Model Based Testing and Model Checking: An Efficient Combination

by

MISHRA ROHIT AJAYKUMAR
202111070

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

June, 2023

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

MISHRA ROHIT AJAYKUMAR

Certificate

This is to certify that the thesis work entitled **Model Based Testing and Model Checking: An Efficient Combination** has been carried out by **Rohit Mishra** for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my/our supervision.

Dr. Saurabh Tiwari
Thesis Supervisor

Acknowledgments

I want to express my deepest gratitude to my supervisor Dr. Saurabh Tiwari, whose guidance and support have been invaluable throughout my research journey. He provided me with insightful feedback, encouraged me to think critically and pushed me to challenge myself every step of the way.

I would also like to thank the faculty members for providing me with a solid academic foundation and for their constant encouragement and motivation.

I want to thank my family for their unwavering love and support and for always believing in me, even during my most challenging moments.

Finally, I would like to thank my friends and colleagues who have provided me with a support network and whose intellectual contributions have greatly enriched my research.

I am grateful for the support of all those who have played a role in my academic and personal development and without whom this achievement would not have been possible.

Contents

Abstract v

List of Principal Symbols and Acronyms vi

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 MBT and Model Analysis . . . 1

1.1.1 Model-Based Testing . . . 1

1.1.2 Model Analysis . . . 2

1.1.3 Combining MBT and Model Analysis . . . 3

1.2 Objective and Problem Description . . . 4

1.3 Thesis Contribution . . . 5

1.4 Organisation of the Thesis . . . 5

2 Preliminaries 7

2.1 GraphWalker . . . 7

2.2 UPPAAL Model Checker . . . 10

2.3 GW2UPPAAL . . . 13

3 Related Work 15

4 Analysis Model to MBT: Proposed Methodology 19

4.1 UPPAAL to GraphWalker . . . 19

4.1.1 Import and Parse the UPPAAL model file . . . 19

4.1.2 Conversion of the model to an intermediate format (Data Structure) . . . 19

4.1.3 Generate the GraphWalker model . . . 20

4.1.4 Export the model to a GraphWalker-supported JSON file . . . 20

4.1.5 Call the Graphviz Python file to generate the PNG image . . 20
4.2 Proposed Solution 20
4.2.1 Algorithm for translation from Analysis to MBT Model . . . 20
4.2.2 Algorithm for state extraction 21
4.2.3 Algorithm for transition extraction 22
4.2.4 Algorithm for global variables 22
4.3 Detailed model transformation analysis based on hybrid tooling . . 23
4.3.1 MBT to Model Analysis using GW2UPPAAL [40] 23
4.3.2 Model Analysis to MBT using UPPAAL2GW 24
4.3.3 Graphviz representation of the models 28

5 Tool Support **31**
5.1 Introduction 31
5.2 Detailed Architecture of the system 32
5.3 Technology Stack 32
5.4 Installation Procedure and Demonstration 34

6 Experimental Analysis and Results **36**
6.1 Experimental Analysis 36
6.2 Features and Limitations 38

7 Conclusion and Future Work **39**

References **40**

Abstract

This thesis aims to combine MBT with model analysis to provide an overall framework for feedback-based model analysis. We have used an MBT tool, GraphWalker, and a model checker, UPPAAL, for transformation, feedback, and analysis. GW2UPPAAL[1] is an existing tool that transforms the GraphWalker model into UPPAAL timed automata and supports a combined analysis and testing process. The tool enables the automatic verification of reachability and deadlocks freedom properties to exploit the results obtained from this analysis step to improve the test model before generating and executing test cases on the system under test. However, based on model analysis results, the test engineer must manually create the new GraphWalker model, which may be time-consuming and error-prone.

We have developed a hybrid approach (a.k.a. UPPAAL2GW) to transform the UPPAAL-derived model to GraphWalker to provide automated feedback of the model checker to the MBT model, which helps the test engineer to use the modified GraphWalker model for test case generation. We have evaluated the overall approach of the toolchain by seeding mutations into the models created by industrial practitioners and verifying whether the tool provides automated feedback to the test engineer. We have also used Graphviz to reflect changes in the MBT models before and after the modifications. Furthermore, we have integrated both tools for automated analysis and feedback. The integration of GW2UPPAAL and UPPAAL2GW tools bridges the gap between MBT and model checking and ensures the overall analysis and feedback for MBT test case generation.

Demonstration Video: https://youtu.be/5OiNkSvZdFs
Source code & Artifacts: https://github.com/codewithmishra/UPPAAL2GW

[1] https://github.com/iyerkumar/GW2UPPAAL

List of Principal Symbols and Acronyms

MBT Model Based Testing

PNG Portable Network Graphics

SUT System Under Test

List of Tables

6.1 UPPAAL2GW Evaluation Analysis 36

List of Figures

1.1 An approach to Model-Based Testing 2
1.2 An approach to model analysis using a model checker 2
1.3 Generic Method for combined MBT and model analysis 3

2.1 Spotify Login Model created in GraphWalker Studio 7
2.2 Guard, Action and Model Generator in GraphWalker Studio 8
2.3 Pet Clinic Multiple Model System Example 9
2.4 Pedestrian Traffic Light System Modelled in UPPAAL 11
2.5 Object and Variable Declaration in UPPAAL 11
2.6 Model Simulator in UPPAAL 12
2.7 Property Verifier in UPPAAL 12
2.8 GW2UPPAAL Translation 14
2.9 Reachability and Deadlock Freedom Checking In Verifyta 14

4.1 Messenger model modelled in GraphWalker 24
4.2 Messenger model transformed using GW2UPPAAL 24
4.3 Modified Messenger model in UPPAAL 25
4.4 Messenger Model Translated using UPPAAL2GW 26
4.5 Pet Clinic model transformed using GW2UPPAAL 26
4.6 PetClinic Multi-Model System Translated using UPPAAL2GW . . . 27
4.7 Graphviz representation of the model 29

5.1 Execution of UPPAAL2GW 31
5.2 Detailed Hybrid Tool Architecture 33
5.3 UPPAAL2GW Execution Output 34
5.4 Graphviz Image Creation Output 35

CHAPTER 1
Introduction

1.1 MBT and Model Analysis

In this section, we will discuss MBT and model analysis. MBT is a technique that uses models to generate test cases and check test results. The model analysis technique uses models to verify and validate system properties and behaviour. We will also explore how MBT and model analysis can be combined to improve software quality and reliability.

1.1.1 Model-Based Testing

Model-based testing is a software testing technique that uses models of the SUT to generate test cases, execute them, and check the results. A model is an abstract representation of the SUT that captures its essential features and behaviours. Models can be expressed in various forms, such as state machines, flow charts, or formal languages. MBT involves generating test cases by executing a model created by studying the requirements of a SUT. The MBT tool traverses different paths of the model using some coverage criteria to generate test cases [40][12][1][29][10]. Figure 1.1 represents a simple approach to Model-Based Testing. It starts with creating a model for a SUT based on the requirements. The model can be in various forms, such as UML, SysML, state machines or some formal language specification. After that, test cases are generated from the model. Test cases can be derived differently, including test requirements, purpose or use cases. These are the abstract test cases. These abstract test cases will then be executed based on the specific statements or methods in the software. Generated test cases are then executed against the SUT, and the results are compared with the expected results derived from the system's behaviour. The results are then analysed to report defects or to improve the model or the SUT accordingly.

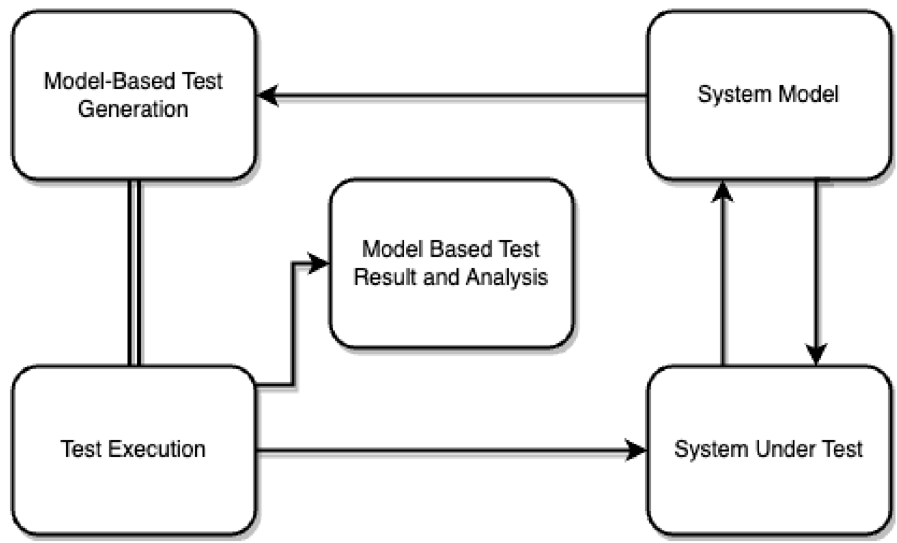

Figure 1.1: An approach to Model-Based Testing

1.1.2 Model Analysis

Model Analysis or model checking is a technique for verifying whether a system's model, which is a finite-state representation of its behaviour, conforms to a given specification or correctness. A model can be expressed in some exact mathematical language, such as propositional logic, temporal logic, or finite state machines. A specification is a set of properties the system should comply with, such as safety properties (avoiding bad states) or liveness properties (eventually reaching good states). Figure 1.2 represents the process of model analysis using a model checker.

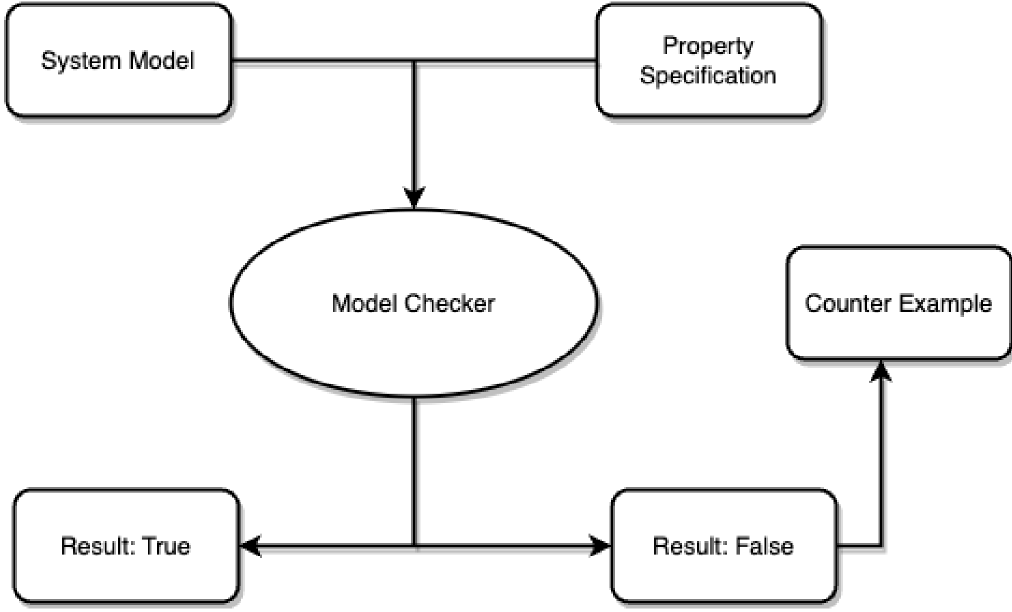

Figure 1.2: An approach to model analysis using a model checker

It starts with building a model of the system under analysis. The model can be obtained from the system's design, implementation, or requirements. The model can be abstracted or simplified to reduce complexity and state space. Together with the model, the specifications of the desired properties of the system are also formed. The specification can be given in a logical formula that can be evaluated on the model's states and transitions. After that, the model and the specifications are given as the inputs to the model checker to check the model against the specifications. This involves verifying whether the model conforms to the specification for all possible behaviours and inputs. After the analysis, the verification results are examined. If the model conforms to the specification, the system is correct for the given abstraction and properties. If the model does not conform to the specification, the verification procedure can provide a counterexample showing a system behaviour that violates the specification. The counterexample can be used to debug or improve the system or the model.

1.1.3 Combining MBT and Model Analysis

MBT and model checking are two techniques for verifying the correctness of software systems. Both methods can help to detect and prevent errors, bugs, and faults in complex and critical systems. However, MBT and model checking also has some limitations and challenges. MBT may not cover all possible behaviours and inputs of the system and may depend on the quality and completeness of the test model. Model checking may face the state explosion problem, the mapping problem, the expressiveness problem, and the scalability problem. Therefore, combining MBT and model checking can be beneficial for improving the verification process and results. Figure 1.3 represents a generic structure for com-

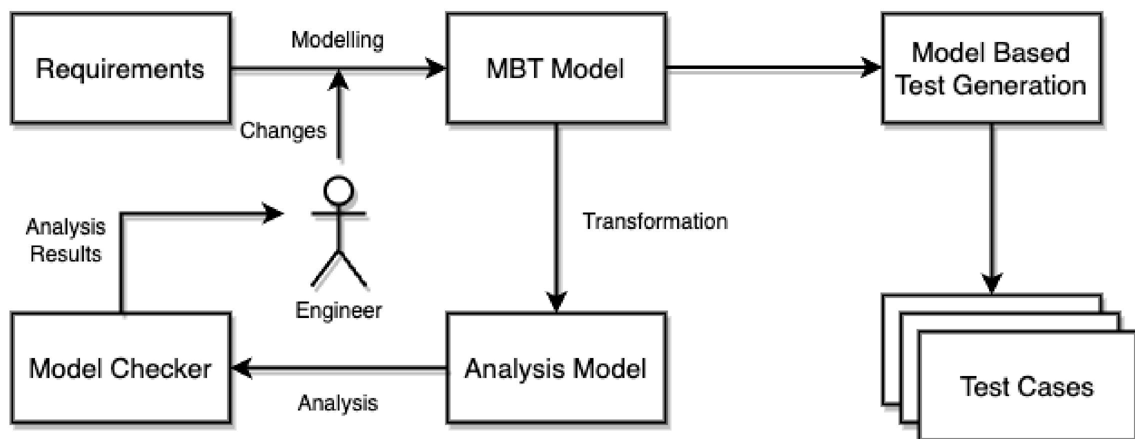

Figure 1.3: Generic Method for combined MBT and model analysis

bined MBT and model analysis. Firstly, requirements must be identified to guide the test generation and analysis. Here a finite-state model can be considered a requirement artefact. After a clear understanding of requirements and specification documents, a model can be obtained which can be used for MBT objectives. For instance, a state model has nodes representing the system's status and edges indicating the actions or choices made when a specific event happens. To verify the model's correctness, model analysis must be performed on the model. For that, an analysis model is required for model checking. A formal analysis state model can be created using global variables, guards, and actions. Given an analysis model of a system, a model checker can analyse the model against some given formal requirements. Based on the results after analysis, a test engineer can change the behavioural model and continue MBT for test generation. A test suite containing a set of test cases is generated by executing the model under a model-based test generation tool.

1.2 Objective and Problem Description

Model-Based Testing (MBT) involves generating test cases by executing a model created by studying the requirements of a System Under Test (SUT). The MBT tool traverses different paths of the model using some coverage criteria to generate test cases [44]. On the other hand, model analysis (i.e., model checking) checks whether the model meets the specified requirements for the given model [5]. The model checker verifies whether the model satisfies a set of properties [11], such as reachability, deadlock-free, safety, and liveness [37].

The aim is to combine MBT with model analysis to provide an overall framework for feedback-based model analysis. Specifically, the work connects the MBT tool with a model checker for automated analysis of a model and automated feedback provision to an MBT tool for test case generation. We have used an open-source MBT tool named GraphWalker to support the tool. In GraphWalker, a set of edges and vertices represent a model. GraphWalker lacks model verification and analysis, so there is a need to combine MBT with a model checker for model analysis before generating test cases. Hence, we used the UPPAAL model checker to analyse the model.

The idea consists of two major components. First, the automated translation of the behavioural model (in GraphWalker) to an analysis model (in UPPAAL). Second, the translation of feedback from the analysis model back to the GraphWalker for the automated generation of test cases. The first part is handled by

GW2UPPAAL [40]. GW2UPPAAL takes a GraphWalker-supported model file as an input and generates a UPPAAL-supported model as an output. It also generates queries to verify the reachability and deadlock freedom that are automatically verified using "verifyta". After the verification through the analysis model, the test engineer manually changes the GraphWalker model. The second component aims to remove the process of manual changes to the GraphWalker model by the test engineer. UPPAAL2GW automatically converts the UPPAAL-supported file to the GraphWalker-supported file, which provides automated feedback for the changes based on the model analysis. Furthermore, we have used Graphviz to show the changes (either insertions or deletions or both) in the behavioural models before and after the changes.

1.3 Thesis Contribution

GW2UPPAAL does not provide feedback to the tester about the verification results or the test model quality, which can lead to missed errors, false positives, or inefficient test cases. So, one of the main contributions of this thesis is to create a feedback mechanism provided by a tool to overcome the limitation of GW2UPPAAL.

We have developed a hybrid approach, UPPAAL2GW, to transform the UPPAAL-derived model to GraphWalker to provide automated feedback of the model checker to the MBT model, which helps the test engineer to use the modified GraphWalker model for test case generation. Leveraging the power of GW2UPPAAL and UPPAAL2GW, we also provide a hybrid toolchain mechanism for efficient MBT and model checking.

Another contribution of this thesis is to create another toolset using Graphviz to represent the changes in two MBT models. The toolset compares two versions of a test model and highlights the differences in states and transitions. The toolset also computes and displays various metrics to measure the changes, such as added, deleted, or unchanged elements using colour codes. The toolset can help the tester to understand and manage the evolution of the test model over time.

1.4 Organisation of the Thesis

The organisation of this thesis is as follows. Chapter 2 introduces GraphWalker, UPPAAL, and GW2UPPAAL, the primary tools used in this work. Chapter 3 re-

views the literature and the related work on MBT and model checking. Chapter 4 presents the methodology and the approach that we have adopted to provide the solution to the problem. It contains the overview and algorithms that are being used. Chapter 5 describes the technical aspects, the technology stack used, and the detailed documentation of the installation and usage of the tool. Chapter 6 reports our experiments to obtain and analyse the results based on the tool and work evaluation. It also discusses the essential features that the work provides as well as the limitations that it faces. Finally, Chapter 7 concludes the thesis and discusses possible future work.

CHAPTER 2

Preliminaries

2.1 GraphWalker

GraphWalker is an open-source toolset for MBT. It allows users to create and edit models using a graphical editor called GraphWalker Studio. A model is a graph that consists of vertices and edges, where vertices represent verifications or assertions, and edges represent actions or transitions. Figure 2.1 represents a model developed in GraphWalker.

Figure 2.1: Spotify Login Model created in GraphWalker Studio

Here, a model can be seen as a collection of states and transitions [41]. The arrows represent a transition from one state to another. The state marked in green is the initial state from which the execution begins. Vertices or nodes represent the verification performed to validate the given requirement. GraphWalker supports both single-model structures as well as multiple models with shared states. GraphWalker stores the model in JSON format.

To generate the test cases, GraphWalker uses generator rules which are some predefined algorithms, to generate a test path [22]. The stop condition also tells GraphWalker when to stop the execution. random(edge_coverage(100)) is an example of the generator and stopping condition combination, which tells GraphWalker to randomly traverse the graph/model until all the edges are traversed at least once. Figure 2.2 represents the generator present in the GraphWalker Studio.

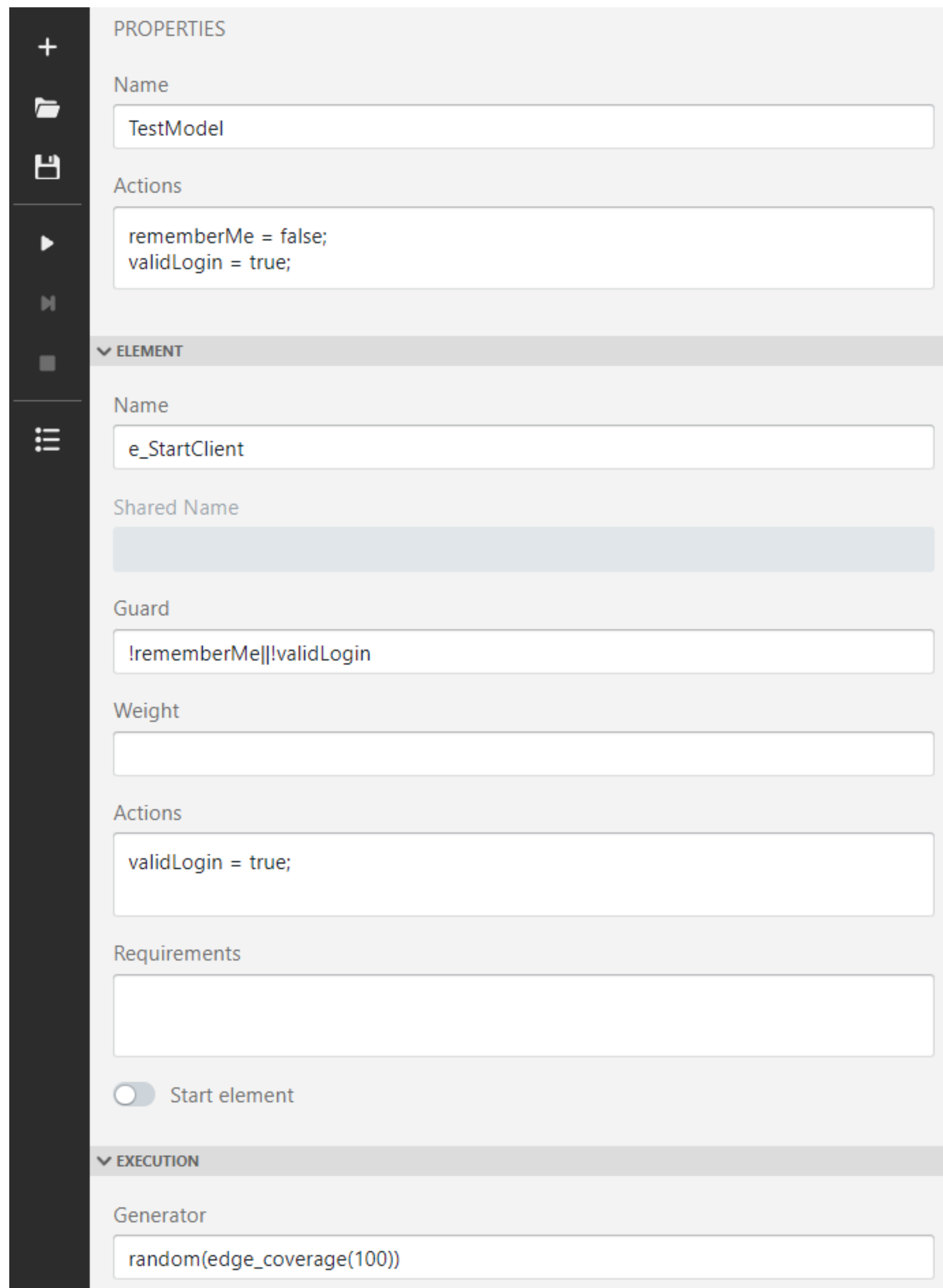

Figure 2.2: Guard, Action and Model Generator in GraphWalker Studio

GraphWalker also supports the use of guards and actions. The guards are those conditions that specify that the transition between the state is only possible when the condition is true. It also supports defining variables called actions which can be used to design, test, and test the model based on the system's expected behaviour. Actions on variables are performed in transitions/edges, and verification is performed on vertices/states [25]. Figure 2.2 shows the panel where global actions, guards and actions for a particular edge are set.

GraphWalker does not interact with the system directly [43], so it uses specific testing frameworks like Selenium to test the model on the system. Testing on models differs from conventional sequential testing, which has a well-defined set of actions to perform. The test sequences generated by GraphWalker are dynamic in nature, so every path generated on the model is a test case, as the model itself is a test idea. Results here are evaluated based on the defined generator and stop conditions. The result is considered a pass if all the vertices pass the checks until the stop condition is reached. Else, the result is considered as fail.

GraphWalker also supports multiple models for a single system concept based on a shared state. Figure 2.3 shows the example of a pet clinic system where a single system is modelled with the help of more than one model. The model represents an online pet clinic system that shows the data of pets and their corresponding owners at a clinic. It also has the data of available Veterinarians for the clinic as well. The vertices shown with the orange colour represent the shared state among the same model. So, for all the vertices having the same shared state name, the execution may jump to any of the shared states randomly. This multiple-model feature is to simplify the understanding of the model.

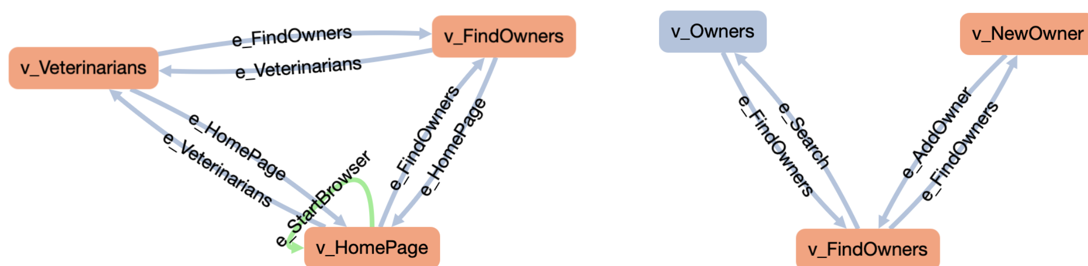

Figure 2.3: Pet Clinic Multiple Model System Example

2.2 UPPAAL Model Checker

UPPAAL is a modelling and verification tool for real-time systems. It is a popular tool for modelling and analysing complex systems, such as communication protocols, control systems [3], and embedded systems [32][45]. UPPAAL is based on a timed automata formalism, which provides a high level of abstraction for modelling real-time systems. UPPAAL has a graphical user interface that allows users to create models using a visual editor. The tool also supports a textual language for defining models, which can be used for more advanced modelling tasks [4]. The models are verified using a model checker, which analyses the system for potential errors and inconsistencies [26].

UPPAAL supports both reachability analysis and model checking. Reachability analysis checks whether a system can reach a particular set of states from the start. It also checks whether a system satisfies a particular or set of properties. UPPAAL uses an efficient algorithm for model checking, which allows it to handle large and complex models. UPPAAL has been used in various applications, including automotive, avionics, and telecommunication systems. It has been used to verify the correctness of safety-critical systems, such as air traffic control systems and train control systems. UPPAAL has also been used to analyse and optimise real-time protocols, such as the IEEE 1394 bus protocol.

Figure 2.4 represents automata created in UPPAAL. It is a Pedestrian Traffic Light example consisting of three states. One state represents the Start state of the system, while the other two represent the Red and Green lights. One variable, 'on', is also defined to keep track of the light which is on currently. Two guards can be seen, which are: on and !on. They are used to control the flow of the system. Two actions can also be seen, which are on = true and on = false, which alters the 'on' variable based on the light which is on currently.

UPPAAL allows us to create multiple objects of the same automata, as seen in Figure 2.5. If we see practically, multiple signals are there on one crossing. So, following that, we create two objects. As seen in Figure 2.5, c1 and c2 are created processes. This is a very important feature supported by UPPAAL as it broadens the scope of model checking with parallel processes and synchronisation. This can be helpful to some real-time problems like mutual exclusion [13], traffic control etc.

Now, to execute the system virtually, UPPAAL provides a simulator which is shown in Figure 2.6. Both manual and automated execution of the model is possible. UPPAAL provides a "Random" option to execute random traces in the

Simulator. It also shows the variable's constraints and values during the simulation process. Furthermore, UPPAAL also generates a sequence diagram of the interaction between multiple processes during the simulation.

One of the important features of the UPPAAL is the verifier [27, 36]. Figure 2.7 represents the verifier screen where the properties are created to check if the system behaves as per the requirement [6]. For example, the query: A[] !(c1.v_Green && c2.v_Green) is used to check that both the lights should not be in the Green state simultaneously. If it is so in real-time, it may lead to an accident. A, E, [] and <> are called quantifiers in UPPAAL. Here, 'A' defines all the paths, and [] defines all the states in the path. && is the operator similar to the other languages and represents the 'and' operator. Similarly, 'E' means there exists a path, and <> means some states in the path. Several combinations of these quantifiers can be used to generate the queries per the requirement and property that must be checked for satisfaction.

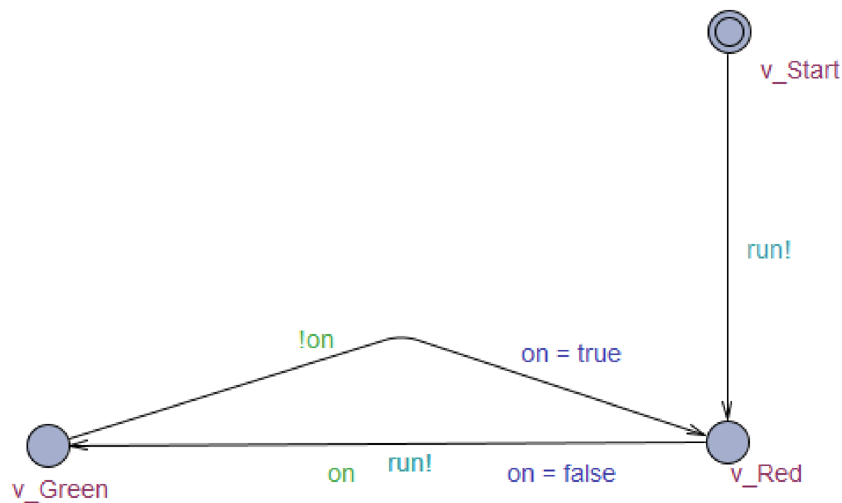

Figure 2.4: Pedestrian Traffic Light System Modelled in UPPAAL

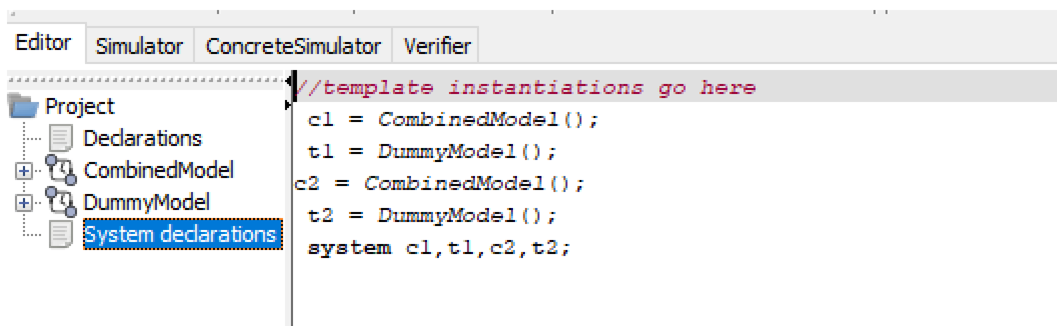

Figure 2.5: Object and Variable Declaration in UPPAAL

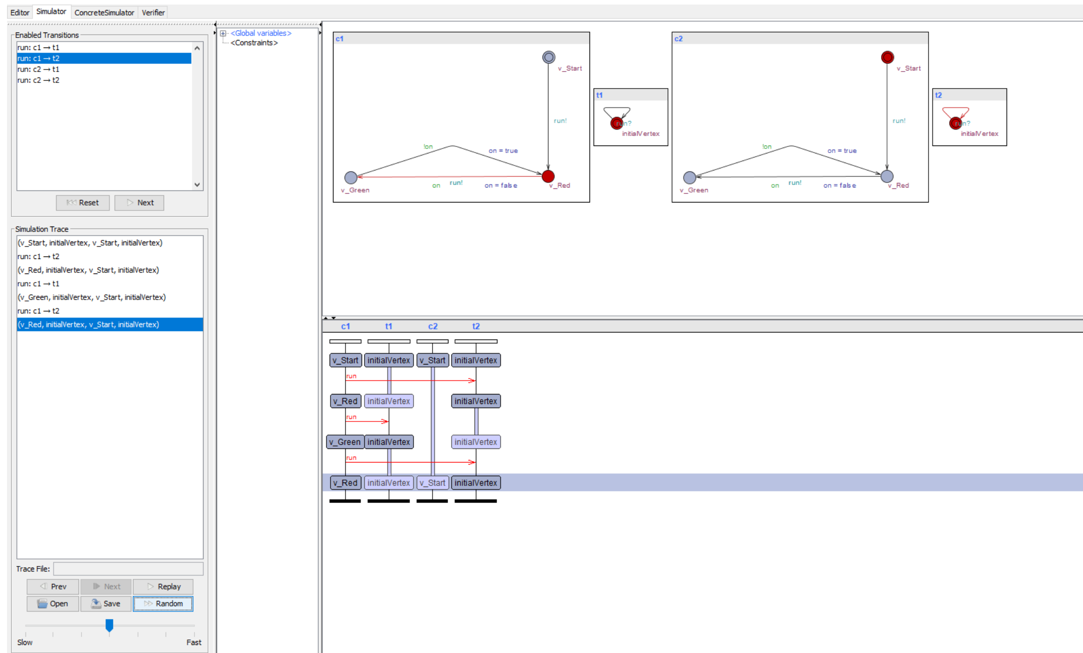

Figure 2.6: Model Simulator in UPPAAL

Editor Simulator ConcreteSimulator Verifier

Overview

```
E<> c1.v_Red
E<> c1.v_Green
E<> c1.v_Start
A[] not deadlock
A[] !(c1.v_Green && c2.v_Green)
E<> !(c1.v_Green && c2.v_Green)
```

Check
Get Trace
Insert Above
Insert Below
Remove
Comments

Query

```
E<> !(c1.v_Green && c2.v_Green)
```

Comment

Status

Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 9,604KB / 46,500KB.
Property is satisfied.

Figure 2.7: Property Verifier in UPPAAL

Both, GraphWalker and UPPAAL use state transition diagrams [35] to represent the model for testing, which makes the desired integration even more fruitful [21].

2.3 GW2UPPAAL

GW2UPPAAL [40] is a tool that transforms models created with GraphWalker into models that can be analyzed with UPPAAL. GW2UPPAAL allows users to combine MBT and automated analysis of behavioural models using GraphWalker and UPPAAL. GW2UPPAAL is implemented in Java and can be executed from the command line. It takes a GraphWalker JSON file as input and generates a UPPAAL XML file as output. The output file can be imported into UPPAAL for further analysis and verification. GW2UPPAAL consists of four main steps: importing the GW model, creating the UPPAAL XML layout, extracting data from GW and generating the UPPAAL model, and exporting the UPPAAL model.

GW2UPPAAL also supports the translation of GraphWalker models with data variables and guards. It maps the data variables to UPPAAL clocks and integers, and translates the guards to UPPAAL expressions. GW2UPPAAL preserves the semantics of the GraphWalker models and ensures that the generated UPPAAL models are equivalent in terms of behaviour and timing. GW2UPPAAL is a useful tool for model-based testing and analysis of software systems. It enables users to leverage GraphWalker's and UPPAAL's benefits, such as graphical modelling, test generation, model checking, and simulation. GW2UPPAAL can be used from the command line directly. Figure 2.8 represents an example of the translation. As shown in the terminal, it computes the number of vertices, edges, and the total time taken for the conversion. Furthermore, "verifyta" is used to check the reachability of each state from the start state as well as the deadlock freedom property of the model. Figure 2.9 represents the "verifyta" screen which displays the satisfaction of the desired properties.

```
C:\Windows\System32\cmd.exe
F:\SE\GW2UPPAAL-main\ToolOutput>java -jar GW2UPPAAL.jar F://SE//GW2UPPAAL-main//ToolOutput SpotifyLo
ginNew.xml F://SE//DirectedWork//inputModels SpotifyLogin.json
Picked up _JAVA_OPTIONS: -Xmx1024M
Start time: 1685002399767
No. of vertices: 3
No. of edges: 9
End time: 1685002399967
UPPAAL Model generated
Total time taken: 200

F:\SE\GW2UPPAAL-main\ToolOutput>
```

Figure 2.8: GW2UPPAAL Translation

```
C:\Windows\system32\cmd.exe
Options for the verification:
  Generating no trace
  Search order is breadth first
  Using conservative space optimisation
  Seed is 1685002400
  State space representation uses minimal constraint systems
←[2K
Verifying formula 1 at /nta/queries/query[1]/formula
←[2K -- Formula is satisfied.
←[2K
Verifying formula 2 at /nta/queries/query[3]/formula
←[2K -- Formula is satisfied.

F:\SE\GW2UPPAAL-main\ToolOutput>
```

Figure 2.9: Reachability and Deadlock Freedom Checking In Verifyta

CHAPTER 3
Related Work

Enoiu et al. [15] proposed an approach to use model checking and logic coverage criteria to automatically generate tests for software systems written in the Function Block Diagram language, a programming standard for safety-critical embedded software. They developed a toolbox called CompleteTest that can transform Function Block Diagram programs into models that can be checked by a model checker and generate test cases that exercise different program artefacts. They conducted an industrial evaluation of their method on 157 programs from Bombardier Transportation AB and measured the time required to generate test cases and the state space size. They reported that their method could efficiently generate test cases that achieve logic coverage for most of the programs and can handle programs of different sizes and complexities.

Another example of applying MBT in an industrial context is the case study by Zafar et al. [44]. They used GraphWalker, an open-source tool for MBT, to test a Train Control Management System developed by Bombardier Transportation AB in Sweden. They followed a three-phase workflow that involved specifying requirements using a domain-specific language based on Gherkin syntax, extracting model elements from the requirements, creating a graphical model of the SUT using GraphWalker Studio, generating test cases using GraphWalker CLI, and executing them on a loop platform. They evaluated the completeness and representativeness of the model, the efficiency and effectiveness of the test cases, and the usability and usefulness of GraphWalker. They found that GraphWalker can support MBT by providing a user-friendly interface for modelling and test generation and enabling randomisation and prioritization of test cases. Their work is relevant to our work because it shows how MBT can be integrated with existing industrial processes and tools [16] and how it can improve the quality and coverage of testing for complex cyber-physical systems.

Eder et al. [14] describe an innovative testing strategy for intricate ATS in many transportation domains. Their strategy incorporates system-level testing in cloud

simulation and on the target system, module-level testing with formal proofs of logical correctness, and agent-based tools for coordinating and optimising test executions. They also demonstrate how their strategy can be justified or complies with current ATS safety standards and certification regulations.

Darwish et al. [38] experimented with comparing two automated MBT approaches for coverage maximization in the automotive industry. They used a real-world SUT and measured the number of test cases, execution time, and fault detection rate. They found that the approach based on combinatorial testing achieved higher coverage and fault detection than the one based on random testing, generating fewer test cases and requiring less execution time.

Ammann et al. [2] proposed using a model checker and mutation analysis to generate test cases from formal specifications. They defined syntactic operators that create variations of a given model and used a model checker to produce counterexamples that distinguish the original model from the variations. The counterexamples serve as complete test cases with inputs and expected results. They applied their method to an example specification and evaluated the test cases on a Java implementation using coverage metrics.

Beyer and Lemberger [8] compare six tools for automatic test case generation and four for model checking on a large suite of C programs. They find that model checkers can find more bugs faster and with fewer program modifications than test-case generation tools. They also introduce a framework for test-based falsification that executes and validates test cases produced by test-case generation tools. They suggest that software model checking is more suitable for finding specification violations than software testing.

Marinescu, Seceleanu, and Pettersson [28] present a framework for component-based evaluation of architectural models described in the EAST-ADL language. They use UPPAAL PORT as a formal semantics for EAST-ADL models and provide tool support for model checking and test case generation. They apply their framework to a Brake-by-Wire system and show its effectiveness in finding errors and generating test cases.

Tiwari, Iyer, and Enoiu [40] propose an approach that combines MBT using GraphWalker with model analysis using UPPAAL. They transform GraphWalker models into UPPAAL timed automata, verifying reachability and deadlock freedom properties. They use the analysis results to improve the test model before generating and executing test cases on the SUT. They evaluate their approach on several model examples. Villani et al. [42] suggest two hybrid approaches that combine model checking with UPPAAL and MBT with ConData [30]. Con-

TEA, a tool linking UPPAAL with ConData and permitting simultaneous use of both methods, supports the integration. The findings demonstrate that combining model checking and MBT aids in the early and thorough detection of design flaws in the system.

Gebizli et al. [20] propose an iterative approach for effective test case generation that combines model-based and risk-based testing. They use Markov Chains as system models, with a probability assigned to transitions between states. These probabilities are adjusted in response to the probability of failure due to memory leaks discovered during test execution. After model improvement, they applied the method to a Smart TV system and found many crash failures. They intend to automate the entire process using an adaptation model based on the history of documented memory leaks.

Gebizli and Sözer [17] propose an approach and a toolset, ARME, for automatically refining test models based on exploratory testing. They contrast the available execution paths in test models with the execution traces captured during exploratory testing. The models are then updated to reflect any missing system behaviour, and model parameters are changed to concentrate on the most frequently run situations. They apply their methodology to three industrial case studies of a digital TV system, and they find several serious flaws that the exploratory testing and first test models had missed. Gebizli et al. [19] also present a novel three-step model refinement approach for MBT. They use Markov Chains as test models and update their state transition probabilities based on the usage profile, fault likelihood and error likelihood. They apply their approach to a Smart TV system and reveal new faults not detected by the initial test models or exploratory testing.

Karna et al. [23] survey the role of model checking in software engineering. They review various model-checking techniques and tools that can be used for software development and analysis. They also examine the applications of model checking at different stages of the software development life cycle, such as debugging, constraint solving, malware detection and verification of different software systems. They highlight the challenges and opportunities of model checking for software engineering research and practice.

Black et al. [9] explore the properties of specification mutation operators for model checking. They mutate a specification and use a model checker to compare it with the provided specification to generate tests or evaluate coverage. They also compare different operators theoretically and empirically.

Njor et al. [31] propose a novel approach for conformance testing of timed systems using UPPAAL, a model checker for real-time systems. They introduce a tool

called Diabolic, which automatically generates test cases from UPPAAL models and executes them on the SUT. Diabolic uses a mutation-based technique to create test objectives that cover different aspects of the system's behaviour, such as timing constraints, data values, and communication events. Diabolic also supports online testing, where test cases are generated and executed online, and offline testing, where test cases are generated beforehand and stored for later execution. The authors evaluate Diabolic in several case studies and show that it can effectively detect faults and measure the conformance of timed systems.

Berdasco et al. [7] resemble a prior study examining software engineers' MBT use in an industry scenario. MBT is a method for automating the design and production of test cases based on a model of the SUT. The authors assess the viability and acceptance of the MBT technique from the standpoint of quality engineers testing an industry software application. They use a tool called FORMAT, which supports adapting test models based on feature models. They compare MBT with manual testing regarding effectiveness, efficiency, and satisfaction. The results show that MBT was more effective and efficient than manual testing. Still, the satisfaction of the quality engineers was lower due to some usability issues of the tool and the lack of training and guidance.

Summary

A lot of work is being done in the area of MBT and model checking. A larger number of those are being done individually in both domains but have a pinch of others for some specific purposes. After GW2UPPAAL, this work can be used as a feedback mechanism from UPPAAL to GraphWalker.

CHAPTER 4

Analysis Model to MBT: Proposed Methodology

This chapter discusses the feedback mechanism transforming the UPPAAL model to GraphWalker. The discussion includes the approach and algorithms used to solve the problem. An example is also provided to support the discussion.

4.1 UPPAAL to GraphWalker

UPPAAL uses XML (Extensible Markup Language) as the file format to store a model. On the other hand, GraphWalker uses JSON (JavaScript Object Notation) to store the model. XML and JSON are used to transfer data between various programs through APIs. Our tool transforms the UPPAAL-compatible XML to GraphWalker-supported JSON to provide model-checking feedback. JAVA is used as a programming language for the transformation. The approach is divided into several steps mentioned ahead.

4.1.1 Import and Parse the UPPAAL model file

In the first step of UPPAAL2GW, the UPPAAL model file is read and parsed, and a JAVA object is created referring to that XML file to fetch information such as edges, vertices, global variables, guards, and actions.

4.1.2 Conversion of the model to an intermediate format (Data Structure)

The JAVA object obtained in Step 1 is then converted to an intermediate format, which can be used to generate a GraphWalker model. This involves extracting the relevant information from the UPPAAL model, such as the states, transitions, and global variables, and storing it in a data structure. For example, states can

be extracted using "//location" tags, and vertices using "//transition" tags can be stored in an Array.

4.1.3 Generate the GraphWalker model

Using the intermediate format generated in the previous step, a skeleton GraphWalker model is generated using JAVA. This involves creating the necessary vertices and edges in the GraphWalker model, setting the appropriate properties such as id, name, initial vertex, guards and actions, and connecting the vertices and edges to form a complete model by mapping the tags from UPPAAL to GraphWalker. For example, "//location" will be mapped to "vertices", "//transition" will be mapped to "edges", and "//declaration" will be mapped to "actions". For the edges, guards and actions will be mapped using "//guard" and "//assignment" tags, respectively. In the end, the 'model' root element is added, followed by all the extracted data corresponding to the model. This is done by appending all the data structures to the GraphWalker model object.

4.1.4 Export the model to a GraphWalker-supported JSON file

Finally, the modified GraphWalker model generated after model analysis is transformed into a file that the test engineer can use in GraphWalker to derive test cases.

4.1.5 Call the Graphviz Python file to generate the PNG image

After the model is exported, the Graphviz Python file is executed using JAVA's runtime execution with initial and new JSON file names as the arguments (similar to executing the code from a terminal with arguments) to generate a PNG image representing the changes in both models.

4.2 Proposed Solution

4.2.1 Algorithm for translation from Analysis to MBT Model

Algorithm 1 is used to develop UPPAAL2GW discussed previously for the retransformation of the model.

The algorithm works by first importing the UPPAAL XML model into the program using a command line argument. The algorithm then fetches the start state

Algorithm 1 Overview of conversion from UPPAAL to GraphWalker

1: Import the UPPAAL XML into the tool.
2: Fetch the start state using the "init" tag.
3: Initialize Arrays to store edges, vertices and variables.
4: Using NodeList object fetch the states, edges and variables using "//location", "//transition" and "//declaration" tags.
5: **for** `each vertex in fetched states` **do**
6: Create a new object using JSonObject.
7: Add the properties like id ("id"), name ("name"), and coordinates ("x" and "y") to the object using respective tags.
8: **end for**
9: **for** `each edge in fetched transitions` **do**
10: Create a new object using JSonObject.
11: Add the properties like id ("id"), source vertex ("source"), target vertex ("target"), guards ("guard") and actions ("assignment") to the object using respective tags.
12: **end for**
13: **for** `each variable in fetched variables` **do**
14: Split the text using "=" flag.
15: Append the variable names and values to the variables array.
16: **end for**
17: Initialize a JsonObject model supported by GraphWalker.
18: Add the model properties like id, name, etc.
19: Add the vertices, edges, and variables to the model.
20: Export the final model file to GraphWalker-supported JSON model.
21: Call the Python file to generate a Graphviz image representing the changes.

from the "init" tag in the UPPAAL XML model. Next, the algorithm initializes the vertices, edges, actions, and guards using JAVA's library JSONArray. For each vertex and transition in the UPPAAL XML model, the algorithm creates a new object using another JAVA library JsonObject. The algorithm then adds the properties like id, name, and coordinates to the object. Finally, it initializes the start state, describes a model generator, initializes a JsonObject model supported by GraphWalker, adds the model properties like id, name, etc. to the model, adds the vertices, edges, actions, and guards to the model, exports the final model file to a GraphWalker supported JSON model, and calls the python file to generate a Graphviz image representing the changes.

4.2.2 Algorithm for state extraction

Algorithm 2 extracts each state's data from UPPAAL XML and creates a vertex in GraphWalker JSON. It begins by using the NodeList object to get all the values

Algorithm 2 Algorithm for state extraction

1: Import the UPPAAL XML into the tool.
2: Fetch the state set from the "//location" tag.
3: Initialize vertices using JsonArray.
4: **for** `each state in vertices` **do**
5:     Create a new vertex object using JsonObject.
6:     Add the properties like id, name, and x and y coordinates to the vertex object.
7:     Initialize the start state.
8: **end for**

from the "location" tag. Then for each element in the stateList, fetch the ID of each state. Further, fetch the name of the state for that particular state. For each state, we create a JsonObject object and store the corresponding id and name of the object. Furthermore, add the coordinates values for each vertex to the object. Also, if the vertex is the start state, assign the value to the corresponding variable. In the end, append the JsonObject object to the previously defined JsonArray vertices object.

4.2.3 Algorithm for transition extraction

Algorithm 3 extracts data of each transition from UPPAAL XML and creates a transition in GraphWalker JSON. The algorithm begins by fetching transition details from the "transition" tag. Then, for each transition, the corresponding ID from the "id" tag, source and target vertex from the "ref" and "label" tag are fetched and stored in the NodeList object. For each transition, guards and actions are also fetched using the "guard" and "assignment" tags. A JsonObject is initialized, and all the properties fetched are added to a particular object for a particular edge. In the end, each object is appended to the JsonArray object.

4.2.4 Algorithm for global variables

All the global variables are defined on the top of the UPPAAL XML file. Algorithm 4 fetches all the variables based on the "declaration" tag, representing global declarations. The variable name and values are fetched for every declaration based on the "=" split. Each variable is appended to the global action variable for the GraphWalker file.

Algorithm 3 Algorithm for transition extraction

1: Import the UPPAAL XML into the tool.
2: Fetch the transition list from the "//transition" tag.
3: Initialize edge object using JsonArray.
4: **for** `each transition in transitions` **do**
5:     Create a new transition object using JsonObject.
6:     Fetch the source and target state using the "ref" tag.
7:     Initialize guard and action object.
8:     **for** `each label in transition` **do**
9:         Fetch guards using the "guard" tag.
10:         Fetch actions using the "Assignment" tag.
11:     **end for**
12:     Append vertex id, source element id, target element id, guards and actions to edge object.
13: **end for**

Algorithm 4 Algorithm for global variable extraction

1: Import the UPPAAL XML into the tool.
2: Fetch the declarations from the "//declaration" tag.
3: Initialize global variable object using Element and String array using newline character.
4: **for** `each variable in variables` **do**
5:     Split the variable from "=" character.
6:     Assign the first part to a variable name object and the second to a variable value object.
7:     Assign the value to the global actions variable.
8: **end for**

4.3 Detailed model transformation analysis based on hybrid tooling

4.3.1 MBT to Model Analysis using GW2UPPAAL [40]

Figure 4.1 shows a model which was modelled using GraphWalker. It represents a basic Messenger System consisting of states and edges representing a system where users can select and chat with some users. The model consists of six states and eleven transitions, as shown in Figure 4.1. GW2UPPAAL is used to convert the model to the corresponding analysis model to perform the automated analysis of the model. The analysis model is shown in Figure 4.2.

GW2UPPAAL also includes guards and actions corresponding to the edges. The properties are also generated in every model based on the number of states present. It checks whether every state is reachable from the initial state. Verifyta

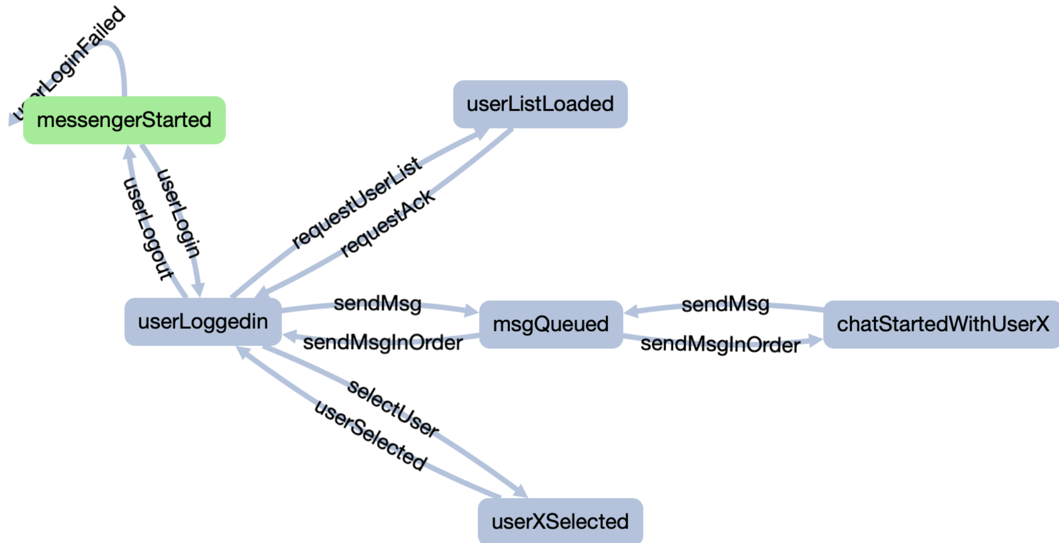

Figure 4.1: Messenger model modelled in GraphWalker

is used to verify the reachability and deadlock freedom property.

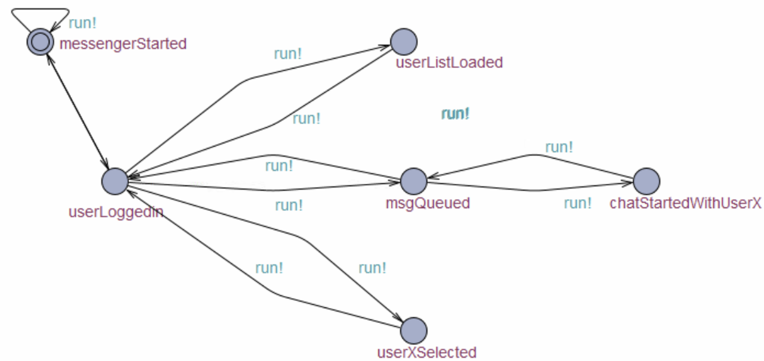

Figure 4.2: Messenger model transformed using GW2UPPAAL

4.3.2 Model Analysis to MBT using UPPAAL2GW

After the translation from MBT to the Analysis model using GW2UPPAAL, the model is analysed based on specific requirements. This way, we can use the analysis results to update the test model and generate test cases that reflect the verified properties. We also present a tool called UPPAAL2GW that automates the bidirectional transformation process and supports a combined analysis and testing workflow. When an error or bug is identified in an analysis model, the challenge is to modify the actual behavioural model to perform MBT and generate the test cases. One way is to modify the model manually and do the necessary changes

which may lead to some errors or mistakes. Another way is to automate the conversion of the analysis model to the behavioural model and do the testing effectively. When bugs or errors are identified in the analysis model, the engineer or the developer needs to fix those bugs or errors in the analysis model and verify it. For the part of re-transformation and feedback, which was given to the engineer to do the changes manually in the behavioural model, the feedback will be directly given to the actual model by creating a new model with the changes. The changes or fixes mentioned here can be some new edge or vertex, some guard or action condition or some variables. The complete idea is to implement a reverse conversion process, where the modified UPPAAL model is converted back to a GraphWalker model with the necessary changes to address the bugs or errors identified during model checking. To achieve this, we have developed a code library that takes as input a UPPAAL XML file and outputs a GraphWalker JSON file with the modifications required to address the identified errors. The resulting GraphWalker model can then be used to generate executable test cases using GraphWalker, thereby providing a complete and automated testing solution.

Using the algorithms discussed in Section 4.3, UPPAAL2GW is developed using JAVA and is evaluated on several models from different domains including a few industrial models.

To support the translation verification, we have used mutations and changes in UPPAAL to evaluate the model. The technique will be discussed in the later chapters. Figure 4.3 is the changed model we got after making some changes from Figure 4.2 based on some logical requirements. We can see that some edges are deleted while some are added. Also, some guards and actions are introduced, along with a variable to support them. After the modifications, the analysis model

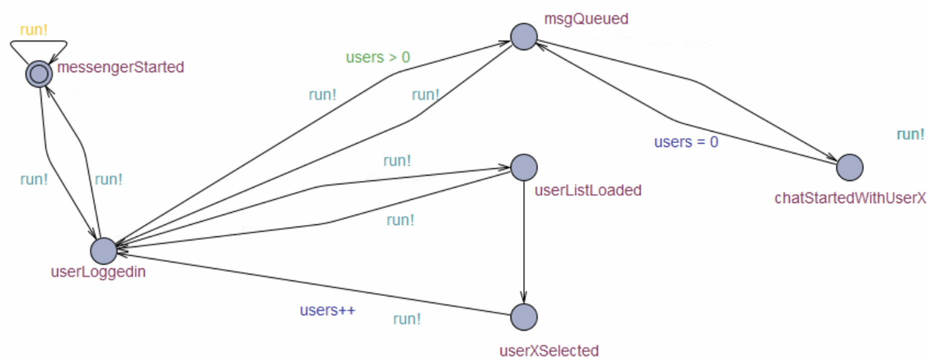

Figure 4.3: Modified Messenger model in UPPAAL

must be converted back to the MBT model to perform Model Based Testing. Figure 4.4 displays the corresponding MBT model representing the analysis model

in Figure 4.3.

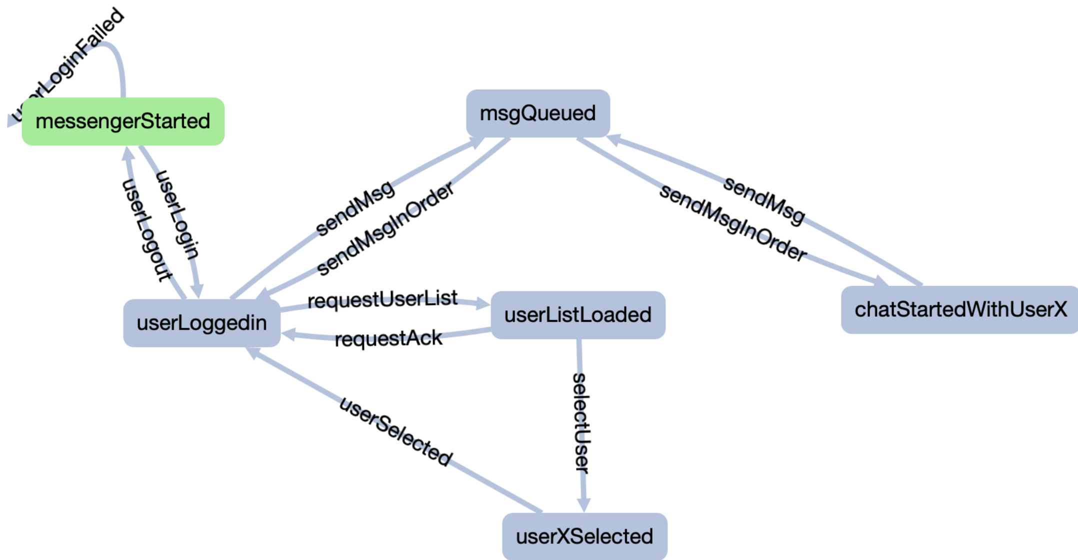

Figure 4.4: Messenger Model Translated using UPPAAL2GW

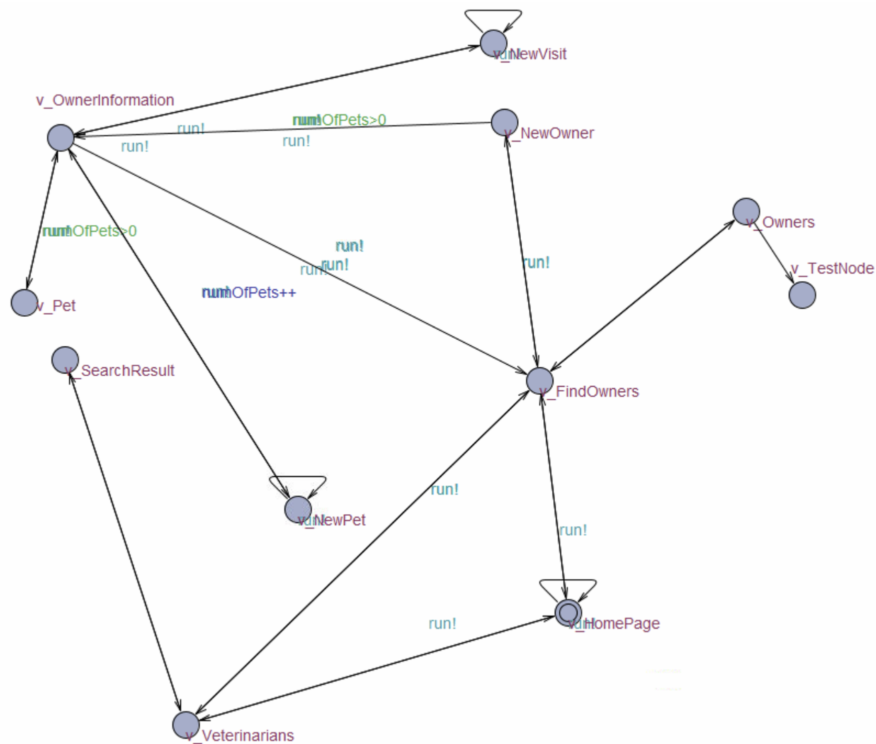

Figure 4.5: Pet Clinic model transformed using GW2UPPAAL

GraphWalker supports multiple model systems, and Figure 2.3 shows the multiple models of the PetClinic system represented in GraphWalker. The GW2UPPAAL tool combines edges and vertices based on shared state names. A corresponding

single analysis model will be generated for a single model system, as shown in Figure 4.2. For multiple model systems like PetClinic (Figure 2.3), a corresponding single analysis model file will be generated using GW2UPPAAL, as shown in Figure 4.5. Considering the flattened model generated in Figure 4.5, the corresponding MBT model will have a single model after the translation using UPPAAL2GW. Figure 4.6 represents the same PetClinic model. When analyzed in GraphWalker using the same edge coverage criteria, there were no differences in the results for the original multiple models or the translated single model. This shows that both models represent the same behaviour for the PetClinic system.

Figure 4.6: PetClinic Multi-Model System Translated using UPPAAL2GW

4.3.3 Graphviz representation of the models

Graphviz[1] is a free and open-source graph visualisation tool. It includes a set of tools for developing and manipulating graph structures and generating visual representations of graphs. The Graphviz tools use the DOT language, which is a plain-text graph description language, to express the structure and attributes of graphs. Here are some key features and use cases of Graphviz:

1. Graph Visualization: Graphviz allows you to create visual representations of graphs, such as directed and undirected graphs, flowcharts, organizational charts, and network diagrams. It provides options for customizing the appearance of nodes, edges, labels, colors, and styles.

2. Automatic Layout: Graphviz includes various layout algorithms that automatically arrange the nodes and edges of a graph based on their relationships and attributes. These algorithms ensure that the generated graph layouts are clear and visually appealing.

3. Programmability: Graphviz provides APIs and libraries for several programming languages, including Python, to programmatically create, modify, and manipulate graphs. This enables you to integrate graph visualization capabilities into your own applications and workflows.

4. Integration with Other Tools: Graphviz can be easily integrated with other software tools and frameworks. For example, it can be used in software engineering to visualize code dependencies, in data analysis to represent complex relationships, in machine learning to visualize decision trees or neural networks, and in documentation to create visual diagrams.

5. Extensibility: Graphviz offers extensibility through plugins and extensions. You can extend its functionality by creating custom graph attributes, layout algorithms, or output formats tailored to your specific requirements.

Representation of changes using Graphviz Library

Based on the mentioned properties discussed in 4.4.1, another library set was developed in Python which is called at the end of UPPAAL2GW to create a pictorial representation of the changes in the models. This is done to increase the understanding and readability of the changes done in the model file. Graphviz takes two GraphWalker JSON model files as input and generates one PNG image file

[1] https://graphviz.org/

representing the changes in the model. Figure 4.7 represents the modifications in models represented in Figure 4.1 and Figure 4.4.

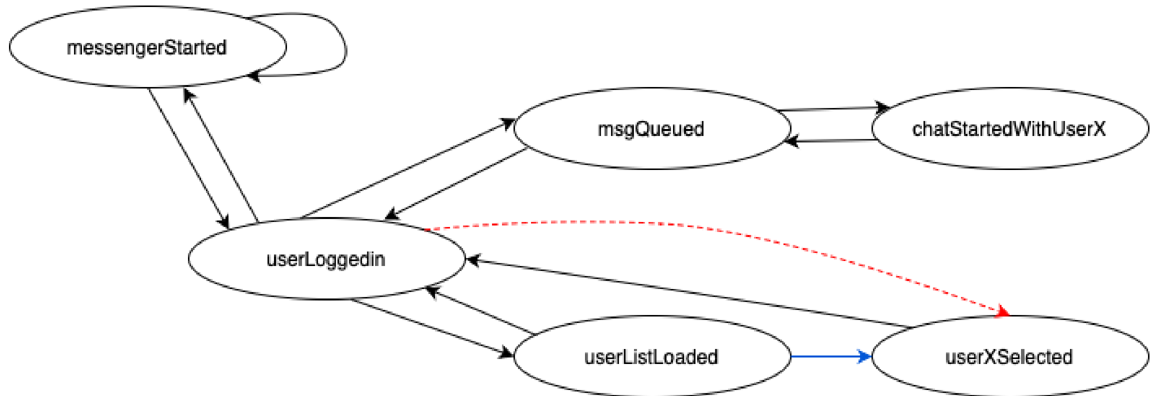

Figure 4.7: Graphviz representation of the model

Different colours in the image represent different changes in the model. As shown in Figure 4.7, red-coloured dashed elements represent the deletion of a particular element. i.e. they are there in the old model but not in the new model file. At the same time, blue-coloured elements represent the added elements. i.e. they were not in the old model but are added in the new model file based on modifications.

Algorithm for image generation

Algorithm 5 is used to generate a different image file using Graphviz. The mapping of all the vertices and edges is done based on both IDs and names to perform an effective mapping. Firstly, both the JSON files are read and assigned to two different variables. After that, a new variable is initialized with the differences in both the JSON files using Python's diff library. In the later part of the algorithm, new vertices and edges are added using a new JSON file, whereas, old vertices and edges are added using an old JSON file. If the element is present in both old and new JSON files, it is considered only once to prevent duplication. Deleted elements are represented using red coloured dashed elements, whereas new elements are represented using blue colour for better understanding.

Algorithm 5 Algorithm to generate Graphviz representation

```
1: Load both the JSON files in two variables.
2: Generate the differences in both the models using Python's diff library.
3: Generate old_vertices, new_vertices, old_edges and new_edges based on
   mapping of name and id of all the elements.
4: Generate deleted_vertices and deleted_edges using set difference.
5: for each vertex in new_vertices do
6:     Create a graph vertex with the name and blue-coloured element.
7: end for
8: for each vertex in deleted_vertices do
9:     Create a graph vertex with the name and red-coloured dashed element.
10: end for
11: for each vertex in old_vertices do
12:     if vertex not in deleted_vertices then
13:         Add it to the graph.
14:     end if
15: end for
16: for each edge in old_edges do
17:     if edge in deleted_edges then
18:         Create a graph edge with the name, source vertex id, and target vertex
   id with the red-coloured dashed element.
19:     else
20:         Create a graph edge with the name, source vertex id, and target vertex
   id.
21:     end if
22: end for
23: for each edge in new_edges do
24:     Create a graph edge with the name, source vertex id, and target vertex id
   with the blue-coloured element
25: end for
26: Create a PNG file.
27: Render the PNG file by adding the image to the PNG.
```

Chapter 5
Tool Support

5.1 Introduction

Combining the algorithms for UPPAAL2GW and code developed for Graphviz representations in a single JAR file, translation and representation can be handled using a single file. JAR implementation is necessary because it helps reduce the efforts of installing additional libraries to run the application. Also, the platform-independent feature of JAVA helps implement the code on any machine or OS that supports JAVA. Figure 5.1 shows the execution of UPPAAL2GW using a command line terminal.

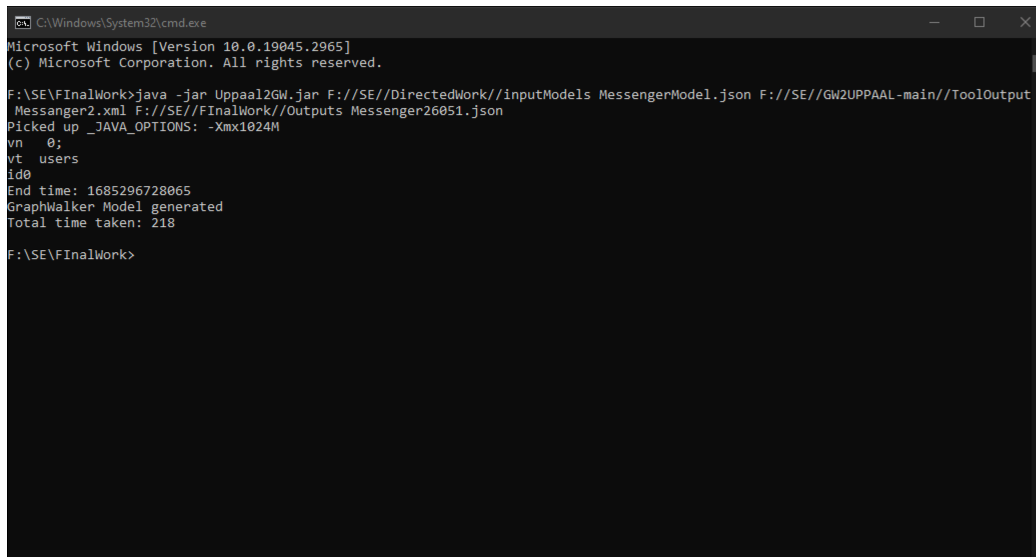

Figure 5.1: Execution of UPPAAL2GW

5.2 Detailed Architecture of the system

Figure 5.2 represents a detailed architecture of the system which includes both, GW2UPPAAL and UPPAAL2GW.

1. A test engineer or a designer is required to develop a behavioural model for a SUT. The model can be a GraphWalker-supported JSON model which can include some vertices, edges, variables, guards, and actions to support the requirements of the system.

2. In the next step, the model needs to get verified against the specifications. Hence, we require an analysis model corresponding to the GraphWalker model. The automated transformation from a behavioural to an analysis model is taken care of by GW2UPPAAL. The output of the tool is a UPPAAL-supported XML model file.

3. GW2UPPAAL uses JAVA libraries to translate from GraphWalker to UPPAAL. The translation process includes a combined model, guards, actions, edges, vertices and variables.

4. After performing the analysis, the changes, if required, based on the results, are done in the analysis model in UPPAAL itself and are saved. The corresponding behavioural model with the changes is required to generate the test cases using GraphWalker.

5. In this step, we automatically re-transform the UPPAAL model into the GraphWalker model, with the feedback, using the UPPAAL2GW tool. The tool also uses JAVA libraries for translation. The translation includes all the changes in either of the actions, guards, vertices, edges or variables. Finally, it generates a corresponding GraphWalker-supported JSON file.

6. The re-transformed model can be imported back into GraphWalker to generate test cases more effectively. With the help of Graphviz, a PNG image showing the changes is also generated, which reflects the changes done from the previous model to the new model.

5.3 Technology Stack

The following technology stack has been used for the development purpose:

1. **JAVA Version:** 11

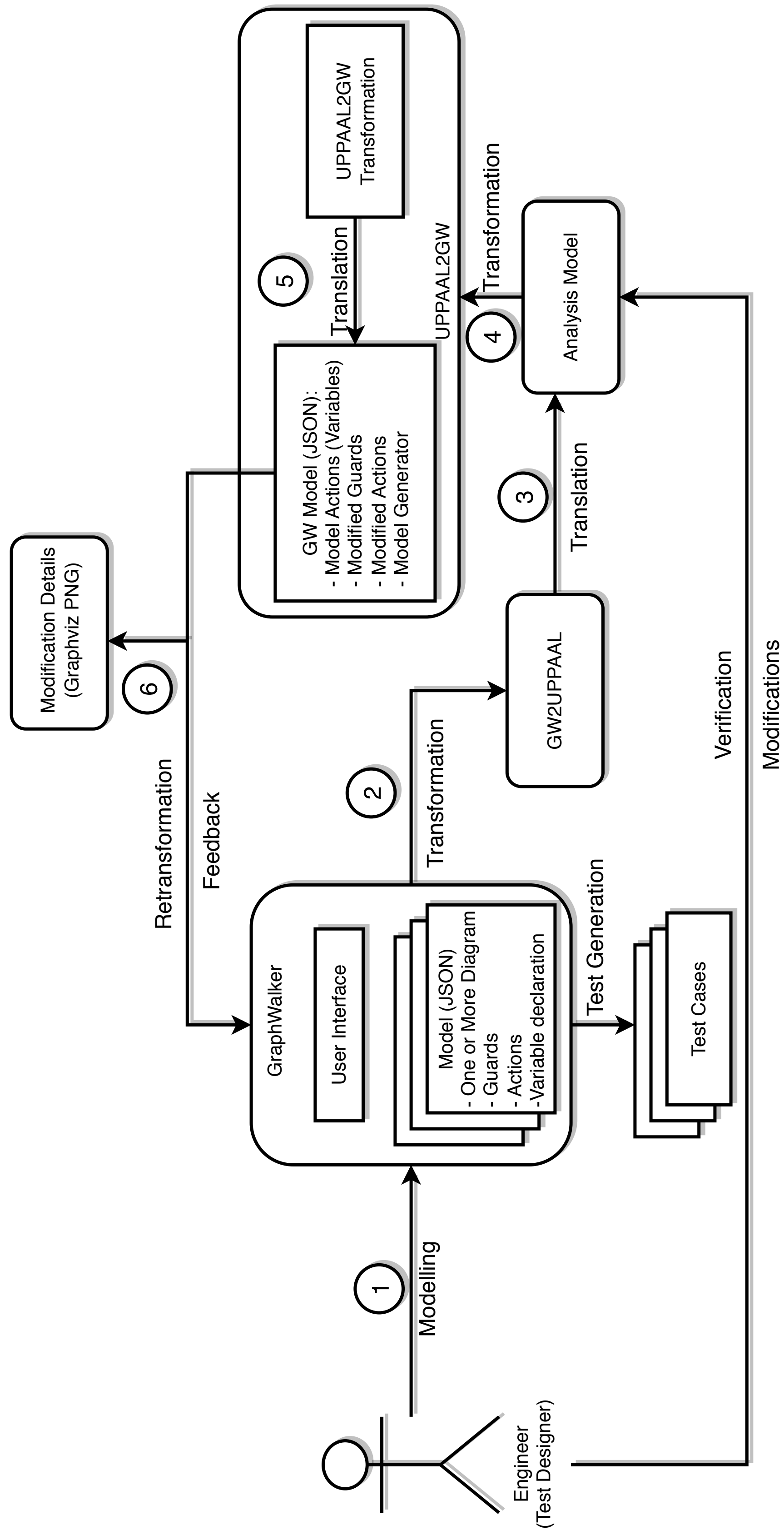

Figure 5.2: Detailed Hybrid Tool Architecture

2. **Python Version:** 3.8
3. **Operating System:** Windows 10
4. **GraphWalker Studio Version:** 4.3.2
5. **UPPAAL Version:** 4.1.26
6. **Graphviz Version:** 8.0.5
7. **IntelliJ IDEA Version:** 2022.2.3 Community Edition

5.4 Installation Procedure and Demonstration

This section provides a brief about the installation and usage of UPPAAL2GW. The installation has some prerequisites that are needed before the execution of the tool. For the tool and GraphWalker both, Java Runtime Environment (JRE) 1.8 or higher is required. Next, UPPAAL[1] needs to be installed. Here, we have used version 4.1.26 for the demonstration purpose. GraphWalker[2] needs to be installed as well. GraphWalker does not require any specific installation file; it comes with a precompiled JAR file which can be executed from the command line to access the GraphWalker Studio. Graphviz[3] needs to be installed along with Python 3.8 or higher, and both need to be added to the PATH. After installing everything, to execute UPPAAL2GW, the user needs the following command java -jar UPPAAL2GW.jar [Initial GraphWalker Model File Path] [Initial JSON Model File Name] [UPPAAL Model File Path] [UPPAAL XML Model File Name] [Output Folder Path] [Output JSON File Name]. The UPPAAL model file path and name can be directly used from GW2UPPAAL's ToolOutput folder or another folder.

```
End time: 1684145606360
GraphWalker Model generated
Total time taken: 269
```

Figure 5.3: UPPAAL2GW Execution Output

Figure 5.3 and Figure 5.4 shows the output of the execution. Note that the Graphviz terminal is directly triggered from UPPAAL2GW JAR, and no extra efforts are required. In the end, the user can open the modified JSON file in Graph-

[1]https://uppaal.org/downloads/
[2]https://graphwalker.github.io/
[3]https://graphviz.org/download/

```
C:\WINDOWS\system32\cmd.exe
Image Generated

C:\Users\admin\Desktop\SE\FInalWork>
```

Figure 5.4: Graphviz Image Creation Output

Walker and continue the MBT process. For a better understanding, the user can see the PNG file created to look for the modifications done in the model.

CHAPTER 6
Experimental Analysis and Results

6.1 Experimental Analysis

We evaluate UPPAAL2GW with several models from different domains. Both single and multiple-model systems were used for verification. We have used the mutation technique to verify the transformation process. By applying mutation, we evaluate the capability of the tool to detect changes in the model and assess its effectiveness in identifying potential changes. This helps in ensuring the reliability and accuracy of the tool. Also, we have used some logical requirements corresponding to the system to mutate the model in UPPAAL for evaluation. These requirements are based on the type and domain of the model.

Table 6.1: UPPAAL2GW Evaluation Analysis

Model	No. of Vertices	No. of Edges	No. of Mutants
ShoppingCart	6	11	6
CanDepositMachine	3	8	-
OnlineChatApp	3	10	1
DoorSystem	4	6	2
SpotifyLogin	3	9	1
MessengerModel	6	11	4
CoffeeMachine	20	33	11
PetClinic	16	25	5
Login System	6	20	2
Industrial Model	22	27	-

Table 6.1 shows the details of the evaluation of several available models. ShoppingCart is a model provided by GraphWalker which represents Amazon shopping cart [34]. Here, a user searches for a book and adds it to the cart three times. The mutations for this model are done for searching a book only once, checking if the quantity is available and then adding it to the cart if possible. CanDepositeMachine models the functionality of receiving money by depositing a can into the

system. The model already takes care of the negative values, no money, etc., features, so no logical mutations were possible for this model. OnlineChatApp, as the name suggests, represents a chat application system including login, logout and chat functionality. We have changed the sequence of execution by replacing the edges from one place to another. DoorSystem represents a system where a door can be open, closed, or faulty. We improved the faulty condition with the logical requirement that a door can be faulty when open or closed and mutated the model accordingly. SpotifyLogin is again a model provided by GraphWalker which replicates the login system provided by the Spotify desktop client. Again, we changed the sequence of events by deleting an edge from the model. MessengerModel is the model that we have discussed till now. It represents a system where the user can select a few users from a list of users and start the chat with them. Considering another logical requirement that a user should not be able to start the chat without selecting zero users from the list, we have modified the model, kept the selection of users only after the list is loaded and introduced a variable that keeps track of the number of users selected. We have introduced some guards and actions as well to support the variable. CoffeeMachine is again a complex model which we have tested. It represents a coffee-making process with a failure-handling mechanism. To mutate this model, we have changed the sequence of events by modifying the edges and vertices. PetClinic is a multi-model system, again provided by GraphWalker, representing the pet clinic system. We have mutated this model based on the occurrence of the events and flow of the system. LoginSystem, again as the name suggests, represents the handling of user login of a website. It also supports the 'remember me' feature and handles the system's failures. The industrial model here is a model that represents a model modelled based on an SRS provided by industry experts. As it was created based on the requirements, we directly tested the translation of the model without any mutations.

The mutations in the models helped in evaluating our tool effectively. With the help of mutations based on logical requirements [24], we could compare the models before and after the modifications. In this way, we could identify whether UPPAAL2GW works correctly or not. With the algorithms and programming, when we tested UPPAAL2GW on the available discussed models, we identified that the translation after modification, if any, was correct, and we could generate the correct model again for the MBT.

6.2 Features and Limitations

In this section, we report some of the benefits of using automated translation with the help of UPPAAL2GW. To begin with, UPPAAL2GW eliminates manual intervention in the feedback process from analysis to the MBT model. Earlier, the feedback from the analysis model to an MBT was provided to the test engineer, and the engineer was responsible for taking the feedback and doing the modifications manually. If handled improperly, human intervention and the manual translation may lead to errors and bugs [18]. In that case, a lot of rework can also happen for a minor modification. Automated translation will help remove all human errors that may be introduced using manual feedback translation.

Furthermore, another important constraint, time, will be much more considering large-scale or complex models [39]. The main task of a test engineer should be testing, not the extra work that can be taken care of by automation [33]. Using automated translation, a lot of time can be saved, and translations done in minutes can be done in a few milliseconds. Also, if the user is unsure about the model generator, UPPAAL2GW considers random edge coverage as a default model generator that can be used directly. Readability and understanding are improved using Graphviz representation of the changes as well. Currently, due to the flattening process by GW2UPPAAL, UPPAAL2GW, when retranslating a multi-model system, it creates only a single MBT model which is the limitation of UPPAAL2GW.

Chapter 7
Conclusion and Future Work

In this thesis, we present a hybrid approach used for combining MBT and model-based analysis. This approach is instantiated using GraphWalker and UPPAAL. A well-known tool, GW2UPPAAL, is being used as a base to transform the behavioural model into an analysis model supported by a model checker. We have developed UPPAAL2GW to bridge the gap between MBT and model checking effectively.

We have used logical requirements to support and use mutation to evaluate the approach. We have used several available models in GraphWalker documentation and models developed by industrial practitioners. GW2UPPAAL automatically creates queries to verify the reachability and deadlock properties. However, specific requirements-based properties are required to be checked manually. Once the verification is done, UPPAAL2GW translates UPPAAL XML back to GraphWalker JSON along with the changes. To increase the readability, a Graphviz image file is also generated for the designer to keep track of the changes done in time. The entire tooling and GW2UPPAAL and UPPAAL2GW effectively bridge the gap between MBT and model checking and can be used together to create a complete hybrid toolchain mechanism for combined MBT and model analysis.

Currently, as UPPAAL does not support multiple model representation of a system and GW2UPPAAL flattens the model based on shared vertex name, generating a single model file for multiple model systems, UPPAAL2GW, when executed, generates a single model file only for MBT. This can be improved in the future by splitting the file again into multiple models to increase the readability to some extent. Also, the user can be involved in providing the model generator criteria to enhance the translation based on the requirements.

References

[1] P. Akpinar, M. S. Aktas, A. B. Keles, Y. Balaman, Z. O. Guler, and O. Kalipsiz. Web application testing with model based testing method: case study. In *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, pages 1–6. IEEE, 2020.

[2] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. IEEE International Conference on Formal Engineering Methods, Brisbane, 1, AS, 1998-11-01 00:11:00 1998.

[3] L. Barros, C. Hirata, J. Marques, and A. M. Ambrosio. Generating test cases to evaluate and improve processes of safety-critical systems development. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 311–318. IEEE, 2020.

[4] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[5] M. Ben-Ari. A primer on model checking. *ACM Inroads*, 1(1):40–47, 2010.

[6] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL—a tool suite for automatic verification of real-time systems*. Springer, 1996.

[7] A. Berdasco, A. Martínez, and C. Quesada-López. Evaluating a model-based software testing approach in an industrial context: A replicated study. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–7. IEEE, 2019.

[8] D. Beyer and T. Lemberger. Software verification: Testing vs. model checking. In O. Strichman and R. Tzoref-Brill, editors, *Hardware and Software: Verification and Testing*, pages 99–114, Cham, 2017. Springer International Publishing.

[9] P. E. Black, V. Okun, and Y. Yesha. *Mutation of Model Checker Specifications for Test Generation and Evaluation*, pages 14–20. Springer US, Boston, MA, 2001.

[10] K. C. Castillos, F. Dadeau, and J. Julliand. Coverage criteria for model-based testing using property patterns. *arXiv preprint arXiv:1403.7259*, 2014.

[11] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on software Engineering*, 24(7):498–520, 1998.

[12] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294, 1999.

[13] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal smc tutorial. *International journal on software tools for technology transfer*, 17:397–415, 2015.

[14] K. I. Eder, W.-l. Huang, and J. Peleska. Complete agent-driven model-based system testing for autonomous systems. *arXiv preprint arXiv:2110.12586*, 2021.

[15] E. P. Enoiu, A. Causevic, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson. Automated test generation using model checking: an industrial evaluation. *International Journal on Software Tools for Technology Transfer*, 18:335 – 353, 2014.

[16] M. M. Eslamimehr. The survey of model based testing and industrial tools. *Master's Thesis, Linköping University*, 2008.

[17] C. S. Gebizli and H. Sözer. Automated refinement of models for model-based testing using exploratory testing. *Software Quality Journal*, 25:979–1005, 2017.

[18] C. Ş. Gebizli and H. Sözer. Impact of education and experience level on the effectiveness of exploratory testing: An industrial case study. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 23–28. IEEE, 2017.

[19] C. S. Gebizli, H. Sözer, and A. Ercan. Successive refinement of models for model-based testing to increase system test effectiveness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 263–268, 2016.

[20] C. Gebizli, D. Metin, and H. Sözer. Combining model-based and risk-based testing for effective test case generation. In *2015 IEEE Eighth International*

Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 1–4, 2015.

[21] W.-l. Huang and J. Peleska. Complete requirements-based testing with finite state machines. *arXiv preprint arXiv:2105.11786*, 2021.

[22] K. Karl. Graphwalker. *URL: www. graphwalker. org [accessed: 2023-02-18]*, 2013.

[23] A. K. Karna, Y. Chen, H. Yu, H. Zhong, and J. Zhao. The role of model checking in software engineering. *Front. Comput. Sci.*, 12(4):642–668, aug 2018.

[24] P. Koopman, P. Achten, and R. Plasmeijer. Testing and validating the quality of specifications. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 41–52. IEEE, 2008.

[25] S. Kriebel, M. Markthaler, K. S. Salman, T. Greifenberg, S. Hillemacher, B. Rumpe, C. Schulze, A. Wortmann, P. Orth, and J. Richenhagen. Improving model-based testing in automotive software engineering. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 172–180, 2018.

[26] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1:134–152, 1997.

[27] M. Lindgren. Practical verification of stateful embedded c code using finite state machines and vcc, 2020.

[28] R. Marinescu, C. Seceleanu, and P. Pettersson. An integrated framework for component-based analysis of architectural system models. In B. Nielsen and C. Weise, editors, *Proceedings of the 24th IFIP International Conference on Testing Software and Systems (ICTSS12) Doctoral Workshop*, pages 1–6. Technical Report No. 12-201. ISBN:1601-0590 Aalborg University, November 2012.

[29] A. Marques, F. Ramalho, and W. L. Andrade. Comparing model-based testing with traditional testing strategies: An empirical study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 264–273. IEEE, 2014.

[30] E. Martins, S. B. Sabião, and A. M. Ambrosio. Condata: a tool for automating specification-based test case generation for communication systems. *Software Quality Journal*, 8(4):303–320, 1999.

[31] E. J. Njor, F. Lorber, N. I. Schmidt, and S. R. Petersen. Conformance testing in uppaal: A diabolic approach. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 35–42, 2020.

[32] R. P. Pontes, P. C. Véras, A. M. Ambrosio, and E. Villani. Contributions of model checking and cofi methodology to the development of space embedded software. *Empirical Software Engineering*, 19:39–68, 2014.

[33] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering*, pages 392–401, 2005.

[34] I. A. Qureshi and A. Nadeem. Gui testing techniques: a survey. *International Journal of Future computer and communication*, 2(2):142, 2013.

[35] H. Robinson. Finite state model-based testing on a shoestring. In *Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999)*. Citeseer, 1999.

[36] A. Sapan, B. Öztekin, E. Ünsal, and A. Şen. Testing openapi banking payment system with model based test approach. In *2020 Turkish National Software Engineering Symposium (UYMS)*, pages 1–4. IEEE, 2020.

[37] V. Schuppan. *Liveness checking as safety checking to find shortest counterexamples to linear time properties*. ETH Zurich, 2006.

[38] N. Setiani, R. Ferdiana, P. I. Santosa, and R. Hartanto. Literature review on test case generation approach. In *Proceedings of the 2nd International Conference on Software Engineering and Information Management*, ICSIM 2019, page 91–95, New York, NY, USA, 2019. Association for Computing Machinery.

[39] P. D. B. d. Silva, A. M. Ambrosio, and E. Villani. Model-based testing applied to software components of satellite simulators. *Modelling and Simulation in Engineering*, 2018:1–14, 2018.

[40] S. Tiwari, K. Iyer, and E. P. Enoiu. Combining model-based testing and automated analysis of behavioural models using graphwalker and uppaal. In *2022 29th Asia-Pacific Software Engineering Conference (APSEC)*, pages 452–456, 2022.

[41] J. Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pages 1–38, 2008.

[42] E. Villani, R. P. Pontes, G. K. Coracini, and A. M. Ambrósio. Integrating model checking and model based testing for industrial software development. *Computers in Industry*, 104:88–102, 2019.

[43] L. Ye. Model-based testing approach for web applications. Master's thesis, 2007.

[44] M. N. Zafar, W. Afzal, E. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui. Model-based testing in practice: An industrial case study using graphwalker. In *14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC 2021, New York, NY, USA, 2021. Association for Computing Machinery.

[45] J. Zander, I. Schieferdecker, and P. J. Mosterman. *Model-based testing for embedded systems*. CRC press, 2017.