



UPPSALA
UNIVERSITET

IT 20 081

Examensarbete 30 hp
November 2020

Transformation of UML State Machine Diagram into Graph Database to Generate Test Cases

Mohammad El-Musleh

Institutionen för informationsteknologi
Department of Information Technology



UPPSALA
UNIVERSITET

**Teknisk- naturvetenskaplig fakultet
UTH-enheten**

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
<http://www.teknat.uu.se/student>

Abstract

Transformation of UML State Machine Diagram into Graph Database to Generate Test Cases

Mohammad El-Musleh

The manual approach in software testing is considered as expensive, error-prone, and time-consuming activity since it depends highly on the test engineers. As well, the process of software testing requires proper planning and resources to design the test cases. For this reason, any approach that can be used to enhance or automate the current testing process is necessary.

This thesis introduces an approach to transform the Unified Modeling Language (UML) behavioral state machines diagram into a graph database inside Neo4j, a graph database software. Moreover, a framework is proposed that fetches test data from the graph database.

Based on the similarity between the state machine notation and the nodes and edges (with properties) in graph databases, a set of rules for representation is presented in this thesis. Along with a framework based on GRANDstack (full-stack framework), the framework should use the pre-built graph database together with other technologies to generate test cases from the inserted requirements specification.

A proof-of-concept is implemented to demonstrate the proposed framework. By using a dedicated schema, the fetched data is matched with the expected results. The results prove that the transformation method and the proposed framework have a good potential to be developed and evaluated with a realistic test from the industry practice.

Keywords: requirement, test data, UML, state machine diagram, black-box testing, test case generation, Neo4j, GraphQL API

Handledare: Veerappa Kaujageri
Ämnesgranskare: Bengt Jonsson
Examinator: Philipp Rummer
IT 20 081
Tryckt av: Reprocentralen ITC

Acknowledgments

I would like to express my gratitude to the National Electric Vehicle Sweden AB (NEVS) for providing me with this great master thesis opportunity at their headquarters in Trollhättan, Sweden.

I would sincerely thank the chapter leader Camilla Lindeblad from Testing department for her valuable support, guidance and feedback during my thesis work. Also, I would like to extend my gratitude to my supervisor Veerappa Kaujageri at NEVS, for following up and providing me with valuable and constructive suggestions and feedback during my work on this thesis. Also, I would like to thank all the staff at NEVS their willingness to provide me with all the support I needed to make it possible for me to reach the desired results in this thesis work.

I would like to express my deepest gratitude to Bengt Jonsson at Uppsala University for his patient guidance, valuable discussions and useful critique of this thesis.

I would also like to thank Uppsala University for providing me with the right knowledge and tools during my two years master's program in embedded systems.

Last but not least, I would like to thank my family and friends for their enthusiastic encouragement throughout my studies.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Problem Statement	2
1.3	Purpose and Goals	3
1.4	Delimitations	3
2	Background and Theory	4
2.1	Requirements	4
2.2	Software Testing	4
2.3	UML Diagrams	5
2.3.1	State Machine Diagram	6
2.4	Graph Database (Neo4j)	9
2.5	Literature Review	10
3	Methodology	14
4	Approach Implementation	17
4.1	Definitions	17
4.1.1	Transition	17
4.1.2	State with Compartments	18
4.1.3	Initial/Final State	19
4.1.4	Junction Pseudostate	20
4.1.5	Composite State and Entry/Exit Point	20
4.1.6	History Pseudostate	22
4.2	Experimentation	23
4.2.1	Metadata (Sample)	23
4.2.2	Building the Graph Database	25
4.2.3	Schema and Fetching Data	27
4.2.4	Result (Client-Side)	29
5	Discussion and Conclusion	35
5.1	Future Work	36
	References	38
	Appendix A : Cypher Query Used	39
	Appendix B: Prototype of the Tool	40

List of Figures

1	Branching tree diagram of different type of system information . . .	6
2	Notation used in an state machine diagram	7
3	Difference between unstructured transitions and entry/exit points . . .	9
4	Example of Neo4j DB nodes and edge with their properties	10
5	Examples of activity diagram with loop structure (a, and b) and concurrent activities (c) [16]	11
6	The interface of UMLTGF tool proposed in the research paper [21]	12
7	The interface of UTG tool proposed in the research paper [22] . . .	13
8	The test generation methodology, the big image	15
9	GRANDstack combination of technologies that work together . . .	16
10	Example of transition notation and representation in Neo4j	18
11	Example of state with compartments notation and representation in Neo4j	19
12	Example of initial and final state notation and representation in Neo4j	19
13	Example of junction notation and representation in Neo4j	20
14	Example of 'default entry' and 'default exit' and representation in Neo4j	21
15	Example of 'explicit entry' and 'explicit exit' and representation in Neo4j	21
16	Representation of shallow history notation	22
17	Representation of deep history notation	22
18	Sample of UML state machine diagram	23
19	Manually created design that represent the sample diagram in Arrow tool	26
20	Neo4j Graph Visualization that represent the sample diagram . . .	26
21	The main page of the implemented tool	29
22	Inserting requirement page in the implemented tool	30
23	List of the inserted requirements in the implemented tool	31
24	Test case result for requirement 001 in the implemented tool . . .	32
25	Test case result for requirement 002 in the implemented tool . . .	32
26	Test case result for requirement 003 in the implemented tool . . .	33
27	Test case result for requirement 004 in the implemented tool . . .	33
28	A prototype of the main page of the proposed tool	40
29	A prototype of the inserting requirement process to the proposed tool	41
30	A prototype of the generating test cases process to the proposed tool	42

List of Tables

- 1 The list of requirements used in the implementation tool 24
- 2 The list of test cases sample that imitate the manual approach . 24

Listings

1	The graphql API schema used for the sample diagram	27
2	A query used to retrieve data for the sample diagram	28
3	The GraphQL response to the requested query	28
4	Cypher query used to build the graph database that represent the sample diagram	39

List of Abbreviations

API	Application Programming Interface
DBMS	Database Management System
DB	Database
ECU	Electronic Control Unit
KVP	Key-Value Pair
NLP	Natural Language Processing
NL	Natural Language
SUT	System Under Test
UML	Unified Modeling Language

1 Introduction

In this section, the thesis project is briefly introduced into four parts. It comprises a general overview, purpose, goals, and delimitations.

1.1 Overview

In the automotive industry, in order to roll out new products into the market, companies are required to go through different stages of a verification and validation process, or the so-called "V-diagram" (system engineering process) [1]. The products vary from one to another in different aspects such as size and software complexity. In general, each product consists of many electronic control units (ECUs) i.e., embedded systems component. ECUs, embedded systems, are connected together via either wired or wireless communication, and each ECU is used to perform specific electronic features.

The fast-growing technology these days increases customers' demands, and implementing their demands increases the ECU complexity which leads to an increase in the number of requirements for the product system. In addition, to analyze and validate such systems, it requires a huge number of tests that need to be done and this is considered a major challenge.

The purpose of the software testing process is to ensure that the final software product is stable and works according to user needs and requirements to avoid any bad outcomes. However, software testing is a very expensive and time consuming task because it highly depends on test engineers. In addition, in the software development life cycle, the software testing is considered as the most important and critical phase in the process [2]. It accounts for 50% of the software life cycle [3].

Furthermore, the time and effort required to do manual testing becomes more infeasible and error-prone [2] which adds more pressure on organizations to develop a better way of working and implementing more automated and smarter methods. For this reason, many companies nowadays are tending to automation development [4] to provide better results with higher quality in faster processing time than manual testing. Therefore, techniques to automate the testing process become more important, especially for large and complex systems. This step intends to make complex products done in a shorter period of time.

Many researches, from both academia and industry, tried to cover a variety of methods to automate the testing process by automatic generation of the test cases, however, not all proposed methods can be applied to the automotive industry [5]. One of the main reasons is the internal factors which the testing process depends on to design test cases.

The test case design phase is done during the software testing life cycle (STLC). The test cases consist of a test data set that can be generated from

many sources such as source code for the system under test (SUT), requirement specification, or experience of testers. The testers have to select and decide the proper test data set that should be used, and it is a decision making task. In addition to that, selecting the right amount of test data can reduce unnecessary execution time as these same data are given to the software program during the test execution phase.

The main purpose of the software testing is to compare the expected result from the test case with the actual result from the test execution and see whether the test case passes or fails.

1.2 Problem Statement

One of the critical issues is that ECUs are getting more complex and testing is becoming more time-consuming besides error-prone; while the manual process that highly depends on the testers in designing the test cases is still the same.

In some situations in black-box testing, the testers receive the requirements specifications in natural language (NL) along with the Unified Modeling Language (UML) diagrams to design test cases and send them for execution. The UML diagrams are used to express information about the system as they help the testers to define more test data and make more efficient test cases. However, this situation requires defining the proper test data from the UML diagrams and from the requirements specification to design the test cases, keeping in mind that this process should be repeated to all requirements. Also, as the ECU complexity increases the time consumed to complete the process increases.

For clarification, the manual approach used to design test cases (output) is referred to as table format that contains at least the following three columns: pre-condition, action, and post-condition. For each column, a list of input and output signals (test data) is sorted and collected from the UML state machine diagrams along with the requirements specification in NL.

Furthermore, the testers collect and sort the test data after viewing, studying, and analyzing the available requirements specifications. They create the test cases during the test case development phase in STLC as described in Section 2.2. It should be noted that the test engineers don't have any role in formulating or creating neither the UML diagrams nor the requirements. Instead, they just import it (input) from the requirements management tool.

This thesis work will be focusing on the process of generating the test cases in a black-box method, particularly in getting test data from the UML state machine diagram, and proposing a method that can be used to assist the tester in creating test cases as described in Section 3.

1.3 Purpose and Goals

The aim or the purpose of this thesis work is to propose a new method that can be used to reduce time and increase the reliability of the testing processes; particularly for the manual approach used in selecting the proper test data from UML state machine diagram to create test cases. This proposed method should reduce time in creating the test cases and increase the reliability of the software testing processes. More description of the thesis goals is mentioned in the methodology Section 3.

The requirements for the thesis work are:

- Investigate the manual approach used in generating test cases.
- Propose an approach to make the UML state machine diagram accessible as a smart diagram.
- Investigate the software and tools that are needed to fulfill the aim of this thesis topic.
- Propose a method to fetch data from the software.
- Make a demo to give a better understanding of the method.

1.4 Delimitations

The boundaries of the thesis project work are predefined in order to achieve the objectives of the thesis work. The following list shows the delimitations.

- The thesis work will not describe the way of writing the requirements and the preferred boilerplate language format to follow.
- The thesis work will consider only the UML state machine diagram.
- The thesis work will test examples showing and demonstrating ideas about what the requirements can be and not to present the real requirements.
- The developed demo during this thesis work is only proof-of-concept.
- There is no evaluation for neither efficiency nor time between the tool and the manual approach.
- There is no testing coverage method used.
- In the UML state machine diagrams (notation graphically)
 - should not contain ambiguous English words, assuming all descriptions as key-value pair (KVP).
 - 'do' activity must be represented as timer not natural language (NL).
 - action label on transition should not be in NL.
- There is no Object Constraint Language (OCL) used.

2 Background and Theory

This section presents the gathered information which is relevant to the thesis topic to give a thorough and better understanding of the whole topic. The first part comprises an overview of the different types of requirements. The second part briefly delineates the software testing. The third part introduces the UML diagrams and the notation of the state machine diagram, followed by an overview of graph databases. The last part summarises the literature review that is related to this topic.

2.1 Requirements

Requirements are categorized in different levels and types. The primary categories of requirements are 'business requirements' (higher-level), 'user requirements', and 'system requirements' (detailed). Business requirements describe the high-level statements of the objectives or needs of the organization as a whole. User requirements are connected in one way or another to the defined user's actions or functions (services expected from the system and its constraints). System requirements describe deeply in details of user requirements, and they include both functional and non-functional requirements [6].

The requirements are produced at the beginning of the system or software development. They must be formulated and written in a way that the non-technical person can understand them [6] as these requirements are analyzed and used by software engineers and by test engineers. The software engineers use them in implementing the module, and the test engineers use them in testing during the process of software testing life cycle (STLC).

The majority of the requirements are written in natural language (NL). Also, other methods used; such as UML diagrams, provide visual imagery for better understanding and reduce a great amount of requirement engineers' efforts in writing specifications. In general, UML diagrams represent different aspects of system information. The UML diagrams are explained later on in this section.

2.2 Software Testing

The three main software testing methods are 'black-box', 'white-box', and 'gray-box'. In black-box testing, the testers define test cases using the requirement specification with no knowledge of the SUT (user's viewpoint). In white-box testing, the testers define test cases using the source code with full knowledge of the SUT (programmer's viewpoint). In gray-box testing, testers define test cases from the requirement specification like black-box and detailed documents describing like white-box with partial knowledge of the SUT (designer's viewpoint) [7]. In this thesis, the black-box testing method is used.

There are six different phases in the software testing life cycle, which are 'requirements analysis', 'test planning', 'test case development', 'test environment setup', 'test execution', and 'test cycle closure'. The phases are described below.

In the first phase which is the requirements analysis phase, the testing team has to view, study, and analyze the available specifications and requirements [8].

In the second phase which is the test planning phase, the testing team needs to plan the testing process. The outcome of this phase results in documents, such as Effort Estimation and Test Plan [8].

In the third phase which is the test case development phase, the testing team manually designs the test cases for test execution in detail. For instance, information about preconditions, actions (trigger), and post-conditions (expected results) signals are defined and written manually. In addition to that, each requirement must have at least two test cases one positive test (correct inputs) and one negative test (incorrect inputs) to verify the requirement.

In the fourth phase which is the test environment setup phase, the testing team setting up the test environment completely[8].

In the fifth phase which is the test execution phase, the testing team starts executing all the test cases in the selected test-environment and comparing the expected results with the actual results from the program [9]. The result can be either passed if the test case is successfully executed or failed if the test case is failed. Whenever there is a failed execution, the tester reports it to the software development team to fix it and test it again to ensure that it is fixed.

In the seventh phase which is the test cycle closure phase, the testing team will meet, exchange information, give feedback about the process, and analyze testing documents to optimize the testing strategies [8].

2.3 UML Diagrams

UML stands for Unified Modeling Language. UML consists of many types of diagrams branched from two categories: "Structural Diagrams" and "Behavioral Diagrams" [10] to represent the static as well as the dynamic behavior of a system respectively. The structural diagrams (static behavior) is branched to class diagram, component diagram, deployment diagram, object diagram, package diagram, profile diagram, and composite structure diagram. The behavioral diagrams (dynamic behavior) is branched to use case diagram, activity diagram, state machine diagram, sequence diagram, communication diagram, interaction overview diagram, and timing diagram [10]. In this work, the aim is to cover the behavioral diagram, specifically the state machine diagram because of the delimitations and the pressing time. It is covered because it is widely used in industrial practice. Hereby, it does not mean that the other diagrams are less important.

2.3.1 State Machine Diagram

The state machine diagram (also known as state chart diagram, state diagram, or state transition diagram) emphasizes on the event-ordered as discrete behavior. There are similarities between the state machine diagrams and the activity diagrams, but activity diagram notation and usage is a bit different [10]. There are two kinds of state machines to express part of the system: either behavior or protocol and they are referred to as 'behavioral state machines' and 'protocol state machines', respectively [10]. This thesis will cover the behavioral state machines only as system information (non-testable requirements) that branched from the UML diagram as shown in figure 1 below.

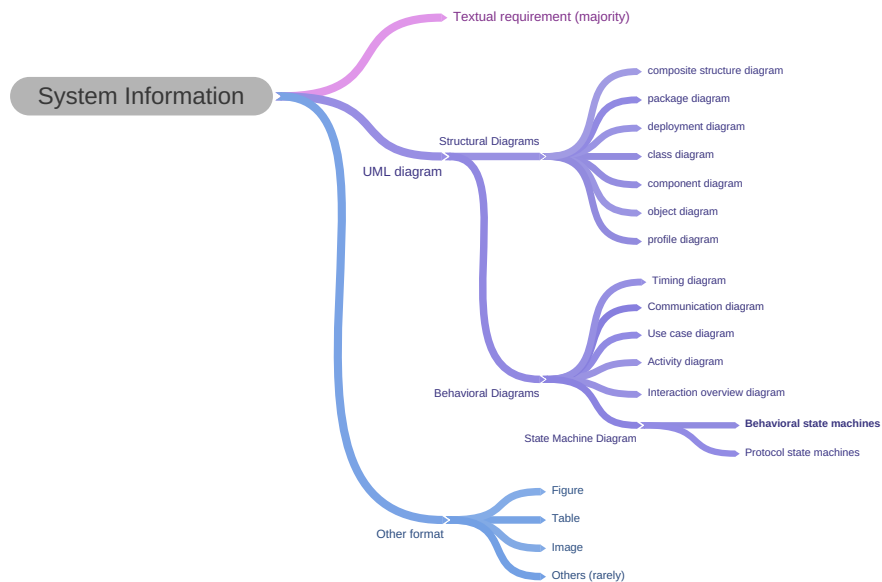


Figure 1: Branching tree diagram of different type of system information

In this thesis, the notations for behavioral state machine used are mentioned according to 'UML Superstructure Specification' booklet [10, p.535-596]. The list of the used UML state machine notations are shown in the following figure 2 below and are followed by brief description. The notations are 'state with compartments', 'initial/final state', 'composite state', 'entry/exit point', 'history pseudostate', 'junction pseudostate' (graphic nodes) and a 'transition' (graphic paths).

Transition notation is shown in the figure 2a. A (simple) transition is a line represents the movement from one state to another, from the source to the target vertex (end with directed arrow). The transition is labeled as 'event [guard] / action' (optional). The event triggers (or activates) the transition. The transition is only triggered once the event occurs and then the guard. The

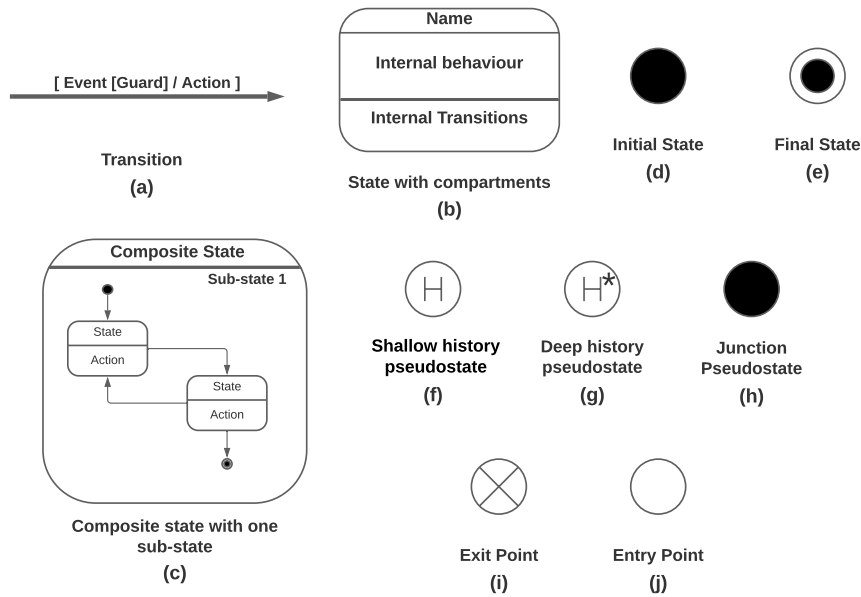


Figure 2: Notation used in an state machine diagram

guard is a Boolean expression and the transition is fired only after the guard is evaluated and the condition is true. If there is no guard on the transition, it is treated as the guard always enabled (true). This result in making the source state becomes inactive while executing its exit activities, afterward the target state becomes active. The action (behavior-expression) is executed once the transition fires, and the way it is written is similar to the entry and exit activity in the state notation.

State with compartments notation is shown in figure 2b. The state is subdivided into three compartments separated by horizontal lines from each other. These compartments are: 'name', 'internal activities', and 'internal transitions'. The 'name' with type string is referred to as the name of the state. It is preferred each state has its own unique name. The 'internal activities' are activity labels as 'entry', 'do', and 'exit'. The entry behavior is always performed first upon entry to the state. The do activity is usually written in NL which describes the operations performed by the state that takes time to complete as long as the state is still active, and if the computation specified by the expression is completed it raises a completion event. The exit behavior is the final step prior to leaving the state as a result of a transition is triggered, and it aborted the do activity even if the computation specified by the expression is not completed. The purpose of the internal transition is similar to self-transition on the state.

Initial state and final state notations are shown in figure 2c, and 2d respectively. The 'initial state' represents the starting of the enclosing state and it is linked to the default state through a single transition. The transition may have action behavior only. On the other hand, the 'final state' indicates the end of

the enclosing state.

Composite state notation is shown in figure 2c. A composite state can be either orthogonal state or not orthogonal state (simple composite state). The orthogonal state means that there is more than one region that works in parallel, and the regions are divided by a horizontal dashed line. The not orthogonal state (simple composite state) means there is only one region that contains a nested state diagram within the region. A simple composite state may have entry and exit activities (do activity is the performed operation in the state and it may start from the initial state).

There is a different transition entry to the composite state. For instance, 'default entry' means the transition terminated on the outer frame of the composite state (boarder-box). Once this occurs, a default entry rule is applied. Another way can be 'explicit entry' (unstructured transition) which means the transition is terminated on one of the sub-state of the composite state. Once this occurs, the entry action in the composite state is executed. Then instead of starting from the default entry rule, the composite state makes the sub-state active and treated as an active state in the diagram and proceed as if it is reached from the initial state through the states that link them. The same rule is applied to all transitions that transitively terminate on the nested sub-state.

Also, there is a different transition that exits the composite state. For instance, a transition source from inside the composite state is terminated at the outside crossing the frame (referred to as explicit exit). Another way to exit the composite state can be when a transition source is attached to the outer frame of the composite state (referred to as default exit). This transition is frequently checked after executing the entry action, and it exits once the transition is triggered. Once any of the referred exit situations occurs, the exit action is executed in a particular order successively followed each other and begins at the innermost active state in the current state configuration, before the action on the transition.

Entry point and exit point notations are shown in figure 2i and 2j respectively. They are used to provide better encapsulation which helps to avoid transition crossing the border of state machine diagram or composite state (unstructured transition), by connecting the outside with the inside through entry point or by exit point in reversed direction. They are semantically equivalent as shown in figure 3 below. Both the entry point and exit point can be placed within the state machine diagram or composite state and outside the border of the state machine diagram or composite state.

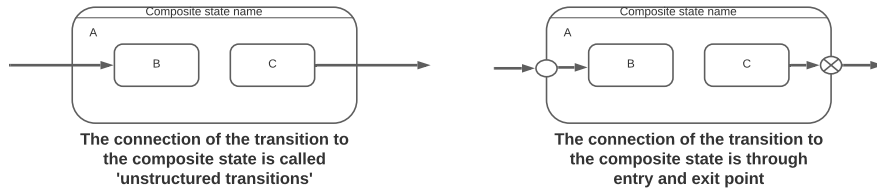


Figure 3: Difference between unstructured transitions and entry/exit points

There are two kinds of history pseudostate which are *shallow history pseudostate* and *deep history pseudostate* notations as shown in figure 2f and 2g respectively. Both shallow and deep history pseudostate can be attached to any composite state but with different representation. If a transition terminates on a shallow history pseudostate, it acts as if the transition is terminated on the first active state directly after the initial state (initial sub-state). On the other hand, if a transition terminates on a deep history pseudostate, it acts as if the transition terminates on a latest active sub-state in the composite state.

Junction pseudostate notation is shown in figure 2. The junction acts as a hub that combines multiple transitions together and defines the transition path from the incoming transitions into a single outgoing transition, according to the fired transition.

2.4 Graph Database (Neo4j)

There are two types of databases, Structured Query Language (SQL) and non-SQL (NoSQL). The graph database or graph-oriented database is a particular type of NoSQL database. Any use case of graph database is represented as a graph structure that consists of 'nodes' and 'edges'. The nodes are circular shapes that represent an entity, while the edges (relationship) are line-shapes that link between two nodes together and define the traversal path by using vertex. Both nodes and edges can contain properties such as a list of Key-Value Pair (KVP). There are many graph database software available today and one of them is Neo4j.

The Neo4j is a database management system (DBMS) and fully ACID (atomicity, consistency, isolation, durability) compliant, and it's the most popular graph database available today and used in production by several companies [11]. The first released to Neo4j version 1.0 was in February 2010 [12] as the first commercial graph databases with ACID guarantees. Some of the features that make Neo4j very popular are Cypher, constant time traversals, flexible, and drivers [13]. Cypher is a query language used with Neo4j. Cypher is simple to use and consider as self-explanatory (declarative) since it is written in visual patterns to represent the nodes and relationships in the graph [14]. In this thesis work the open-source version of Neo4j software is used.

Figure 4 below shows an example of a two-node type person with properties name and age. The two-nodes are linked with an edge. The relationship is 'knows' and it contains only one property which is friendship.

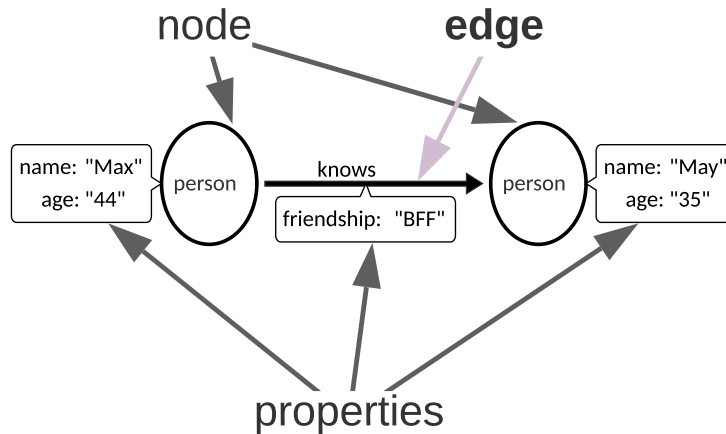


Figure 4: Example of Neo4j DB nodes and edge with their properties

2.5 Literature Review

In general, using a UML diagram to automate the testing process has been a remarkably interesting topic for many years. Among the published works, there are many different methods used for various testing levels from various UML diagrams. Even though UML is widely employed in industry and research, there is little literature that proposed transformation of the UML diagram into a graph.

Kundu, and Samanta [15] are the first to talk about mapping the UML diagram (specifically the activity diagram) to the activity graph. Their article was published in 2009. They proposed a set of rules for conversion procedure (mapping) from activity diagram into activity graph (directed graph) as nodes and edges to represent a construct and the flow in the diagram respectively. Some of the examples of the activity diagram used are shown in the figure 5 below.

In [16], two methods are proposed in describing the mapping implemented for UML class diagram in graph databases where it is either implicit or explicit. In implicit mapping, for each objects class name, class attribute, and association are represented respectively in the graph database as nodes (name is the element type), node properties as KVP, and transition with the same label name. However, in explicit mapping, it is similar to implicit mapping, but the prop-

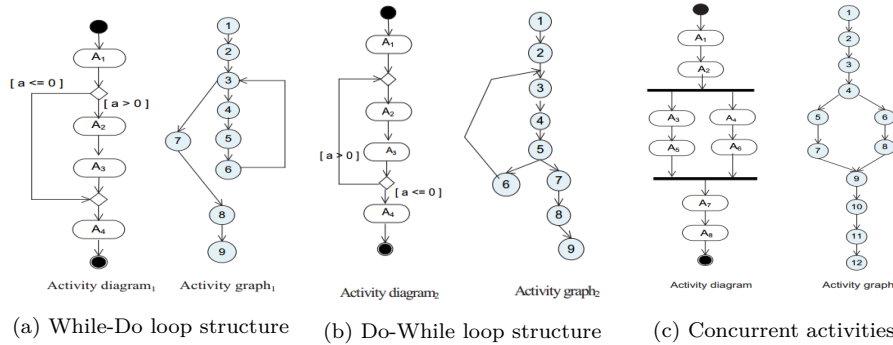


Figure 5: Examples of activity diagram with loop structure (a, and b) and concurrent activities (c) [16]

erties are represented as separated nodes with the transition label 'instanceof' sourced from the parent node.

A literature survey in [17] is conducted to give a better understanding of UML state chart diagram based on testing techniques from 21 studies. The result of the findings shows that it is important to have an intermediate model (such as a graph or a table) integrated (translate) from the UML state chart diagram as another description in order to generate the test cases. In addition, it shows that the most used coverage criteria are path coverage, transition coverage, and state coverage in addition to the weakness of path coverage criterion in loops that lead to loss of path.

In [18], the researchers proposed a technique to generate test cases automatically using the UML state machine diagram. They proposed criteria to transform the state machine diagram into a finite state machine (FSM) and use object constrained language (OCL) that represent the test conditions. This approach used a breadth-first traversing algorithm to list all possible paths in generating test cases.

In [19], a 'UML to Graph DB' framework is proposed (open-source plugins for Eclipse Papyrus¹). The tool acts as a model-to-model transformation (M2M), by mapping the UML class diagram and the OCL constraints into a graph database and graph queries.

In [20], a framework for generating test data from software specification is proposed. The UML state machine diagram is used as a software specification and for generating test data. A genetic algorithm technique is used in generating test data. The tool applies the transition coverage level on the UML diagram used which means the quality of the test data is defined based on the number of transitions in the diagram it can cover. The result from the experimented tool with 4 different UML state machine diagram shows that the tool gives high coverage for the system without the final state because it chooses only one solution from start to final state which leads to some transition that can

¹<https://eclipse.org/papyrus/>

not be reached (called 'infeasible transition'). Moreover, the tool has a 'looping problem' that occurs if a state has looping action such as self-transition and it requires to reach some attribute before transitioning to the next state.

In [21], the authors, in their booklet, use the UML activity diagram as a gray-box method to derive out of it test scenarios to generate test cases to archive path coverage. In the booklet, five basic paths are defined based on assumptions. In addition to that, the researchers created a 'UMLTGF' tool that takes the UML activity diagram (imported) and processed it. Afterward, the tool generates the test scenarios for it, and once a test scenario is selected, it generates a test case at the bottom of the tool panel and described as the 'Input Sequence', 'Operation Sequence', and 'Output Sequence'. A screenshot of the tool is shown in the figure 6 below.

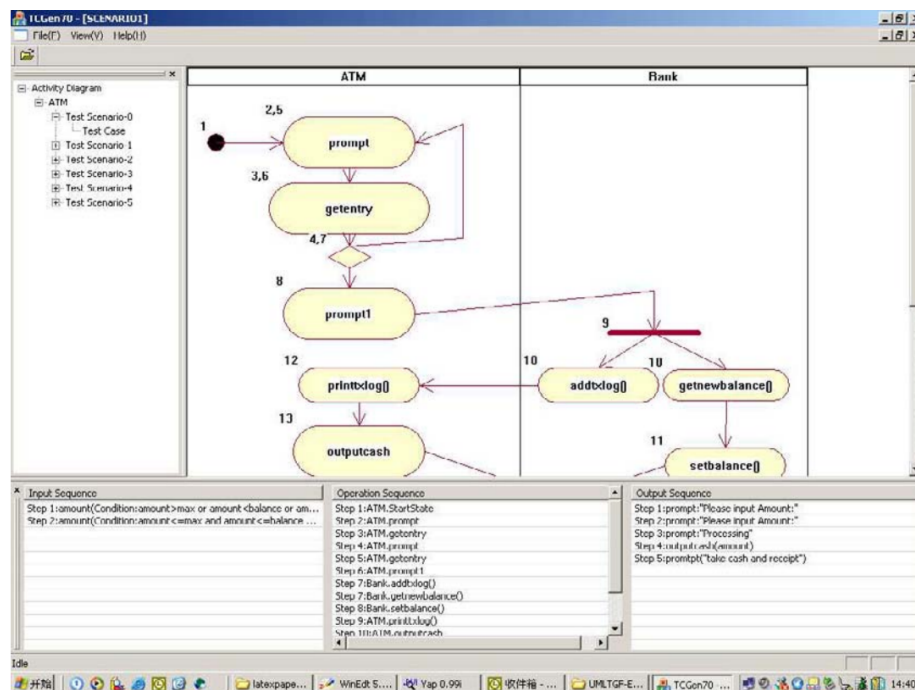
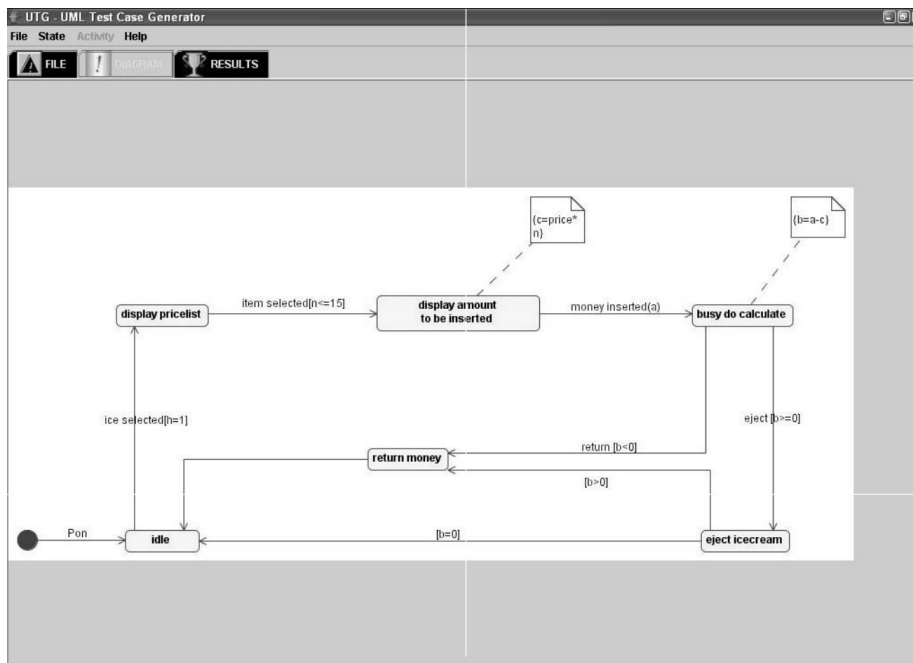


Figure 6: The interface of UMLTGF tool proposed in the research paper [21]

The work presented in [22] is similar to [21] in generating test data based on transition path coverage from the UML diagram. However in [22], a UML state machine diagram is used. The developed tool name is 'UTG'. The tool generates test cases in three main steps which are 'select predicate', 'transform predicate', and 'generate test data'. In addition, the tool allows the user to import an XML file of a state diagram. After importing it, the tool displays the test cases automatically. A screenshot of the tool interface is shown in the figure 7 below.



(a) UTG with an example state machine diagram

The Path followed for this Predicate is :
Pon ice selected item selected[n<=15] money inserted(a) eject [b>=0]

INITIAL STATE	FINAL STATE	TEST DATA
busy do calculate	busy do calculate	a=167, price=14, n=12
busy do calculate	eject icecream	a=168, price=14, n=12

The Path followed for this Predicate is :
Pon ice selected item selected[n<=15] money inserted(a) eject [b>=0] [b>=0]

INITIAL STATE	FINAL STATE	TEST DATA
eject icecream	eject icecream	a=168, price=14, n=12
eject icecream	return money	a=169, price=14, n=12

The Path followed for this Predicate is :
Pon ice selected item selected[n<=15] money inserted(a) eject [b>=0] [b=0]

INITIAL STATE	FINAL STATE	TEST DATA
eject icecream	eject icecream	a=169, price=14, n=12
eject icecream	idle	a=168, price=14, n=12

The Path followed for this Predicate is :
Pon ice selected item selected[n<=15] money inserted(a) return [b<0]

INITIAL STATE	FINAL STATE	TEST DATA
busy do calculate	busy do calculate	a=168, price=14, n=12
busy do calculate	return money	a=167, price=14, n=12

(b) generated test cases corresponding to the example in (a)

Figure 7: The interface of UTG tool proposed in the research paper [22]

3 Methodology

In this thesis work, a proposed approach and framework are used and described to generate test cases in an attempt to enhance the testing process; especially in generating test cases from the UML state machine diagram. The proposed approach is to transform the state machine diagram into a graph database inside Neo4j DBMS, and the framework is the development process that uses the transform graph databases with other technologies to generate test cases.

Article [15] describes the transformation of UML activity diagram to graphical one after applying specific rules. In the same way article [16] describes the transformation of UML class diagram to graphical one after applying specific definition. Accordingly, this thesis work proposes an approach to describe the transformation of UML state machine diagram into graphical one after applying the definitions. The definitions are formulated based on the similarities between the notation in UML state machine and the one described in the literature review, as well as, brainstorming a new design method after going through the tools documentation. The definitions are represented as 'implicit' mapping in the graph database.

The findings of the survey [17] describe the importance of integrating UML diagram into an intermediate model (such as a graph) to generate the test cases and the weakness of path coverage in loop like self-transition on a node that needs to perform a transition after reaching a specific value, together with the development of Neo4j graph database management system and its declarative Cypher syntax. Both of the UML state machine diagrams and Neo4j databases have common similarities in implementation. In addition to the development of GraphQL API that fetches the exact request data from nodes and relationships inside the graph DB, all these together reveal that there is huge potential to create a framework based on this technology to generate test cases.

The proposed framework is based on the black-box testing method in creating test cases since it doesn't require any knowledge or collect any test data from the SUT. The desired tool should convert the UML diagram into a smart-graph that can be identified and retrieved. For this purpose, a graph database is built according to the definition assumption used to transform the UML diagram to graph DB. Furthermore, the desired tool should take the requirement specification in NL as input, and it should generate test cases for the inserted requirement that matches the manual approach (output). The figure 8 below illustrates the proposed thesis work and the relation between them. In the same figure, the dashed line in orange represents the proposed approach and the dashed line in dark blue represents the proposed framework.

In this thesis, there will be no testing coverage criterion used (such as using path coverage). However, the method used to generate the test cases for the proof-of-concept is by using a dedicated GraphQL API schema to fetch data. This is will be taken into consideration in future work.

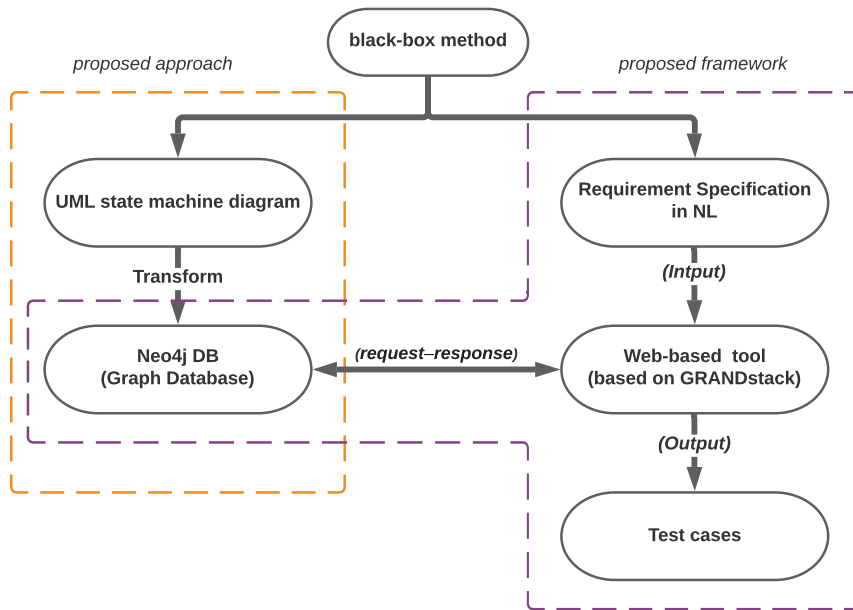


Figure 8: The test generation methodology, the big image

The process of building the graph in the database out of the UML state machine diagram is done manually. There are two ways to interact with the tool. It is either by 'data scientist' or 'end-user'. The 'data scientist' transforms manually the UML diagram to Neo4j DB (mutation). While the 'end-user' (tester) inserts the requirement and generates test cases to the selected one.

The tool is implemented as web-based, and there are many other alternatives to fetch data from the database and end up getting the same result. However, the web-based approach is used due to its simplicity and accessibility anywhere. Moreover, it is easier to customize, install, and maintain quickly. Likewise, it gives the advantage of using GRANDstack starter project¹ with essential settings configured.

The GRANDstack is a full-stack framework, but it uses specific technologies (GraphQL API², React³, and Apollo⁴) along with the Neo4j database. The name GRAND is the combination of the first character from each tool name [23]. The back-end consists of GraphQL API and Apollo server and the front-end consists of React and Apollo client.

The figure 9 below shows the process of GRANDstack in the thesis work.

¹<https://github.com/grand-stack/grand-stack-starter>

²A query language for APIs <https://graphql.org/>

³A JavaScript library, <https://reactjs.org/>

⁴<https://www.apollographql.com/>

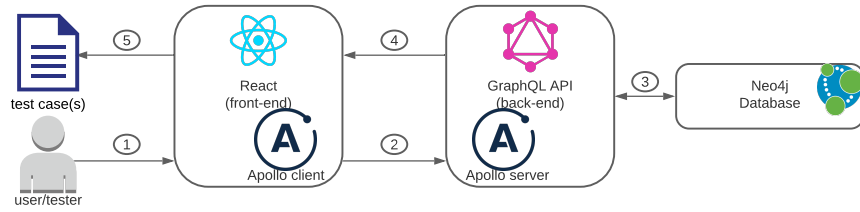


Figure 9: GRANDstack combination of technologies that work together

The process can be described as follows: (1) the end-user inserts a requirement to the tool to get test cases for it, optionally the tool can store the requirement, (2) the requirement specification is split into tokens and by using name-entity-recognition to define the query parameter, the Apollo client sends the query request to the Apollo server, (3) according to the schema defined in the GraphQL API, it sends out of it a Cypher query to the Neo4j DB to fetch the data, (4) the fetched data are sent back from the Apollo server to the Apollo client, and finally (5) the React (ES6 JavaScript) is used to map the retrieved data and sort them in a test case structure and present them to the end-user.

4 Approach Implementation

This section is divided into two parts, the first part explains the definitions used to transform the UML state machine into graph database. In the second part, it is followed by demonstration to a proof-of-concept using a sample to observe the framework.

4.1 Definitions

The proposed interpretation of behavioral state machines (denoted as graphically) into the Neo4j database (denoted as neo4j-design) is chosen based on the similarities in design and on a test using Cypher query to verify the fetching possibility. It should be noted that choosing the right path between the nodes is not only based on the way of representation in neo4j-design but also based on the requirement specification and the schema that uses Cypher query to fetch the data. A representation for UML state machine notation to neo4j-design and other recommendations about the process are mentioned below. The entire process can be covered by 1 type of a relationship ('linked') and 2 types of a node ('state' and 'junction').

The outer frame (border-box) that covers the entire diagram is used to represent part of the system and its name is labeled on the top left corner. The information written inside a comments/note symbol (rectangle with the upper right corner bent) or in the diagram empty space is used to describe additional information to the reader.

4.1.1 Transition

The 'transition' is represented in neo4j-design as an edge (relationship), and the type name (caption) can be any name as long as it's applied to all related transitions and this is essential (for instance 'linked'). Each transition can have two KVP properties. The key properties are 'condition' and 'action', and their values are with type string. The value for the 'condition' property contains the combination of event and guard on the transition label as one condition with 'AND' operator between them since both event and guard are conditions that need to occur before making the transition to the next state. The value for the 'action' property is exactly the same information mentioned in the action label on the transition. In this way, the action signal can be easily placed together with the next state prior to its activities.

The direction of the edge (neo4j-design) is always uni-directional and in the same direction as represented graphically. The rule should be applied to almost all the transition relationships between all types of nodes. This includes when multiple relationships are sourced from one node and terminated on another node.

For instance assuming transition have

```
[x==2 && [z!=x && !t] / tt=false ]
```

then the property on edge (transition) should looks like

```
condition:"x==2 && [z==2 && !t]", action:"tt=false"
```

Figure 10 below illustrated the example.

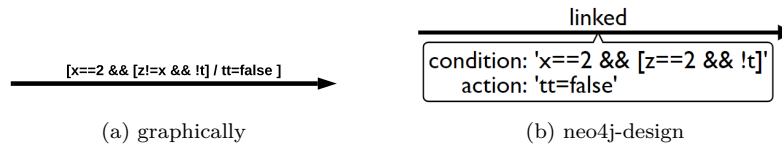


Figure 10: Example of transition notation and representation in Neo4j

4.1.2 State with Compartments

The 'state with compartments' is represented in neo4j-design as a node and the chosen type name is 'state'. Each state can have two KVP properties. The keys properties are 'names' and 'signals', and their values are with type string. The value for the 'name' property contains the exact same state name graphically (mentioned at the top of the state followed by horizontal line) and it's essential; whereas the value for the 'signals' property contains the combination of all associated action (entry and exit) and activity (do) that are mentioned inside the state. Their KVP information is presented in an 'entry', 'do', then 'exit' sequence separating each signal with a semicolon. It should be noted that activity (do) is represented as a timer (wait/delay) according to the time described in the state; otherwise, it requires more effort by the tool to define the timer. The reason behind doing it in such way because in testing all this information are mentioned together and combining them together will make it easier to fetch at once and make them as test steps. The event action in the state should be implemented as self-transition on the state.

For instance assume a state have the following:

```
name: "idle" entry/ xx= "invalid"; yy="on"; zz=0;
do/ wait_ms(200); exit/ vv=True; e="active";
```

then the property on node (state) should looks like

```
name:"idle",
signals:"xx=invalid; yy=on; zz=0; wait_ms(200); vv=True; e=active;"
```

Figure 11 below illustrated the example.

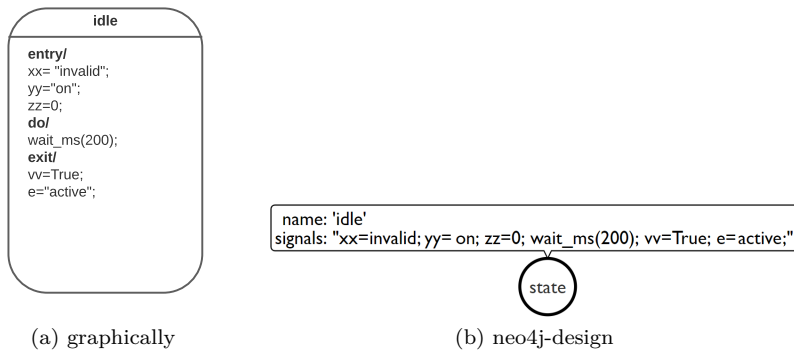


Figure 11: Example of state with compartments notation and representation in Neo4j

4.1.3 Initial/Final State

The 'initial state' and the 'final state' are represented in neo4j-design as a node as same as 'state with compartments' representation which means that it contains the same properties. However, the 'name' property should only be mentioned and it should contain value related to the context of a requirement specification. For instance, the value of the name property for the initial state can be referred to as 'start' followed by the name of the state machine diagram or the composite state. Similarly, for the final state but instead of 'start' it can be 'end', i.e., start if it's initial state and end if it's the final state. In addition, the 'terminate pseudostate' can be represented as same as the final state but with the name 'final' instead of 'end'.

Figure 12 below illustrates an example of a system with the name 'Vehicle status' and it contains initial state, 'Vehicle off' state, and final state and they are represented in neo4j-design as states with name property 'start vehicle status', 'Vehicle off', and 'end vehicle status' respectively.

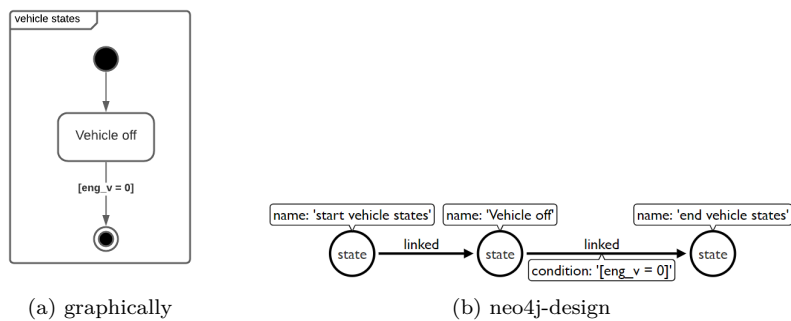


Figure 12: Example of initial and final state notation and representation in Neo4j

4.1.4 Junction Pseudostate

The 'Junction Pseudostate' is represented in neo4j-design as a node and the type name 'junction'. There are no properties in the node. The reason behind doing it in such a way is that it gives the advantage of using query to define the right path between two states that are linked with a junction and that has other transitions attached to it as well. In addition, if its represented as a direct transition between the two nodes with the combined condition may become complicated and hard to track as well as it is error-prone. Furthermore, if it is represented as a node with type 'state' without the need to add values to the properties, it will lead to fetch all connections to the target state and the result of the query will fetch all the paths from source to target through all connected nodes. However, by using a junction node, it can be more specific in fetching the data.

Figure 13 below illustrates the neo4j representation to junction notation has one input and two output transition.

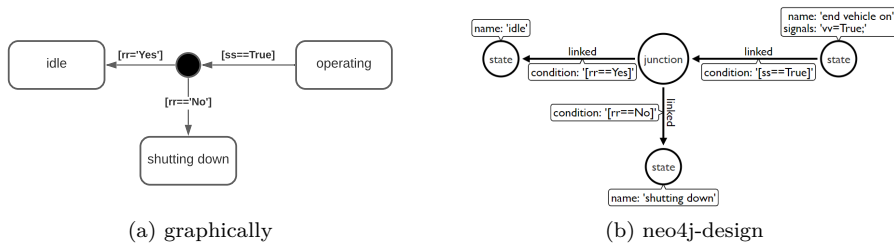


Figure 13: Example of junction notation and representation in Neo4j

4.1.5 Composite State and Entry/Exit Point

The 'composite state' representation in neo4j-design is explained in two parts referred to as 'internal' and 'transition link'. The 'internal' is the nested states inside the simple composite state (one region). The nested states are treated as another state machine diagram and follow the same rules starting from the initial state and end on the final state if existed. The 'transition link' referred to transitions that entering and leaving the composite state. It is preferred to avoid using unstructured transitions, and instead, it is preferred to use 'entry point' and 'exit point'.

The 'entry point' and 'exit point' are represented in neo4j-design as a node as same as the 'state' representation, which means that it contains the same properties. For the entry point, the 'name' property should have 'entry' followed by the composite state name as value, and the signal property should contain the entry action from the composite state only. While for the exit point, the 'name' property should have 'exit' followed by the composite state name as value, and the signal property should contain the exit action from the composite state only.

In this way, it's simpler to describe the four different situations for entering and leaving the composite state as follows: (1) the 'default entry' is represented as all transitions are terminated on the entry point instead of the border-box of the composite state and a new transition with empty property is used to link between the entry point and initial state in the composite state; (2) the 'default exit' is represented as a transition from every nested state in the composite state exclude the initial state to the exit point (new) through a new transition with an empty property, and a new transition should be sourced from the created exit point and terminated on the outside state that is selected graphically. (3) the 'explicit entry' is represented as a transition with its property is terminated on the entry point (new) instead of the target destination (i.e. state or junction) and a new transition with empty property is used to link between the entry point and the target destination; (4) the 'explicit exit' is represented as the new transition with empty property is terminated on the exit point instead of the target destination and a transition with its property is used to link between the exit point and the target destination.

The two figures below (figure 14, and 15) illustrate 'default entry', and 'default exit' as well as 'explicit entry' and 'explicit exit'. Figure 14 illustrates the neo4j representation to 'default entry' and 'default exit'. Figure 15 below illustrates the neo4j representation to 'explicit entry' and 'explicit exit'.

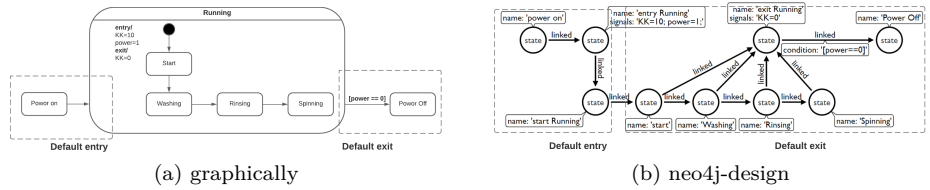


Figure 14: Example of 'default entry' and 'default exit' and representation in Neo4j

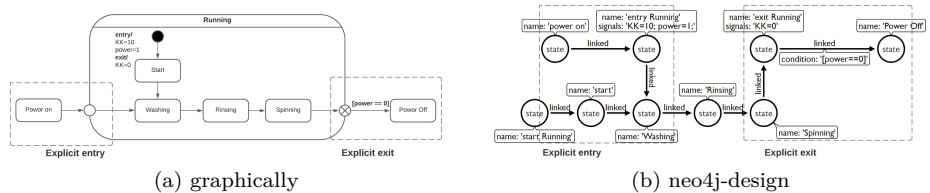


Figure 15: Example of 'explicit entry' and 'explicit exit' and representation in Neo4j

In this way of representation, it's simpler to build in graph database since this way requires less effort than using many transitions to cover all possible connections separately which is an advantage. Another advantage is the ability to define the entry and exit actions in a way query can fetch correctly. In case the composite state has multiple default exit transitions that are terminated on

the same state and contain different trigger conditions, then the right way to define the path is depending on the specific query to fetch the proper data, and the information should be described in the inserted requirement specification.

4.1.6 History Pseudostate

There is no representation for both kinds of 'History Pseudostate' as neo4j-design. The history pseudostate should be done manually by linking the terminated transition at the history notation to the proper inner sub-state. Afterward, the result should be treated as a composite state that have explicit entry transition link.

The shallow history notation should be removed and the transition terminates on a shallow history should be replaced by a transition terminates on the first inner-sub state in the composite state. Figure 16 below of a washing machine illustrates the representation of the shallow history needed to be considered before transforming the diagram to a graph.

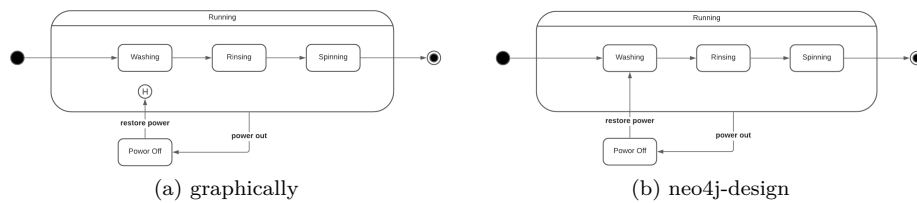


Figure 16: Representation of shallow history notation

On the other hand, The deep history notation should be removed and the transition terminates on a deep history should be replaced by a transition terminates on the latest active inner sub-state in the composite state. Figure 17 below of a washing machine illustrates the representation of the deep history needed to be considered in transforming the diagram to a graph. It should be noted that in this example the 'rinsing' state is the most recent inner sub-state before power out transition is fired.

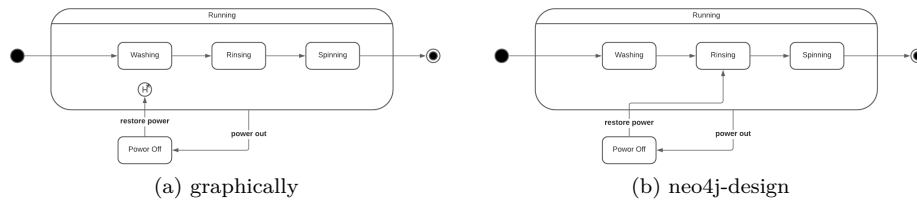


Figure 17: Representation of deep history notation

4.2 Experimentation

To examine the proposed framework, a sample of data is used and applied to a proof-of-concept implementation.

4.2.1 Metadata (Sample)

A UML state machine diagram is used as a sample that illustrates a simple concept of a high voltage car battery which contains multiple states and a transitions linked between them, as shown in the figure 18 below. The list of signals mentioned on the bottom refers to the type and the possible range values for the signals mentioned in the notations.

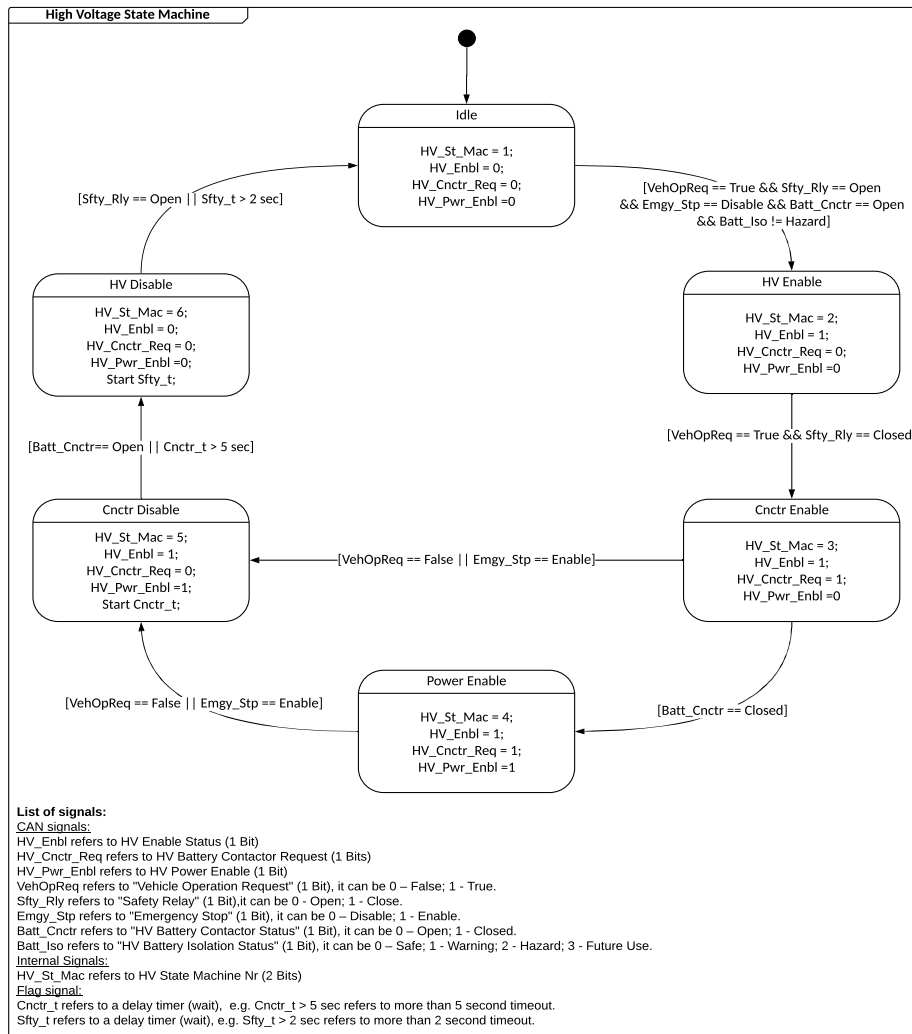


Figure 18: Sample of UML state machine diagram

A list of requirements specifications is used in the table 1 below. They are formulated without following any boilerplate, as a base for the process and can be changed if needed to generate a more complex one.

Table 1: The list of requirements used in the implementation tool

ID	Description
001	In state idle the HVSM shall make the transition to state HV Enable if Vehicle Operation Request is True AND Safety Relay is Open AND Emergency Stop is Disabled AND HV Battery Contactor Status is Open AND HV Battery Isolation Status is not Hazard
002	In state HV Enable the HVSM shall make the transition to state Cnctr Enable if Vehicle Operation Request is True AND Safety Relay is Closed
003	In state Cnctr Enable the HVSM shall make the transition to state Power Enable if HV Battery Contactor Status is Closed
004	In state Power Enable the HVSM shall make the transition to state Cnctr Disable if Vehicle Operation Request is False OR Emergency Stop is Enabled

A list of test cases is suggested in table 2 below. The table shows collected test data from the UML sample diagram to imitate the manual approach in writing test cases. To evaluate the accuracy of the tool, it should give results match the one it table 2.

Table 2: The list of test cases sample that imitate the manual approach

Req. ID	Pre-condition	Action	Expected Result
001	HV_St_Mac=1 HV_Enbl=0 HV_Cnctr_Req=0 HV_Pwr_Enbl=0	VehOpReq=1 Sfty_Rly=0 Empy_Stp=0 Batt_Cnctr=0 Batt_Iso=!2	HV_St_Mac==2 HV_Enbl==1 HV_Cnctr_Req==0 HV_Pwr_Enbl==0
002	HV_St_Mac=2 HV_Enbl=1 HV_Cnctr_Req=0 HV_Pwr_Enbl=0	VehOpReq=1 Sfty_Rly=1	HV_St_Mac=3 HV_Enbl=1 HV_Cnctr_Req=1 HV_Pwr_Enbl=0
003	HV_St_Mac=3 HV_Enbl=1 HV_Cnctr_Req=1 HV_Pwr_Enbl=0	Batt_Cnctr=1	HV_St_Mac==4 HV_Enbl==1 HV_Cnctr_Req==1 HV_Pwr_Enbl==1
004	HV_St_Mac=4 HV_Enbl=1 HV_Cnctr_Req=1 HV_Pwr_Enbl=1	VehOpReq=0 Empy_Stp=1	HV_St_Mac==5 HV_Enbl==1 HV_Cnctr_Req==0 HV_Pwr_Enbl==1

The UML state machine diagram sample used in figure 18 above doesn't cover all the described notation in the definition in Section 4.1. However, this sample will be used to just illustrates the final output result of the process and give a better understanding of the proposed framework.

From the data presented in tables 1, and 2, and in figure 18 above, a brief explanation can be as follows: the requirements in table 1 are formulated in a simple pattern that can be described as: *source state* shall go to the *target state* once the *transition* condition is valid. Accordingly, the test case result in table 2 for pre-condition, action, and expected result refers to *source state*, *transition*, and *target state* respectively. Each row in the table represents one test case result for a requirement in the requirements list, according to the corresponding ID number. And in each column, a test data is collected from the UML diagram in figure 18 and sorted afterward. The data collected in pre-condition and post-condition columns correspond to the signals activity in the *source state*, and target state respectively, and the action column corresponds to the event and guard on the transition between the two states.

4.2.2 Building the Graph Database

The notations inside the sample diagram are initial state, state, and transition. The aforementioned definitions are applied to transform the simple state machine diagram into the graph inside Neo4j DBMS. A brief explanation about the steps used for building the database will be as follows. The initial state and states are represented as a node with type state. The initial state value to name propriety is chosen 'start'. Each state value to the name property is the name of the state graphically, and the value to the 'signals' property is the combination of all the activities inside the state. The transition is represented as a relationship with the label 'linked', and apparently, all transitions have 'condition' and no 'action' properties. The value to the condition property is the exact same information mentioned on the transition graphically. In the sample diagram, there is a reset timer which is represented as 'Cnctr_t = 0' in 'Cnctr Disable' and 'Sfty_t = 0' in 'HV Disable', and the 'sec' is removed from the transition.

The graph database can be built by using either Cypher query or Arrow¹ tool (third-party software) that generate Cypher query out from the inserted design. The Cypher query is used to build the graph database in Neo4j via its terminal console interface. In addition, the terminal console takes Cypher query and fetches the requested data. This can act as primary testing for the built database. After that, the database is ready to use.

The Arrow's tool is simple and requires only drag-and-drop nodes and links them to generate the Cypher query (the generated Cypher query is attached in Appendix A, listing 4) which is used without any modification to create the graph database inside Neo4j. The figure 19 below illustrates the sample diagram after designing it manually on the Arrow website.

¹Arrow is a graph diagram library, <https://github.com/apcj/arrows>

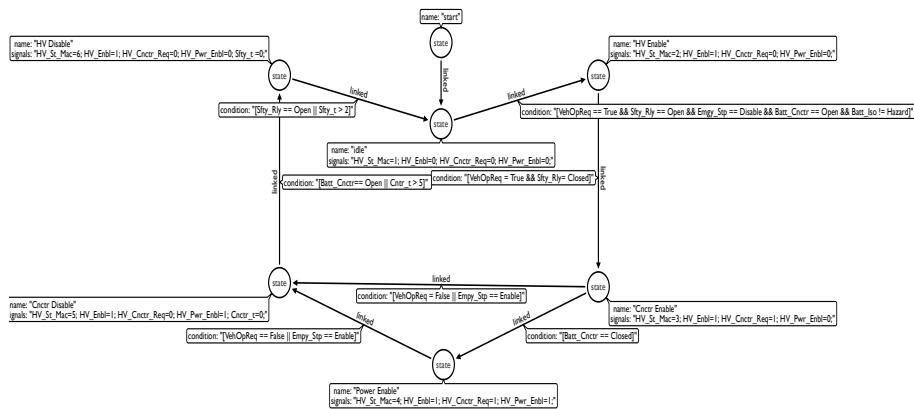


Figure 19: Manually created design that represent the sample diagram in Arrow tool

Graph visualization to the inserted Cypher query in the Neo4j DB is shown in the figure 20 below.

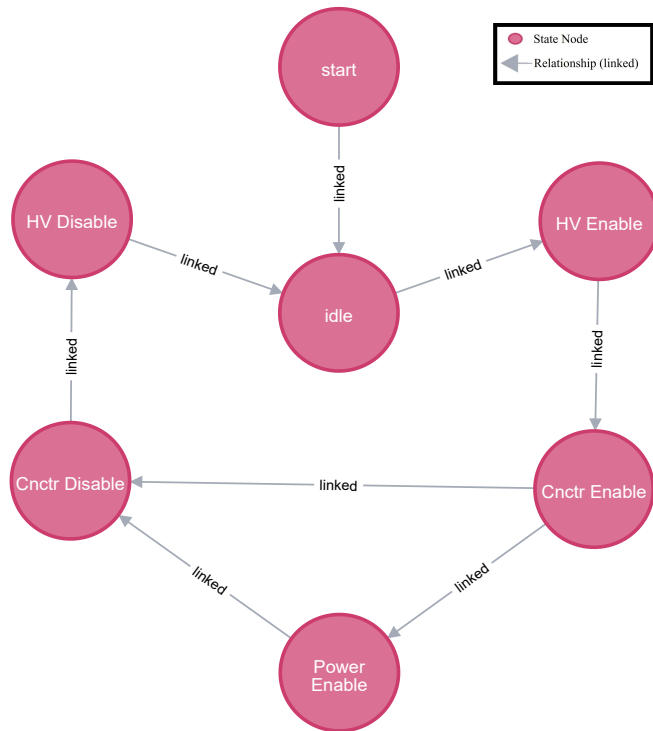


Figure 20: Neo4j Graph Visualization that represent the sample diagram

4.2.3 Schema and Fetching Data

The GraphQL API is a query language for APIs used in the back-end. The GraphQL API uses a schema that specifies all of the types and fields available in the graph, and the schema can include additional variables not necessarily needed to be included in the graph, and any property can be modified in the resolver [24].

The query should be formulated according to the defined schema. The query request should return all the data requested from the database nothing more or less (i.e., returns requested data only). Definitely, it always returns predictable results unlike the Restful API that returns all fields including the not requested ones. Likewise, the GraphQL API allows to construct dynamic queries in many different ways, for instance, by using 'fragment' to write the code once and use it multiple times, or 'aliases' to request the same node type many times, passing variables in arguments, and using a directive like 'include' and 'skip' that acts as if statement [24].

The dedicated schema for this sample consists of one type of nodes and one relationship. The schema can be described in many different ways; especially when using the Cypher query. For this sample, the schema used is shown in listing 1 below.

```
1 type state {
2   name: String!
3   signals: String
4   transition(sourcestate: String, targetstate: String): String
5     @cypher(
6       statement: "MATCH (x:state {name: $sourcestate})-[y:linked]->(z:state {
7         name: $targetstate}) RETURN y.condition"
8     )
9   dictionary: [String]
10  state: state @relation(name: "linked", direction: "OUT")
}
```

Listing 1: The graphql API schema used for the sample diagram

The type 'state' schema is formulated to fetch the same result described in the table 2 above in Section 4.2.1. For this reason, there are four arguments which are: the 'name', 'signals', 'transition', and 'state'. The 'name' and 'signals' are the main property in the node with type state. The 'transition' takes two arguments which are the source state and the target state names and by using Cypher query to fetch the condition only in the transition between the two states. For instance, to get the transition data between the state 'HV Enable' (source state) and 'Cnctr Enable' (target state) as shown in figure 18 above in Section 4.2.1, they are referred to LabelA and LabelB respectively in the Cypher pattern shown below. According to the transition definition described in the above section, the transition can have two KVP the 'condition' and 'action', and in this sample, the 'condition' property is only used. For this reason, the Cypher query only returns the condition property from the transition between the two states.

//Cypher pattern

(node1:LabelA)-[rel1:RELATIONSHIP_TYPE]->(node2:LabelB)

The 'dictionary' is a list of strings used later on on the front-end implementation to act as a dictionary that contains all the names of states. This is handled by the resolver on the back-end. The last argument 'state' is an overlapped (nested) state that refers to the next target state in the diagram and acts as another iteration.

It is a dedicated schema for testing the sample only, but once the process is fully covered then a generic schema can be used to fetch data from different diagrams (universal schema). This is considered as future work.

A test query is used to fetch data according to the defined variables. There are many possible structures to formulate the query and one of them is shown in the listing 2 below.

This query takes two-parameters (variables). It should be a state's name and referred to the source and target state. The aliases feature is used to request the same type node multiple times and it's referred to as first, second, and third in listing 2 below. The intention by structuring them in the following way is to make each alias describe part of the test cases. For instance, the first, second, and third referred to as pre-condition, action, and post-condition; respectively. This way makes it simple to map the result of the fetched data later on.

As an example, a test is done on requirement 003 from the table 1 above in Section 4.2.1, aiming to get the result that matches test case 003 in the table 2 above in Section 4.2.1 to verify that the query can fetch the desired data. The requirement 003 mentioned the states name 'Cnctr Enable' (source state) and 'Power Enable' (target state) which are used to return value to the 'fname' and 'lname' variables, respectively in the query. The response to the request query is shown in the listing 3 below.

```
1 query UmlSampleTest(  
2   $fname: String!  
3   $lname: String!) {  
4   first: state(name: $fname) {  
5     signals  
6   }  
7   second: state(name: $fname) {  
8     transition(sourcestate: $fname  
9       , targetstate: $lname)  
10  }  
11  third: state(name: $lname) {  
12    signals  
13  }
```

Listing 2: A query used to retrieve data for the sample diagram

```
1 {  
2   "data": {  
3     "first": [{  
4       "signals": "HV_St_Mac=3;  
5         HV_Enbl=1; HV_Cnctr_Req=1;  
6         HV_Pwr_Enbl=0;"  
7     }],  
8     "second": [{  
9       "transition": "[Batt_Cnctr  
10        == Closed]"  
11     }],  
12     "third": [{  
13       "signals": "HV_St_Mac=4;  
14         HV_Enbl=1; HV_Cnctr_Req=1;  
15         HV_Pwr_Enbl=1;"  
16     }]}  
17 }
```

Listing 3: The GraphQL response to the requested query

From the output data, by comparing the result of each of the aliases: first,

second, and third with the test case result that structured as pre-condition, action, and post-condition, respectively. The result of the example in listing 3 above is fully matched with the result in the table 2 above in Section 4.2.1, except it should be sorted and return signals values from the list of signals.

4.2.4 Result (Client-Side)

In the front-end implementation, the process is applied to achieve the main goal by making the tool take the testable requirements and generate test cases (like the black-box method) for them in a graphical user interface. A prototype of the tool is implemented and illustrated in appendix B.

The figures below illustrate a final implementation of the two main goals of the tool which are inserting requirements and generating test cases. Along with the main page of the tool.

The figure 21 below illustrates the main page of the tool. The tool has a fixed navigate bar at the top. The rest of the page gives information about the tool.

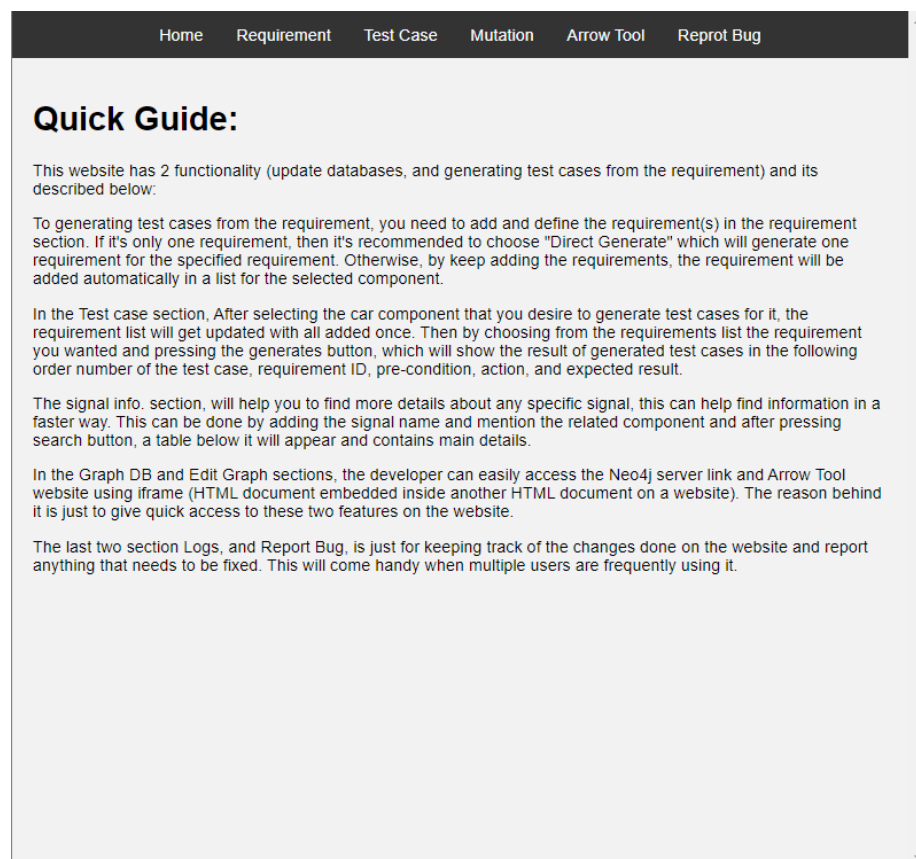


Figure 21: The main page of the implemented tool

The following front-end implementation for the sample used consists of two main React components (the two goals mentioned in the prototype).

Figure 22 below shows the first component of the tool. This component is used to insert and save the requirements in browser local storage to retrieve on the second component later on. The text fields are filled with requirement ID '003' from table 1 above in Section 4.2.1, as a demonstration of the insertion process. It should be noted that it is done manually by the user/tester. The lower part under the 'Submit' button shows an additional component that displays the saved requirement in the browser local storage once clicked.

The screenshot shows a web application interface with a dark navigation bar at the top containing links for Home, Requirement, Test Case, Mutation, Arrow Tool, and Reprot Bug. The main content area is titled "Requirement:" and contains a form with the following fields:

- ID:** A text input field containing "003".
- Title:** A text input field containing "enable HVSM S3".
- Description:** A text input field containing "In state Cnctr Enable the HVSM shall make the transition to state Power Enable if HV Battery Contactor Status is CI".
- Car Model:** A dropdown menu with "#Car 1" selected.
- Car Component:** A dropdown menu with "Battery" selected.

Below the form is a large green button labeled "Submit". Underneath the "Submit" button, there is a section titled "Requirement List:" which displays "Number of saved requirements is 4" next to a red circular icon with the number 4. To the right of this text is a yellow "Refresh" button with a circular arrow icon. At the bottom of this section is another large green button labeled "Show List".

Figure 22: Inserting requirement page in the implemented tool

Figure 23 shows a table contain all the saved requirement in the browser local storage in a overlay on the center of the page. It should be noted that all

requirements are listed together and no traceability between them.

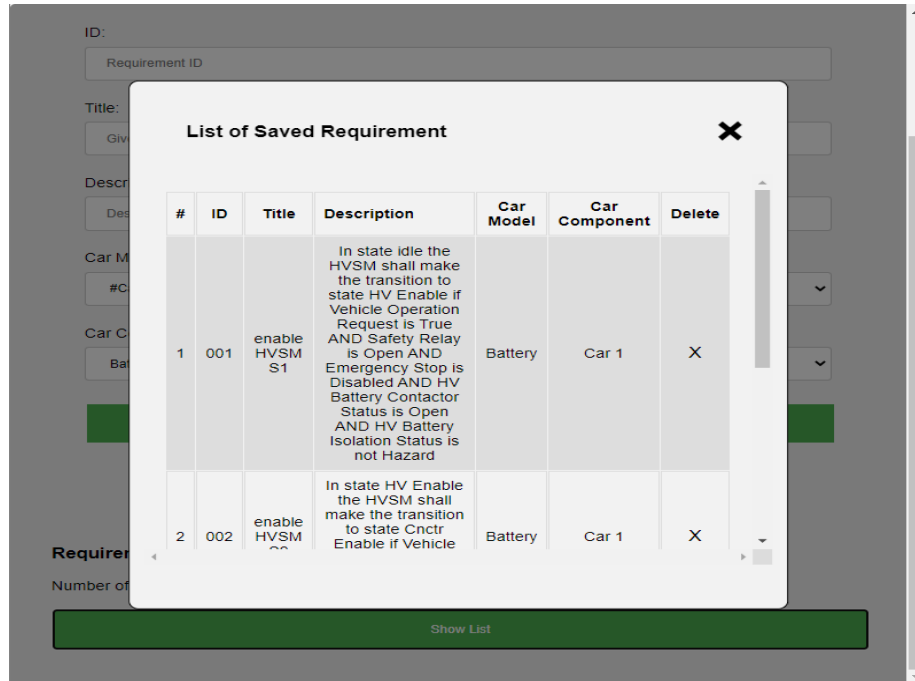


Figure 23: List of the inserted requirements in the implemented tool

In figures 24, 25, 26, and 27 below shows the second component of the tool. This component is used to process the inserted requirements by using named-entity-recognition (part of NLP pipeline) to get the state’s name as a token and return them as a value to the variables in the query. The query used is similar to the requested query in the back-end but with one additional state with aliases 'fourth' that contains 'dictionary' property. The query is sent to the back-end through Apollo-client and the response to the request query is returned from the back-end through Apollo-server. Afterward, the fetched data are sorted into three columns inside the final table as pre-condition, action, and post-condition after replacing their value property with the proper signal value from the list of signals and define the input and output signal as '=' and '==' sign respectively.

The implementation of the second component and the result of the generated test cases for each requirement presented in table 1 above in Section 4.2.1 is illustrated as follows. The figures 24, 25, 26, and 27 shows the checkbox is selected for requirements '001', '002', '003', and '004' respectively. Also in each figure, the result of the generated test case is shown in the lower part below the 'Result' header. It should be noted that the tool generates only one test case at this stage and it should generate multiple test cases (valid and invalid test) to verify the requirement.

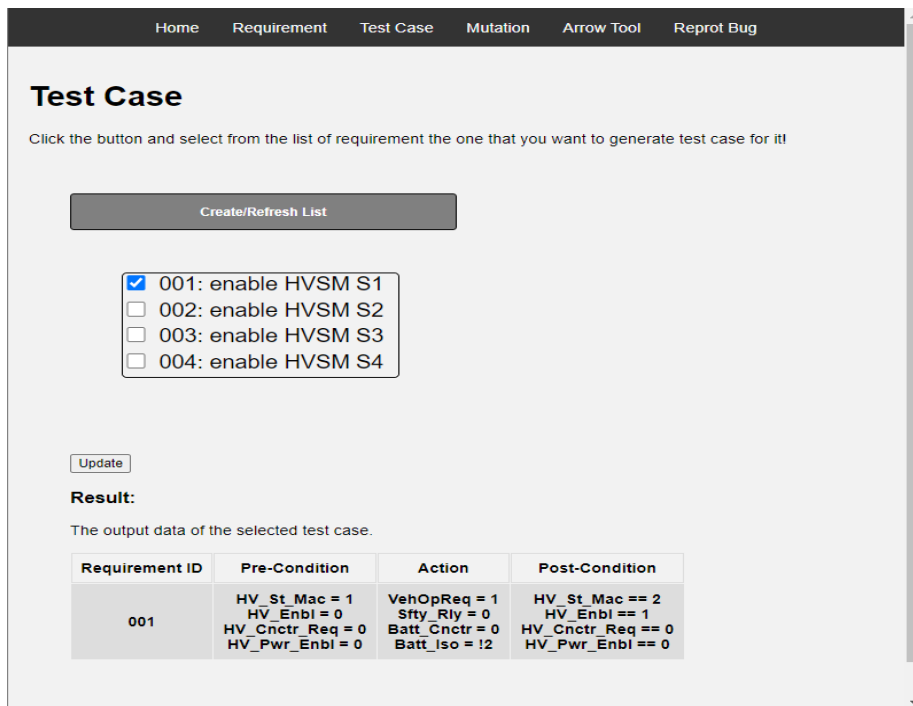


Figure 24: Test case result for requirement 001 in the implemented tool

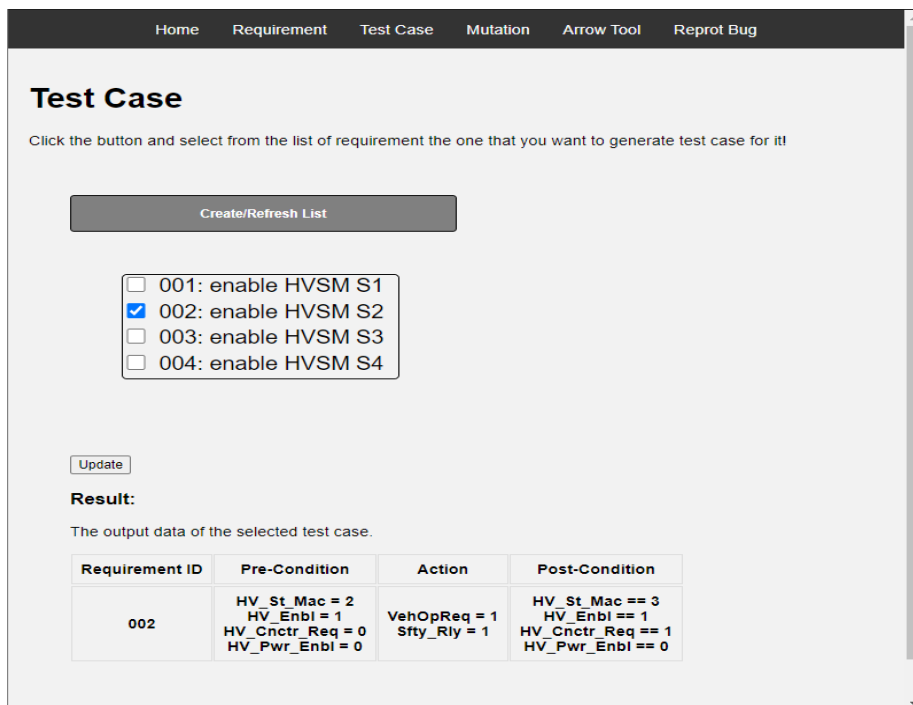


Figure 25: Test case result for requirement 002 in the implemented tool

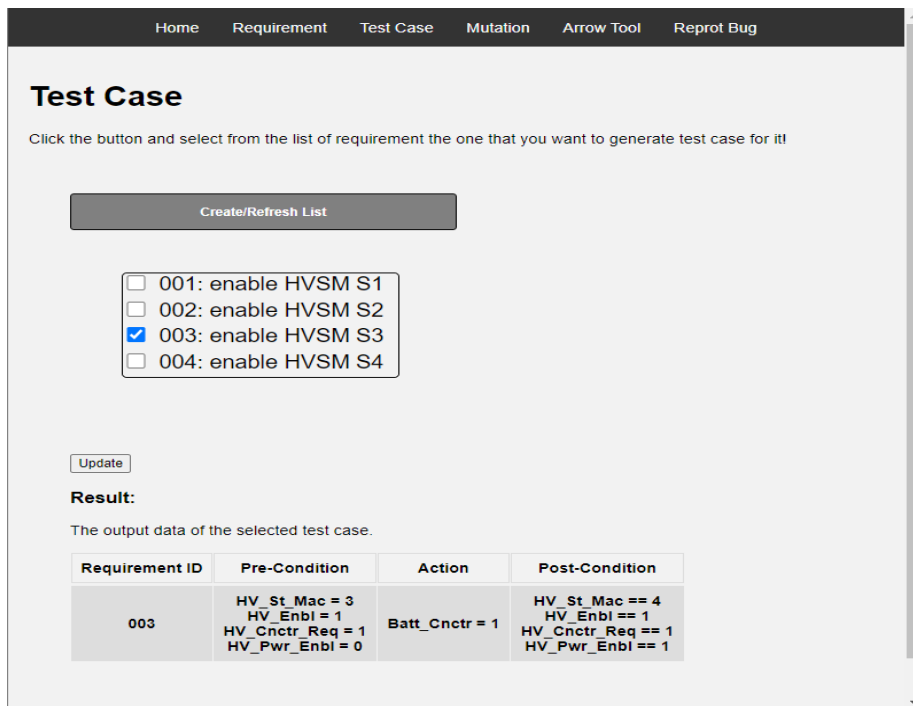


Figure 26: Test case result for requirement 003 in the implemented tool

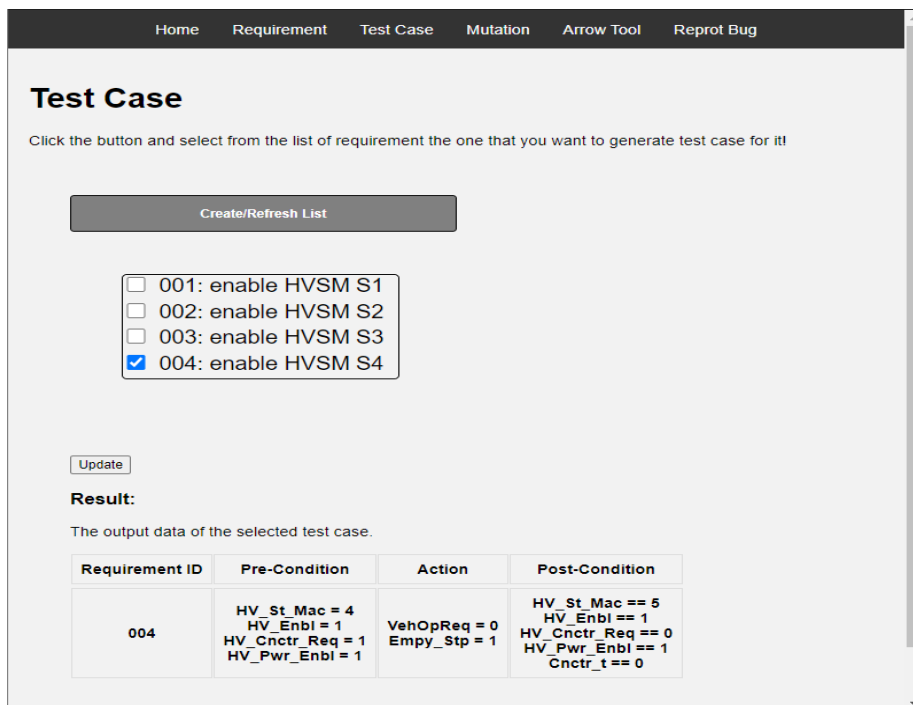


Figure 27: Test case result for requirement 004 in the implemented tool

The result of the generated test case for each requirement '001', '002', '003', and '004' illustrated in the figures 24, 25, 26, and 27 above respectively matched as same as the desired test case results in table 2 above in Section 4.2.1. This mean that, each requirement presented in table 1 above in Section 4.2.1 matched with the structure and content of the test case in table 2. Thus, the tool took the requirement as input and generate test cases for it as output.

5 Discussion and Conclusion

This thesis has suggested a new method to transform the UML state diagram into a graph database inside Neo4j DBMS and has suggested a framework that uses the transformed graph database to fetch the proper test data and create a test case according to the inserted requirement specification. A proof-of-concept has been implemented to demonstrate the proposed framework.

After acquiring a better understanding of the implemented technology, it has been noticed that converting UML state diagram into graph databases can be applied, and it becomes clear that the definitions used in transformation together with the capability of GraphQL API are very flexible and can be represented in many different structures to imitate the tester experience in collecting test data. In addition, the result from the experimentation made on a sample shows that generating test cases is possible and it matches the results in the expected sample. For this reason, it can be considered as a systematic way of mapping the UML state machine into a graph database taking into consideration that it requires more work to be done in improving the process to be able to adapt eventually.

During the implementation process, different methods to structure the graph using different definitions are tested. Two of the methods are described and referred to as test-1 and test-2. Each one has a drawback.

In test-1, all the signals in the UML diagram are needed to be mentioned as a list of KVP in the schema. The advantages of this method are: (1) the fetched data are more sorted and easier for mapping, (2) define the data type for each signal, and (3) the query can take variables from the properties to fetch data. While the disadvantages of this method are: (1) the query becomes very large even with using fragment and interface elements, and (2) requires frequently updating the schema and rebuild the back-end.

In test-2, making the connection is as explicit as mentioned in [17] by representing the state with name property as a parent node and the signals are listed in a different node linked to it. There are only two disadvantages of this method: (1) the graph diagram becomes more complex for the designer to implement, and (2) the fetched data need extra steps in mapping.

For this reason, one comes to conclude that the proposed approach in this thesis is a feasible solution, since it uses a few KVPs that contain all the signals as a string and gives the advantage to build the back-end once and apply it to all graphs.

In this framework, the coverage is fully based on the capability of the Cypher query to declare the proper path between the nodes. For this reason, the inserted requirement should provide a parameter to the query to fetch the test data. The test cases are generated according to the inserted requirement (input) to define the scenario needed to cover, contrary to the coverage method used in [18, 20, 21] which automatically generated test cases directly from the UML

diagrams based on a dedicated algorithm. Likewise, the generated test data is fetched using the query from the graph diagram. The complexity of the query is classified according to the number of transitions which are fired and the more transition fired, the more complex it becomes.

This study has a few limitations where the 'protocol state machines' is not covered. The orthogonal composite state is not described and intended to be implemented according to the description in [16]. There is not enough testing made to cover self-transition and 'internal transition' (event) of the state compartment situations. The generated test cases are not completed because the tool generates only one test case for each inserted requirement.

Nevertheless, there are several challenges were faced during the implementation such as parsing the requirement specification and defining its entity as the requirement can be described in many different ways and it can become complex and ambiguous to process. Similarly, the UML state machine can have a tricky situation that needs a specific Cypher query to fetch the desired path. For instance, the composite state can has multiple 'default exit' transitions that terminate on the same state outside the composite state with different trigger conditions.

As a conclusion, the suggested transformation method and the framework show good potential to be developed and evaluated with a real test from the industry practice. Afterward, it can be adopted to improve the testing process.

5.1 Future Work

The future work will be mainly focusing on improving the tool, such as:

In the front-end, consider creating an additional database to cover all the list of signals to become easier to retrieve and replace the string with the right value after fetched the test data. Moreover, considering using object-constrained language (OCL) as described in [18]. Furthermore, traceability between the requirements and making the tool generate test cases for all selected requirements at the same time.

In the back-end, discover all possible situations in selecting proper test data and accordingly improving the query and the schema. The schema should be implemented in a manner to fetch any node in the graph similarly to path coverage. Besides, setting up the GraphQL mutation that will give the user the privilege to make changes to the databases in the front-end.

In the graph database, extend this approach to cover more UML diagrams and propose a representation for each diagram notation to graph database or another NoSQL database and fetch data using the GraphQL API. Also, applying the proposed method to transform a model-to-model (M2M) in [19] on the UML state machine diagram to automating the current manual process in transforming UML diagram to a graph database.

References

- [1] Leon F Osborne, Jeff Brummond, Robert Hart, Mohsen Zarean, Steven M Conger, et al. Clarus: Concept of operations. Technical report, United States. Federal Highway Administration, 2005.
- [2] Rajib Mall. *Fundamentals of software engineering*. PHI Learning Pvt. Ltd., 2018.
- [3] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.
- [4] Business News Daily. "workplace automation is everywhere, and it's not just about robots". <https://www.businessnewsdaily.com/9835-automation-tech-workforce.html>. Accessed: 2020-05-12.
- [5] Michael Chui, James Manyika, and Mehdi Miremadi. Where machines could replace humans—and where they can't (yet). *McKinsey Quarterly*, 30(2):1–9, 2016.
- [6] Sai Ganesh Gunda. Requirements engineering: elicitation techniques, 2008.
- [7] The Software Testing Fundamentals. "gray box testing". <https://softwaretestingfundamentals.com/gray-box-testing>. Accessed: 2020-05-12.
- [8] The Performance Lab. "software test life cycle (stlc) importance". <https://performancelabus.com/software-test-life-cycle-stlc-importance/>. Accessed: 2020-05-12.
- [9] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.
- [10] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.
- [11] Mahesh Lal. *Neo4j graph data modeling*. Packt Publishing Ltd, 2015.
- [12] The GraphQL Foundation. "the top 10 ways to get to know neo4j". <https://neo4j.com/blog/the-top-10-ways-to-get-to-know-neo4j/>. Accessed: 2020-05-12.
- [13] Neo4j Technologies. "what is a graph database?". <https://neo4j.com/developer/graph-database/>. Accessed: 2020-05-12.
- [14] Neo4j Technologies. "cypher query language". <https://neo4j.com/developer/cypher/>. Accessed: 2020-05-12.
- [15] Debasish Kundu and Debasis Samanta. A novel approach to generate test cases from uml activity diagrams. *J. Object Technol.*, 8(3):65–83, 2009.
- [16] Vincent Delfosse, Roland Billen, and Pierre Leclercq. Uml as a schema candidate for graph databases. *NoSql Matters 2012*, 2012.

- [17] Yasir Dawood Salman and Nor Laily Hashim. Automatic test case generation from uml state chart diagram: a survey. In *Advanced Computer and Communication Engineering Technology*, pages 123–134. Springer, 2016.
- [18] Md Azaharuddin Ali, Khasim Shaik, and Shreyansh Kumar. Test case generation using uml state diagram and ocl expression. *International Journal of Computer Applications*, 95(12), 2014.
- [19] Gwendal Daniel. Umltographdb: Mapping uml to nosql graph databases, 2016.
- [20] Chartchai Doungsa-ard, Keshav Dahal, Alamgir Hossain, and Taratip Suwannasart. Test data generation from uml state machine diagrams using gas. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 47–47. IEEE, 2007.
- [21] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *11th Asia-Pacific software engineering conference*, pages 284–291. IEEE, 2004.
- [22] Philip Samuel, Rajib Mall, and Ajay Kumar Bothra. Automatic test case generation using unified modeling language (uml) state diagrams. *IET software*, 2(2):79–93, 2008.
- [23] Grandstack. "getting started with grandstack". <https://grandstack.io/docs/getting-started-neo4j-graphql>. Accessed: 2020-05-12.
- [24] The GraphQL Foundation. "introduction to graphql". <https://graphql.org/learn/>. Accessed: 2020-05-12.

Appendix A : Cypher Query Used

```
1 CREATE
2   ('0' :state {name:"start"}) ,
3   ('1' :state {name:"idle",signals:"HV_St_Mac=1; HV_Enbl=0; HV_Cnctr_Req=0;
4     HV_Pwr_Enbl=0;"}),
5   ('2' :state {name:"HV Enable",signals:"HV_St_Mac=2; HV_Enbl=1; HV_Cnctr_Req
6     =0; HV_Pwr_Enbl=0;"}),
7   ('3' :state {name:"Cnctr Enable",signals:"HV_St_Mac=3; HV_Enbl=1;
8     HV_Cnctr_Req=1; HV_Pwr_Enbl=0;"}),
9   ('4' :state {name:"Cnctr Disable",signals:"HV_St_Mac=5; HV_Enbl=1;
10    HV_Cnctr_Req=0; HV_Pwr_Enbl=1; Cnctr_t=0;"}),
11   ('5' :state {name:"Power Enable",signals:"HV_St_Mac=4; HV_Enbl=1;
12    HV_Cnctr_Req=1; HV_Pwr_Enbl=1;"}),
13   ('6' :state {name:"HV Disable",signals:"HV_St_Mac=6; HV_Enbl=1;
14    HV_Cnctr_Req=0; HV_Pwr_Enbl=0; Sfty_t =0;"}),
15   ('0')-[:'linked' ]->('1'),
16   ('1')-[:'linked' {condition:"[VehOpReq == True && Sfty_Rly == Open &&
17     Emgy_Stp == Disable && Batt_Cnctr == Open && Batt_Iso != Hazard]"
18     }]->('2'),
19   ('2')-[:'linked' {condition:"[VehOpReq = True && Sfty_Rly= Closed]"
20     }]->('3'),
21   ('3')-[:'linked' {condition:"[VehOpReq = False || Emgy_Stp == Enable]"
22     }]->('4'),
23   ('3')-[:'linked' {condition:"[Batt_Cnctr == Closed]" }]->('5'),
24   ('5')-[:'linked' {condition:"[VehOpReq == False || Emgy_Stp == Enable]"
25     }]->('4'),
26   ('4')-[:'linked' {condition:"[Batt_Cnctr== Open || Cnctr_t > 5]" }]->('6'),
27   ('6')-[:'linked' {condition:"[Sfty_Rly == Open || Sfty_t > 2]" }]->('1')
```

Listing 4: Cypher query used to build the graph database that represent the sample diagram

The main *Cypher syntax* used in listing4 listed below.

```
//node
(variable:Label {propertyKey: 'propertyValue'})

//relationship
('node1 id')-[variable:RELATIONSHIP_TYPE {propertyKey: 'propertyValue'}]->('node2 id')
```

The node Cypher syntax is used to create the nodes for the label 'state' and each one has a unique variable number that can be used to describe its relationship with other nodes. The property key for node 'state' are 'name' and 'signals' and their values are described in property value, as mentioned in the definitions in the 'Approach Implementation' section. According to the listing4, the code from line 2 until line 8 uses the node Cypher syntax.

The relationship Cypher syntax is used to create a connection between two 'state' nodes. This is being done by referring to the 'node1 id' and 'node2 id' by their variable number only. There is no need to mention any variable for the relationship in creating a graph database. The relationship type used is 'linked' and applied for all transitions. The property key and property value used are the same methods in the node Cypher syntax and as described in the definitions in the 'Approach Implementation' section. According to the listing4, the code from line 9 until line 16 uses the relationship Cypher syntax.

Appendix B: Prototype of the Tool

A prototype of the proposed tool in the thesis work is illustrated and explained in this appendix. It should be noted that it's implemented after getting a better understanding of the input and output of the desired tool.

The user interface (UI) prototype is implemented using Adobe Xd¹ and a 'fitness dashboard template'² is used. Many changes made on the template to give better overview about the proposed tool functionality.

The figures below illustrate a design prototype of the two main goals of the tool which are inserting requirements and generating test cases. Along with the main page of the tool.

The figure 28 below illustrates the main page of the proposed tool. The tool has a fixed navigate bar at the top. The rest of the page gives information about the tool. Besides, a section shows the recent updates and changes made to the tool.

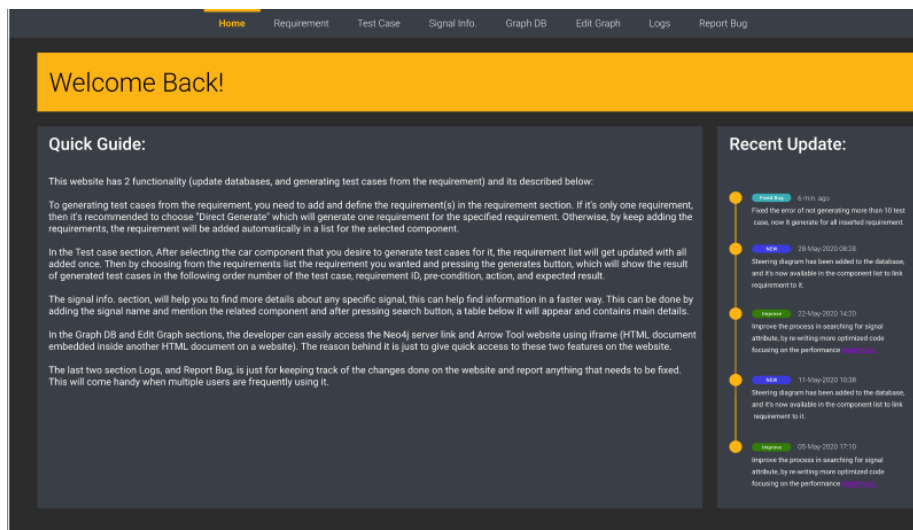


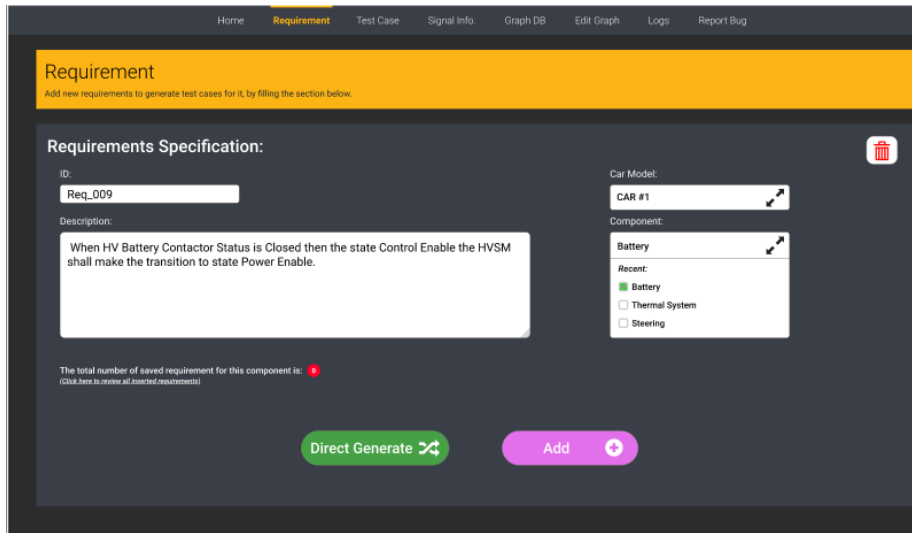
Figure 28: A prototype of the main page of the proposed tool

In figure 29 below illustrates the first goal of the tool which is the inserting requirements. The figure 29a shows text fields and selection boxes to allow the user to insert the requirements (input to the tool). The lower part shows two buttons, one to generating test data directly for the added requirement, while the other one is to store the requirement in the tool database to retrieve later. The figure 29b shows a table format that appears on the center of the page as overlay. The overlay list all the inserted requirement and stored on the tool

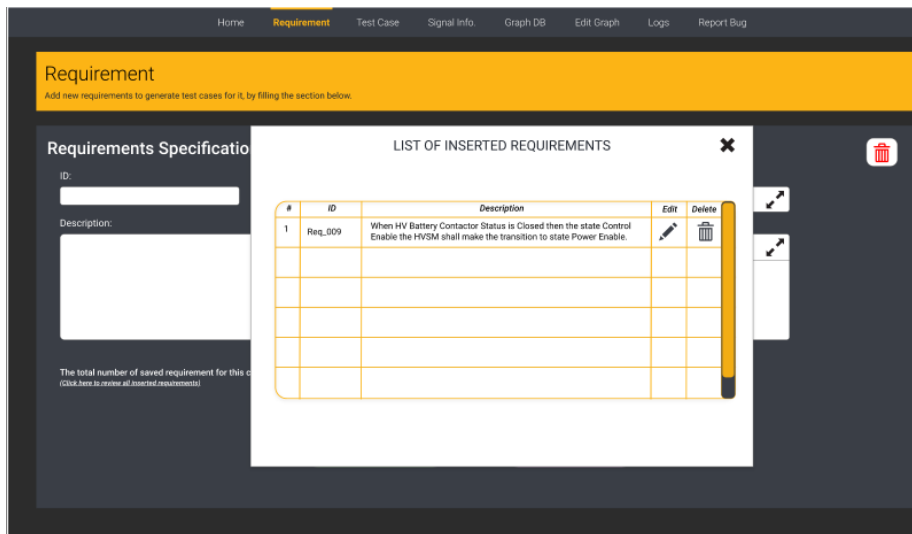
¹<https://www.adobe.com/products/xd.html>

²<https://www.xdguru.com/fitness-dashboard-xd/>

database. The user needs to select the blue button with the label 'Add' that located at the bottom of figure 29a in order to add more requirements.



(a) Inserting requirement to the tool

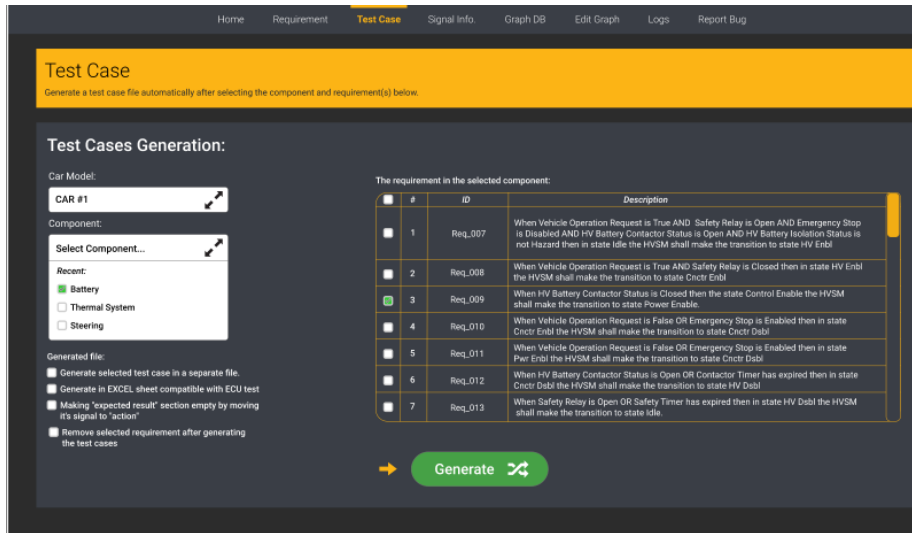


(b) A list of all inserted requirement to the tool

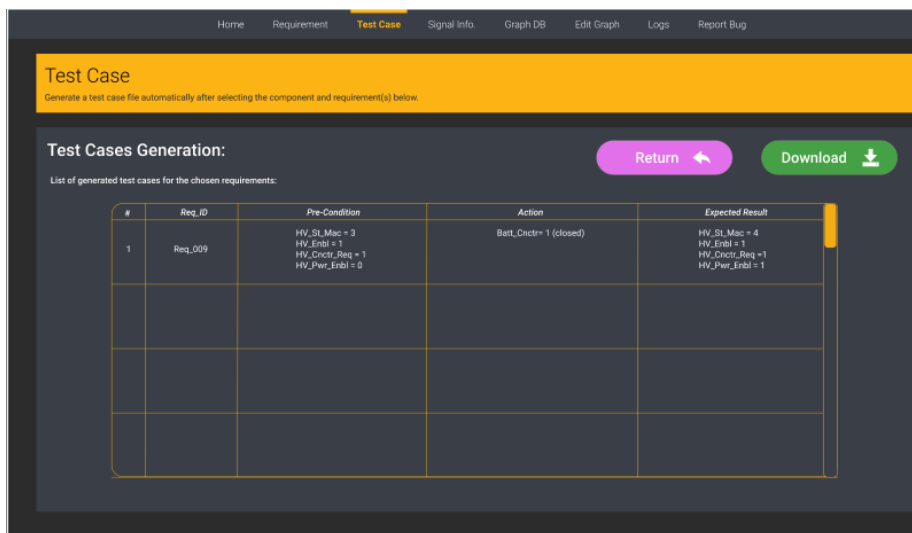
Figure 29: A prototype of the inserting requirement process to the proposed tool

In figure 30 below illustrates the second goal of the tool which is generating test cases for the selected requirements. The figure 30a shows selection boxes to specify the domain field of the product, to list all the previously inserted requirements in a table format located on the right side of the page. From the list of inserted requirements, the user needs to choose the desire requirements by

clicking on the checkbox located in the first column and prior to the requirement ID. After selecting the requirements the user needs to click the button with the label 'Generate' at the lower part of the page. The figure 30b shows the generated test case (output of the tool) for the selected requirement earlier in 30a. The purpose of the button located on the upper right side with the label 'download', to create a file that can be used directly to the software(s) at the test execution phase during the process of the software testing life cycle.



(a) Choosing requirements to generate test cases for them



(b) A generated test case for one of the requirements

Figure 30: A prototype of the generating test cases process to the proposed tool