed AuST testing tool for IoT-based services. The proof of concept is based on the achieved implementation efforts. At first, the general test process is verified with atomic and composite IoT-based services. Major intermediate steps of the test case derivation are outlined and discussed with the example services introduced in Chapter 2. As the major outcome of the test automation parts of derived TCs are shown. The evaluation of the test execution shows what kind of failures can be detected before and during test execution. Among the functional verification of the test process, computation time measurements are conducted to verify that the automated test derivation process can be executed in a reasonable time frame.

As the second part of the evaluation, the performance of the test case diversity optimisation are empirical measured and the results are shown. The performance measurements include investigations of the similarity score distribution, the influence of the different minimisation algorithm to the average similarity score among the selected TCs and the needed computation time. Based on different generic services, different aspects of the proposed algorithms are investigated and verified. The analysis of the failure detection rate concludes
the evaluation section. The failure detection rate verifies the claim that it is possible to select a group of TCs that has a higher failure detection rate than random selection if the TCs have most divers characteristics.

7.2 Evaluation of Test and Emulation Automation with AuST

7.2.1 Setup

The evaluation of the AuST takes place within the IoT.est testbed. The testbed is hosted by an Intel(R) Xeon(R) CPU X5460@3.16GHz with 16 GB RAM. The test component is placed on a separate virtual machine with one virtual CPU and five GB RAM. The SUTs are placed in the sandbox runtime-environment running on a different virtual machines that can be reached through a virtual network during test execution.

For computation time performance analysis, the outlined results are repeated 10 times to receive meaningful measurements. The results of the computation time can not be utilised to make general statements about the performance since the results are very dependent on software and hardware settings. Instead, the measurements can only be used to compare the performance among different setups. Within this section the variation is based on different SUTs. Since the testbed is realised with virtualisation, it can be expected that significant improvements can be achieved by hardware adaptations. Therefore, limitations to the scalability are discussed separately in Section 7.3 with a more enhanced hardware setup.

7.2.2 Example SUT I: An Atomic IoT-based Service

To emphasis testing of atomic services, the conducted experiment is based on an atomic service developed during the IoT.est project. As outlined in Section 2.1.3 the example service is a realisation of a camera control service, that can control multiple CCTV cameras at different locations. The service contains actuator characteristics through adjustable pan, zoom and tilt configurations via a RESTful interface and sensor characteristics through an interface which can be utilised to ask about the camera position (three set interfaces for pan, zoom, and tilt). As a consequence, if the service asks about a camera position, the response depends on previous interactions with the service. Therefore, the service can be interpreted as a stateful service. The next paragraphs highlight the major aspects of the test process.
IoT-behaviour Model The starting point of the evaluation is the IoT-behaviour model of the SUT. It is out of scope to verify the process of model derivation. Nevertheless, the utilised model is (semi-) automatically derived, based on the service description (WADL file) and a manually created sequence file, describing a dedicated order of interface utilisation. In this way, it is assured that at first the actuation interface to change the parameters of zoom, tilt and pan is invoked in a dedicated order before the actual camera position is request. This ordered steps can be categorised as an initiation sequence, required to ensure that the SUT is in a known state during test execution. The methodology is based on a manual sequence description (described in [139]) and is mainly developed by Mr. Daniel Kümp. Using a sequence description is one possibility to model the interconnection between the actuator and sensor characteristics although, it does not reflect the SUT behaviour since it is not limited by the order of interface invocations. For a more complete behaviour description it would require an initiation sequence that results in a behaviour model in a way that every combination of invocations are allowed. Figure 7.1 shows the resulting initiation sequence, where the interface to set zoom, pan, and tilt are represented as 1, 2, 3. To ensure that every sequence of invocation can be performed, each invocation that has not been invoked previously, results in a state change. In case S7 is reached it is ensured that all three interfaces have been invoked at least once.

![Figure 7.1: Complete Initialisation Sequence for Camera Service with Three Set Interfaces.](image)

For demonstration simplifications, only one order is realised and the resulting IoT-behaviour model is shown in Figure 7.2. The Figure shows state and transition elements of
the behaviour model for visual validation purposes shown in the AuST tool. The simplified
initiation sequence only contains the transitions setZoom_1_0, setPan_2_0, setTilt_3_0 and
getPosition_4_0 and therefore, the number of required transitions and states is reduced.
Another simplification of the realisation is, that the invocation of the position interface is not
modelled during S1,S2,S3. As a consequence, it is assured that the response of the position
can always be compared to one or more changes of the pan, tilt, zoom. Although the service
also allows requesting the position without previous changes it was not in scope of the test
evaluation to verify this circumstance and therefore, this behaviour is not modelled.

![Diagram of IoT-Behaviour Model of Camera Service Shown in AuST Editor View.](image)

**Test Case Derivation** To emphasis the test derivation process, the test paths are discovered
with the breadth-first algorithm. As a consequence, nine unique paths are discovered that
traverse each transition at least once on one of the created paths. In the following the
transformation step for the path initialState – > setZoom_1_0 – > end is outlined in some
details. For demonstration purposes the explanation focuses on important logical aspects and
therefore configuration steps are not discussed in detail. Note, that each path ends with the
end state to ensure that the automated generation process is finalised with the same end. It only reflects the end of a test case and does not necessary have to represent the behaviour of the service. Table 7.1 depicts the first step while going through the model elements in the current test case. The model object *InitialState* is used to create the general test case structure and assures that the test case stops after a defined time by adding a timer. Afterwards, the TTCN-3 element *function* is created and added to the test case. TTCN-3 functions are utilised to separate different steps of the test execution. These reusable functions are used to represent the different states of the SUT.

| Table 7.1: TTCN-3 Translation of Model Object *InitialState*. |
|---------------|---------------------------------------------------------------|
| **Action:** Add timer to enable timeout | **TTCN-3 Output:** testcaseMaxExecutionTimer.start; |
| **Create function** | **function** start_1_0() **runs on** c { ...} |
| **Add function to Test Case** | testcase tc_1() **runs on** c **system** sys { start_1_0(); ... } |

The next element is a *Transition* (*setZoom_1_0*) and consists of an *Event* that describes that an HTTP POST request is received by the SUT and an *Action* that describes the output reaction of the SUT to this input message. Table 7.2 sketches the transformation from the model object event to a send operation and the storage of a sent values for later usage. Since the IoT-behaviour model is created from the service point of view, the translator inverts certain expressions for the purpose of testing. In this case the event of a transition becomes a send call (*f_request(...)*).

| Table 7.2: TTCN-3 Translation of Model Object *Event*. |
|---------------|---------------------------------------------------------------|
| **Action:** Create send call and local variable | **TTCN-3 Output:** |
| **template** HttpRequest req_setPan_1_0 := { postRequest := { url := "http://10.1.1.42:80/CameraService/iot/Camera/zoom/10.11.127.6/1.27", ... } } v_PositionResponse_zoom := 1.27; f_request(p1, req_setZoom_1_0); v_req_setPan_1_0 := req_setZoom_1_0; |

Subsequently, the action part of the transition is utilised to derive TTCN-3 code. Initially a new function for the next state is created. Afterwards the defined response of the SUT is
translated into TTNC-3. Then, the TTCN-3 element *alt* is used to formulate the possibilities of the SUT behaviour. At first, the failure case for delayed or unexpected service responses is designed. After that the followed approach assumes deterministic service behaviour with only one possible valid reaction. This expected behaviour is included at the beginning of the *alt* element and includes a call to the next function (state) created before. Table 7.3 shows the discussed transformation process of the model object action.

Table 7.3: TTCN-3 Translation of Model Object Action.

<table>
<thead>
<tr>
<th>Action: Create Target Call</th>
<th>TTCN-3 Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S1_1_2();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Create expected response message</th>
</tr>
</thead>
<tbody>
<tr>
<td>var template GETResponse resp_setZoom_1_0 := {</td>
</tr>
<tr>
<td>statusCode := (200 .. 299),</td>
</tr>
<tr>
<td>content := {rawContent := omit, plainTextContent :=?},</td>
</tr>
<tr>
<td>headers := ?</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Form <em>alt</em> for Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt {</td>
</tr>
<tr>
<td>[] testcaseMaxExecutionTimer.timeout {</td>
</tr>
<tr>
<td>tcMaxExecutionTimeout_1();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>[] any port.receive { unexepctedStateReached_1(); }</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Create reply element in <em>alt</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>alt {</td>
</tr>
<tr>
<td>[ischosen(req_setZoom_1_0.postRequest)] p1.getreply(</td>
</tr>
<tr>
<td>POSTreq: {req_setZoom_1_0.postRequest} value</td>
</tr>
<tr>
<td>resp_setZoom_1_0) -&gt; value v_resp_setZoom_1_0 {</td>
</tr>
<tr>
<td>S1_1_2();</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

While the link to the next function has been created during the action transformation, in the last step the function itself is created at the time the next element (*NormalState*) of the test case is inspected. Table 7.4 reveals the resulting TTCN-3 output.

Table 7.4: Translation of Model Object NormalState.

<table>
<thead>
<tr>
<th>Action: Create function</th>
<th>TTCN-3 Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>function S1_1_2() runs on c {...}</td>
<td></td>
</tr>
</tbody>
</table>

At the final stage of the test case the model, object *EndState* is reached. This completes the TTCN-3 code creation by setting the verdict to pass. If all functions, corresponding requests and response messages have been transmitted during the test case execution this
final statement indicates that the SUT has the expected behaviour for this test case. Table 7.5 shows the resulting TTCN-3 code.

Table 7.5: TTCN-3 Translation of Model Object EndState.

<table>
<thead>
<tr>
<th>Action</th>
<th>TTCN-3 Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set verdict</td>
<td>setverdict(pass, &quot;End−state reached&quot;)</td>
</tr>
</tbody>
</table>

**Test Execution** During the execution the derived test cases can detect multiple failures. During the evaluation the test cases are executed against the SUT with real attached cameras and also with emulated resources. During the test execution with a real camera it is detected that setting the parameters *zoom*, *tilt*, and *pan* cause a change in the response of *getPosition* request but this change include a jitter effect that has not been modelled. Therefore, the test cases including the *getPosition* response failed during execution. If this jitter effect is not seen as a wrong behaviour by the service developer, easy adaptation of the test case can allow modelling such a behaviour since TTCN-3 elements exist that can reason about matches in value ranges.

The characteristics of detectable failures are further elaborated with a SUT that is only connected to emulated resources. To emphasis the capability to detect failures, the resource emulation and the SUT are manipulated to demonstrate different types of failures that can be detected with the derived TCs. Figure 7.3 shows the test execution of three different test case realisations (with different data values) of the nine paths discovered with a breadth-first search. In the button right (*TTCN-3 Graphical Logging*) a visual representation of test case 15 (*tc_15*) is shown. After the initiation sequence (*POSTreq zoom, pan, tilt*) the position is requested and a mismatch is detected. In the top, the mismatch is highlighted with red colour at the *zoom* parameter. Such kind of mismatch is also detected with the real cameras due to jitter.

Other kind of failures that are detected include:

- **Unexpected messages**: the SUT or its application server does not answer with the expected message. For HTTP messages, this means for example that the response is a 4xx message instead of 2xx (*tc_2* in Figure 7.3).
- **Timeout**: the SUT does not answer in the defined time duration.
- **Unexpected data type**: the content of the message does not represent the expected behaviour.
- **Unexpected data value**: the data values are not as expected.
• *Data values out of range:* the values are not in the expected data value range.

![Test Execution Results with Data Value Failure in TTworkbench Execution View.](image)

**Performance Evaluation**  The average computation time of the individual steps of the test derivation and execution are shown in Table 7.6 on the facing page. The experiment has been repeated 10 times to ensure convincing results. For the SUT used during the performance evaluation, the test data generation is statically integrated into the behaviour model. Therefore, no data generation takes place during the discussed test process. The results indicate that the most time is required to compile the standardised TCs with the external compiler *TTthree* from Testing Technologies. The test execution time is mainly limited by the number of involved communication interactions. Within this scenario the test cases are typically executed within 1 s. The results are influenced by the virtual network between the virtual machines and the involved resource emulation and may vary under different conditions.
Table 7.6: Computation Time Measurement of the Test Process with the Atomic Service Example.

<table>
<thead>
<tr>
<th>Component</th>
<th>ACT in s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Finder</td>
<td>$2.7 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Data Generation</td>
<td>–</td>
</tr>
<tr>
<td>Test Case Creator</td>
<td>$16.9 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Case Compiler</td>
<td>30.1</td>
</tr>
<tr>
<td>Test Execution</td>
<td>up to 10 s</td>
</tr>
</tbody>
</table>

Note: ACT: Average Computation Time.

7.2.3 Example SUT II: A Simple Composite IoT-based Service

The simple composite IoT-based service is utilised to verify the generation of TTCN-3. The service logic is as follows: the service requests the current status of a window (‘Open’, ‘Close’) and then altering the state of the window (e.g., close the window if it is open). Therefore, the service reacts to the current status of the window with different messages, which can be detected from the TEE.

Internet of Things Behaviour Model The IoTBM is shown in Figure 7.4. This model is translated into Velocity objects [11], that are translated into the TTCN-3 TCs, that are shown in the following paragraphs.

![Figure 7.4: IoT Behaviour Model for Simple Composite Service Shown in AuST Editor.](image)
Test Cases  The test cases can be derived with the W-method resulting in 105 TCs or with breadth-first search resulting in six unique TCs. Here, for visualisation purposes the TCs are derived with depth-first methodology. In the following, the main automatically created TTCN-3 files are shown (limited to one TC). The source files of all TCs can be found at http://kat.ee.surrey.ac.uk/AuST/SimpleIoTService.tar.gz. Listing 7.1 shows the definition of the system module. The module defines ports that are added to the component in order to enable exchange of messages and import instructions define the characteristics of the port. Here, two ports are defined, one is the default port, utilised in case there is a user interface where the SUT can be directly invoked. The second port represents the communication capability between the SUT and the connected resources. In this example the SUT is connected to an atomic service that can open and close a window and is also able to respond to a status request, answering the uncertainty whether the window is open or closed. From test execution perspective this communication to an atomic service requires that the TEE can emulate this behaviour and therefore is able to receive and send messages as expected from the SUT.

Listing 7.1: TTCN-3 m_c_sys.

```
module m_c_sys {

    import from m_p_HTTP all;
    import from m_p_HTTP all;

    type component sys {
        port HTTP p1;
        port HTTP Port_WA500;
    }
}
```

Listing 7.2 shows which message types can be utilised by the port (GETreq, POSTreq, PUT, DELETE).

Listing 7.3 and 7.4 show parts of one example TC, that is created out of the IoTBM. The test case is based on the path initialState → initial → initialised → sid → 5A... → CheckWindow → Woff → OpenWindow (cf. Figure 7.4). Three dots indicate, that one ore more lines are not shown to shorten the listing. The shown lines represent the main steps that need to be done to execute the test case. The TC also includes alternative paths and reactions (timeout, unexpected response, etc.) that are not shown within the following listings.

The module m_tc_3, shown in Listing 7.3, imports predefined functions, and other TTCN-3 files (line 3 to 7) such as the before discussed module m_c_sys. Then, the testcase is defined
and the ports are mapped. Afterwards, in line 14 the SUT is deployed and started with a developed external plug-in adapter of the TTworkbench. Subsequently, function `start_3_0` is executed to start the test verification. This function is a placeholder caused by the initial transition included in the IoTBM and can be utilised in case further initialisation steps are needed (e.g., manual adoptions of SUT to be testable). With the beginning of function `start_3_0` the test interaction begins. A HTTP request message is defined at line number 20-23. It is expected that this message will be send from the SUT to the connected atomic service. Since the aim is to emulate the atomic service, the TEE will receive this message and verifies the correct structure. In case the message is received, the function `CheckWindow_3_4` is called. A response message is created which includes an XML structure (line three in Listing 7.4). Due to the data generation, based on guards, the `v_WindowState_activeState` values have been set to ‘Close’ and sent to the SUT. It is expected that the SUT answers this response with a ‘Open’ statement (line 11-14 in Listing 7.4). In case the expected message is received, the TC ends with verdict ‘pass’ (line 25 in Listing 7.4).

**Test Execution**  During test execution, the derived test cases ensure that the composite service can be tested in the sandbox environment without interconnecting the SUT with the atomic service. The example TC, shown in the last paragraph, highlights that the TEE can act as the connected atomic service and can therefore define which SUT behaviour is currently under investigations. Consequently, it is ensured that the IoT-behaviour model is tested completely. With the concept of data value generation based on guards, it is also possible to detect failures in the behaviour model before test execution. It can therefore be assured that every modelled transition can be traversed (no dead code) during test execution. At the time of test execution the detectable failures of composite service are similar to the detectable failures of atomic service. Detectable failures include:
7. Evaluation Results

Listing 7.3: TTCN-3 Example Test Case Part I.

```tcl
module m_tc_3 {
  import from m_c_c all ;
  import from m_c_sys all ;
  import from m_p_HTTP all ;
  import from Functions all ;
  import from xsd0 language "XSD" all;
  testcase tc_3() runs on c system sys {
    map( mtc:p1, system:p1 );
    map( mtc:Port_WA500, system:Port_WA500 );
    ...
    var Functions.DeployResult v_res := f_deployRequest(SERVICE_ID, URI);
    start_3_0();
  }
  function start_3_0() runs on c {
    ...
    initialised_3_2();
  }
  function initialised_3_2() runs on c {
    var template HttpRequest req_sid_5A..._3_0 :=
    { getRequest := {
    ...
    }
    alt {
      [ischosen(req_sid_5A..._3_0.postRequest)] Port_WA500:getcall(POSTreq: {
        req_sid_5A..._3_0.postRequest}) \-> sender clientAddress_Port_WA500 {
        ...
        CheckWindow_3_4();
        ...
      }
    }
    ...
  }
}
```

- **Unexpected messages**: the SUT or its runtime environment does not answer or request with the expected message.

- **Timeout**: the SUT does not request/answer in the defined time duration.

- **Unexpected data type**: the content of a message does not represent the expected behaviour.

- **Unexpected data value**: the data values are not as expected.

- **Data values out of range**: the values are not in the expected data value range.
7.2 Evaluation of Test and Emulation Automation with AuST

```tcl
function CheckWindow_3_4() runs on c {
    ...
    function CheckWindow_3_4() runs on c {
    ...
    template WindowStateStatus WindowStateStatus_expectedValue := {
        ...
    };  
    if (v_WindowStateStatus_activeState == "Close") {
        template GETResponse resp_sid_5A..._3_0 := {
            ...
        };
        Port_WA500.reply(GETreq:{−} value resp_sid_5A..._3_0) to clientAddress_Port_WA500;
        var template HttpRequest req_WOff_9_1 :=
            ...
        alt {
            [ischosen(req_WOff_9_1.postRequest)] Port_WA500.getcall(POSTreq: {req_WOff_9_1.
                postRequest}) −> sender clientAddress_Port_WA500 {
                ...
            } ...
        }
    } 
    function OpenWindow_3_6() runs on c {
        end_3_8();
    }
    function end_3_8() runs on c {
        setverdict(pass, "End−state on path 'start − initial−> initialised −sid−5A...−> CheckWindow
                −WOff−> OpenWindow −toEnd−> end' reached");
    }
}
```

Listing 7.4: TTCN-3 Example Test Case Part II.

- **Logical Failures**: the SUT does not have the described logical behaviour. The unexpected behaviour can include, timing behaviour, messages and message data values.

What makes it more advanced compared to the concept for atomic service testing is that the emulation of atomic services is integrated into the test case derivation and execution process in an automated way. Therefore, it emphasises early stage testing in a sandbox environment.

**Performance Evaluation**  Table 7.7 on the next page shows the average computation time of the main steps of the test process. For this example, test data is generated during test case creation. To highlight the time for test data generation, it is shown individually. The test data generation is needed due to the guard involvement and is realised with *Drools* [165] rules. The utilisation of Drools engine resulted in rather long delays due to an initialisation, which has not been optimised. It is expected that further code improvements can significantly lower the time to generate data values. Similar to the atomic service example, also for composite service most of the computation time is required to compile the TTCN-3 TCs. The automated
deployment of the SUT and its required re-deployment after each TC results in rather high overhead for each TC. Especially the undeployment of the SUT can require up to 30 seconds due to internal timers of the sandbox runtime environment.

Table 7.7: Computation Time Measurement of the Test Process with the Simple Composite Service Example.

<table>
<thead>
<tr>
<th>Component</th>
<th>ACT in s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Finder</td>
<td>$0.3 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Data Generation</td>
<td>$20.1 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Case Creator</td>
<td>$12.4 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Case Compiler</td>
<td>55.7</td>
</tr>
<tr>
<td>Test Execution</td>
<td>up to 33 s for each TC</td>
</tr>
</tbody>
</table>

Note: ACT: Average Computation Time.

7.2.4 Example SUT III: A Complex Composite IoT-based Service

To emphasize that the followed approach can be applied to more complex services, a composite service with two different temperature sensors, one actuator to open a window, and one actuator to turn on an air condition is connected. Depending on the outside and inside temperature of a room, a window is opened or closed and the air condition is turned on or off. The corresponding IoTBM is shown in Figure 7.5 on the facing page.

Due to the IoTBM characteristics the test cases can be derived in the same manner as the simple composite service automatically. The integration of three different sensors and actuators results in the derivation of individual ports for each atomic service. The path finder identified 11 paths with the breadth-first search algorithm. The TCs can be downloaded at: http://kat.ee.surrey.ac.uk/AuST/ComplexExampleTestCases.tar.gz.

The Table 7.8 on the next page shows the average computation time of the test derivation and execution process. Also this more complex composite service can be tested with reasonable computation time. The results indicate that the compilation and execution are by far the most time consuming steps. The test execution requires slightly more time then for the simple composite service. The reason is based on the runtime environment requiring more time for deployment. Further scalability analysis in the next section are further supporting this finding.
Figure 7.5: IoT Behaviour Model for Simple Composite Service Shown in AuST Editor.

Table 7.8: Computation Time Measurement of the Test Process with Complex Composite Service Example.

<table>
<thead>
<tr>
<th>Component</th>
<th>ACT in s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Finder</td>
<td>$0.6 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Data Generation</td>
<td>$38.8 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Case Creator</td>
<td>$24.4 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>Test Case Compiler</td>
<td>76.0</td>
</tr>
<tr>
<td>Test Execution</td>
<td>up to 35 s for each TC</td>
</tr>
</tbody>
</table>

Note: ACT: Average Computation Time.
7.3 Empricial Test Process Execution Time Evaluation

To emphasis the performance of the test process realised with AuST three different generic services are utilised to measure the performance. The experiments are conducted with Intel Core i7-5820K running at 4.2 GHz (overclocked) and 64 GB RAM. The experiments are repeated ten times.

The first of the three different generic services is realised with an example state machine with five states, ten transitions (and one transition for initiation), two input messages and two output messages and as a result the W-method creates 132 TCs with full state and transition coverage. The example IoTBM is depicted in Figure 7.7. In order to create a more complex but valid model, a systematic model generation approach is applied for the two later generic services. Based on the defined number of input and output symbols a behaviour model is created. The procedure to derive the generic model is as follows:

i. A permutation table is created based on the allowed output symbols, where the order is relevant. The length of each row is equal to the number of output symbols.

ii. Each row is interpreted as state and each row of each column is a transition, where the first element of the column is the input symbol and the row value is the output symbol.

iii. For each state, the first transition target state is the current state and the following transition target state is linked to the \( n+i \) state, where \( n \) is the current state number and \( i \) the transition number of the current state.

**Example:** If the generic model has two inputs (message events) \([a,b]\) and two outputs (message actions) \([0,1]\) the permutation looks as shown in Figure 7.6:

<table>
<thead>
<tr>
<th>Output Symbols</th>
<th>a ( T_{1,1} )</th>
<th>b ( T_{1,2} )</th>
<th>( T_{1,2} )</th>
<th>( T_{1,2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Symbols</td>
<td>( T_{2,1} )</td>
<td>( T_{2,2} )</td>
<td>( T_{2,2} )</td>
<td>( T_{2,1} )</td>
</tr>
<tr>
<td>( S1 )</td>
<td>( T_{3,1} )</td>
<td>( T_{3,2} )</td>
<td>( T_{3,2} )</td>
<td>( T_{3,1} )</td>
</tr>
<tr>
<td>( S2 )</td>
<td>( T_{4,1} )</td>
<td>( T_{4,2} )</td>
<td>( T_{4,2} )</td>
<td>( T_{4,1} )</td>
</tr>
<tr>
<td>( S3 )</td>
<td>( S4 )</td>
<td>( S4 )</td>
<td>( S4 )</td>
<td>( S4 )</td>
</tr>
</tbody>
</table>

Figure 7.6: Permutation Table for Generic Service IoT-behaviour Model Derivation.
7.4 Test Case Diversity Experiments

The first state (gray background) is created out of the second row. The first transition of the first state \((T_{1,1})\) is based on the input symbol \(a\) and the expected output \(0\). The second transition of this state \((T_{1,2})\) is based on the input symbol \(b\) and the output symbol \(0\) and the target state is state two. The other states and transitions are created with the same procedure.

The empirical measurement of the average computation time is shown in Table 7.9. The table shows three different example services with an increasing number of transitions, states and respectively numbers of TCs from 132 TCs up to 5105 TCs (derived with W-method). The measurements support the statement that the designed steps of path derivation and TTCN-3 TC creation can be performed in reasonable time. In contrast, test case compilation and execution increases significantly with more complex services.

One possibility to overcome such a long test process is to reduce the number of TCs that are created, compiled and executed with the proposed test case diversity reduction methodology. The results of the test case diversity analysis are shown in the next Section.

Table 7.9: Computation Time Measurement of the Test Process.

<table>
<thead>
<tr>
<th>Component</th>
<th>SiSe ACT in s</th>
<th>SiSe STD in s</th>
<th>GS3x3 ACT in s</th>
<th>GS3x3 STD in s</th>
<th>GS4x4 ACT in s</th>
<th>GS4x4 STD in s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path Finder</td>
<td>0.7 \cdot 10^{-3}</td>
<td>0.18 \cdot 10^{-3}</td>
<td>2.0 \cdot 10^{-3}</td>
<td>0.4 \cdot 10^{-3}</td>
<td>0.047</td>
<td>0.4 \cdot 10^{-3}</td>
</tr>
<tr>
<td>TC Creator</td>
<td>0.41</td>
<td>0.013</td>
<td>1.19</td>
<td>0.8 \cdot 10^{-3}</td>
<td>70.05</td>
<td>0.44</td>
</tr>
<tr>
<td>TC Compiler</td>
<td>177.84</td>
<td>0.847</td>
<td>133.35</td>
<td>0.06</td>
<td>&gt; 720</td>
<td>10</td>
</tr>
<tr>
<td>Test Exec.</td>
<td>7</td>
<td>1</td>
<td>14</td>
<td>1</td>
<td>&gt; 5000</td>
<td>100</td>
</tr>
</tbody>
</table>

Note: ACT: Average Computation Time; STD: Standard Derivation; SiSe: Simple Service with 5 States, 10 Transitions, 2 Input Messages, 2 Output Messages and 132 TCs; GS3x3: Generic Service with 27 States, 81 Transitions, 3 Inputs Messages 3 Output Messages and 328 TCs, GS4x4: Generic Service with 256 States, 1024 Transitions, 4 Input Messages, 4 Output Messages and 5105 TCs.

7.4 Test Case Diversity Experiments

In the following the test case diversity experiments are outlined. It covers aspects of similarity score distribution, the average similarity score among the selected test cases and the resulting failure detection rate.
7.4.1 Empirical Similarity Scoring Distribution

The first experiment is utilised to investigate the similarity scoring distribution based on the different described similarity score algorithms. As described within the next subsection, an example IoTBM is developed. To make sure, that the results can be easily repeated with other state machines, it is a rather random order of states and transitions without having a specific application in mind. All experiments have been conducted with an Intel Core i7-3720QM @ 2.6 GHz Processor and 16 GiB RAM with Ubuntu 14.04 and the implementation is based on Eclipse Kepler, Java version 1.7, EMF version 2.1.3 and Equinox OSGi version 1.2.

Setup

![Diagram](image)

Figure 7.7: Example IoT-Behaviour Model.

An example state machine with five states, ten transitions (and one transition for initiation), two input messages and two output messages is implemented. As a result the W-method derives 132 TCs with full state and transition coverage. The example state machine is depicted in Figure 7.7. As an example, the input symbols are \([a,b]\) and output symbols are defined as two integer data types with data range restrictions \([5..1000],[-1000..2]\). In order to be executable in the AuST test tool, also transitions for the initialisation (e.g., starting of the SUT) and end transition exist but do not affect the experiments.

Evaluation Results

The outlined experiments, start after the derivation of the TCs. For each setup, the experiment has been repeated 10000 times to ensure convincing results. Figure 7.8 shows the Empirical Cumulative Density Function (ECDF) of the similarity score based on the different algorithms to compute the similarity score. The utilisation of the Hamming Similarity Scoring \(H_O\)
results in only twelve different possible similarity score values. This behaviour is caused by the omission of the order of elements in the TCs. A higher diversity of similarity score values can be achieved by applying the Levenshtein algorithm \((Le)\) where the order of elements is taken into account. Another characteristic of the Levenshtein algorithm is, that the resulting similarity score values have a smaller median value. Also, if the sample quantiles are connected with a regression, the slope of the resulting function is higher between 0.1 and 0.5 (axis of abscissae). Therefore, the test case optimisation algorithms need to handle similarity score values with smaller differences and at the same time being more diverse (due to more possible similarity score values). Another approach to have more distinctive similarity score values is based on the Data-Enhanced Hamming Similarity Scoring \(H_A\) and \(H_B\). Both algorithms result in a steep slope of the sample quantiles while \(H_A\) produces a lower similarity score for a low similarity, which is always lower than without data-enhancements (as discussed in Section 6.2.2). \(Le, H_A,\) and \(H_B\) have the advantage to be more distinctive and \(H_O\) requires less computation time to generate the similarity score values (cf. Table 7.10). Depending on the SUT, the integration of the order of symbols \((Le)\) or data integration \((H_A, H_B)\) result in more distinct similarity score values. It is therefore the question if more distinct values are worth additional computation time. The next section will focus on the TC group selection algorithms and also quantify the difference between \(Le\) and \(H_O\) based on the average similarity score.

![Figure 7.8: Empiric Cumulative Density Function of the Different Investigated Similarity Scoring Algorithms.](image)
Table 7.10: Median Computation Time of the different Similarity Scoring Algorithms.

<table>
<thead>
<tr>
<th>Similarity Score Algorithm</th>
<th>Median Execution Time in Milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Similarity Scoring (H₀)</td>
<td>1.9</td>
</tr>
<tr>
<td>Levenshtein (Le)</td>
<td>21.9</td>
</tr>
<tr>
<td>Data-Enriched Hamming Similarity Scoring (Hₐ)</td>
<td>137.6</td>
</tr>
<tr>
<td>Weighted Data-Enriched Hamming Similarity Scoring (Hₙ)</td>
<td>140.3</td>
</tr>
</tbody>
</table>

7.4.2 Test Case Group Selection

The similarity scoring values are the inputs for the selection optimisation algorithms. This section shows the characteristics in terms of their optimisation and measured time complexity of the different algorithms and highlights influences of the different similarity score algorithms.

Setup

The experiment is based on the example IoT behaviour model introduced in Section 7.4.1 with five states, ten transitions and two input messages (events) and two output messages (actions). To focus on the most appealing characteristics of the TC group optimisation algorithms, the output messages (actions) are defined as \([0,1]\). The results are shown with boxplots to visualise the distribution of the 10000 repetitions of the experiments. The boxplot whiskers show the lowest datum still within 1.5 Inter-Quartile Range (IQR) of the lower quartile and the highest datum still within 1.5 IQR of the upper quartile. Outliers are indicated with a circle. The time measurements indicate the implemented computation effort, although it does not replace theoretical analysis of the algorithm complexity. For each combination of the similarity score algorithm and minimisation algorithm, the experiment is repeated with the different target number of TCs from 7 to 120 TCs. Due to computation time limitations, the experiments are conducted with a step range of factor two between the target number of TCs. Although, the results follow a continuous curve and it is not expected to have a divergent behaviour between these values.

Evaluation Results

To emphasise the characteristics of the average similarity score of the selected TCs, Figure 7.9 shows the average similarity score with random selection of a group of TCs. The figure shows ten different experiments. Each of these experiments is conducted with a dedicated number of selected TCs (7, 15, 30, 60, or 120) and with the random selection algorithm (either with one RS or 100 RS100 iterations) and Hamming Similarity Scoring (H₀). RS is the
baseline performance if there is no optimisation strategy. It shows, that median of the average similarity scores slightly increases with a higher target number of selected TCs. At the same time, the diversity decreases since the influence of individual TCs decreases. Between seven and 60 selected TCs, the lower datum of the RS shows, that a group of TCs exists, which have a lower average similarity compared to the median of the average similarity scores detected by the RS algorithm. This lower datum prove, that it is possible to optimise the group of selected TCs.

Note, that this optimisation possibility decreases in case the target number of TCs increases since only a few TCs are not selected. For example, in case 120 TCs are selected only twelve are not selected (in this example) and therefore the variation possibility is low compared to a lower number of selected TCs. To prove that algorithms exist which can outperform random selection it is therefore more relevant to investigate small number of selected TCs (< half of all available TCs).

The second shown algorithm, Random Search with 100 Iterations (RS100), shows how close random search can get to the performance of the lower datum of RS with 100 iterations at each run, where the group with the lowest average similarity score is selected. The outliers indicate that random search could still be improved with a higher number of iterations.

![Figure 7.9: Average Similarity Score Between the Selected TCs with Hamming Distance Scoring and Random TCs Selection with one and 100 Iterations (Circles are Outliers).](image)

In the following, it is the goal to prove that there exists an algorithm that requires equal or less computation time compared to random search with 100 trials and can find an optimised group (lower average similarity score). The reason why random search with 100 iterations is selected is, that the conducted experiments indicated that the computation time is comparable to the proposed algorithms. Figure 7.10 shows the different TCs selection optimisation algorithms with different number of selected TCs and with Hamming Similarity Scoring
The Diversity-based Steady State Genetic (SSG) is performed with a population size of five (comparable computation time) and without mutation since the preliminary results indicated that mutation does not enhance the similarity score results. The Greedy Algorithm (Gr) shows a higher median of the average similarity scores compared to RS100 and also compared to random search without repetitions. The proposed Group Greedy (GGr) algorithm has the lowest median of the average similarity scores as well as the lowest diversity of the average similarity scores. Also, SSG significantly lowers the average similarity score compared to RS100 but has a slightly higher occurrence of outliers. While Group Hill Climbing (GHC) can outperform RS100 between 15 and 60 selected TCs, it shows a higher diversity and median of the average similarity scores for 7 TCs. Also, the occurrence probability of outliers is higher with Group Hill Climbing compared to any other investigated algorithm.

To emphasize the performance of SSG, GHC and GGr, compared to RS100, Table 7.11 shows the optimisation gain of the different algorithms. Thereby, the gain is the reduction of the median of the average similarity scores compared to RS100. It also includes a comparison between Hamming- ($H_0$, $H_A$, $H_B$), and Levenshtein-based Similarity Scoring.

Note, that this possible gain decreases in case the target number of TCs increases since only a few TCs are not selected. For example, in case 120 TCs are selected only twelve are not selected (in this example) and therefore the variation possibility is low compared to a lower number of selected TCs.

In case $H_0$ is used, GGr shows the best optimisation between 7 and 30 TCs (up to 11.9% gain with 30 selected TCs) and SSG is always the best solution for 60 and 120 selected TCs (up to 8.8% gain with 60 selected TCs). In addition, if Levenshtein ($Le$) is applied, SSG has always the highest gain (up to 11.6% for 7 and 15 selected TCs). It is therefore the best choice to select SSG with Levenshtein to optimise the similarity score within the discussed example service.

The data value integration of $H_A$ and $H_B$ does not positively influence the performance gain of the different selection algorithms (although still outperform random search up to 6.6% for 60 TCs). The higher diversity of the TCs (due to data value integration) reduces the benefit of the different selection algorithms and therefore it required to further elaborate if other data distance algorithms could improve the performance. Within this baseline experiment, a linear distance computation between individual data values is applied and further experiments could try to use a different approach (e.g., data value classes). This would result in a larger distance between individual similarity score values (due to the limited number of classes). An open research question is also how the data value integration into the similarity investigation can enhance the failure detection rate. It is rather complicated to invent a randomised example
Figure 7.10: Average Similarity Score Between the Selected TCs with Different Selection Algorithms with Hamming Similarity Scoring.
with data value related failures that can be used as a fair basis for evaluating the influences of data values and therefore it is left out for future research to identify such an example. Another future direction would be to identify and use existing services with known data value related failures for such an evaluation.

Table 7.11: Median Gain between Different Algorithm Combinations and RS100.

<table>
<thead>
<tr>
<th>Algorithm Combination</th>
<th>7</th>
<th>15</th>
<th>30</th>
<th>60</th>
<th>120</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSG $H_O$</td>
<td>7.8</td>
<td>10.3</td>
<td>11.6</td>
<td>8.8</td>
<td>1.5</td>
</tr>
<tr>
<td>GHC $H_O$</td>
<td>-7.8</td>
<td>-1.0</td>
<td>2.9</td>
<td>4.9</td>
<td>1.4</td>
</tr>
<tr>
<td>GGr $H_O$</td>
<td>8.9</td>
<td>10.7</td>
<td>11.9</td>
<td>8.1</td>
<td>0.2</td>
</tr>
<tr>
<td>SSG $H_A$</td>
<td>0.0</td>
<td>4.9</td>
<td>3.0</td>
<td>2.7</td>
<td>0.9</td>
</tr>
<tr>
<td>GHC $H_A$</td>
<td>-0.8</td>
<td>-0.3</td>
<td>2.1</td>
<td>2.6</td>
<td>0.8</td>
</tr>
<tr>
<td>GGr $H_A$</td>
<td>-0.8</td>
<td>-0.5</td>
<td>1.4</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td>SSG $H_B$</td>
<td>0.0</td>
<td>3.3</td>
<td>5.8</td>
<td>6.6</td>
<td>1.2</td>
</tr>
<tr>
<td>GHC $H_B$</td>
<td>-2.4</td>
<td>-2.8</td>
<td>1.4</td>
<td>4.0</td>
<td>1.2</td>
</tr>
<tr>
<td>GGr $H_B$</td>
<td>-1.3</td>
<td>1.0</td>
<td>2.2</td>
<td>1.8</td>
<td>-0.2</td>
</tr>
<tr>
<td>SSG Le</td>
<td>11.6</td>
<td>11.6</td>
<td>11.2</td>
<td>9.0</td>
<td>1.5</td>
</tr>
<tr>
<td>GHC Le</td>
<td>-0.8</td>
<td>4.1</td>
<td>6.4</td>
<td>6.5</td>
<td>1.4</td>
</tr>
<tr>
<td>GGr Le</td>
<td>9.3</td>
<td>10.1</td>
<td>10.1</td>
<td>7.8</td>
<td>0.0</td>
</tr>
</tbody>
</table>

The performance of the random search clearly depends on the number of trials and therefore, to prove that the proposed algorithms perform better than random search they need to have comparable computation times. Otherwise, the comparison should be performed with a higher number of trials. Table 7.12 therefore shows the median computation time in dependency of the number of selected TCs and the chosen optimisation algorithm. Within this setup, only GHC with 120 selected TCs requires more computation time than RS100. SSG and GGr always using less computation time and at the same time these algorithms decrease the median of the average similarity scores compared to RS100. For this setup, GGr is the best choice for test case group optimisation between 7 and 30 TCs (up to 10.5 times faster than with RS100 with 7 selected TCs and 8.9 % lower average similarity score) while SSG can outperform GGr and RS100 in terms of a lower average similarity score but with higher computation time compared to GGr. Nevertheless, also in the worst case with 120 selected TCs SSG is 1.7 times faster than RS100.

**Conclusion of Table 7.12:** All computation times lie in a relative narrow interval (0.01 - 2.37 ms) and all algorithms require less computation time than random search with 100 trials within the important area (7 - 60 TCs). As a consequence, the previously discussed gain of
the algorithms compared to random search is not caused by additional computation time but caused by the algorithm characteristic. Note that the performance could also be enhanced with random search and a higher number of trial but this would require addition computation time. The proposed GGR and SSG are therefore outperforming random search since they do not require more computation time (even less) and one the same time decreasing the similarity score between the selected TCs (up to 11.6%).

Table 7.12: Median Computation Time in Milliseconds as a Function of the Different Minimisation Algorithms with Different Numbers of Selected Test Cases with Hamming Similarity Scoring.

<table>
<thead>
<tr>
<th>Selection Algorithm</th>
<th>Number of Selected Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td>RS</td>
<td>0.01</td>
</tr>
<tr>
<td>RS100</td>
<td>0.21</td>
</tr>
<tr>
<td>GGr</td>
<td>0.02</td>
</tr>
<tr>
<td>GHC</td>
<td>0.01</td>
</tr>
<tr>
<td>SSG</td>
<td>0.10</td>
</tr>
</tbody>
</table>

### 7.4.3 Scalability

Within this section, a generic behaviour model derivation is applied to verify the performance of the algorithm for larger models. The computation time is investigated with two models with up to 5000 selected TCs.

**Setup**

In order to create a more complex but valid model, a systematic model generation approach, discussed in Section 7.3, is utilised. Based on the defined number of input and output symbols a behaviour model is created generically. Due to the generalised approach it is easy to derive IoT-behaviour models with a dedicated complexity and therefore it is possible to investigate scalability performance with an increasing model complexity.

**Evaluation Results**

To emphasis the scalability of the following test case diversity approach, two experiments with the generic model creation are performed. The first experiment is based on four inputs and three outputs resulting in 1605 TCs. Table 7.13 shows the median execution time of
the different TC group selection algorithms for different numbers of selected TCs with Levenshtein similarity scoring. In order to align the computation time, the experiment has been conducted with RS with 500 trials. Without the alignment it would have been unclear if the reduction of the average similarity score is only based on a higher computation time.

Table 7.13: Median Computation Time as a Function of the different Minimisation Algorithm with different Numbers of Selected Test Cases with Levenshtein Scoring for the Generic Model with Four Inputs and Three Outputs.

<table>
<thead>
<tr>
<th>Selection Algorithm</th>
<th>Median Computation Time in Milliseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Selected Test Cases</td>
</tr>
<tr>
<td>RS500</td>
<td>11.59</td>
</tr>
<tr>
<td>SSG</td>
<td>10.95</td>
</tr>
<tr>
<td>GHC</td>
<td>0.14</td>
</tr>
<tr>
<td>GGr</td>
<td>0.27</td>
</tr>
</tbody>
</table>

The comparison with 500 trials is a compromise since the computation time depends on the number of selected TCs and some algorithms perform better with small/large number of TCs. Future experiments could include the nearest computation approximation between all algorithms and all target number of selected TCs individually by changing the numbers of trials for the RS algorithm. The results shown in Table 7.13 indicate, that all algorithms require more computation time with an increasing number of selected TCs. Future improvements could try to reverse the optimisation goal (increase the average similarity) for numbers of selected TCs larger then \( n/2 \), where \( n \) is the number of all TCs. Instead of searching for the group of TCs with a low average similarity score, the optimisation could search for the opposite group, which has a high average similarity group and then remove those TCs. This reversion has the potential to enable the same computation time for \( k > n/2 \) then \( k < n/2 \), where \( k \) is the number of selected TCs. Due to the goal to significantly reduce the number of TCs, the performance between eight and 800 TCs is more relevant than larger numbers of selected TCs. 800 TCs being the most challenging target number of TCs, since the possible number of groups is maximised at \( n/2 \). While Random Search with 500 Iterations (RS500) requires the highest median computation time for eight TCs, only SSG requires less computation time than RS500. Therefore, to identify the best algorithm for a SUT the target number of TCs has to be taken into account when selecting the algorithm. Within this experiment, GGr has the lowest average similarity score (shown in Figure 7.11), while requiring more computation time for 200, 800 and 1600 selected TCs. GHC has the lowest median computation time between 8 and 200 TCs, although it has a lower gain then
SSG and GGr, compared to RS500. The GHC can be further improved by changing the stopping criteria of the algorithm. Further average similarity reductions can be achieved by inspecting a dedicated number of neighbours (e.g., five neighbours) before the optimisation is stopped.

![Figure 7.11: Average Similarity Score Between the Selected TCs with Different Selection Algorithms with Levenshtein Similarity Scoring for a Generic Model with Four Inputs and Three Outputs.](image)

In order to prove that the average similarity score can be decreased for the generic model in the same way as for the example SUT shown in Section 7.4.2, Figure 7.12 shows the average similarity score as a function of the different algorithms with different numbers of selected TCs with Hamming Similarity Scoring. The average similarity scores are higher compared to the previous example since the TCs are longer and the order of elements are not taken into account. Similar to results shown in Figure 7.9 the utilisation of GGr and
SSG results in the lowest average similarity, where GGr has the highest gain (6.1 %) to RS with 500 trials and 200 selected TCs. The experiments are conducted based on generic services and it can be further investigated if the methodology could be even more beneficial for other kind of services. For example, the design of the generic service tends to create rather subsequently IoT-behaviour models. The target states of the transitions are static neighbour states and thus do not allow very divers test cases. Future work could include more randomised model generation to highlight situations where the test case diversity optimisation might even be more beneficial.

![Average Similarity Score Between the Selected TCs with Different Selection Algorithms with Hamming Similarity Scoring for Generic Model with Four Inputs and Three Outputs.](image)

The second experiment for scalability investigations is based on a generic model created with four inputs and four outputs, which results in the creation of 5185 TCs. The median
computation time of the different algorithms are shown in Table 7.14. Both, \textit{GGr} and \textit{GHC} require less computation time than \textit{RS500} between 8 and 40 TCs while \textit{SSG} is more effective for 2500 and 5000 TCs. All algorithms have computation time lower than one second between 8 and 200 TCs (lowest: \textit{GGr} 0.7 ms with 8 selected TCs), and \textit{SSG} shows the lowest computation time for 2500 TCs (1.38 s, 3.5 times lower than \textit{RS500}). The table shows, that even the highest median computation time (\textit{GGr}) for 2500 selected TCs is 20 minutes. Compared to the time, that is required to execute these additional 2685 TCs, this is still very effective and therefore the followed test case diversity approach scale within the investigated range with up to 5000 TCs. In the domain of IoT-based services, it is not expected to have larger SUTs with the followed service concept. In order to apply the results in other domains, further experiments are required to verify the scalability also for even higher number of TCs.

Table 7.14: Median Computation Time as a Function of the Minimisation Algorithm with Different Numbers of Selected Test Cases with Hamming Similarity Scoring for the Generic Model with Four Inputs and Four Outputs.

<table>
<thead>
<tr>
<th>Selection Algorithm</th>
<th>Median Computation Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of Selected Test Cases</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>RS500</td>
<td>0.03</td>
</tr>
<tr>
<td>SSG</td>
<td>0.12</td>
</tr>
<tr>
<td>GHC</td>
<td>$0.3 \cdot 10^{-3}$</td>
</tr>
<tr>
<td>GGr</td>
<td>$0.7 \cdot 10^{-3}$</td>
</tr>
</tbody>
</table>

### 7.4.4 Failure Detection Rate

The goal to reduce the average similarity for a dedicated number of selected TCs is based on the assumption, that a lower average similarity is more divers and therefore can find more failures. To verify this assumption, different generic example services and their representation as an IoT behaviour model are generated. Afterwards, failures are inserted into the IoT-behaviour model and it is identified if the tests cases, that are selected by the test case diversity analysis, can identify the injected failure.

#### Setup

The experiment is conducted with the manual created example service shown in Section 7.4.2 and also with generically developed services as discussed in Section 7.4.3. A failure is inserted into the generic behaviour model. A generic service is created, based on Python 2.7...
and cherryPy [172]. At the beginning, it is verified that the selected test cases can detect the injected failures (in case that the 'right' TCs are selected). Nevertheless, to gain confident results, it is needed to repeat the experiment very often due to randomisation effects (for example, a random selection of the initial test case group) and this is very time consuming since multiple setups need to be verified. To overcome this time consuming approach the inserted failures are limited to transition failures that are based on the exchange of the target states to two transitions that have the same start state and a distinguishable output. This type of failure can be detected with every TC where the manipulated transitions are executed. It is therefore possible to analyse the selected test cases and identify whether the TC can detect the failure or not. As a consequence it is not necessary to execute those TCs any more. Instead, the failure detection can be directly specified. With the limitation to this kind of transition failure it is possible to repeat the experiment 2000 times and gain confidence about the results. The shown experiment results are therefore limited to the specified transition failures. Future work could include the theoretical analysis of failure detection of TCs for state failures. It requires storing the knowledge about the distinguishing sequence of states, created during the W-method and analyse the presence of those sequences in the selected TCs.

**Evaluation Results**

The first experiment is conducted with the example service introduced in Section 7.4.2 and 132 generated TCs with the W-method. Figure 7.13 shows the failure detection rate as a function of different selection algorithms and Hamming Similarity ($H_0$). It shows that $GGr$ can enhance the failure detection rate up to 29 % compared to random search with 100 iterations $RS_{100}$ and four selected TCs. It can almost guarantee (0.98) that the inserted failure can be detected with the selection of four TCs. Also, for the selection of two TCs the Group Greedy ($GGr$) algorithm can detect more failures than random search.

To verify that the results outlined in the last figures are not only caused by the specific SUT further experiments are conducted with the discussed generic model. Figure 7.14 shows the experimentation results with a service that consists of 78 transitions, 26 states and three defined inputs (as events) and three defined outputs (as actions). As a consequence 656 TCs are derived out of the IoT-behaviour model. The figure shows that, there is only little difference in the failure detection rate between Hill Climbing, Group Greedy and random search. Nevertheless, $GGr$ and $GHC$ still have a higher failure detection rate than random search. In case only 30 TCs are selected, the positive influence of the minimisation algorithm increases. $GGr$ increases the failure detection rate up to 16.8 % compared to random search.
7.4 Test Case Diversity Experiments

Figure 7.13: Failure Detection Rate with Group Greedy and Random Search with N=100 of an Example Service with 132 TCs.

Figure 7.14: Failure Detection Rate with Different Selection Algorithms and Hamming Similarity Scoring of an Generic Service with 656 TCs.
The Figure 7.15 shows that also the applied similarity algorithms significantly affect the failure detection rate. With the Levenshtein similarity algorithm the failure detection rate increases for $GGr$ from 0.782 to 0.895 for 60 TCs and from 0.66 to 0.78 for 30 TCs. One explanation for this behaviour is that Levensthein similarity results in more diverse similarity scores which enable a better selection of diverse TCs by the minimisation algorithm. The only algorithm where this behaviour cannot be detected is the Diversity-based Steady State Genetic Algorithm (SSG). The measured low similarity score among the selected TCs seems not to result in a higher detection rate than random selection. Possible reasons are that the investigation is focused on transition failures, due to their distinctive detection, but the diversity investigation also takes other model elements such as the states into account. Another reason is that the realisation of the tournament algorithm for parent selection tries to find diverse TCs. This can result in the selection of one short and one long TC and due to the steady state characteristic (a new population is never 'worse' than their parents) the parents can 'survive' their children and therefore short TCs are not eliminated. Future work could include to investigate if this limitation can be overcome with more flexibility in the population generation. The generation could either enable an increase of the similarity score for one population or include mutation into the parent and children selection. Another future research topic would be to investigate any patterns in the distribution of the selected TCs that enhance or limit the failure detection rate (e.g., only a local maximum is found or only TCs with transitions connected to unknown states are selected).

Overall the conducted experiments show that the failure detection rate can be significantly enhanced by applying the proposed minimisation algorithm, especially the Group Greedy algorithm. The enhancements are shown for transition based failures with different generic example services. To approve the findings at a broader scope, future work could include more diverse approaches to create generic services and also apply the test case diversity optimisation with existing services that include known failures.

7.5 Discussion of Evaluation Results

The evaluation outlined within this chapter aims at verifying the proposed concepts for early stage testing of IoT-based services in an automated manner.

At first, the designed test process is verified with one atomic service example and two composite services. The evaluation results show the major steps from the IoT-behaviour model to executable TCs. The automated derivation of the resulting TTCN-3 TCs are briefly explained and the integrated approach of resource and AS emulation are highlighted. The generated TTCN-3 proof that the IoT-behaviour model includes all relevant information to
7.5 Discussion of Evaluation Results

Figure 7.15: Failure Detection Rate with Different Selection Algorithms and Levenshtein Similarity Scoring of Generic Service with 656 TCs.

It is further shown that the IoT-behaviour model can be automatically transformed into executable TCs. Also, the automated integration of emulated AS is emphasised. The execution of the TCs show, that failures such as unexpected messages, data types or data values can be detected during test execution. Also, the process of test case derivation can ensure that the IoT-behaviour can be tested completely. This means, that the test data generation takes guard conditions into account and can therefore detect if data values exists that can fulfil these conditions. In case that the model includes guards that can not be satisfied, it is detected a priori (before test execution). To emphasise the scalability of the automated test process, the computation time is empirically determined. The results indicated that the designed steps of path analysis, data generation and TTCN-3 test case creation scale and the majority of the computation time is needed for the later steps to compile and execute the TCs. The findings support that it will not always be possible to compile and execute all available TCs. Instead, it is eminent to select a group of TCs, that have a high probability to detect failures, for test execution. Overall, the outlined experiments based on three different IoT-based services are a good basis to verify the developed testing tool AuST. Further work could include alternative example services to revamp the prototypical implementation to a product.
The second part of the evaluation focuses on the test case diversity optimisation methodology. Different aspects of the similarity investigation such as the similarity score distribution, the selection of a group of TCs and the resulting failure detection rate are empirically investigated.

The evaluation of the similarity scoring algorithms shows that Hamming similarity scoring results only in twelve different possible score values for an example service with 132 TCs. More diverse similarity score values can be achieved by applying Levenshtein. Another opportunity to enhance the similarity score diversity is based on the introduced concept of data value integration. Initial results show that the both proposed Date-Enhance Hamming Similarity Scoring result in a steep slope of the sample quantiles and are more distinctive than Hamming Similarity Scoring. The outlined integration of data values prove that data values can be taken into account for test case diversity analysis. Nevertheless, future work is required to identify if the additional resource effort is adequate to improve the overall selection process. Since the author did not have access to an example service with known failures based on data values or specific data range violations, the conclusive experiments are omitted and identified as future work.

The results of the test case diversity analysis show that the proposed Group Greedy (GGr) algorithm and Diversity-based Steady State Genetic (SSG) algorithm can outperform random search in terms of a lower average similarity score value for the selected TCs. For example, for a model with 132 TCs, Group Greedy shows the best optimisation between 7 and 30 selected TCs (up to 11.9 % gain with 30 selected TCs compared to random search with 100 trials). Diversity-based Steady State Genetic algorithm (SSG) is always the best solution for 60 and 120 selected TCs (up to 8.8 % gain with 60 selected TCs). In addition, with Levenshtein similarity scoring, SSG has always the highest gain (up to 11.6 % for 7 and 15 selected TCs) and require less computation time than random search with 100 trials.

The experiments on scalability show that the proposed GGr and SSG algorithms scale for up to 5000 selected TCs. For example, SSG requires 1.38 s (3.7 times faster then RS with 500 trials) to select 2500 TCs out of 5185 TCs and even using GGr for selecting 2500 TCs from 5185 is performed within 20 minutes. Which is still a high reduction of the test execution time due to the TCs reduction. Without the reduction the compilation and execution of derived TCs would have taken more than two hours.

The proposed approach shows that the GGr and SSG can find a group of TCs with a lower average similarity score and at the same time require less computation time than with random search.

The aim of the investigated test case reduction is to optimise the failure detection rate. Experiments to validate this goal are conducted based on two generic service examples. The
results show for the first service with 132 TCs that $GGr$ can enhance the failure detection rate up to 29% compared to random search with 100 repetitions in case four TCs are selected for execution. The findings are also confirmed by the second experiment conducted with a larger generic service. The measurements show that the highest improvement can be achieved with $GGr$ and Levenshtein similarity with 30 TCs (16.8% improvement compared to random search with 100 repetitions). The results prove that the proposed test case diversity optimisation can enhance the failure detection rate significantly. The experiments are conducted based on generic services and it can be further investigated if the methodology can be even more beneficial for other kind of services. For example, the target states of the transitions are relative neighbour states and thus does not allow very divers test cases. Future work could include more randomised model generation to highlight situations where the test case diversity optimisation might even be more beneficial.
Chapter 8

Conclusion

This chapter concludes the thesis and summarises the conducted work and findings for IoT-based service testing. Furthermore, future research directions are discussed.

8.1 Summary of Contributions and Findings

The potential affects of deploying malfunctioning IoT-based services into the physical environment demand for more advanced test capabilities. So far, only manually driven test approaches have been applied in the domain of IoT. The outlined work has therefore investigated how IoT service development can benefit from the capabilities of automated test derivation with Model-Based Testing (MBT) principles.

The main objective of this thesis was to provide means for domain specific automated testing at an early stage of the service lifecycle in a scalable manner. For this purpose, a framework for testing has been designed and subsequently implemented as the so called Automated and Scalable Model-Based Testing Tool for IoT-based Services (AuST). Scalability is ensured with a test case diversity methodology, that tries to find a group of diverse test cases. This way, the failure detection rate has proven significant enhancements.

As the baseline for the conducted work, state of the art in the fields of MBT, it’s automation and test case selection optimisation have been investigated with regard to re-utilisation. As a behaviour model, Extended Finite State Machine (EFSM) and LTS have been identified as the best candidates for modelling IoT-based services. However, none of the existing solutions is able to cover aspects of the control characteristics of composite IoT-based services and do not include means to enable testing at an early stage, neither, without deploying actuator and sensor nodes in a testbed.

As a consequence, a test framework has been introduced, covering aspects for testing IoT-based services which have not been addressed so far. As a starting point, a test-driven
lifecycle management approach has introduced the concept of early stage testing. The lifecycle management ensures a defined information exchange to derive IoT-based services and make those information also available for testing purposes. The intention to derive and execute tests – before the SUT is deployed in the productive runtime environment – emphasises the aim of early stage testing. Since the presented research work has been conducted within the IoT.est project, an integrated test architecture is proposed. It ensures a clear definition and separation of various steps for test derivation and execution. The concept for sandbox testing is made concrete with novel concepts for simplified emulation of IoT resources and atomic IoT-based services. The analysis of existing emulation concepts for sensor and actor emulation (a specific type of IoT-resource) has led to a new concept for test friendly atomic service design by code injection into the SUT itself. The chosen approach requires encapsulating the connection between the SUT and its associated IoT resources. Furthermore, a new IoT resource emulation interface has been designed to emulate the communication between the SUT and its associated IoT resources at a logical level, hence hiding syntactical communication details from the test process. Moreover, the concept of composite IoT-based service testing relies on an emulation concept. The SUT is hidden by automatically emulating connected AS. In this way, the whole behaviour model can be tested without the need for involving external atomic services. The proposed approach enables the time scale to be accelerated during test execution. In the scope of the framework design, the representation and application of available knowledge has been investigated. Such knowledge cover service interface description documents (e.g., WADL/WSDL files) or process description documents (e.g., BPMN files) which may be utilised for deriving the service behaviour model automatically. Atomic service behaviour model derivation has been proposed based on RIF-PRD. Both the resource emulation and the example service description lend weight to the argument that the rule language is a promising candidate for automated model derivation for IoT-based services. However, a resilient proof is out of scope of this thesis and further work is required to realise such a process task in an automated manner. The suggested test architecture together with the automated emulation concept have resulted in a well defined test process. As a starting point, a novel so called IoT behaviour model has been introduced, that covers aspects of modelling the behaviour of the IoT-based service from a test perspective as well as addressing needs to enable automated transformation of this model into standardised and executable TCs. The model also includes approaches for ensuring that the emulation of atomic services can be realised within the test derivation process without further intervention. The realisation of connected AS for CS testing has been identified as another important aspect of the test data generation. The concept of test data generation includes the ability to specify data values based on defined
data types, their physical representation (e.g., inclusion of knowledge that the value represents the temperature) and restrictions based on current path conditions. Current path conditions are motivated by the aim to create deterministic TCs. In other words, a path to be tested is pre-defined and therefore, the emulated AS needs to act according to this specific path and its requirements. As part of the concept, this path and the interrelated reactions with previous events (such as received data values) have to be explored. The required sensor values are identified and path related constraints are included in the data values restrictions, utilised by the test data generation. With this technique, emulated AS communication messages can be created automatically and each path of the model can be tested.

As proof of concept, a new testing tool called AuST has been designed and implemented. It contains implementations for automatically deriving and executing TCs from the proposed IoT-based behaviour model and encourages testing IoT-based services in a sandbox environment. A multi-modal model transformation process ensures that syntactical and logical aspects of the test case derivation process can be separated. In this way, it is ensured that further adaptations, addressing only one of those two aspects (syntactical or logical aspects) can be realised easily. The implementation also includes methodologies to automatically deploy the SUT in a sandbox environment and therefore ensures that the test process can be demonstrated as completely automated.

One of the shortcomings of MBT is that the number of TCs increase exponentially with the number of states and transitions of a SUT (with path analysis with W-method). To improve scalability of the advised automated test process, test case diversity optimisation is applied. The distance algorithms Hamming and Levenshtein are adjusted to the problem of pair-wise test case similarity investigation. Hamming similarity is extended to include two novel approaches for considering data values when calculating the similarity scores. Algorithms have been designed for finding a group of TCs that have a low average similarity score. Novel algorithms include Group Greedy, Diversity-Based Steady State Genetic and Group Hill Climbing. The algorithms are evaluated with regard to their computation time complexity. Pseudo code examples make it easy to share the findings with the research community.

The evaluation of the designed test process has been verified with one atomic service example and two composite services. The generated TTCN-3 document has proven that the IoT behaviour model includes all relevant information to represent IoT-based services for the purpose of function testing in a sandbox environment. It has further been shown that the IoT-behaviour model, realised with EFSM principles, can be automatically transformed into executable TCs. Also, the automated integration of emulated AS has been discussed. The execution of the TCs has shown, that failures such as unexpected messages, data types or data
values, can be detected during test execution. To investigate the scalability of the automated test process, the computation time has been empirically determined. The results indicate that the designed steps of path analysis, data generation and TTCN-3 test case creation scale well and the majority of the computation time is needed for the later steps to compile and execute the TCs. The findings support that it will not be possible to compile and execute all available TCs for more realistic and therefore larger services. Instead, it is eminent to select a group of TCs, that have a high probability to detect failures, for test execution. Overall, the outlined experiments based on three different IoT-based service prove the applicability and efficiency of the test framework and the developed testing tool AuST.

The second part of the evaluation has focused on the test case diversity optimisation methodology. Different aspects of the similarity investigation such as the similarity score distribution, the selection of a group of TCs and the resulting failure detection rate have been empirically investigated.

The evaluation of the similarity scoring algorithms shows that Hamming similarity scoring results only in 12 possible score values for an example service with 132 TCs. More diverse similarity score values can be achieved by applying Levenshtein. Another opportunity to enhance the similarity score diversity is based on the introduced concept of data value integration. Initial results show that the proposed Date-Enhance Hamming Similarity Scoring results in a steep slope of the sample quantiles and is more distinctive than Hamming Similarity Scoring. The conducted integration of data values proves that data values can be taken into account for test case diversity analysis.

The results of the test case diversity analysis show that the proposed Group Greedy (GGr) algorithm and Diversity-based Steady State Genetic (SSG) algorithm can outperform random search with regard to a lower average similarity score value for the selected TCs. For example, for a model with 132 TCs, GGr shows 11.9 % gain with 30 selected TCs compared to random search with 100 trials. The SSG is always the best solution for 60 and 120 selected TCs. In addition, SSG has always the highest gain (up to 11.6 % for 7 and 15 selected TCs) with Levenshtein similarity scoring and requires less computation time than random search with 100 trials.

The experiments on scalability show that the proposed GGr and SSG algorithms scale well for up to 5000 selected TCs. SSG is 3.7 times faster than RS with 500 trials to select 2500 TCs out of 5185 TCs and even using GGr for selecting 2500 TCs from 5185 is performed within 20 minutes – which is still a considerable reduction of the test execution time due to the TCs reduction. Without this decrease in number, the compilation and execution of derived TCs would have taken more than two hours.
The proposed approach shows that the $GGr$ and $SSG$ can find a group of TCs with a lower average similarity score and at the same time require less computation time than with random search.

The investigated test case selection has primarily aimed at optimising the failure detection rate. Experiments to validate this goal have been conducted based on two generic service examples. The results show for the first service with 132 TCs that $GGr$ can enhance the failure detection rate up to 29% compared to random search with 100 repetitions if four TCs are selected for execution. The findings are also confirmed by the second experiment conducted with a larger generic service. The measurements show that the best improvement can be achieved with $GGr$ and Levenshtein similarity with 30 TCs (16.8% improvement compared to random search with 100 repetitions). The results prove that the proposed test case diversity optimisation can enhance the failure detection rate significantly.

The outlined findings and contributions have been published at ten conferences, in one book chapter and submitted to two journals. The large number of publications emphasises the relevance of the addressed topics and shows the acceptance of the outlined work by the research community.

Overall, the proposed test process and its realisation with the AuST tool verifies the hypothesis that an automated emulation of external resources can enable an automated testing of IoT-based service at an early stage of the lifecycle. The findings prove that the selection of test case can be optimised, compared to baseline approaches, and can achieve a higher failure detection rate without increasing the overall test execution time.

8.2 Future Work Directions

The conducted work opens new avenues and constitutes a solid basis for future work. With the development of AuST, different aspects such as non-functional tests, case studies or extended test case diversity algorithms can be directly evaluated on top of the conducted work. The next subsections highlight the most relevant future work directions.

8.2.1 Case Studies

The AuST tool has proven that the test automation can be realised for the described example services. During development, it was already visible that the design of the tool is flexible and robust (e.g., failures of some aspects did not stop the whole tool, adjustments and extensions could be relatively easily integrated). However, the major purpose of the implementation
is the proof of concept and therefore the implementation is at a prototype level. The most eminent step to create a product out of it would be to verify the functionality with a broader set of example services and with sets of known failures of these example services. Case Studies on real IoT systems can significantly improve the applicability of the developed tool and can lead to a well-engineered testing tool.

8.2.2 Non-Functional Testing

The outlined MBT concept for early stage testing with automated emulation of connected resources ensures that functional tests can be realised for IoT-based services. Future work could extend the test process to address also non-functional objectives. Although the IoT-behaviour model is already capable of describing test targets, such as response times, future work is required to extend the tests with regard to load tests, how a IoT-based service can deal with contradicting sensor values (e.g., two sensors given different information about current situation), or failure recovery (e.g., how an IoT service can recover from failures) etc. Due to the realisation of the IoT-behaviour model with EMF, the multi-modal transformation process and the utilisation of TTCN-3, further extensions required for non-functional testing can be easily integrated.

8.2.3 Multi-modal Model Transformation

The test cases are derived from the IoT-behaviour model which is based on a multi-modal model transformation. In this way, a more strict separation between logical and TTCN-3 specific syntactical issues can be achieved. However, the design of the IoT-behaviour model is influenced by the goal to create TTCN-3 TCs. As an example, it would be difficult to define elements such as message representation without already taking the TTCN-3 language itself into account. Future work could try to define a language independent processing step for a even more clear separation. This would enhance its adaptability to new test case languages as well as increasing the possibility to improve/extend the code to other test aspects such as different test objectives, e.g., non-functional tests.

8.2.4 Test Case Diversity Investigations

The results of the test case diversity investigations show that the failure detection rate can be significantly increased by the novel algorithm called Group Greedy. Future work can
elaborate if alternative algorithms or adaptations exist which may provide further enhancements. Some of the proposed algorithms (e.g. Group Hill Climbing, Diversity-based Steady State) include various parameters (e.g. number of visited neighbours, mutation percentage, concept to find best matching parents) that can be further investigated. Moreover, other methodologies, e.g. cluster-based search or combinations of local and global optima search algorithm such as mimetic search, are promising candidates that deserve further elaboration. The outlined documentation of the proposed algorithms, including pseudo code, time complexity analysis and its realisations is an excellent basis to benchmark future findings.


Appendix A

Publications

In the following, all published and submitted papers (in the scope of IoT and testing) where the author has been participated are outlined and the contributions are highlighted.


Abstract: The development of context-aware applications is a difficult and error-prone task. The dynamics of the environmental context combined with the complexity of the applications poses a vast number of possibilities for mistakes during the creation of new applications. Therefore it is important to test applications before they are deployed in a life system. For this reason, this paper proposes a testing tool, which will allow for automatic generation of various test cases from application description documents. Semantic annotations are used to create specific test data for context-aware applications. A test case reduction methodology based on test case diversity investigations ensures scalability of the proposed automated testing approach.

Contribution to the state of the art described in this paper:

• Highlighted the developed testing tool in the scope of context provisioning systems (Introduction and Section II.),

• Explained the test case derivation process for ASs and CSs (Section III.),

• Described the proposed test case reduction methodology (Section IV.),

• Demonstrated initial results of the failure detection rate evaluation (Section V.).

Abstract: In Internet of Things (IoT), mobility, distribution and heterogeneity among components and users lead to several system concepts and approaches. The complexity of such ubiquitous system demands for enhanced systematic approaches for evaluation and testing. This paper outlines key concepts of evaluation and testing methodologies and revises current approaches for the IoT. IoT-based applications interact with the physical environment via sensor and actuator nodes and this changes the way testing need to take place. To overcome current drawbacks of testing concepts for the IoT-based applications, methodologies of model-based testing are identified for utilisation. Model-based testing concepts such as behaviour modelling, test automation and test case reduction techniques are reviewed and assessed in the scope of IoT-based application testing. Current limitations are outlined and future research directions are discussed.

Contribution to the state of the art described in this paper:

• Explain why testing and evaluation is challenging and important for IoT (Introduction and Section II.),

• Compared different evaluation and testing methodologies for IoT systems (Section III.),

• Identified potential methodologies for testing IoT-based applications and its shortcomings (Section IV.).


Abstract: Abstract—Test Case Diversity investigations promise to reduce the number of Test Cases to be executed whereby addressing one of the drawbacks of automated model-based testing. Based on the assumption that more diverse Test Cases have a higher probability to fail, algorithms for distance analysis and search based minimisation techniques can help to enhance the quality of selection. This work discusses the application of Hamming Distance and Levenshtein Distance to compute similarity scores and outlines how Random Search and Hill Climbing can be applied to the problem of group optimisation based on pairwise Test Case similarity scores. The evaluation results, conducted with a test framework for automated test derivation and execution for IoT-based services, indicates that proposed Group Hill Climbing algorithm can outperform Random Search and at the same time utilising less computation time. The inclusion of the sequence-based Levenshtein algorithm shows advantages over the utilisation of the set-based Hamming-inspired scoring methodology.

Contribution to the state of the art described in this paper:

• Proposed novel Group Hill Climbing Approach for test case selection,

• Corrected Levenshtein algorithm as intended by the initial proposal,
• Described methodology to develop and compare different algorithms for test case reduction,

• Provide evaluation results and discussed their meanings.


Abstract: Automated test derivation is expected to be one of the key drivers of a rapid creation of robust Internet of Things (IoT) applications. The paper describes a two-step approach how concepts for semantically described IoT services can be used to derive functional test cases to test services in a sandbox environment. In the first step, the description of the service is used to generate a state based model of the service behaviour and its interfaces. Therefore, a methodology to enrich service descriptions for (semi-) automated test derivation and the required IoT specific adaptations are discussed in detail. These descriptions are used to generate customised test data and to achieve full parameter combination coverage. In the second step, the generated extended finite state machine model is analysed to create test cases in a standardised testing notation. Utilising this two-step automation approach enables test developers to evaluate and influence resulting test cases. The implementation proves that the envisaged extension can amplify the usefulness of web services descriptions for the test derivation for IoT services by reducing the effort to create and execute test cases.

Contribution to the state of the art described in this paper:

• Proposed the characteristics of the required IoT-behaviour model (Section VI.),

• Detailed description how the behaviour model is utilised to derive executable TCs (Section V. and Section VI.C. and VI.D.).


Abstract: Automated test derivation is expected to be one of the key drivers of a rapid creation of robust Internet of Things (IoT) applications. The poster describes a two-step approach how concepts for semantically described IoT services can be used to derive functional test cases to test services in a sandbox environment. In the first step the description of the service is used to generate a state-based model of the service behaviour and its interfaces. Therefore, a methodology to enrich service descriptions for (semi-) automated test derivation and the required IoT specific adaptations are shown. These descriptions are used to generate customised test data and to achieve full parameter combination coverage. In the second step the generated extended finite state machine model is analysed to create test cases in a standardised testing notation. Utilising this two-step automation approach enables test developers to evaluate and influence resulting test cases. The implementation proves that the envisaged extension can amplify the usefulness of web services descriptions for the test derivation for IoT services by reducing the effort to create and execute test cases.
Major contributions to the poster paper:

- Contributed to the overall structure,
- Described the process steps for test case derivation and execution (Section V.B.-V.D.).


Abstract: Major challenges in developing services for the Internet of Things (IoT) are based on heterogeneous interfaces and radio technologies. This paper proposes a knowledge driven service life cycle, which enables a structured utilisation of semantic descriptions for re-usability and testing. Furthermore the approach facilitates the process of encapsulating IoT resources into services.

Contribution to the state of the art described in this paper:

- Contributed to the more precise definition of different life cycle phases, especially for the test derivation and test execution phase (Section 2.2).


Abstract: Encapsulating IoT resources in services with comprehensive semantic descriptions of the involved components is a promising approach to overcome current silo architectures in the IoT domain. More important than the enablement of service composition, automated test derivation is expected to be on of the key drivers of rapid creation of robust IoT applications. The paper describes how concepts for semantically described web services can be transferred into the IoT domain. Therefore, methodology to enrich service descriptions for (semi-) automated test derivation and the required IoT specific adaptations are discussed in detail. First prototypical implementations prove that the envisaged lightweight extension can amplify the usefulness of web services descriptions for the test derivation for IoT-based services.

Major contributions to the Paper:

- Discussed test derivation architecture (Section 5.),
- Contributed to the requirement analysis about the information needed to derive a behaviour model (Section 6.1),
- Provided further details about the test concept retaliation with example test cases (Section 6.2).


Abstract: The information generated from the Internet of Things (IoT) potentially enables a better understanding of the physical world for humans and supports creation of ambient intelligence for a wide range of applications in different domains. A semantics-enabled service layer is a promising
approach to facilitate seamless access and management of the information from the large, distributed and heterogeneous sources. This paper presents the efforts of the IoT.est project towards developing a framework for service creation and testing in an IoT environment. The architecture design extends the existing IoT reference architecture and enables a test-driven, semantics-based management of the entire service lifecycle. The validation of the architecture is shown through a dynamic test case generation and execution scenario.

**Major contributions to the Paper:**

- Discussed requirements of test-driven service creation (Section 3.3),
- Provided a case study of a simple service, including early emulation concepts (Section 6),
- Proposed architecture building blocks for testing (Section 5),
- Highlighting the followed sandbox testing concept (Section 6).


**Abstract:** Efficient testing of IoT-based services suffers from the underlying heterogeneous nature of IoT resources and hinders the process of rapid service creation and deployment. Real world effects, based on the behaviour of IoT-based services, tend to prevent the straightforward execution within the productive environment. Current solutions for testbeds, involving physical or virtual IoT resources, appear to require intense capacities for time and resources. This paper describes a new approach for testing IoT-based service based on a code insertion methodology, which can be derived from the semantic description of the IoT-based service. The proposed IoT resource emulation interface is described from the semantic, architectural and implementation perspective. The paper compares its applicability and efficiency with classical approaches and expose high emulation capabilities while minimising the testing effort.

**Major contributions to the Paper:**

- Provided an early stage testing concept for AS based on resource emulation (Section II. and Section III.),
- Proposed code insertion to provide test friendly AS interfaces (Section II.),
- Prove the applicability of production rules for the followed emulation concept based on an example service (Section II.A. and II.B.),
- Discussed different possibilities to test services with connected resources and compared their advantages and drawbacks (Section I. and V.).

Abstract: Concepts for Internet of Things (IoT) are currently limited to particular domains and are tailored to meet only limited requirements of their narrow applications. To overcome current silo architectures, we propose a business oriented service composition of IoT enabled services with (semi-) automated model based testing capabilities. Explicit description of services as well as the target environment allows for automated design and execution of tests, hence enabling fast and robust IoT based service provision. This work proposes a semantic description of the test design and execution process to enable reasoning of test behaviour and suitability in the different phases of a service life cycle. The proposed work describes a test model and an appropriate test architecture. A first testbed implementation demonstrates their applicability. The proposed approach enriches current views of IoT architectures with knowledge from the field of service oriented architectures and makes them usable in distributed environments with partial unreliable resources by introducing a formalised integration of automated testing into the life cycle management.

Contribution to the state of the art described in this paper:

• Provided an initial test architecture for IoT-based service testing and a description of the test process (Section V.),

• Proposed a test driven service life-cycle management (Section IV.A),

• Discussed an extended semantic test model (Section III.B),

• Highlights early experiments with the design of test cases for a context provisioning middleware (Section VI.).


Abstract: Semantic modelling for the Internet of Things has become fundamental to resolve the problem of interoperability given the distributed and heterogeneous nature of the “Things”. Most of the current research has primarily focused on devices and resources modelling while paid less attention on access and utilisation of the information generated by the things. The idea that things are able to expose standard service interfaces coincides with the service oriented computing and more importantly, represents a scalable means for business services and applications that need context awareness and intelligence to access and consume the physical world information. We present the design of a comprehensive description ontology for knowledge representation in the domain of Internet of Things and discuss how it can be used to support tasks such as service discovery, testing and dynamic composition.

Contribution to the state of the art described in this paper:

• Proposed initial ontology for service testing with defined test targets and relations to the SUT (Section IV.C.),
• Discussed opportunities to use these test ontologies for model derivation and test documentation (Section IV.C. and Section V.),

• Provide paragraph about known ontology-based testing techniques (Section II.).


**Abstract:** To date implementations of Internet of Things (IoT) architectures are confined to particular application areas and tailored to meet only the limited requirements of their narrow applications. To overcome technology and sector boundaries this paper proposes a dynamic service creation environment that employs i) orchestration of business services based on re-usable IoT service components, ii) self-management capable components for automated configuration and testing of services for things, and iii) abstraction of the heterogeneity of underlying technologies to ensure interoperability. To ensure reliability and robustness the presented approach integrates self-testing and self-adaptation in all service life cycle phases. The service life cycle management distinguishes the IoT service creation phase (design-time) and the IoT service provision phase (run-time). For test-friendly service creation (1) semantic service descriptions are employed to derive semi-automatically services and related tests, (2) and testing is systematically integrated into a Service Creation Environment. For reliable and robust service provisioning the presented system (3) forces validation tests in a sandbox environment before deployment and (4) enables run-time monitoring for service adaptation. The system under test is modelled by finite state machines (FSM) that are semi-automatically composed of re-usable test components. Then path searching algorithms are applied to derive automatically tests from the FSM model. The resulting tests are specified in the test control notation TTCN-3 and compiled to run the validation tests.

**Contribution to the state of the art described in this paper:**

• Provide an example service that could be tested (Section 5),

• Initial experimentation about testing tool and required views (Section 5),

• Initial thoughts to test service at design time and identified BPMN as candidate for process description (Section 5).