Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Bachelor of Science in Computer Science
15.0 credits

AUTOMATED MODEL GENERATION USING GRAPHWALKER BASED ON GIVEN-WHEN-THEN SPECIFICATIONS

Joakim Korhonen
jkn17013@student.mdh.se

Examiner: Wasif Afzal
Mälardalen University, Västerås, Sweden

Supervisors: Eduard Paul Enoiu
Mälardalen University, Västerås, Sweden

July 2, 2020

Abstract

Software testing is often a laborious and costly process, as testers need extensive domain-specific knowledge and engineering experience to manually create test cases for diverse test scenarios. These scenarios in many industrial projects are represented in requirement specification documents. Since the creation of test cases from these requirements is manual and is error-prone, researchers have proposed methods to automate the creation of tests and execution of tests. One of the most popular approaches is called model-based testing. Model-based testing uses models to manually or automatically create tests based on existing models. Since most of the effort in model-based testing lies in the creation of the model, this thesis aims at improving a model-based testing tool. This improvement is for generating a model from Natural language as this is what requirements usually are written in. Given-When-Then is a test-case writing template used to specify a system's behavior. To implement the natural language processing into a model-based testing tool, an extension for Graphwalker was created. Graphwalker is a popular open-source model-based testing tool, which can create, edit, and test the models created. The extension is using requirements as input written in natural languages and then creates a model based on the requirements provided. Graphwalker's models are based on finite state machines that have elements such as vertices and edges. The model also can change its state, change values of variables, and block access to certain elements. Graphwalker can however not generate models from natural language requirements. This thesis shows how one can transform natural language requirements into models.

The extension is implemented to use requirements through both manual input and via a JSON file and it is processing the text and tags each word. These tags will then be used to interpret the sentence meaning and will either create a transition, change a value, or block access to a selected element. The results of this thesis show that this extension is an applicable method to automatically generate models for the GraphWalker tool. This extension can be used and improved by both researchers and practitioners.

Contents

1. Introduction

Software testing is often a laborious and costly process, as the usual practice is to create test cases manually based on requirements representing the expected behavior [1]. Software testing is an important part of software engineering as it shows that the software executes as expected and is running according to its requirements [2]. Since manual test creation is both error-prone and slow as it requires diverse test scenarios; researchers have proposed multiple techniques to automate both the creation and the execution of tests. However, many of these approaches tend to be complex and expensive to deploy in practice, hence their adoption is not widespread [1]. One popular approach to generate tests is called Model-based Testing (MBT), this approach uses existing models to either manually or automatically create tests. The process includes generating test cases from specifications models representing the systems requirements and desired functionality. The generated test cases are then executed on the System-Under-Test (SUT) to obtain a pass or fail verdict [3]. Different MBT techniques have been developed and applied in different contexts [4]. Consequently, there is a need to investigate the use of MBT and how modeling and test generation can improve the current practices of manually creating MBT models from natural language requirements. As these requirements are in natural language, the ideal solution would be to automatically generate the models from the requirements.

The requirement documents used during the development of the SUT can also be used to create the tests as these describe the SUT's behavior. Usually, these are written in plain natural language. Natural language helps all involved parties to understand and be able to specify what behaviors the system should have [5]. In this thesis, an extension for a model-based testing tool will be created. This extension will allow the tool to accept natural language requirements to generate a test model since this is one of the most laborious steps in MBT. The tool used in this thesis is called Graphwalker and uses finite state machines as its model basis, it doesn't have any functionality which allows it to generate new models. To automatically generate models from the requirement documents the tool will be extended to use the Given-When-Then template, which is a popular test-case and requirement template used in the industry. This template is used to format the natural language into sentences that are easier to interpret into test-cases [6]. The models created by the extension can then be used to create test cases, these test cases can then be used to form a test suite and will be used to test the SUT. These tests will return various results such as how many tests passed and how many failed or if any tests were blocked for other reasons [3]. This means the scientific method used in this thesis is to perform an experiment on an implementation created for this thesis, to see if it can create models from Given-When-Then requirements.

There have been many approaches proposed to automatically generate tests, however many of these approaches assume the creation of specific models that take time and resources [1][2]. Because of the overhead cost of MBT it is not widely used in the industry [7], which also led to a need to decrease the time and resources needed to create tests in MBT. This together with the lack of experimentation limits the evidence on how effective the generated models are. One way to make this task less costly and time-consuming is to directly generate models from the requirements. This leads to the main question investigated in this thesis: "How to automatically create models out of Given-When-Then natural language requirements?".

The goal of this thesis was to create an extension for an existing open-source tool called Graphwalker. This extension adds support for the generation of a model from natural language requirements written according to the Given-When-Then template. The goal is also to describe an approach of how such a system may process natural language according to the Given-When-Then template.

The main result of this thesis is an extension for Graphwalker, which can process natural language requirements and create a testable model out of it. The extension will however be limited to its processing of natural language, as it will follow the structured natural language in the Given-When-Then template [6] and the use of natural language processing is outside the scope of this thesis. The structured natural language limits the natural language in a way that it can be interpreted more easily into different parts of a model. This thesis also includes how this extension is processing the natural language requirements to create a model and what limitations were found. The processed text can be obtained by either a graphical interface (by writing the requirements inside of the tool) or directly through a JSON file.

2. Background

In this section, we outline some background information related to software testing, the use of finite state machines for testing, behaviour-driven development, and the Given-When-Then requirement template.

2.1. Software Testing

Software testing is a big part of software development, it is used to validate the software to see if it runs as expected and gives the expected results. It also aims to not only find the faults of the program; it should also aim for improvements to the software. Its main goal is to measure the specification, functionality, and performance of a program. Software testing has a couple of steps, these include an analysis of the requirements [3] [2]. These requirements are specifications of how the program should act and is often in natural language; it can however be expressed in other languages such as a modeling language (UML, etc). The next step is to plan the test and then develop the test cases; this will allow the creation of a test suite that can be used on the SUT. The results of the test will give if the test case either passed or failed or if the test case didn't run for a different reason. Software testing can also be either manual or automatic. Manual tests are created manually by a tester and automatic test generation is performed by a tool or through scripts.

2.2. Finite State Machines

Finite state machine (FSM) is a mathematical model of computation. An FSM will read a series of inputs and use the input to switch state. An FSM can only have one active state at a time. Each state has certain states it can switch to and the state will be switched depending on where its transitions lead. Transitions are representing how the different states may change; the transitions may also lead back to its original state [8]. For example, the active window on a computer shows the web browser while the user selects the home screen; then the computer's active window transitions into the home screen.

The states and transitions are often represented as graphs with the states being the nodes and the transitions as edges (pointed arrows). Each transition will have the information for when a state should change and are also represented as pointed edges which only goes one way (as the condition for a transition should not apply both ways). States may also change different components associated with the machine such as variables in the system; this is called an action. For example, a state for a shop could be to add a product to a basket, then it would add an item to the basket and add the items price to the total cost of the basket. Transitions can have guards, a guard is the way a state machine will declare that if its conditions are not satisfied the system may not proceed to the state it connects with [9]. Figure 1 shows a finite state machine of how a simple web-shop may operate. This shows that for an added product it will add the item to the items array (the cart) which will increase the total. In the model, an element can add a guard that does not allow access to the checkout page if there are no items are in the cart.

2.3. Behaviour-Driven Development

Behavior-driven development(BDD) [10] is an agile software development process that was created as a response to the issues in Test-Driven development (TDD). TDD is an approach that relies on short development cycles and promotes the writing of automated tests before the functional code, this supposedly leads to improved quality and productivity of the software. The issue with this is that many developers are rather confused while using it, since it does not specify where to start and what to test, or why tests would fail. In this way, the tester is analyzing the state of the SUT rather than the behavior. Hence, BDD was created by focusing on defining specifications for expressing the behavior of the SUT rather than what state it has. BDD tests are also supposed to improve the way the tests are written such that they are more easily understandable. One of the reasons BDD was considered easier to understand was because it used common language where the language structure is based on the business domain. Requirement templates are also included in BDD and use user stories which looks like the following: As a [role], I want [feature], so I can

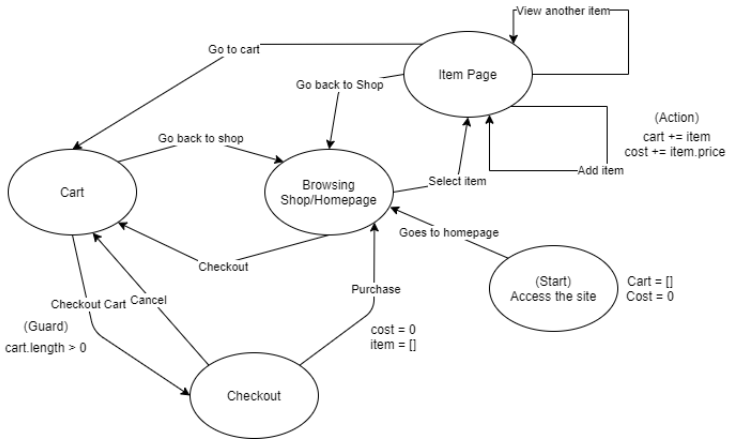

Figure 1: An example of a finite state machine with several states and transitions.

get [benefit]. This is the basis for expressing the template for how scenarios are written. For example, this can be given in a Given-When-Then format, where it will show the context, events, and outcome.

2.4. Model-Based Testing

Model-based testing is a testing technique that is used to either semi- or fully automatically generates tests based on pre-existing models [7]. The pre-existing models can either be manually created or generated, both of which are based on the requirements of the SUT. The models themselves are abstract representations of the systems and need to be explicit in model-based testing [2].

Model-based testing [7] has three stages, which are to design the functional test, determine the test generation criteria, and generate the test. When designing the functional test, the test model must represent the expected operational behavior. Test designers may also use modeling languages such as UML to define the model structure, based on points of control and observation in the system, such as the systems dynamic behavior, the entities associated with the test, and the test data. The model is then linked by elements such as states, transitions, and decisions to fully explore the requirements for a complete test. The models are often based on finite state machines. These contain different states, transitions, events for the different transitions, and guard conditions. Determining the test generation criteria is however important as models can usually generate an infinite number of tests. Ideally, test designers must limit the number of generated tests. A common approach for this test selection is based on the structural-model coverage. The generation of a test in model-based testing is fully automated, with test cases being events of the system, with input parameters, expected output, and return values for each event or action. We outline the MBT process in Figure 2. The model is shown in Figure 1 can be used in this MBT process.

2.5. Given-When-Then

Given-When-Then is a template to structure sentences into test cases, it was created for BDD. It was developed as a part of behavior-driven development by Daniel Terhorst-Nort et. al [6] and appears in several frameworks, and it can also be seen as a reformation of the Four-Phase test pattern [6].

Given-When-Then breaks down scenarios in three sections: the Given part, the When part, and the Then part. The Given part specifies the state of the overall world and sets the pre-conditions, the When part describes the scenario's behavior, and the Then part describes the result of the behavior. Some examples can be seen in Table 1. Example 1 can be read as follows: Given (The system is online), When (The system gets shut down, Then (The system is offline). The when section also does not need to concern all the subjects in the Given and Then part. The Then section describes the side effects on the system. The four-phase pattern [6] is similar to this

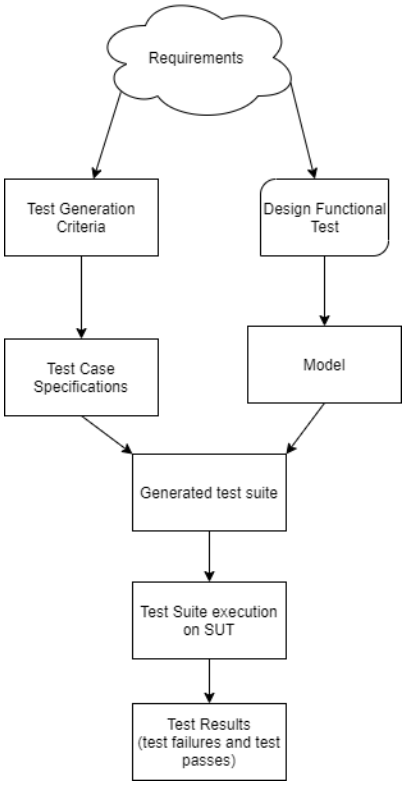

Figure 2: An overall view of MBT process from the creation of test cases and how test results are obtained.

template except it also features a function called tear down, in which it restores the system to its original state.

2.6. Graphwalker

Graphwalker[1] is an open-source tool used to create models in terms of graphs and is able to generate test cases from these graphs which are based on finite state machines. It contains two modules used for this purpose: the CLI tool and the GUI version called Graphwalker studio. The CLI module is a tool for working with graphs. This module has different subsets of commands for its offline mode that can generate a test sequence that can be executed or used to check the correctness of the model. The CLI can only work with already created models and is compatible with formats such as graphml or JSON. The online mode however can connect to the SUT directly and test it dynamically, where the output may be a graphml or a JSON file. The CLI can be used to change which element is the starting point in the model. The models contain different directed paths, which means the tool follows all the alternate paths to create test cases. The Studio version has a graphical interface that can be used to edit models.

The model can contain at least two elements that can be added: vertices and edges. As these models are based on finite state machines, elements may also have actions and guards. The actions will be taken when the execution of the model reaches that element. The guard will not allow access to the element if the condition is not reached. These elements and functions work similarly in the CLI version where the vertices are the states and edges are the directed transitions. If the model reaches a vertex which is protected by a guard, the execution will fail as the model has no state to go next. Studio can also run the model and shows all the different paths the model can take, which also shows all the available test cases. The generation depends on what kind of generator is chosen, which means the generator needs to decide how the tool will generate a path. There are several algorithms that can be used such as the random generator. In this way, the tool

[1] https://graphwalker.github.io/

Table 1: Examples of Given-When-Then requirements

	Example 1	Example 2	Example 3
Given:	The system is online	The system has 100 GB left in the hard drive and 8 GB RAM	The user is not logged in
When:	The system gets shut down	The system install a program that takes 40 GB	The user access the profile page
Then:	The system is offline	The system has 60 GB left in the hard drive and 8 GB RAM	The user is sent to the login page

picks random targets. Nevertheless, the stop condition needs to be stated clearly. For example, one should use the following: all edges, vertices are covered, or other similar conditions.

Graphwalker [9] is based on JavaScript which means the elements are structured using JavaScript objects that contain different information depending on each type of element. All elements have names, actions, guards, and requirements. However, vertices must also contain what position they are in and edges contain which vertices they connect to. An example for how a vertex looks like is shown as follows: $\{"name" = "Vertex","id" = "v1","position" = \{"x" = 0,"y" = 0\},"action" = "", guard = ""\}$. This is placing a vertex named Vertex in position (0,0). Other functions such as updating the element are using the id v1. Edges are declared similarly and have the same structure, but instead of the position, these contain source and target vertices. The model is has a similar structure as the elements but contains a generator, a model name, all the elements, and local variable definitions.

3. Related Work

Anand et al. [11] points out that test case generation has been one of the most active research topics in software testing for more than three decades, and has led to many different techniques and tools, including test generation based model-based methods [12].

Related to this thesis, Marques et al. [2] compared MBT with manual testing. They used two approaches: the manual ad-hoc tests and MBT using a tool named TaRGeT. In this experiment they tested 54 experimental units, the result of the experiment was statistically equivalent in terms of defects found, however the study showed manual testing was better at finding system logic defects, MBT were better at evaluating the system documentation and more complex test cases.

Flemström et. al [13] convert natural language requirements for vehicular systems into test cases. The paper includes a process of five stages to transform these requirements. It starts with the analysis of the requirements and is using the information to create first abstract test cases and then create a prototype before finalizing them. It uses an approach called Guarded Assertions in the form of an executable test case. The results of this study showed how to translate natural language requirements into passive test cases, for vehicular systems. However, this approach needs more development before it could reach industrial use since this work is only describing the transformation process.

Fischbach et. al [14] notes that pseudo-code often appears in requirements and to semi-automate the generation of tests they have proposed to use machine learning to translate segments into a language called Cause-Effect-Graphs(CEG) that can be used to derive test cases. This study makes three contributions: an algorithm that is used to detect appropriate descriptions, an algorithm to translate the requirements into CEG, and a study demonstrating the proposed solution which leads to 86% saving in time without losing test quality.

Carvalho et al. [15] points out that MBT is not often used at the beginning of a project since it needs the input models upfront. They focused on an approach to generate models from a controlled natural language, by starting with an analysis of the requirements and their semantics. This allows the requirements to be represented as a transitions relation and this representation can be used to generate test cases. This approach generated 94% of the tests manually written, but in seconds. However, these results suggest that this approach depends on concrete specifications.

Sarmiento et al. [16] present a tool that implements an approach for generating test cases from natural language requirements. This tool is named C&L and translates natural language into behavioral models and uses these to create test cases. The tool requires a controlled natural language called scenario language. It has been used and evolved by PUC - Rio.

Yue et al. [17] notes that test generation often relies on test case specifications, for both the creation and execution of tests. This requires a lot of time by the test engineers, hence they introduce another test case language named Restricted Test Case Modeling. Secondly, they introduce a tool called aToucan4Test to create both manual- or automatic executable test cases, based on its coverage criteria. These experiments were successfully executed on systems created by Cisco and the report discusses its future uses in an industrial context.

4. A Method for Transforming Given-When-Then Requirements to a Graphwalker Model

4.1. Overall Method

The goal of this thesis is to create and evaluate a tool that can generate a model from natural language requirements written in Given-When-Then format. This model should then generate executable test cases. The proposed method is an extension of Graphwalker, a tool that does not support automatic generation of new models. Graphwalker is extended to accept natural language requirements written in the Given-When-Then template and generate a model based on these requirements.

Graphwalker is currently able to generate models from a JSON file. These JSON files are made up of vertices, how they are connected through edges, and what information the elements contain. This information may contain what kind of actions and guards are used and what kind of generator should be used to run the model. However, the JSON files are often pre-made through the tool itself to obtain a proper structure. The tool is also able to run the model to generate a path, which could be used as a test case [9]. The extension is using the current functions to create and update the models. However, as the generation from JSON files is specific to the structure in these files, it is not be used. Hence, the extension generates the models through Graphwalker's create and update elements functions. Graphwalker was chosen since it is an available open-source MBT tool that is still in development and does not support the generation of new models. There are several other tools that show similar functionality however many are not open-source or don't have all the functionalities of Graphwalker. There are some who also already support automatic test generation such as fMBT[1].

To generate models, Graphwalker is first extended to accept natural language either via manual input or JSON files. Graphwalker is then extended to recognize and use the Given-When-Then template so it is able to create the desired elements and actions. The models can then be used to create test cases through Graphwalkers other module, the CLI.

This method is implemented in a tool supporting the use of natural language requirements written in Given-When-Then and how these can be interpreted as a model and then be used to create test cases. This tool was evaluated based on its ability to create a usable model from natural language requirements. The method was implemented to contribute to the answer of the research question, by giving an insight into how to transform Given-When-Then structured natural language into models.

4.2. Implementation of a Graphwalker extension

This section gives a detailed explanation of how the extension was created, and how the different parts of the extension interact with each other and with Graphwalker. It describes what this thesis has contributed to transforming Given-When-Then requirements into a Graphwalker model.

4.2..1 Implementation Process

The implementation process is divided into the following steps: (1) evaluate the different ways Graphwalker can automatically generate an input model and (2) extend Graphwalker's capability to process textual requirements. Initially, the focus was on developing an initial version of this transformation. The reason for this choice relates to the capability to generate directly from textual information. During this process, we observed that actions and guards can be directly created.

Another step in this process (3) involved the improvement of the text processing such that the input textual requirement could be more general in structure. The fourth step (4) involved the implementation of actions and guards, such that the extension can be used to add variables and change these. Another addition is to block access to certain elements through guards. The last stage (5) of this process involved the development of the processing part that implemented the reading from a certain file. This involved the setting of an action or guard by selecting the element in Graphwalker studio. In order to perform the implementation, time was spent on

[1] https://01.org/fmbt

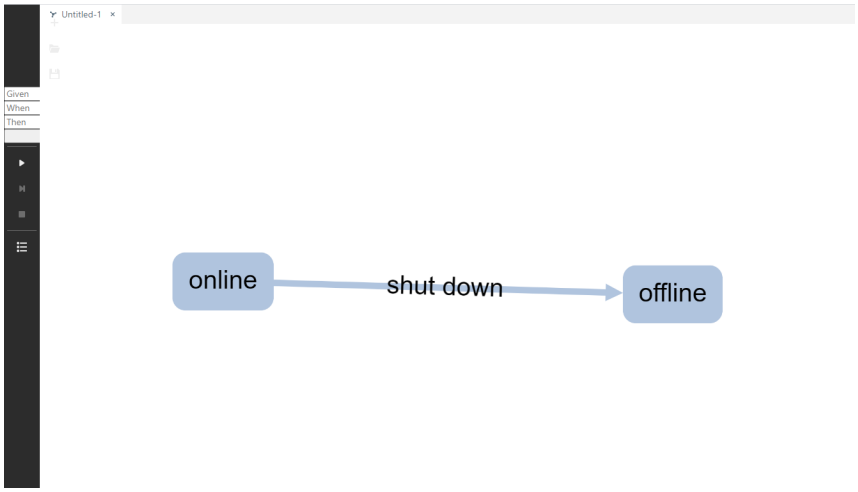

Figure 3: A snapshot showing the initial prototype used to add an element. The other buttons are however displaced.

learning how Graphwalker functions and how to process Given-When-Then requirements. This time was influenced by the fact that Graphwalker is not specifically documented for modifying its existing functionalities and how to use the current source code to extend it. For example, the GIT repository contains several modules, including the studio module which comes paired with the previous version of Graphwalker studio. The structure of the program does not fully explain what parts of the code are connected to certain parts of the GUI. The processing of Given-When-Then requirements has been a difficult task since there is no standard way on how to perform such an analysis. Some of the functions mentioned in the next sections are standard functions of Graphwalker on which the extension is based on, such as createModel or updateElement.

4.2..2 Initial Prototype Implementation

The initial prototype focused on a simplified version of language processing and not all features were implemented. The initial prototype was used for an initial evaluation of how to generate models and elements and did not focus on extending the GUI. I featured text boxes on an existing menu (as shown in Figure 3). The first step focused on generating a model and showed that models could be generated in Graphwalker. At this stage, the elements could not be added so the extension could only add two vertices and one edge. To create a model the extension used a function named createModel, it created a new tab with the added vertices and edge.

The extension only used certain sentences in the form of ”subject is event/state”, where the subject is identified and a function was used to separate the sentence so everything before a certain marker was considered an element. This method was also used for the next step, which was to generate separate elements. Instead of the createModel function, this step used the createElement function. These differ in how they are used as the createModel would always have access to its elements and there was no way of obtaining this information when using createElement. To keep track of all elements and to check if they already exist, all elements were added to their specific array. Using this functionality, the extension was used to create a model without actions. However, vertices could be used to transition to the same state or another state and form a model. The extension uses the names of the vertices to check if they already exist or not and connects them through this.

4.2..3 Updated Version of the Implementation

The initial prototype confirmed the possibility to create an extension that can process structured natural language requirements. To extend this, a partial part-of-speech tagging (POS tagging) [18] was implemented, by letting the extension tag words based on what the rest of the sentence meant. Proper POS tagging will tag every word in the sentence with a certain category. The

partial POS tagger only considers certain words, and based on what kind of tags are included it will determine what the rest of the sentences means. For example, the first example in Table 1 describes a transition, while example 2 describes an action in a transition. The next section will cover in more depth how the extension interprets sentences.

Another functionality of this extension related to the support for actions and guards. This was implemented by using a function called updateElement. This allows the selected model to be updated with an action or guard, taken from directly from the sentences. The words used for these tasks differ from states and transitions as they describe the actual behaviors of states and transitions. Actions and guards also require the variables to already be recorded. This was implemented by adding them to arrays since the extension needed to check if any such variable existed in the model. In the case were no variables with the specified name existed, the plugin created the specified variable. Also, the extension limits the Given part in the sense that it can only assign a new variable or change its starting value. This was done by updating the model through the function updateModel which did not need a specified element. However, the When and Then parts of a requirement needed a specified element to target and used updateElement. Guards belong to the When part of a requirement as it often describes a certain condition. Actions belong to the Then part of a requirement as these describe an outcome of an event. This deviates from the usual Given-When-Then format as the When part may also describe an action.

The last part of creating the implementation involved the creation of a file reader. This is currently limited to JSON files since it fetches the data written in a Given-When-Then format. It can then use the same functions as for when the input comes from text boxes. As the extension needs to focus on an element, it cannot create the desired action or guard if it does not have access to either the element's id or name. Since every action or guard would need the name of the desired element, the element could be selected by going through all elements and matching the name to obtain the identification id. This id was necessary to obtain as Graphwalker's function selectElement is using this id.

4.2..4 Processing the Natural Language Requirements and Using the Extension

The extension is processing the Given-When-Then requirement parts separately. For every part of the requirement, the extension is using a different set of words and a different processing of these words. The transformation starts with the interpreter, which will recognize what type of word based on a couple of predefined word lists. For example, if it detects a verb it will recognize that the sentence is intended to declare a state. This will make the rest of the process assume the model needs to create a new transition. We note here, that the words used later in the sentence can change this assumption.

The next part of this process is the word tagging which decides the meaning of the sentence. The sentences are the Given-When-Then requirements. Figure 4 shows the implemented method operates the word tagging and how it uses the resulting dictionary. The process goes through several stages. The first part is involved when the requirement is used in the main function modelUpdater (shown as 1 in Figure 4). This function calls the implemented Dictionary function (shown as 2 in Figure 4, function is shown in Appendix B) and it is used to interpret the sentence. To perform this it sends the words to the interpreter (3 in Figure 4). The interpreter will assign tags to each word. The tag depends on the word lists or if its a number or logical parameter (e.g., true and false). These values are returned to the Dictionary, which will use it to form a structure of the sentence. This means that it will process the tags to observe what category each word belongs to. The Dictionary may also override the tags from the interpreter. If a process starts (e.g., action, guard, or transition) the Dictionary only tags the most relevant words and assigns the others with an *"other"* tag. For example, the function is searching for a number if it encounters a previous word such as *"higher"* or "<" and assigns the most likely word as the subject of this function. The subject is used to determine what element should be changed. This statement is interpreted as a guard (e.g., *"higher"* or "<" is a comparison). This is then used to determine new tags as part of the function (shown as 5 in Figure 4). Exactly how it determines the new tags are in Appendix B and shows that the extension mostly considers what has happened before the current word. Certain situations may however allow words that are after the current word to change, such as if its a guard it will find the subject through looking after words tagged with *"other"* and assign

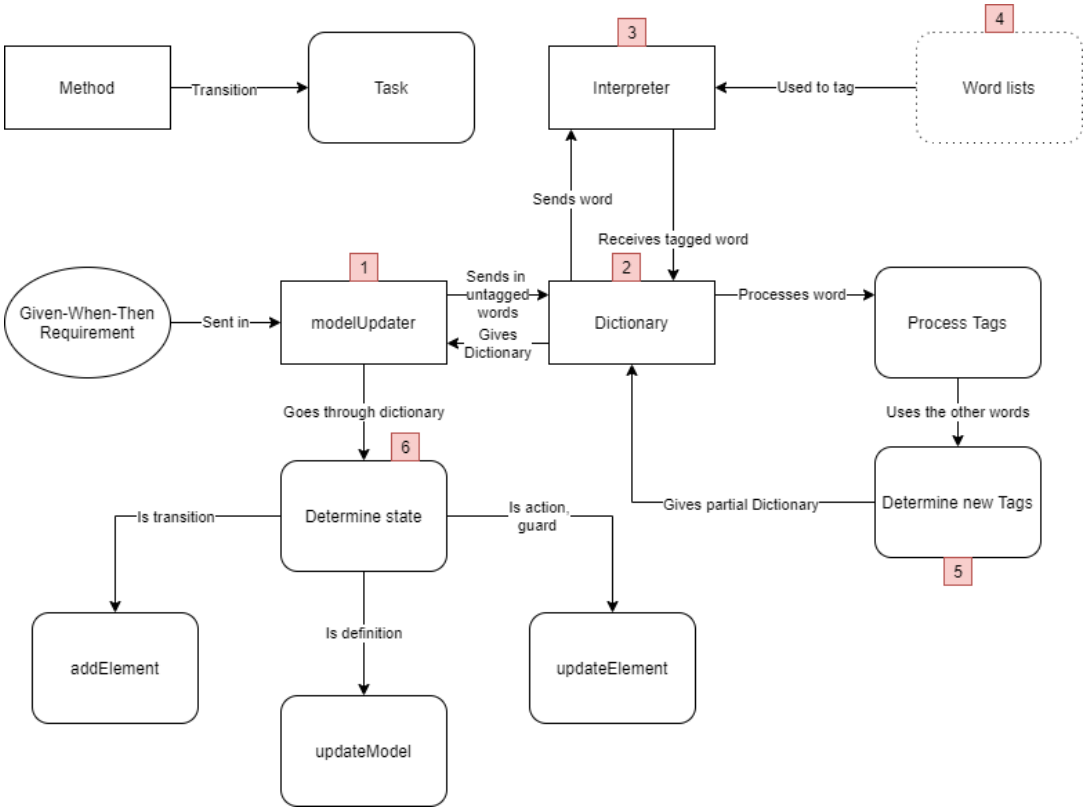

Figure 4: An overall view of how the word tagging system and the main model updater is using the dictionaries for text to model transformation.

it to be the subject. This gives the function a partial Dictionary that contains the words paired with its tag. This function loops through the sentence. When the Dictionary is completed, it sends this information back to the main function. The extension use three dictionaries: one for the Given statement, another one for the When statement, and the last one for the Then statement of each requirement. Appendix B shows how the Dictionary also makes distinctions based on what statement the sentence if for, which is retrieved from the main function. The main function is used to understand the sentence purpose (shown as 6 in 4). This step leads to three outcomes: either it adds an element, updates the model, or updates an existing element. This is decided depending on the dictionaries tags. The extension assumes the use of a transition. If there is no action, definition, or guard, this leads to the use of the addElement function. In the case of a definition, it leads to updateModel where the default value is (0 or false). When the extension finds an action or guard, it is using the updateElement function. The Given, When, Then statements give different interpretations. Since the Given part gives a state and creates a definition, the When statement gives an event and can only create a guard. The Then statement gives the state after the event and can only create an action. This was assumed since it makes it clearer what kind of elements the statements could create.

As an example, the extension is processing the sentence *"The system is online"* in the following manner: the main function sends in the sentence to the dictionary. The dictionary sends it to the interpreter and receives the tag for each word. In this case, it would obtain *"The:presubject, system:other, is:state, online:other"*. The dictionary would then process the words and assign new tags. The sentence would be tagged as: *"The:presubject, system:subject, is:state, online:other"* and this information is sent back to the main function as a Dictionary. The word *"system"* had its tag changed as the previous word is recognized as a *"presubject"*. When returned the main function, the extension is recognizing this sentence as a state and will add a transition if the rest of the statements are not considered to be actions or guards and if the rest of the statements are not empty.

Table 2 shows the commands that are implemented in the extension. It states what word list

Table 2: Overview of the commands implemented in the transformation extension.

	Given	When	Then	Example 1	Example 2
Verb(State Of Being)	Assumes new transition	Assumes new transition	Assumes new transition	The System (is) online	The system (opens) the file
Assignment	Defines variable with value	-	Assigns new action	var1 is (set) to true	login (=) true
Comparisons	-	Assigns new guard	-	var1 is (higher) than 1	login (==) true
Operations	-	-	Assigns new action	var1 adds 1	var1 += 1

belong to which command. These word lists are limited to certain categories shown as follows:

- Verb(state of being) = is, are, shows, gets, etc.
- Assignment = set, sets. =
- Comparisons = higher, lower, equals, ==, ¡, ¿=, etc.
- Operations = adds, subtracts, *=, /=, etc

The extension is also able to distinguish if the requirement contains any numbers or logic (true or false) and this affects what the extension assumes about the sentence.

4.2..5 The Proposed Extension and the Use of Graphwalker

Figure 5 shows an overview of how Graphwalker studio works and which parts the extension is using. For example, the extension cannot be used if there is no existing model created. The extension does not support the opening of a new tab since this gives conflicts in the lists of elements and variables. As shown in Figure 5, some functions enable the use of other functions. For example, create element enables deletion and updating of that element. Also, when running the model the extension is using the specified generator that can be changed in the model. In addition, its stop conditions need to be specified to run and display a test sequence. We also only show the most relevant features and not others since Graphwalker could also use other tabs, link together models, and more functions.

The functions themselves need different inputs. As mentioned before some also need the element to be selected beforehand. The first functions, as shown in Figure 5, can either be to create model or load model. These will either create a new model or load an existing one. These functions create or generate a starting model. Create model does not need any input but will return a JavaScript object in the form of the model. The object contains the name, id, generator, definitions (called actions), edges, and vertices. It is contained in a similar structure as the vertex element described in the Background Section. One difference is that the generator is in the following format: "generator" = "algorithm(stop condition)". This shows what algorithm will be used and what stop conditions are set for the model. The create element function needs to receive the element that it should create. An example of how this looks like is shown in the Background section, as the vertex mentioned above. Delete requires the element to be selected and this allows access to its id and will delete this specific element. Update element also requires the element to be selected but will require input in the form of ("target", operation). The target is what field the update wants to change and operation what the field should perform. An example of this operation is updateElement('action', ("var1=true;").split("
n"));. The split is necessary as it will convert the action to an array. This is used to extend the current list of actions. This also shows how the model is updated. However, as this is the definition of a variable, it can only assign a value, as an action in an element can assign an operation such as "+=" to the actions field. Both fields are also called actions, where the guard has a separate field. Graphwalker can also save the current model. This will download a JSON file and can be used to load the model. This can also be used in the CLI parts of Graphwalker to obtain a test sequence

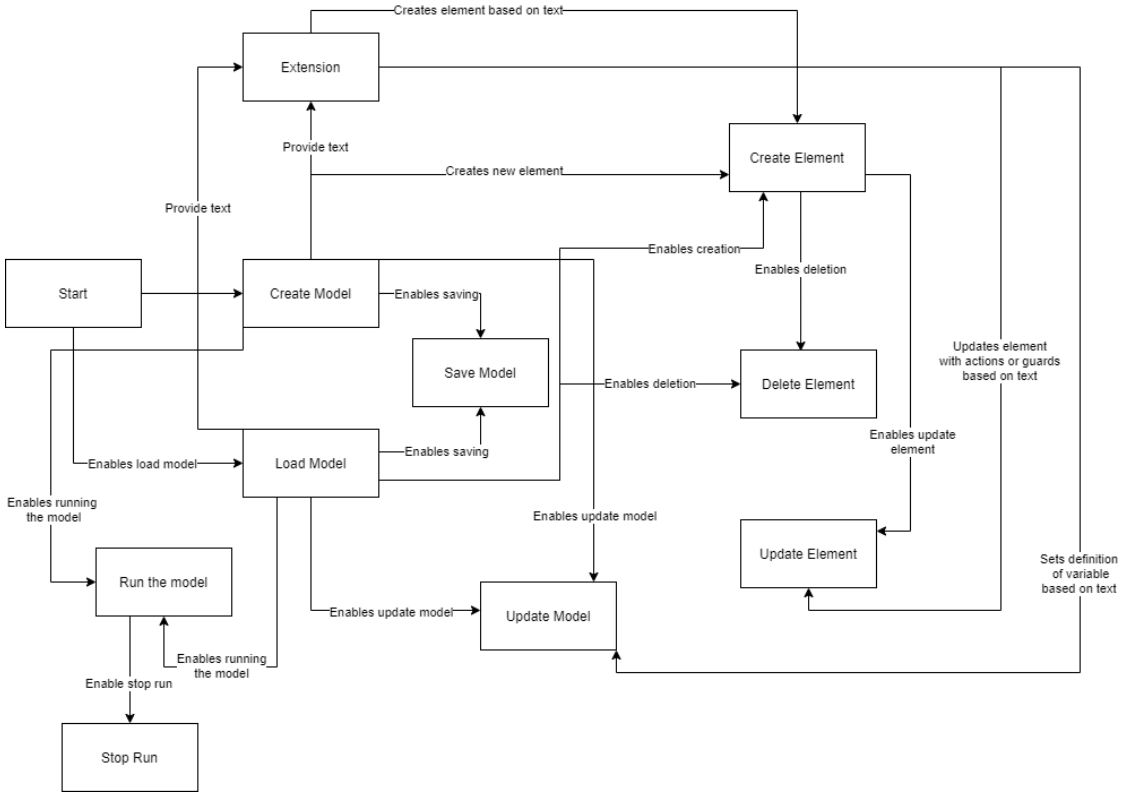

Figure 5: An overview of the proposed extension and how Graphwalker operates and enables different features.

or check its viability. This can also be checked in the studio GUI. However, this does not give any output, except for the visual information shown on the screen.

4.2..6 Limitations

Currently, there are certain limitations on using this extension. One of them relates to the inability to recognize what element is tied to an action or guard through text. Also, the "and" operators in the Given-When-Then are used to extend the statement's functionality with additional tasks, so it can add two or more actions at the same time. This needs to be taken into account to be able to handle more complex requirements.

Both these mentioned issues are related to the Given-When-Then template and are left for future work. The current language processing is also limited. However, as this is a process that is widely researched, it can probably be improved in different ways.

The extension does not support all features available in Graphwalker. For example, the method can have multiple tabs for one model, and these are connected through a field called shared names. For this work, this does not add much functionality except for the visual representation of the models.

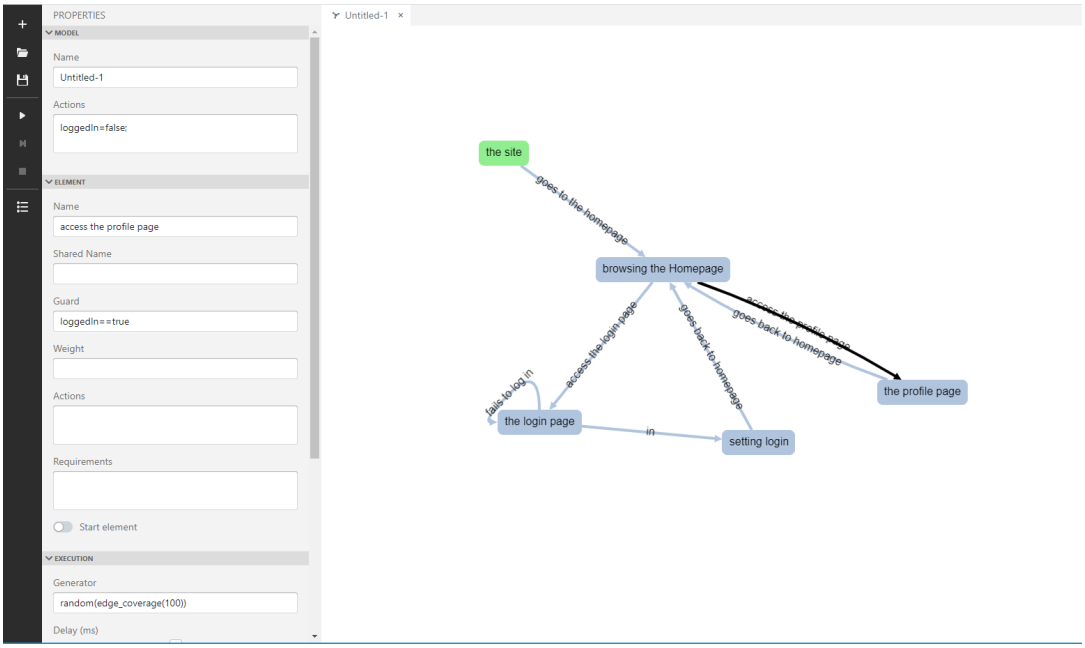

Figure 6: A generated model that was generated from the JSON in Appendix 7.

5. Experimental Results

The main result of this thesis is an extension[2] of the Graphwalker tool for MBT. Based on this work, this tool can process structured natural language requirements written according to a certain Given-When-Then template and create a model based on these requirements. This extension is only used for Graphwalker studio as we could not find a way for the CLI version to create models.In this thesis we have shown the description of how the extension was implemented. Currently, the extension, is able to create a model with vertices, edges, actions, and guards. This allows the creation of a finite state machine, which is the basis of using Graphwalker for model-based testing. This extension shows that natural language requirements can be used in the creation of Graphwalker models and that natural language processing could be included in Graphwalker. We have described how to generate models in Graphwalker as this tool could not directly generate a model from outside the GUI interface. The extension is also able to generate graphs from JSON files such as the one shown in Appendix 7. In this section we show the experimental results of using this extension on a real example as shown in Figure 6. This model was generated in 175 milliseconds, the others are a bit smaller and took around 100 milliseconds to generate. This shows that this extension is efficient in transforming realistic requirements. In Figure 6 we can observe what elements are selected in the black border and that the green-colored vertex is the start element. We also show the resulting test sequence obtained from test generation using the model in Figure 6. We ran Graphwalker on the obtained model in Figure 6 with a random generator that should cover all edges and obtained the following result:

1. {"currentElementName":"the site "}
2. {"currentElementName":"goes to the homepage "}
3. {"currentElementName":"browsing the Homepage "}
4. {"currentElementName":"access the login page "}
5. {"currentElementName":"the login page "}
6. {"currentElementName":"in "}

[2]https://github.com/JocksTheGul/graphwalkerGiven-When-Then

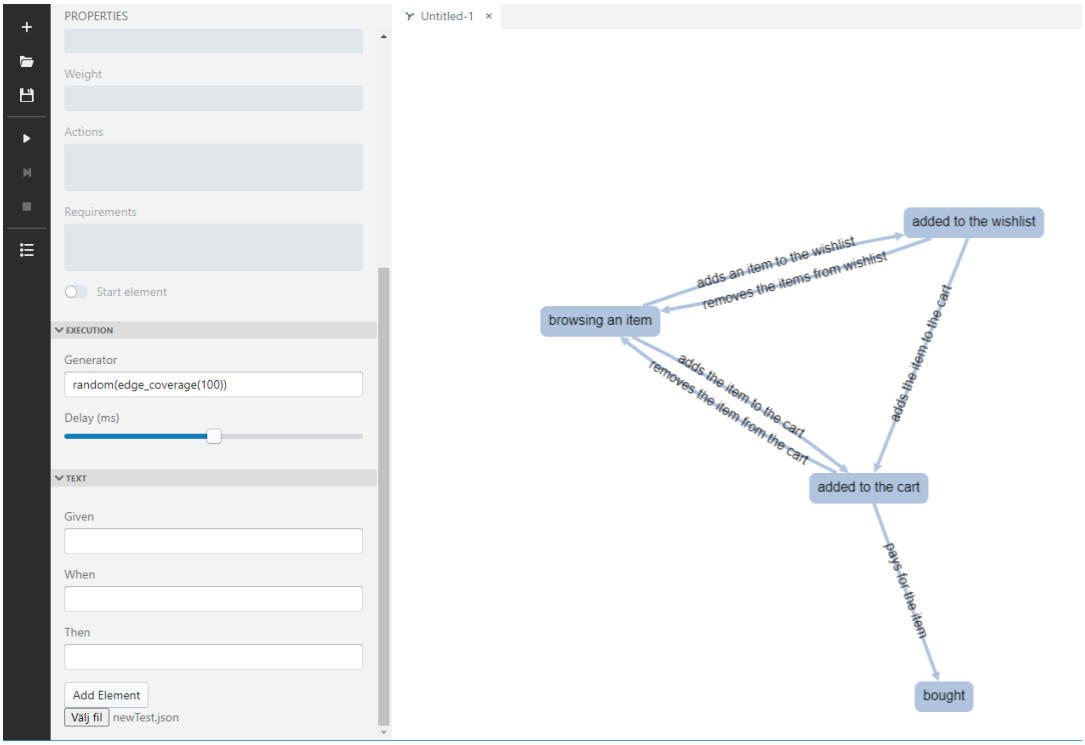

Figure 7: A generated model showing the result of the model generation for a correct version of an example model

7. {"currentElementName":"setting login "}
8. {"currentElementName":"goes back to homepage "}
9. {"currentElementName":"browsing the Homepage "}
10. {"currentElementName":"access the login page "}
11. {"currentElementName":"the login page "}
12. {"currentElementName":"fails to log in "}
13. {"currentElementName":"the login page "}
14. {"currentElementName":"in "}
15. {"currentElementName":"setting login "}
16. {"currentElementName":"goes back to homepage "}
17. {"currentElementName":"browsing the Homepage "}
18. {"currentElementName":"access the profile page "}
19. {"currentElementName":"the profile page "}
20. {"currentElementName":"goes back to homepage "}
21. {"currentElementName":"browsing the Homepage "}

This result was obtained using the Graphwalker CLI as the Studio version cannot be used to extract test sequences. Nevertheless, this version can be used to visually show a sequence in the model editor. The model was imported by saving the file and using it in the CLI version.

In Figure 7 and 8 we show some of the limitations of this model generation. The extension will not be able to interpret correctly when some of the sentences are changed. They show how

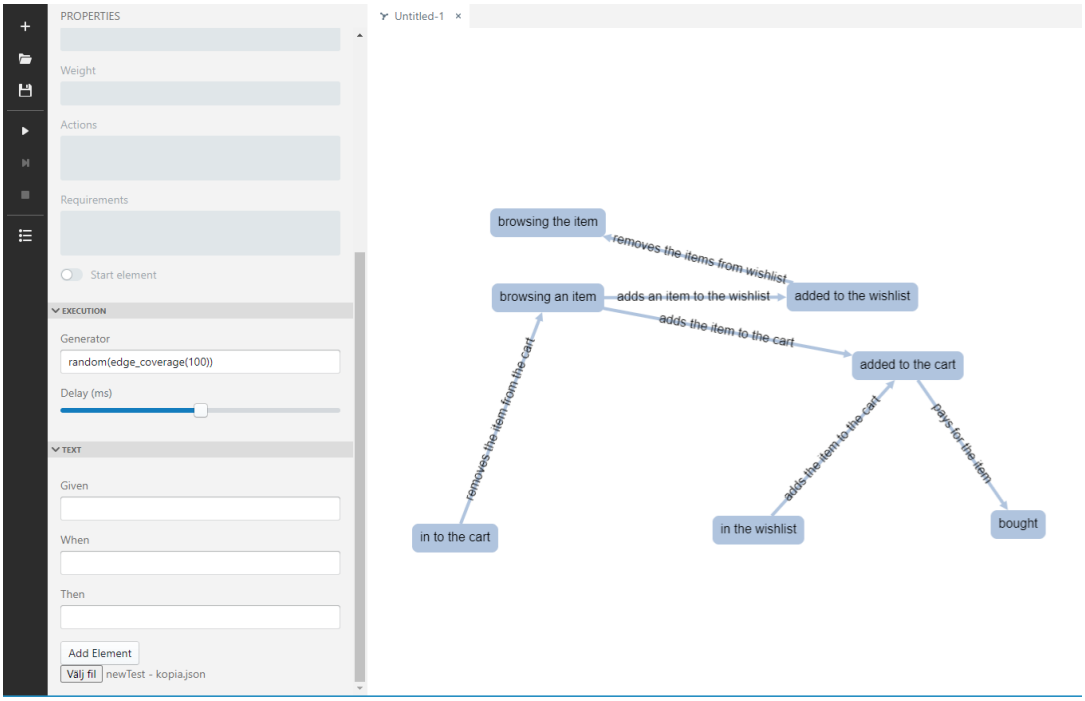

Figure 8: A generated model showing the result of a failed version of an example model due the use of different terms when referring to the same vertex.

a system might act for the state of an item. They do however differ in some ways. For example, if a statement is added stating "The item is added to the wishlist", a vertex called "added to the wishlist" will be created. The safest way to use this vertex again would be to use the same sentence. However, if we refer to it using a sentence such as "The item is in the wishlist", which actually is a more accurate statement according to the Given-When-Then template and it would create another vertex and transition to it instead. Figure 8 shows the correct statements according to the Given-When-Then template. In addition, Figure 7 shows the statements adapted to the extension. The exact differences can be seen also in Appendix B and Appendix B.

Another limitation of this extension relates to the inability to recognize certain statements, which happens when these do not contain the keywords in the word-lists. In Figure 9 and 10 we show two models of a game model, one of which shows when the extension cannot recognize a statement. When this happens, the model generation simply does not add the transition and the subsequent vertices. The only difference in the used statement is "The game starts" (in the failed version) and "The game is started" (in the adapted version). The correct requirement for these models can be seen in Appendix B.

5.1. Ethical and Societal Considerations

This thesis does not handle any sensitive data and is based on developing an extension of an open-source software. As this thesis builds upon previous research it is not involving any human subjects. However, it can help testers and developers during the test creation process, which could reduce the time and cost it would require to use model-based testing.

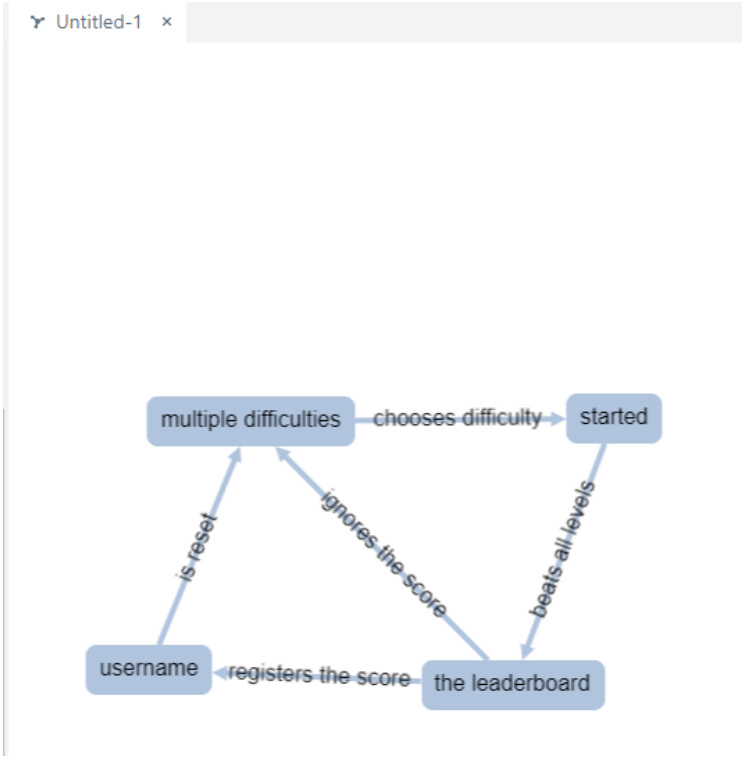

Figure 9: An example of a correct model for a game leader-board.

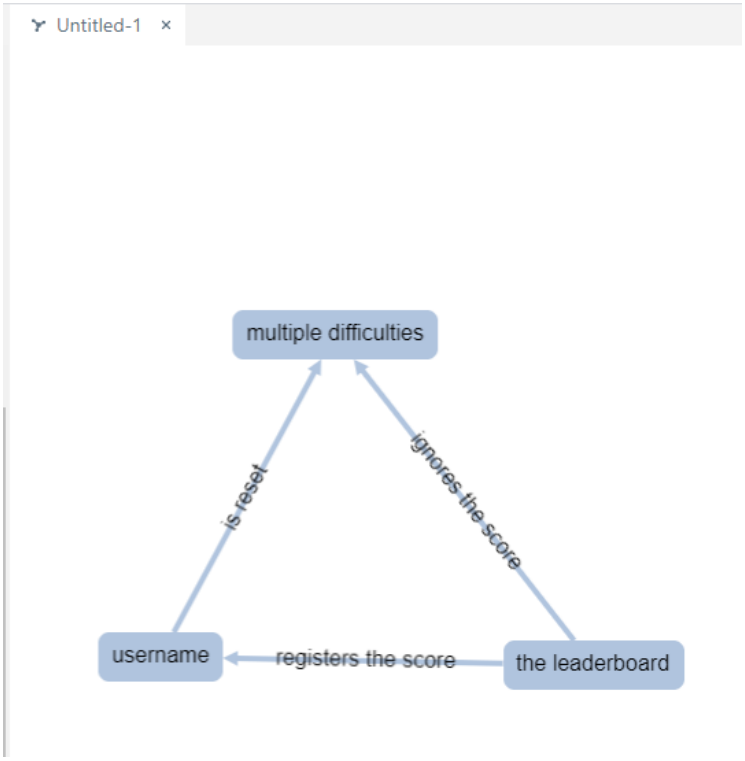

Figure 10: An incorrect model of a leader game, when the extension cannot recognize a statement.

6. Discussion

The proposed extension enables Graphwalker to generate models from structured natural language written according to the Given-When-Then template. These models are able to change their internal values in their elements through the text in the requirement. This contributes to the goal of the thesis since it is focusing on a limited form of Given-When-Then templates. There are features, such as support for the "and" operator, that are not supported in the current version of this extension. This would require the extension to increase its modularity which would improve the extension's overall quality, by letting different parts handle what currently the main function is performing. The extension is letting Graphwalker create elements and add actions or guards to them. We do not change the structure of the program except the addition of a couple of HTML elements. The natural language processing part is limited to certain phrases and could be improved. For reaching a more generic and industrial use, this extension needs to be further developed. The word tagging can also be improved by connecting it to a dictionary. The whole language processing might also be improved by letting another system interpret the meaning of these sentences. We observed some limitations when working with Graphwalker. As there was no known way of accessing the current model as a whole, this would allow the use of tabs and it would give easier access to existing elements. The other limitation is related to the way the extension could automatically place vertices. This was solved by simply counting the current elements and calculating the position of each element

We argue here that these results could also inspire other researchers on how to process these Given-When-Then requirements and how to use them. This work could also be used in other projects that do not include MBT, as Given-When-Then can be used in most approaches for software testing and is often used in BDD. It could also help the development of Graphwalker to be able to generate models out of text or other formats. As currently, this tool is only able to create models manually or trough an existing file created by similar tools or the tool itself.

6.1. Future Work

This extension could be improved in several ways. As described already, it could use a more advanced system to handle the natural language processing. The suggestion for this would be Gherkin [19], as this system can process Given-When-Then requirements. For this to work, one would need to state a feature the current statements are supposed to create. As it would improve how words are processed and give the extension support for proper Given-When-Then requirements, this system would also be able to use the information gathered to automatically detect what element the statement is referring to.

As mentioned above, this extension could also use the ability to access the current model. This could also assist in placing the elements in the GUI interface, as the current way of placing vertices is not taking into account the visual representation. This could be used to identify which model works and would solve the conflicting lists of elements and variables. This could enable the extension to create multiple models and could connect these.

The implementation doesn't have a comparison of how well it does compared to manual testing, this could be done over an openly available program that gives access to a ready-to-use test suite. Then the extension could be used to create a test suite of this program to see how well it would do against the manual one.

7. Conclusions

In this thesis, we propose and create an extension to Graphwalker[9] that could generate models out of natural language requirements written according to the Given-When-Then template[6]. Given-When-Then is used to structure sentences to allow easier transformation to test cases and is structured as a scenario that gives the context, the event, and the result of the event. The automated transformation begins with processing a sentence and tags every word. These words can affect the other words tags depending on what the extension specifies the sentence is supposed to include. The tagged words are then used to determine the meaning of all the sentences specified in Given-When-Then. We contribute to the state of the art by creating Graphwalker models from textual descriptions. The elements we can create are related to both vertices and edges, who have the ability to change internal values in a system. These can also block access to certain elements forming a graph [8].

The implementation of this extension can be improved for industrial use. A better way of processing of words should be able to properly use the Given-When-Then template. Most of the issues in the extension could be solved if another system called Gherkin [19] could be integrated. This would allow the extension to have a proper interpretation of the sentences and will give support for proper Given-When-Then requirements, as this is what Gherkin is processing.

References

[1] C. Wang, F. Pastore, A. Goknil, L. Briand, and M. Z. Iqbal, "Automatic generation of system test cases from use case specifications," 07 2015, pp. 385–396.

[2] A. Marques, F. Ramalho, and W. L. Andrade, "Comparing model-based testing with traditional testing strategies: An empirical study," pp. 264–273, 2014.

[3] GeeksforGeeks, "Software testing: Basics," Apr 2019. [Online]. Available: https://www.geeksforgeeks.org/software-testing-basics/

[4] L. Villalobos-Arias, C. Quesada-López, A. Martinez, and M. Jenkins, "Model-based testing areas, tools and challenges: A tertiary study," *CLEI Electronic Journal*, vol. 22, no. 1, 2019.

[5] S. Kamalakar, S. Edwards, and T. Dao, "Automatically generating tests from natural language descriptions of software behavior," *ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 238–245, 01 2013.

[6] M. Fowler, "bliki: Givenwhenthen," Aug 2013. [Online]. Available: https://martinfowler.com/bliki/GivenWhenThen.html

[7] I. Schieferdecker, "Model-based testing," *IEEE Software*, vol. 29, pp. 14–18, 01 2012.

[8] A. C. Pinheiro, A. Simapoundso, and A. M. Ambrosio, "FSM-Based Test Case Generation Methods Applied to Test the Communication Software on Board the ITASAT University Satellite: A Case Study," *Journal of Aerospace Technology and Management*, vol. 6, pp. 447 – 461, 12 2014. [Online]. Available: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S2175-91462014000400447&nrm=iso

[9] K. Karl and N. Olsson, "Graphwalker/graphwalker-project." [Online]. Available: https://github.com/GraphWalker/graphwalker-project/wiki

[10] C. Solis and X. Wang, "A study of the characteristics of behaviour driven development," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2011, pp. 383–387.

[11] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[12] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[13] F. Daniel, E. Eduard, A. Wasif, S. Daniel, G. Thomas, and K. Avenir, "From natural language requirements to passive test cases using guarded assertions," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 470–481.

[14] J. Fischbach, M. Junker, A. Vogelsang, and D. Freudenstein, "Automated generation of test models from semi-structured requirements," 20190822.

[15] G. Carvalho, F. Barros, F. Lapschies, U. Schulze, and J. Peleska, "Model-based testing from controlled natural language requirements," vol. 419, 04 2014, pp. 19–35.

[16] E. Sarmiento, J. C. S. d. P. Leite, and E. Almentero, "C l: Generating model based test cases from natural language requirements descriptions," in *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*, 2014, pp. 32–38.

[17] T. Yue, S. Ali, and M. Zhang, "Rtcm: a natural language based, automated, and practical test case generation framework," 07 2015, pp. 397–408.

[18] sketchengine, "Pos tags and part-of-speech tagging," Oct 2019. [Online]. Available: https://www.sketchengine.eu/pos-tags/

[19] A. Hellesøy, J. Wilk, M. Wynne, G. Hnatiuk, and M. Sassak, "Gherkin reference," 2020. [Online]. Available: https://cucumber.io/docs/gherkin/reference/

A JSON Test


```
{
  "requirements": [
    {
      "given": "The browser access the site",
      "when": "The browser goes to the homepage",
      "then": "The browser is browsing the Homepage",
      "name": ""
    },
    {
      "given": "The browser is browsing the Homepage",
      "when": "The user access the login page",
      "then": "The browser shows the login page",
      "name": ""
    },
    {
      "given": "The browser shows the login page",
      "when": "The user successfully log in",
      "then": "The site is setting login",
      "name": ""
    },
    {
      "given": "The browser shows the login page",
      "when": "The user fails to log in",
      "then": "The browser shows the login page",
      "name": ""
    },
    {
      "given": "The site is setting login",
      "when": "The site goes back to homepage",
      "then": "The browser is browsing the Homepage",
      "name": ""
    },
    {
      "given": "The browser is browsing the Homepage",
      "when": "The user access the profile page",
      "then": "The browser shows the profile page",
      "name": ""
    },
    {
      "given": "The browser shows the profile page",
      "when": "The site goes back to homepage",
      "then": "The browser is browsing the Homepage",
      "name": ""
    },
    {
      "given": "loggedIn = false",
      "when": "",
      "then": "",
      "name": ""
    },
    {
      "given": "",
      "when": "",
      "then": "loggedIn = true",
      "name": "setting login"
    },
    {
      "given": "",
      "when": "loggedIn == true",
      "then": "",
      "name": "access the profile page"
    }
  ]
}
```

B Dictionary

```
modelDictionaryALT = (text, mode) => {
    var dictionary = Object.create(null);
    var dictionaryArr = [];
    var state = false;
    var action = false;
    var guard = false;
    var arr = text.split(" ");
    for (let i = 0; i < arr.length; i++) {
      var arrType = this.interpreter(arr[i]);
      if (mode === status.GIVEN) {
        if (state && arrType !== type.PREACTION) {
          dictionary[arr[i]] = type.OTHER;
        }
        else {
          if (arrType === type.STATE) {
            dictionary[arr[i]] = type.STATE;
            state = true;
          }
          else if (arrType === type.PREACTION) {
            dictionary[arr[i]] = type.PREACTION;
            state = false;
            action = true;
          }
          else if (arrType === type.PRESUBJECT) {
            dictionary[arr[i]] = type.PRESUBJECT;
            dictionaryArr.push([arr[i], dictionary[arr[i]]]);
            i++;
            dictionary[arr[i]] = type.SUBJECT;
          }
          else {
            if (dictionary[arr[i - 1]] === type.SUBJECT && arr[i].slice(-1) === 's') {
              dictionary[arr[i]] = type.STATE;
              state = true;
            }
            else {
              dictionary[arr[i]] = this.isValue(arr[i]);
            }
          }
        }
      }
      else if (mode === status.WHEN) {
        if (state && arrType !== type.GUARD) {
          dictionary[arr[i]] = type.OTHER;
        }
        else {
          if (arrType === type.STATE) {
            if ((dictionaryArr[0][1] === type.OTHER || dictionaryArr[0][1] ===
type.SUBJECT) && arr[i] === 'is') {
              dictionary[arr[i]] = type.GUARD;
              state = false;
              guard = true;
            }
            else {
              dictionary[arr[i]] = type.STATE;
              state = true;
            }
          }
          else if (arrType === type.PREACTION) {
            dictionary[arr[i]] = type.GUARD;
            state = false;
            guard = true;
          }
```

```
          else if (arrType === type.GUARD) {
            dictionary[arr[i]] = type.GUARD;
            state = false;
            guard = true;
            if (dictionaryArr[i - 1][0] === "is" && dictionaryArr[i - 1][1] ===
type.GUARD) {
              dictionaryArr[i - 1][1] === type.OTHER;
            }
          }
          else if (arrType === type.PRESUBJECT) {
            dictionary[arr[i]] = type.PRESUBJECT;
            dictionaryArr.push([arr[i], dictionary[arr[i]]]);
            i++;
            dictionary[arr[i]] = type.SUBJECT;
          }
          else {
            if (dictionary[arr[i - 1]] === type.SUBJECT && arr[i].slice(-1) === 's') {
              dictionary[arr[i]] = type.STATE;
              state = true;
            }
            else {
              dictionary[arr[i]] = this.isValue(arr[i]);
            }
          }
        }
      }
      else if (mode === status.THEN) {
        if (state && arrType !== type.PREACTION && arrType !== type.OPERATIONS) {
          dictionary[arr[i]] = type.OTHER;
        }
        else {
          if (arrType === type.STATE) {
            dictionary[arr[i]] = type.STATE;
            state = true;
          }
          else if (arrType === type.PREACTION) {
            dictionary[arr[i]] = type.PREACTION;
            state = false;
            action = true;
          }
          else if (arrType === type.OPERATIONS)
          {
            dictionary[arr[i]] = type.OPERATIONS;
            action = true;
            state = false
          }
          else if (arrType === type.PRESUBJECT) {
            dictionary[arr[i]] = type.PRESUBJECT;
            dictionaryArr.push([arr[i], dictionary[arr[i]]]);
            i++;
            dictionary[arr[i]] = type.SUBJECT;
          }
          else {
            if (dictionary[arr[i - 1]] === type.SUBJECT && arr[i].slice(-1) === 's') {
              dictionary[arr[i]] = type.STATE;
              state = true;
            }
            else {
              dictionary[arr[i]] = this.isValue(arr[i]);
            }
          }
        }
```

```
    }
    dictionaryArr.push([arr[i], dictionary[arr[i]]]);
  }
  if ((action === true || guard === true) && dictionaryArr[0][1] === type.OTHER) {
    dictionaryArr[0][1] = type.SUBJECT;
  }
  console.log("Model DictionaryALT: ");
  console.log(dictionaryArr);
  return dictionaryArr;
}
```

C Correct Model Item


```
{
  "requirements": [
    {
      "given": "The user is browsing an item",
      "when": "The user adds an item to the wishlist",
      "then": "The item is added to the wishlist",
      "name": ""
    },
    {
      "given": "The item is added to the wishlist",
      "when": "The user removes the items from wishlist",
      "then": "The user is browsing an item",
      "name": ""
    },
    {
      "given": "The item is added to the wishlist",
      "when": "The user adds the item to the cart",
      "then": "The item is added to the cart",
      "name": ""
    },
    {
      "given": "The user is browsing an item",
      "when": "The user adds the item to the cart",
      "then": "The item is added to the cart",
      "name": ""
    },
    {
      "given": "The item is added to the cart",
      "when": "The user removes the item from the cart",
      "then": "The user us browsing an item",
      "name": ""
    },
    {
      "given": "The item is added to the cart",
      "when": "The user pays for the item",
      "then": "The item is bought",
      "name": ""
    },
    {
      "given": "The item is added to the wishlist",
      "when": "The user removes the items from wishlist",
      "then": "The user is browsing an item",
      "name": ""
    }
  ]
}
```

D Failed Model Item


```
{
  "requirements": [
    {
      "given": "The user is browsing an item",
      "when": "The user adds an item to the wishlist",
      "then": "The item is added to the wishlist",
      "name": ""
    },
    {
      "given": "The item is added to the wishlist",
      "when": "The user removes the items from wishlist",
      "then": "The user is browsing the item",
      "name": ""
    },
    {
      "given": "The item is in the wishlist",
      "when": "The user adds the item to the cart",
      "then": "The item is added to the cart",
      "name": ""
    },
    {
      "given": "The user is browsing an item",
      "when": "The user adds the item to the cart",
      "then": "The item is added to the cart",
      "name": ""
    },
    {
      "given": "The item is in to the cart",
      "when": "The user removes the item from the cart",
      "then": "The user is browsing an item",
      "name": ""
    },
    {
      "given": "The item is added to the cart",
      "when": "The user pays for the item",
      "then": "The item is bought",
      "name": ""
    }
  ]
}
```

E Game


```
{
  "requirements": [
    {
      "given": "The Game presents multiple difficulties",
      "when": "The user chooses difficulty",
      "then": "The game is started",
      "name": ""
    },
    {
      "given": "The game is started",
      "when": "The user beats all levels",
      "then": "The game shows the leaderboard",
      "name": ""
    },
    {
      "given": "The game shows the leaderboard",
      "when": "The user registers the score",
      "then": "The Game takes username",
      "name": ""
    },
    {
      "given": "The game takes username",
      "when": "The game is reset",
      "then": "The Game presents multiple difficulties",
      "name": ""
    },
    {
      "given": "The game shows the leaderboard",
      "when": "The user ignores the score",
      "then": "The Game presents multiple difficulties",
      "name": ""
    }
  ]
}
```