Systematic mapping study on combining model-based testing and behavior-driven development or test-driven development

Rida Kamal

Master of Science (Technology)Thesis

Supervisors: Adj. Prof. Dragos Truscan

Co-supervisor: MSc. Junaid Iqbal

Department of Information Technology

Faculty of Science and Engineering

ACKNOWLEDGEMENT

First of all, my heartfelt praises and special thanks to God for his endless blessings upon me throughout my research work.

I would want to extend my sincere gratitude to my research Supervisor, Dragos Truscan, Senior Lecturer, and Adjunct Professor at the Faculty of Science and Engineering, Åbo Akademi University. His unconditional support enabled me to write this thesis and carry forward with my motivation. I would also like to express my genuine thanks to my co-supervisor, Junaid Iqbal, Doctoral student at the Faculty of Science and Engineering, Åbo Akademi University. Without his help, it would be exceedingly difficult and unmanageable for me to complete my research. It was an honor for me to work under the supervision of both. I would like to thank my parents and family for their moral support and love. I would also want to express my special thanks to my daughter, Wafa Zainab, who sacrificed her mother-daughter time and allowed me to work.

Finally, I would like to convey my heartfelt thanks to my husband, Tauseef Hassan, who stood by my side through every thick and thin and bore with me at my worst. Without his support, it will not be possible for me to achieve any goal of my life.

ABSTRACT

Recently, in the field of software development, Agile has been one of the most adopted software development methodologies. The key strength of the Agile approach is validation. Moreover, in the field of software testing, Model-Based Testing (MBT) methods have emerged to ensure verification. However, both methods seem to evolve independently and in a different direction. Combining the potencies of both methodologies (i.e. validation and verification) can be significantly effective in the software testing field. In this thesis, we will perform a Systematic Mapping Study (SMS) to investigate the available methodologies that combine either Model-Based Testing and Test-Driven Development (TDD) or Model-Based Testing and Behavior-Driven Development (BDD). TDD and BDD are two of the most popular Agile approaches. An SMS provides a methodology and protocol of the type of research reports and results that have been published by categorizing them and often gives a visual summary, the map, of its results. This SMS will explicitly focus on the integration of TDD or BDD (TDD/BDD) with MBT. The main objective of this SMS is not to weigh the validity of the given method or to compare the quality of different approaches. Rather, the goal is to characterize the available methodologies that integrate MBT and TDD/BDD approaches and extract the information provided in that study.

Keywords: Model Based testing, Test Driven Development, Behavior Driven development, Systematic Mapping Study

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AD	Agile Development
Adj.	Adjunct
ATDD	Acceptance Test-Driven Development
AUT	Automation
AWS	Amazon Web Service
BBT	Black-Box Testing
BDD	Behavior-Driven Development
D-MBTDD	Delta Model-Based Test-Driven Development
DFD	Data Flow Diagram
DSDM	Dynamic Systems Development Method
DSE	Design Space Explosion
EMF	Eclipse Modeling Framework
EMR	Elastic MapReduce
ERP	Enterprise Resource Planning
FDD	Feature-Driven Development
FSM	Finite State Machine
FUML	Foundational Unified Modeling Language
GUI	Graphical User Interface
HLM	High-level Modeling
HTML	Hypertext Markup Language
IOCO	Input-Output Conformance
LLM	Low-Level Modeling
MbRe	Model-Based Requirement Engineering
MBT	Model-Based Testing
MBTDD	Model-Based Test-Driven Development
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDPE	Model-Driven Performance Engineering
MDWE	Model-Driven Web Engineering
MSC	Message Sequence Charts
MSc.	Master of Science
Prof	Professor

QC	Quality Criteria
ROC	Return On Capital
RQ	Research Question
RST	ReStructured Text
RUP	Rational Unified Process
SAL	Smart Action Language
SDLC	Software Development Life Cycle
SMS	Systematic Mapping Study
SOA	Software Oriented Architectures
SPL	Software Product Line
SRS	System Requirement Specification
STALE	Structured Test Automation Language framework
STT	Smart Testing Tool
SUT	System Under Test
TD-DSE	Test-Driven Design Space Explosion
TD-MBSE	TD-model-Based System Engineering
TDA	Test-Driven Approach
TDD	Test-Driven Development
TDDM	Test-Driven Development of Models
TDM	Test-Driven Modeling
TFD	Test-First Development
TSV	TabSeparated Values
UID	User Interaction Diagram
UML	Unified Modeling Language
WBT	White-Box Testing
XP	eXtreme Programming

1. INTRODUCTION

In recent decades, there were many approaches that have emerged in the Software testing industry. Model-Based Testing and Test-Driven Development/Behavior-Driven Development are three of them [11]. However, these methods seem to evolve independently and in a different direction.

TDD is an iterative, incremental, and light-weighted development procedure in which requirements are added to the application in short iterations [11]. The cycle starts with describing behavioral requirements of the application in the form of use-cases or user stories. Based on these use-cases or user-stories, the testers write test cases that initially fail, as the requirements are not implemented yet. Then the testers write the code to pass that requirement and continue the process until all the tests are passed. Next, the code is refactored, if required. In this way, TDD enables the developer to specify the expected behavior of the system and start the development process.

Behavior-driven development is an extension of TDD [12, 34]. The key difference between the two approaches is that the BDD uses a ubiquitous language to express the system behavior. Both the technical and non-technical stakeholders of the project share this ubiquitous language. BDD aims at bringing every individual stakeholder of the system on the same page. In TDD, tests are more focused on requirement specification while in BDD, tests are written in user-centered language. However, tests are specified ahead of the code in both development methods [34]. BDD utilizes the ubiquitous language that provides a common technical vocabulary. This language is understood by all the technical and nontechnical stakeholders of the project and transforms requirements specifications into executable test cases [12]. The requirement definitions are specified using a scenario description. These scenario descriptions are mapped into test-methods automatically, utilizing supporting tools [12]. Then, the developers implement the test-methods manually to turn the feature definition into executable test cases [12]. BDD comes with drawbacks. For example, manual generation of test cases can lead to weak and error-prone test cases that can be less effective if the requirements are not appropriately specified [34].

MBT approaches help to overcome the need for application verification [11] and depends on models [17]. Models represent the expected behavior of the system and provide an overview of the entire system [17] and based on these models, test cases are generated automatically [17]. MBT approaches do not support the iterative way of development, instead, they work like a waterfall fashion.

Since these methods (MBT and TDD/BDD) have their strengths and limitations, integrating them can benefit the testing industry by improving the quality and efficiency of testing methods.

This Systematic Mapping Study (SMS) intends to explore the existing methodologies that combine MBT and TDD/BDD and leverage the benefits of both approaches. According to Peterson et al [44], a systematic

mapping study provides a methodology and protocol of the type of research reports and results that have been published by categorizing them and often gives a visual summary, the map, of its results.
Moreover, this thesis will explicitly focus on the integration of Test-Driven Development or Behavior-Driven Development (TDD/BDD) with Model-Based Testing (MBT). We do not aim at weighing the validity of the proposed methodologies, neither compare the integrity of different methods. Instead, the goal is to characterize the available methodologies that integrate MBT and TDD/BDD approaches and extract the provided. To the best of our knowledge, no previous SMS has been conducted to review the approaches integrating MBT and TDD/BDD. Some of the papers have discussed the work regarding combining MBT and agile development but did not debate on TDD and BDD explicitly.
The rest of the thesis is structured as follows. Chapter two provides the introduction of MBT, Agile software development life cycle, TDD, and BDD with a little background where applicable. The study design, research method, and data extraction strategy are discussed in chapter three. Chapter four analyzes the results extracted from the selected primary studies. Threats to validity are discussed in chapter five, and the last chapter concludes the whole research work.

2. BACKGROUND

This section provides background and introduction of Model-Based Testing, Test-Driven Development, and Behavior-Driven Development.

2.1 Introduction to Software Testing

During the primitive times of software testing, testing, and debugging were not two separate tasks and were carried out by the same developer/s to ensure that the system was running correctly [17]. Software testing came out as a separate notion in 1957 and "software tester" became a distinct profession [17]. The software testing process ensures that the system performs all the functionalities and solves the problems as intended [17].

With the passage of time and the rise in market demands, the idea of software testing evolved. Ammann et. al [45] discussed the characterization of the maturity level of test processes. There are five levels of software testing that range from 0 to 4 [45]. At level 0, testing and debugging is the same process. At level 1, the objective is to ensure the correctness. The purpose of testing at level 2 is to find the failures in the software. Testing at level 3 intends to minimize the risk of using the software. While at level 4, software testing is a rational discipline to enhance the quality of the software [45]. There are different testing levels based on software activities [45]. The V-model in software development gives the standard scenario for testing levels [45]. The V-model is one of the software development life cycles in which processes are executed in a progressive manner. Each phase in the V-model is related to the testing phase [45]. Figure 1 represents V-model.

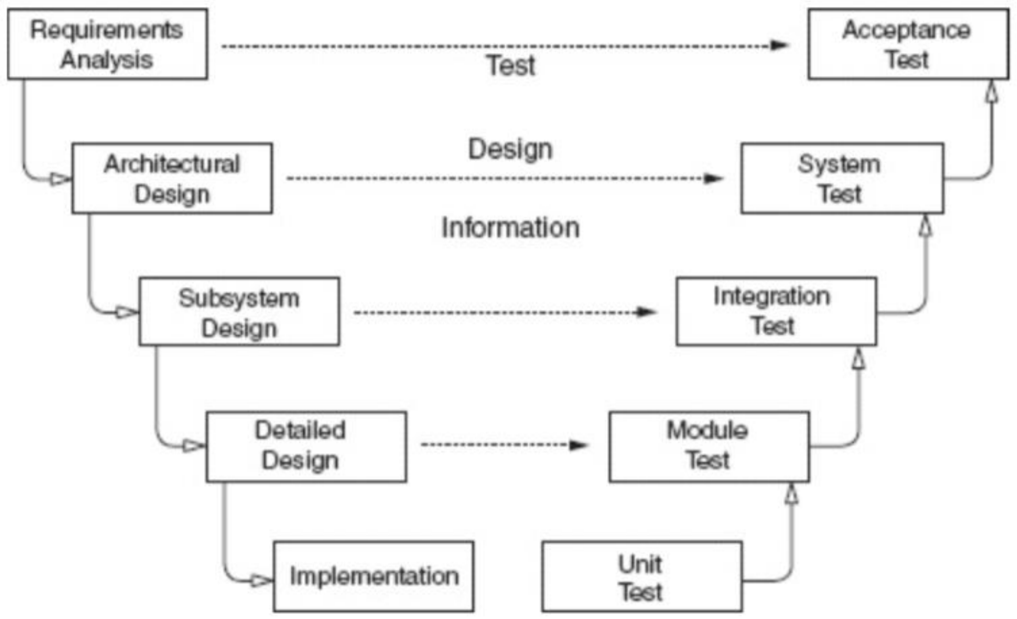


Figure 1.1 Software development activities and testing levels – the "V-Model" [45]

- Unit testing: Unit testing is performed at the implementation phase of software development. At this phase, the core code of the system is developed by the developers. Unit testing is devised to evaluate the individual units of the system [45].
- Module testing: Module testing is implemented at the detailed design phase of software development. Detailed design phase of software development defines the logical structure and behavior of each module. Module testing is applied to evaluate each module individually and to determine the interaction between component units with the corresponding data structures [45].
- Integration testing: Integration testing is carried out at the subsystem design phase. This phase intends to specify subsystem behavior and structure. The purpose of integration testing is to evaluate the system with respect to its subsystem specification and ensure proper communication between modules [45].
- System testing: System testing is performed at the architectural design phase, also known as the system design phase. In this phase of software development, the components of the system are designed based on user requirements. System testing ensures the assembled system matches the overall requirements [45].

- Acceptance testing: Acceptance testing is performed at the requirement analysis phase of the software development process. This step intends to capture and analyze user requirements. Acceptance testing identifies if the final product meets the user requirement or not [45].

Software testing is a process of system verification and validation. The verification deals with building the software in the correct way, and system validation conforms that software meets the user requirements [21]. Generally, software testing technique is categorized in to two main testing techniques, i.e. White-box testing and Black-box testing [21]. In white-box testing, tests are derived from the source code of the system, while in black-box testing, tests are derived from the external description of the system [21]. In terms of Verification and Validation (V&V), white-box testing is used for verification, while the black-box testing is used for validation purposes [21].

2.2.4 White-Box Testing

White-box testing is a software testing method that inspects the code and internal structure of the application to detect the logical errors [21, 32, 42]. In white-box testing, testers create the test cases based on the internal structure of the code and examine whether the code is working as intended [32, 46]. This method is known as white box testing as the testers have the explicit knowledge about the internal structure of the code [21, 32, 42]. White-box testing provides code coverage, path coverage, and condition coverage [21, 42].
Given below are some of the white-box testing techniques.

- **Statement coverage**: In this technique, testers execute every statement in the code at least once [21, 32, 42]. Statement testing is also known as node testing.
- **Branch coverage:** In this technique, each edge/branch is traversed to test all the possible outcomes [21].
- **Condition coverage**: In this technique, testers check the conditional statements like if-else statements or other conditional loops present in the code [42]. Condition coverage tests the outcome of each condition [21, 32].

2.2.3 Black Box Testing

Black-box testing is a software testing method that inspects the behavior of software from the user's perspective. It scrutinizes the system without having knowledge of the source code or its internal paths [21]. Black-box testing can be applied to any level of testing. Based on requirement specifications, it only focuses on the interface errors, software malfunctions, faulty functions, and errors in the expected output of the system [18].

2.2.3.1 Techniques of Black Box Testing

Given below are some of the techniques of black-box testing [18, 21, 41].

- **Equivalence class testing**: In equivalence class partitioning, a system's input values are classified based on their resemblances in the outcomes [41]. While testing, one input from each group is selected instead of all the available inputs. Thus, it reduces the number of possible test cases, avoids unwanted testing, and saves time and effort [21].
- **Boundary Value testing**: This technique deals with the values on boundaries. In many applications, the functional behavior of the system changes at its boundaries [18, 41]. The testers verify whether the system accepts a specific range of input values.
- **Decision Table Testing**: A decision table is created based on possible scenarios to generate test cases [18, 41]. The decision table sets causes and their effects in a matrix form, which helps in formulating test cases.
- **Robustness Testing**: A component of a system is robust when it never crashes [18]. IEEE defines robustness of a component or a system as "the degree to which is the component, or a system performs the intended tasks properly in the presence of invalid inputs "[18]. Robustness testing focuses on testing the robustness of software.
- **State Transition Testing:** In this testing technique, various states of the SUT are tested [41]. These states can change based on conditions. These conditions trigger states that become scenarios and are tested by the testers.
- **Model Based Testing:** In this testing technique, test cases are generated automatically using models (discussed in Section 2.2.1) [1].

Model-Based Testing is an enhancement to black-box testing, used for system validation [17]. MBT applies a set of devised inputs to the system and then specifies the system's behavior in response to those inputs. Model-Based Testing uses models that are derived from system requirements, for test-generation. It is necessary to acknowledge that it is not feasible to test everything, so it must be decided beforehand, what to test and what to leave? This is where models come in. Models are not the system itself, but they are the expected behavior of an SUT. They are an abstraction of the real-time multiplex systems and more straightforward than the SUT and later used by test generation tools to generate tests automatically. The subsequent section provides a brief overview of Model-Based Testing and Models.

2.2 Introduction to Model-Based Testing

Software testing became a fundamental activity in the software development process as the systems became larger, complex, and interoperable. The importance of testing is associated with the complexity of the system and with its application domain [17]. Due to the increased market competition and product variants, an 'early product release' became the strategy of companies against their competitors to capture more market. Consequently, it led software engineers to develop new testing strategies continually and to introduce a more viable method to meet the challenges in the software development and maintenance process. The complexities in software applications also increased the demand for faster but reusable testing strategies. Moreover, due to the complexity and growing size of the software systems, software testers are required to execute many test cases to test the system. Consequently, the testers started lacking an answer to what to test first? What is more important? And what can be discarded? That is where the Model-Based Testing technique comes-in [17]. Model-Based Testing (MBT) depends on models that represent the expected behavior of the system [17]. Instead of writing massive test case specifications manually, models are drawn to generate the tests automatically using a tool. Hence, MBT methods automate the test case generation process and provide transparency and objectivity [17].

2.2.1 Types of Models

Models are the visual representation of any information or data. In the case of Model-Based Testing, models are used to specify the expected behavior of the system under test. Given below are some of the many types of models [21, 22].

State transition machines diagram: State transition models describe the states of the system. These models have one input and one output where the input conditions of the machine trigger the output and determine the transitions between the states [36]. Figure 1.2 gives a graphical representation of a state diagram.

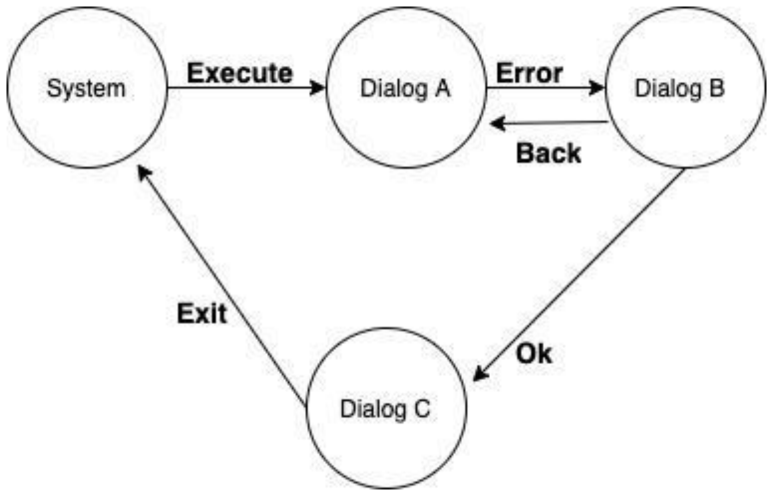


Figure 1.2 State transition machine [36]

Dependency Graphs: The dependency graphs are a directed graph that comprises edges and nodes. Nodes determine the properties or functions of the system. An edge shows the dependency and connections among the nodes within a system. Figure 1.3 shows a graphical representation of the dependency graph [37].

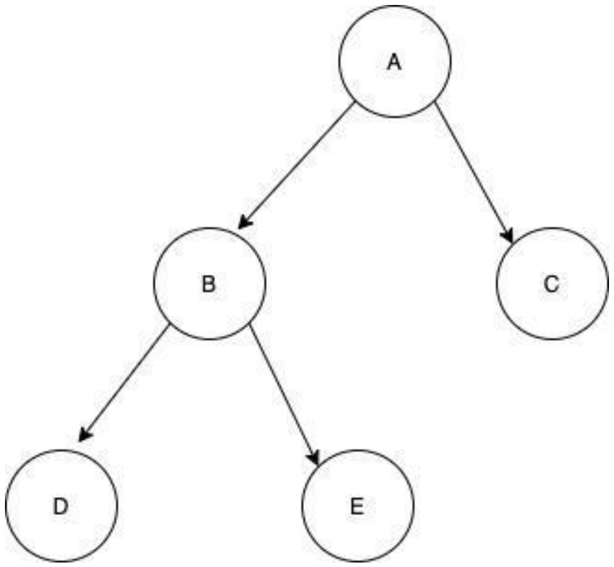
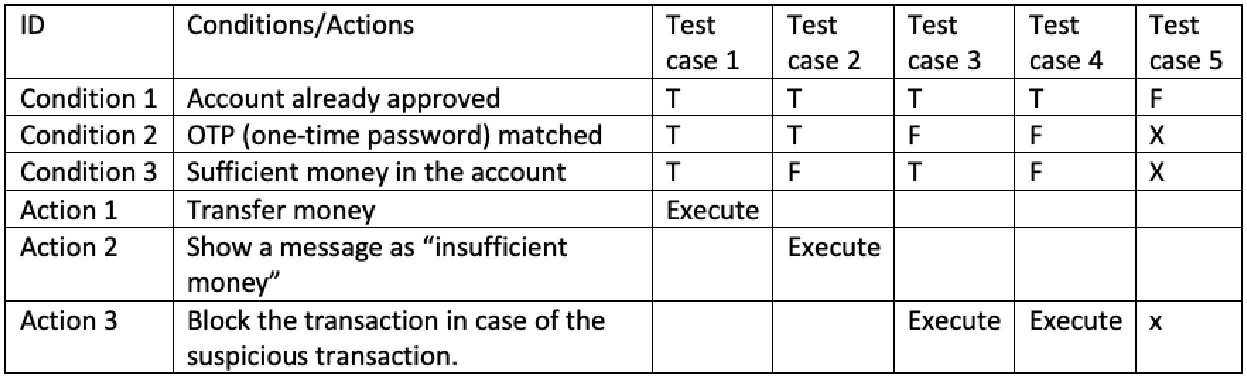


Figure 1.3 Dependency graph

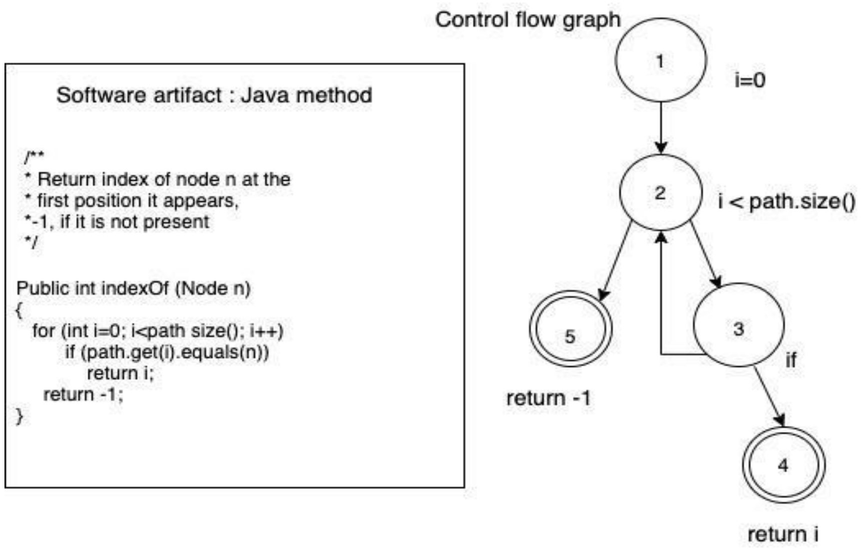
Decision Tables: Decision tables are useful to determine the logical relationships between the functions of the system. The test cases are identified based on input combinations in the decision table. In the decision table, the conditions are inputs, and the actions are the output of the system. Figure 1.4 shows a graphical representation of the decision table [38].



ID	Conditions/Actions	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5
Condition 1	Account already approved	T	T	T	T	F
Condition 2	OTP (one-time password) matched	T	T	F	F	X
Condition 3	Sufficient money in the account	T	F	T	F	X
Action 1	Transfer money	Execute				
Action 2	Show a message as “insufficient money”		Execute			
Action 3	Block the transaction in case of the suspicious transaction.			Execute	Execute	x

Figure 1.4 Decision table [38]

Control Flow graph: Control flow graphs facilitate test generation process. Control-flow graphs determine the overall control flow structure of the system. Figure 1.5 shows an example of a control flow graph.



```
/**
 * Return index of node n at the
 * first position it appears,
 *-1, if it is not present
 */

Public int indexOf (Node n)
{
   for (int i=0; i<path size(); i++)
       if (path.get(i).equals(n))
           return i;
   return -1;
}
```

Figure 1.5 Control flow diagram [45]

Data Flow diagram: In Black- box testing, Data Flow testing is a technique that inspects the sequences of actions to test the paths. It focuses on path selection through the system's control flow. The data flow diagrams are constructed based on the control flow graph [45]. However, the data flow diagrams have no conditions, decisions, or conditional loops. Figure 1.6 provides a graphical representation of data flow diagram.

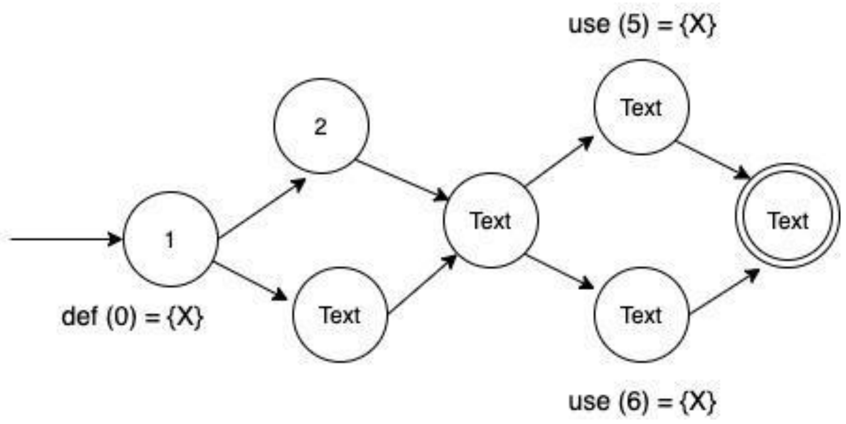


Figure 1.6 Data flow graph [45]

2.2.2. Overview of MBT Development Steps.

Generally, Model-Based Testing includes more straightforward steps. Given below are the different steps of MBT [17, 21].

- **Model development**: The first step in Model-Based Testing is to develop a model for a System Under Test (SUT). There can be one or multiple models for the same functionality of the SUT.
- **Define several inputs for the designed model**: After designing the models, distinct sets of both the challenging and feasible inputs are determined to check the validity of the system.
- **Determine the expected output for this model**: As per requirement specifications, the expected behavior of the system is predefined. This expected output is the model itself, and it specifies how the system should behave in response to those inputs.
- **Test generation**: When model development is completed along with the input and output designing phase, the test cases are automatically generated using an MBT tool. These automatically generated test cases are sequences of actions with predefined inputs and expected output for a given action [17].
- **Test execution:** The generated tests are executed using different MBT tools. Based on the results obtained during test execution, the decision is made. If the test output conforms to the expected output, then the "pass" verdict is assigned, otherwise, "failed" [21]. The failed tests show an inconsistency between the expected behavior of SUT and the actual behavior of SUT.
- **Result analysis**: Results collected from the test execution are analyzed to traceback the error in MBT models [17].

The Figure 1.7 provides a graphical representation of an MBT process.

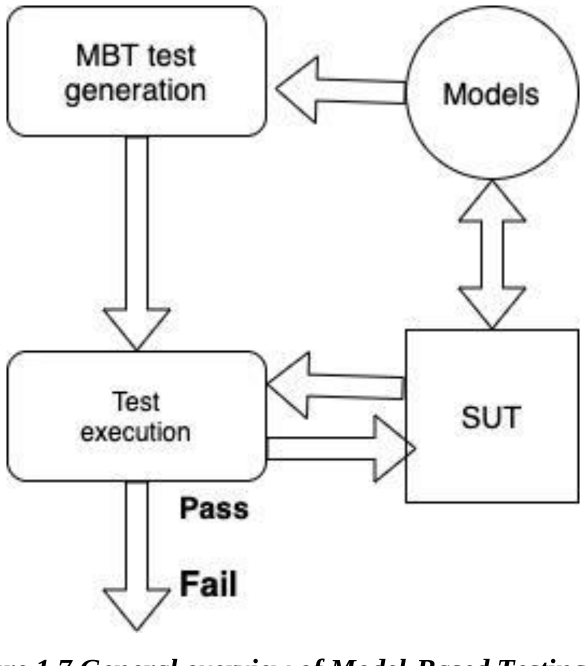


Figure 1.7 General overview of Model-Based Testing

2.2.5. Approaches to Deploying Model-Based Testing

In Model-Based Testing, tests can be executed offline or online. In offline MBT, test cases are first generated and then executed. Test cases can be generated both manually and/or automatically. While in online MBT, requirement-based models are created first. These models are used to generate the sequences at runtime using an online MBT tool. The tests are generated and executed one at a time. Moreover, the online MBT can be deployed in combination with automated test execution [17].

2.2.6. Benefits of Using a Model-Based Testing Approach in The Software Development Process

Models in MBT are an effective way in a software development process to keep all the stakeholders on one page. Models remove the ambiguities, especially among the development team and the testers. MBT models are beneficial in identifying requirement specifications, particularly when requirements are in a text form. By providing a visual representation of text or even thoughts, it makes it easy to validate the requirements. Models also help in identifying problems in the initial stages. Hence MBT is both time and cost-efficient [22,23,24]. Additionally, MBT models facilitate testers to visualize the tests and manage complexities by providing an overview of what is being tested [34]. Models significantly reduce the size of document-based specifications. It is important to keep in mind that document-based specifications could comprise several hundreds of pages. Assessing such a substantial number of pages on a daily basis is not an easy task. Even harder is to check if all the test aspects are fully covered or not. By representing the ideas of tests through models, it makes it easy to demonstrate the covered or left-out paths. It provides fewer chances of errors and overall improves the satisfaction level of the tester's job.

Furthermore, test generation in MBT is well-supported by several testing tools [22,24]. Various test generators are available in the market to automate the process. Different test generators might have a distinct set of input-output. However, they all perform the same task i.e. automated test generation. Moreover, these tools also facilitate testers in other testing activities e.g., documenting the generated interfaces or test-script that can be used later for test execution. After designing the models, testing tools can generate various test suites by replicating the information from the same models. Thus, it also reduces the recurring expenses of test suite generation.

2.2.7. Limitations in Model-Based Testing.

Model-Based Testing requires skilled testers with the required knowledge. MBT has a steep learning curve. Sometimes, it can be highly challenging to identify the best approach in MBT because the diversity in MBT approaches can be drastically confusing. Modeling can be hard. It could take a significant amount of time to think about the architecture for a complex system and fine-tuning it with a testing tool could be even harder [41]. Models can have bugs [17]. It is easy to respond to added information or changes in requirements lately if the tools are scaled appropriately. By making only a few changes in the model will enable the test generator to propagate these changes to all the test cases without making any extra effort. Likewise, if there are bugs in the model, the test generator will propagate these bugs to all the test cases [17].

Model-Based Testing can lead to test case explosion [17]. The ISTQB glossary has defined test case explosion as "the disproportionate growth of the number of test cases with growing size of the test basis, when using a certain test design technique" [17]. However, this discussion is out of the scope of this thesis.

2.3 Introduction to Agile Software Development Process

The agile software development process is an incremental and iterative approach to the development process model. It focuses on evolving requirements and generating solutions through shared efforts of the team, management, and customers [28]. It encourages adaptability and customer satisfaction through the prompt delivery of a product. In this method, the product is split up into smaller modules [28]. These modules are prioritized realistically and delivered in separate releases. Each module is delivered in an iteration [29]. Usually, one iteration is two to three weeks long. Each iteration consists of the following necessary steps [29].

The first step is to plan the iteration. All the stakeholders of the product focus on a common goal by describing the objectives, procedures, and outcomes. The team members are highly encouraged to communicate about the challenges and reservations they have [29]. Iteration execution determines the overall work models and processes. The team incrementally provides demos of their stories to the product owner as soon as they complete a task. The team members also collaborate to share the progress of work and to review product backlogs. Iteration review takes place to analyze the completed work. The teams present their increments and the product owner provides feedback to their work. In this phase, some stories are accepted based on the requirements while others selected for further refinements. The product backlog is revised again for the next iteration planning [29]. In the Iteration retrospective meeting, the team members assess the processes, review the results achieved from previous steps, refine the stories, and find new ways to refine the entire process. Before the next iteration planning, the product backlog is reprioritized [29].

The agile software development approach supports a wide range of Software Development Life Cycle (SDLC). For example, practices like eXtreme Programming (XP), agile modeling, and practical programming are supported by the Agile development method [28]. Kanban and scrum focusing on workflow management, also follow Agile methodology [27]. Some Agile methods support requirement specification related activities e.g., Featured-Driven Development (FDD) facilitates whole SDLC [28].

After the publication of the agile manifesto in 2001, some of the Agile methods like Dynamic Systems Development Method (DSDM), FDD, Rational Unified Process (RUP), XP, and clear being most popular are collectively known as Agile methodologies [28].

Agile manifesto principle states:

· *"Individuals and interactions over processes and tools*

· *Working software over comprehensive documentation*

· *Customer collaboration over contract negotiation*

· *Responding to change over following a plan* [30]*."*

The left-hand values are higher than the right-hand values.

2.3.1 Overview of Agile Software Development Practices

Some practices support the Agile software development process in their whole software development process i.e., from the requirement gathering phase until coding and testing. Test-Driven Development and Behavior-Driven Development are the two most popular approaches among them [35].

2.3.1.1 Introduction to Test-Driven Development

Test-Driven Development is a software development approach that encourages the testers to write the test cases before writing the code [31]. In this discipline, the testers proceed by writing automated test cases for all the small functions that are needed to be implemented and then write the code to pass through these tests. A new code is written only if an automated test has failed [31]. These tests specify what the code is supposed to do. The core objective of TDD is to avoid unnecessary duplication of code and to keep it simple and clear.

Steps of TDD

The TDD process consists of the following steps [31, 33]:

1. The first step in TDD is to *write a test case*. The process starts by picking up a requirement and writing such a clear test case for the selected requirement that is fully understandable by the tester to write the code for it.
2. Next, the developers *run the tests*. Here, the test must fail. If the new test passes, it shows that there is no new code required. It means that either the needed behavior exists already, or the new test has a fault. In this case, the code needs modification.
3. The third step requires developers to *write the code that will pass these tests.*
4. Once all the tests are passed, the developers need to *modify and adjust the code* to improve the structure of the code without modifying its external behavior. This is called code refactoring.
5. At last, repeat from step 1 until step 4.

Figure 1.8 provides a graphical representation of TDD [13]

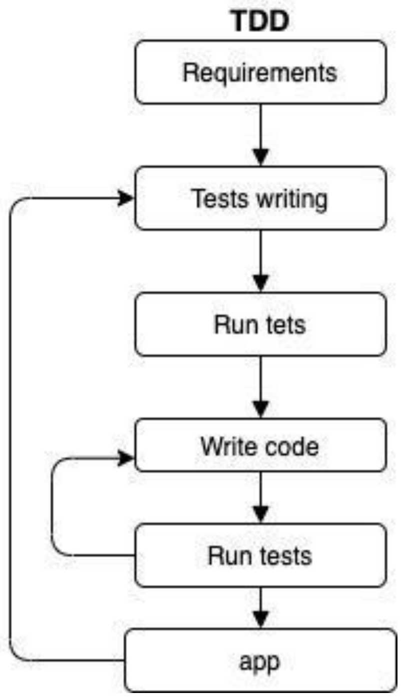


Figure 1.8 overview of Test-Driven Development [13]

Benefits of TDD

Given below are some advantages of TDD [31].

In TDD, errors and bugs can be detected at an early stage of implementation. This reduces a considerable amount of time, cost, and effort. TDD provides a more straightforward and clear structure of code that is easily maintainable. Moreover, the code is simple and easy to understand, so in case of the absence of any team member, other members can easily replace and work on it. Thus, TDD can enhance the performance of the overall team and advocates knowledge sharing.

Pitfalls of TDD [31]

There are no silver bullets in TDD. Tests can find only those bugs that are introduced in the code. If the developer is unaware of a problem, then the tests probably will not be able to help. In the beginning, TDD is slow and time consuming since it could take much time for a tester to imagine the interfaces, code writing, and test cases. In case of any change in the requirements later, the tests should be maintained accordingly, which might cost a lot, especially in terms of time.

2.3.1.2 Introduction to Behavior-Driven Development

Behavior-Driven Development is an agile software development approach and an enhancement to a TDD approach [34]. Like TDD, test cases are written before code. However, in BDD, test cases are user-oriented and more concerned about the overall behavior of the system. To understand TDD, all the stakeholders, including the user/business owners, should have enough technical knowledge to understand the unit test framework [34]. However, in BDD, the shared language can be understood easily by non-tech stakeholders, as it is written in a common language, also known as ubiquitous language [34].

Before implementing Behavior-Driven Development, few essentials should be taken into consideration. There should be well-defined and concrete requirements [34]. It can be achieved by converting specifications into user stories, and each specification must be a valid user scenario [34].

Overview of Behavior-Driven Development cycle

BDD comprises the following steps [35].

1. The first step in BDD is *to identify business features*.
2. In the next step, all the stakeholders collaborate about the requirements and elaborate it further *to identify the scenarios.*
3. The third step is *to recognize all the scenarios* that can guide the testers. These scenarios, under the selected feature, will act as automated tests.
4. Next, is *to determine the procedure* for each scenario.
5. The fifth is *to run the tests* and the tests must fail at this point.
6. When a test is failed, a *new code must be written to pass the tests*.
7. Next, the developer *refactoring the code* to improve the structure of the code without modifying its external behavior and create a reusable automation library
8. Eight, run all the features to pass the test.
9. At last, test reports are generated.

Figure 1.9 gives an overview of the BDD process below.

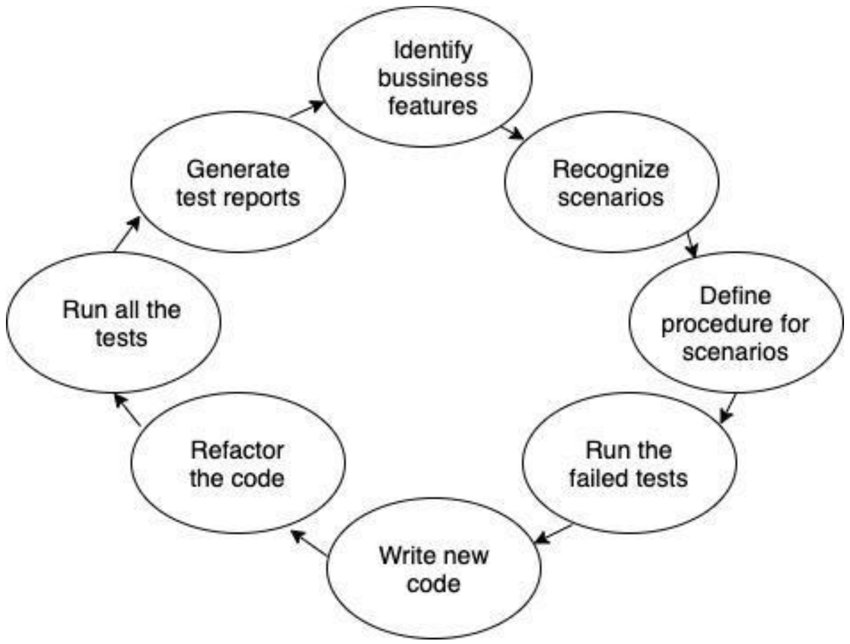


Figure 1.9 overview of Behavior driven development process

3. RESEARCH METHOD

The objective of this systematic mapping study is to investigate the existing research on the strategies to integrate Model-Based Testing and TDD/BDD methodologies in software engineering domains and whether or not they support each other.

The guidelines proposed by Petersen et al. [44] have been followed in this thesis to achieve the objectives mentioned above. The subsequent sections of this paper discuss the procedure and protocol of the study.

3.1 Study Design

The protocol of a study demonstrates the strategies used in mapping studies and provides a clear overview of the procedures and steps taken to perform a Systematic Mapping Study (SMS). Therefore, it is essential to develop a study protocol before conducting the SMS as it lessens the chances of researcher bias.

To conduct our SMS, we leveraged the well-known guidelines proposed by Petersen et al. [44]. Based on the research questions (listed in Section 3.2), the following protocol is used in this thesis: we selected a specific set of electronic databases to conduct the search (listed in Section 3.3.1). We designed an optimal search string (discussed in Section 3.3.2) for each source that could extract the relevant set of search results. Based on the study selection criteria (listed in Section 3.3.3), we analyzed the abstracts of the obtained primary studies and filtered out the irrelevant papers. After reading the full text of all the collected primary studies, we applied snowballing to include more relevant studies that could not be found during initial searches. Next, based on the quality assessment criteria (provided in Table 3.4) we assessed the quality of each primary study to exclude the papers which do not meet the minimum quality requirement. At last, required information from the remaining primary studies was extracted to answer the research questions. The extracted information is presented in the data extraction form (shown in Table 3.6)

Snowballing is a procedure used during conducting systematic literature review to identify and include more relevant studies in the research work [43]. Figure 3.1 illustrates the snowballing procedure in detail

[43].

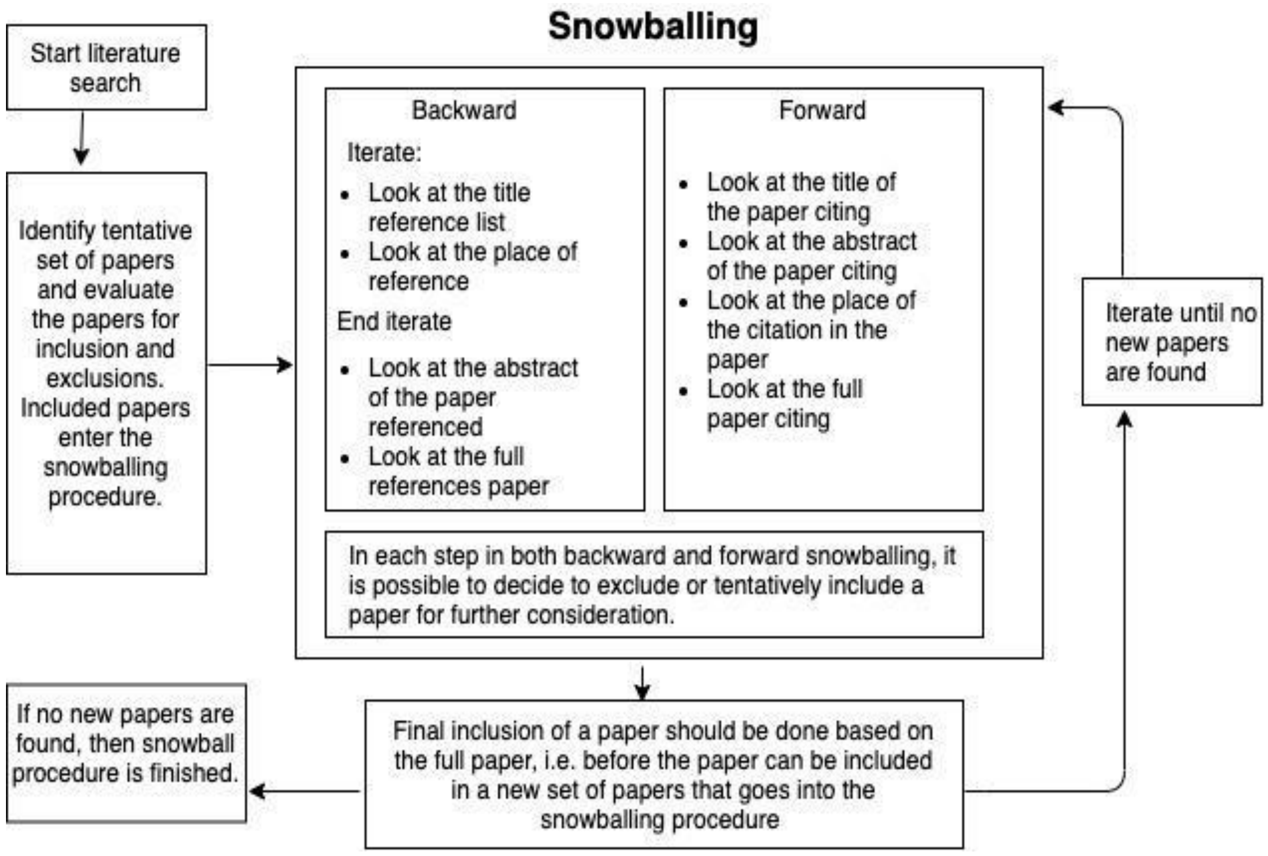

Snowballing
Start literature search
Identify tentative set of papers and evaluate the papers for inclusion and exclusions. Included papers enter the snowballing procedure.
Backward
Iterate:
• Look at the title reference list
• Look at the place of reference
End iterate
• Look at the abstract of the paper referenced
• Look at the full references paper
Forward
• Look at the title of the paper citing
• Look at the abstract of the paper citing
• Look at the place of the citation in the paper
• Look at the full paper citing
In each step in both backward and forward snowballing, it is possible to decide to exclude or tentatively include a paper for further consideration.
Iterate until no new papers are found
If no new papers are found, then snowball procedure is finished.
Final inclusion of a paper should be done based on the full paper, i.e. before the paper can be included in a new set of papers that goes into the snowballing procedure

Figure 3.1 snowballing procedure [43]

Figure 3.1 shows that there are two types of snowballing procedure. (i) Backward snowballing, (ii) Forward snowballing. In backward snowballing, new studies are identified using the reference list. The new papers are reviewed and included based on their general information i.e. title, author, publication year and the study selection criteria specified in Section 3.3.3. while in forward snowballing, the new papers are identified based on those papers citing the paper being examined [43].

3.2 Research Questions.

The fundamental step in a systematic mapping study is to identify the research questions explicitly. Since, the whole study is determined by the research questions, therefore, specifying the right question is necessary to stem the subsequent findings accurately. Hence, the research questions not only need to be eloquent to both researchers and practitioners but also precise to achieve error-free findings. In this study, we are interested in the research or the work carried out to combine Model-Based Testing with Behavior-Driven Development or Test-Driven Development. Based on this objective, the following research questions are defined.

RQ.1: Which studies discuss integrated methodologies of MBT and TDD/BDD in the context of industrial environments?

This research question aims at inspecting the research papers that discuss the application of these (TDD/BDD and MBT) approaches in an industrial environment as well as in academic and research environments. Moreover, this question intends to extract the outcome information of applying these approaches to a particular application domain, as stated in the corresponding study.

RQ.2: Are there studies/research papers available that focus on the tools that support TDD/BDD and MBT?

This research question aims at identifying the trends in studies or research work that discuss the existing tools that support TDD/BDD and MBT. Moreover, in research this question, we try to identify and state the newly developed tools introduced by the author of that study.

RQ.3: What methodologies or solutions are presented to combine MBT and TDD/BDD?

This research question intends to explore and review the studies that discuss the methodologies that combine MBT and TDD/BDD so that both approaches can be merged and used for testing. The purpose of this question is not to compare the integrity or validity of different approaches but only to state the proposed method as presented in that study.

3.3 Search Strategy

The objective of the search strategy is to retrieve the most relevant primary studies as many as possible. Based on the objective, the source selection of this thesis, search string, and study selection criteria are defined and discussed in subsequent parts of this chapter.

3.3.1. Source Selection

One of the critical aspects of conducting a well-organized search strategy is to define the publication source. Instead of aiming for a confined set of publications or conferences, we selected seven resourceful electronic databases that hold sufficient publications in the field of software engineering. These databases are *IEEE Xplore, ACM Digital Library, Science Direct, Springer, Web of Science, Google Scholar, and Scopus.* Moreover, the desired search items are research papers, journal papers, research articles, and conference papers.

3.3.2 Search String

To conduct the research and find relevant primary studies in the specified electronic databases, an automatic search was applied. The automatic search was carried out by entering a specific set of keywords called "search string" on the search engine of the given database. The search string was constructed based on the primary goal of the thesis. The string was designed not only to automate the search but also to obtain an up-to-the-mark and a high precision rate in primary studies. The string led to a considerable number of results. In different databases, the search engine handles the search string differently. Considering this, distinct search strings were generated for each database. However, the keywords and their semantic meaning were consistent, and only the syntactic structure was changed.

The search engine in Web of Sciences generated results that contained many irrelevant papers, particularly from the Hardware and Network Engineering fields. According to the exclusion criteria no. 3, papers not related to Software Engineering must be excluded. To achieve this, the papers that contained the word "CDMA" and "satellite" in their title, were explicitly excluded. Given below are the search strings selected for each database. The semantic meaning of all the strings is the same while the syntax varies for each database.

1. **IEEE**: (model* OR Model-Based* Testing) AND (Test-Driven* OR Behavior-Driven*)
2. **SCIENCEDIRECT**: (Model-Based Testing) AND ("Test-Driven Development" OR "Behavior-Driven Development")
3. **SPRINGER**: "model-based" AND (Test-Driven AND Behavior-Driven)
4. **WEB OF SCIENCES**: TS=(Behavior-Driven Development) OR TS=(Test-Driven Development) NOT TS=satellite NOT TS=CDMA AND TS=(Model-Based Testing) OR TS=(Model*Based Testing) OR TS=(Behavior-Driven Development) AND TS=(model* OR Model-Based Testing)
5. **ACM**: record abstract:(+model-based +testing +test +driven +testing +behavior-driven) AND acmdlTitle:(+model +based +testing)
6. **SCOPUS**: ("Model-Based Testing") AND (Test-Driven OR Behavior-Driven) AND (test AND generation)
7. **GOOGLE SCHOLAR**: "Model-based Software Testing" AND ("Test-Driven Development" OR "Behavior-Driven Development")

Table 3.1 shows the results returned from each electronic database after applying the specified search string. This table is created using MS word.

Table 3.1 Electronic database

source	Total Search results
IEEE	162
Science direct	269

Springer	22
Web of sciences	221
ACM	80
Scopus	220
Google Scholar	47
Total	1021

3.3.3 Study Selection Criteria

Inclusion criteria

The following are the inclusion criteria used to discard irrelevant studies:

1. Papers whose abstracts, titles, or keywords or any of the other terms that we specified in Section 3.3.2.
2. Papers that are written in English.
3. If an extended version (e.g., book chapter or journal paper) of a conference paper found in the search results with more technical details, only the extended version was included.
4. Papers published later than the year 2003.
5. Papers that explicitly discuss Model-Based Testing and Test-Driven/Behavior-Driven Development.

Exclusion Criteria

The following are the exclusion criteria used to discard irrelevant studies:

1. Papers not subject to peer review.
2. The publication is a secondary study (e.g., a literature review);
3. Papers not related to Software Engineering.
4. Duplicated papers (e.g., returned by different search engines).

3.3.4 Primary study selection

Figure 3.2 shows the number of publications identified at each phase of the process. Initially, the search string returned one thousand and twenty-one results from all the electronic databases. After refining papers through abstract, two hundred and eight papers were selected. Then, we applied inclusion and exclusion criteria and obtained ninety-nine papers. After full-text reading, ten papers were selected as primary studies.

At last, we performed the snowballing procedure and obtained six new studies. Thus, a total of *sixteen papers* were selected as a primary set of papers.

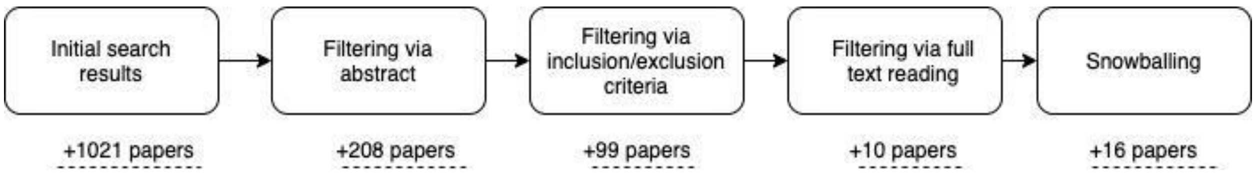


Figure 3.2 Number of primary studies at each stage during the selection process

The Table 3.2 gives a thorough overview of the primary study selection of this thesis. In this Table, the column 'Source' lists all the electronic databases selected for this SMS. The number of total papers returned by each publication source is presented in the column 'Returned search result.' The 'Filtering via abstract' column shows the total number of studies obtained after filtering through the abstract of that study. Column 'Filtering via inclusion/exclusion' shows the number of papers achieved from each database, after applying inclusion and exclusion criteria. The 'Filtration via full-text reading' column presents the number of papers obtained after full text-reading. The last column 'Snowballing' represents the number of studies included after performing snowballing procedure.

Table 3.2 Results returned after snowballing

source	Returned search result	Filtering via abstract	Filtering via inclusion/exclusion criteria	Filtration via full-text reading	Snowballing
IEEE	162	40	21	6	+3
Science direct	269	68	8	0	+1
Springer	22	10	4	0	0
Web of sciences	221	36	36	1	+1
ACM	80	27	13	0	0
Scopus	220	21	14	3	+1

Google Scholar	47	6	3	0	0
Total	1021	208	99	10	+6

The Table 3.3 shows the detailed overview of distribution of papers over databases. The table also provides general information about the papers, i.e. Study title, library catalog, date of publish, and the names of the authors.

Table 3.3 Distribution of studies over channels

Distribution of studies over channels.				
Study ID	Study Title	Library catalog	Publish date	Author of the study
1	Enhancing Test-Driven Development with Model-Based Testing and Performance Analysis	IEEE	2008	Sebastian Wieczorek, Alin Stefanescu, Mathias Fritzsche, Joachim Schnitter
2	Binding Requirements and Component Architecture by Using Model-Based Test-Driven Development	IEEE	2012	Dongyue Mou, Daniel Ratiu
3	D-MTBDD: An Approach for Reusing Test Artefacts in Evolving Systems	IEEE	2006	Tha ́ıs Harumi Ussami, Eliane Martins, Leonardo Montecchi
4	Skyfire: Model-Based Testing with Cucumber	IEEE	2016	Nan Li, Anthony Escalona, Tariq Kamal
5	Test-Driven Modeling of Embedded Systems	IEEE	2015	Allan Munck, Jan Madsen
6	Formal Test-Driven Development with Verified Test Cases	IEEE	2012	Bernhard K. Aichernig, Florian Lorber, Stefan Tiran
7	Test-Driven Modeling for Model-Driven Development	IEEE	2004	Yuefeng Zhang
8	Lightweight Model-Based Testing for Enterprise IT	IEEE	2018	Elodie Bernard, Fabrice Ambert, Bruno Legeard, Arnaud Bouzy

9	The Adaptation of Test-Driven Software Processes to Industrial Automation Engineering	IEEE	2010	Reinhard Hametner, Dietmar Winkler, Thomas Östreicher, Stefan Biffl, Alois Zoitl
10	Agile Development Cycle: Approach to Design an Effective Model-Based Testing with Behavior-Driven Automation Framework	Scopus	2014	Sandeep Sivanandan, Yogeesha C. B
11	Crossing Model-Driven Engineering and Agility Preliminary Thought on Benefits and Challenges	Scopus	2010	Vincent Mahé, Benoît Combemale, and Juan Cadavid
12	A Model-Driven Approach for Behavior-Driven GUI Testing	Web of Sciences		Hendrik Bünder, Herbert Kuchen
13	Bridging Test and Model-Driven Approaches in Web Engineering	Scopus	2009	Esteban Robles Luna, Julián Grigera, Gustavo Rossi
14	Test-Driven Development of UML Models with SMART Modeling System	Web of Sciences	2004	Susumu Hayashi, Pan Yibing, Masami Sato, Kenji Mori, Sul Sejeon, and Shusuke Haruna
15	Behavior-Driven Development of Foundational UML Components	Science Direct	2010	Ioan Lazar, Simona Motogna, Bazil Parv
16	Improved Under specification for Model-based Testing in Agile Development	Scopus	2010	David Farago

3.3.4 Study Quality Assessment

Based on the inclusion and exclusion criteria specified in Section 3.3.3, 16 papers were finalized and selected as a primary study for this systematic mapping study. In this section, we are aiming at investigating the papers more profoundly to ensure the quality. The goal is to identify whether the selected papers fulfill the required quality standard. In order to sort out the papers that satisfy the quality criteria, we created a quality assessment checklist to obtain a rational set of papers that are appropriate enough for the data extraction phase, as shown in Table 3.4. There is a total of 6 quality criteria on the list. Each criterion has a maximum and minimum score i.e., 2 and 0, respectively. The quality of each primary study will be assessed against this checklist.

Table 3.4 Quality assessment checklist

Quality criteria #	*Quality criteria*	Criteria definition
Qc1	Does the paper mention the goal/aim of the study properly?	- what is the basic objective of the study?
Qc2	Does the paper state the environment or software domain on which the study is based?	- In which software domains TDD, BDD, and MBT testing are applied?
Qc3	Does the paper specify any tool/tools that support TDD, BDD, MBT or their own introduced method?	- Are there any existing tools that support TDD, BDD, MBT, or all of them?
Qc4	Does the paper describe any development method to combine TDD, BDD, and MBT?	- What strategy or development did it use to combine the three approaches? Is there any technique available?
Qc5	Is the proposed methodology/tool support validated through a case study?	Has the author used any running example or a case study in detail to validate the proposed methodology or tool support?
Qc6	Does the paper state the result?	- what was the conclusion? Whether the project was a success or a failure?

Another Table 3.5 represents the distribution of primary studies over the criteria list. This table will help to demonstrate whether a study falls into the category of the corresponding criteria. A paper can have a maximum of 12 points and a minimum of 0 points. The passing score for a paper is 3, which is 25% of the total. Hence, a paper scoring less than three will be dropped out and will not be included in this SMS. As illustrated in table 3.5, study 4, 6, and 13 have highest marks i.e. 12 while the study 11 has lowest marks i.e. 4. However, all the papers fulfilled the quality assessment criteria.

Table 3.5 Distribution of papers over quality criterion

Paper No.	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*	P13	P14	P15	P16
Qc1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Qc2	2	1	0	2	2	2	2	2	2	0	0	2	2	2	0	0
Qc3	0	0.5	0	2	0	2	0.5	2	0	2	0	1	2	2	2	0
Qc4	2	1	2	2	2	2	2	0	2	0	0	0	2	0	2	2
Qc5	1	0	0.5	2	2	2	2	2	2	0	0	2	2	2	0	0
Qc6	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
total	9	6.5	6.5	12	10	12	10.5	10	10	6	4	9	12	10	8	6

3.3.5. Data Extraction

In this phase, we created a table form to extract the data. After reading the full text from all the sixteen studies, we extracted the information required to answer all the research questions. Two types of data have been collected from each study i.e., general information and the information directly associated with the review questions and quality assessment. The general information includes study ID, the title of the paper, the author's name, publication date, and the publication venue. Table 3.3 shows the general information. While the second type of information is related to the research questions. We targeted the data that describes the domain or platform/environment in which the study has been conducted. Moreover, the column "methodology based" indicates the methodology/approach discussed in that study. The last two contents of this table are the "type of data extracted" and the "supporting tools." The data extraction table form is shown in Table 3.6. A general overview of the research contribution of the selected primary studies, shown in Figure 3.3, represents the number of primary studies concerning its contribution type. We have developed the graph in which the vertical bars show the number of primary studies while the horizontal values indicate the contribution type. The graph shows that eleven out of sixteen papers discuss the methodologies to integrate MBT with TDD, while four papers focus on the integration of MBT with BDD. Ten out of sixteen studies illustrated their proposed approaches using an appropriate case study, experiment, or an example. Ten papers either suggest a supporting tool or introduce their tool/s to support the proposed process.

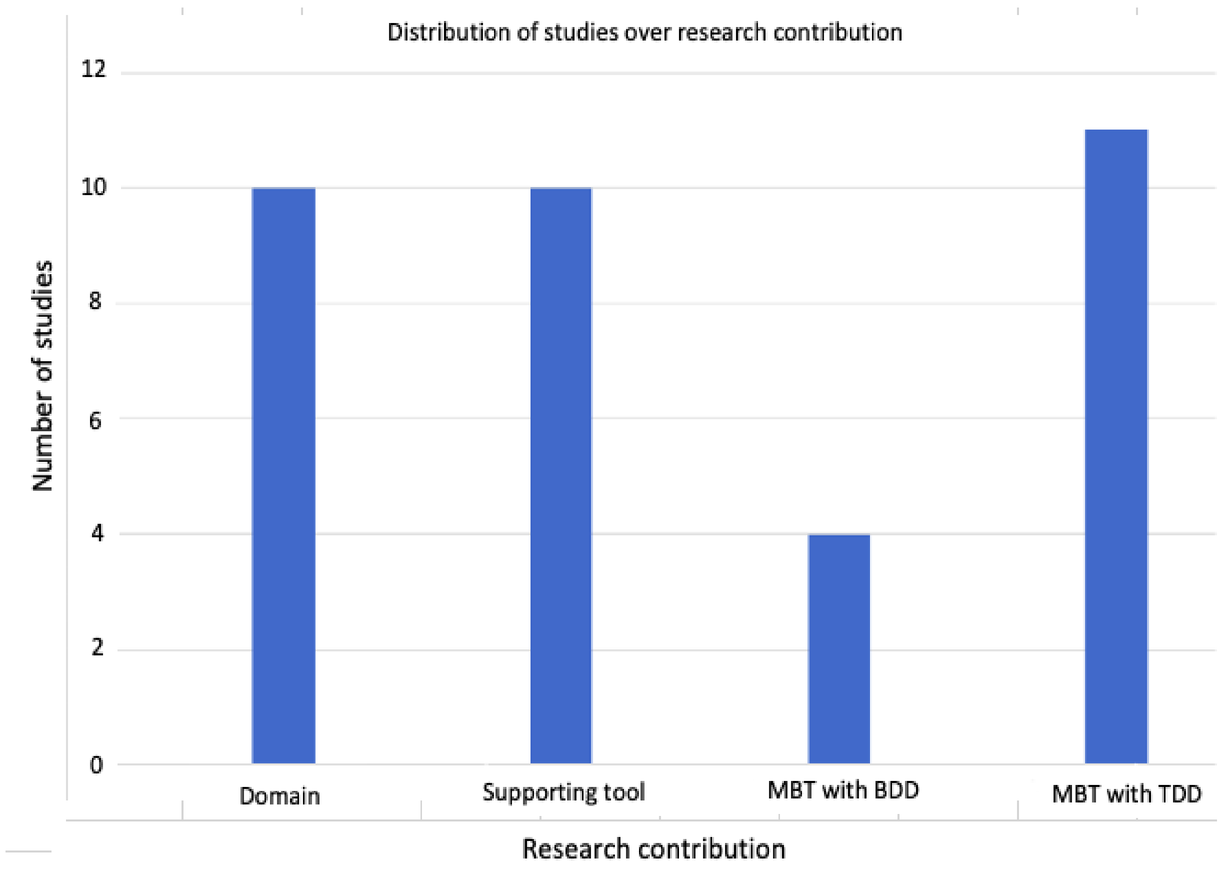


Figure 3.3 Study distribution with respect to contribution type

Furthermore, some papers could answer more than one research question, while some studies could answer only one research question. Some of the studies that did not answer a single research question were included in the original list as they passed the criteria. A pie chart is provided in Figure 3.4 that presents the number of papers answering the research questions. The pie chart is created using Excel sheets. In the chart, RQ denotes the research question while the number depicts the research question number.

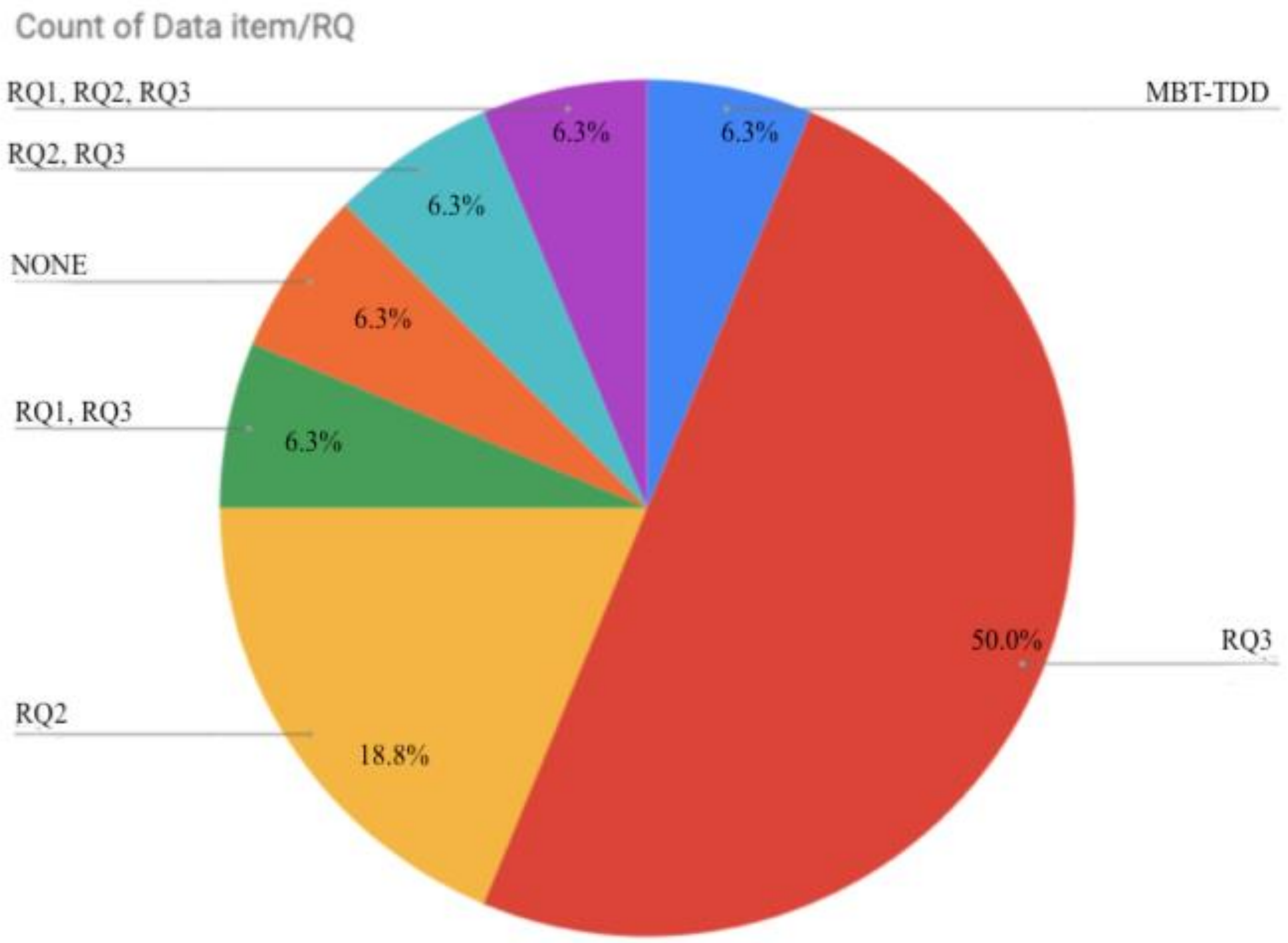


Figure 3.4 Number of research papers relevant to the research question

Moreover, to eliminate the biases of the researcher, a table content "type of data" is labeled against each primary study. The objective of this content is to determine whether the given study is serving the purpose. Furthermore, the results were reviewed by two other researchers (supervisors) to affirm the validation of the data extracted.

For the result validation, we created a table to enlist the most frequently used words appearing in the abstract and the title of the paper and used an online generator tool to create a word cloud. Moreover, the variations occurring in the same words were clustered in a single word—for example, models, models, and modeling as a model. Next, a word cloud was generated to provide a visual representation of the most frequently used words as shown in Figure 3.5.

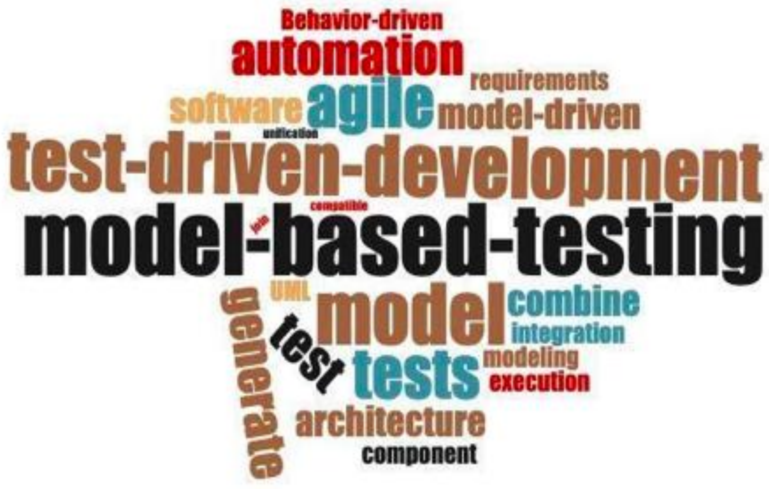


Figure 3.5 Word cloud of most frequent words.

Figure 3.5 that the five most frequently occurring words are, Test-Driven Development, Model-Based Testing, model, tests and agile, which is related to the specified selection criteria. However, the word Behavior-Driven Development occurs in the Figure but not in the five most frequently occurring words. To answer the research questions, we designed a data extraction form (shown in Table 3.6) to aggregate data from all 16 the primary studies. We gathered two types of information from each study: general information like paper title, authors, and publication year; and information directly related to the research questions.
In order to avoid errors and mitigate researcher's biasedness during the data extraction phase, we used a form as shown in Table 3.6.

Table 3.6 [Data extraction form]

Type	Data Item	Type of Information collected
General	Paper tile	Title of primary study
	Author(s)	Set of names of authors
	Year	Calendar year of publication
RQ1	Domain and environment	Context and applicable domain of the primary study

RQ2	Toolset/tool support	Tools used for practicing the combined approach
RQ3	Methodology	Techniques/ methods used to practice MBT and TDD/BDD

Further, Table 3.7 gives an overview of the data extracted. The targeted data are ‘domain or platform/environment’ in which the study has been conducted. The ‘supporting tools’ column enlist the tools that are discussed in the paper. The last two data elements of this table are "methodology based" and the "type of data extracted.” ‘Methodology based’ column indicates the methodology/approach that is discussed in the paper.

Table 3.7 [Data extraction Table]

Author	**domain/environment**	**described supporting tools Y/N**	**methodology based**	**type of information extracted**
Wieczorek et al 2008	Business Domain (ERP software/Business software)	N	MBT-TDD	development method to combine agile and MBT
Mou et al 2012	Not specified	Y(AF3)	MBT-TDD	methods that can combine TDD and MBT
Ussami et al 2006	Not specified	N	MBT-TDD	reusing test artifacts.
Li et al 2016	Business domain	Y(Skyfire)	MBT-TDD	Integration of sky fire is with BDD
Munck et al 2015	Industrial domain (Embedded system, hearing aid)	N	TD-MBSE(MBT-TDD)	Modification of TDD and expansion into TD-MBSE
Aichernig et al 2012	Industrial domain (Car alarm system)	Y	MBT-TDD	formal TDD cycle and its benefits and limitations.
Zhang et al 2004	Industrial domain (Motorola’s telecom system)	Y (Telelogic TAU SDL Suite)	TDM(MBT-TDD)	The workflow of TDM and comparison b/w TDM and TDD

BERNARD et al 2018	Business domain (e.g. ERP, bespoke)	Y(Yest)	MBT	decision tables and MBT for ATDD in Agile
Hametner 2010	Business domain	N(only UML family)	MBT-TDD	an adapted TFD process regarding test case generation
Sivanandan et al 2014	Not specified	Y(Graphwalker)	MBT-BDD	The integration between the Graphwalker and Robot Framework with Robot Framework regarding the online testing approach.
Mahé et al 2010	Not specified	N	MBT-AGILE	perspectives based on related works conducted to combine MDE and Agile
Bünder	Not specified (GUI Testing)	Y(Slang)	BDD-MBT	Conclusions based on controlled experiments regarding the comparison between proposed and some existing tools.
Luna et al 2009	Not Specified (Online bookstore, Web application)	Y(WebML, Web Ratio, MDWE)	TDD-MDE	integration of TDD and MDE
Hayashi et al 2004	Not specified (Hardware smart appliances)	Y(SMART)	TDD-MDD	basic functionality of the proposed tool and its comparison with other UML tools.

Lazar et al 2010	Not specified	Y(bUML)	BDD-MDD	overview of BDD profile, introduction to the bUML tool.
Farago et al 2010	Not specified	N	MBT-AD	overview of agile and MBT integration

4. RESULT AND ANALYSIS

To answer the given research questions, all the 16 primary studies were analyzed independently. The results are based on the conclusions and opinions developed by each study.

4.1 Research Question 1:

Which studies discuss integrated methodologies of MBT and TDD/BDD in the context of application domains?

In response to the given research question, we analyzed the domains of all the 16 primary studies independently. In this research question, the application domains are classified into three categories are as follows: (i) industrial domain: represents case studies that involve real-time embedded systems. This domain deals with both the hardware and software systems. (ii) business domain: represents case studies which involve the real-time business processes that generate business revenues. This domain only concerns software systems. (iii) academic domain: represents case studies that are based on theories, experiments, or conceptualized examples. The extracted results are based on the conclusions and opinions developed by the author of each study. An overview of the findings of all the 16 primary studies is shown in Table 4.1.The table shows only those studies that discuss the proposed methodology through a case study. In this case, only ten papers are involved.

Table 4.1 Overview of primary studies based on application domains

Study ID	Domain	Example used	Proposed method	Research objective
1	Business domain	Business process component	TDD AND MBT	Performance analysis

4	Business domain	Web service (ROC)	Skyfire	Test generation.
5	Industrial domain	Hearing aid	TD-MBSE	Modeling of system architecture and behavior
6	Industrial domain	Car alarm system, Anti-theft alarm	Ulysses	Model refinement and test case generation
7	Industrial domain	Telecommunication system	Test-Driven modeling	Automatic testing through simulations.
8	Academic domain	Online website, Train ticketing system	Yest	Modeling requirements
9	Academic domain	Automation system, Waterworks control application for the irrigation system.	Adapted TDD	Test case generation
12	Academic domain	Controlled experiment, AB/BA crossover at Swiss bank	SLANG	Automatically executable GUI test case generation
13	Academic domain	Web application, Online book store	Integration of TDD and MDWE	Tests transformation
14	Academic domain	Remote control system, A room air conditioner with remote controller	TDDM methodology	Modelization and execution of test cases.

Wieczorek et al. [1] presented an approach to integrate Test-Driven Development with Model-Based Testing for developing a business application based on Service-Oriented Architectures (SOA). Business management software like Enterprise Resource Planning (ERP) are usually vast and complex as they combine several functions from many organizational parts into a single logical system. To cope with the complexity of such systems, SAP is using several intensive testing techniques during SDLC. SAP is one of the business management software manufacturing firms [1]. The study [1] suggests using MBT on system-level and TDD on component level. The method proposed (discussed in Section 4.3) by Wieczork et al. [1] is designed for SOA based software development.

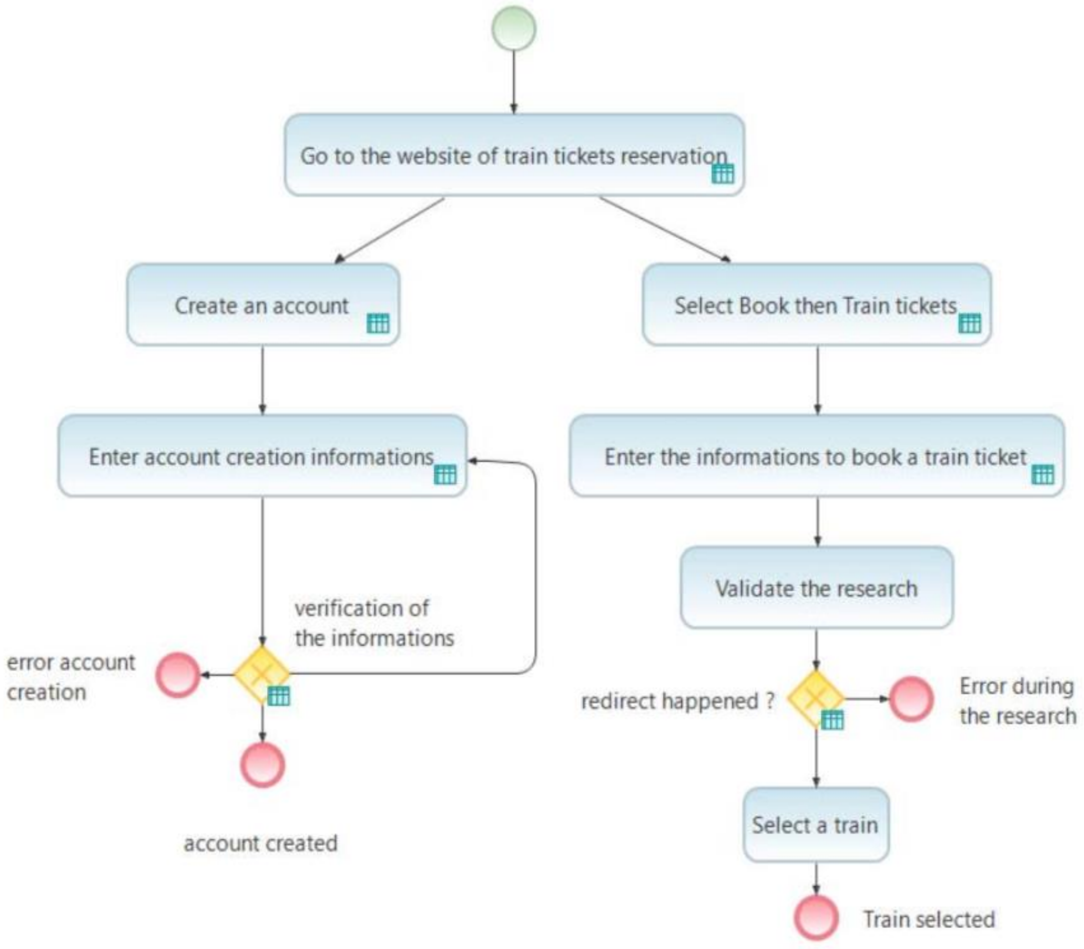
As demonstrated in [4], a Model-Based Testing tool is presented, called Skyfire, that leverages the mapping mechanism of a Behavior-Driven Development tool known as Cucumber [4]. The study [4] presented an industrial case study that discusses a real-word product at Medidata, Roc. Roc is a web service, developed in Java, that wraps the Elastic MapReduce (EMR) of Amazon Web Services (AWS). For large data applications, Roc works as an infrastructure. It offers simpler APIs to use EMR to monitor and process high volume data. The methodology presented in the study [4] uses Skyfire to test Roc. Skyfire has been used to generate Cucumber test scenarios elicited from the behaviors of Roc. The behavior of the Roc is specified with the help of Unified Modeling Language (UML) state diagram [4]. Generally, the MBT approaches require a significant amount of time and concentration on the input values and the interaction between the system actions. With Skyfire, the practitioners need not to focus on the input values or mapping information of the models [4]. They are only required to concentrate on the system behavior and to comprehend the interactions among the system components.

Munck et al. [5] focused on the utilization of formal and statistical model checking with the formalism of timed automaton (discussed in Section 4.3). The results obtained from the proposed technique, known as Test-Driven Model-Based Systems Engineering (TD-MBSE), is applied to an industrial case regarding an open system architecture of hearing aid. The study [5] suggests using formal verification instead of applying simulation for the behavior verification of the system. TD-MBSE is facilitated by a model checking tool using UPPAAL for modeling the architecture and behavior of the system. The TD-MBSE framework enables engineers to apply a Test-Driven approach for use-cases, scenarios, requirements, architectural modeling, and behavioral modeling. However, the study [5] only focuses on the modeling of behavior and architecture of the system. The experimentation states that the Test-Driven approach is facilitated by UPPAAL queries to model the architecture and behavior of the system. The process of modeling the project comprised the following steps: (i) gathering data: the complete information about the system is gathered from several stakeholders from the hardware and software department. (ii) process data. The UPPAAL templates are used for the verification of architectural requirements as well as to model the timed automaton. However, the models are developed based on the proposed method (iii) to obtain results. In order to verify the requirement, the proposed method suggests applying formal model checking to obtain the results from the models. (iv) to find alternates. To explore the design space of the system, other alternative designs with different architectural and behavioral solutions are created.

Aichernig et al. [6] introduced a development process to leverage the advantage of TDD, MBT, and formal methods. The author presented a technique to integrate formal methods into an existing SDLC process. The study [6] illustrated the proposed methodology using a case study 'car alarm system', inspired from Ford's automotive project MOGENTES [6]. In this case study, the abstract test cases are generated through formal models. Based on the requirements, these abstract test cases are verified with model-checking. After model refinement, the engineers apply the TDD approach by implementing test cases one by one until all the tests pass. In this case study, Back's Action system [6] is used as a modeling language and the tool 'Ulysses' (discussed in Section 4.2) for test case generation. At the same time, model checking is performed through the CADP toolbox (discussed in Section 4.2).

Zhang et al. [7] presented the idea of Test-Driven Modeling (TDM) that implements the extreme programming Test-Driven pattern to a model-driven development process. The process is divided into six phases (described in Section 4.3). The TDM process is illustrated through a real-world industrial case study named as "Motorola's iDen division." In this case study, the proposed methodology "TDM" is used to migrate an extensive telecom system to a new platform. There are nine subsystems out of which six are generated automatically, and all the nine subsystems are deployed separately. Around sixty people are working on the project, however, divided into groups. Each team is held responsible for a distinct task and attends weekly meetings in person. Motorola is using SDL and Telelogic TAU SDL suite in this project. The TDM process uses Message Sequence Charts for unit test cases and system analysis. The overall process involves six phases in which tests are automated through simulations, and executable models are used as design documents for the software system.

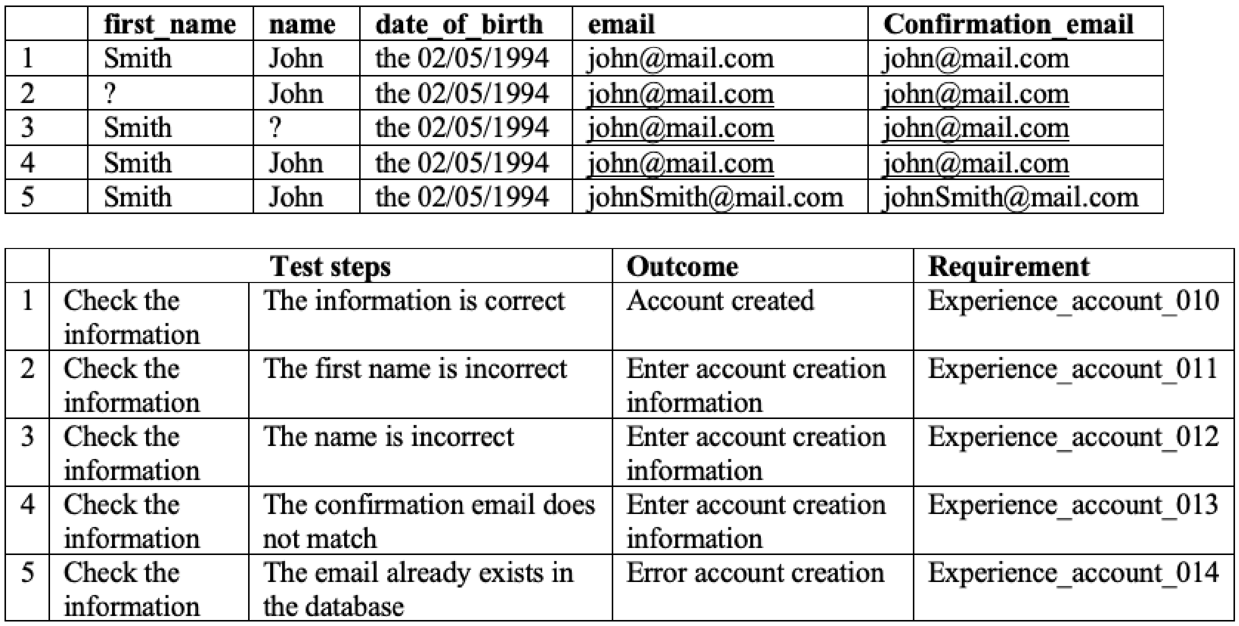
Bernard et al. [8] proposed a lightweight MBT for a specific category of applications, i.e., Enterprise IT systems. The study demonstrated that the reason for the slow-paced growth of MBT approaches in the testing world is the steep learning curve. A lightweight MBT tool called Yest is introduced that only serves Enterprise IT applications. The proposed tool is demonstrated through a case study of an online train ticketing system. The website is intended to book the train tickets online by creating an account first. Yest involves graphical representation of the workflow of the system that is also connected with the decision table. The graphical representation of the workflow of the online train ticketing system isystem is shown in Figure 4.1.



Go to the website of train tickets reservation
Create an account
Select Book then Train tickets
Enter account creation informations
Enter the informations to book a train ticket
Validate the research
verification of the informations
error account creation
account created
redirect happened ?
Error during the research
Select a train
Train selected

Figure 4.1 workflow of online train ticketing system

Figure 4.2 provides the graphical representation of the decision table associated with the workflow of the online train ticketing system.



	first_name	name	date_of_birth	email	Confirmation_email
1	Smith	John	the 02/05/1994	john@mail.com	john@mail.com
2	?	John	the 02/05/1994	john@mail.com	john@mail.com
3	Smith	?	the 02/05/1994	john@mail.com	john@mail.com
4	Smith	John	the 02/05/1994	john@mail.com	john@mail.com
5	Smith	John	the 02/05/1994	johnSmith@mail.com	johnSmith@mail.com

	Test steps		Outcome	Requirement
1	Check the information	The information is correct	Account created	Experience_account_010
2	Check the information	The first name is incorrect	Enter account creation information	Experience_account_011
3	Check the information	The name is incorrect	Enter account creation information	Experience_account_012
4	Check the information	The confirmation email does not match	Enter account creation information	Experience_account_013
5	Check the information	The email already exists in the database	Error account creation	Experience_account_014

Figure 4.2 Example of decision table at node "verification of the requirement" [8]

Practitioners can use decision tables and workflow-based graphs without requiring prior modeling skills. Hence the dashboard gives an overview of the static and behavioral structure of the system and makes it easier to check the requirements coverage. Furthermore, the paper discusses the way MBT can further enhance Acceptance Test-Driven Development (ATDD) [8].

Hametner et al. [9] discuss the need for Test-Driven Development approaches in the automation systems engineering where modeling and testing the real-world systems is complex and challenging. The study [9] proposes a method which identifies a set of UML models that derive test cases systematically. The proposed method is evaluated using an industrial case study i.e. an automated waterworks system for an irrigation system. The system intends to regulate the flow of water in the tank based on sensor data. For modeling the requirements, an open-source graphical tool was used. The requirements are prioritized and used to generate test scenarios and test cases. Different selected models were used from the UML diagram family to conduct the initial feasibility study. The study indicated that models could support both the dynamic as well as static modeling activities. The system requirements can be derived from the static model. In contrast, the interaction diagrams and behavior models facilitate test case generation based on models. However, the size and complexity of the system directly influence the selection of models.

Bünder et al. [12] suggest transforming the low fidelity prototypes into automatically executable Graphical User Interface (GUI) test cases to enhance the efficiency of test case generation. The study [12] reports on the generation of automatically executable GUI test cases by integrating low fidelity prototypes with BDD-like requirements. SLANG [12] is the proposed language that serves the purpose of this study. In order to assess the potential of the proposed method, the study [12] accounts for a controlled experiment "AB/BA crossover" based on industrial experiences at large Swiss banks. In this case study, the proposed method is compared with the JBehave tool. In this experiment, the components of varied sizes were selected from a project. All three components used SLANG for the specification and automation of GUI test cases. The experiment was held out by 17 participants. The results state that SLANG is time efficient as compared to JBehave. On average, the participants took 174 seconds to identify the automated test cases while in JBehave, it took 476 seconds. Consequently, SLANG increases the time efficiency by 63% and makes it cheaper to write GUI test cases.

Luna et al. [13] presented a method-independent methodology that integrates TDD with model-driven approaches in Web Engineering. The study [13] discussed the transformation of tests together with model refactoring. The concept is illustrated using a case study based on a web application "online bookstore." The overall approach involves the structure like TDD, except for the code writing phase. Instead of writing the code manually, the code is generated through models using the Model-Driven Web Engineering (MDWE) i.e. tool WebML's MDD tool and WebRatio (discussed in Section 4.2). All the requirements are modeled using mockups, and user interaction diagrams. Navigation unit tests are written and run against each requirement one by one. Next, instead of writing code, WebML's MDD tool and WebRatio tool are used to upgrade the model and generate a running application capable of creating book lists with links to the relevant pages. Then, test adaptation is made to ensure the validity of the application generated from the models. Lastly, new tests can be written when new functionalities are added. In this way, TDD style is retained while leveraging the benefit of a model-driven approach by working at a higher level of abstraction and using support tools for code generation.

Hayashi et al. [14] proposed a methodology for Test-Driven Development of models (TDDM) [14] that is based on a tool "SMART." SMART is an experimental UML 2.0 modeling tool that guides on developing models based on compiler error messages of test cases. The study [14] elaborates the TDDM methodology by means of a case study. i.e., a room air conditioner with a remote controller that starts with eliciting requirements utilizing user stories. Next, the test-suite table is developed to formalize and instantiate the user stories. Now the test-suite table is translated into test cases by SMART. The test-suites are compiled to detect errors. Besides error detection, SMART not only guides on how to fix the errors but also suggests what should be the next step. However, the user must inspect and guarantee the guidance provided by SMART to develop the models as per requirement. Different studies have used different test generation methodologies for distinct application domains. To provide an overview of the distribution of test generation methods over application domains, we have created bubble graph, as shown in Figure 4.3.

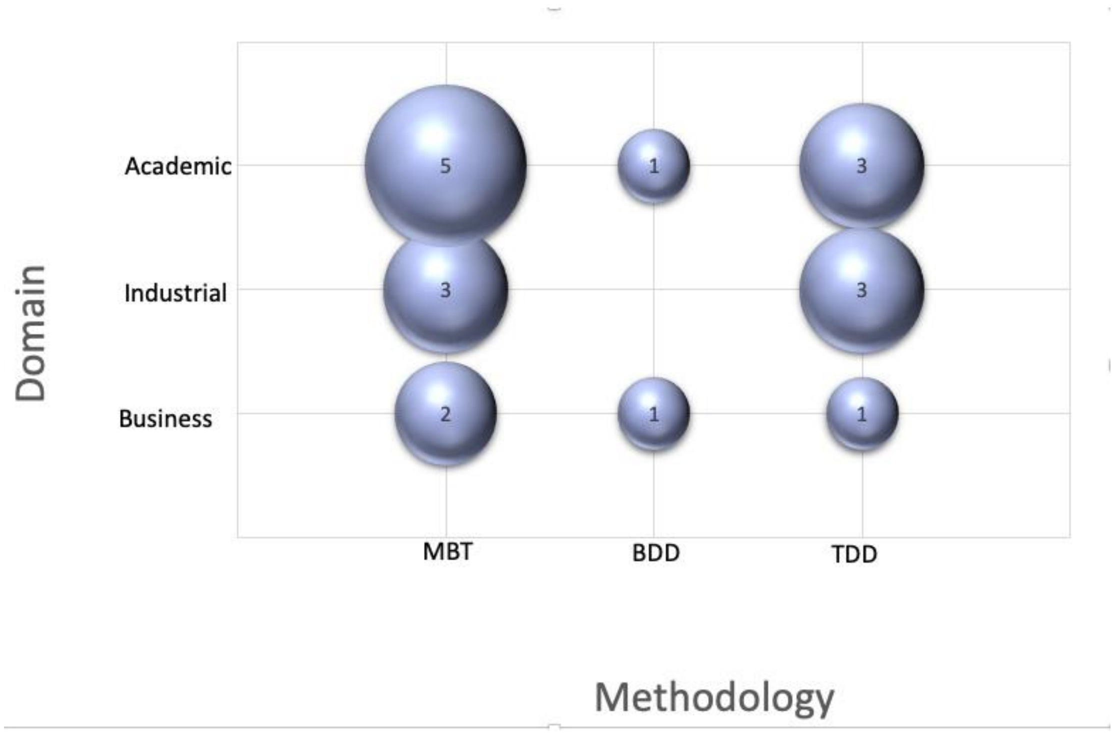


Figure 4.3 Application domain and test generation methods

4.2 Research Question 2:

Are there studies/research papers available that focus on the tools that support TDD/BDD and MBT?

The objective of this research question is to identify the studies that discuss the tools which support TDD/BDD and MBT.

Mou et al. [2] proposed an approach leveraging the benefits of the unification of Model-Based Testing, Test-Driven Development, and Model-Based Requirement Engineering (MBRE) [2]. The proposed method suggested the formalization of requirements into well-defined models to enable automatic tests generation using the MBT approach. The generated tests can be utilized to check the consistency between the system architecture and the system requirements continuously. Deep integration between system architecture and system requirements are required to operationalize the approach proposed in the study [2]. Furthermore, it is required to formalize the system specifications into models and verify the test cases based on system architecture. A model-based framework called "AF3" was proposed and used to inspect the interaction among requirement models, test cases, and system architecture. It can directly assign a formal model to the corresponding requirement and linked with the corresponding part of system architecture. However, the study did not discuss the structure or functionality of the proposed framework.

Li et al. [4] introduced an MBT tool, “Skyfire.” Skyfire is developed based on Structured Test Automation Language framEwork (STALE). STALE is used to read the UML models that help in generating abstract tests. By providing a test automation language, STALE enables its user to create mappings from models to test code [4]. Moreover, STALE can generate both the abstract and concrete test cases. The concrete test cases, using Gherkin language (Cucumber), are generated based on the mappings. The proposed tool was only used to parse the UML models and to generate abstract test cases based on STALE [4]. The generated abstract test cases were also translated to Cucumber test scenarios.

Figure 4.4 provides a graphical representation of the Skyfire architecture. We refer to study [4] for technical details.

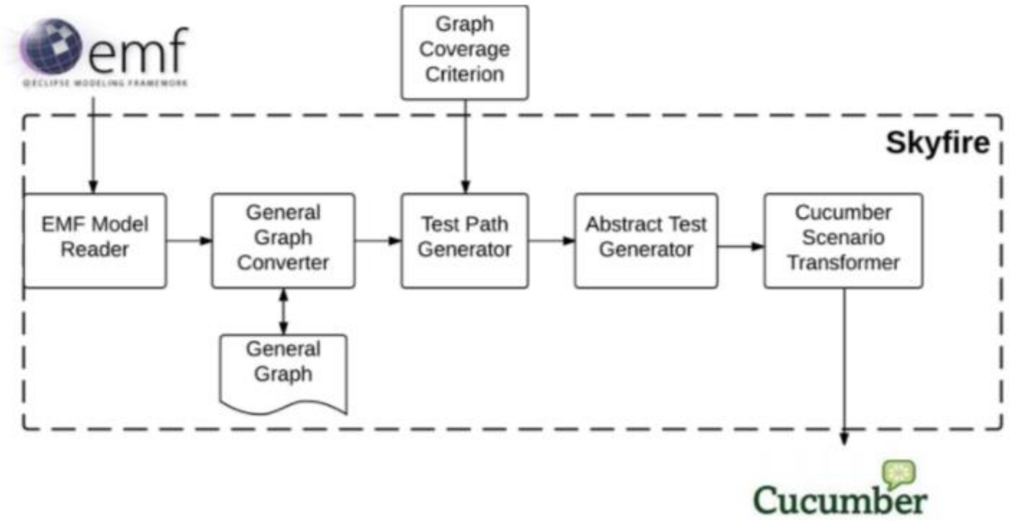


Figure 4.4 Skyfire architecture [4]

Aichernig et al. [6] proposed an approach which generates abstract test cases from formal models. The test cases were verified against the requirements through model checking. After verifying the models, the models were refined and a TDD approach is applied on the refined models. In the TDD approach, the test cases were implemented in a systematic order (the proposed methodology is provided in Section 4.3). To support the proposed approach, the study [6] leverages the combination of CADP toolbox and Ulysses without combining them into a toolset. The CADP toolbox operationalizes the model checking process. CADP provides a model checker to evaluate the labeled transition system. The tool Ulysses generates test cases. Ulysses is an Input-Output Conformance (ioco) checker for the behavioral system that ensures the perseverance of the original attributes of the abstract models. The input-output conformance relation determines which system under test conforms to the system specification [16]. Ulysses takes a two-action system, one is original, and the other one is a mutant of the original. If the test cases generated from Ulysses for two distinct models show distinct behavior, it depicts that there exists a non-conformance between the two models. Ulysses generates test cases that distinguish the mutant model. Ulysses also investigates if the later or formerly generated test cases can kill the mutant. By killing a mutant means distinguishing a mutant. Only if both of the test cases in the directory are unable to kill a new mutant, Ulysses generates a new test case. The new test cases generated by Ulysses are in one of the input formats of CADP, and CADP label them as a transition system. These test cases are text files that define the edges and vertices in a labeled directed graph. However, generating too many test cases could kill the mutant but this could prolong the test case generation time and make it more expensive.

Zhang et al. [7] proposed a method, called TDM (discussed in Section 4.3), which applies the extreme programming Test-Driven pattern to an MDD environment. The TDM process uses Message Sequence Charts (MSCs) for unit test cases and system analysis. An MSC is a system's use case scenario, based on the system requirement specification. In order to implement the suggested approach effectively, the Telelogic TAU SDL Suite tool was selected. The tool fulfills the essential required criteria to implement TDM successfully. The tool "Telelogic TAU SDL Suite" supports Message Sequence Charts at distinct levels. This enables TDM to implement a Test-Driven paradigm at both the system and subsystem levels. The paper [7] does not discuss the structure or workflow of the Telelogic TAU SDL Suite but highlights its basic aftermaths.

Bernard et al. [8] proposes a lightweight MBT tool supported approach that only supports Enterprise IT applications. The motivation is to loosen up the steep learning curve of MBT approaches. The proposed tool graphically explains the workflow of the system that can also be connected with the decision tables. In this way, practitioners can easily work with these graphs and decision tables without requiring any previous modeling skills. To serve the purpose, a lightweight MBT tool called Yest was introduced. Yest is a graph-based tool that is developed to test the progress, behavior, and business rules of the system with a short learning curve. In this way, Yest differs from traditional Model-Based Testing tools. The working pattern of Yest is illustrated through a case study as discussed in Section 4.1. Figure 4.5 shows an example of the use of the Yest modeling element.

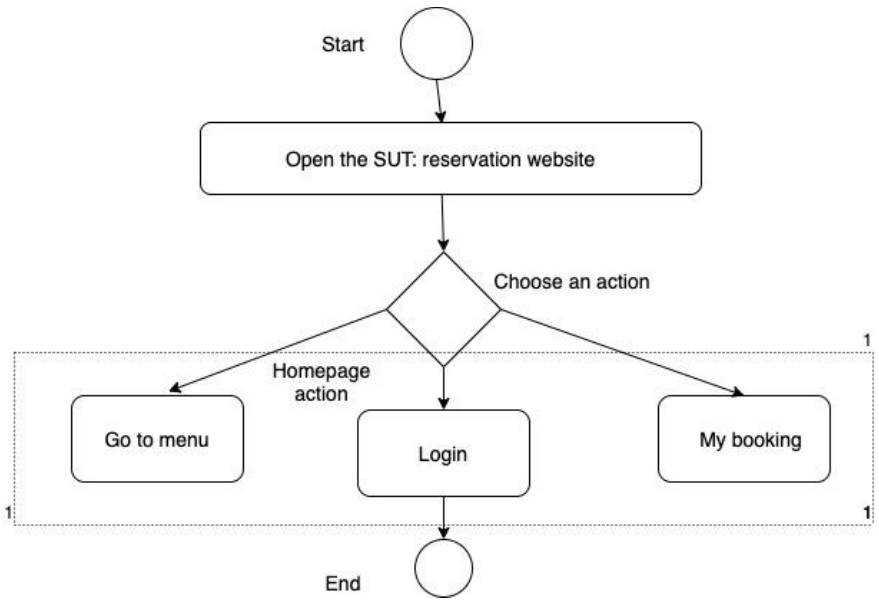
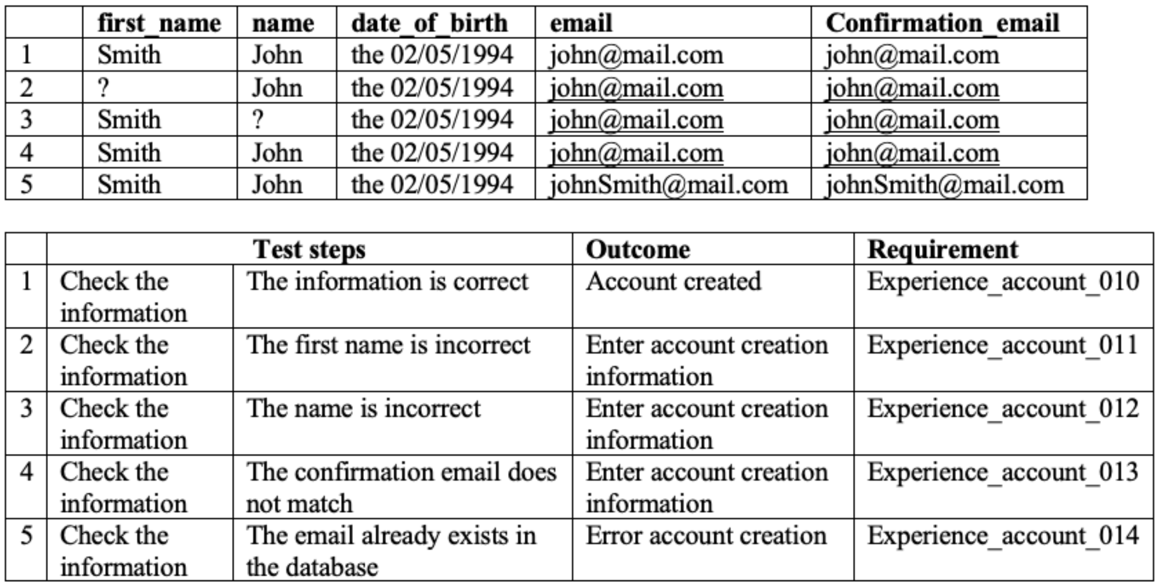


Figure 4.5 Yest modeling element [8]

The tool comprises two nodes, that is inner nodes and external nodes. The external node has two kinds: 1- start point 2- endpoint. The inner nodes are of three types. 1- task (used to represent the actions on the system under test), 2- selection, and 3- sub-process. All the nodes are connected through connectors, describing the workflow of the system. The selection nodes and task nodes are linked with a decision table. These nodes define the test cases and direct the workflow of the abstract test data, test actions, test generation, and the expected outcomes. Figure 4.2 represents an example of the decision table connected with the given example of Yest modeling element, used in the study [8].



	first_name	name	date_of_birth	email	Confirmation_email
1	Smith	John	the 02/05/1994	john@mail.com	john@mail.com
2	?	John	the 02/05/1994	john@mail.com	john@mail.com
3	Smith	?	the 02/05/1994	john@mail.com	john@mail.com
4	Smith	John	the 02/05/1994	john@mail.com	john@mail.com
5	Smith	John	the 02/05/1994	johnSmith@mail.com	johnSmith@mail.com

	Test steps		Outcome	Requirement
1	Check the information	The information is correct	Account created	Experience_account_010
2	Check the information	The first name is incorrect	Enter account creation information	Experience_account_011
3	Check the information	The name is incorrect	Enter account creation information	Experience_account_012
4	Check the information	The confirmation email does not match	Enter account creation information	Experience_account_013
5	Check the information	The email already exists in the database	Error account creation	Experience_account_014

Figure 4.2 Example of decision table at node "verification of the requirement" [8]

The decision table shown in Figure 4.2 is provided by the study [8]. The table shows that there are five sections: Requirements, objectives, test steps, conditions, and Outcomes. The test step defines the order of test action. Test data and outcomes are used to set conditions in order to manage the flow. The decision table helps in the guidance of test case generation and requirement verification.

Sivanandan et al. [10] proposed an approach of integrating Model-Based Testing with Behavior-Driven Development. In this study, MBT is used to investigate a Behavior-Driven test automation framework design and to explore how effective this can be during Agile developments. The motivation is to enable a non-developer to execute the tests without holding technical skills and supports the online testing approach. The designed automation framework was experimented upon the integration of an open-source MBT test automation tool “GraphWalker” with a BDD framework and Robot framework in order to enhance Agile development. The study [10] discussed the basic features of the Robot Framework and Graphwalker. The generic concept is illustrated in figure 4.6.

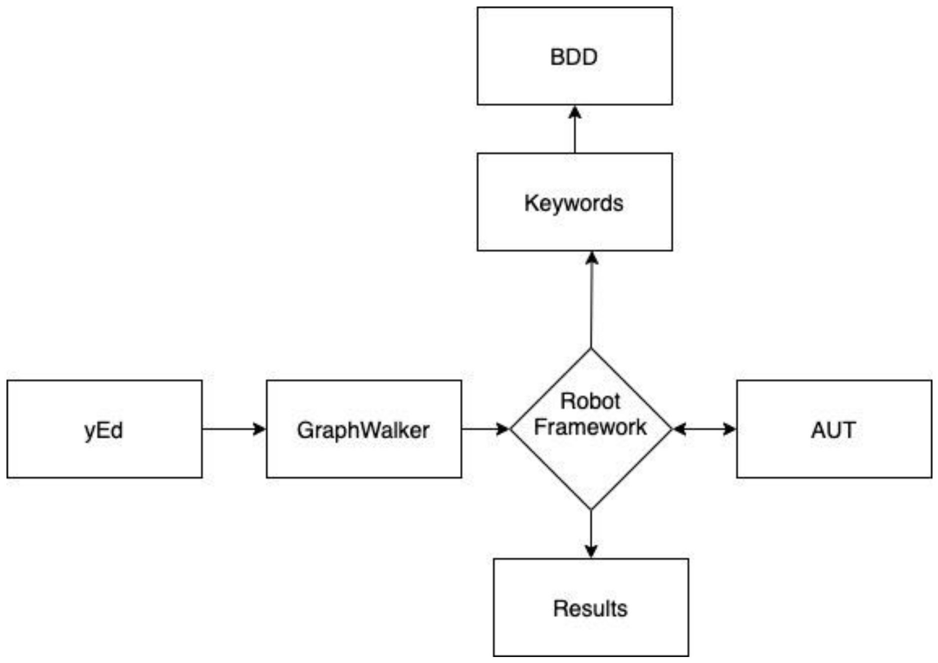


Figure 4.6 Testing Application Under Test (AUT) using BDD technique [10]

The Figure 4.6 provides a graphical representation of testing an AUT using Behavior-Driven Development technique. Models are designed by model designers using yEd and the keywords required for models are built by Automation Engineers. BDD based use cases are written by the business analyst. In order to execute these use cases, the models are triggered. The designed automation framework was experimented upon the integration of the Graphwalker and the Robot Framework to investigate the performance of a web automation library “Selenium 2.0”, along with the Robot Framework and enable the online testing approach [10]. Graphwalker is an MBT tool used to generate graphs by reading models [10]. The tool generates both on-the-fly (online) and off-the-fly (offline) test sequences from Finite state machine (FSM) and Extended FSM. An FSM is a model consisting of vertices interconnected through edges. Robot Framework is a generic test automation framework for ATDD and acceptance testing. It uses a Behavior-Driven and Keyword-Driven based method and has flexible tabular test data syntax. The same syntax can be utilized to create new keywords from existing ones. The test libraries for Robot Framework are implemented either by java or python; these test libraries can extend their testing capabilities. Test suites can be written in normal text, HTML, ReStructuredText (RST) or TabSeparatedValues(TSV) format. The study [10] suggests that this integration can extend the testing capacity of Graphwalker on the Agile front by utilizing a wide range of library support of the Robot Framework. Different libraries from the Robot Framework for several domains enables this integration for Model-Based Testing.

Bünder et al. [12] presented an approach to improve the efficiency of the test case generation process by combining BDD requirements with low-fidelity prototypes. In order to support the objective, the study suggested a specification language called "Slang." Slang is fully capable of generating executable test cases to test graphical user interface. The study [12] investigates the potential of combining BDD- like feature descriptions with low-fidelity prototypes. To quantify the economic advantage of the introduced approach. The work aims to support the scenario definition and feature description by unifying test scripts and requirement definition under a single feature definition directory. The domain-specific language of Slang resembles standard English sentences, which makes it feasible for non-technical domain experts. However, it is more complex and less flexible than natural language as it has a limited number of valid keywords (given, when, then). In order to illustrate the Slang capabilities, the study [12] used a controlled-experiment based case study of a web application (discussed in Section 4.1). Given below in figure 4.7 is the architecture of Slang workbench.

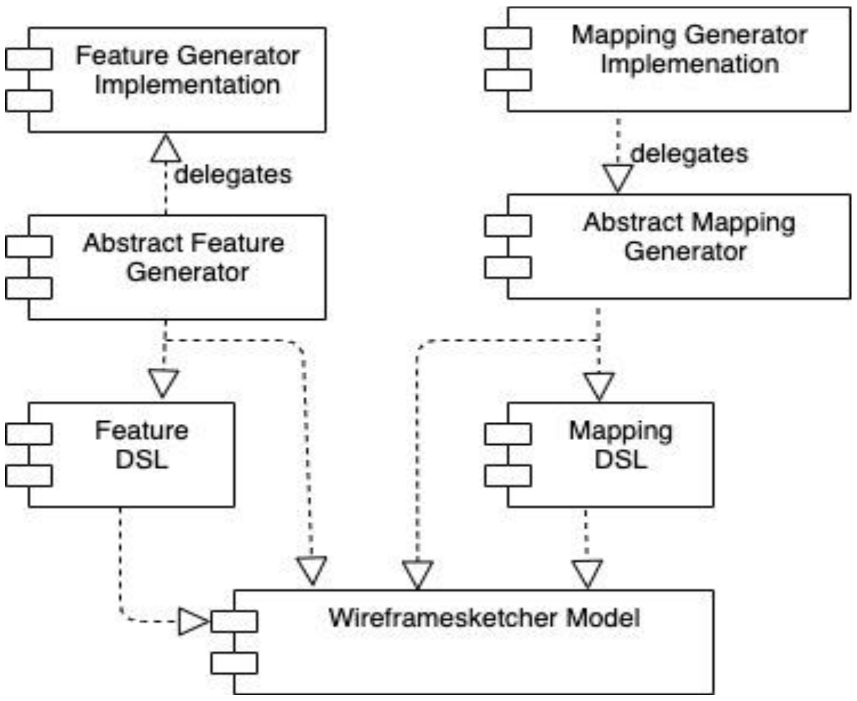

Feature Generator Implementation
Mapping Generator Implemenation
delegates
delegates
Abstract Feature Generator
Abstract Mapping Generator
Feature DSL
Mapping DSL
Wireframesketcher Model

Figure 4.7 Slang architecture [12]

Slang is divided into five components. The feature DSL and the mapping DSL contain tooling implementations and syntax definitions that support framework sensitive proposals and highlights grammar for both the DSL components. The WireFrameSketcher component encapsulates the graphical editor and formal models to store screens and their widgets. GUI models are developed and maintained by WireFrameSketcher. Figure 4.7 illustrates that abstract mapping and abstract feature generators integrate mapping and feature definitions with WireFrameSketcher to generate automatically executable graphical user interface test cases [12]. The study discussed the architecture of Slang in detail and explained through a case study (illustrated in Section 4.1)

An approach for generating GUI test cases was proposed by Luna et al. [13]. The study suggested the integration of MDD and TDD by developing models from test scripts. In order to operationalize the proposed method (discussed in Section 4.3), the WebRatio and the WebML design tools were used. The study does not focus on the working strategy or the architecture of the WebRatio or WebML design tools. However, it discusses the main steps and the feasibility of using these tools as a proof concept. As explained in the study [13], the data models are the main content of WebRatio and WebML, so the procedure begins with identifying entities for interactions and developing data models utilizing User Interaction Diagrams (UIDs). In this phase of the development, the specifications of ER models are supported by WebRatio. Then the navigation sequences are mapped user interface diagrams to a WebML diagram. Finally, WebRatio runs the prototype and generates the application. Now tests can be adapted to check the consistency of models with the corresponding requirements. The study aims to work with models while the generation of code is left to the support tool.

Hayashi et al. [14] proposed an approach of Test-Driven Development of models. The study aims to focus on the construction of models. The approach is based on an experimental UML 2.0 based modeling tool called “SMART.” Hence, the methodology is language independent and easy-going with Agile Model Development. Based on error reports generated by the compiler of test cases (the study did not discuss how the test cases are compiled and error reports are generated), SMART guides the user about the next step in building models.
Furthermore, the proposed methodology generates fakes. A fake can provide the correct value for a given test case. The study provided a complete architecture of a SMART model and illustrated through an example. A model generated by SMART comprises its name, properties, functions, events, constructors, and encapsulates the following five sections. 1- Use case, 2- state machine, 3- User interface, 4- Composite structure, and 5- Sequence diagrams. SMART's own language called “Smart Action Language (SAL)” compiles all the programs into actions and actions into execution. Smart Testing Tool (STT) is used to test TDDM [15]. A SMART model also contains various documents. These documents and their interconnections are managed by a traceability tool called “Smart Traceability Web Tool (STWT)”. However, Test-Driven behavioral guidance is a unique feature of SMART. The highlights of the proposed method is discussed in Section 4.3.

Lazar et al. [15] proposed a methodology of integrating Model-Driven Development (MDD) and BDD to generate test methods. A UML profile is defined to enable the creation of an executable foundational Unified Modeling Language (fUML). The test methods are generated using executable fUML, and BDD requirements can be executed automatically. Further, the study introduced a tool “bUML” to support all the BDD activities and fUML models. The novel feature of bUML tool is to manage the user story and scenario status automatically. The study did not discuss the basic architecture of the tool. However, it explained its usage in the proposed methodology (discussed in Section 4.3).

4.3 Research Question 3:

What methodologies/solutions are proposed to integrate MBT and TDD/BDD?

This research question intends to explore studies that discuss the practices that combine MBT and TDD/BDD. We aim at investigating the methodologies that integrate Model-Based Testing with Test-Driven Development or Model-Based Testing with Behavior-Driven Development, so that the best of both the practices can be merged and used for testing. The purpose of this question is not to compare the integrity or validity of different approaches but only to state the proposed method as presented in that study. Wieczorek et al. [1] proposed a method that uses Model-Based Testing on the system level and the Test-Driven Development on component level. In this approach, MBT describes the design of test models as well as enables the generation of integration tests. The method is specially designed for the development of business applications based on Service-Oriented Architectures (SOA). Figure 4.8 illustrates the overview of the proposed method.

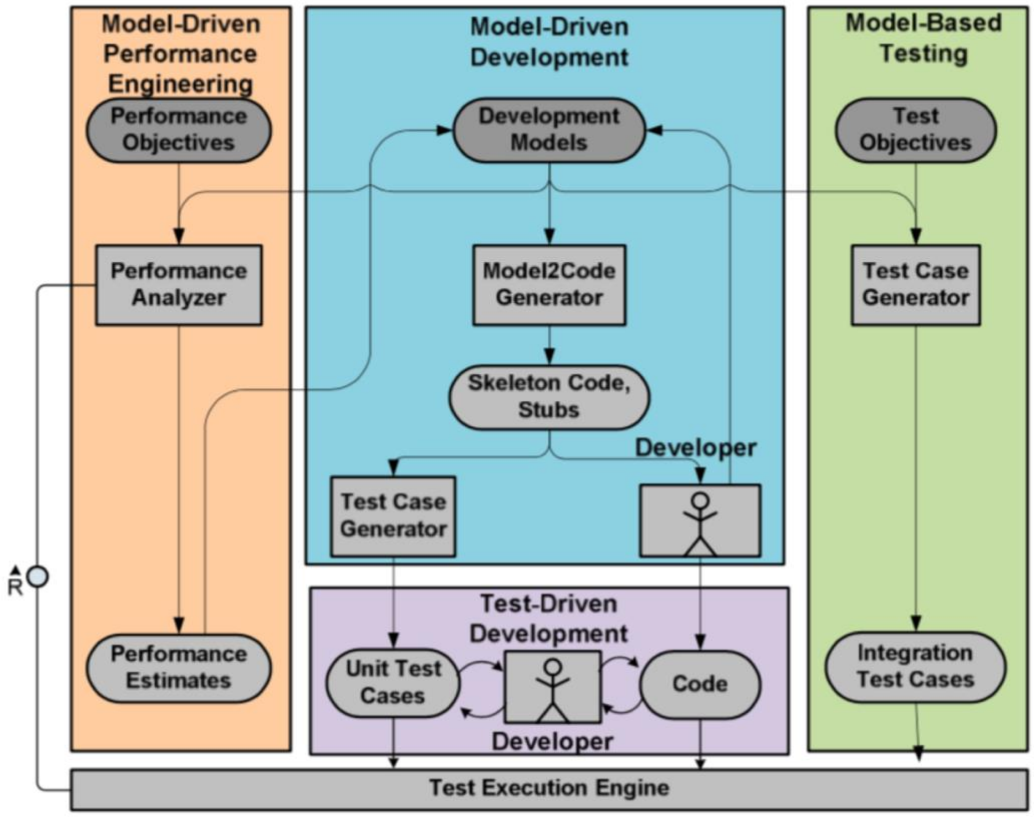


Figure 4.8 Overview of the method proposed by Wieczorek et al. [1]

Figure 4.8 shows that the primary purpose of this approach is to use MBT for the inter-component service integration and TDD for the development of business components. The methodology starts with modeling the business components based on user requirements. Two types of models are created based on requirement specifications. (i) structural model: to identify and connect the business components, (ii) behavioral models: to identify the business flow. Figure 4.8 shows that the Model2Code generator takes development models as data input to generate stubs and skeleton automatically. Developers refine the models and then generate the code using Model2Code generator. Next, performance analysis takes place in parallel to the MDD process to envisage the performance-related problems at earlier stages. The performance analyzer, with the help of performance parameters, predicts the performance-related issues.

After both the structural and the behavioral models are developed in compliance with the performance requirements, MDD takes the lead to the next step to refine the local behavior of the models and derive the code automatically. At the abstraction level, a TDD approach is implemented. The developers create the unit tests for a given function, execute and run the tests, and refactor their own code after the test success. These steps are repeated for every added functionality. Next, Integration testing is operationalized after the system components are developed. As shown in figure 4.8, the study suggests using Model-Based Testing for integration testing. The study [1] explained the outcomes of using different approaches such as MBT, MDD, TDD, and MDPE at distinct phases. However, the core idea of the paper is to describe the design of the test models only.

Mou et al. [2] proposed an idea to integrate the approaches of MBT, TDD, and model-based requirement engineering (MbRe), systemically. The proposed method was named as" Model-Based Test-Driven Development (MBTDD)." Utilizing formal refinement strategies from MbRe, both the architecture and requirement models can be linked together by test cases that are generated automatically. These test cases also enable continuous checking of system architecture against the system requirements. The study [2] explained the basic overview of the proposed methodology is shown in Figure 4.9.

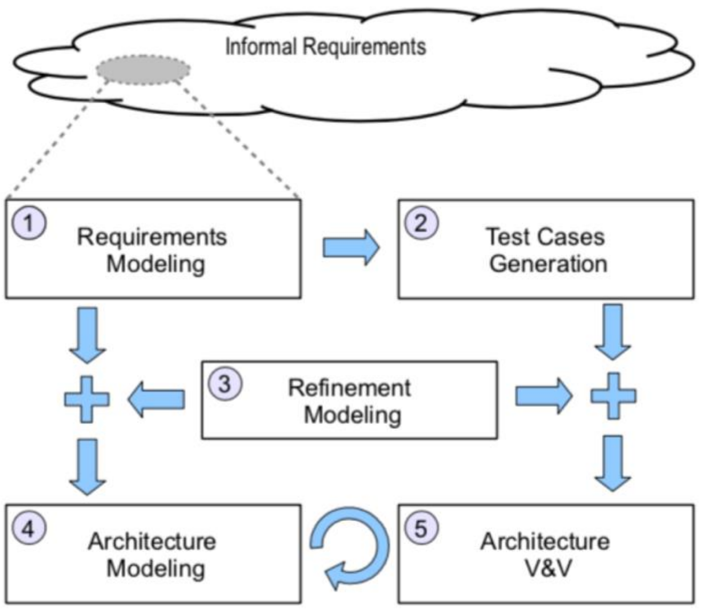


Figure 4.9 overview of MBTDD [3]

The proposed method comprises the following five main activities.

1. The process starts with modeling the requirements.
2. Based on these models, test cases are generated as per MBT practices.

3. Refinement modeling is achieved by defining a concrete strategy to implement the requirement models into a concrete system architecture.
4. Architecture modeling is carried out based on the refinement models and requirement models. The refinement models enable component architecture to implement the requirement model. With each requirement model implementation, the refinement model enables the test cases to be executed on the architecture.
5. The automatically generated test cases can verify the component architecture at any stage.

In this way, the study [2] suggested an approach of enabling Model-Based Agile development with rapid iteration. A subset of system requirements is formalized into models. These formalized requirements are used to generate tests. Then, the requirements are implemented, and their consistency is continuously verified against the requirement specification using automatically generated tests. Further, Mou et al. [2] discussed a challenge in this approach i.e. trading off the novel property of TDD. The novel feature of TDD is a light-weightedness. By defining models at the beginning of the procedure requires more effort than merely writing test cases. Hence, the light-weighted property of Test-Driven Development is partly lost.

Ussami et al. [3] proposed an approach that aims to enable the reuse of test cases by integrating Model-Based Test-Driven Development (MBTDD) with the concept of Model-Based regression testing. In each MBTDD iteration, test models are modified to define a new set of behaviors. This could lead to the evolution of test models as well as the entire system. Consequently, two main problems occur. (i) reusing the test cases generated previously, (ii) identifying the test artifacts needed to develop a new feature. The study [3] intended to deal with these problems. According to the proposed methodology, the notion of delta-oriented model-based Software Product Line (SPL), regression testing can be used to develop a new system by reusing the test models and the test cases. The deltas determine the disparities between the product variants. The deltas also specify both the existing valid test cases for a variant, and the valid one to be created. The system behavior is represented by employing Finite State Machine (FSM) and is used to derive test cases. The delta-modeling concept is utilized to specify the variation among the product variants, and the amendments made in the test models are used to validate the former test case. The former test cases are evaluated to identify the valid test models and can be reused. Then, the new test models are used to create and update the new test cases. Hence, to develop the new features, these newly evolved test cases are used, and the reusable test cases are utilized as regression tests. The reusability of test artifacts occurs with the evolution of the system.

Li et al. [4] introduced a Model-Based Testing tool integrated with Cucumber. Cucumber is a Behavior-Driven Development tool that has been used widely in the software industry for software testing purposes.

The BDD test scenarios are handwritten. Cucumber provides clarity about coverage and mapping mechanisms for test scenarios to execute tests. However, the test scenarios are created manually and are weak since they are generated manually. Skyfire enables the automatic generation of Cucumber test scenarios. The procedure starts by reading behavioral UML diagrams. Then, to satisfy graph coverage criteria, Skyfire specifies all the required elements that exist in the diagram to develop effective test scenarios. Next, the tests are converted into Cucumber scenarios. The practitioners develop Cucumber mappings for the Cucumber scenarios. Besides creating useful tests, Skyfire is compatible with both the Continuous Integration (CI) and Agile Development (AD). With Skyfire, the testers are required to generate test scenarios based on UML diagrams. Furthermore, the mapping mechanism of Cucumber can be leveraged to define the system's salient features by allowing testers to only focus on model development. The paper discusses the structure and implementation of the tool are briefly described in section 4.2.

Munck et al. [5] focused on the modeling of the system architecture and leveraged the benefit of TDD and Model-Based System Engineering (MBSE) together. The study [5] explained that the utilization of Model-Based System Engineering techniques such as hierarchies, abstractions, the customized outlook for an individual stakeholder, and simulations have made the verification and validation of complex systems possible. However, with the evolution of the system, such techniques will result in more complex models to be comprehended by human minds. That's where the need for modeling mechanisms is realized, that can replace the manual model checking with an automated analysis mechanism. The proposed approach aims to solve this problem by enabling the system engineers to handle the complex model systems as well as to enhance their quality. The study focused on the utilization of formal model and statistical model checking with the formalism of timed automata. The method used formal verification instead of applying simulation for the behavior verification of the system. to support the methodology, an existing model checking tool UPPAAL was used for modeling the system architecture and the system behavior (discussed in Section 4.2). The TD-MBSE framework enables engineers to apply a Test-Driven approach for modeling the uses-cases, scenarios, requirements, architecture, and behavior of the system. However, in this study, the focus is on system architecture. Figure 4.10 gives an overview of TD-MBSE of modeling of architecture and behavior.

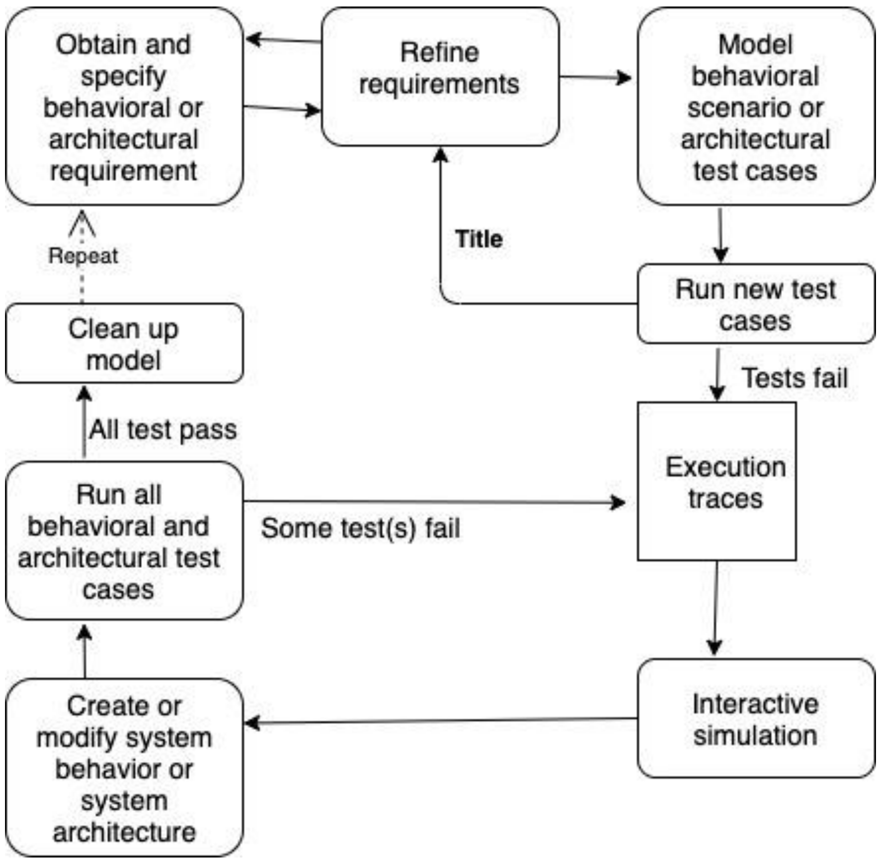

Obtain and specify behavioral or architectural requirement
Refine requirements
Model behavioral scenario or architectural test cases
Repeat
Title
Run new test cases
Clean up model
Tests fail
All test pass
Execution traces
Run all behavioral and architectural test cases
Some test(s) fail
Create or modify system behavior or system architecture
Interactive simulation

Figure 4.10 TD-MBSE of modeling of architecture and behavior [5]

The proposed approach is TDD based, where the test cases written for each test scenario must fail initially. Then, the testers write the code to pass all the tests. The process continues until all the requirements are implemented successfully. The proposed approach is achieved by expanding every step of TDD. The first step of TDD is to write tests. The proposed method modifies it by specifying behavioral and architectural requirements and refine in a way that they can be modeled using formal methods. In the case of modeling behavioral specifications, modeling elements like activity, sequence, or stat machine are helpful. If the behavioral requirements contain time constraints, then other techniques such as timed automaton formalism can be utilized. However, for architectural requirements, different approaches can be used. For example, system properties can represent quality features of architectural requirements. The architectural requirements are expressed as invariants and are checked continuously by a model verifier. The model verifier used in this paper is UPPAAL. The next step of TDD of running new test cases is modified for TD-MBSE by using execution traces (generated for failed test cases) and interactive simulations to catch and handle both the behavioral and architectural errors. The step of modifying the system model in TD-MBSE replaces the TDD step of writing code. The TDD step of running the test case is similar to the one in TD-MBSE. Except in TD-MBSE, to run a new test case, execution traces in conjunction with interactive simulation are used. Finally, the last step in both approaches is identical. In TDD, the code is refactored while in TD-MBSE, the models are restructured. One of the significant phases in system engineering, not observed in TDD, is Design Space Exploration (DSE) [5]. In DSE, to get the most optimized solutions, the models are analyzed to find the best possible alternate designs. Therefore, DSE is applied to the entire process. In the case of behavioral variants, the queries from the basic design are employed to evaluate the performance. In the case of structural variants, first, the property simulation is used to estimate the values of the system properties, then formal model checking is applied for performance verification. However, in both cases, to verify each design variant, each query needs to be redefined or added. Figure 4.11 illustrates the approach of Test-Driven Design Space Exploration (TD-DSE).

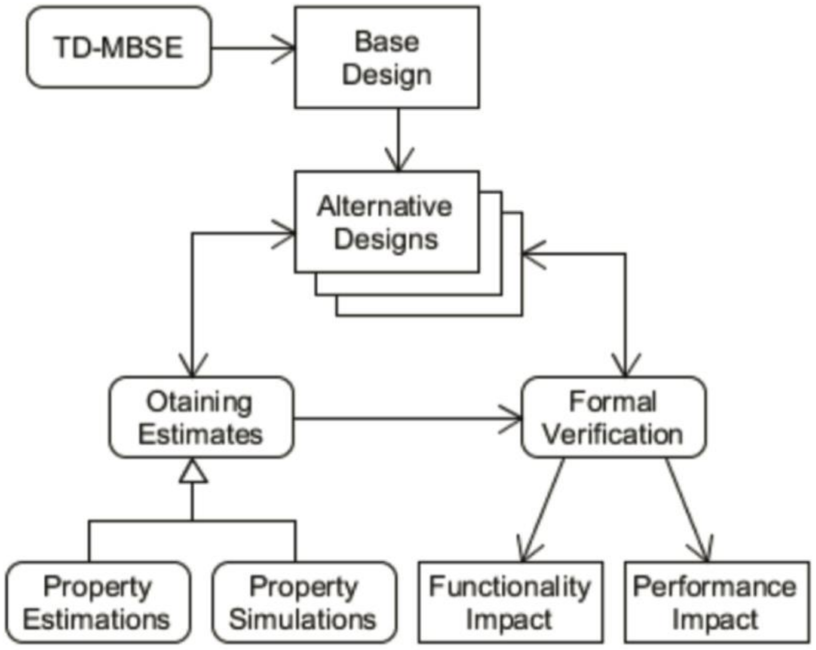


Figure 4.11 Test-Driven Design Space Exploration (TD-DSE) [5]

Aichernig et al. [6] proposed a methodology to integrate MBT, model checking, formal models, TDD, and model refinement into an existing Agile SDLC. In this approach, abstract test cases are generated using formal models and verified against system requirements through model checking. The study [6] aims to verify the tests and not the models because of two reasons. (i) tests are necessary for safety-certification processes not the models. (ii) not all the modeling tools provide model checkers. The novel features of this methodology are model checking of the generated test cases and the integration of TDD and formal techniques. Furthermore, with the evolution of the system, the system functionalities and the abstract test cases are evolved. This can lead to updating hundreds of test cases manually. In order to avoid manual editing, the study proposes a concept of automatic generation of test cases from the updated models. That is where MBT adds to the process. The test generation takes place through mutation-based testing and is evaluated at the implementation level. In this study [6], the proposed methodology is explained through an industrial case illustrated in Section 4.1. After refining the models, a TDD approach is applied to implement the test cases in a systematic manner. The tools "CADP" and "Ulysses" are used to automate the whole process. The CADP toolbox operationalizes model checking while the tool "Ulysses" generates test cases (discussed in Section 4.2). In the initial phase of TDD of the proposed method, the developers fix the testing interfaces of the SUT and determine the functionalities related to the SUT or the environment. Next, the testing interface of the SUT needs to be fixed at the implementation level. Additionally, a test driver must provide an interface for concrete observation and run the tests. The next phase is an iterative phase, where partial model generation takes place. Partially generated models define the critical features of the SUT that should be implemented first. Then the abstract test cases are generated by formal models and verified against system requirements by model checking. The generated test cases are implemented in a TDD manner and refactored if required. After the completion of a cycle, another set of requirements is refined and restart the cycle. The focus can be shifted between TDD and model refinement.

Zhang et al. [7] proposed an approach of Test-Driven Modeling (TDM). TDM uses simulations of message sequence charts for automated testing and executable models as active system design documents. These MSCs can be used for unit test cases, document designing, or analysis of the generated system. The study is based on an industrial case (discussed in Section 4.1). The results obtained from the experiment showed that TDM enhances the quality and productivity of massive projects and can detect high numbers of code errors. The study [7] also outlined the basic workflow of TDM as illustrated in Figure 4.12 below.

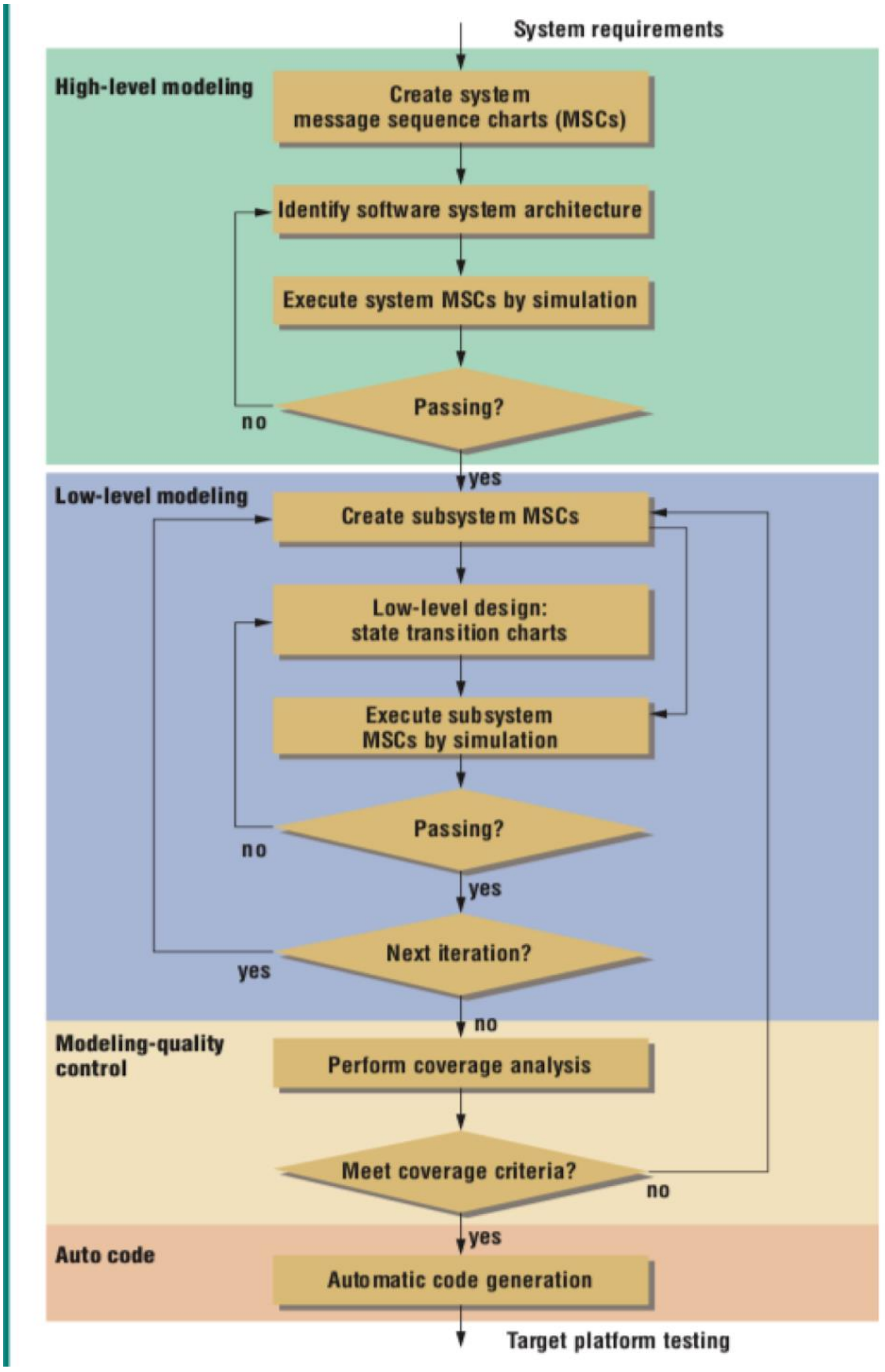

System requirements
High-level modeling
Create system message sequence charts (MSCs)
Identify software system architecture
Execute system MSCs by simulation
Passing?
no
yes
Low-level modeling
Create subsystem MSCs
Low-level design: state transition charts
Execute subsystem MSCs by simulation
Passing?
no
yes
Next iteration?
yes
no
Modeling-quality control
Perform coverage analysis
Meet coverage criteria?
no
yes
Auto code
Automatic code generation
Target platform testing

Figure 4.12 Workflow of TDM [7]

The primary concept of TDM is to implement eXtreme Programming Test-Driven (XP-TD) approach to a Model-Driven Development process [7]. The DtM method has six basic steps. However, the study briefly discussed all the phases, except system requirement specification and target platform testing phase.

1. **System Requirement Specification (SRS) High-level modeling (HLM):** HLM intends to specify the MSCs from the SRS and identify the system architecture. An MSC is a system's use case scenario. In this way, HLM is Test-Driven as the MSCs are created first, and then the system architecture is defined against the generated MSCs. Furthermore, simulations are used to generate automated MSCs to verify the consistency between system architecture and MSCs. Finally, these MSCs are refined and executed for unit testing. The study did not discuss the refinement of MSC.
2. **Low-level modeling (LLM):** LLM intends to identify the static and dynamic properties of the subsystem. The subsystem MSCs define the dynamic behavior of the subsystem. Like HLM, LLM is also Test-Driven, as the subsystem MSCs are generated first. The subsystem's state-transition charts are defined based on subsystem MSCs. Next, the subsystem MSCs are refined using signal data and executed using simulations. Finally, the consistency between the state-transition charts and the subsystem MSCs are verified against the subsystem requirement specification.
3. **Modeling-Quality control:** In TDM, the message sequence charts are treated as test cases. An MSC can include other MSC, so a hierarchy of MSC can be built-in TDM. Such a set of MSCs can be executed together by a modeling tool as a test suite and are known as modules. The modeling tool Telelogic TAU SDL Suite is used in the process (as described in Section 4.2). After the completion of an MSC execution, Telelogic TAU SDL Suite generates a report to determine which MSCs have failed or passed the test. The modeling tool also generates coverage analysis trees utilizing executed MSCs and indicates which state-transition or design elements have been covered.
4. **Automatic code generation:** Similar to MDD**,** TDM is also model-based, and the code is generated by the modeling tool automatically.
5. **And target platform testing**

According to the study [7], TDM is productive and can enhance the quality level through automatic code generation, simulation, and coverage analysis.

Bernard et al. [8] presented a light-weighted MBT graphical tool to decrease the time and effort required to learn the testing-skills to work with MBT tools. Section 4.2 discusses the proposed tool, called "Yest." Furthermore, the paper also discussed the strategy through which Yest could work in an Agile way, for example, with Acceptance Test-Driven Development (ATDD). In the industry, the MBT adoption is slow because of its great learning curve [8]. At the same time, Agile development trends demand faster and reliable testing methods. The study proposed leveraging the benefits of both MBT and Agile approaches by re-designing MBT tools and approaches. It may help practitioners or researchers to amalgamate both methods and to cope with the increasing complexity of systems. A workflow-based graphical design supports an effective collaboration between all the stakeholders regardless of their knowledge background. Such designs also help in better understanding the requirements and their implementation. Besides proposing a tool, the study [8] did not present any full-fledged procedure to integrate both the approaches; however, it drew attention to a few areas that can serve the purpose. In order to support agile methods, model-based solutions need to fulfill many requirements, such as a short learning curve and easy adoption of MBT tools by functional testers [8]. Moreover, the MBT approaches should be able to model all types of applications regardless of the sizes and without requiring full modeling of the system. The MBT methods should also be able to support both the automated and manual test execution.

Hametner et al. [9] proposed a methodology that generates test cases based on models and Test-First Development (TFD) concepts. The proposed method is known as an adapted TFD process. The method discussed the need for Test-Driven Development approaches in automation systems engineering. The adapted TFD process is based on a set of UML models that derives the test cases systematically. TFD has the following four fundamental steps: 1- test definition. 2- test implementation (here the tests must fail). 3- function implementation and test (here all the functionalities related to test cases are implemented successfully), and 4- code refactorization. After refactoring the code, a new set of requirements is selected and the process is repeated until all the requirements are implemented. Based on the models, test case generation requires two types of modeling notations. 1- models for static design 2- models for collaboration. With the help of these collaborative diagrams, test cases can directly be derived from the models and enable the automated test-case generation process. Based on UML family diagrams, the automated test-case generation process comprises the following eleven steps.

- Requirement specification
- requirement prioritization and test-case derivation
- use-case scenario derivation using UML models.
- risk evaluation
- interface and components specification for interaction among components using UML component diagrams
- presenting models of the physical layout of the system and collaboration paths between the components using deployment diagrams

- presenting static designs of each component and their inter-relations. Class diagrams help in modeling the components and systems at the implementation level.
- Presenting states and inter-transitions using state charts. Based on risk, state charts can be used at any level e.g., at the system level for behavioral representation and at the implementation level for each component's behavioral representation.
- Representing time-based events using Sequence charts to illustrate the inter-communication and the triggers initiating these communications among objects
- Representing the behavioral workflow of the system as well as an existing decision path, using Activity diagrams.
- Represent the system's behavior and events based on time and duration constraints using Timing diagrams.

In this way, the proposed method introduced an adapted TDD approach in automation system engineering, where test cases can be generated using models and modeling notations.

Sivanandan et al. [10] presented a procedure to integrate an MBT tool "Graph-walker" with "Robot Framework." The objective of this integration is to identify the design of a BDD-based test automation framework using Model-Based Testing as well as their effective usage in Agile development trends. Further, the study [10] aims at exploring the effectiveness of Selenium 2.0 (a web automation library) when used with Robot Framework to enable the online testing approach. This can provide a non-developer with an opportunity to create and execute test automation scripts. The paper did not describe the methodology as a general. However, the method was illustrated and experimented on the integration of Graph-walker and Robot Framework. as discussed in section 4.2. We refer to the study [10] for the technical details of Graphwalker, Behavior-Driven automation framework, and Robot Framework.

The method proposed by Bünder et al. [12] transforms requirement descriptions from Behavior-Driven Development into automatically executable graphical user interface test cases in the form of wireframes and prototypes. In BDD, a ubiquitous language is used to describe the user requirements. These requirements are mapped into the test methods automatically using a tool support. However, the available BBD tools are not integrated with GUI description, so the test methods are required to be implemented manually, which makes it error-prone. The study [12] mitigated this issue and presented a Model-Driven method which extends BDD approach by using low-fidelity prototypes. The proposed method transforms requirement descriptions from Behavior-Driven Development into automatically executable graphical user interface test cases in the form of wireframes and prototypes. A Domain-Specific Language (DSL) is used to define the required BDD features. The study introduced a specification language "SLANG" to operationalize the proposed process. SLANG combines BBD-like requirement specifications with wireframes to enhance the efficiency of the test generation process by providing automatically generated executable GUI test cases. The presented method is flexible and easy to understand as the generated test cases are presented in a human-readable format. However, it also limits the number of authorized keywords, so it is more complex and less flexible than natural language. The study did not elaborate on the proposed methodology in general; however, it discussed the architecture of SLANG (presented in Section 4.2). Further, the usage of SLANG is illustrated through a controlled-experiment based case study and the results obtained from that experiment are shown in Section 4.1.

Luna et al. [13] introduced a method-independent approach integrating Agile trends and Model-Driven Development, and the process combines Model-Driven Web Engineering (MDWE) and TDD approaches. The proposed method has the same structure as TDD except instead of writing code manually; models are used to generate it automatically using MDWE tools. Furthermore, prior to system development, system behavior is described in terms of tests that can be used to define the application models. The study has explained the steps of the proposed methodology through a case study (discussed in Section 4.1) and the tools WebML and WEB Ratio (discussed in Section 4.2). Figure 4.13 represents the proposed methodology.

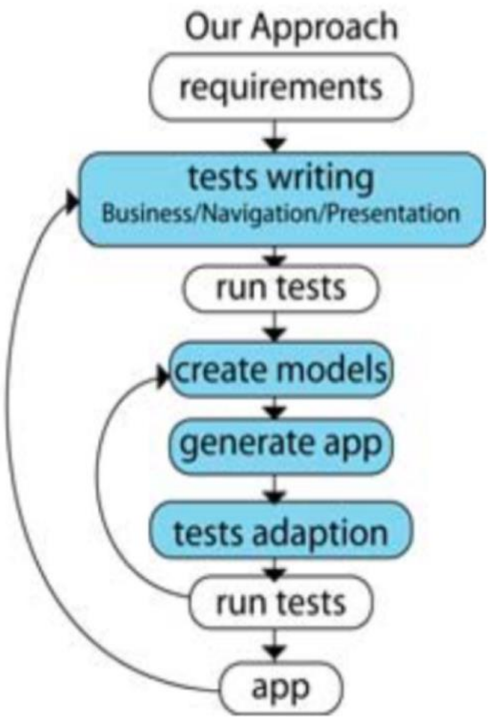


Figure 4.13 Proposed method by Luna et al. [13]

The process starts by gathering requirements and modeling them in terms of use cases, presentation mockups, and User Interaction Diagrams (UIDs). Based on a use case, an interaction test is derived against the corresponding presentation mockup that specifies the user interface interaction and navigation even before the development. Next, the model development and the code generation take place to get a running prototype of the system. Here, the proposed method differs from the Test-Driven approach. This phase uses the MDWE approach, where MDWE tools generate the code instead of writing it manually. Furthermore, tests are used to verify the implementation and are adapted accordingly and are called test adaptation. In the case of test failure, the models can be tweaked to regenerate the application and repeat the process until all the tests pass. Once the iteration for a use case is completed, a new use case/requirement can be added. This process is repeated until all the requirements are implemented. The main focus of this paper is the development cycle and not the tests itself. In order to enhance and validate the behavioral and navigational interface of an application, the study suggested using Black-box interaction tests.

Hayashi et al. [14] proposed an approach of Test-Driven Development of models, also called TDDM. The methodology is operationalized with the help of a supporting UML modeling tool called “SMART” that uses its own language called SAL (discussed in Section 4.2). The study explained the TDDM using a case study of a distributed system (mentioned in Section 4.1). The steps of the proposed approach are briefly provided below.

- The methodology starts with eliciting system requirements in the form of use cases or user-stories.
- These use cases are formalized into a testsuite table or scenario instances.
- The tool SMART is used to translate the testsuite table into the test cases.
- Next, the test cases are compiled. Here, some error reports are generated since no models are built yet (the study [14] did not discuss how the test-suites are compiled).
- Based on the compilation error reports, the introduced tool "SMART" provides suggestions for the next step. For example, at this point SMART will prompt a suggestion to declare a model with model elements. However, the user must guarantee and inspect these suggestions to make the models.

SMART is based on experimental UML 2.0 that provides Test-Driven behavioral guidance. It guides on deriving models based on the compiler errors from test cases. These compiler errors from test cases help in determining the behavioral properties of the system with which codes can be fixed. The SMART generates a fake. A fake returns the accurate value for a given test case.

Lazar et al. [15] proposed an approach in which a UML profile is defined to enable the developers to create executable Foundational UML (fUML) models using a Behavior-Driven Development approach. Foundational UML defines a “basic virtual machine for the UML, and the specific abstraction supported thereon, enabling the transformation of compliant models into different executable forms [15]”. fUML standard can create executable UML models by providing a simplified subset of UML abstract syntax. The executable UML refers to the semantic execution of action components required to complete the computations [15]. The study [15] focused on the behavioral and structural constructs specified by the fUML specifications. The fUML structural construct comprises classes, packages, properties, functions, and relations, while the fUML behavioral construct comprises activities. This shows that stories are modeled as classes, while scenarios are modeled as activities based on classes. In order to facilitate the user to generate executable scenarios, a BDD library is defined containing BDD activities. The proposed methodology starts with the business analysts eliciting the requirement and creating user stories and user scenarios. These scenarios can be described in Gherkin style and can be applied to any fUML action. Next, the scenarios are implemented by the developers using a BDD model library containing fUML activities. We refer to the study [15] for the technical details of the BDD model library and the example implementing the scenarios. Next, the study introduced a tool "bUML " to enable the developer to create fUML models based on the defined UML profile and BDD library. The bUML tool supports all the BDD activities and helps in updating the system status automatically after each scenario execution. For a fast and a simple development of executable scenarios, a concrete syntax is introduced with which the user can build fUML Models through textual editors.

Farago ́et al in [16] investigated how to integrate the strengths of agile and MBT methods i.e., validation from Agile Development (AD) and verification from MBT approaches. The study theoretically discussed that both the methods could benefit from each other. Besides, the strength of AD i.e., validation, there are shortfalls of AD in the testing world such as insufficient test coverage, less flexible manually written test cases, high maintenance, and inability to backtrack failed test cases to the requirement definition. To overcome these deficits, Formal methods can be utilized. Model-Based Testing is a light-weighted formal method for the test generation from the requirement definition. The study used Model-Based Testing for conformance testing. Conformance testing deals with the automated conformance checking between SUT and requirement specifications. Integrating AD and MBT can bring changes to the Test-First Development (TFD) phase in AD. In TFD, the developers use shorter iterations to complete small tasks, and the design is refined after the test cases are specified, and then the features are implemented. In this phase, MBT is utilized for regression testing and metrics such as code coverage and requirement specification. Therefore, instead of writing tests, specifications are written, and the test-first development transforms into specification-first development. MBT tools are used to read these specifications and can generate automatically executable tests This consequently provide sufficient code coverage, minimal maintenance, and easy backtracking to specifications from failed test cases. Similarly, AD can aid MBT by reducing its rigidness towards changed requirements. It can be achieved by applying the iterative and incremental style of AD to MBT at the specification level. The study suggested that the test generation process must be able to manage and refine the models iteratively. The study generally discussed the integration of MBT and agile development but did not provide any concrete methodology or introduced any testing tool.

The results show that the majority of research works focus on the process of test-case generation. It can be a motivation for more research in the future. Figure 4.14 shows the research focus map for the last eighteen years.

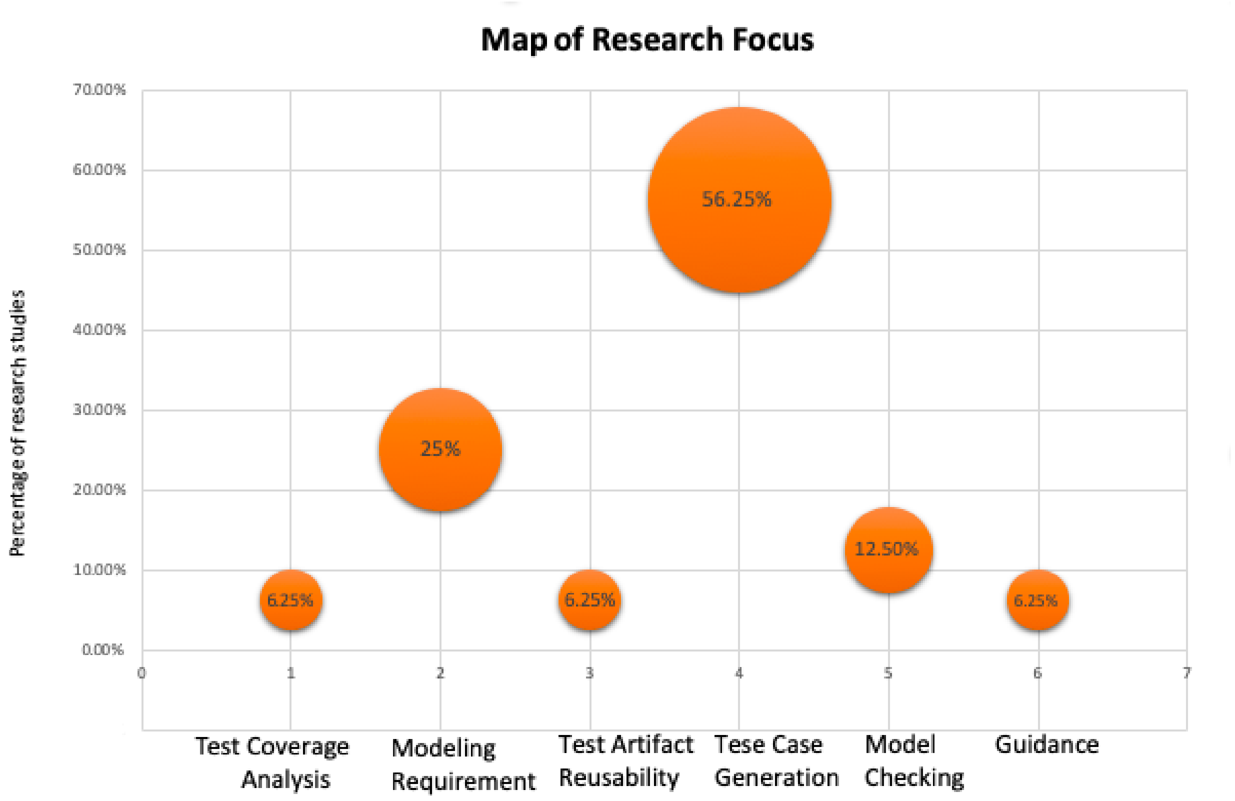


Figure 4.14 Research focus

5. THREATS TO VALIDITY

While writing this SMS, the research work had few threats to validity. One of the key issues concerns the selection of the database for this SMS. The search was limited to six electronic databases only. i.e., IEEE,

ScienceDirect, Springer, ACM, Web of sciences, Google Scholar, and Scopus. Moreover, we used a single search string using the same limited keywords for every database. Thus, it might miss some or many relevant studies. However, after full-text reading and during applying the snowballing procedure, we found six more papers that fulfilled the inclusion criteria. It showed that not many papers pass the specified inclusion and exclusion criteria, which gives an assurance that not many articles are missed.
Another threat to validity could be the inclusion and exclusion criteria. According to the criterion, papers should explicitly discuss TDD/BDD with Model-Based Testing approaches. Many studies are available that generally discuss the integration of MBT and Agile methods, and not the TDD/BDD. Such studies might have some information that could be beneficial for this SMS and serve the same purpose. However, the objective of this SMS is to investigate the methodologies that combine MBT and TDD/BDD and including other agile development approaches will be out of the scope of this thesis. Another threat to the validity could be the passing criteria of this SMS. There were some studies that passed the criteria, although they did not provide the required information. It shattered our confidence about the inclusion or exclusion of such papers. Likewise, there could be some or many papers that could have provided the required information but are not included in our SMS as they did not pass the criteria.

6. CONCLUSION

In this SMS, we presented the review of work combining Model-Based Testing and two of the agile development methods i.e., Test-Driven Development and Behavior-Driven Development. The main objective of this SMS is to explore the studies that discussed the approaches to integrate both the development trends to synergize the benefits of both the methods at distinct levels of a development cycle. The purpose of this thesis is to explore different domains that used MBT incorporation with TDD/BDD and to identify the available tools that support the proposed integrated methods.
This thesis does not aim at making any merit comparison or validity analysis of the proposed approaches. Instead, the purpose is to extract the exact information as presented by the author of that study.
To the best of our knowledge, no previous SMS has been conducted to review the approaches integrating MBT and TDD/BDD as a whole. Some of the papers have discussed the work regarding combining MBT and AD but not debated explicitly on TDD/BDD. This SMS has been conducted diligently based on 16 relevant primary studies. We included sixteen studies that were published between the years 2003 and 2020. These studies were found relevant to the specified research questions after identifying 917 papers through 7 electronic databases.
According to the results analyzed in this SMS, MBT and TDD/BDD are three of the most significant approaches in the software testing industry. However, they are different in the context of their development

strategy. Their unification can overcome the deficits of each other. In answer to the research question 1, approximately sixty three percent of the papers have illustrated their proposed methodologies through industrial cases, controlled experiments, or examples. Surveys are excluded from this paper. Twenty percent of the papers discussed the web application domain. Twenty percent of the papers debated on Service-oriented architectures. A total of ten percent of the papers illustrated their methods through a controlled experiment. Twenty percent of the papers used an automation system to explain their proposed approaches. Ten percent of the papers discussed the telecommunication system. Embedded systems have been discussed by ten percent of the primary studies. Ten percent of the papers discussed remote control systems. A general overview of the research contribution of the selected primary studies is shown in Figure 3.3 below.

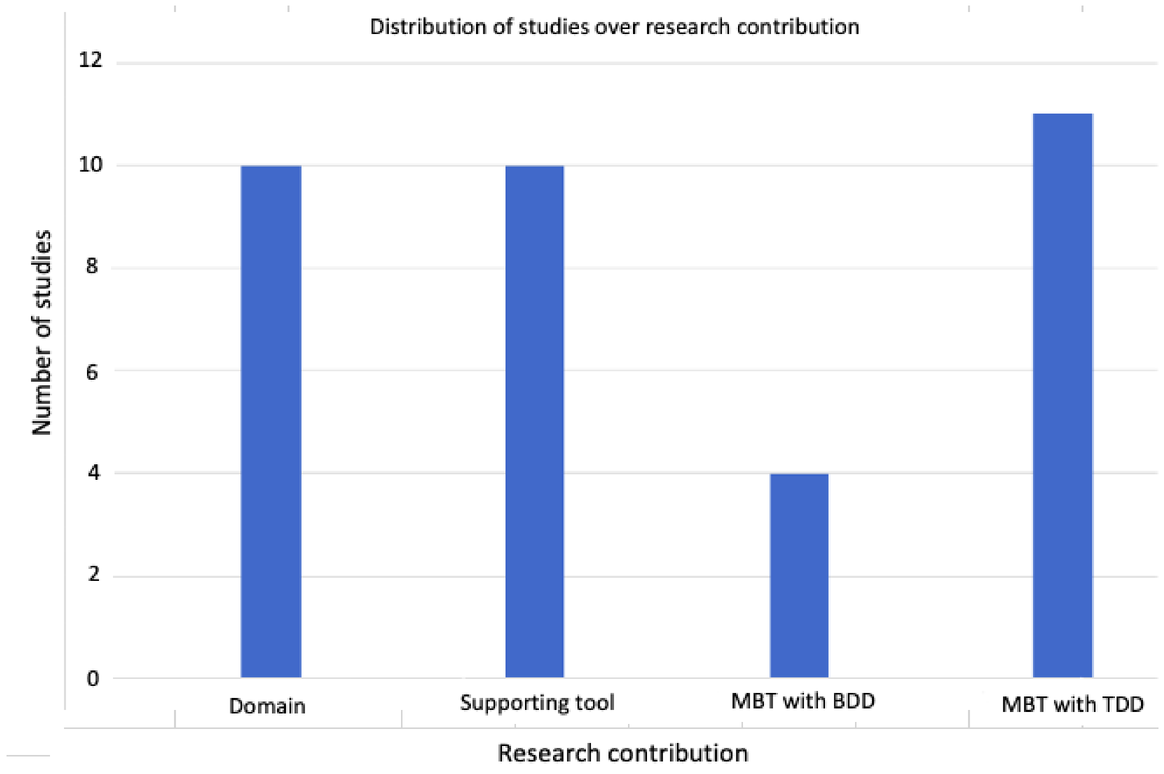


Figure 3.3 Study distribution with respect to contribution type

Based on this SMS, we analyzed that sixty-two and half percent of the primary studies have either practically or theoretically, implemented their proposed methodologies to combine MBT and TDD/BDD approaches using various testing tools. The results showed that the major focus of the work to combine MBT and TDD/BDD is on the test case generation and modeling the requirements at the specification level. Out of the selected primary studies, approximately fifty six percent of papers focus on test case generation. Twenty five percent of the studies discussed requirement modeling. Approximately six percent of the studies introduced methods to reuse the test artifacts. A total of six percent of the papers provided test coverage analysis strategies. Approximately thirteen percent of the studies focused on model checking methods. While approx. six percent of the selected studies presented methods that can provide stubs and guidance on the next step in the testing method. This thesis can facilitate the researcher for their future research work, who intend to cover the untouched areas in the related field as well as improve the identified strategies.

7. REFERENCES

[1] S. Wieczorek, A. Stefanescu, M. Fritzsche and J. Schnitter, "Enhancing test driven development with model-based testing and performance analysis," *Testing: Academic & Industrial Conference - Practice and Research Techniques (taic part 2008)*, Windsor, 2008, pp. 82-86.

[2] D. Mou and D. Ratiu, "Binding requirements and component architecture by using model-based test-driven development," *2012 First IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*, Chicago, IL, 2012, pp. 27-30.

[3] T. H. Ussami, E. Martins and L. Montecchi, "D-MBTDD: An approach for reusing test artefacts in evolving system," *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, Toulouse, 2016, pp. 39-46.

[4] N. Li, A. Escalona and T. Kamal, "Skyfire: Model-based testing with cucumber," *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Chicago, IL, 2016, pp. 393-400.

[5] A. Munck and J. Madsen, "Test-driven modeling of embedded systems," *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP & International Symposium on System-on-Chip (SoC)*, Oslo, 2015, pp. 1-4.

[6] B. K. Aichernig, F. Lorber and S. Tiran, "Formal test-driven development with verified test cases," *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Lisbon, 2014, pp. 626-635.

[7] Y. Zhang, "Test-driven modeling for model-driven development," in *IEEE Software*, vol. 21, no. 5, pp. 80-86, Sept.-Oct. 2004.

[8] B. Elodie, A. Fabrice, L. Bruno, and B. Arnaud, "Lightweight model-based testing for enterprise IT," *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Vasteras, 2018, pp. 224-230.

[9] R. Hametner, D. Winkler, T. Östreicher, S. Biffl and A. Zoitl, "The adaptation of test-driven software processes to industrial automation engineering," *2010 8th IEEE International Conference on Industrial Informatics,* Osaka, 2010, pp. 921-927.

[10] S. Sivanandan and Yogeesha C. B, "Agile development cycle: approach to design an effective model based testing with behaviour driven automation framework," *20th Annual International Conference on Advanced Computing and Communications (ADCOM)*, Bangalore, 2014, pp. 22-25.

[11] V. Mahé, B. Combemale, and J. Cadavid, "Crossing model driven engineering and agility: pareliminary thought on benefits and challenges," *3rd Workshop on Model-Driven Tool & Process Integration, in conjunction with ECMFA,* France, 2010.

[12] H. Bünder, and H. Kuchen,"A Model-Driven approach for behavior-driven GUI testing," Proceedings *of the 34th ACM/SIGAPP Symposium on Applied Computing - SAC '19*, New York, 2019, pp. 1742-1751

[13] E. R. Luna, J. Grigera, and G. Rossi, "Bridging test and model-driven approaches in web engineering," *Web Engineering, 9th International Conference, ICWE 2009*, Berlin, 2009, pp. 136-150.

[14] S. Hayashi, P. YiBing, M. Sato, K.Mori, S. Sejeon, and S. Haruna, "Test-driven development of UML models with SMART modeling system," *The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference*, Portugal, 2004, pp. 395-409

[15] L. Lazar, S. Motogna, and B. Pârv, "Behaviour-driven development of foundational UML components," *Electronic Notes in Theoretical Computer Science,* 2010, vol. 264, pp. 91-105.

[16] D. Faragó, "Improved underspecification for model-based testing in agile development," *Second International Workshop on Formal Methods and Agile Methods*, Italy, 2010, pp. 63-78

[17] A. Kramer, "Introduction to model-based testing, Process aspects of MBT, and executing model-Based tests" in *Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester*, 2016, ch. 1,3,9, [Online]. Available: https://learning.oreilly.com/library/view/model-based-testing-essentials/9781119130017/c01.xhtml#c1

[18] M. A. Khan and M. Sadiq, "Analysis of black box software testing techniques: A case study," *The 2011 International Conference and Workshop on Current Trends in Information Technology (CTIT 11),* Dubai, 2011, pp. 1-5.

[19] O. P. Puolitaival, "Model-based testing," VTT technical research center, Finland, [Online]. Available: https://www.cs.tut.fi/tapahtumat/testaus08/Olli-Pekka.pdf/. [Accessed 2020].

[20] J. Lindholm, "Model-based testing,", University of Helsinki, Finland, 2006, [Online]. Available: https://www.cs.helsinki.fi/u/paakki/lindholm.pdf/. [Accessed 2020].

[21] S. Nidhra, and J. Dondeti, "Black box and white box testing techniques - A literature review", *International Journal of Embedded Systems and Applications (IJESA),* India, 2012, vol. 2, No. 2

[22] "Model based testing tutorial: what is, tools & example," Guru99. [Online]. Available: https://www.guru99.com/model-based-testing-tutorial.html. [Accessed 2020].

[23] G. Sypolt, "The challenges and benefits of model-based testing," SAUCELABS, [Online]. Available: https://saucelabs.com/blog/the-challenges-and-benefits-of-model-based-testing /. [Accessed 2020].

[24] G. Joshi , " Benefits and drawbacks of model-based testing," FINDNERD, [Online]. Available: http://findnerd.com/list/view/Benefits-And-Drawbacks-of-Model-Based-Testing/10221/. [Accessed 2020].

[25] A. Huima, "Top three problems with model-based testing," Conformiq, [Online]. Available: https://www.conformiq.com/2012/01/top-three-problems-with-model-based-testing/. [Accessed 2020].

[26] "Iteration retrospective," SAFe, [Online]. Available: https://www.scaledagileframework.com/iteration-retrospective/. [Accessed 2020].

[27] "Agile software development," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Agile_software_development#Agile_software_development_methods. /. [Accessed 2020].

[28] "SDLC - Agile Model," Tutorialspoint.com, [Online]. Available: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm. /. [Accessed 2020].

[29] "Iterations," SAFe, [Online]. Available: https://www.scaledagileframework.com/iterations/. [Accessed 2020].

[30] K. Beck et al., "Manifesto for Agile Software Development," Agilemanifesto.org. [Online]. Available: https://agilemanifesto.org/. [Accessed 2020].

[31] K. Kulkarni, "What is Test-driven development(TDD)? Tutorial with example," Guru99, [Online]. Available: https://www.guru99.com/Test-Driven-development.html#2. /. [Accessed 2020].

[32] L. Williams, "White-box testing," 2006, [Online]. Available: https://students.cs.byu.edu/~cs340ta/spring2019/readings/WhiteBox.pdf /. [Accessed 2020].

[33] J. Hartikainen, "What's the difference between unit testing, TDD and BDD?," CodeUtopia, [Online]. Available: https://codeutopia.net/blog/2015/03/01/unit-testing-tdd-and-bdd/.[Accessed 2020].

[34] J. Nair, "What is BDD? An introduction to behavioral driven development," Testlodge, [Online]. Available: https://blog.testlodge.com/what-is-bdd/. /. [Accessed 2020].

[35] T. A. Gamage, "Behavior Driven Development (BDD) & Software Testing in Agile Environments," Medium, [Online]. Available: https://medium.com/agile-vision/behavior-driven-development-bdd-software-testing-in-agile-environments-d5327c0f9e2d. /. [Accessed 2020].

[36] A. Ghahrai, "Software testing fundamentals, Question and Answers," DevQA, [Online]. Available: https://www.testingexcellence.com/state-transition-testing/. [Accessed 2020].

[37] E. Dietrich, "What is dependency graph," Ndepend, [Online]. Available: https://blog.ndepend.com/without-dependency-graph-flying-blind/. [Accessed 2020].

[38] Rajkumar, "Decision table test case design technique," Software testing material, [Online]. Available: https://www.softwaretestingmaterial.com/decision-table-test-design-technique/. [Accessed 2020].

[39] "Transaction-flow testing and data-flow testing," Software Testing Methodologies, [Online]. Available: http://www.mcr.org.in/sureshmudunuri/stm/unit3.php. /. [Accessed 2020].

[40] G. Holzmann, "Software analysis and model checking," *International Conference on Computer Aided Verification,* New Jersey, 2002.

[41] "What is black-box testing?," Software testing class, [Online]. Available: https://www.softwaretestingclass.com/what-is-black-box-testing/. [Accessed 2020].

[42] "What is white-box testing?," Software testing class, [Online]. Available: https://www.softwaretestingclass.com/white-box-testing/. [Accessed 2020].

[43] P. Juneja and P. Kaur, "Software Engineering for Big Data Application Development: Systematic Literature Survey Using Snowballing," *2019 International Conference on Computing, Power and Communication Technologies (GUCON)*, NCR New Delhi, India, 2019, pp. 492-496.

[44] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic Mapping Studies in Software Engineering," *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, Italy, 2008.

[45] P. Ammann, and J. Offutt, “Goals of software testing, Software testing foundations” in *Introduction to software testing*, England: U. 2016, ch. 1, 2, sec. 1.2, 2.3.