Generating Test Adapters for ModelJUnit

Ardalan Hashemi Aghdam

Halmstad University, June 2017
ABSTRACT

Concretization is one of the most labor-intensive phases of the model-based testing process. This study concentrates on concretization of the abstract tests generated from the test models. The purpose of the study is to design and implement a structure to automate this phase which can reduce the required effort specially in every system update. The structure is completed and discussed as an extension on a model-based testing tool named ModelJUnit using adaptation approach. In this structure, the focus is mainly on bridging the gap in data-level between the SUT and the model.
ACKNOWLEDGEMENTS

I would like to thank my thesis supervisors Dr. Mohammad Reza Mousavi and Ms. Mahsa Varshosaz for the continuous support of my thesis, for their patience and immense knowledge.
I must express my gratitude to my parents, for their continuous support and encouragement.
CONTENTS

1 INTRODUCTION 1
   1.1 Background 1
   1.2 Problem Statement 2
   1.3 Outline 2

2 MODEL-BASED TESTING 3
   2.1 The MBT Process 3
   2.2 Test Model 4
      2.2.1 Model Abstraction 5
      2.2.1.1 Functional Abstraction 5
      2.2.1.2 Data Abstraction 5
      2.2.1.3 Communication Abstraction 6
      2.2.1.4 Temporal Abstraction 6
   2.3 Test Selection Criteria 6
      2.3.1 Functional Criteria (Explicit test case specifications) 6
      2.3.2 Coverage-based criteria 7
         2.3.2.1 Structural coverage criteria 7
         2.3.2.2 Data coverage criteria 7
      2.3.3 Stochastic and random criteria 8
   2.4 From Abstract Tests to Executable Concrete Tests 8

3 AUTOMATION OF TEST ADAPTATION (MODELJUNIT EXTENSION) 11
   3.1 General Workflow 12
   3.2 Abstract Data Generation 13
   3.3 Data Conversion 14
   3.4 Defining an Abstract Operation for the Execution 15
   3.5 The Extension Storage 17
   3.6 Data Setting: Fresh 17
   3.7 Test Case Adaptation Tracing 17
   3.8 Simple Case Study: Stack 18
      3.8.1 Stack EFSM Model 18
      3.8.2 How the Auto-Adapter Works In Practice 20
         3.8.2.1 Scenario 1 20
         3.8.2.2 Scenario 2 22
         3.8.2.3 Scenario 3 25
   3.9 Conclusion 26

4 MODELJUNIT EXTENSION IN PRACTICE & EVALUATION 27
   4.1 Conventional Adapter 27
   4.2 Auto-Adapter 27
4.3 The Auto-Adapter Interface: Abstract Operation 28
  4.3.1 Abstract Operation in Modeling Stage 28
    4.3.1.1 Benefits of Combining the Data Requirement with Abstract Operation 29
  4.3.2 Abstract Operation in Adaptation Stage 29
4.4 Case Study: CRUD 30
  4.4.1 CRUD Model 30
  4.4.2 Abstract vs. SUT Operations 31
  4.4.3 Auto-Adapter Converters vs. Conventional Adapter Implementation 32
  4.4.4 Auto-Adapter Interface vs. Conventional Adapter Interface 34
  4.4.5 Model Code: Size Comparison 35
4.5 Conclusion 36
4.6 Comparison with Randoop for Future Work 36
4.7 Link for the Extension 37

BIBLIOGRAPHY 39
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The Process of Model-Based Testing</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The Extension Workflow</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Input Data Generation and Input/Output Adaptation</td>
<td>13</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Stack EFSM</td>
<td>18</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Stack EFSM with Illegal State</td>
<td>25</td>
</tr>
<tr>
<td>Figure 6</td>
<td>The CRUD Model</td>
<td>30</td>
</tr>
<tr>
<td>Figure 7</td>
<td>CRUD Model Operations Description</td>
<td>31</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Abstract and Equivalent SUT Operations</td>
<td>31</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Flight Booking SUT Conversions for CRUD Model</td>
<td>33</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 1  Conventional Adapter and The Extension Converters Code Size 33
Table 2  Results of Conventional Adapter 34
Table 3  Results of Auto-Adapter 34
Table 4  Model Size for Flight Booking SUT 35
Table 5  Model Size for Grade Manager SUT 36
<table>
<thead>
<tr>
<th>Listings</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The Extension Data Generator Samples</td>
<td>14</td>
</tr>
<tr>
<td>3.2 The Extension Data Converter Sample</td>
<td>15</td>
</tr>
<tr>
<td>3.3 Abstract Operation Class Methods</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Stack EFSM ModelJUnit Actions</td>
<td>19</td>
</tr>
<tr>
<td>3.5 Model Operations of Scenario One</td>
<td>20</td>
</tr>
<tr>
<td>3.6 ModelJUnit Execution Results</td>
<td>21</td>
</tr>
<tr>
<td>3.7 The Extension Tracing Log File</td>
<td>22</td>
</tr>
<tr>
<td>3.8 Push and Pop Model Operations in Scenario Two</td>
<td>23</td>
</tr>
<tr>
<td>3.9 Assertions in Scenario Two</td>
<td>23</td>
</tr>
<tr>
<td>3.10 Adaptation Steps of the Model Push and Pop Operations</td>
<td>24</td>
</tr>
<tr>
<td>3.11 Illegal State Exception Handling</td>
<td>25</td>
</tr>
<tr>
<td>3.12 Tracked Method Exception in Log File</td>
<td>25</td>
</tr>
<tr>
<td>4.1 Create Abstract Op with Flight Booking Concrete Configuration</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Modified Conventional Adapter Interface For Flight Booking SUT</td>
<td>34</td>
</tr>
</tbody>
</table>
ACRONYMS

MBT  Model-based Testing
SUT  System Under Test
EFSM Extended Finite State Machine
CRUD Create Retrieve Update Delete
INTRODUCTION

1.1 BACKGROUND

Testing is one of the most important phases of software development cycle. It is crucial to detect possible faults in software before they lead to failures in the system and consequently to huge financial or even vital costs. Testing is performed on different levels, which can vary from testing a simple functionality to testing the system as a whole. Nowadays, testing is mostly automated and the test scripts are written once and maintained for future updates. This could be more time saving rather than tedious and insufficient manual testing at each change of software. However, in reality due to lack of time and strict deadlines the developers might not be able to design test cases precisely. Model-based testing (MBT) is a technique in which test cases are generated automatically from an abstract model of the system. Basically, it is used for functional black-box testing. Using MBT, mentioned drawbacks of previous techniques could be eliminated if the model is designed efficiently and the time spent on designing of the model could be less than writing test scripts. However, there are some challenges of using this technique that should be considered. One of the challenges is that the generated test cases can be too abstract and not compatible with actual implementation called system under test (SUT). There are several existing MBT tools [1, 2]; many of which are developed for academic and research purposes. These tools have different capabilities that differ in some aspects such as modeling language, existence or lack of model simulation, capability of execution of test on SUT (Online or Offline) [3]. One of the academic MBT tools is ModelJUnit [1] which extends JUnit to support model-based testing. The notation used for modeling is Extended Finite State Machine (EFSM) and the model is implemented in Java. Adapters can be used to make abstract test cases executable on SUT; in other words, the adapter makes it viable to convert abstract function calls to concrete ones for real-world implementations while generating and executing test cases (Online Testing). However, abstractions in behavior of SUT could lead to difficulty in development of adapters.
1.2 PROBLEM STATEMENT

In model-based testing, test models are abstractions of the behavior of the system. Due to this abstraction, generated test cases lack the adequate concrete details to be executable on the implementation which facilitates the software verification. Thus, abstract test cases should be concretized to fulfill the testing process and make the execution feasible. As the abstraction level of the model or the complexity level of SUT increases, this phase requires more effort. One of the approaches that can be used to overcome the problem is to exploit an adapter to close the gap between the model and the SUT (adaptation approach). Test cases generated from the model contain a set of abstract operations and these operations represent the SUT operations in a simplified manner. Basically, the adapter converts the abstract operation calls to concrete operation calls with correct format and data. However, implementing/maintaining the adapter is an error-prone and also a tedious job. Thus, effective ways to automate the adaptation phase needs to be investigated.

1.3 OUTLINE

In this thesis, a structure to automate the test adaptation phase of model-based testing will be proposed. The structure will be designed by evaluating the essential steps of adaptation that are required to be done to make the abstract test cases executable. As mostly the data abstraction leads to difficulties in adapter development, this issue will be the backbone throughout the study. Also, the proposed structure will be implemented as an extension for academic MBT tool, ModelJUnit.
Model-based testing is a black box testing technique. In black-box testing the software functionality is examined without any knowledge of internal implementation and details of the code are hidden in a process box in which only inputs and outputs are taken into account. Simply, MBT is a technique that automates the design of black-box tests. In this technique test cases are generated automatically from an abstract model of the system. The model is abstract as it only illustrates the critical aspects of the SUT and it is designed from the requirements of the software which should be also met by the implementation. The MBT has many advantages as it automates the testing process only using a test model and reduces the required intensive effort of code-centric approaches. It noticeably lessens the maintenance costs of the test suites and supports regression testing. However, still challenges for this context exist and it will be discussed more in details in the following parts of the chapter.

2.1 THE MBT PROCESS

Model-based testing consists of five main steps [4]:

1. Modeling the SUT
2. Generating abstract test cases from the model
3. Concretizing abstract test cases
4. Executing concrete tests on the SUT
5. Analyzing the test results

The above-mentioned steps are illustrated in Figure 1. In the first step, an abstract model of the system is designed. This step is important as the model should be conforming to the requirements of the system. Also, modeling could reveal the ambiguities and contradictions in the requirements of the system. Hence, it gives the opportunity to reconsider requirements and improve them [5]. In the second step, some criteria are considered to generate test cases from the designed model. For example, the focus could be on a particular part of the model or a specific coverage criterion depending on the language used as model representation. Moreover, different heuristics could be used for the generation step. The third step is concretization step in which abstract tests become executable. After this step, concrete test cases are present and ready for step four to get executed on SUT. Finally, in the
last step, the results of the execution are evaluated and main reasons of test fails are investigated. Mainly, test fails can be originated from two major reasons, fault in SUT or fault in test case. Also, in worst case scenario failed test could be a result of both which even makes it more complex to know where it is originated from. Test case faults are due to failures in models or adapters. Hence, efficient design and development of models become an important issue in MBT.

![Diagram of Model-Based Testing Process]

Figure 1: The Process of Model-Based Testing

2.2 TEST MODEL

The test model is an abstraction of the behavior of the system, which is designed with respect to the requirements and specification of the SUT. There are numerous modeling languages that have been introduced to describe a test model. Some of them have been successfully used in a group of application domains such as communication protocols, processors and graphical user interfaces. According to different nature of application domains, the choice of modeling notation and language
is crucial. One of the key features of the test model is to be abstract. The meaning of abstraction is discussed in the following subsection in details, since it could be dependent on the knowledge of target system combined with the formality degree of requirements/specification. Two main approaches exist for deriving test models. In case of availability of a formal specification of the system the model could be derived by adapting it for the testing purpose. However, extra treatment is necessary but this could ease the automation of MBT. Also, there is another approach that test model is designed using specification that is written in a natural language such as English [6]. However, the second one is more labor-intensive but it could reveal the ambiguities and contradictions in the requirements of the system.

2.2.1 Model Abstraction

Code-oriented testing techniques deal with actual system implementation. Nowadays, due to increasing complexity of systems, the mentioned techniques lead to an unstructured way of testing. MBT facilitates testing process with a focus on critical behaviors of the system using a simplified and abstract model. The abstraction level is important as it determines how the following three phases of MBT are shaped: designing a model, concretization of test cases and evaluation of the actual and expected results. Also, it reduces the effort as it is not necessary to directly validate the complex implementation. However, this is an optimist view as the mentioned effort is mostly eliminated from modeling phase and could still exist in the concretization phase. The mentioned problem will be discussed in the following sections in details. In [7], the principles of abstraction are categorized by referring to MBT case studies on different application domains. The following subsection contains a brief explanation of possible principles that could be used for model abstraction.

2.2.1.1 Functional Abstraction

This type of abstraction is about the omitted behaviors of system which would not be useful in the test model. This principle is mostly used because of the existence of non-critical functionalities in most of cases.

2.2.1.2 Data Abstraction

Data abstraction is also the one of common principles used in the modeling phase. Systems could contain complex data types and it is desired to reduce the complexity level of these data types in test model. However, this could lead to the information loss and extra effort is required to concretize them to be able to execute test cases on SUT. This abstraction can be classified as output and input abstraction.
A major input abstraction can be the case that concrete data types are represented as equivalence classes. Also, output abstraction which omits or simplifies the outputs of SUT could decrease the power of test evaluation with regard to actual and expected behavior [8].

2.2.1.3 Communication Abstraction

This abstraction is related to the domain of communication systems in the context of protocol testing in which lower levels in ISO/OSI reference model are simplified and abstracted. The operations of corresponding levels are represented atomically by a single signal or ignored at all.

2.2.1.4 Temporal Abstraction

In the systems that timing of events has the main role in testing; precise timing of the system events might not be used as same in the abstract model. The difference between the abstract and concrete time steps can prohibit the detection of faults and mapping between them is required which is even more challenging in the domain of distributed real-time systems.

2.3 TEST SELECTION CRITERIA

Along with the test model, the test selection criteria must be defined to drive the test generation. The selection criteria determine that what should be tested and gives the control to choose the tests. Also, it directs the reduction that should be done to decrease the number or length of the test cases generated from the abstract model. As an example, the reduction could be done by considering a functional constraint or specifying the maximum aimed coverage or randomly. None of the selection criteria rely on the SUT implementation and they are related to the test model of the system. The most used criteria for test selection is given in the following and the list could be extended [4]:

- Functional Criteria: Explicit test case specifications
- Coverage-based criteria: Structural and Data coverage
- Stochastic and random criteria

2.3.1 Functional Criteria (Explicit test case specifications)

In these criteria, the selection of the test is made explicitly by using a formal notation. It is described formally as a test objective to control the test generation. The possible examples could be: it can restrict the explored paths of the model by the functional constraint or it
can assure that the particular behavior is tested. In practice, in some cases, the formalism for test case specifications and the test model are indistinguishable as they are defined with same notations. However, they also might use different notations depending on the tool being used.

2.3.2 Coverage-based criteria

These criteria measures that how well the generated tests covers the model. As in code-oriented techniques we need the white-box coverage statistics to measure how well the system code is tested, also in MBT, it is required to have the coverage criteria where the focus is only on model. However, both are not contradicting but they are complementary to have an idea about the adequacy of the testing process. Moreover, it is good to mention, most of the coverage criteria in MBT are adopted from white-box testing. The reason is the possible resemblance of the modeling notation to the programming language features. In addition, the coverage-based criteria also can be applied to the test generation phase to limit the number of test cases as a termination condition. In the following two mostly used model coverage criteria are discussed.

2.3.2.1 Structural coverage criteria

These coverage criteria are related to the structure of the model. It can originate from the code-oriented approaches such as control-flow or data-flow coverage as same concepts can be applicable in some model notations. Also, it can be derived only from special features of the model notation such as transition-based coverage.

2.3.2.2 Data coverage criteria

These coverage criteria are used to assign values for the test inputs. For a single input variable, it is possible to either choose only one value or assign all the values of the target domain. However, often the aim is to have more than one test and also not too many. Obviously, a single variable is not always the case and sometimes different combination of values is required for multiple input variables. The following data coverage criteria can be applied to reduce the number of tests and eliminate the test explosion problem for a single or collection of variables despite having satisfying tests [4]:

- Boundary-oriented coverage
- Statistical data criteria
- Pairwise coverage
The first coverage criteria is derived from boundary value testing which is a test design technique that the boundaries of input domain are used as test values. This criteria is helpful as most of errors can occur in extreme areas rather than center of the input domain. In statistical data criteria, a statistical distribution is given for an input variable to determine the aimed random-value coverage. Also, in pairwise coverage criterion, for any number of inputs, possible combination of values of all pairs of inputs are assigned for tests. This coverage is effective as it is believed that most of the faults are due to combinations of at most two inputs.

2.3.3 Stochastic and random criteria

Stochastic and random criteria can be discussed from two aspects. First one is related to having the randomness in input data that is required to be fed into the test cases in case of data-oriented systems. It is believed that choosing random approach for inputs is cost-effective as well as a good strategy for fault detection. Moreover, different types of statistical distributions can be considered for the data variables if required. Also, this aspect is discussed in coverage-based criteria, but due to the importance and relevance, it is preferable to mention it in this section either. The second one relates in a way that test case sequences are generated and deals with statistical test case generation where environment determines the SUT usage. The possible example could be a Markov chain where probabilities are attached to each transition to drive the generation in basis of how the SUT is more or less likely to be used.

2.4 From Abstract Tests to Executable Concrete Tests

As it is discussed in 2.1, concretization step is responsible to make the abstract test cases executable. This phase is required as because the API level of the model is not compatible in comparison to the API level of the SUT. Considering the purpose of MBT, in a test model often some unnecessary details is abstracted away and it represents a part of the system behavior that is desired to test cases be generated from. Thus, several problems arise from this abstraction that affects the concretization phase and bring more difficulties to handle it. The brief explanation of common problems are as follows:

- Model only represents some aspects of SUT, not the whole behavior
- Model omits some inputs and outputs of the system which are not relevant for testing purpose
- Model simplifies complex data structures of the SUT
• One model operation can represent a sequence of SUT operations

There are three approaches that can be applied in this step to overcome the problem: adaptation approach, transformation approach and mixed approach. All of these approaches are shown in the related step of Figure 1. In transformation approach, abstract test cases are transformed to executable test scripts by a script generator. However after generating test scripts minor changes are required to manually execute them and often completely executable tests are not the case. In adaptation approach, as the test cases are executed, each abstract function call is interpreted to a concrete function call in SUT using an adapter. Thus, execution and concretization happen at the same time. This approach is suitable for online testing which means concretization and execution phase take place while the generation of test cases occurs. Hence, this approach could be a good choice for overnight testing of non-deterministic systems [10]. Also, there is a mixed approach that the test scripts which are obtained from abstract test cases call adapters to handle low-level SUT operations. This approach is beneficial as it makes the adapter not to be dependent on the model and the adapter can be reused on different models or testing scenarios [4].
In this chapter, a structure to automate the adaptation phase of MBT is presented. The structure is implemented as an extension of the online MBT tool, ModelJUnit [1]. The tool is a Java library that extends the JUnit and the model representation is an EFSM which is also written in Java programming language. Any activity related to a transition of the EFSM is executed by a ModelJUnit action method which is annotated by @Action. Any transition can contain a trigger that leads to a state change in the SUT. The corresponding operation call which triggers the SUT can be located in two possible approaches in ModelJUnit action. In case of availability of the SUT, a system operation can be called directly. However, this is not desired as the model should be abstract and by increasing complexity of the SUT, the model will contain too much concrete details. The second approach is to use an adapter. The adapter contains the operations that are the abstract view of the concrete system operations. It is a wrapper around the SUT and implements all the steps of the test case that is required to be concretized before execution. Nevertheless, the concrete details are hidden in the model but the adaptation phase should be handled in the adapter code. The development of conventional adapter is labor-intensive and error-prone and failures of test cases can originate from the faults that are occurred in this stage. Moreover, in practice for systems that evolves quickly, modifying the adapter requires an effort that could be even more than testing the system manually [11]. Undoubtedly, a clear structure is required to eliminate the mentioned drawbacks and also ease the error tracking to reduce the required effort in this stage. The structure that will be discussed is deducted mostly from two major steps of adaptation, concretization of the inputs as well as abstraction of outputs. Thus, the focus is more on the data level abstraction in the domain of Java projects. Moreover, as the tool is implemented on Java, the powerful features of this programming language were beneficial in developing the different features of the extension. In the following sections general workflow and the main features of the tool extension will be discussed. Also, the features will be exemplified in a simple manner by adapting the abstract test cases extracted from a stack model to be executed on its implementations.
3.1 General Workflow

Figure 2 illustrates the general workflow of the ModelJUnit extension that automates the adaptation phase. The main unit of the tool extension is the auto-adapter which consists of different sub-units by itself.

![Figure 2: The Extension Workflow](image)

One of the inputs of the auto-adapter is an initialized object of the SUT. It acquires all the required information about the SUT class and whenever the model abstract operation is fed in, it adapts and then executes the operation on the SUT. The model abstract operation can contain the input and output parameters which all are defined by abstract data types. Because of the model abstraction, trivial parts of the concrete values are excluded in the possible data that can be used for the cited inputs. Also, same exclusion exists for the outputs as the unnecessary parts don’t have any key role on testing. To drive the execution of the test cases, these abstract inputs are required to fetch the actual values. Thus, the random test data generators are considered in the auto-adapter. In general, besides the cost-efficiency of the random approach for test data generation, it is a good strategy for fault detection of the system [9]. Also, in case of any necessity, external data generators can be added to generate desired values as the existing random generators can contain illegal values depending on the system requirement. Moreover, the generated input values are raw and abstract data types are used to define them. Hence, a data conversion process (concretization) is required to provide the real SUT input values defined by the concrete data types. Additionally, for concrete output values of the SUT, same conversion process (abstraction) can be required. The abstracted output values are used in the oracles to validate the system behavior in the test model. Figure 3 illustrates how the mentioned steps are done by the tool extension. The basic data type casting and straightforward conversion repository...
is embedded in the auto-adapter as default. Despite of this repository, as the fact of complex data templates and the existence of concrete data types in the system, the SUT-specific converters are required to be implemented to automate the adaptation phase for the test case execution. Noticeably, by using the proposed structure, most of the effort of the adaptation is spent to provide SUT-specific converters. Moreover, as the tracing feature of the extension, in each system update by analyzing and detecting any failure arisen from this phase and correcting the converters, the extra effort would be less. In addition, in the extension, storage (persistence) feature is considered to store the generated test data. The stored test data can be used as expected values of assertions as well as can be applied for the context variable manipulation of the EFSM.

![Diagram](image)

**Figure 3: Input Data Generation and Input/Output Adaptation**

### 3.2 Abstract Data Generation

In the extension of the ModelJUnit, any data generator is annotated by `@Generator`. It is represented by two parameters: a label and a data type. Every input in the model is assumed to be declared by an abstract data type. For any defined input, auto-adapter explores all the generators to produce a proper value by matching the input type with the generator data type. For most of the basic data types, random generators are considered and labeled by the default keyword. These generators are embedded in the extension and whenever a type is assigned for an input parameter, auto-adapter exploits them automatically. Moreover, it is possible to extend the generators for the same or non-existing basic data types to generate the desired values
and target a specific data coverage criterion. This group of generators is called external and they are implemented by the user. In case of an overlap with the existing type of generator, they can be given a specific label and be specified with the same label for an intended input. Otherwise, the auto-adapter invokes the default-labeled one to provide a value for the abstract input. Additionally, if necessary, it is also possible to extend the default data generators for more complex data types but the actual aim is to keep the data type as simple as possible. In Listing 3.1, the implementation of a build-in and an external data generator are shown. For existing extension generators, Jfairy\cite{11} is used as a base tool for random test data generation.

\subsection{Data Conversion}

The data conversion process is required at two stages:

- Concretization of the abstract input values
- Abstraction of the concrete output values

In the extension of ModelJUnit, data converters are annotated by @Converter. The parameters that can identify a converter are as follows: a label, a base data type and a target data type. The converters labeled by default keyword are exploited by auto-adapter automatically without any need to specify it for possible inputs or outputs of the abstract operation. This is done by figuring out the differences between the desired type in the model and the actual type in the SUT and finding the
Every abstract operation defined in the model represents an SUT operation in a simplified manner. It can contain input and output parameters. To make the abstract operation executable on the SUT, adaptation is required. As the extension is implemented on the online MBT tool, adaptation and execution phase of the MBT occurs at the same time. Furthermore, in the proposed structure, test data is produced for input parameters at the run-time. Thus, while defining an abstract operation, necessary data setting for each input should be configured to assist the testing process. The Listing 3.3 shows the methods of a class in the extension which is dedicated to create an abstract operation. However, some of the provided functionalities are optional and the usage is mostly dependent on the data level abstraction of the SUT as well as what could be helpful for verification.
Listing 3.3: Abstract Operation Class Methods

```java
public void newOp();

public void addTargetName(String name);

public void addParameter(int targetOrder, Class<?> type, boolean fresh, boolean store, String label);

public void setReturnType(Class<?> type);

public ArrayList<Object> createOp();

*In case that the "default" labeled data converters or generators are not aimed to be used:

public void specifyConversion(int targetOrder, String convLabel);
public void specifyRetConversion(String convLabel);
public void specifyGenerator(int targetOrder, String genLabel);

*Real output can be stored for conversion if needed:

public void storeConcreteOutput(String Label);
```

purposes. The most minimal case is that the possible names of SUT operation are only included and the auto-adapter generates random data in case of omitted input parameter to invoke the target method.

Any of the input parameters that are appended to an abstract operation is required to match with an input parameter of the equivalent SUT operation. We call the cited parameters, abstract and concrete inputs respectively. Multiple abstract inputs can also be assigned to a single concrete input. The abstract inputs are preferred to be declared by the basic data types as they have embedded data generators in the auto-adapter and they can receive a value automatically. However, the extension gives the possibility to add any data generator for any data type and to specify it in case of necessity. As in ModelJUnit, random transitions are taken to generate different combinations of the test cases; it is suitable to supply the abstract inputs of the test case operations by fresh test values. So, in case that the data generator starts to provide the repetitive values, it means that all of the data that generator can provide is covered by the executed test cases. This could be enabled by a true fresh bit while appending an abstract input to the abstract operation. Moreover, by enabling the store bit it is possible to retrieve the generated values from the data storage by referring to the specific label considered for the abstract input at the run-time. Besides the input adaptation setting, the output value of the SUT
operation can also be abstracted by considering an abstract data type. After determining the input/output setting along with the provided SUT-specific converters to the auto-adapter, all the conversions of the abstract input and concrete output values are done automatically. Finally, it is also possible to specify a non-default converter if multiple converters from an abstract type to a concrete type (or vice-versa) with different conversion process exist.

3.5 THE EXTENSION STORAGE

The following are the two categories of the run-time data that can be explicitly indicated to be stored by the auto-adapter:

- Generated abstract input values
- Concrete output values

Each set of the values is identified by a label. Any label illustrates a unique identifier of an array-list that can contain the sequence of an abstract input or a concrete output values. As the auto-adapter has the full control of the test data generation, adaptation and execution, the developer must assign a label to enable the storage for any specific input or output. This can be done in the extension abstract operation using the methods in the fifth and the last line of the Listing 3.3. This functionality is considered in the extension as the above mentioned values can have an important role in test verification and also concretization step. Also, by referring to the label, the access to any existing array-list is possible by using the predefined storage methods. This facilitates the modification whenever the array-list contains the input data state which requires to be updated.

3.6 DATA SETTING: FRESH

Whenever the fresh parameter of the fifth method illustrated in the Listing 3.3 is enabled for an input, the auto-adapter tries to generate the values that are not given before by re-executing the generator until some threshold times. However, in case that the threshold is passed, it means the generator is not capable to generate new values and the auto-adapter stores the warning message in the corresponding test case of the log file. This issue is discussed further in the following section and also can be observed in fourth line of Listing 3.10.

3.7 TEST CASE ADAPTATION TRACING

Whenever test cases are generated from the model and the on-the-fly testing starts, all the related activities about how the test cases are adapted and executed can be logged by the extension. In ModelJUnit,
it is possible to track the state changes of the model for each of the test cases. However, the logging feature gives the possibility to track all the SUT methods that are executed within a particular test case. It stores the information such as, for which abstract data types, the values are generated or what conversion methods are used for them to feed the required concrete values into the SUT method. This feature is a powerful adapter error tracker and it makes feasible to spend less effort to know what changes or additions are required to be done to handle the possible adapter failure. In a nutshell, it traces all the test cases in context of adaptation. Moreover, in case that the fresh bit is enabled for an input while its corresponding generator contains a finite amount of values, it informs that the data provided by the generator is covered by the executed test cases. In the condition that is described further, it can even inform that how many test cases are executed to reach the full coverage of the input. Such a condition is true when all the possible input values are provided by the user in its generator. It seems promising that by some changes in the proposed structure it is even possible to track the percentage of data coverage in addition to the coverage criteria related to the EFSM structure. The logging feature will be exemplified in following section by a simple case study in details.

3.8 SIMPLE CASE STUDY: STACK

An EFSM test model of the stack is implemented using ModelJUnit. It is adapted and executed by the extension on two stack implementations which differ in data types that are used. Although the stack is a basic example but its simplicity will make it easy to describe the extension features discussed above.

3.8.1 Stack EFSM Model

Figure 4 illustrates an example EFSM to test the basic behavior of the stack. Transitions are labeled as follows:

Mutator (Action), [Guards] / Context Variable Manipulation
In model implementation, states of the EFSM are defined by an enumerator consisting of two members: Empty and NotEmpty. The initial state is Empty and the model is restricted by adding a maximum limit of two elements. The EFSM also contains a context variable to track the number of present elements and also to be used for the decision logic of the guards. As a reminder, in ModelJUnit model class, each transition activity is executed by a method annotated by @Action which is allowed to be executed if the related guard method call returns one. ModelJUnit action method can consist of several parts such as state change, context variable manipulation and the abstract operation call which evolves the SUT. Mostly, in context of MBT, the cited abstract operation is actually expressed as an action which can be confused with the ModelJUnit action. A related source[5] also mentions it as a mutator operation which is preferred to be used in rest of the content.

Listing 3.4: Stack EFSM ModelJUnit Actions

```java
@Action
public void Push() throws Throwable {
    adapter.adapt(Push);
    Items++;
    state = States.NotEmpty;
}
public boolean PushGuard() {
    return state == States.Empty || (state == States.NotEmpty & Items<max);
}

@Action
public void Pop() throws Throwable {
    adapter.adapt(Pop);
    Items--;
    if(Items==0)
        state = States.Empty;
    else
        state = States.NotEmpty;
}
public boolean PopGuard() {
    return state == States.NotEmpty;
}
```

As it can be seen in the stack EFSM, multiple transitions with the same mutator named push and pop exist. It leads to an additional user-defined code and also repetitive ModelJUnit action methods with the same mutator operation but partial distinction in the related guard method and the next state update part. However in [13], this issue
has been discussed but the problem is mainly solved by defining an appropriate EFSM syntax in an MBT tool named Modbat[2] which is highly inspired by ModelJUnit [14]. Inspired from the idea, the stack EFSM is implemented with a focus on reduction by having only two ModelJUnit actions for each mutator as a result of additional logic in the guard method and the state change part of the action method. The related code is shown in Listing 3.4. Clearly, the reduction leads to an unstructured implementation but also compact as a plus. Meanwhile, difficulties can be encountered while placing the state-related oracles, but as the focus is to adapt the model abstract operations and there is no reason that this approach can cause a problem in our context, it is adequate to only state this issue to this extend. Finally, for each test case generated by the tool, all of the required initialization demonstrated on Figure 4 takes place in the reset method of the model class besides the SUT initialization.

3.8.2 How the Auto-Adapter Works In Practice

3.8.2.1 Scenario 1

First, for any operation which is planned to be used in the test model, abstract operations must be defined. For the meantime, the model code is the same as described in the section above without any oracle implementation to detect the SUT failures.

Listing 3.5: Model Operations of Scenario One

```java
public class StackOperations {
    private AbstractOperation absOp = new AbstractOperation();

    public ArrayList<Object> Push() {
        absOp.newOp();
        absOp.addTargetName("push");
        return absOp.createOp();
    }

    public ArrayList<Object> Pop() {
        absOp.newOp();
        absOp.addTargetName("pop");
        return absOp.createOp();
    }
}
```

As a starting point, we assume a scenario that none of the inputs of the operations are mentioned and we define them only by the knowledge of possible target names to see how the auto-adapter handles it. The defined push and pop mutator operations are shown in Listing 3.5. In model code, the adapter class is initialized by an instantiated SUT object. Each of the created operations requires to be an input for the
adapt method of the adapter class to be executed in the right location. This can be seen in the line 3 and 15 of the Listing 3.4 where the operation information is given to the auto-adapter to start the search and invocation process. In this scenario, the SUT is an initialized object of the stack which is implemented by the Double data type.

Scenario 1 Results:
The random test case generator is considered for the execution. It starts to do the random walk in the EFSM to generate and execute the test cases in random finite paths until the determined number of steps are traversed. As the selected number of steps is 5, only one test case is generated. Despite that, the full action, state and transition coverage are satisfied. Listing 3.6 shows the output results provided by the tool.

Listing 3.6: ModelJUnit Execution Results

done (Empty, Push, NotEmpty)
done (NotEmpty, Pop, Empty)
done (Empty, Push, NotEmpty)
done (NotEmpty, Push, NotEmpty)
done (NotEmpty, Pop, NotEmpty)
action coverage: 2/2
state coverage: 2/2
transition coverage: 4/4

As it is seen in the results, the execution is done successfully without any exception occurrence during the run-time. In case of any occurrences, the failures can originate from the auto-adapter and the main causes need to be found and fixed. As the extension has a feature to track all the interactions between the test model and the SUT, it logs all the related information. It stores any SUT related activity associated with all steps of the test cases and includes the details regarding how the adaptation is taken place for each one of them. The log file of scenario one is illustrated in Listing 3.7. As an example, in first step of the executed test case, abstract push operation is called from the test model. At the beginning, auto-adapter finds the equivalent method by a given information determined in the model operation definition stage. Later, auto-adapter detects an omitted parameter in the relevant SUT method and generates an appropriate data with taking the target data type into account. In case that the data demand are done for all of the existing parameters, the method invocation starts. Also it is good to mention, in case of any failure related to a particular adaptation step, the error messages appear to guide the tester to fix the problem. The tester can easily determine the fault by analyzing the errors and the location it is occurred. The possible scenario for this situation could be the lack of a default generator for the specific data type of omitted parameter.
Listing 3.7: The Extension Tracing Log File

Tescase number: 1

method is found as push in SUT
trying to generate input data. /label: omitted SUT
parameter/target order: 0/target type: java.lang.
Double/generator: default
default generator for class java.lang.Double is found
generated data: -8675.045994058255 / label: omitted
SUT parameter

SUT method push is executed

method is found as pop in SUT
SUT method pop is executed

method is found as push in SUT
trying to generate input data. /label: omitted SUT
parameter/target order: 0/target type: java.lang.
Double/generator: default
default generator for class java.lang.Double is found
generated data: -9817.865600471156 / label: omitted
SUT parameter

SUT method push is executed

method is found as push in SUT
trying to generate input data. /label: omitted SUT
parameter/target order: 0/target type: java.lang.
Double/generator: default
default generator for class java.lang.Double is found
generated data: 5925.3100908486285 / label: omitted
SUT parameter

SUT method push is executed

method is found as pop in SUT
SUT method pop is executed

3.8.2.2 Scenario 2

In this scenario oracle implementations are included in the model
code in contrast to the previous one. The assertions that compare the
actual and expected values can be categorized into two groups: state-
related and action-related. The SUT can contain a group of methods
which only inform the current state of the system and does not lead to
any changes in case of execution. In the test model, we will call them
query operations in contrast to the mutator ones. Stack implementation
contains a query method that checks whether the current state is empty
or not. This query operation is added to the model operations beside
of push and pop mutators. It is included to verify the equality of
the current states in the model and the SUT. Also, for any sequences
of push and pop actions, the popped values should be same as the
values that are pushed before. To validate the cited behavior, abstract push and pop operations require to contain additional information in comparison to the scenario one operations. Listing 3.8 shows the updated push and pop model operations for this scenario.

**Listing 3.8:** Push and Pop Model Operations in Scenario Two

```java
public ArrayList<Object> Push(){
    absOp.newOp();
    absOp.addTargetName("push");
    absOp.addParameter(0, int.class, true, true, "PushData");
    absOp.specifyGenerator(0, "limited");
    return absOp.createOp();
}

public ArrayList<Object> Pop(){
    absOp.newOp();
    absOp.addTargetName("pop");
    absOp.setReturnType(int.class);
    return absOp.createOp();
}
```

For verification purposes, abstract input in the type of Integer is appended to the push operation. Also, store bit is enabled to retrieve the generated input data by its specified label, "PushData". This can be done by using storage methods of the auto-adapter. Additionally, an external data generator which is shown before in Listing 3.1 is specified for the input. This generator contains the finite amount of data and as the fresh bit is true, full data coverage can be notified if adequate number of test cases are executed. Moreover, for the pop operation, return data type used in the model is included. In this scenario, same as the previous one, stack implementation is the one that uses the data type of Double. However, Integer data type is considered in the test model and the test assertion is implemented on basis of that. The Listing 3.9 illustrates all the assertions of the model code.

**Listing 3.9:** Assertions in Scenario Two

```java
1. Assuring the correct data state of the SUT:
   int actual = (Integer) adapter.adapt(Pop);
   int expected = (Integer) storage.popStorage("PushData");
   Assert.assertEquals(expected, actual);

2. Assuring the state change of the SUT:
   boolean expected=false; or boolean expected=true;
   boolean result = (Boolean) adapter.adapt(isEmpty);
   Assert.assertEquals(expected, result);
```
In the model code, the first assertion of Listing 3.9 is located in the pop ModelJUnit action with the required addition of actual output to the mutator operation. Also, for the second assertion which ensures the state change, by choosing the right expected value, it is placed right before the Empty or NonEmpty state change line when the mutation of SUT is done.

Scenario 2 Results:
The random generator is set to generate test cases until satisfying 25 steps of walk in the graph. The execution is done successfully and three test cases are adapted and executed on the SUT without any assertion failures. Listing 3.10 shows the adaptation steps of the model push and pop operations in the third executed test case. The generated input value for push operation is casted to the SUT data type using embedded converters of the auto-adapter. Also, the output value of the SUT is casted back to its original format in the test model. This gives an insight about how the concretization of the abstract input values as well as abstraction of output values can be automated using SUT-specific converters. Finally, in the fourth line of the Listing 3.10, shrink message regarding the provided input data is notified.

Listing 3.10: Adaptation Steps of the Model Push and Pop Operations

```java
method is found as push in SUT
    trying to generate abstract data. /label: PushData/
        target order: 0/type: java.lang.Integer/generator: limited
    limited generator for class java.lang.Integer is found
    all data are tested for: PushData. 20 repetition to generate fresh data
    generated data: 8 / label: PushData
6 trying to concretize/convert generated data. /target type: java.lang.Double/converter: default
    default converter from java.lang.Integer to java.lang.Double is found
    SUT method push is executed

11 method is found as pop in SUT
SUT method pop is executed
    trying to make output data abstract. /from java.lang.Double to java.lang.Integer/converter: default
    default converter from java.lang.Double to java.lang.Integer is found
    output data from SUT is: 8
```
3.8.2.3 **Scenario 3**

As it is shown in Figure 5, an illegal state is added to the previous EFSM to check how the auto-adapter handles a faulty state.

![Stack EFSM with Illegal State](image)

**Figure 5: Stack EFSM with Illegal State**

Also, in the model code, an assertion part is included for the cited state to ensure the exception occurrence. The related code snippet can be seen in **Listing 3.11**.

**Listing 3.11: Illegal State Exception Handling**

```java
boolean exceptionOccurred = false;
try {
    // Call Adapter
    adapter.adapt(Pop);
} catch (IllegalStateException e) {
    exceptionOccurred = true;
} //... Assertion
Assert.assertTrue(exceptionOccurred);
```

In this scenario, the SUT is an object of the stack class which is implemented using the data type, String. The oracle implementations that are considered in the previous scenario are also the same.

**Scenario 3 Results:**
The execution of generated tests is done successfully and no assertion failures are detected. The **Listing 3.12** shows an step in the log file related to a test case in which transition to the illegal state is taken.

**Listing 3.12: Tracked Method Exception in Log File**

```java
method is found as pop in SUT
SUT method pop threw exception: java.lang.
    IllegalStateException
```
3.9 Conclusion

In this chapter, the features of the extension of ModelJUnit that automate the adaptation layer are discussed. The formation of an abstract operation of the model is illustrated. Also, it is shown that how the abstract input/output data setting, appended to the operation, assists the verification process and how the extension makes the execution of tests feasible by automating the provision of the abstract values using built-in data generators. Moreover, the role of the SUT-specific converters are discussed which must be implemented in the proposed ad-hoc layout to be exploited by the main unit of the extension (auto-adapter) to complete the execution of tests. Finally, three scenarios on a simple case study are discussed to illustrate the capabilities of the extension. It is also mentioned that, how the tracing capability of the extension can be helpful to track the adaptation phase and decrease the maintenance response time in case of failures. For future work, as the similarity of applied assertion-based oracles (output or exception) and the possible expected behaviors, it can be investigated to apply these oracles as a setting to the extension abstract operation to reduce the model size and the developer effort. Also, as the basic structure of ModelJUnit, the most appropriate EFSM syntax in the context of MBT can be investigated.
In this chapter, the usage of the ModelJUnit extension (auto-adapter) will be described and put into practice. Also, it will be compared with the conventional adapter to illustrate how the extension contributes to the flexibility and the modularity of the adaptation step as well as model size and portability. The evaluation will be done on a case study which focuses on four main functionalities: Create, Retrieve, Update and Delete (CRUD).

4.1 CONVENTIONAL ADAPTER

In conventional way of the adaptation, an adapter interface consists of a set of methods. These methods need to be implemented to make the test execution feasible. In the test model above-mentioned methods are named abstract operations. The motivation behind the test adapter is to separate the model and the SUT and isolate the model from concrete details to have more portability of the model. However, an adapter interface illustrates a simplified view of the SUT. In the context of model-based testing, an adapter can have any of the following two requirement that always have to be considered in the development phase:

- Handles incompatible input and output data format considering the SUT implementation
- Implements a group of SUT operations

4.2 AUTO-ADAPTER

The main intention of the research is to deal with the first requirement of the conventional adapter. The extension (auto-adapter) consist of three main components: data converters, data generators and data storage. In case of existing class type converters from model input type to SUT input type and also from SUT output type to model output type, the auto-adapter exploits them to automate the conversion process. However, this doesn’t mean the only type conversions occur in converters. Any converter can be responsible for the following tasks or a combination of them:

1. Conversion of class types (Can be both primitive or reference type)
2. Transformation of data (Not depending on types)

3. Mapping the data

The class type conversion is straightforward but the other two tasks need to be clarified. As an example, a value that refers to the price is considered in the model as the type of Integer and it requires to be transformed to the String type in the following template: \{ "id": 100, "price": 30 \}. Thus, in addition to the type conversion, the data needs to be also transformed. So, the first two tasks are combined. However, the mapping is mostly associated with storage feature of the extension and will be exemplified later in the chapter. Finally, the second requirement of the conventional adapter is not in the scope of the data abstraction and we suggest the two layer of adapter in which the last layer is automated.

4.3 THE AUTO-ADAPTER INTERFACE: ABSTRACT OPERATION

Abstract operation is the operation that is used in model which can contain the input and output parameters. The parameters can be defined by any primitive or reference type, simplified in the data in comparison to the target SUT operation parameters. Essentially, in the conventional way, the abstract operations are equivalent to the adapter interface methods that are required to be implemented as a bridge between the SUT and the model. However, the characteristic of the SUT can lead to possible modification in the primary adapter interface. Any change in the interface can also affect the model. In contrast to the conventional adapter, the extension defines the abstract operation using a set of predefined methods (3.4). Any input or output is assigned with the type that is considered in the model. Also, input parameters are included with the data setting regarding what data should be fed into each abstract parameter. In the extension, defining of the abstract operation starts from modeling stage and it is refined and completed in adaptation stage for execution.

4.3.1 Abstract Operation in Modeling Stage

Soon after the modeling stage, each abstract operation contains the predefined extension methods regarding how the test data for each input parameter must be handled in the run-time. The work that can be done through this stage after assigning a type for each input parameter is as follows:

- Each input parameter can be assigned to a model-related (external) data generator in case that there is no need to use an existing data generator recognized by the same input type automatically.
• The generated data can be configured to be a fresh (non-repetitive) by enabling a single bit.

• The generated data for a parameter can be stored by enabling a single bit for oracles or context variable manipulation.

• Also, it can be stored for the decision logic of the other parameter’s data generator which is dependent to the value of the same parameter.

Moreover, it can contain an assignment of the output type in which the output oracle can be implemented in model abstraction level.

4.3.1.1 Benefits of Combining the Data Requirement with Abstract Operation

Existing ModelJUnit environment is not user-friendly to provide the large amount of test data for the inputs. But in the provided extension, this can be easily handled by assigning data generators. Thus, the extension makes it feasible to set the data requirement for each of the inputs in a easy manner while in the modeling stage. In addition, the compact nature of generators makes future modification more convenient.

4.3.2 Abstract Operation in Adaptation Stage

In addition to the last stage, the concrete configuration of the SUT is added to the abstract operation. The concrete configuration consists of:

At early stage before starting to design class type converters:

• The target SUT operation name and input parameter order

At stage of designing converters:

• In case of conflicting converters from a parameter type considered in abstract operation to SUT parameter type or vice versa with different process: specifying converter by label.

• In case that the concrete output is used by another converter or is an object that should not be lost and be used for another SUT input (Mapping task of conversion): storing the concrete output

• In case that the generated input value of a operation needs to be used for another operation’s input conversion (Mapping task of conversion): enable the store bit of the input to be used in converter
4.4 CASE STUDY: CRUD

CRUD is the abbreviation of four basic functions named Create, Retrieve, Update and Delete respectively. CRUD is mainly implemented in data-base applications. However, as the nature of it, the model that is designed, can be applicable into different groups of applications. In this section, an implemented model of the CRUD in ModelJUnit will be evaluated by adapting it into two SUTs using both conventional adapter and the implemented extension. The SUTs are a web service client: flight ticket booking and a simple java project: course grade manager.

4.4.1 CRUD Model

The Figure 6 illustrates the abstract EFSM model of the CRUD. It consist of three actions: Create Entry, Update Any Entry and Delete Any Entry. The variable EntryCount is responsible to track the entries and avoid the illegal transitions. Also, all transitions consist of the assertion that retrieves all the existing entries one by one to confirm the expected behavior. As the model contains only one state due to the abstraction but it also consist of the hidden data state of existing entries. Using the extension, the cited state can be handled by the build-in storage feature. However, in conventional approach an extra effort is required to store the data state. Moreover, the Delete Any Entry action have the assertion to assure that the performed operation on randomly selected entry is done as expected. This oracle is only applicable on web server client SUT and ignored in the grade manager SUT as retrieving all of the entries assures it.

![Figure 6: The CRUD Model](image)

Each entry data in the model is of the form: a name and a corresponding value which are equivalent to passenger name and the number of seats in flight booking SUT. The same entry can be also
equivalent to student name and the score of the initialized course in grade manager SUT. The oracles are implemented using both of the name and its value, in advance. The table in Figure 7 illustrates all the model operations including inputs and outputs.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Store in Storage</th>
<th>Target SUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Entry</td>
<td>Fresh Random String (Name)/ Random Int (Value) [0, 5]</td>
<td>-</td>
<td>Name, Value</td>
</tr>
<tr>
<td>Update Any Entry</td>
<td>Fresh Random String (Name)/ Random Int (Value) [0, 5]/ Random Int (Entry Number) [0, EntryCount]</td>
<td>-</td>
<td>Name, Value, Entry Number</td>
</tr>
<tr>
<td>Delete Any Entry</td>
<td>Random Int (Entry Number) [0, EntryCount]</td>
<td>-</td>
<td>Entry Number</td>
</tr>
<tr>
<td>For All: Retrieve Entry</td>
<td>Int (Entry Number) [0..EntryCount]</td>
<td>String (Name), Int (Value)</td>
<td>-</td>
</tr>
<tr>
<td>Retrieve Deleted Entry Error</td>
<td>Int (Deleted Entry’s Number)</td>
<td>Null</td>
<td>-</td>
</tr>
</tbody>
</table>

Note: Init Operation is excluded in the table.

Figure 7: CRUD Model Operations Description

4.4.2 Abstract vs. SUT Operations

Each abstract operation that is considered in the model has its equivalent operation in the SUT. The tables in Figure 8 illustrates the CRUD operations and the equivalent SUT operations differed by input and output data and types. In conventional adapter, the interface methods

<table>
<thead>
<tr>
<th>Primary Adapter Interface Methods / Abs Ops</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>Name (String), Value (Int)</td>
<td>-</td>
</tr>
<tr>
<td>Update</td>
<td>Name (String), Value (Int), Entry# (Int)</td>
<td>-</td>
</tr>
<tr>
<td>Delete</td>
<td>Entry# (Int)</td>
<td>-</td>
</tr>
<tr>
<td>Retrieve</td>
<td>Entry# (Int)</td>
<td>(Name, Value) (Object)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flight Booking Methods</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BookFlight</td>
<td>(Name, #ofSeat) (BookingFlight Class Type)</td>
<td>PNR (Long)</td>
</tr>
<tr>
<td>UpdateBookedFlight</td>
<td>(Name, #ofSeat, PNR) (BookingFlight Class Type)</td>
<td>-</td>
</tr>
<tr>
<td>DeleteBookedFlight</td>
<td>PNR (Long), Name (String)</td>
<td>-</td>
</tr>
<tr>
<td>GetBookedFlight</td>
<td>PNR (Long)</td>
<td>(Name, #ofSeat) (BookingFlight Class Type)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grade Manager Methods</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>(Name, Score) (StudentScore Class Type)</td>
<td>-</td>
</tr>
<tr>
<td>replace</td>
<td>(Name, Score) (StudentScore Class Type), Index (Int)</td>
<td>-</td>
</tr>
<tr>
<td>remove</td>
<td>Index (Int)</td>
<td>-</td>
</tr>
<tr>
<td>get</td>
<td>Index (Int)</td>
<td>[Name, Score] (StudentScore Class Type)</td>
</tr>
</tbody>
</table>

Figure 8: Abstract and Equivalent SUT Operations
are the same as the abstract operations required to implement the SUT. However, as it can be seen in Figure 8, in comparison to the adapter interface, partial differences, particularly in the flight booking methods exist.

In flight booking SUT, for each entry, a record number (PNR) is used to track the entry. The PNR number is assigned by the server after the flight has been booked. It is the output of the BookFlight method. However, in the model, PNR value is the sequence number when the entry is created due to abstraction. In conventional adapter, this can lead to a change in adapter interface as the sequence number doesn’t meet the demand of the SUT. Including more concrete details in the adapter interface inputs and outputs is equal to having more complex model. However, in the extension, each defined abstract operation is combined with the extra SUT configurations to make the adaptation more convenient. As the PNR is the concrete output of the BookFlight method, it can be stored and used in conversion process of the extension SUT-specific converters. As it is discussed before, any concrete output from the SUT can have a clue for another input concretization. This illustrates the mapping task of a converter that the auto-adapter exploits in the run-time. The Listing 4.1 shows the create operation defined and completed for flight booking SUT using the extension.

Listing 4.1: Create Abstract Op with Flight Booking Concrete Configuration

```java
public ArrayList<Object> Create() {
    absOp.newOp();
    absOp.addTargetName("BookFlight");
    absOp.addParameter(0, int.class, false, false, "Not Used"); /* Generated Data Not Used - Only for converter invocation */
    absOp.addParameter(0, String.class, true, true, "Name");
    absOp.addParameter(0, int.class, false, true, "Value");
    absOp.specifyGenerator(0, "Value");
    absOp.storeConcreteOutput("PNR");
    return absOp.createOp();
}
```

4.4.3 Auto-Adapter Converters vs. Conventional Adapter Implementation

The principle of the MBT received much attention in software testing, as it is more convenient to design or use the existing model rather than diving into the complex behavior of the system using code-centric approaches. However, the complexity originated from the code is not totally eliminated and it notably appears in the stage that the model is required to be adapted to the SUT for the test execution. One of the difficulties in this stage is to bridge the data level gap
between model and the SUT. The auto-adapter handles the data gap using data converters recognized by initiator type to target type. It induces the mindset to realize the differences of the abstract and concrete data. One of the most important advantages of the converters accompanied by the storage feature of the extension is the ability to use the concrete SUT output of an operation for conversion of any other model operation’s input, without any change in the abstraction level of the model. However, using conventional adapter, this is only feasible by adding more concrete details to the interface methods. Moreover, using the converters can reduce the repetition in early development phase as the same conversion can occur multiple times.

Also, it gives the possibility to make the portable set of converters that can be used in different models. The table in Figure 9 illustrates input and output conversion from model to flight booking SUT and vice versa respectively. As it can be seen, the group of parameters in the model can target the single parameter in the SUT. The first observation is that the six conversions are handled by four converters. The extension converters provides more modular implementation in

<table>
<thead>
<tr>
<th>CRUD Model</th>
<th>Flight Booking SUT</th>
<th>Conversion</th>
<th>Conv Type</th>
<th>Method ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (String), Value (Int)</td>
<td>(Name, #ofSeat) (BookingFlight Class Type)</td>
<td>✓</td>
<td>Many2One (C)</td>
<td>1</td>
</tr>
<tr>
<td>Entry# (Int), Name (String), Value (Int)</td>
<td>(Name, #ofSeat, PNR) (BookingFlight Class Type)</td>
<td>✓</td>
<td>Many2One (C)</td>
<td>1</td>
</tr>
<tr>
<td>Entry# (Int) PNR (Long), Name (String)</td>
<td>✓ ✓</td>
<td>Single (C)</td>
<td>2,3</td>
<td></td>
</tr>
<tr>
<td>Entry# (Int) PNR (Long)</td>
<td>✓</td>
<td>Single (C)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(Name, Value) (Object[])</td>
<td>(Name, #ofSeat) (BookingFlight Class Type)</td>
<td>✓</td>
<td>(A)</td>
<td>4</td>
</tr>
</tbody>
</table>

(C)=Concretization, (A)=Abstraction

Figure 9: Flight Booking SUT Conversions for CRUD Model

bridging the data gap between the model and the SUT. In addition, it defines the flexible tasks that a conversion process can contain (4.2). The conventional adapters lacks these definitions and the developer only focuses on solid implementation. In Table 1 it can be observed that the amount of the adapter code has a noticeable reduction using the extension.

Table 1: Conventional Adapter and The Extension Converters Code Size

<table>
<thead>
<tr>
<th>SUT</th>
<th>Conventional Adapter Implementation (Lines)</th>
<th>The Extension Converters (Lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Booking</td>
<td>38</td>
<td>17</td>
</tr>
<tr>
<td>Grade Manager</td>
<td>29</td>
<td>18</td>
</tr>
</tbody>
</table>
4.4.4 Auto-Adapter Interface vs. Conventional Adapter Interface

As it is discussed earlier in this section, depending on the target SUT, possible modifications can occur in the adapter interface in the stage of adaptation. In this part, the number of changes in the interface is analyzed using both conventional and automated approach. Also, the influence of the change on the model code is measured for the two samples of SUTs. First table of the Figure 8 is taken as the reference for primary interface of the both approaches. However, it is also good to mention that the operation related to SUT initialization is excluded in the table and the final measurement is also affected by that. The Listing 4.2, illustrates the methods of the conventional adapter interface which is modified for flight booking SUT.

Listing 4.2: Modified Conventional Adapter Interface For Flight Booking SUT

```java
public void Init();
public Long Create(String Name, int NoOfSeats);
public void Update(String Name, int NoOfSeats, Long PNR);
public void Delete(Long PNR, String Name);
public Object[] Retrieve(Long PNR);
```

Table 2: Results of Conventional Adapter

<table>
<thead>
<tr>
<th>SUT</th>
<th>Parameter Changes in Conventional Adapter Interface</th>
<th>Additions in Model Code (Lines)</th>
<th>Total Number of Changes in Model Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Booking</td>
<td>5</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>Grade Manager</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3: Results of Auto-Adapter

<table>
<thead>
<tr>
<th>SUT</th>
<th>Additions in Auto-Adapter Interface's SUT Configuration (Lines)</th>
<th>Additions in Model Code (Lines)</th>
<th>Total Number of Changes in Model Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flight Booking</td>
<td>1 + 2 (CI) = 3</td>
<td>1 (NC)</td>
<td>1 Addition (NC)</td>
</tr>
<tr>
<td>Grade Manager</td>
<td>1</td>
<td>0</td>
<td>1 Reduction</td>
</tr>
<tr>
<td>Both</td>
<td>9 + 2 (CI) = 11</td>
<td>1 (NC)</td>
<td>1 Addition (NC) + 1 Reduction = 2</td>
</tr>
</tbody>
</table>

NC = Non-Conflicting, CI = For Converter Invocation
The nature of the flight booking SUT influences the conventional interface with five changes in the parameters compared to the primary interface. In addition, it leads to thirteen modification in the model code. Obviously, after this amount of changes the model is no more portable between two of the SUTs. The cited results of the conventional adapter can be seen in Table 2. However, using the extension, only three additions are required to its interface. Among three additions, only one of them is actually factual. The other two additions originate from the base idea of the extension which is the reflection ability of the Java language. Moreover, the other important observation is that the only one addition is required to the model code and it relates to updating the storage status. However, this addition is non-conflicting with the other SUT and the same model can be applicable to the grade manager source code. This illustrates that the model is still portable between both SUTs. The mentioned results of the auto-adapter can be seen in Table 3. Finally, as it can be seen in Table 3, the extension interface is capable to handle both of the SUTs at the same time. This proves the flexibility of the auto-adapter interface whereas the mentioned case can rarely happen using the conventional approach.

4.4.5 Model Code: Size Comparison

In this part, the effects of the extension to the model size is evaluated. A comparison is made by the lines of code that are required to implement the ModelJUnit actions using the extension, conventional and no adapter. The results can be seen in Table 4 and Table 5 for flight booking and grade manager SUTs, respectively. Before the discussion, it is better mention that the assertion regarding the retrieving of deleted entry is not applicable to the grade manager. However, assuring the successful delete entry operation is handled by retrieving all the entries as the sequence order matters for this SUT. The unused oracle in Table 5 illustrates the cited point.

<table>
<thead>
<tr>
<th>ModelJUnit Action</th>
<th>No Adapter (Lines)</th>
<th>Conventional Adapter (Lines)</th>
<th>The Extension (Auto-Adapter) (Lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Entry</td>
<td>38</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>Update Any Entry</td>
<td>37</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>Delete Any Entry</td>
<td>36</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

According to the results, first observation is that the total lines of code in Create and Update actions are reduced to even less than half in comparison to the case that the conventional adapter is used. However, this reduction is not present in Delete action and no noticeable differ-
Table 5: Model Size for Grade Manager SUT

<table>
<thead>
<tr>
<th>ModelJUnit Action</th>
<th>No Adapter (Lines)</th>
<th>Conventional Adapter (Lines)</th>
<th>The Extension (Auto-Adapter) (Lines)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Entry</td>
<td>33</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>Update Any Entry</td>
<td>33</td>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>Delete Any Entry</td>
<td>25-2 (UO)=23</td>
<td>13-2(UO)=11</td>
<td>14-2(UO)=12</td>
</tr>
</tbody>
</table>

UO = Unused Oracle

ence is observed. Basically, the cited reduction refers to the actions that the number of inputs are in raise or the state of the data is required to be kept. Also, as it is expected, in the case that non of the adapters are used, the maximum number of code lines are implemented in the actions. In this case, some of the lines are related to accessing the private methods which can be also visible in Table 1.

4.5 CONCLUSION

In this chapter, the ModelJUnit extension (Auto-Adapter) is evaluated in comparison to the conventional adapter using a case study. The reductions in the adapter implementation and the model code is observed and it is proved that the extension leads to more compact model and adapter code. Moreover, it is shown that the auto-adapter interface leads to the least possible numbers of modifications in the model code and it doesn’t affect the portability of the model. In addition, it is illustrated that the same interface can target two different SUTs whereas it can rarely happen using the conventional interface. As a future work, the re-usability of the extension converters on different models can be investigated.

4.6 COMPARISON WITH RANDOOP FOR FUTURE WORK

Randoop [15] is a tool that uses feedback-directed random test generation to automatically create unit tests. Similar to the auto-adapter implemented in this thesis, it uses the reflection to collect the information of any java classes. It exploits the group of classes and creates the meaningful random sequence of method calls. In auto-adapter, one of the tasks of the converter is explained as a type conversion. It looks feasible to eliminate the need to this group of converters by implementing the feedback-directed approach. This might be done by providing the classes of the reference types to the auto-adapter beside the SUT class. Thus, the reference types are assumed to be the existing input types of the SUT operations. Also, the random generators can
be more effectively exploited by the auto-adapter using the filtering mentioned in [16].

4.7 LINK FOR THE EXTENSION

The link to a github repository which contains the source code of the work that has been done in this thesis:

https://github.com/ardalanha/AutoAdapter-ModelJUnit-Extension/


