JUNE 1, 2014

TRACEABILTY IN MODEL-BASED TESTING

M.SC. THESIS

AHMED DAWOUD

UNIVERSITY OF TAMPERE
SCHOOL OF INFORMATION SCIENCES
COMPUTER SCIENCE / INT. TECHNOLOGY
M.SC. THESIS
SUPERVISOR: ELENI BERKI
JUNE 2014

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

CODE Fragments

This page intended to be plain

Acknowledgement

This thesis would not have been conceivable without the help, backing and patience of my supervisor Eleni Berki. I would like to express my sincere gratitude to her. Likewise, I wish to express my grateful and thankfulness to my father.

University of Tampere
School of Information Sciences
Computer Science
Ahmed Dawoud: Traceability in model-based testing
M.Sc. thesis, 47 pages, 20 index and appendix pages
June 2014

Traceability is a substantial attribute of software quality assurance. Traceability ensures the precise translation of requirements throughout the software development life cycle. The prevailing techniques employed to trace requirements are manual as yet. The model-based testing is an entirely automated test generation approach; test suit is generated from models that describe system's behavior. This thesis analyses and discusses the automation of requirements to test cases tractability by means of model-based approach. A heuristic method is inferred to achieve traceability automation through model-based testing. Moreover, an application and evaluation of the method is conducted to define opportunities and deficiencies.

Keywords

Quality Assurance, Requirements, Software Testing, Black-Box Testing, Model-Based Testing, Automatic Test Generation, Modelling, and Traceability.

1. Introduction

Software project success depends crucially on fulfilling customer's needs and requirements. During the development process, the requirements are translated into various artifacts depending on the current development phase, for instance, requirements are translated into design models in the design phase, while those requirements are transformed into classes and methods in the implementation phase. There is a necessity to trace requirements during the software lifecycle to ensure the customer's needs are covered correctly, completely and precisely.

The goal of software engineering is to produce high quality software. IEEE standard glossary defines software quality as the degree to which a system, component, or process meets specified requirements and customer or user needs and expectations [IEEE Glossary, 1990]. The Validation and Verification (V&V) is a software development activity to provide confidence in the quality by detecting defects [Pezzè and Young, 2007].

In order to maintain quality attributes, e.g., correctness and testability; tests should be traceable to customer requirements. Requirements-to- test case traceability links requirements with test cases used to verify that the requirement is probably implemented. Currently, linking test cases to requirements is a manual process.

The current traceability techniques are manual approaches, for example, the linkage is done manually and managed by tools like DOORS. This thesis focuses on inferring a heuristic method to automate the traceability process between requirements and test suite. Since, Model-Based Testing is automation of specifications based approaches, I will discuss the potentiality to automate the requirements-to-tests tracing process. Moreover, I will apply the proposed method to a real world case study.

The thesis tackles the following question

RQ1: How to exploit the model-based testing to attain the requirements-test cases traceability automation?

The preceding question itself reveals sub-questions to inspect

RQ1.1: What are the various aspects of requirements' modulation in the model-based testing?

RQ1.2: What is the traceability model dominating model-based testing theoretically and practically?

RQ1.3: What are the opportunities and deficiencies in the recommended approach?

The scope of this thesis is to examine and analyze the automation of tracing requirements to test cases through model-based testing approach, to ensure that each requirement is at least covered by a test case, thus the client's confidentiality in product quality is asserted.

The research will cover requirements to test suit; the MBT cannot link requirements to the entire development process artifacts; for instance MBT provides no ability to trace requirements to design models, classes, methods and system units.

A systematic literature study on traceability and model based testing will be undertaken in Chapters 2 and 3 respectively; in order to obtain solid background knowledge for development of the model based testing.

The thesis will examine the requirements as an essential element of the software development life cycle. The second element to study is the software testing process, and different testing approaches. The research elaborates and emphasizes the relationship between software testing and requirements. The thesis forward to present an inclusive review for the model-based testing process.

The second phase in the thesis is an analysis and examination of traceability concept in model-based testing. Thereafter, in Chapter 4 the thesis proceeds a survey of current traceability implementations in various software tools. The tools selection was systematically comprehensive based on a previous study.

Subsequently, a heuristic procedure and required steps to automate the traceability will be defined. That framework formulates a systematic approach to achieve the automation of requirements-test cases traceability.

The proposed procedure is applied practically to a case study in Chapter 5. The case study is an Android game developed as a project at school of information science. Ultimately, in Chapter 6 a qualitative evaluation of the suggest framework will be conducted to specify utilities and deficiencies of the suggested approach.

Chapter 2
Background

2.1. Software Testing

A notorious incident of software failure is the rocket Ariane 5. Only after 40 seconds, the flight veered off its path and exploded. After investigations engineers had claimed that the reason was an unhandled floating point exception [Arian5, 1996].

Validation and Verification (V&V) are software development activities concerned with building the right product and build it in the right way [Boehm, 1979]. There are two approaches to conduct the V& V which are static and dynamic. Static approach analyzes and inspects software representations such as requirements specifications and system models, while the dynamic approach executes the system implementation with actual data and evaluates the results.

IEEE software engineering glossary defines Software Testing as," a concurrent lifecycle process of engineering, using and maintaining test-ware in order to measure and improve the quality of the software being tested. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior" [IEEE, 1990]. The fundamental goal of the testing process is achieving quality. In the context of software development, quality means the conformance to client requirements. There is a proportional relationship between quality and requirements conformance. Testing improves product quality by identifying defects and problems. Nevertheless, testing shows the presence of faults, it cannot reveal their absence [Dijkstra et. al., 1972].

Software testing definition had been profoundly changed over the last four decades. The changes have been caused by the increase in software complexity and the rapid changes in technology.

In 1979, Glenford Myers defined testing as, "the process of executing a program or system with the intent of finding errors," such a definition reflects the simplicity of testing process at that time of the first edition of [Myers et al., 2011]. In contrast to the current test definition, Myers's definition lacks for accuracy. In addition, testing prevents errors, and guarantees the system under test to satisfy requirements specifications. Myers states in the third edition of his classic book the test process had a dramatic shift.

Meanwhile, the complexity of software have escalated, and the importance and vitality of testing have increased in parallel. The proportional relationship between testing and complexity of software systems imposed a change in the software development process models.

When software process was simple and streamline enough the Water-fall model was satisfactory because of its concurrency. Craig and Jaskiel [2002] claimed that "The Waterfall model is particularly difficult to use successfully from the testing viewpoint".

Furthermore, they suggested the STEP as test oriented development model. However, at the first glance the reader cannot ignore the correspondence of the STEP and the standard V-model. The V-model is a software development model which can be presumed to be the modification of the waterfall model. Instead of falling down in a linear way as in the waterfall model, the process of the V-model is bent upwards after the coding phase, to form a letter V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing; each phase relies on verification from the preceding phase. Thus, the main focus of the V-model is testing after each phase [Mathur and Malik, 2010]. The V-model defines four testing levels, .i.e., unit, integration system, and acceptance testing. Unit testing works on the smallest components of the targeted the system under test. Substantially, components refer to implementation's items, .e.g., methods, classes, modules and interfaces. At this level, the test cases are written by the programmers, aiming to verify their code. Software system is comprised of subsystems, which, in turn, consist of integrated modules and units. As consequences of units' integration, faults and errors may arise. Integration test is essential to ensure the quality of interface and interactions between components and modules. System testing assures the system under test meets its functional and business requirements. Acceptance testing is the highest level, done by real system users. Also known as 'beta testing', 'application testing' or 'end user testing'.

Huang [2009] categories testing approaches into two approaches code-based and specifications based. It is common to refer to the two terms as white-box and black-box terms respectively. White-Box testing is also known as 'structure-based testing'. Therefore, it focuses on internal states of the system under test; it requires internal knowledge of the components. The white-box approaches inspect the structure of the small units; hence the code is the target. Code structure consists of statements, loops, paths and branches. White-box tests the core elements of the unit's structure.

Black-box testing is recognized as functional testing or specifications-based testing. The different names point out that the principal concern is system functionality, based on customer requirements. Black-box tests software against the specification of its external behavior without the knowledge of internal implementation details. It ignores the internal structure of the system under test [Agarwal et al., 2010]. There are three elements in the testing process, namely inputs, systems under test and outputs. The test oracle verdict the actual test cases output against the expected output. The test cases are derived from system requirements specifications.

The two types of testing are complementary; there is a need to check the code as much as we need to verify system functionalities.

2.2. Requirements

The ISO, IEC and IEEE defines software quality with six characteristics [ISO, 2010]:

1. *The degree to which a system, component or process meets specified requirements*

2. *The ability of a product, service, system, component or process to meet customer or user needs, expectations or requirements*

3. *The totality of characteristics of an entity that bears on its ability to satisfy stated and implied requirements*

4. *Conformity to user expectations, conformity to user requirements, customer satisfaction, reliability and level of defects present*

5. *The degree to which a set of inherent characteristics fulfils requirements*

6. *The degree to which a system, component or process meets customer or user needs or expectations*.

Evidently, requirements are closely coupled with the quality as noticed in the previous standard definition. Requirement is "a software capability needed by the user to solve a problem to achieve the objective and a software capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documentation [Wiegers, 2003]. A requirement is a condition or capability needed by the user or constraint imposed by a contract, standard or specification.

Requirements are classified into two categories [Sommerville, 2010]. Firstly, functional requirements are statements of services the system should provide, how the system should react to particular inputs and how the system should behave under a particular situation. Those requirements can be modeled with use cases, and analyzed by sequence diagrams, state chart or other forms. Moreover, functional requirements are traceable. Secondly, non-functional requirements (NFR) are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process and standards. NFRs often apply to the system as a whole, and they cannot be implemented as a program module.

The activities of requirements engineering are elicitation, analysis, specifications, validation, and management.

Requirements elicitation is the process of acquiring requirements through interaction with stakeholders, organizational documents, and domain knowledge. Requirements elicitation is "a synthetic process consisting of social communication and information mining [Zhang, 2007]." Requirements are discovered through consultation with stakeholders, from system documents, domain knowledge and market studies. At this stage, the main goal is to understand that the system being developed and its constraints. Different techniques are used in the elicitation process: interviews, brainstorming, Joint Application Design (JAD), observation and ethnography, requirements reuse, and prototyping.

Requirements analysis classifies the information received from users to functional requirements, nonfunctional requirements, business rules, suggested solutions, and extraneous information. The foremost goal of the analysis process is to resolve conflicts among requirements through negotiation. At this stage, the collected requirements are analyzed to specify incompleteness, and inconsistency. Furthermore, requirements prioritization takes place to balance the benefits and costs of such requirements.

The requirements are documented and precisely described. All of the essential requirements of the software and the external interfaces are documented. The documentation defines functions, performance, design constraint, and quality attributes. Each requirement is identified in such way that, its achievement is capable of being objectively verified by a prescribed method; for example inspection, demonstration, analysis, or test.

Requirements validation is the process of checking the requirements for validation. Quality criteria are used to ensure the validity of the requirements specification. Requirements are validated according to the question "did we get the requirements right?" Requirements' quality attributes include correctness, completeness, unambiguity, traceability, priorities, consistency, and verification [Lutowski, 2005].

Finally, requirements management is to keep the integrity and accuracy of requirements. It consists of the subsequent phases: change control, version control, requirements tracing, and requirements status tracking.

2.3. Requirements traceability and software testing

Gotel and Finkelstein [1994] define traceability as the ability to describe and follow the life of a requirement in a forwards and backwards direction. Traceability links bidirectional requirements to design, implementation and testing. Each requirement is uniquely identified. Traceability is used to manage the change in requirements. Constantly, change in stakeholders' needs lead to change in requirements, and this change has a domino effect through the development life cycle. If the requirements are traced this eases the change management process. Traceability is a powerful technique to validate requirements, by mapping test cases to uniquely identified requirements.

Traceability is categorized into two classes: pre-traceability and post-traceability, both of them are bidirectional. Pre-traceability from requirements origins to requirements specification; tractability origins may be stakeholders, domain requirements and standards. Post-traceability from requirements to the development life cycle, here the requirements mapped to use cases for implementation and test cases for validation and verification.

During requirements development process, there are four abstraction levels of requirements: stakeholder's requirements, system requirements, subsystems requirements and components requirements. Requirements' details increase as we move from one level to the lower.

Stakeholder's requirements describe services expected from the system; constraints imposed, and the way in which the system provides relevant services. At this level, requirements are supposed to be written in understandable and clear form for non-technical persons. As consequences of the requirements informality, ambiguity and non-clarity are critical problems. Moreover, the presentation of requirements is inconsistent and incomplete.

System requirements are the basic guidelines for the system being designed. They are detailed level for user requirements. Analysis is conducted to determine what is exactly to be implemented and the describe the system intended to be produced

Subsystem requirements components' requirements define the facts for the implementation of functions. Software requirements define goals that are necessary for the implementation of the intended features on a specific hardware component. Software level scenarios describe the desired system, internal behavior and interaction.

Requirements tractability used to assure the high level requirements are translated accurately into the lower level. Requirements traceability refers to "The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match." [IEEE 610]. Requirements tractability is bidirectional activity. Forward tractability flows from customer need to system's implementation, where backward is the opposite direction from the final product to customer needs.

According to, Bender RBT Inc [2009]:., the requirements-based testing process addresses 2 issues

- Validating that the requirements are correct, complete, unambiguous, and logically consistent.
- Designing a necessary and sufficient set of test cases from those requirements to ensure that the design and code fully meet those requirements.

The two objectives mentioned above are the Holy Grail for software quality. However, there are issues originate when dealing with requirements. Requirements are not computable; in effect requirements in the specification document are still in natural language. Model-based testing approach induces towards requirements for processing by redefine requirements in formal format that can be automatically manipulated.

MBT generates test cases from a model; that model is derived from system requirements specifications. This process implies the transformation of informal or semi-formal requirements into a formal representation. Requirements at this stage can be processed by a test engine.

Utting and Legeard [2006] allege that "Finding requirements issues is a substantial benefit of model-based testing because requirements problems are the principal sources of system problems. Requirements' faults found decrease the faults propagation to subsequent design and implementation phases. In addition, discovering faults early reduce the maintenance cost. This may be the area where model-based testing will have the largest impact on software development practices. It encourages early modeling, and this early modeling finds requirements and design".

Nevertheless, finding requirements inconsistency and incompleteness is essential early in the modelling phase. The MBT capabilities are revealed at the phase of the test case generation. The test suit is adapted according to test plan. Meanwhile, MBT is a requirement based testing approach, there is a primary trait that distinguishes MBT from traditional requirements based testing approach; it is the automation.

Chapter 3
Model- Based Testing

Model-based testing is a black box testing approach, whereas the test cases are entirely generated from a model partially or fully represents the aspects and of the system under test. The remarkable feature of the MBT is to stimulate the testing process towards automation. In contrast with the traditional testing approaches, e.g., manual testing, script-based testing and capture and play testing, MBT process is fully automated testing approach. Automation leads to less cost, less time, and increased number in available test cases. The test cases are originated from a model, which is a behavioral description. System's behavior is inputs, actions, restrictions, and output of the system.

3.1. Taxonomy of MBT processes

To automatically verify system functionalities, the starting point is requirements. Since, requirements are not computable, and they are in a form of natural language or descriptive models like use cases; the MBT initiates to transform requirements into a computable form. The naive view of MBT is a set of test cases generated from specifications; thereafter, apply those test cases to the system under test. However, technically there are issues to manipulate for instance, how to adapt the abstract test cases to the SUT, there should be a transformation phase.

Utting [2005] suggested two different workflows for MBT; the first model consists of four phases which are

- building an abstract model of the system under test
- validating the model
- generating abstract tests from the model
- refining those abstract tests into concrete executable tests.

The second workflow composed of five steps are shown in Figure 1: *Modeling*, *Test generation*, *Transformation*, *Execution*, and *Analysis.*

The first model neglects test execution and test analysis phases, in addition the first two steps can be combined in a single step. This thesis adopts the second workflow, as it is more detailed and comprehensive. Subsequent sections manipulate the process in details.

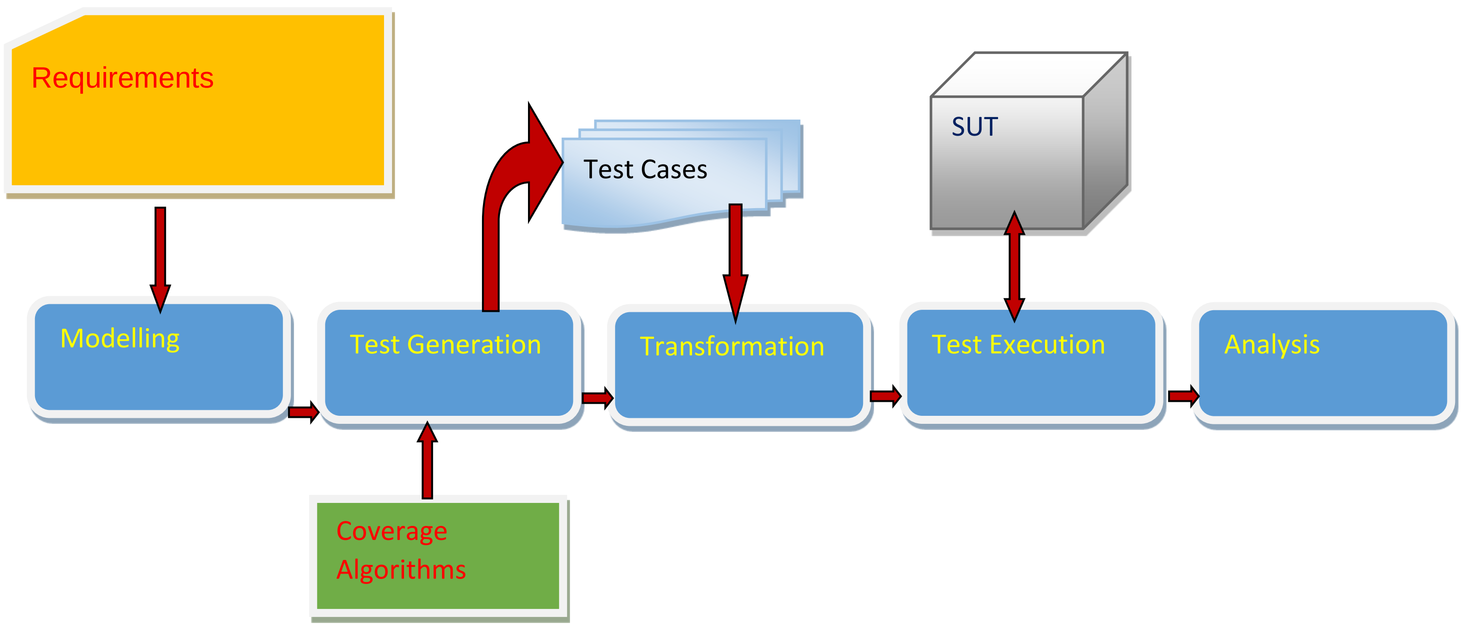


Figure 1. Model based Testing Process

3.2. Modelling

A model is an abstract view of a system that neglects system's details. Complementary system models are developed to demonstrate the system's context, interactions, structure, and behavior [Somerville, 2009]. Modeling is a translation of informal requirements document in a precise formal specifications. MBT targets system functionalities, for this purpose MBT uses a model that represents behavioral aspects of the SUT. Behavior model describes the internal logic of the SUT, behavior model ignores the internal implementation and structure of the SUT. In MBT, a model is likely designed to cover some aspects of the system rather than the comprehensive SUT behavior; this depends on the required abstraction level, the characteristics of the SUT, and the test's targets. A wide range of formal notations are available. El-Far and Whittaker [2001] lists the modelling notations as: finite state machines, state-chart, Markov chains, unified modelling language, and grammars notations. However, some of those notations can be synthesized in consistent groups, since they have the same characteristics. There are two groups of modelling notations:

- transitions based, e.g. finite state machines and ULM statemachines
- grammar based like Z, B and VDM.

The selection of appropriate notation depends on two factors. Firstly, the type of the targeted system under test; whether it is a data-oriented or a control-oriented system. For the data oriented there is a need to flexible representation of data variable; notations like B or Z will be an appropriate selection. In control-oriented systems the main concerns are the systems' states and transitions between them; in this case finite state notations is the right option. The second element in selecting the right modelling notation is to decide what is the test target? For example, if the safety is the goal, a transition based notations will be a good selection, while a model written in VDM or B notation is suitable for usability targeted testing as wider range of system input variables can be tested.

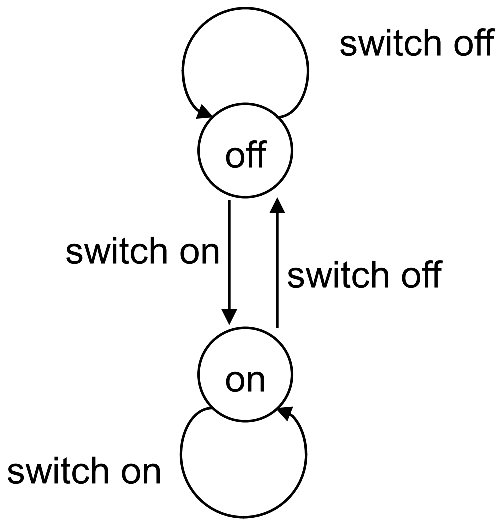
3.2.1. Finite state machines

Input /output behavior is specified from the viewpoint of describing the system states. Possible transitions to other states and the actions that trigger them are described for each state. A typical instance of state-based model notations are finite state machines. Finite state machine notation is a representation or modelling of the system elements described below:

- Input or trigger, is an event that is generated internally or externally and might cause a state transition.
- State is a behavior description, the information stored in system at one point in time, simply it is an object in object oriented programming. States may produce actions.
- Action or transition is a movement from a state to another, it triggered by stimulus.

FSM is described as $M = (S, i, T, \Sigma, \delta)$.

Where S is a finite, non-empty set of states; i is the initial state ($i \in S$); T is the finite set of terminal states ($T \subseteq S$); Σ is a finite alphabet of symbols or events used to mark transitions between states; and δ is a transition function that describes the next state of the FSM given the current state and a symbol from the alphabet [Laplante and Ovaska, 2012].



input events	current state	next state	outputs
Switch on	on	on	-
Switch on	off	on	click
Switch off	on	off	click
Switch off	off	off	-

Figure 2. FSM light graph and table

FSM is represented as graphs or tables. Figure 2 shows the states and possible transitions for a simple light switch. A switch states are {ON, OFF}, those states are changed by switching light {Switch up, witch down}. State and transitions concepts are essential in all behavior modelling notations.

However, Harel [1987] argues that FSMs are not suitable to model complex systems. Because of unmanageable flat states structure. In addition, Laplante and Ovaska [2012] state two more defects in FSM modelling notations

- The internal aspects of the modules cannot be depicted.
- Inter-task communication between multiple FSMs is difficult to depict.

FSMs are suitable to represent reactive systems, but, its description capabilities are limited as there is no conditioning notations. Extended FSMs version of FSMs, e.g., state-charts and UML state-machines are developed to improve modelling capabilities.

3.2.2. Markov Chains

Markov chains are stochastic models, which can also be used to software modelling [Prowell, 2005]. Markov chains are probabilistic state machines, meaning that the transitions of the machine are augmented with probabilities, which are used to select which transition to choose whenever leaving a state. For example, figure 3 shows the probability of the transition from S1 to S3 is 0.05.

A stochastic process { X_n } is called a Markov chain if

$\Pr\{ X_{n+1} = j \mid X_0 = k_0, \ldots, X_{n-1} = k_{n-1}, X_n = i \}$

$= \Pr\{ X_{n+1} = j \mid X_n = i \}$ ← transition probability for every $i, j, k_0, \ldots, k_{n-1}$ and for every n.

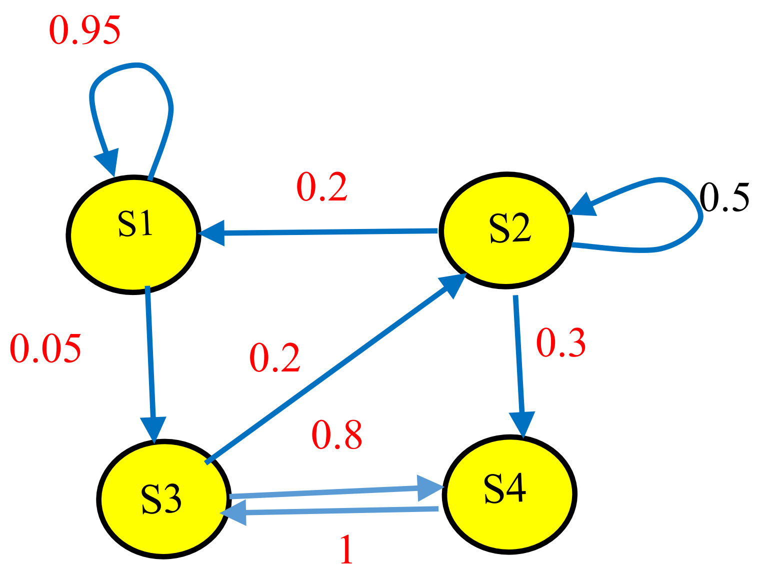


Figure 3. Markov Chain example

They can be easily used to measure, e.g., software's reliability and average time of failure. Markov chains still have the same deficiencies as FSM. The growth of the number of states and transitions impacts in the readability.

3.2.3. Statecharts

Statecharts were originally developed by Harel [1987] to overcome the complexity of systems modeling. Statecharts are visual extension of finite state machines that specifically addresses modeling of complex or real-time systems [El-Far and Whittaker, 2001]. Substantially, statecharts are finite state machines extended with hierarchy, concurrency and communication [Liuying, et al., 1999]:

Statechart = FSM+ Hierarchy + Concurrency + Communication.

Statecharts introduced state hierarchies by grouping states in super state using conjunction and disjunction operators AND & OR respectively. Furthermore, time, concurrency and synchronization were introduced. The depth is the aspect of hierarchy, it means an alternative states of an object, and this results in an OR states. An OR state is a super-state of its sub-states. Concurrency is an AND-decomposition of a state. If state S is to be in all of its components, then S is an AND state**.**

This offers the possibility for expanding states into lower-level state machines to model complex or real-time systems. Furthermore, conditions and triggers can be imposed on transitions. Statecharts model represents system states as nodes and events as arcs between these nodes. When an event occurs, the system shifts from one state to another.

Modelling capabilities introduced by statecharts over the primitive finite state machine are significant. However, it does not conform to object oriented approaches. An adjusted notation was introduced by combining UML and statecharts.

3.2.4. UML Statecharts

Unified Modelling Language (UML) is general purpose language to specify, visualize, construct and document the artifacts of a software system. A behavioral object oriented model can be constructed by using statecharts notation in combination with UML. Figure 4 is a UML states chart modelling of a light lamp previously modelled with FSM.

.

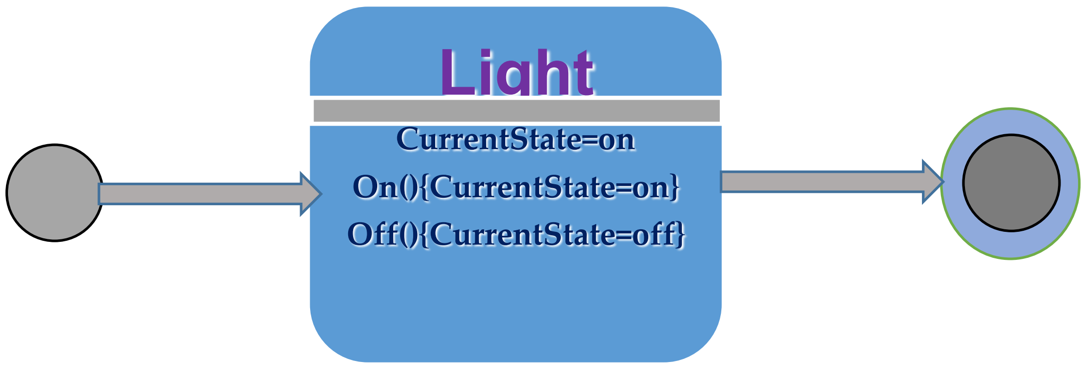


Figure 4. UML state machine for light switch

Sommerville [2009] suggests two methods to describe behavior aspects assigned to UML by specified declaratively using the object constraint language (OCL) or expressed using UML's action language. The action language is a very high-level programming language where you can refer to objects and their attributes and specify actions to be carried out.

3.2.5. Grammar-based specification languages

Specify admissible system states or value) at some arbitrary snapshot, using mathematical entities like sets, relations, first order predicate logic (pre-condition/post conditions, invariants). The specification is expressed as a system state model. This state model is constructed using well understood mathematical entities such as sets, relations, sequences and functions. System's behavior is specified by defining how they affect the state of the system model. Operations are also described by the predicates given in terms of pre and post conditions [Utting, 2006]. The most widely used notations for developing model based languages are Vienna Development Method (VDM), Zed (Z) and B. Each method has its own mathematical model, identification of input/output and its structure.

For example, code fragment1 models a coffee machine, represented as B schema structure. B-Method is a formal method for the development of program code from a specification in the abstract machine notation.

```
MACHINE Coffee                          The name of the machine is Coffee

VARIABLES coins                         We need a variable coins

INVARIANT coins : NAT                   coins is a natural number

INITIALISATION coins := 0               We will start with coins set to 0

OPERATIONS                              Now we have the operations

addcoins(amount)=                      We need an operation addcoins

PRE amount : NAT                        Amount is a natural number

THEN coins := coins + amount            We set  coins

        coins + amount

END;
```

Code fragment 1. B model for a simple coffee machine

3.3. Test generation

Generating targeted test cases suit is the second phase in the MBT process. Test cases are obtained from the appropriate SUT model. A model can generate an enormous number of test cases, manipulating a huge number of test cases is expensive and time exhaustive. Test generation criteria are appointed to direct test cases to meet test strategy and targets according to test plan. Coverage criteria measure the conformance of test case suite to the model to assure the quality of test cases; coverage criteria also known as stopping criteria. Utting [2006] categorized test generation criteria as structural, data, random, mutation and requirements coverage.

3.3.1. Model structure coverage criteria

Structure criteria depend on the modelling notation. In case of grammar-based specifications languages, for instance Z or B notations, model structure relies on conditions and data sets. For example, code fragment 2 shows Z model, ReadMaster(a,p) will be executed only if the condition r is ok true.

```
procedure Access(a : ADDR; var p : PAGE);

var r : REPORT;

begin

GetChange(a; p; r );

if r= ok then

ReadMaster (a; p)

end;
```

Code fragment 2. Conditions in Z model

The model structure in this case is similar to code-based structures, analogous techniques to cover the model are used. In the case of using transition-based models, the model structure consists of two elements states and transitions.

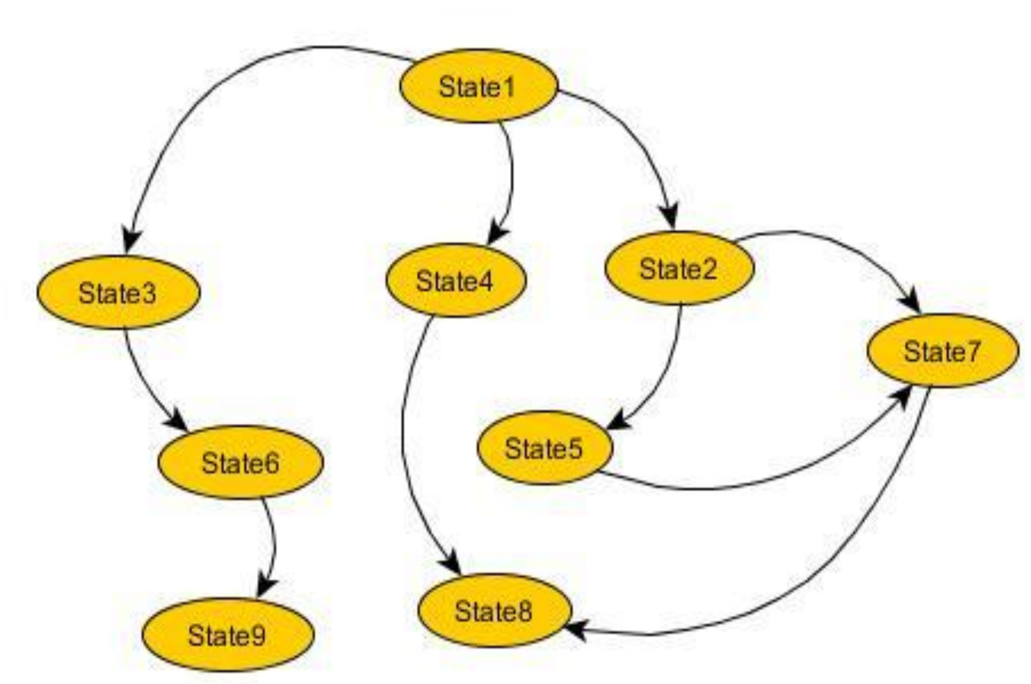


Figure 5. State-based structure

Figure 5 demonstrates a simple transition machine, algorithms like shortest path or A* can be used to traverse the graphs. The Chinese Postman algorithm is the most efficient way to traverse each link in the model. In the case of Markov chains graphs. Robinson [1999] uses The Most Likely Paths First algorithm which manipulates the probabilities assigned with each transitions. In transitions based models the coverage criteria are all states, all transitions, all transitions pair, or all paths.

3.3.2. Data criteria

Data values are partitioned into deferent sets. At least one representative from each set is used for a test case, for example using boundaries value testing.

The random criteria are suitable for environmental models. The environment model represents the behavior of the environment of the SUT. Test cases are generated using probabilistic models. In other words, test cases are generated based on the probabilities that are assigned to the transitions.

Monkey testing is a software testing technique in which the testing is performed on the system under test randomly. Input data that are used to test is also generated randomly and keyed into the system. There are two types of monkeys which are the dumb and smart monkeys; relying on the level of knowledge. In case of dumb monkeys there is no knowledge of the properties of SUT, while the smart monkeys have basic knowledge of the SUT. [Nyman, 2001]

In fault based criteria this coverage, pre-specified faults are tested for the absence. The model is mutated. Then tests are generated to distinguish between the original and mutated model.

Fault-based testing is also known as 'Mutation-based testing', it is techniques is to generate test cases that can detect the injected errors. This means that a generated test case shall fail if it is executed on a system-under-test that implements the faulty model. The power of this testing approach is that it can guarantee the absence of certain faults. In the case of unit testing the mutant could be a change in a statement, a condition, or a variable. While in MBT the mutant can be changed in a state, a guard or a transition. The effectiveness of mutant-based testing depends on the selection of mutant and fault scenarios [Pezzè and Young, 2007].

In requirements based criteria, the requirements act as test targets coverage criteria, when the elements of a model are associated with the requirements.

3.4. Transformation

Transformation is a translation process from abstract tests in tests suite to an executable scripts, also recognized as adaption phase. Utting suggests three methods to adapt the abstract test cases. Firstly, adaptors are written to engage the gap between SUT and the test cases .Secondly, test scripts are explicitly written from test pool. Lastly, a hybrid approach is adopted to make test cases more conform to SUT and write scripts for execution tests cases.

3.5. Tests execution

They are two modes to execute test cases:

- In online MBT test cases are directly executed on-the-fly. This mode of execution is quite random and the test strategy is to achieve maximum coverage. Online MBT strategy is effective and suitable for non-deterministic systems.

- In offline MBT the test suit generated and the execution is postponed. This mode helps in testing specific functions of the systems; for instance, testing critical states of the system that depends crucially on timing or to assure system's security functions.

3.6. Results analysis and reporting

The final phase in the MBT process is analysis, where the actual results are compared to the expected results. Tracing test suit to requirements is one essential method at this phase. During MBT life cycle, there are three levels of traceability. Firstly, the conversion of informal requirements to formal notation, this occurs at modelling phase. Secondly, from model to test cases, where test suit is generated by requirements coverage criteria. Thirdly, from test cases to requirements, a traceability matrix reports any deficiencies in requirements. Thesis analyses and discuss different approaches to achieve functional requirements traceability in MBT.

3.7. MBT metrics

Metrics are used to evaluate the generated test suit. In rigorous words, measure the conformance of the test cases to the test plan. Furthermore, metrics are exploited as stop conditions to fulfill the test. The available metrics are discussed in this section.

3.7.1. State coverage metrics

Measuring traversed states against all model states is an effective tool to measure the test suit. The test cases generated from a model supposedly tremendous; hence, the tester should have his own condition to fulfil the testing. The condition is a percentage of states or transitions of paths that the test engine must traverse. For instance, in Figure 5 the test terminated if the engine traversed 70 percent of states. This is straightforward in that model but in case the model has a huge number of states, with hundreds of states it is worth defining exit states or a coverage percentage if there are no critical states like states that adhere security or safety.

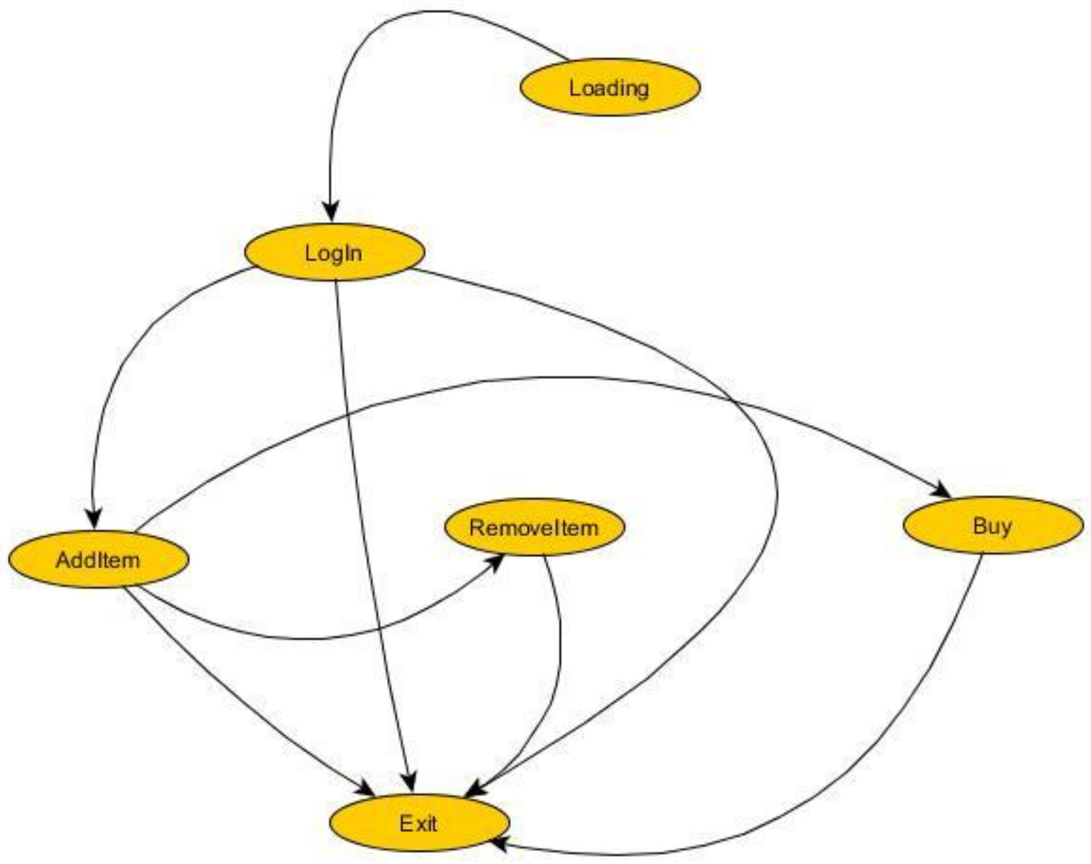


Figure 6. Fulfilling tests by approaching pre-defined traversed states

Moreover, the test suit conformance is measured by traversing a specific state or set of them, for example; the test suit is conforming to test targets if it visited the purchasing state in figure 6.

3.7.2. Requirements coverage metrics

Requirements coverage metrics are explicitly measuring the percentage of requirements covered in the test suit. This relationship can be expressed as a traceability matrix. In addition, a single requirement or a set of them are used to measure the conformance.

3.8. MBT benefits

MBT has a significant trend in software testing technologies. Interest in MBT not only limited to academia. Likewise, considerable software companies had applied the MBT in several projects, for example, Microsoft implemented its own version of MBT Spec Explorer. In 2003, 35 Microsoft product teams engaged in model-based testing with 600 of Microsoft testers involved in some form [Robinson, 2003]

The main advantage of MBT is the automation of test suit generation. The automation of the testing process reduces testing time, less time translated to less effort and less cost. MBT reveals undetected errors and defects that cannot be detected by traditional testing approaches, for example a study of applying MBT to NASA's system [Gudmundsson et al., 2013].

The MBT is a requirements-based approach; modelling requirements exposes contradiction incompleteness, inconsistency and deficiencies in the informal requirements. Detecting errors in requirements is a fundamental concern in the entire system development. Formalization of requirements eases the communication between teams and individuals in using a model of user behavior that can serve as point of reference to all, as the concepts are evident and clearly defined [Utting, 2006]. In addition, there is a flexibility in requirements changing.

3.9. Limitations of model based testing

The deployment of model based testing into an organization requires significant efforts and investments. Robinson [2003] and Utting and Legeard [2006] deduced a set of obstacles which intercept the expansion of model based testing. These include the following:

o MBT is functional oriented approach. There exists a difficulty in testing non-functional requirements, like security, usability, and performance

o Testers should dominate high skills. They need to be familiar with the modeling, which means knowledge of diverse forms formal specification notations. In addition, expertise in tools and scripting language is required.

o MBT proved productivity in real time systems, where there is a relative small number of states and transitions. However, when it comes to the enterprise information systems there is a huge number of states, which not consistently supported by recent modeling notations.

o A large initial effort in terms of man-hours are required. The type of the model has to be carefully selected. Different parts of software have to be divided so that the modelling is easier because of the smaller areas and the actual model has to be built.

o Models themselves have also some drawbacks. The biggest one of those is the explosion of state-space needed. Even a simple application can contain so many states that the maintenance of the model becomes difficult and tedious task.

Chapter 4
Requirements Traceability in Model-Based Testing

Chapters 2 and 3 emphasized the relationship between MBT and the requirements. Accordingly, the relationship between MBT and requirements traceability is distinct. Thus, there are remarkable questions to tackle when manipulating the traceability in MBT:

- What is the traceability model in MBT?
- How to automate the traceability, and exploit the automation characteristics of MBT, where the requirements are implicitly included in the model?
- What are the current implementations and limitations?

4.1 Traceability model of MBT

In traditional testing, the test suit generated manually or partially automated and then linked manually to requirements directly. While, in MBT the relationship is further complicated. Traceability features in MBT are

- A ternary nexus connects the requirements, model and the test suit.
- Traceability is restricted to test cases, we cannot trace other system's artifacts for example we cannot trace requirements to classes or design models (design models are distinguished from MBT testing models).
- Traceability is bidirectional activity

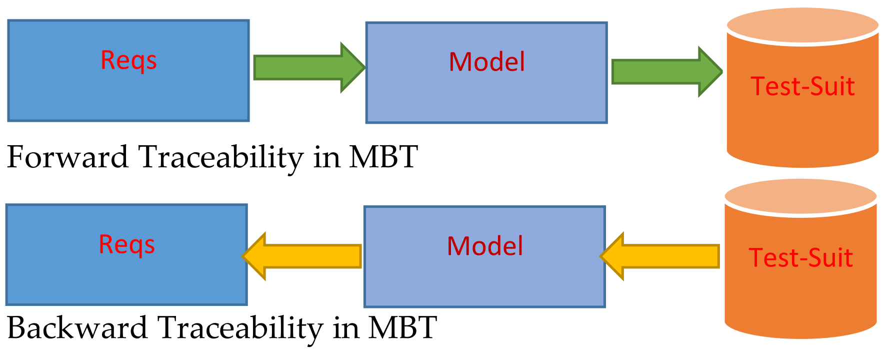


Figure 7. Traceability model in MBT

4.2. Traceability implementations in MBT tools

A model based testing tools survey was conducted by Shafique and Labiche [2010]. The survey was predicated on a four criteria one of them was requirements coverage. Table 1 compares 9 MBT tools from requirements implementations perspective.

Tool	**Model Type**	**Category**	**Requirements Coverage**
GOTCHA-TCBean	FSM	IBM Internal	Not Implemented
Mbt (replaced by graphwalker)	FSM/EFSM	Open Source	Supported
MOTES	EFSM	Research	Not implemented
TestOptimal	FSM	Commercial	Not Implemented
AGEDIS	UML/AML	Research	Not Implemented
ParTeG	UML	Research	Not Implemented
Conformiq Qtronic	UML	Commercial	Supported
Test Designer	UML	Commercial	Supported
Spec Explorer	FSM/ASM	Commercial	Supported

Table 1. MBT tools that supporting requirements [Shafique and Labiche, 2010]

In consonance with the study result, the thesis will discuss two tools, which support requirements, Conformiq Qtronic, Spec Explorer will covered subsequently. In addition, the thesis discusses a suggested implementation of Simulink. Furthermore, the tools are from a single group of modeling notations. To be coherent and comprehensive, the thesis will also demonstrate LEIRIOS tool that falls in grammar based modeling group.

4.2.1. Conformiq Qtronic

Conformiq is a commercial tool implemented in Eclipse. Modeling notation is UML State Machines in Qtronic Modeling Language (QML) [Conformiq, 2009]. Tests are exported to test management tools or TTCN-3. The keyword requirement is added to a model. Moreover, it is included in the design configurations. This implies the user can define a requirement as test generation criteria. Figure 8 shows requirements attached to transitions. A Conformiq requirement management tool connector provides features to check requirement catalogs from management tools like DOORS and HP QualityCenter against QML requirement statements specified in the model. A traceability matrix is generated for test suit analysis.

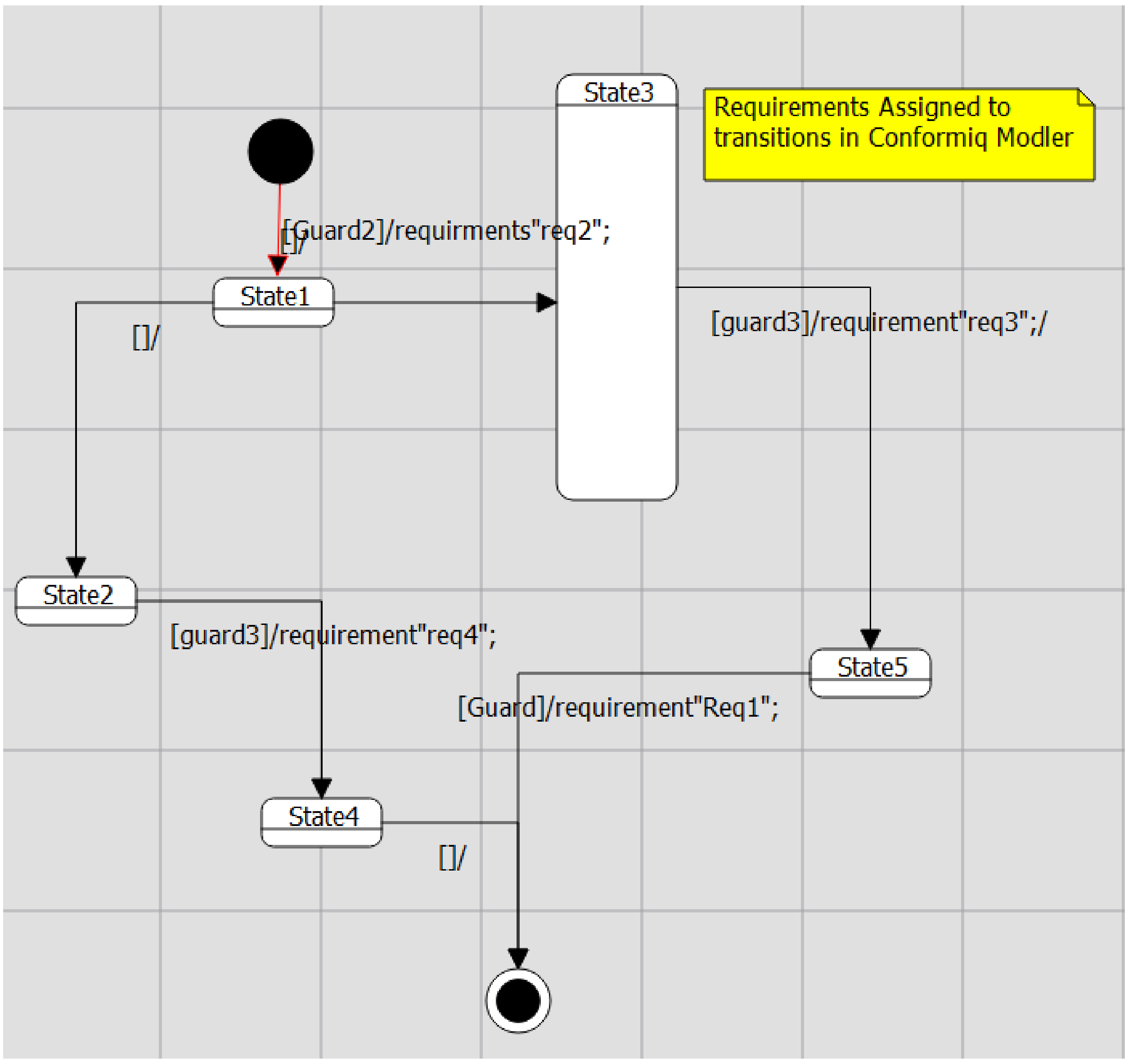

State3
Requirements Assigned to transitions in Conformiq Modler
[]/
[Guard2]/requirments"req2";
State1
[]/
[guard3]/requirement"req3";/
State2
[guard3]/requirement"req4";
State5
[Guard]/requirement"Req1";
State4
[]/

Figure 8. Tagging requirements in Qtronic

4.2.2. Spec Explorer

Spec Explorer is Microsoft implementation of the MBT. It extends Visual Studio to model systems' behavior and generates test cases automatically. Spec# is the modelling notation.

Spec Explorer declaratively supports associating requirements to preconditions and updates by calling specific library methods in C# code. Requirements are covered in each step (and in the path leading to each state) are recorded as part of the exploration results and transferred to generate test suit. The method Capture() in class ***Requirement*** in package *Microsoft. Modeling* points to a requirement. When Spec Explorer encounters a capture method through the exploration, it adds the requirement to the set of the captured requirements [Spec#, 2005].

Requirement.Capture (RequirementId.Make("Req1", 01, "Descreption"))

```
[Rule]
static bool AddJob(int jobId, string jobName)
{
    // Requirement 1: Job identifiers MUST be greater than zero.
    // Requirement 2: Active jobs MUST have a unique identifier.
    // Requirement 3: Job identifier MUST be encrypted with SHA-01.

    Condition.IsTrue(jobId > 0, "req-01");

    bool success = !activeJobIds.Contains(jobId);
    if (success)
    {
        activeJobIds.Add(jobId);
        Requirement.Capture("req-02");
    }

    // This requirement can be only validated by the adapter.
    Requirement.AssumeCaptured("req-03");

    return success;
}
```

Code fragment 3. Tagging requirement in Spec#

Code fragment 3 shows The [Rule] keyword represents a transition in FSM, the Requirement.Capture() is assigning a requirement to the transition.

4.2.3. Simulink and DOORS

Blackburn et al. [2005] suggested their own traceability implementation through linking DOORS and Simulink. The suggested mode adds requirements module from DOORS to Simulink molder TTM, the linkage between the DOORS and TTM done over requirements IDs (the concept is analogous to the primary keys in rational database concept).

4.2.4. LEIRIOS Test Generator

Bouquet et al. present traceability in MBT via LTG [[Bouquet et al., 2005]]. LEIRIOS utilizes B modeling notation. In Chapter 3, B was grouped in the grammar based notations. In code fragment 4 the requirements are tagged in the model as **/*@REQ: DISABLE3 @*/.** For instance, in case the first IF predicate is applicable then the requirements disabled3 is proofed. The test engine tracks and manipulates the requirements tags for analysis and traceability matrix generation.

```
PRE
code_cc : CODE
THEN
IF (blocked_chv1_status = blocked) THEN
sw := 9840 /*@REQ: DISABLE3 @*/
ELSE
IF (enabled_chv1_status = disabled) THEN
sw := 9808 /*@REQ: DISABLE2 @*/
```

Code fragment 4. Tagging requirements in B method code.

4.3. A heuristic method for requirements tracing in MBT

Driven from the MBT characteristics and the preceding analysis of requirements traceability in MBT tools, requirements tracing procedure is recapitulated as:

- Requirements tagging
- Requirements as test generation Criteria
- Requirements analysis
    - Requirements coverage criteria
    - Requirements traceability matrix.

4.3.1. Requirements tags

The first phase in MBT is to formalize requirements to attain a model; that model represents functional aspects of the SUT. The model is a logical form of requirements, but there is no explicit mention for requirements themselves, the requirements are implicitly included. Requirements tagging techniques are used to explicitly link requirements to the model. Requirement identifier is uniquely refers to a requirement record; consecutively, the identifier is annotated in the model. Tagging achieves two goals, firstly, requirements document review against the model, and this guarantees every requirement is mentioned. Secondly, requirements explicitly processed, for example, requirements can be targeted for testing. Tagging annotation depends on the modelling notation.

The vending machine requirements were elaborated in textual form. There are 6 requirements for describing the machine behavior from the point of view of the user. They are:

R1. Pushing a start button shall activate drinks available list.

R2.The customer selects a drink from the list then the price is displayed

R3.The customer inserts coins, only one and two euros are accepted

R4. The customer is able to dispense the drink if his coins balance is equal to or greater than the price.

R5.If the customer balance is greater than the price; the machine shall return the change.

R6.The client is able to cancel the transaction and get back his money.

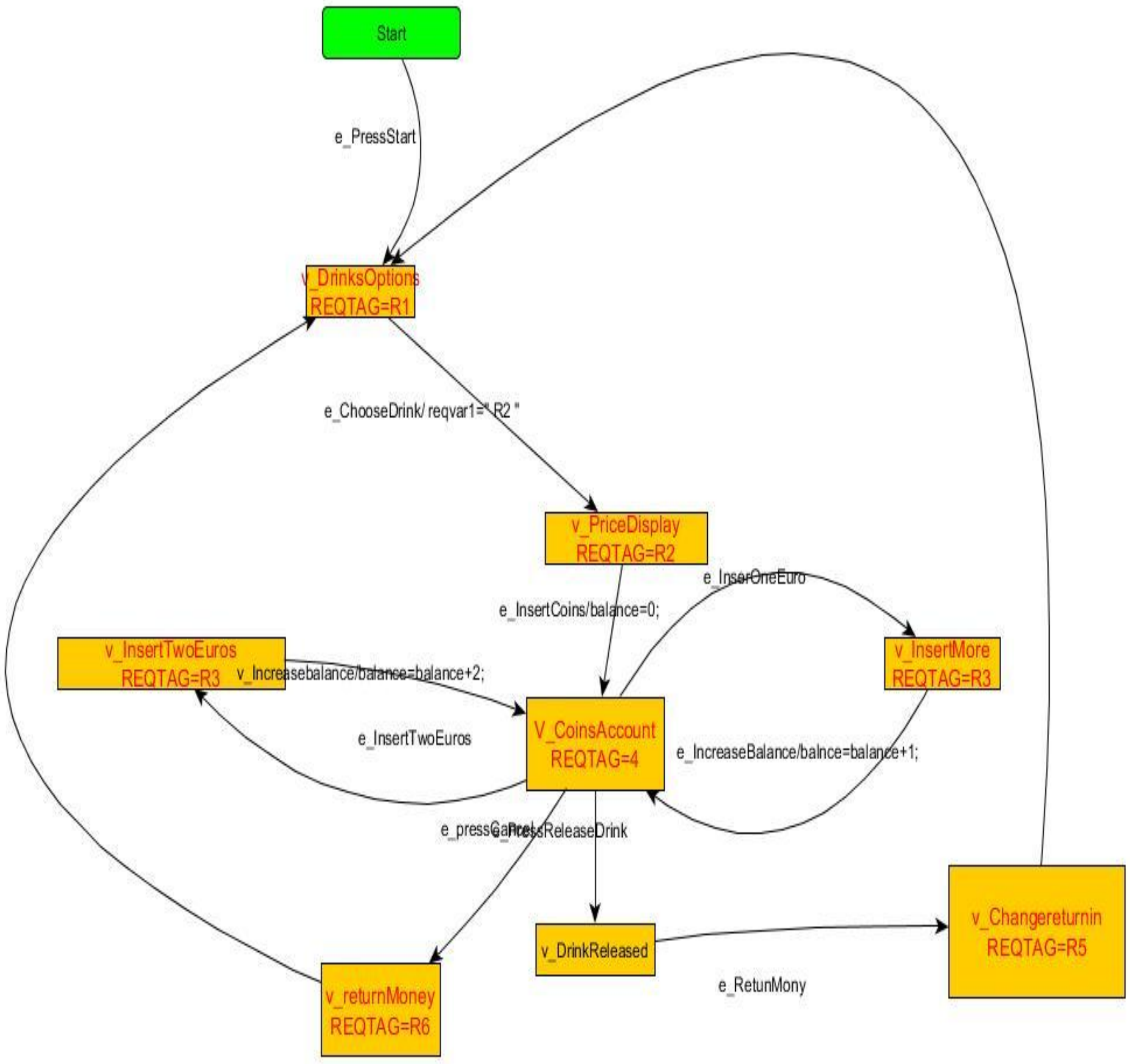


Figure 9. Tagging requirements in a transition based model.

Figure 9 is an extended FSM for drinks vending machine requirements. Requirements Ids are explicitly annotated to in states and transitions. For instance, R2 assigned with vertex (state) v*_pricedisplay;* R2 refers to requirements specification R2 is "The customer selects a drink from the list then the price is display". In early modelling phases it is easier to define requirements deficiencies and incompleteness by directly mapping requirements ids and the mode.

Requirements tagging is relying on modelling notation that had been adopted. In the previous example they are connected to EFSM states and transitions. Code fragment 5 is LEIRIOS Test Generator (LTG/B) tag notation within the B machines for R4.

```
/*@BEGIN_REQ: R4 @*/

IF (Balance >Price) ||

dispencse (drink) ||

/*@END_REQ: R4 @*/
```

Code fragment 5. Tagging requirement 4 in LEIRIOS

4.3.2. Requirements as test generation criteria

The test suite is generated based on requirements. There are two approaches to obtain test cases from requirements. Firstly, design test to target a specific requirement. Secondly, test generation based on requirements coverage ratio.

4.3.3. Test for Requirements

Test for requirements approach targets testing for a specific requirement or a collection of requirements. In the case that the existence of specific features are essential, this approach can generate test cases to assure the quality and persistence of them. In Figure 9 if the test objective is to ensure the machine allows the cancellation and money return, then the test objective is R6. Consider the following line of code:

```
RandomPathGenerator (new ReachedRequirement("R6")
```

This line of code is targeting testing R6 in DVM example. The test generator is randomly running till the stopping condition is fulfilled; the stopping condition is to ensure that the requirement R6 is passed. In this example, the generator neglects other requirements, states and transitions, this means it may or may not tests other requirements. The test statistics are shown below

```
Coverage Edges: 15/27 => 55%

Coverage Vertices: 7/9 => 77%

Unvisited Edges:  12

Unvisited Vertices: 2

Test sequence length:  23

 [R3, R4, R2, R1, R6]

PASSED
```

Figure 10. Testing output statistics coverage ratio generated by GraphWalker

The statistics in figure 10, shows that the test generator only covered 55 percent of edges, and only visited seven out of nine states and test sequence generated was 23.The remarkable point is that the requirements statistics here requirements R3,4,2, are passed and when the test reached R6 the targeted requirement the generator stopped.

4.3.4. Requirements coverage ratio

The test is fulfilled if requirements targeted coverage percentage is reached. For example, the tester may target 5 out of 6 requirements to be verified. This approach is applicable if the tester is testing for requirements with low priority. Requirements priorities are calculated according to the following formula by Wiegers [2003]:

$$\text{Priority} = \frac{\text{value \%}}{(\text{cost \%} * \text{cost weight}) + (\text{risk \%} * \text{risk weight})}$$

4.3.5. Requirements traceability matrix

Traceability matrix links requirements to test cases. In MBT, test cases are generated automatically from the model. Meanwhile, the model includes requirements identifiers which links informal requirements to the model. Practically, MBT traceability matrix maps requirements identifiers to test cases.

Chapter 5
A Case Study- Puzzle Game for Android

5.1. System under test

The SUT is to an Android game with puzzles related to computer science, logical reasoning and/ or mathematics. The game consists of four puzzles types. The game is supposed to introduce the player to the world of computer science in an interesting and playful manner. The entire game is undergone to MBT, unless the testing of two puzzles was adapted to the suggested method.

5.2. Testing framework tools

- GraphWalker

GraphWalker is an open source tool for generating offline and online test sequences from FSM/EFSM models [Graphwalker.org]. GraphWalker proved efficiency in real-world projects. GraphWalker utilizes yEd to deliver FSM models.

- Android ChimpChat

An android library provides device accessibility. It is used as adaption library between abstract test cases and the SUT. ChimpChat is a host-side library that provides an API for communication with an instance of an android device. The communication is performed over and android debug bridge ADB.

- TestNG

It is an open source testing framework, originally adopted from JUnite 3, with new features. It is used in test execution phase, to direct and manage the test cases.

5.3. Testing procedures

Testing method is MBT five phases, together with the   proposed modifications suggested in the prior chapter.

5.3.1. Modelling

The first phase in the process is to formalize the requirements. The requirements for a number grid puzzle are:

R1: The sum of numbers in every row must be equal to 15, the sum of numbers in every column must be equal to 15, and the sum of numbers in two diagonals must be equal to 15

R2: When you fill in all fields with numbers, push the button Check. If your solution is correct, you will be switched to the map to choose the next puzzle to solve; if your solution is not correct, all fields will be cleared.

Because the number of the requirements above is limited, for clarification and demonstration, I will assume, an entering of a number in each grid is a separate requirement, so the number of requirements will increase.

The GraphWalker utilizes yEd for modeling the requirements into FSM/EFSM format. The states are named vertices, while transitions called edges. The edges assigned-when EFSM mode is enabled - with guards and statements.

GraphWalker is a transition based tool. The prior chapter illustrated the transitions-based tools uses keywords to tag requirements on transitions and states. GraphWalker is not an exception. GraphWalker's keyword REQTAG used to explicitly mention requirements in the model. Figure 11 represents a FSM model for number grid puzzle requirements.

There are two types of requirements tags, i.e., static and variable requirements. The static requirements are simply a string representing an Id for a requirement; if it is reachable by the test engine then it is validated as passed otherwise it could not be covered with a test cases. For instance REQTAG = req1, req2.

Variable requirements are variables that can carry different values for the parsed requirements. It is a simple processing of requirements Ids. For example, (${reqtag1} can be assigned unique Ids depending on the test generator.

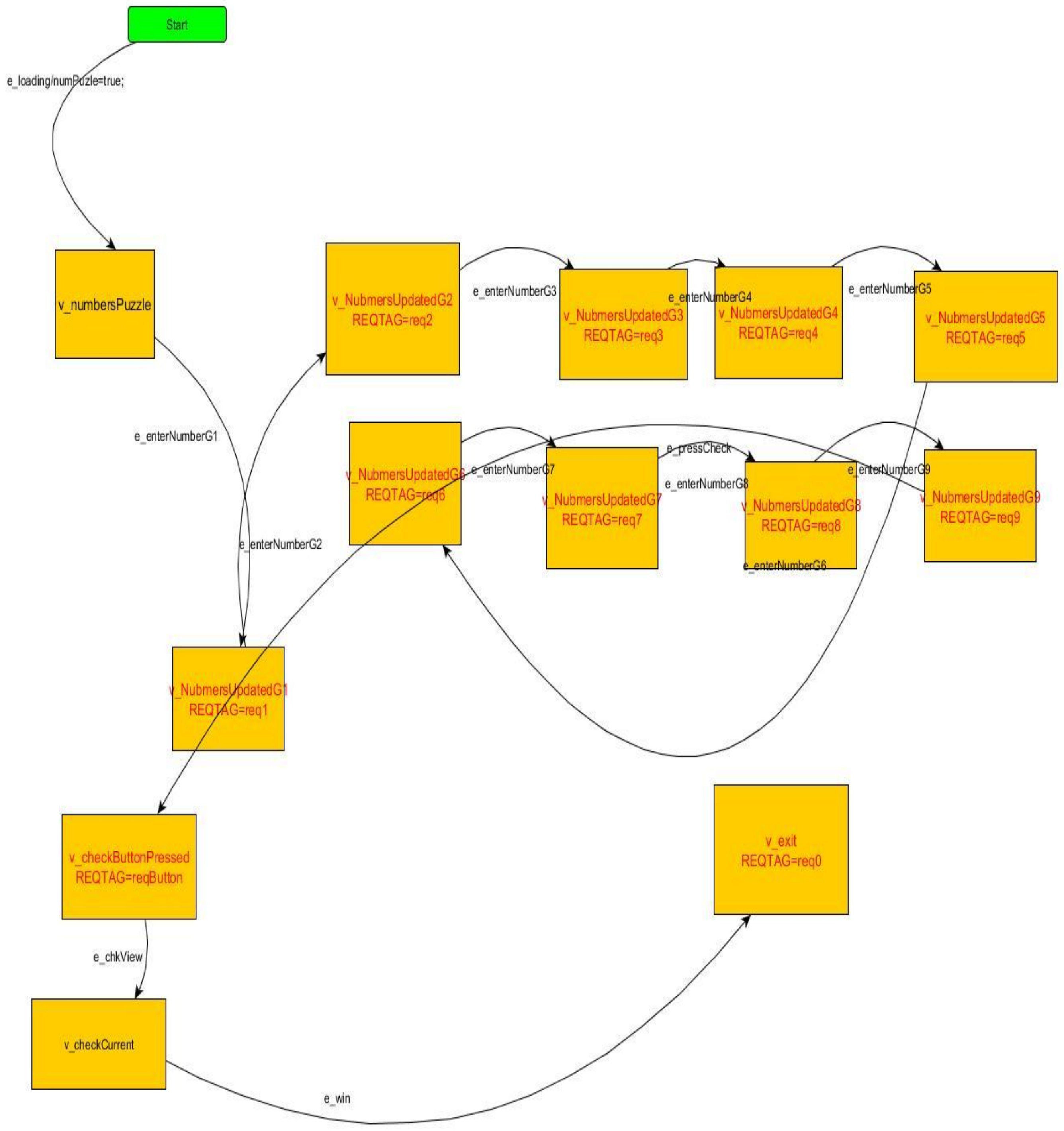


Figure 11. Number grid puzzle model

5.3.2. Test suit generation

GraphWalker supports various test generation criteria. Since the models are in FSM or EFSM format, the test generation criteria are concerned with traversing through states and transitions. For instance, A* algorithm is used to traverse the graph. The generators have stop conditions, for example, the generator stop when it reaches a specific vertex or an edge.

A remarkable stop condition for the fall in the scope of the thesis is requirements as a stop condition. They are two conditions concerning requirements as test generation criteria. First explicitly defines specific requirements as test target, the test stop when it reaches these requirements. This condition is effective in validating critical requirements. The second condition is requirements' coverage as test generation criteria, Requirement coverage must be between 0 and 100; it is efficient to target a specific percentage of requirements to be covered by the testing process, this may strengthen the acceptance testing or smoke testing. But the tester should be careful when using this condition in critical systems testing, since if the generator reaches the defined percentage, it will stop. This implicitly means there is a possibility not to cover all requirements.

In the number grid puzzle, the ***new*** *ReachedRequirement("req0")* is targeting a specific requirement req0.

The generator ***RequirementCoverage*** *(1)* targets 100 percent coverage for the requirements. This means the test only stops when all requirements are covered.

5.3.3 Test harness

This phase is converting the abstract test cases to an executable tests. An android library chimpchat implements interaction between the SUT and the test cases.

```
device.shell("am start -n com.sis.uta.puzzleGame/.MainActivity")  ;
public void e_enterNumberG2() {

device.touch(444,167 , TouchPressType.DOWN_AND_UP) ;

device.touch(760,406 , TouchPressType.DOWN_AND_UP) ;
device.type(getRanNum());

device.press(PhysicalButton.BACK, TouchPressType.DOWN_AND_UP);
System.out.println("Enter A Number In Grid 2");
}
```

Code fraction 6. Adapting the abstract test cases.

In code fraction 6, the first line load the SUT, the lines from 2 to the end are an implementation for an edge -transition- in the model, it uses the absolute resolution for the interaction.

As test cases are generated they promptly executed. The online testing mode is employed.

5.3.4 Test analysis

If the test is targeting a specific requirement, then the generator will ensure that the requirement must be covered by a test case. In code fragment 7, the test covers 50 percent of the requirements, in the model.

```
ts =new TestSucess(file, true, new RandomPathGenerator(new
RequirementCoverage(.5) ), false);
```

Code fraction 7. Covering 50 percent of requirements

This test generator a coverage rate of 50 percent used to generate a test suit. The number puzzle grid model has 11 requirements, the generator randomly selects 6 out of 11 requirements. Through using the same technique, we can cover all requirements. Figure 10 shows the testing output statistics for the code fraction 7, where the test covered half of the requirements.

```
NO of Requirements 11
Requirementes Covered

Requirements  req6   Passed
Requirements  req5   Passed
Requirements  req4   Passed
Requirements  req3   Passed
Requirements  req2   Passed
Requirements  req1   Passed
Statistics for multiple models
Statistics for NumbersGridPuzzle:
Coverage Edges: 11/28 => 39%
Coverage Vertices: 8/14 => 57%
Unvisited Edges:  17
Unvisited Vertices: 6
Test sequence length:  14
Coverage Edges: 11/28 => 39%
Coverage Vertices: 8/14 => 57%
Unvisited Edges:  17
Unvisited Vertices: 6
Test sequence length:  14
Sucess Model Done
PASSED: testForSuccess

===============================================
    Default test
                     Tests run: 1, Failures: 0, Skips: 0
```

Figure 12. Test output covering 50 percent of the requirements

In case the testing target is a specific requirement for example, in code fragment 8 the test target is req0, if the target fulfilled by reaching the req0, then the test stops, whatever requirements have been tested.

```
ts =new TestSucess(file, true, new  RandomPathGenerator(new ReachedRequirement("req0") ),
false);
```

Code fragment 8. Targeting req0

Figure 13 shows the test output for the code fragment 8. The last requirement covered was req0, thereafter, the generator exits.

```
O of Requirements 11
Requirementes Covered

Requirements  reqButton    Passed
Requirements  req9   Passed
Requirements  req8   Passed
Requirements  req7   Passed
Requirements  req6   Passed
Requirements  req5   Passed
Requirements  req4   Passed
Requirements  req3   Passed
Requirements  req2   Passed
Requirements  req1   Passed
Requirements  req0   Passed
Statistics for multiple models

Statistics for NumbersGridPuzzle:
Coverage Edges: 21/28 => 75%
Coverage Vertices: 14/14 => 100%
Unvisited Edges:  7
Unvisited Vertices: 0
Test sequence length:  31
Coverage Edges: 21/28 => 75%
Coverage Vertices: 14/14 => 100%
Unvisited Edges:  7
Unvisited Vertices: 0
Test sequence length:  31
Sucess Model Done
                      PASSED: testForSuccess
```

Figure 13. Test output for requirement req0

The outcomes of the testing process are:

- TestNG report
- Log File for each test case the log file shows the test sequences (scenario)
- Test models statistics

5.4. Case discussion

The number of test cases generated from the models is huge, this ensures the stability of the product, but still to the testing approach prone Dijkstra law, it does not prove the absence of errors but discover them. The test cases, which are applied proved that, the product covered the customer's requirements. However, the testing approach, could not discover deficiencies as GUI bugs.

On the technical level, there were difficulties with the adoption of the MBT. For example, in the adaption phase it is not quite efficient to use the absolute resolution to access the SUT. There are libraries that can access the SUT graphical interface directly like Robotium or Espresso. However, the game was developed using LIBGDX library that is not compatible with such libraries.

Chapter 6
Conclusions, and Future work

6.1. Limitations

Traceability in MBT encounters drawbacks and limitations originated from MBT characteristics.

MBT is designed for testing functional requirements. The models are representing the behavioral aspects of the SUT. Non-functional requirements cannot be traced to any specific code module. Thus, NFR cannot be traced to test cases.

The suggested traceability method in MBT is restricted to requirements-test suit only. Several software development artifacts are not covered in the MBT traceability model. For instance, design models and coding modules are not involved in the process. Even, if a Model Driven Development (MDD) approach is used, still the models used in testing are different from ones used in design and then implementations, two separate. However, a key point in MBT traceability if specific requirement is tested in the SUT, this means it is implemented and designed already, because the testing is the last phase in the process, but still there is no linkage between different artifacts in the development process.

In practice, there are difficulties in adopting MBT [Robbinson, 2003]. The connection between requirements tools and MBT generators is essential to fully automate and link the requirements to the models. There are few trials for example, Simulink and Conformiq in general.

.

6.2. Conclusions

MBT is a trend in software testing automation. It is ideally investing requirements specifications to automatically generate test suit. Since, MBT is an automation of specification based testing approach, it is a breakthrough to automatically trace requirements to test suit. To achieve a transparent method to trace requirements to test suit, the thesis deduces the following heuristic

1. Modelling
    i. Requirements tagging
2. Test suit generation
    i. Requirements as test generation criteria (partially or fully target requirements)

3. Adaption
4. Execution
5. Analysis
    i. Requirements Coverage ratio
    ii. Requirements Traceability matrix

The proposed method is an adaption of general MBT process, to explicitly manipulate requirements in testing process. The requirements are annotated in the model as tags and linked to requirements specifications documents. Thereafter, the test generation criteria basically requirements, either targeting specific requirements or randomly target the entire requirements pool. In the last phase, the testing statistics concentrate on the requirements coverage.

Despite the method proved efficiency in handling the case study, still there a necessity to combine it with other testing approaches like code-based, to cover the entire development life cycle.

One major obstacle in adopting MBT to trace requirements is its limitations to handle nonfunctional requirements. This limitation originates from the nature of MBT that targeting functional requirements which are prone to modelling.

6.3. Future work

The thesis handled traceability in MBT from straightforward procedural perspective. There is a necessity to manipulate other aspects like effectiveness of the concept within the exhaustive information systems rather than the real time or critical systems, in other words, how far the traceability can be achieved in complex systems etc. enterprise applications.

MBT itself still have open questions, regarding the nature of the approach, like dealing with non-functional requirements and complexity of the modelling phase that requires special skills and training.

MBT requires additional support on tools' level. Despite the existence of a commercially successful tools like Qtronic, still there is a lack of a tool that cover the entire process of requirements and MBT.

References

[Abbors et al., 2009] Abbors, F., Truscan, D., and Lilius, J., Tracing Requirements in a Model-Based Testing Approach, In: *Proc. of the First International Conference on Advances in System Testing and Validation Lifecycle, 2009. VALID '09. 123-128.*

[Agarwal et al., 2010] Agarwal, B.B., S.P. Tayal and M. Gupta, *Software Engineering and Testing: An Introduction*, 1 ed., Hingham, Jones and Bartlett Learning, 2010.

[Arian5, 1996] ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, http://sspg1.bnsc.rl.ac.uk/Share/ISTP/ariane5r.htm , Paris, 1996 July.

[Bender, 2009] Bender RBT Inc., "Requirements Based Testing Process Overview", 2009.

[Blackburn et al., 2005] Blackburn, M.; Busser, R., Nauman, A., and Morgan, T., "Life cycle integration use of model-based testing tools In: *Proc. of the 24th Digital Avionics Systems Conference, 2005. DASC 2005.*

[Boden et al., 2005] Boden, M., Busser, D., Blackburn, M., and Nauman, A., Extending Simulink Models With Natural Relations To Improve Automated Model-Based Testing, In: *Proc. of sew, 29th Annual Workshop on Software Engineering, IEEE/NASA, pp.325-332, 2005*

[Boehm, 1979]. Boehm, B., Guidelines for Verifying and Validating Software Requirements and Design Specifications," Euro IFIP 79, North Holland (1979), p. 711-719

[Bouquet et al, 2005] Bouquet, F., Jaffuel, E., Legeard, B., Peureux, F., and Utting, M., 2005. Requirements traceability in automated test generation: application to smart card software validation. *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1-7.

[Conformiq, 2009] Conformiq Qtronic User Manual (2009) Available: http://www.conformiq.com/downloads/Qtronic2xManual.pdf

[Craig and Jaskiel, 2002] Craig, D., Jaskiel, P., *Systematic Software Testing,* Norwood, MA, USA: Artech House, May 2002.
[Dalal et al., 1999] Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C.M., Patton, G.C., Horowitz, B.M., "Model-based testing in practice,*" In: Proc. of 1999 International Conference on Software Engineering, 1999, 285-294.*

[Dijkstra et al., 1972] Dijkstra, W., *Notes on structured programming*. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, Structured Programming. Academic Press, London, 1972.

[El-Far and Whittaker, 2001] El-Far, K., Whittaker, A., Model-based Software Testing, In: J.J. Marciniak (ed.), Encyclopedia of Software Engineering, Wiley, 2001.

[Gotel and Finkelstein, 1994] Gotel, O. C Z, Finkelstein, A., "An analysis of the requirements traceability problem," *In: Proc. of the First International Conference Requirements Engineering, 1994, 94-101.*

[Gudmundsson et al., 2013] Gudmundsson, V., Schulze, C., Ganesan, D., Lindvall, M., and Wiegand, R., "An initial evaluation of model-based testing," In: *Proc. of Software Reliability Engineering Workshops 2013, ISSREW '13, IEEE International Symposium, 4-7.*

[Harel, 1987] David Harel, Statecharts: A visual formalism for complex systems. In: *Science Computer Programming Vol.* **8**, 1987, 231-274.

[Huang, 2009] Cleland-Huang, J.; Hayes, J.H.; Domel, J. M., "Model-based traceability," In: *Proc. of Traceability in Emerging Forms of Software Engineering, Workshop 2009. TEFSE '09, 6-10.*

[Huang, 2009] J. C. Huang, *Software Error Detection through Testing and Analysis,* June 2009.

[IEEE Glossary, 1990] IEEE *Standard Glossary of Software Engineering Terminology,*" IEEE Std. 610.12-1990, 1-84.

[ISO, 2010] International Organization for standardization. Systems and software engineering, vocabulary *ISO/IEC/IEEE 24765:2010.*

[Laplante and Ovaska, 2012] Laplante, P., Ovaska, S., *Real-Time Systems Design and Analysis: Tools for the Practitioner*, 4 ed., Wiley-IEEE Press, 2012.

[Lutowski, 2005] Rick Lutowski, R., *Software Requirements: Encapsulation, Quality, and Reuse*, 1 ed., Auerbach Publications, 2005.

[Mathur and Malik, 2010] Mathur, S., Malik, S. "Advancements in the V-Model". *International Journal of Computer Applications* 1(12), 2010, 29–34.

[Myers et al., 2011] Glenford, M., Sandler, C., and Badgett, T., *The Art of Software Testing*, 3 ed., Wiley & Sons, December 2011.

[Naslavsky et al., 2007] Naslavsky, L., Ziv, H., and Richardson, J., Towards Traceability of Model-Based Testing Artifacts. In: *Proceedings of the 3rd international workshop on Advances in model-based testing* (A-MOST '07). ACM, New York, NY, USA, 105-114.

[Nyman, 2001] Nyman, N., In Defense of Monkey Testing. Test Automation SIG Group, 2001.

[Pezzè and Young, 2007] Pezzè, M., Young, M., *Software Testing and Analysis-Process, Principles, and Techniques*, 1 ed., John Wiley & Sons, April, 2007.

[Robinson, 1999] Robinson, H., "Graph Theory Techniques in Model-Based Testing", *In: Proc. of International Conference on Testing Computer Software, 1999.*

[Robinson, 2003] Robinson, H., "Obstacles and opportunities for model-based testing in an industrial software environment," *In: Proc. of 1st European Conference on Model-Driven Software Engineering (2003), Nuremberg, Germany, Dec. 2003, 118–127.*

[Shafique and Labiche, 2010] Shafique, M., Labiche, Y.A., Systematic review of model basedtesting tools. Carleton University, Department of Systems and Computer Engineering. Technical Report (SCE-10-04) (2010).

[Sommerville, 2010] Sommerville, I., *Software engineering*, 9 ed., Addison-Wesley, March 2010.

[Spec#, 2005] Spec#tool(public release, March 2005) http://research.microsoft.com/specsharp

[Prowell, 2005] Prowell, J., "Using Markov Chain Usage Models to Test Complex Systems", In*: Proc. of the 46th Hawaii International Conference on System Sciences, 2005, HICSS*, '05, 318.

[Tassey, 2002] Tassey, G., *The Economic Impacts of Inadequate Infrastructure for Software Testing*, May2002.

[Utting and Legeard, 2006] Utting, M., Legeard, B., *Practical Model Based-Testing*, Morgan-Kaufmann, 2006.

[Utting, 2005] Utting, M., The Role of Model-Based Testing. In *Verified Software: Theories, Tools, Experiments*, Bertrand Meyer and Jim Woodcock (Eds.). *Lecture Notes In Computer Science*, Vol. **4171**. Springer-Verlag, Berlin, Heidelberg. 2005, 510-517.

[Veanes et al., 2008] Veanes, V., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., and Nachmanson, L., Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, in *Formal Methods and Testing*, vol. **4949**, Springer Verlag, *Lecture Notes in Computer Science*, 2008, 39-76.

[Wiegers, 2003] Wiegers, K.W.,"*Software Requirements*", 2nd, ed.Microsoft Press, 2003.

[Zhang, 2007] Zhang, Z., Effective Requirements Development - A Comparison of Requirements Elicitation Techniques. In*: Proc. of the 15th Conference on Software Quality Management: 2007, SQM '07, 225–240.*

Appendix

This is a sample code for the cade study introduced in chapter 5. The first class is the main class responsible for handling model and specify test generation criteria. The second class is the adaption phase where the interaction between the test cases and the system under test takes place. The system under test was implemented with LibGDX library that caused difficulty in using libraries like Robotium, so the interaction was in absolute resolution.

```java
package com.dawoud.uta;
import org.testng.Assert;
import org.testng.annotations.Test;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.net.URISyntaxException;
import java.util.Set;
import org.graphwalker.ModelBasedTesting;
import org.graphwalker.StatisticsManager;
import org.graphwalker.conditions.EdgeCoverage;
import org.graphwalker.conditions.ReachedRequirement;
import org.graphwalker.conditions.ReachedVertex;
import org.graphwalker.conditions.RequirementCoverage;
import org.graphwalker.conditions.TimeDuration;
import org.graphwalker.conditions.VertexCoverage;
import org.graphwalker.exceptions.StopConditionException;
import org.graphwalker.generators.A_StarPathGenerator;
import org.graphwalker.generators.CombinedPathGenerator;
import org.graphwalker.generators.RandomPathGenerator;
import org.graphwalker.generators.RequirementsGenerator;
import org.graphwalker.machines.FiniteStateMachine;
import org.graphwalker.multipleModels.ModelHandler;
import org.graphwalker.statistics.RequirementCoverageStatistics;


    public class NewTest {

            TestSucess ts;

            TestNum testNum;

            TestGraphColoring gc;

            TextPuz txtp;
```

```
        PicturePuz pic;

        ModelHandler modelhandler = new ModelHandler();

        PrintWriter testCases ;

        @Test

        public void testForSuccess() throws InterruptedException,
        StopConditionException, URISyntaxException, FileNotFoundException {

                File file = new File("Resources/NumberPuzzleSuc.graphml");

                ts =new TestSucess(file, true, new  RandomPathGenerator(new
                ReachedVertex("v_exit") ),  false);

                modelhandler.add("NumbersGridPuzzle",ts );

                modelhandler.execute("NumbersGridPuzzle");

        Assert.assertTrue(modelhandler.isAllModelsDone(), "Not all models
        are done");

        String actualResult = modelhandler.getStatistics();

        ModelBasedTesting mbt=ts.getMbt();

        mbt.passRequirement(true);

        RequirementCoverageStatistics rcs=  new
        RequirementCoverageStatistics(mbt.getGraph());

        System.out.println( "NO of Requirements "+rcs.getMax());

        FiniteStateMachine mch= mbt.getMachine();

        mbt.populateMachineRequirementHashTable();

        Set<String> reqs=mch.getCoveredRequirements();

        String[] tmp = (String[]) reqs.toArray(new String[0]);

        System.out.println("Requirementes Covered\n");

        for (int i = 0; i < tmp.length; i++) {

                        System.out.println("Requirements \t "+tmp[i]+"\t
                        Passed  ");

        }

                System.out.println(actualResult);

                System.out.println(mbt.getStatisticsString());

                System.out.println("Sucess Model Done");

        }


    @Test

        public void testNumbersGrid() throws InterruptedException,
        StopConditionException, URISyntaxException {

        System.out.println("new Test >>>>>>>");

        File file = new File("d://MBT/NumPuz.graphml");

        System.out.println("new Test >>>>>>>File ");
```

```
            testNum=   new TestNum(file, true, new RandomPathGenerator(new
            ReachedVertex("v_exit")), false);

            testNum.setDevice(ts.getDevice());

            ModelHandler mh =new ModelHandler();

            mh.add("RandomTest", testNum);

            System.out.println("ADB IS UP ");

            mh.execute("RandomTest");

            Assert.assertTrue(mh.isAllModelsDone(), "Not all models are done");

            String actualResult = mh.getStatistics();

            System.out.println(actualResult);} }
```

```
package com.dawoud.uta;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.util.Collection;
import java.util.Iterator;
import java.util.Random;
import com.android.chimpchat.ChimpManager;
import com.android.chimpchat.adb.AdbBackend;
import com.android.chimpchat.core.IChimpDevice;
import com.android.chimpchat.core.IChimpView;
import com.android.chimpchat.core.ISelector;
import com.android.chimpchat.core.PhysicalButton;
import com.android.chimpchat.core.TouchPressType;
import com.android.chimpchat.hierarchyviewer.HierarchyViewer;
import com.android.ddmlib.AndroidDebugBridge;
import org.graphwalker.Util;
import org.graphwalker.exceptions.InvalidDataException;
import org.graphwalker.generators.PathGenerator;

public class TestSucess extends org.graphwalker.multipleModels.ModelAPI {
    AdbBackend ab ;
    IChimpDevice  device ;
    PrintWriter testCases  ;
  public TestSucess(File model, boolean efsm, PathGenerator generator,
boolean weight) {
    super(model, efsm, generator, weight);
    ab = new AdbBackend();
    device = ab.waitForConnection();
    try {
             testCases = new PrintWriter("Resources/TestForSucess.txt", "UTF-
8");
       } catch (FileNotFoundException | UnsupportedEncodingException e) {
             // TODO Auto-generated catch block
             e.printStackTrace();
       }
  }
 /**
```

```
    * This method implements the Edge 'e_check' *   */
   public String getRanNum(){
          Random rand = new Random();
            int pickedNumber = rand.nextInt(10);
            String num =  Integer.toString(pickedNumber) ;

            if (pickedNumber==0){
               pickedNumber = rand.nextInt(10);
            }
            else{
               num = Integer.toString(pickedNumber);

            }
          return num;
   }
   public void e_loading() {
          System.out.println("Loading >>In Sucess case ");
          testCases.println("Loading >>In Sucess case ");
 device.shell("am start -n com.sis.uta.puzzleGame/.MainActivity")  ;
 try {
        Thread.sleep(20000);
 } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
 }
 device.touch(400,320 , TouchPressType.DOWN_AND_UP) ;
 try {
        Thread.sleep(3000);
 } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
 }
 device.touch(720,340 , TouchPressType.DOWN_AND_UP) ;
 try {
        Thread.sleep(3000);
 } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
 }
 for (int i = 0 ;i<12;i++){
 device.touch(770,175 , TouchPressType.DOWN) ;
 try {
        Thread.sleep(300);
 } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
 }
 device.touch(781,251 , TouchPressType.UP) ;
 }
 try {
        Thread.sleep(10000);
 } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
 }

 System.out.println("Loading>>>>>>>>>>>>>>>");
 testCases.println("Loading");
 Collection<String> idList= device.getViewIdList();
 Iterator <String>idListIte =idList.iterator();
```

```
while (idListIte.hasNext())       {
System.out.println(idListIte.next());

}
  }
  /**
   * This method implements the Edge 'e_enter'
   *
   */
  public void e_enter() {
         testCases.println("enter E");
  }
  /**
   * This method implements the Edge 'e_enterNumberG1'
   *
   */
  public void e_enterNumberG1() {

         try {
                      Thread.sleep(3000);
              } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
              }
              device.touch(40,205 , TouchPressType.DOWN_AND_UP) ;
              try {
                      Thread.sleep(3000);
              } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
              }
              device.touch(33,302 , TouchPressType.DOWN_AND_UP) ;
              try {
                      Thread.sleep(2000);
              } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
              }
              System.out.println("Enter A Number In Grid 1");
               testCases.println("Enter A Number In Grid 1");

  }
  /**
   * This method implements the Edge 'e_enterNumberG2'
   *
   */
  public void e_enterNumberG2() {
         try {
                      Thread.sleep(3000);
              } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
              }

              device.touch(130,215 , TouchPressType.DOWN_AND_UP) ;

              try {
                      Thread.sleep(3000);
              } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
```

```
                        e.printStackTrace();
                }
                device.touch(195,469 , TouchPressType.DOWN_AND_UP) ;
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                device.touch(150,469 , TouchPressType.DOWN_AND_UP) ;
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                System.out.println("Enter A Number In Grid 2");
                 testCases.println("Enter A Number In Grid 2");
        }
        /**
         * This method implements the Edge 'e_enterNumberG3'
         *
         */
        public void e_enterNumberG3() {
                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                device.touch(240,215 , TouchPressType.DOWN_AND_UP) ;

                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                device.touch(240,355 , TouchPressType.DOWN_AND_UP) ;
                try {
                        Thread.sleep(2000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                System.out.println("Enter A Number In Grid 3");
                testCases.println("Enter A Number In Grid 3");
        }
        /**
         * This method implements the Edge 'e_enterNumberG4'
         *
         */
        public void e_enterNumberG4() {
                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
                }

                device.touch(35,310 , TouchPressType.DOWN_AND_UP) ;

                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                device.touch(40,180 , TouchPressType.DOWN_AND_UP) ;
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                System.out.println("Enter A Number In Grid 4");
                testCases.println("Enter A Number In Grid 4");
    }
    /**
     * This method implements the Edge 'e_enterNumberG5'
     *
     */
    public void e_enterNumberG5() {
            try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                device.touch(120,320 , TouchPressType.DOWN_AND_UP) ;

                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                device.touch(135,140 , TouchPressType.DOWN_AND_UP) ;
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                System.out.println("Enter A Number In Grid 5");
                testCases.println("Enter A Number In Grid 5");
    }
    /**
     * This method implements the Edge 'e_enterNumberG6'
     *
     */
    public void e_enterNumberG6() {
        try {
                    Thread.sleep(3000);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }

            device.touch(240,320 , TouchPressType.DOWN_AND_UP) ;

            try {
                    Thread.sleep(3000);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }

            device.touch(240,86 , TouchPressType.DOWN_AND_UP) ;
            try {
                    Thread.sleep(2000);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();}
            System.out.println("Enter A Number In Grid 6");
            testCases.println("Enter A Number In Grid 6");
}


/**
 * This method implements the Edge 'e_enterNumberG7'
 *
 */
public void e_enterNumberG7() {
        try {
                    Thread.sleep(3000);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }

            device.touch(40,400 , TouchPressType.DOWN_AND_UP) ;

            try {
                    Thread.sleep(3000);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }

            device.touch(45,266 , TouchPressType.DOWN_AND_UP) ;
            try {
                    Thread.sleep(2000);
            } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            }

            System.out.println("Enter A Number In Grid 7");
            testCases.println("Enter A Number In Grid 7");
}
```

```
/**
 * This method implements the Edge 'e_enterNumberG8'
 *
 */
public void e_enterNumberG8() {
       try {
                Thread.sleep(3000);
           } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           }

           device.touch(150,400 , TouchPressType.DOWN_AND_UP) ;

           try {
                Thread.sleep(3000);
           } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           }

           device.touch(135,128 , TouchPressType.DOWN_AND_UP) ;
           try {
                Thread.sleep(2000);
           } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           }

           System.out.println("Enter A Number In Grid 8");
           testCases.println("Enter A Number In Grid 8");
}

/**
 * This method implements the Edge 'e_enterNumberG9'
 *
 */
public void e_enterNumberG9() {
       try {
                Thread.sleep(3000);
           } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           }

           device.touch(240,400 , TouchPressType.DOWN_AND_UP) ;

           try {
                Thread.sleep(3000);
           } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           }
           device.touch(245,320 , TouchPressType.DOWN_AND_UP) ;
           try {
                Thread.sleep(2000);
           } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
           }
```

```
			System.out.println("Enter A Number In Grid 9");
			testCases.println("Enter A Number In Grid 9");
	}

	/**
	 * This method implements the Edge 'e_pressCheck'
	 *
	 */
	public void e_pressCheck() {
		try {
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			device.touch(560,315 , TouchPressType.DOWN_AND_UP) ;
			try {
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			device.touch(400,265 , TouchPressType.DOWN_AND_UP) ;
			try {
				Thread.sleep(3000);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			System.out.println("Press Ckeck Button ");
			testCases.println("Press Ckeck Button ");
	}

	/**
	 * This method implements the Edge 'e_win'
	 *
	 */
	public void e_win() {

		System.out.println("You WOn!!");
		testCases.println("You WOn!!");
	}

	/**
	 * This method implements the Vertex 'v_NubmersUpdatedG1'
	 *
	 */
	public void v_NubmersUpdatedG1() {
		testCases.println(" v_NubmersUpdatedG1");

	}
	/**
	 * This method implements the Vertex 'v_NubmersUpdatedG2'
	 *
	 */
	public void v_NubmersUpdatedG2() {
		testCases.println("v_NubmersUpdatedG2"); }

	/**
 * This method implements the Vertex 'v_NubmersUpdatedG3'
 *
 */
public void v_NubmersUpdatedG3() {
       testCases.println("v_NubmersUpdatedG3");
}

/**
 * This method implements the Vertex 'v_NubmersUpdatedG4'
 *
 */
public void v_NubmersUpdatedG4() {
       testCases.println(" v_NubmersUpdatedG4");
}
/**
 * This method implements the Vertex 'v_NubmersUpdatedG5'
 *
 */
public void v_NubmersUpdatedG5() {
       testCases.println(" v_NubmersUpdatedG5");

}

/**
 * This method implements the Vertex 'v_NubmersUpdatedG6'
 *
 */
public void v_NubmersUpdatedG6() {
       testCases.println(" v_NubmersUpdatedG6");

}

/**
 * This method implements the Vertex 'v_NubmersUpdatedG7'
 *
 */
public void v_NubmersUpdatedG7() {
       testCases.println(" v_NubmersUpdatedG7");
}

/**
 * This method implements the Vertex 'v_NubmersUpdatedG8'
 *
 */
public void v_NubmersUpdatedG8() {
       testCases.println(" v_NubmersUpdatedG8");

}

/**
 * This method implements the Vertex 'v_NubmersUpdatedG9'
 *
 */
public void v_NubmersUpdatedG9() {
       testCases.println(" v_NubmersUpdatedG9");

}

/**
 * This method implements the Vertex 'v_checkButtonPressed'
```

```
     *
     */
    public void v_checkButtonPressed() {
           testCases.println(" v_NubmersUpdatedG9");

    }

    /**
     * This method implements the Vertex 'v_exit'
     *
     */
    public void v_exit() {
           try {
                      Thread.sleep(3000);
                } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
                }

           device.press(PhysicalButton.HOME, TouchPressType.DOWN_AND_UP);
           try {
                      Thread.sleep(3000);
                } catch (InterruptedException e) {
                      // TODO Auto-generated catch block
                      e.printStackTrace();
                }

           testCases.println(" Exit to Home ");
           testCases.close();

    }

    /**
     * This method implements the Vertex 'v_numbersPuzzle'
     *
     */
    public void v_numbersPuzzle() {
    }

    public void v_checkCurrent() {
    }

    public void e_chkView() {
    }

    public IChimpDevice getDevice(){
           return device;
    }

    public void setDevice(IChimpDevice dev){
           device=dev;     }
}
```