TESTING MICROSERVICE APPLICATIONS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Özgür ÖZTÜRK

December 2023
İZMİR

We approve the thesis of **Özgür ÖZTÜRK**

Examining Committee Members:

Prof. Dr. Tolga AYAV
Department of Computer Engineering, İzmir Institute of Technology

Prof. Dr. Onur DEMİRÖRS
Department of Computer Engineering, İzmir Institute of Technology

Assoc. Prof. Dr. Tuğkan TUĞLULAR
Department of Computer Engineering, İzmir Institute of Technology

Assoc. Prof. Dr. Ahmet Tuncay ERCAN
Department of Management Information Systems, Yaşar University

Asst. Prof. Emrah İNAN
Department of Computer Engineering, İzmir Institute of Technology

8 December 2023

Prof. Dr. Tolga AYAV
Supervisor, Department of Computer Engineering,
İzmir Institute of Technology

Prof. Dr. Onur DEMİRÖRS
Supervisor, Department of Computer Engineering,
İzmir Institute of Technology

Prof. Dr. Cüneyt Fehmi BAZLAMAÇCI
Head of Computer Engineering Department

Prof. Dr. Mehtap EANES
Dean of the Graduate School of Engineering and Sciences

ACKNOWLEDGMENTS

First of all, I would like to thank you my thesis advisor, Assoc. Prof. Tolga Ayav. He always guided and encouraged me throughout my graduate education. Whenever I encountered a problem or got stuck, he calmed me down and guided me in the right direction. Therefore, I thank you very much for both your guidance and your patience.

I would also like to thank you Prof. Dr. Onur Demirörs and Assoc. Prof. Tuğkan Tuğlular for sharing their knowledge and inspiring interest in my thesis topic. Thanks to the training they provided, I have completed this study.

Finally, I must express my deepest gratitude to my wife, mother, father and sisters, who provided me with constant support and constant encouragement throughout my working years and throughout the research and writing of this thesis. Without them this success would not have been possible. Thank you.

ABSTRACT

TESTING MICROSERVICE APPLICATION

This thesis contributes to the testing processes of microservice architecture. Microservices provide a scalable, reliable and cloud-based environment that is frequently preferred in today's technology applications. It consists of small, loosely coupled, isolated applications that work in harmony. In this study, microservice application is modeled using timed automata and model checker-based testing methods are exploited to generate test cases automatically. To this end, UPPAAL model checker tool is utilized. The model of the microservice application is mutated with respect to a set of fault hypotheses and these mutant models are verified against certain properties defined by system or application specifications. The returned counterexamples from the model checker are used to constitute the test cases. The entire process is automated and experimentally run for an example application. The generated test cases are also shown to be efficiently detect the errors. The proposed testing methodology has the benefits like a faster test generation process and achieving test cases with better fault detection capability.

ÖZET

MİKROSERVİS UYGULAMALARININ TESTİ

Bu tez, mikroservis mimarisinin test süreçlerine katkıda bulunmaktadır. Mikroservisler günümüz teknoloji uygulamalarında sıklıkla tercih edilen ölçeklenebilir, güvenilir ve bulut tabanlı bir ortamda sağlamaktadır. Uyum içinde çalışan küçük, gevşek bağlı, izole uygulamalardan oluşur. Bu çalışmada, mikroservis uygulaması zamanlanmış otomatlar kullanılarak modellenmiş ve test senaryolarının otomatik olarak oluşturulması için model denetleyici tabanlı test yöntemlerinden yararlanılmıştır. Bu amaçla UPPAAL model denetleyici aracından yararlanılmaktadır. Mikroservis uygulamasının modeli, bir dizi hata hipotezine göre mutasyona uğratılır ve bu mutant modeller, sistem veya uygulama spesifikasyonları tarafından tanımlanan belirli özelliklere göre doğrulanır. Model denetleyiciden döndürülen karşı örnekler, test senaryolarını oluşturmak için kullanılır. Tüm süreç otomatikleştirilmiştir ve örnek bir uygulama için deneysel olarak çalıştırılmıştır. Oluşturulan test senaryolarının aynı zamanda hataları etkili bir şekilde tespit ettiği de gösterilmiştir. Önerilen test metodolojisi, daha hızlı bir test oluşturma süreci ve daha iyi hata tespit kapasitesine sahip test senaryoları elde etme gibi avantajlara sahiptir.

TABLE OF CONTENTS

ACKNOWLEDGMENTS....................i

ABSTRACT....................ii

TABLE OF CONTENTS....................iv

LIST OF FIGURES....................vi

LIST OF TABLES....................viii

CHAPTER 1. INTRODUCTION....................1

CHAPTER 2. DESIGN SPECIFICATION....................4
- 2.1. Monolithic Architecture....................4
- 2.2. Microservice Architecture....................6

CHAPTER 3. MODEL CHECKER-BASED TESTING AND UPPAAL....................10
- 3.1. Model Checker-Based Testing....................10
- 3.2. Uppaal....................11

CHAPTER 4. PROPOSED METHOD....................16
- 4.1. Creating the Model....................17
- 4.2. Verifying the Model....................29
- 4.3. Simulating the Model....................31
- 4.4. Creating Test Cases....................34
  - 4.4.1. Deleting transitions 35
  - 4.4.2. Updating Guard and Transition Values 36
- 4.5. Cleaning Test Scenarios 40
- 4.6. Verifying Test Models 42

4.7. Making Sense of Verify Output of Test Models 43

CHAPTER 5. DISCUSSIONS AND RESULT...46

CHAPTER 6. CONCLUSION AND FUTURE WORK 50

CHAPTER 7. REFERENCES...52

LIST OF FIGURES

Figure **Page**

Figure 1. Monolithic Architecture....................5

Figure 2. Microservice Architecture....................8

Figure 3. The editor in the Uppaal GUI....................13

Figure 4. The simulator in the Uppaal GUI....................14

Figure 5. The verifier in the Uppaal GUI....................14

Figure 6. The Test Strategy Flow 17

Figure 7. Model template structure....................18

Figure 8. BusinessProcess template structure....................19

Figure 9. BusinessProcess .xml file structure....................20

Figure 10. Circuit breaker structure....................22

Figure 11. Gateway template structure....................23

Figure 12. Gateway Template .xml file structure....................24

Figure 13. Microservice template structure....................25

Figure 14. Microservice template .xml file structure....................26

Figure 15. Transaction between “START” and “Before Microservice Response” has been broken....................27

Figure 16. Only in Microservice3, the tarasaction between “START” and “BeforeMicroserviceResponse” is broken....................27

Figure 17. MicroserviceFail template .xml file structure....................28

Figure 18. Uppaal verifier tab....................29

Figure 19. Uppaal simulator tab....................31

Figure 20. Enabled transitions....................32

Figure 21. Simulation trace....................32

Figure 22. Variables and constraints....................33

Figure 23. Simulation trace screen....................33

Figure 24. Simulation screen....................34

Figure 25. Transitions deleted model....................36

Figure 26. Model sytem declarations section....................36

Figure 27. BusinessProcess1 object..37
Figure 28. Gateway1 object..38
Figure 29. Microservice1, Microservice2 and Microservice3 object.............................40

Figure 30. Fail scenario example for Microservice1...41
Figure 31. Fail scenario example for Microservice2...41
Figure 32. Verify function..42
Figure 33. XTR file format example...43
Figure 34. A certain part of the expansion of the XTR file format................................44
Figure 35. Uppaal visual version of Figure 34..45

LIST OF TABLES

Table **Page**

Table 1. Transitions creation data of deleted scenarios..47

Table 2. Transitions query test data of deleted scenarios..48

Table 3. Data of scenarios created by updating Transitions and Guard values...............48

Table 4. Query test data created by updating Transitions and Guard values..................49

CHAPTER 1

INTRODUCTION

Microservice architecture is a highly popular architecture today. It has increased rapidly in the technology sector since 2015 (*1*). Nowadays, scalable, reliable and cloud-based environments are preferred when developing applications. New approaches and patterns have been prepared for development in these environments. One of these is microservice architecture (*2*). Microservice architecture creates small, loosely coupled, isolated applications that work in harmony. While microservices have high dependencies within themselves, their dependencies on other microservices are low. As the use of this popular architecture increases, new needs emerge. One of these areas is testing processes.

Microservice architectures are complex structures, therefore they are complex and costly in testing processes. Testing processes are a very important step in application development processes. The testing processes of this architecture, which is highly dependent and costly, are equally affected by this situation and are costly and complex.

Nowadays, more than one testing method is applied to test microservice architectures. Because when microservice architectures are developed, they are developed on an event-based basis, and these events may or may not be dependent on other events. Therefore, when we develop an application, we can create it using one or more microservices. At the same time, microservice architecture may require more than one software language, more than one database technology and more than one server technology. Considering all these flows, testing processes appear as a costly, complex and difficult process in the face of monolithic applications. In microservice architecture, multiple testing methods are mainly used to evaluate security, compatibility, traceability, complexity, performance, effectiveness and scalability. Some of these test methods; unit testing, component testing, integration testing, contract testing and end to end testing (E2E) (*3*). Testing processes consist of a complex strategy. Because it is necessary to consider microservices as a whole, both within themselves and with their dependencies, and to develop a testing strategy.

Unit tests: Testing the smallest part that can be used in services. In this way, we isolate the smallest parts in applications and make sure they work correctly. Since microservices are created on an event-based basis, there will be a testing process suitable for our architecture and it will be easy to implement.

Component tests: We create microservices based on events. After testing the smallest parts in microservices, we need to consider them as a whole. For this reason, it is the method by which we test whether the microservice we have created provides the functionality we promise.

Contract tests: APIs make a contract among themselves to communicate with each other. We can see this contract as the data format that APIs send to each other. Any significant change to the APIs may cause this format to be incorrect and communication may not occur. To avoid this situation, the accuracy of the contracts must be tested. Microservices may also communicate with each other, so this is an important test to ensure that there are no errors in these communications.

Integration tests: Integration tests are used to detect and fix interface errors. There can be more than one API in a microservice architecture. Therefore, we must make sure that these APIs present the right information in the right format. In this way, we can evaluate how all modules work when combined.

End-to-end tests (E2E): End-to-end tests generally test the functionality of the application. It is tested considering the application's scope of use and features.

It is very important that applications that receive constant updates appear to the user with minimum errors. Any mistake can cause great damage or face severe repercussions within seconds. For this reason, it was a source of motivation for us to work within the scope of microservice architecture, which is a popular application architecture, and the testing processes that are critical when developing applications.

In our study, we used the Uppaal tool, which is a modeling tool that models real-time systems using time automata and allows these systems to communicate using time variables, simulate and verify them. Using the Uppaal tool, we created a general application flow simulation by modeling our business process, gateway model and microservice modules in our system. We used the model checker-based testing method as the testing approach. Thus, we were able to create test scenarios and test our processes. We have automated the testing processes using the Python programming language. In this way, we can quickly create test cases of the model we have previously created with the values we have given and verify them. In this way, we can see at what point in our model, in which time period it works and in which time period it receives errors. In this way, we can detect and correct fault areas during testing processes.

CHAPTER 2

DESIGN SPECIFICATION

Today, software applications are built on specific architectures according to needs and preferences. Built software applications are divided into layers within themselves. These are the **client (frontend) layer** where users experience and interact, the **server layer** where the service and logic components of the applications are located, and finally the **database layer** where the data obtained in the applications are collected (*4*). We will talk about two architectures used when developing software applications. These; Monolith Architecture and Microservice Architecture.

There are differences between **Microservice** and **Monolithic** architectures. In microservice architecture, each service has an event and should have as little dependency on other services as possible. However, in monolithic architecture, all services are collected in one place. For this reason, if there is an interruption in any of the services in applications developed with microservice architecture, the service of the relevant service will be disrupted. However, in applications developed with monolithic architecture, the entire structure is affected by this situation (*5*). Let's evaluate Netflix, one of the popular applications today. Basically, it consists of three microservices audio, video and subtitles. Even if one of these services is unable to provide service, we may not miss the flow of the movie we are watching because other services can provide service. However, considering that it is developed with a monolithic architecture, we may miss the flow of events of the movie we are watching in case any service cannot provide service.

2.1. Monolithic Architecture

Structures that collect software components under a single roof are called monolith structures. Since monolith structures do not have separate components and modules, they do not need a distribution tool to organize the communication between these modules (*6*). The client, server and database layers we mentioned in monolith structures are under a single roof. Therefore, a negative situation that may occur in any of these areas will affect the whole system.

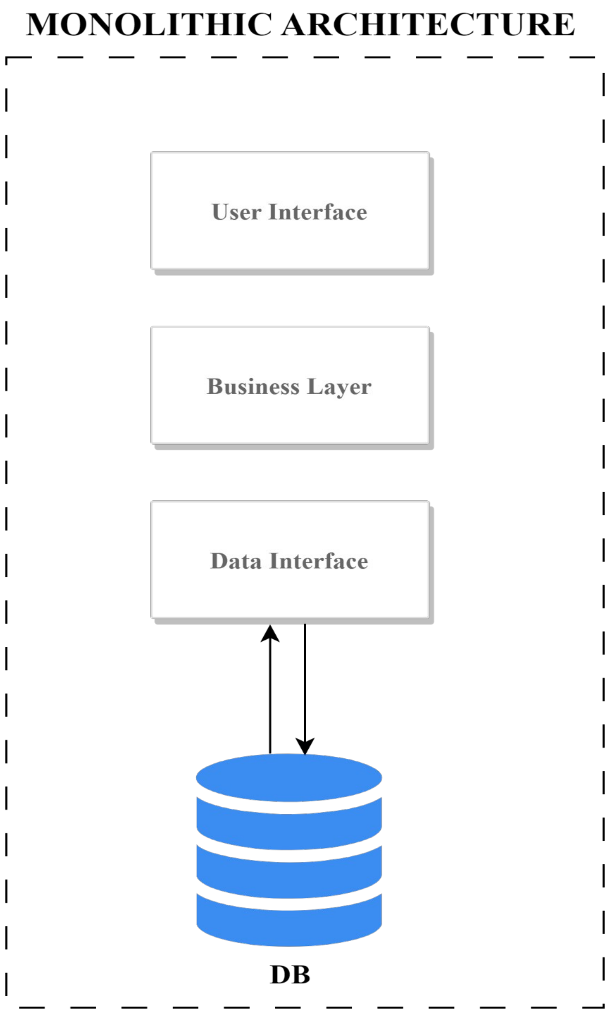


Figure 1. Monolithic Architecture.

In monolith structures, when system growth is needed, vertical growth is generally achieved.

Vertical growth; In order to increase the performance of the system, it ensures faster processing of incoming traffic to the system by increasing processor power, capacity and bandwidth. Thus, faster data flow is provided to the system, and since the provided data will be processed faster, the user will be returned in a shorter time. But this growth structure has disadvantages. Structures that achieve vertical growth also have limited redundancy, scaling and flexibility capabilities. For this reason, we may experience problems in the future. As a result, we can increase the resources of the existing server to a certain level.

2.2. Microservice Architecture

In the developing software world, it has prepared the infrastructure to develop scalable, reliable and cloud-based applications rather than preferring traditional monolithic applications. New architectural approaches and patterns have been developed to carry out these works. Microservice architecture is one of them. The goal of microservice architecture is to create small, loosely coupled, isolated applications that work in harmony (*2*).

The increasing capabilities of cloud systems combined with the latest developments in software architectures have paved the way for the development of much more scalable, responsive and reliable applications. With the provision of these opportunities, microservices-based architecture is becoming widespread in leading institutions. When developing microservice-based systems, they consist of multiple microservices that are compatible with each other and have as few dependencies as possible. While microservices have high dependency on themselves, their dependency on other microservices is low. Multiple microservices can be used to provide complex services (*2*).In this way, services can be scaled, shaped and fault tolerance is minimized (*4*).

The popularity of microservice architecture has increased rapidly in the technology sector, especially since 2015. Today, companies such as Netflix, Amazon, LinkedIn, Uber, SoundCloud, and Verizon have adopted microservice-based approaches in the services they provide (*1*). It seems that popular microservices will be used and will continue to be used in smart city applications and many other areas (*4*).

We can position microservices on two basic features. These are evolutionary design and choreography. Since we have more than one service in the applications, these services need to communicate with each other. They establish this communication without any center. Thus, this represents our choreography feature. Evolutionary design, on the other hand, advocates the creation of new services for this new module when a new module is added to our applications (*4*).

Microservice architecture is an architecture built on SOA (Service Oriented Architecture). In SOA architecture, applications can be distributed on one or more machines. It is the architecture that allows distributed applications to communicate in the distribution system. However, while SOA is an architecture used at the enterprise level, microservice architecture is an architecture used at the application level.

Microservice-based architectures of service-oriented architecture (SOA) are distinguished from each other under three main headings. These ; Size, boundary context and independence. In terms of size, microservices are smaller and have only one task. Considering the boundary context, a microservice combines all dependent functions into a single service. Regarding independence, microservices are independent services (*1*).

The software language or database technology in which the application is used may not be suitable for performing some tasks brought by the application using microservices. Therefore, we may need to choose different programming languages and different database technologies. Microservice architecture makes this possible. Thus, we may have used more than one programming language and database technology in the applications we developed. Developing applications as separate components also allows developers to better understand the written code and intervene faster. Thus, microservice architecture provides modularity, high cohesion and loose coupling. Microservice architecture helps us manage code duplication and increasing complexity by separating independent services in large-scale applications. This helps us isolate errors that may occur in the services and prevents the error from affecting the entire system. Since each microservice has a separate service in applications created in microservice architecture, the duties and responsibilities of the teams to be formed in the development of the services will be clear. In this way, the management of processes can be easily achieved by both the application side and the teams. These situations affect each other and create a chain positive effect in the fields of testing processes, security processes, user feedback and application reputation.

Microservices have advantages as well as disadvantages. First of all, managing applications consisting of more than one microservice as a whole can become both costly and complex. Testing processes and examining error situations are equally costly and complex. When we consider the reverse processes, we will need to first test the dependencies of each microservice and then test itself.

Key features of microservices:

- **Decoupling:** Services should be largely independent of each other.
- **Componentization:** We need to switch services easily.
- **Business Capabilities:** Its structure should be simple and task specific.
- **Autonomy:** Each service should have its own specific task so that development and testing will be faster.

- **Continuous Delivery:** If the services are separate, the distribution cost will decrease.
- **Decentralized Governance:** Separate tools and programming languages can be used in each service.
- **Agility:** When we update services we can do it quickly.

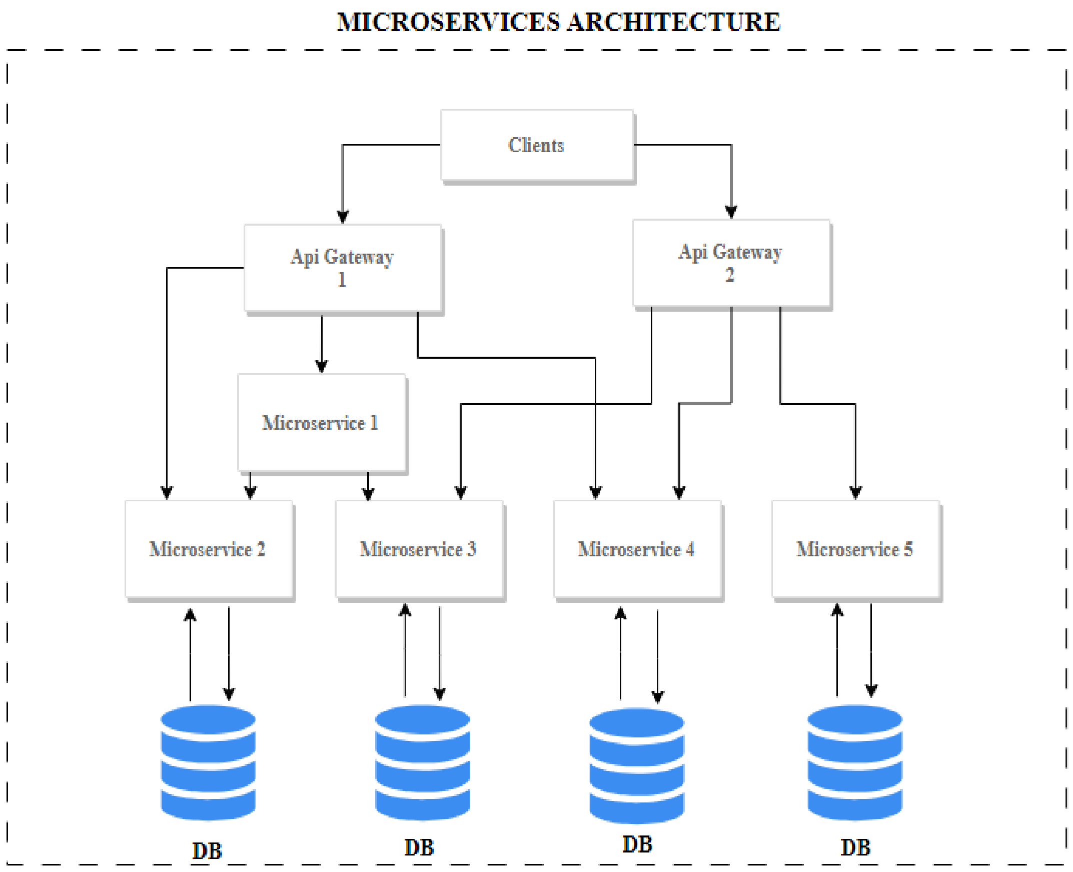


Figure 2. Microservice Architecture.

In microservice architecture, horizontal growth is suitable when system growth is needed. By achieving horizontal growth in systems, redundancy, scalability and flexibility are ensured.

Redundancy; Since systems can grow horizontally, they consist of more than one server. For this reason, in case of a negative situation that may occur in the servers (due to malfunction, increase in traffic, etc.), we may need to increase server resources. Therefore, we can increase the number of servers in our system.

Scalability; If an increase in load is observed in our system, we increase the performance of the system by increasing the number of servers to respond to users. Thus, our system becomes responsive to users.

Flexibility; Due to the situations mentioned above, we can quickly increase or decrease the resources of our system when necessary.

CHAPTER 3

MODEL CHECKER-BASED TESTING AND UPPAAL

3.1. Model Checker-Based Testing

If an application you will test is based on scenarios, you can use model checker-based testing. Since our study is based on scenarios, we applied the model checker-based testing method. In this testing method, we designed a model for applications and then created the states and relationships of this designed model. Thus, when we tested our application, we determined when and in what state our model would be and where and how it would proceed. This model we created was created to reflect the behaviors expected from the system. Thus, we were able to create and evaluate test scenarios based on the model designed to realize the expected behaviors.

During the model creation processes, we viewed the application as a whole from a framework. In this way, we were able to better see the details we needed to pay attention to while performing our tests. In the Model Checker-Based Testing method, we generally experienced our model by performing process flow scenarios to check our model. We updated our model again by seeing the parts that we overlooked and forgot to model in these flow scenarios. This allowed us to make sure that the entire system was working as we wanted.

Model Checker-Based Testing is based on testing by taking into account the scenarios in the counter example rather than the scenarios we will test. The states and relationships of the model were created by multiple variables. Developing the model relationally and parametrically in this way enabled us to automatically create counter scenarios that may be outside our scenarios. Thus, the inputs taken from the scenarios became our test inputs, and the processing of these values by the model and producing a result formed the output of our test scenarios (*7*).

In the Model Checker-Based Testing testing method, the models of the applications and the situations in these models are taken as input, and the time-dependent value affecting the transactions is taken as input. Queries were written to test that the model we designed worked to reflect the behavior expected from the system. While writing these queries, we wrote them according to Uppaal's query standards. The syntax used in queries to test our model are as follows;

- A <> (condition) When the expression is used with a condition, it means that the model will take at least one journey in time and only in one time period.
- A [] (condition) When used with a condition, it means that the model will always take at least one journey over time.
- E <> (condition) When the expression is used with a condition, it means that all journeys in the model will take place in only one time period.
- E [] (condition) When used with a condition, it means that all journeys in the model will always occur over time.

To determine whether the created model is suitable, we verified our model by creating conditions with the syntax mentioned above. The model checker created output by checking the input it received. This output gave us an output of the progress of the scenarios realized on the model (*8*).

To carry out the testing processes with the Model Checker-Based Testing method, we evaluated the counter tests as test scenarios and created more test scenarios by making changes to the input values of these test scenarios. We used the mutation method on our model to increase the test scenarios (*9*).

3.2. Uppaal

Uppaal is a tool for modeling real-time systems and allowing us to simulate and verify these systems. The Uppaal vehicle was built jointly by two universities. These are the Fundamental Research in Computer Science at Aalborg University and the Department of Information Technology at Uppsala University in Sweden (*10*). The Uppaal tool takes its name from the names of the two universities that played a role in its development.

The Uppaal tool allows the creation of large models. While doing this, it creates timed automata in parallel. It also allows created models to communicate using shared discrete and clock variables (*11*). It has channels for communication between processes.

Of these channels, binary ones are used to synchronize a pair of operations. The other channel, the broadcast channel, sends events to all processes (*12*). The Uppaal tool uses “Timed Automata” when creating models. Timed Automata are used when modeling the behavior of real-time systems. Timed Automata use a finite number of real-valued clock variables and realize transitions in states using timing constraints (*13*). The transitions of the automata are compared with the clock variables to ensure the transitions, thus determining the behavior of the automata.

We can represent Timed Automata formally as follows.

Timed Automaton is a tuple $<C,\Sigma,L, L^0, I, E>$

- C: Elements is a finite set of clocks of timed automaton,
- Σ: Elements is a finite set of labels of timed automaton,
- L: Elements is a finite set of locations of timed automaton,
- $L^0 \subseteq L$ is a set of initial locations of timed automaton,
- $I \subseteq L$ is the set of accepting locations of timed automaton.
- $E \subseteq L \times \Sigma \times 2^C \times \Phi(C) \times L$ is a set of edges, called transitions of timed automaton,
    - 2^C is the set of clock constraints involving clocks from C,
    - $\Phi(C)$ is the powerset of C

Timed automaton $(s, \sigma, g, \lambda, s')$ represents an edge from E is a transition from locations s to s' with action σ, g is guard value and λ is a clock resets. (*13,14*).As a result, the models we created in Uppaal consist of timed automatons.

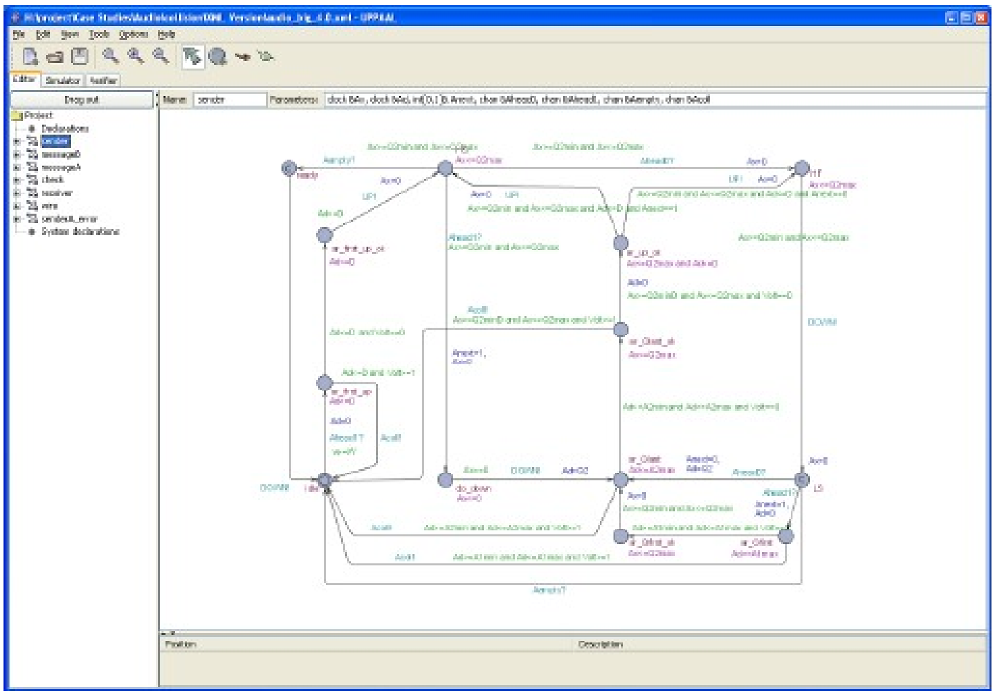


Figure 3. The editor in the Uppaal GUI (*10*).

We can examine the Uppaal tool in three main sections. These; Description language, Simulator, Model-Checker.

Description Language: It is a protected command language that has the data types required to create the model. It has an extended content with time and data variables (*10*).

Simulator: It is a verification tool where we can see the whole of the designed model and its relationships, and also correct our mistakes before verifying (*10*).

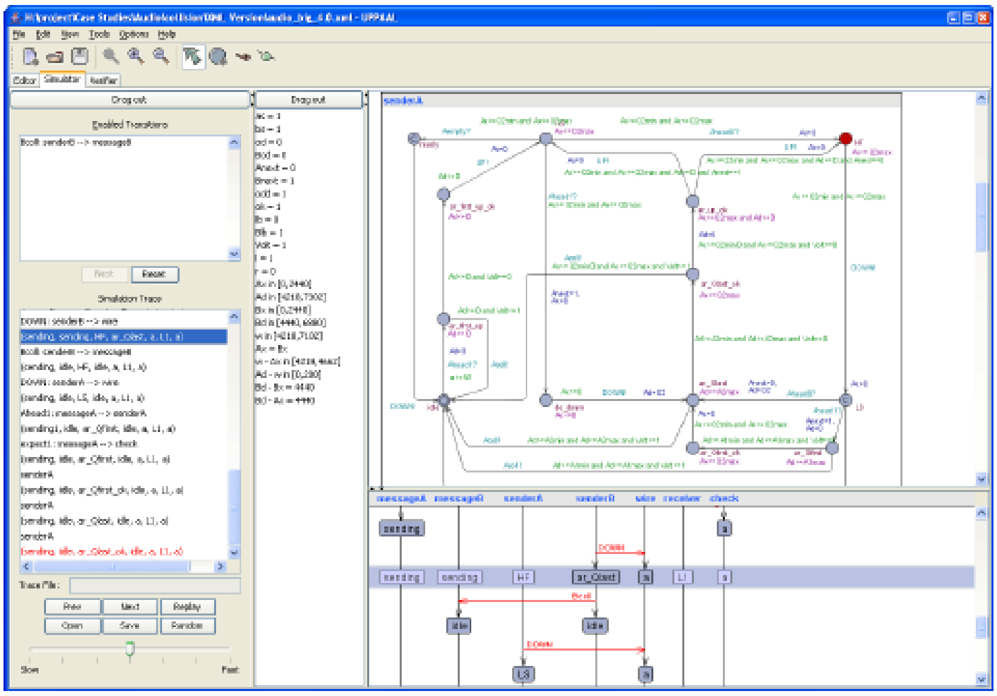
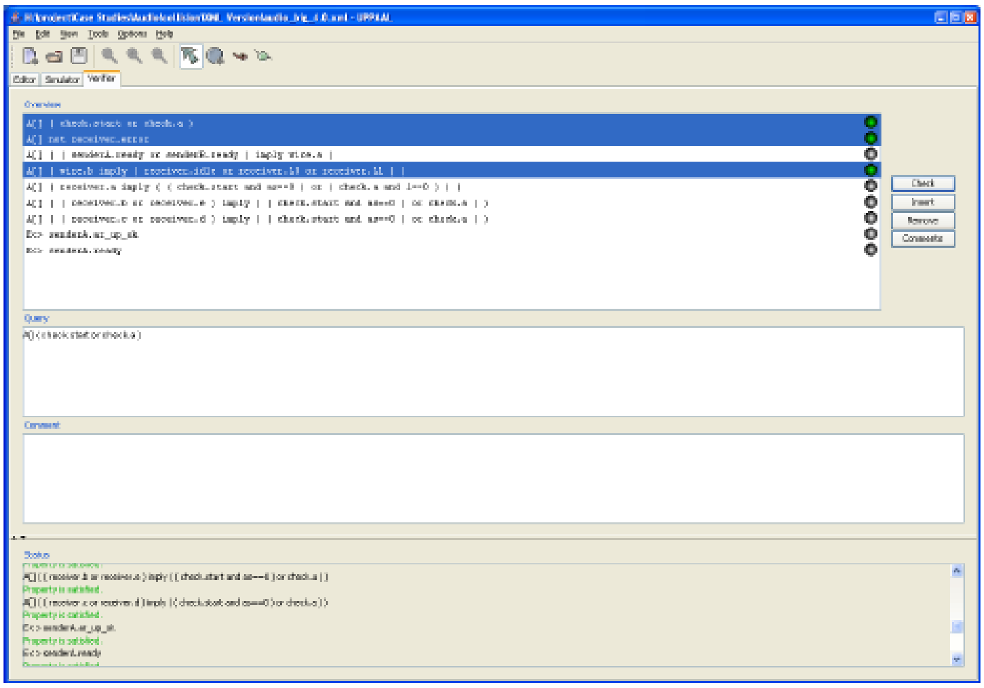

File Edit View Tools Options Help
Editor Simulator Verifier
Enabled Transitions
Simulation Trace
Prev Next Replay
Open Save Random
Slow Fast
senderA
messageA messageB senderA senderB wire receiver check

Figure 4. The simulator in the Uppaal GUI (*10*).

Model-Checker: It analyzes and checks the accessibility of the created scenarios by looking at the state space of the created system (*10*).



```
A[] ( check.start or check.a )
A[] not receiver.error
A[] ( ( senderA.ready or senderB.ready ) imply wire.a )
A[] ( wire.b imply ( receiver.idle or receiver.L0 or receiver.L1 ) )
A[] ( receiver.a imply ( ( check.start and ss==0 ) or ( check.a and l==0 ) ) )
A[] ( ( receiver.b or receiver.e ) imply ( ( check.start and ss==0 ) or check.a ) )
A[] ( ( receiver.c or receiver.d ) imply ( ( check.start and ss==0 ) or check.a ) )
E<> senderA.ar_up_ok
E<> senderA.ready
```

```
A[] ( check.start or check.a )
```

```
A[] ( ( receiver.b or receiver.e ) imply ( ( check.start and ss==0 ) or check.a ) )
Property is satisfied.
A[] ( ( receiver.c or receiver.d ) imply ( ( check.start and ss==0 ) or check.a ) )
Property is satisfied.
E<> senderA.ar_up_ok
Property is satisfied.
E<> senderA.ready
Property is satisfied.
```

Figure 5. The verifier in the Uppaal GUI (*10*).

With the Uppaal tool, you can simulate the models you created with scenarios and exportthem. In this way, it provides an output that shows how your model performed in which situations and variable values. In this way, you can improve your model.

To test the model you have developed, it is developed using the model checker-based testing method. First, the workflow of the system you plan to test is determined. The workflow of the system you specify is adapted to the model. You take the system apart to adapt it to the model. Breaking it into parts is to carry out testing studies in detail and comprehensively by updating the dependencies as desired while performing the test scenarios to be implemented in the system. Each separated piece is referred to as a template in uppaal. The templates created consist of “locations”. Templates are a set consisting of “Timed Automaton”. Since templates consist of "Timed Automaton", template behavior is determined by restricting the behavior of locations in time-dependent changes and conditions. After the model is created, the scenarios prepared for the model are checked by going to the verifier tab of the Uppaal tool. To examine the flows of the model in detail, go to the simulator tab and examine the verified scenario in detail.

CHAPTER 4

PROPOSED METHOD

We develop our applications modularly using microservice architectures, one of today's popular architectures. Each module has different tasks and these services should be as loose coupling as possible. Since services have low dependency and are modular, services can be scaled, shaped and fault tolerances can be minimized.

In order for applications to work stably, tests and maintenance must be carried out. In this study, we carried out a study on model checker-based based testing of microservices. Testing is necessary to ensure that microservices encounter the least errors and problems. However, considering the modular structure of microservices and the possibility of their dependence on other services, testing costs and complexity increase.

In our study, the Model Checker-Based Testing method was used together with the UPPAAL tool. We prepared a sample application model so that we can apply the Model Checker-Based Testing method. There were situations we had to pay attention to while preparing the model.

These;

- The model we will prepare should be as close as possible to the architecture used in real-life applications and have a structure that can be developed,
- Being able to change our model in a generic way,
- Being able to automatically generate test-case scenarios,
- Being able to make sense of the results of the test cases we wrote,

We carried out our study considering the above mentioned situations. While preparing the model, we made sure that the model could be improved if necessary and that it was as close as possible to the architecture used in real-life applications.

We used parameters in our model to automatically create test cases, so we did not have to create the model again and again so that we could test as many scenarios as we needed. Creating the model in this way made it easier for us to create test cases. At the same time, we will be able to create mutant scenarios and increase our number of scenarios.

To create test cases, we wrote test case generation using the python programming language. Thus, you can create test scenarios automatically.

In order to make sense of the results of the written test cases, scripts were written using the python programming language. Therefore, we automatically made sense of the entire process and evaluated our testing processes.

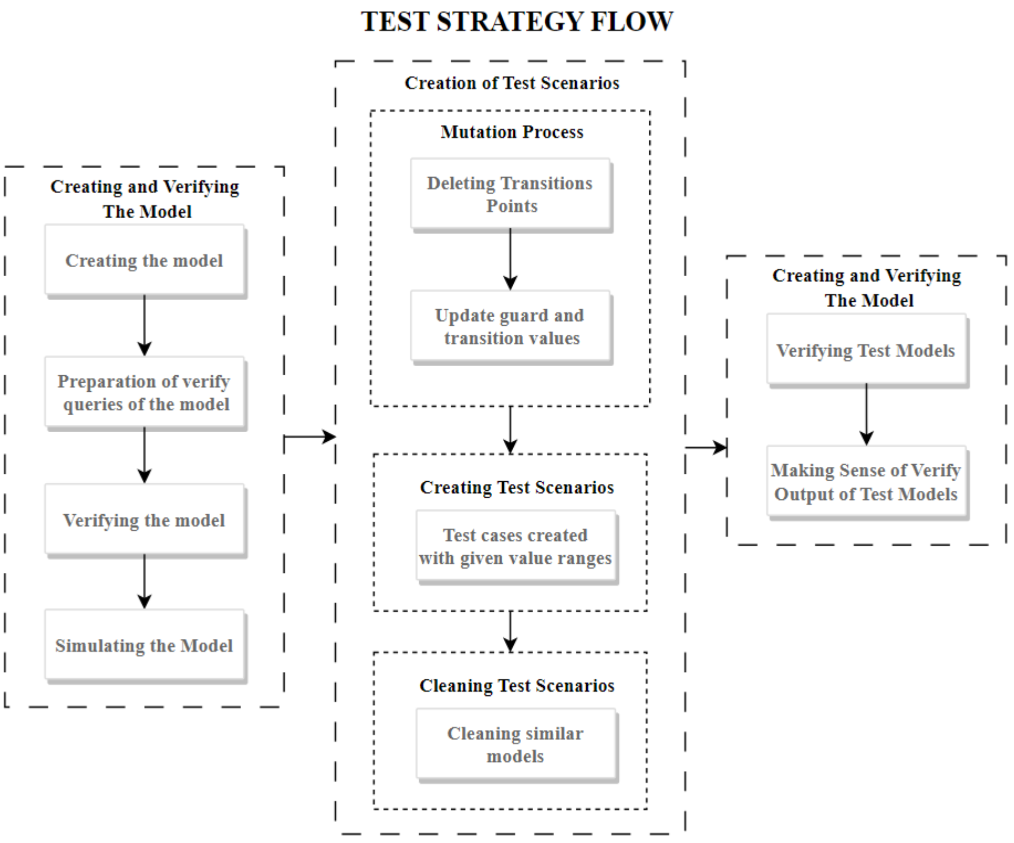

TEST STRATEGY FLOW
Creating and Verifying The Model
Creating the model
Preparation of verify queries of the model
Verifying the model
Simulating the Model
Creation of Test Scenarios
Mutation Process
Deleting Transitions Points
Update guard and transition values
Creating Test Scenarios
Test cases created with given value ranges
Cleaning Test Scenarios
Cleaning similar models
Creating and Verifying The Model
Verifying Test Models
Making Sense of Verify Output of Test Models

Figure 6. The Test Strategy Flow

We carried out our study to improve testing processes in microservice applications. In our study, we used the model checker-based testing method with the UPPAAL tool. We carried out our study by following the steps below.

4.1. Creating the Model

We created our main model using the uppaal tool. “Templates” are used to create our models in the uppaal tool. In templates, we associate states, which we call locations, with transactions. Since the response times of the services are evaluated in milliseconds, we evaluated the "clk" and "c" values, which are the clock type we used in our model, in milliseconds (ms). We used four templates to create our model.

These;

- BusinessProcess,
- Gateway,
- Microservice,
- MicroserviceFail

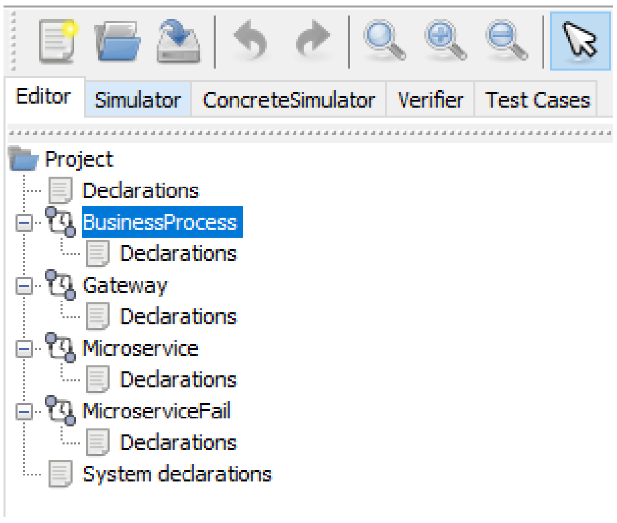


Figure 7. Model template structure.

BusinessProcess: This template is our starter template (*Fig 8*). Test Cases have been prepared based on parameters in order to automatically create scenarios.Thus, the template is to create test cases by giving parametric values, determining time intervals according to the parameters we want. It starts with the “START” location. We aimed to make accurate measurements in terms of time by updating the c(clock) value to "0" in the first location. Invariant value c<=0 indicates the time when the incoming request will leave the specified location. Then, the request comes to the “BeforeGatewayRequest” location. It determines when to leave this location and how long to wait by looking at c<delay and c>=startclk values. After the "BeforeGatewayRequest" location, the request was transferred to the "GatewayRequest" location and directed to the "Gateway" template. If the request is successful, the transaction will be concluded in the "GatewayResponse" and "END" locations.

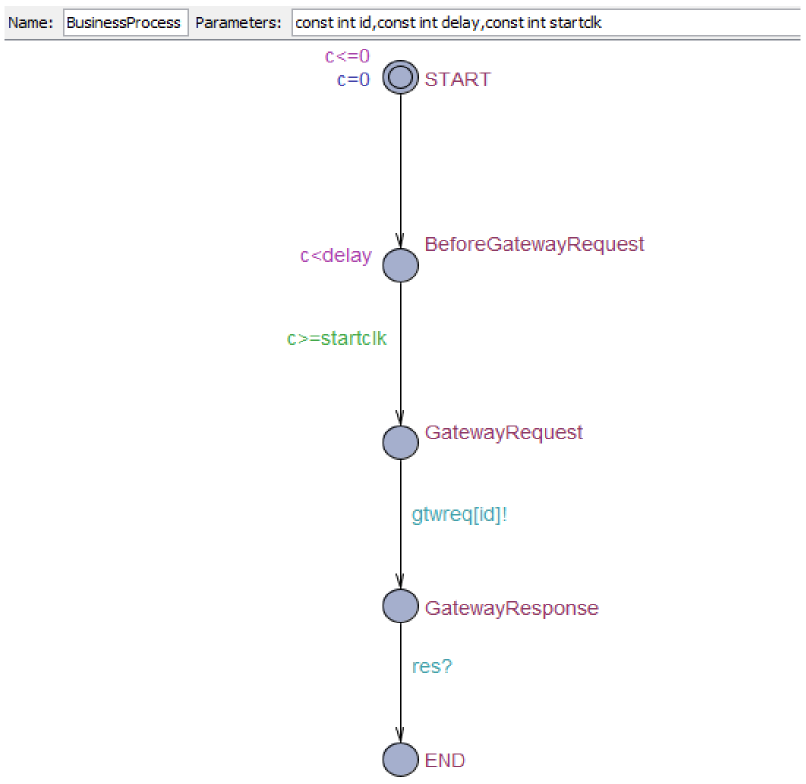
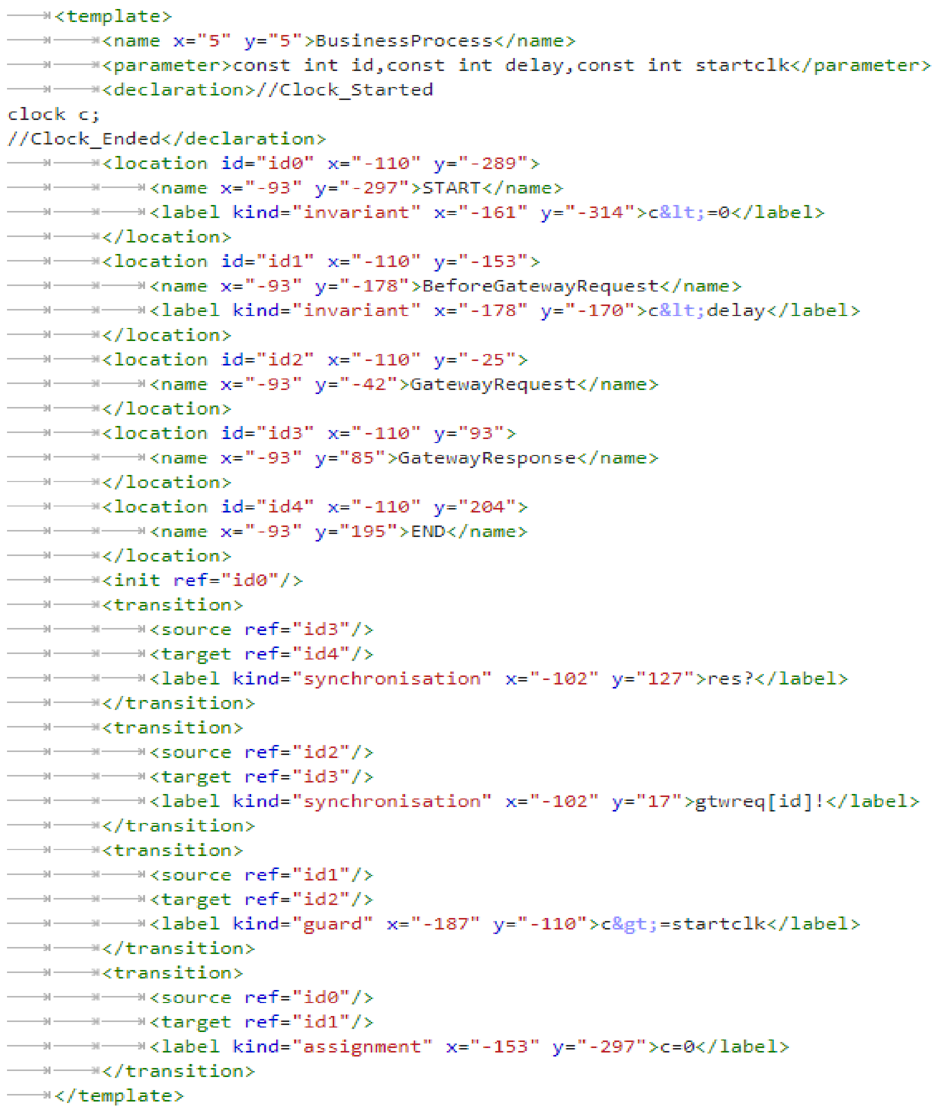

Name: BusinessProcess Parameters: const int id,const int delay,const int startclk

Figure 8. BusinessProcess template structure.

The output of the model created in the Uppaal tool is kept in the .xml file extension. The .xml file extension output of the BusinessProcess model is shown in Figure 9.



```
	<template>
		<name x="5" y="5">BusinessProcess</name>
		<parameter>const int id,const int delay,const int startclk</parameter>
		<declaration>//Clock_Started
clock c;
//Clock_Ended</declaration>
		<location id="id0" x="-110" y="-289">
			<name x="-93" y="-297">START</name>
			<label kind="invariant" x="-161" y="-314">c&lt;=0</label>
		</location>
		<location id="id1" x="-110" y="-153">
			<name x="-93" y="-178">BeforeGatewayRequest</name>
			<label kind="invariant" x="-178" y="-170">c&lt;delay</label>
		</location>
		<location id="id2" x="-110" y="-25">
			<name x="-93" y="-42">GatewayRequest</name>
		</location>
		<location id="id3" x="-110" y="93">
			<name x="-93" y="85">GatewayResponse</name>
		</location>
		<location id="id4" x="-110" y="204">
			<name x="-93" y="195">END</name>
		</location>
		<init ref="id0"/>
		<transition>
			<source ref="id3"/>
			<target ref="id4"/>
			<label kind="synchronisation" x="-102" y="127">res?</label>
		</transition>
		<transition>
			<source ref="id2"/>
			<target ref="id3"/>
			<label kind="synchronisation" x="-102" y="17">gtwreq[id]!</label>
		</transition>
		<transition>
			<source ref="id1"/>
			<target ref="id2"/>
			<label kind="guard" x="-187" y="-110">c&gt;=startclk</label>
		</transition>
		<transition>
			<source ref="id0"/>
			<target ref="id1"/>
			<label kind="assignment" x="-153" y="-297">c=0</label>
		</transition>
	</template>
```

Figure 9. BusinessProcess .xml file structure.

Gateway; This template is our second template (*Fig 11*). It was developed based on parameters to automatically create test scenarios. Gateway template provides the relationship between BusinessProcess template and Microservice template. In microservice applications, gateways forward the request to the relevant microservices. The gateway template starts with the “START” location. At the “START” location, the c(clock) value is updated with the value “0” to evaluate the time correctly. At the same time, we assigned the value "errorrate" to the "temperrorrate" value in order to operate on the "errorrate" value. Before the microservice template goes, the request goes to the “BeforeMicroserviceRequest” location. The conditions we set here are the “delay” and “startclk” values. The “delay” value indicates the maximum time to wait at the location. The “startclk” value specifies how long it takes to move to the next location. In this way, we determine the "Microservice" template arrival interval of the request as [delay, start clk].

Our next location is “CircuitBreakerClose”, this location is a critical location for model. It is a pattern used in gateway structures. The reason why it is used is if the microservice to which the requests will be sent gives an error, we do not send the requests back to the server of the faulty microservice. In this way, we prevent service interruptions, excessive resource usage and blocking. Instead, we pass it through gradual checks and send it to the server. We can group these stages under three headings.

These;

- Closed: At this stage, CircuitBreaker is closed. In this case, since the status in the microservice is less than the threshold value, it forwards the requests to the relevant microservice. This tells us that everything is going well.
- Open: At this stage Circuit Breaker is open. In this case, since the status in the microservice is greater than the threshold value, we do not forward the requests to the relevant microservice. This indicates that there is a problem in the microservice and directs the request according to the determined rules to prevent other problems in the system.
- Half-Open: At this stage, Circuit Breaker is half-open. In this case, it decides whether to send incoming requests to the microservice by looking at the threshold value or the defined rules. In order to switch from half-open state to “Close” state and transmit requests, the "errorrate" value must be less than the "threshold" value, and in order to pass from the “Open” state to the "threshold" value, the "errorrate" value must be greater than the "threshold" value.

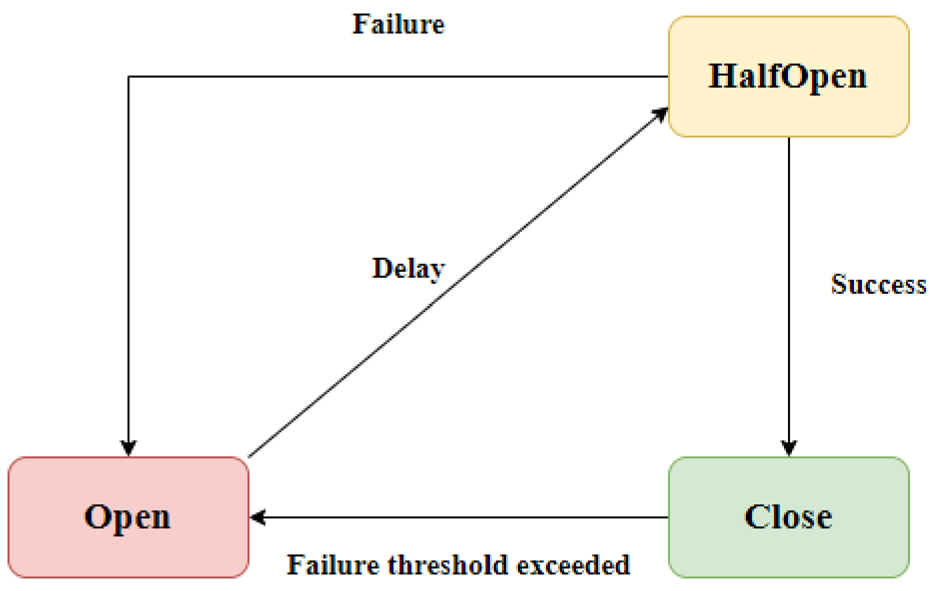


Figure 10. Circuit breaker structure.

Circuit Breaker is a pattern used in projects using gateways. We included this pattern in our model to make it related to real-life. The part where we position the Circuit Breaker is after the "BeforeMicroserviceRequest" location and before the "MicroserviceRequest" location. Thus, we will evaluate the requests sent to microservices.

When the request coming to our model reaches the "CircuitBreakerClose" location, the "threshold" value is compared with the "errorrate" value. If the "threshold" value is higher than the "errorrate" value, the request goes to the microservice. If it is low, the Circuit Breaker structure comes into play and prevents the request from being sent. Thus, it moves to the “CircuitBreakerOpen” section. It keeps the request as open as the "error count" number we define with the parameter here, without sending it to the microservice. After receiving the number of "errorcount" requests, we reach the "CircuitBreakerHalfOpen" location by decreasing the "temperrorrate" value at the rate we defined. Here, if the "threshold" value is higher than the "temperrorrate" value, the request goes to the microservice, or if the "threshold" value is lower than the "temperrorrate" value, the request cannot go to the microservice. The main purpose of Circuit Breaker is to prevent service interruptions, excessive resource usage and blocking. Therefore, in order to prevent these situations in the modeled system, different Circuit Breaker scenarios can be developed by taking into account the priority criteria of the application. For example, you can switch between the Circuit Breaker stages, close, open and half-open, depending on the number of requests, the time period on the servers or the response codes received by grouping the response codes. We created this scenario by determining a rate by which we would reduce the number of failed requests (errorcount) and the "errorrate" value along with it.

We also studied error rate, threshold, error count and rate values parametrically to create scenarios during the testing processes.

The structure of the gateway template can be seen in Figure 11.

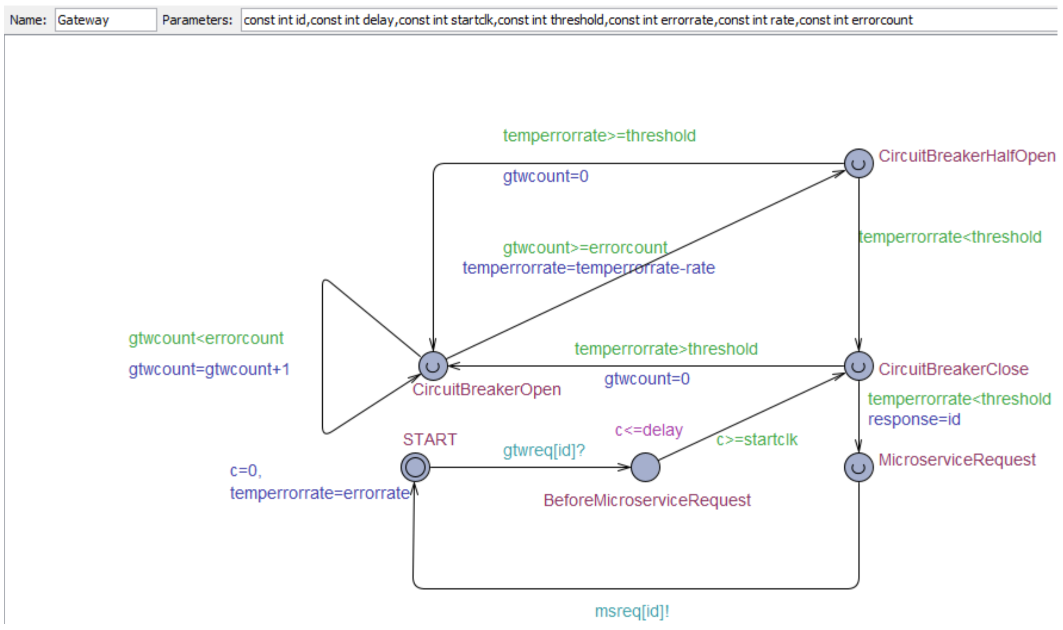
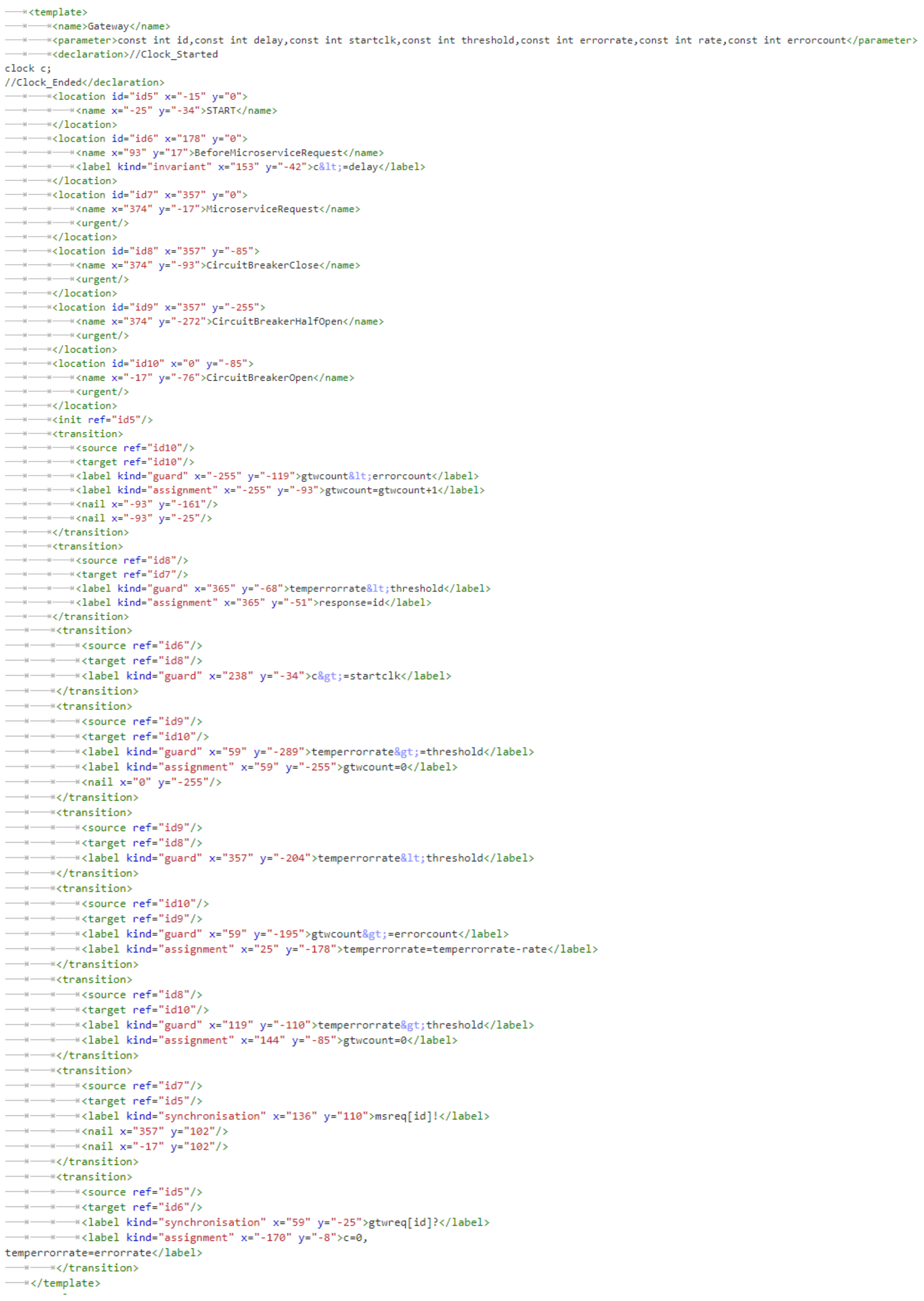


Figure 11. Gateway template structure.

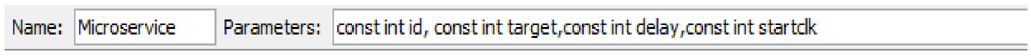
The output of the Gateway template we created is included in our model file as .xml file format (*Fig. 12*).



```xml
<template>
    <name>Gateway</name>
    <parameter>const int id,const int delay,const int startclk,const int threshold,const int errorrate,const int rate,const int errorcount</parameter>
    <declaration>//Clock_Started
clock c;
//Clock_Ended</declaration>
    <location id="id5" x="-15" y="0">
        <name x="-25" y="-34">START</name>
    </location>
    <location id="id6" x="178" y="0">
        <name x="93" y="17">BeforeMicroserviceRequest</name>
        <label kind="invariant" x="153" y="-42">c&lt;=delay</label>
    </location>
    <location id="id7" x="357" y="0">
        <name x="374" y="-17">MicroserviceRequest</name>
        <urgent/>
    </location>
    <location id="id8" x="357" y="-85">
        <name x="374" y="-93">CircuitBreakerClose</name>
        <urgent/>
    </location>
    <location id="id9" x="357" y="-255">
        <name x="374" y="-272">CircuitBreakerHalfOpen</name>
        <urgent/>
    </location>
    <location id="id10" x="0" y="-85">
        <name x="-17" y="-76">CircuitBreakerOpen</name>
        <urgent/>
    </location>
    <init ref="id5"/>
    <transition>
        <source ref="id10"/>
        <target ref="id10"/>
        <label kind="guard" x="-255" y="-119">gtwcount&lt;errorcount</label>
        <label kind="assignment" x="-255" y="-93">gtwcount=gtwcount+1</label>
        <nail x="-93" y="-161"/>
        <nail x="-93" y="-25"/>
    </transition>
    <transition>
        <source ref="id8"/>
        <target ref="id7"/>
        <label kind="guard" x="365" y="-68">temperrorrate&lt;threshold</label>
        <label kind="assignment" x="365" y="-51">response=id</label>
    </transition>
    <transition>
        <source ref="id6"/>
        <target ref="id8"/>
        <label kind="guard" x="238" y="-34">c&gt;=startclk</label>
    </transition>
    <transition>
        <source ref="id9"/>
        <target ref="id10"/>
        <label kind="guard" x="59" y="-289">temperrorrate&gt;=threshold</label>
        <label kind="assignment" x="59" y="-255">gtwcount=0</label>
        <nail x="0" y="-255"/>
    </transition>
    <transition>
        <source ref="id9"/>
        <target ref="id8"/>
        <label kind="guard" x="357" y="-204">temperrorrate&lt;threshold</label>
    </transition>
    <transition>
        <source ref="id10"/>
        <target ref="id9"/>
        <label kind="guard" x="59" y="-195">gtwcount&gt;=errorcount</label>
        <label kind="assignment" x="25" y="-178">temperrorrate=temperrorrate-rate</label>
    </transition>
    <transition>
        <source ref="id8"/>
        <target ref="id10"/>
        <label kind="guard" x="119" y="-110">temperrorrate&gt;threshold</label>
        <label kind="assignment" x="144" y="-85">gtwcount=0</label>
    </transition>
    <transition>
        <source ref="id7"/>
        <target ref="id5"/>
        <label kind="synchronisation" x="136" y="110">msreq[id]!</label>
        <nail x="357" y="102"/>
        <nail x="-17" y="102"/>
    </transition>
    <transition>
        <source ref="id5"/>
        <target ref="id6"/>
        <label kind="synchronisation" x="59" y="-25">gtwreq[id]?</label>
        <label kind="assignment" x="-170" y="-8">c=0,
temperrorrate=errorrate</label>
    </transition>
</template>
```

Figure 12. Gateway Template .xml Dosya Yapısı

Microservice; This template is our third template (*Fig. 13*). It was developed based on parameters to automatically create test cases. It starts with the "START" location. We aimed to make accurate measurements on the time side by updating the c(clock) value to "0" in the first location. It continues with "BeforeMicroserviceResponse". At this location, we split into two roads. If our microservice has a dependency on another microservice, we direct it there with the "BeforeMSChannel" location. We continue the process by taking into account the "delay" and "startclk" values in the microservice we direct. If there is no dependency on any microservice, we perform the response operation in the "delay" and "startclk" value ranges ([delay, startclk]) that we have given in the parameters and direct it to the "BusinessProcess" template. With the microservice template, we can define more than one microservice and associate them with each other.



Name: Microservice Parameters: const int id, const int target,const int delay,const int startclk

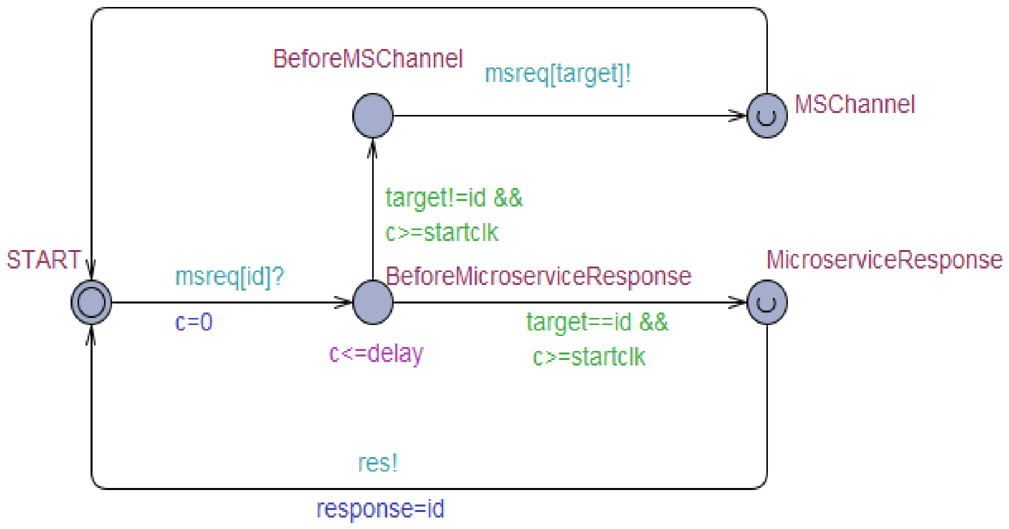
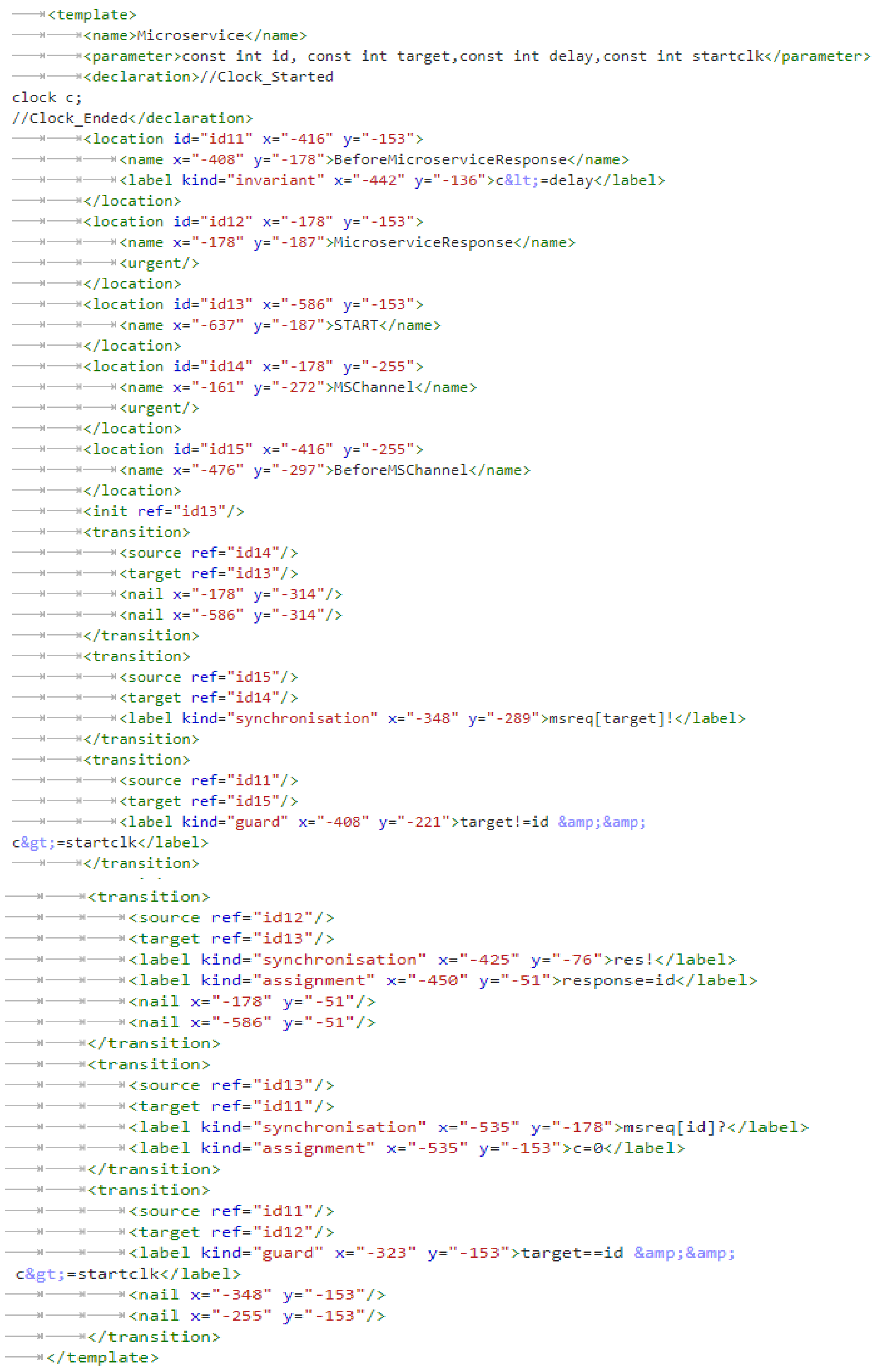


Figure 13. Microservice template structure.

The location and transaction links of the microservice template we created are included in the .xml file extension (*Fig. 14*).



```xml
	<template>
		<name>Microservice</name>
		<parameter>const int id, const int target,const int delay,const int startclk</parameter>
		<declaration>//Clock_Started
clock c;
//Clock_Ended</declaration>
		<location id="id11" x="-416" y="-153">
			<name x="-408" y="-178">BeforeMicroserviceResponse</name>
			<label kind="invariant" x="-442" y="-136">c&lt;=delay</label>
		</location>
		<location id="id12" x="-178" y="-153">
			<name x="-178" y="-187">MicroserviceResponse</name>
			<urgent/>
		</location>
		<location id="id13" x="-586" y="-153">
			<name x="-637" y="-187">START</name>
		</location>
		<location id="id14" x="-178" y="-255">
			<name x="-161" y="-272">MSChannel</name>
			<urgent/>
		</location>
		<location id="id15" x="-416" y="-255">
			<name x="-476" y="-297">BeforeMSChannel</name>
		</location>
		<init ref="id13"/>
		<transition>
			<source ref="id14"/>
			<target ref="id13"/>
			<nail x="-178" y="-314"/>
			<nail x="-586" y="-314"/>
		</transition>
		<transition>
			<source ref="id15"/>
			<target ref="id14"/>
			<label kind="synchronisation" x="-348" y="-289">msreq[target]!</label>
		</transition>
		<transition>
			<source ref="id11"/>
			<target ref="id15"/>
			<label kind="guard" x="-408" y="-221">target!=id &amp;&amp;
c&gt;=startclk</label>
		</transition>
		<transition>
			<source ref="id12"/>
			<target ref="id13"/>
			<label kind="synchronisation" x="-425" y="-76">res!</label>
			<label kind="assignment" x="-450" y="-51">response=id</label>
			<nail x="-178" y="-51"/>
			<nail x="-586" y="-51"/>
		</transition>
		<transition>
			<source ref="id13"/>
			<target ref="id11"/>
			<label kind="synchronisation" x="-535" y="-178">msreq[id]?</label>
			<label kind="assignment" x="-535" y="-153">c=0</label>
		</transition>
		<transition>
			<source ref="id11"/>
			<target ref="id12"/>
			<label kind="guard" x="-323" y="-153">target==id &amp;&amp;
 c&gt;=startclk</label>
			<nail x="-348" y="-153"/>
			<nail x="-255" y="-153"/>
		</transition>
	</template>
```

Figure 14. Microservice template .xml file structure

MicroserviceFail; This template is our fourth template (*Fig. 17*). It is exactly the same as the Microservice template. We develop models via templates. Therefore, if we make a change in the location or transaction processes of a template, it affects all templates. This situation affects our processes in a way we do not want. For example, if we have two microservices, there are cases where we break the relationships between the locations to create test-case scenarios. But we only want to break the relationship between a microservice and the locations. That's why we created the “MicroserviceFail” template. As seen in Figure 15, the transaction values of both microservices are broken. But we only want to disconnect one microservice. For this reason, in our model, we create the microservice from which we want to break the transaction with the "MicroserviceFail" template. We create the microservice whose transaction we do not want to break with the "Microservice" template. In this way, we do not break the transaction in all microservices, but we break the transaction in the ones we will test (*Fig. 16*).

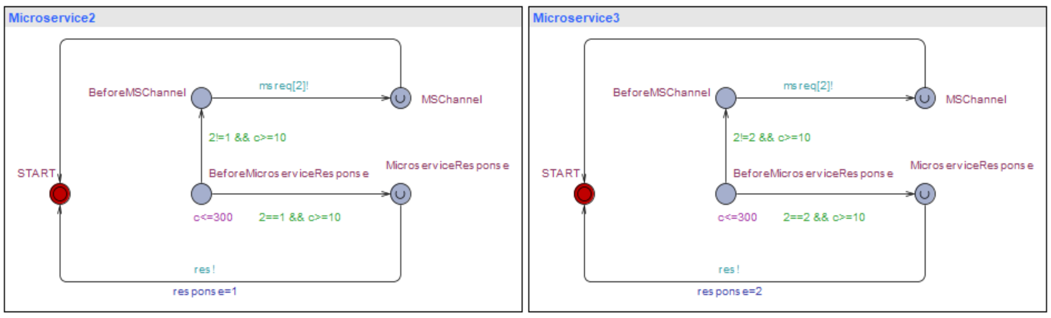


Figure 15. Transaction between “START” and “Before Microservice Response” has been broken.

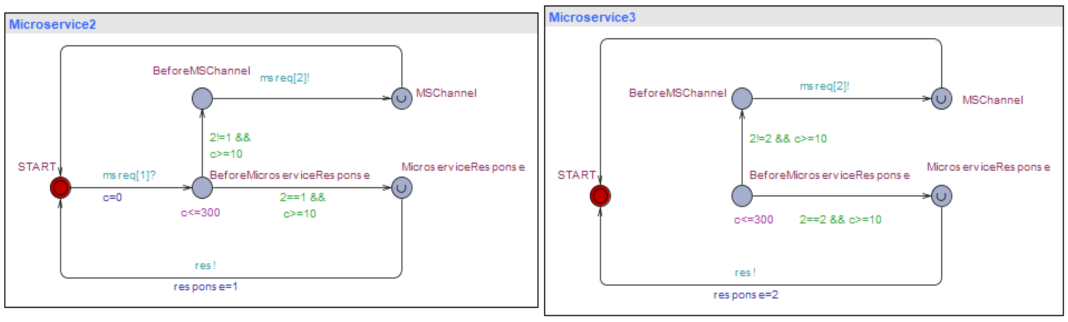
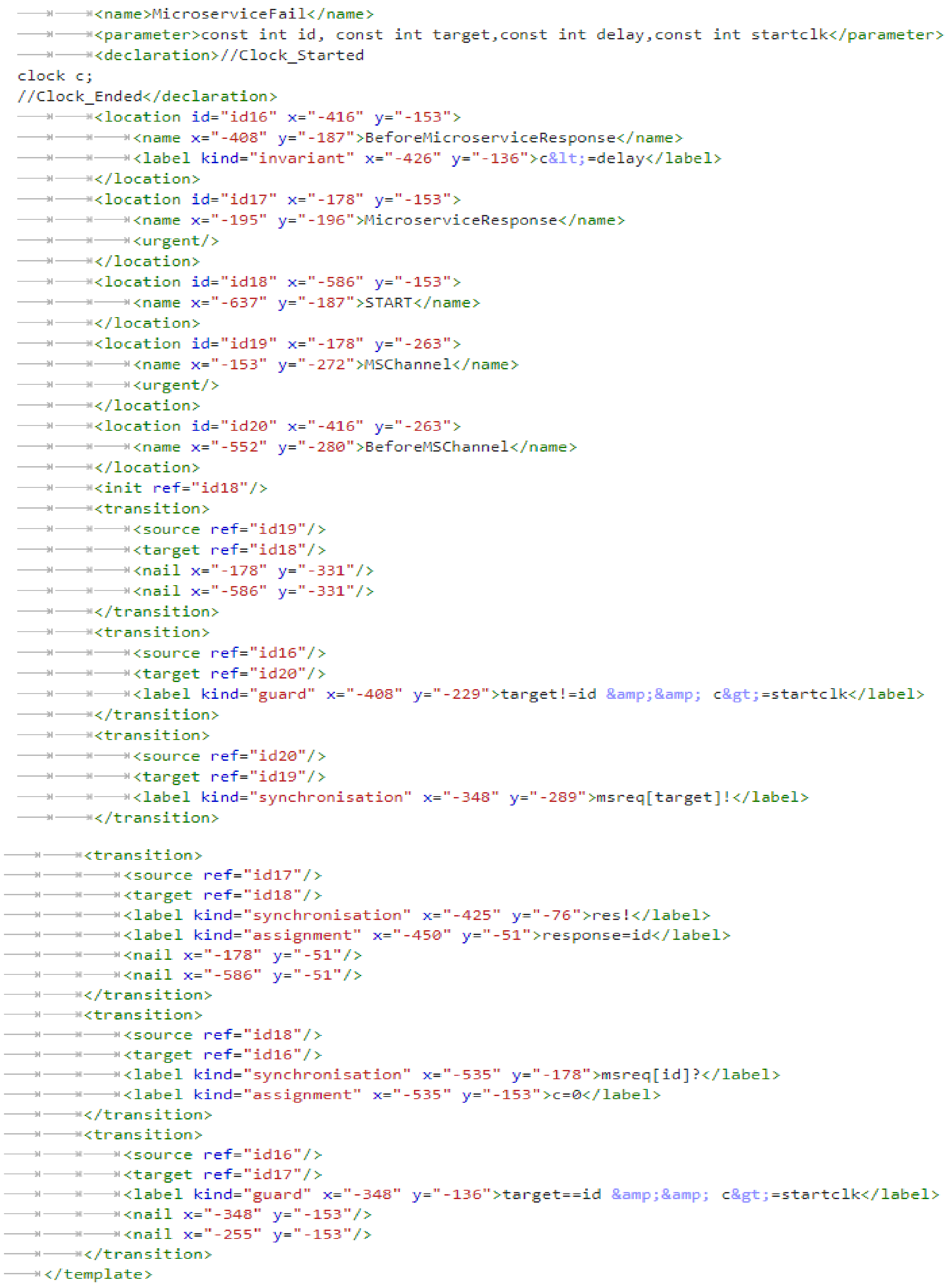


Figure 16. Only in Microservice3, the tarasaction between “START” and “BeforeMicroserviceResponse” is broken.



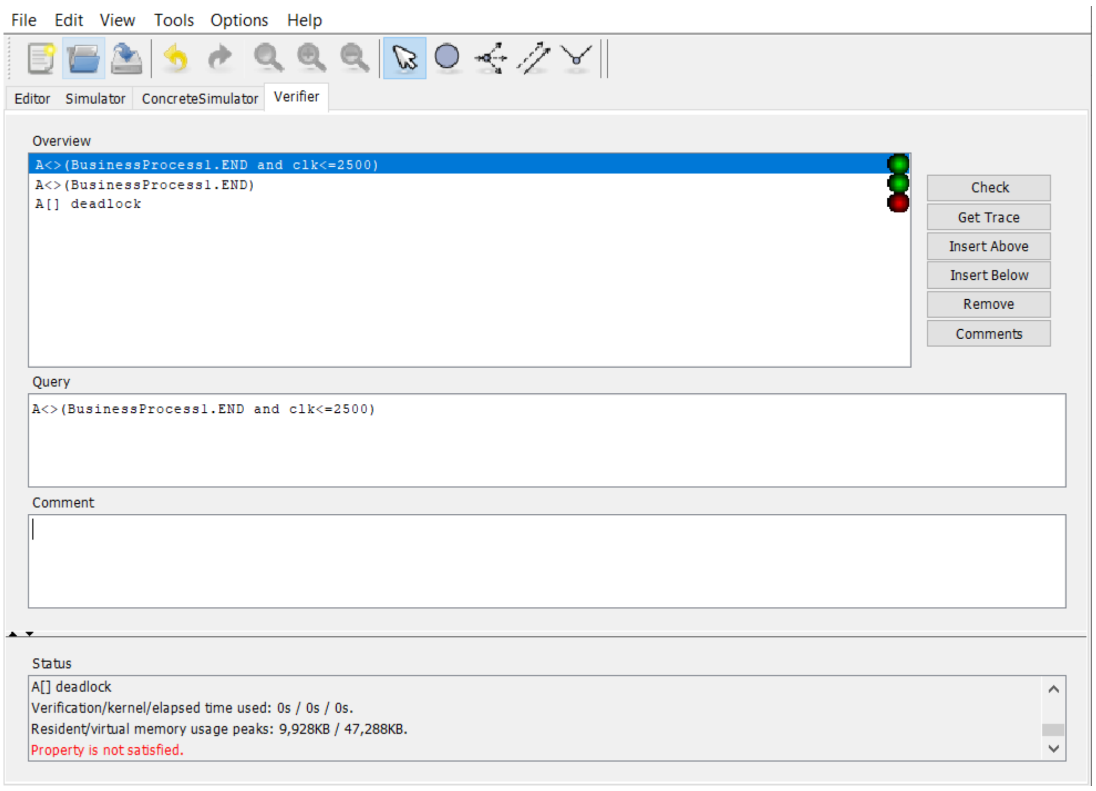
```
        <name>MicroserviceFail</name>
        <parameter>const int id, const int target,const int delay,const int startclk</parameter>
        <declaration>//Clock_Started
clock c;
//Clock_Ended</declaration>
        <location id="id16" x="-416" y="-153">
            <name x="-408" y="-187">BeforeMicroserviceResponse</name>
            <label kind="invariant" x="-426" y="-136">c&lt;=delay</label>
        </location>
        <location id="id17" x="-178" y="-153">
            <name x="-195" y="-196">MicroserviceResponse</name>
            <urgent/>
        </location>
        <location id="id18" x="-586" y="-153">
            <name x="-637" y="-187">START</name>
        </location>
        <location id="id19" x="-178" y="-263">
            <name x="-153" y="-272">MSChannel</name>
            <urgent/>
        </location>
        <location id="id20" x="-416" y="-263">
            <name x="-552" y="-280">BeforeMSChannel</name>
        </location>
        <init ref="id18"/>
        <transition>
            <source ref="id19"/>
            <target ref="id18"/>
            <nail x="-178" y="-331"/>
            <nail x="-586" y="-331"/>
        </transition>
        <transition>
            <source ref="id16"/>
            <target ref="id20"/>
            <label kind="guard" x="-408" y="-229">target!=id &amp;&amp; c&gt;=startclk</label>
        </transition>
        <transition>
            <source ref="id20"/>
            <target ref="id19"/>
            <label kind="synchronisation" x="-348" y="-289">msreq[target]!</label>
        </transition>

        <transition>
            <source ref="id17"/>
            <target ref="id18"/>
            <label kind="synchronisation" x="-425" y="-76">res!</label>
            <label kind="assignment" x="-450" y="-51">response=id</label>
            <nail x="-178" y="-51"/>
            <nail x="-586" y="-51"/>
        </transition>
        <transition>
            <source ref="id18"/>
            <target ref="id16"/>
            <label kind="synchronisation" x="-535" y="-178">msreq[id]?</label>
            <label kind="assignment" x="-535" y="-153">c=0</label>
        </transition>
        <transition>
            <source ref="id16"/>
            <target ref="id17"/>
            <label kind="guard" x="-348" y="-136">target==id &amp;&amp; c&gt;=startclk</label>
            <nail x="-348" y="-153"/>
            <nail x="-255" y="-153"/>
        </transition>
    </template>
```

Figure 17. MicroserviceFail template .xml file structure.

4.2. Verifying the Model

We need to verify the main model we created. Because we need to make sure that our main scenario works correctly so that the other mutant scenarios we create can work correctly. Therefore, we used the Uppaal tool to verify our model. We can test the queries we wrote in the "Verifier" tab of the Uppaal tool (*Fig. 18*).



```
A<>(BusinessProcess1.END and clk<=2500)
A<>(BusinessProcess1.END)
A[] deadlock
```

```
A<>(BusinessProcess1.END and clk<=2500)
```

```
A[] deadlock
Verification/kernel/elapsed time used: 0s / 0s / 0s.
Resident/virtual memory usage peaks: 9,928KB / 47,288KB.
Property is not satisfied.
```

Figure 18. Uppaal verifier tab

We wrote queries for our main model. We will check the accuracy of our main model according to the queries. There are certain rules for writing queries in Uppaal. We can show the following queries as examples of these rules;

- **A<>:** The situation we wrote in the expression indicates that the model will take at least one journey in time and only in one time period.
- **A[]:** The situation or feature in which we write the expression indicates that the model will always take place at least one journey over time.

- **E<>:** The situation in which we wrote the expression indicates that all journeys in time in the model will take place in only one time period.
- **E[]:** The situation in which we wrote the expression indicates that all journeys will always take place over time in the model.

We prepared the following queries to reverse our model.

Prepared queries;

- A[] deadlock
- A<>(BusinessProcess1.END)
- A<>(BusinessProcess1.END and clk<=2500)

A[] deadlock => The part of the model that the query controls is that it enters an infinite loop at every time period in at least one trip. We do not expect this query to be validated. Because if it is verified, it means there is a problem with the model we prepared. This issue means that the request ends up going into an infinite loop before completing successfully.

A<>(BusinessProcess1.END) => The part the query verifies in the model is whether the location of the BusinessProcess1 object is "END" in just one time period of at least one trip. In this case, requesting when the request comes means giving us a successful response. In this case, this is what we want and expect for our model.

A<>(BusinessProcess1.END and clk<=2500) => The part that the query verifies in the model is that at least one journey must occur only in a time period if the location of the BusinessProcess1 object is “END” and the “clk” value is less than or equal to 2500. In this case, it means that the request responds to us within "2500 milliseconds" when the request starts. In this case, this is the situation we want and expect for our model.

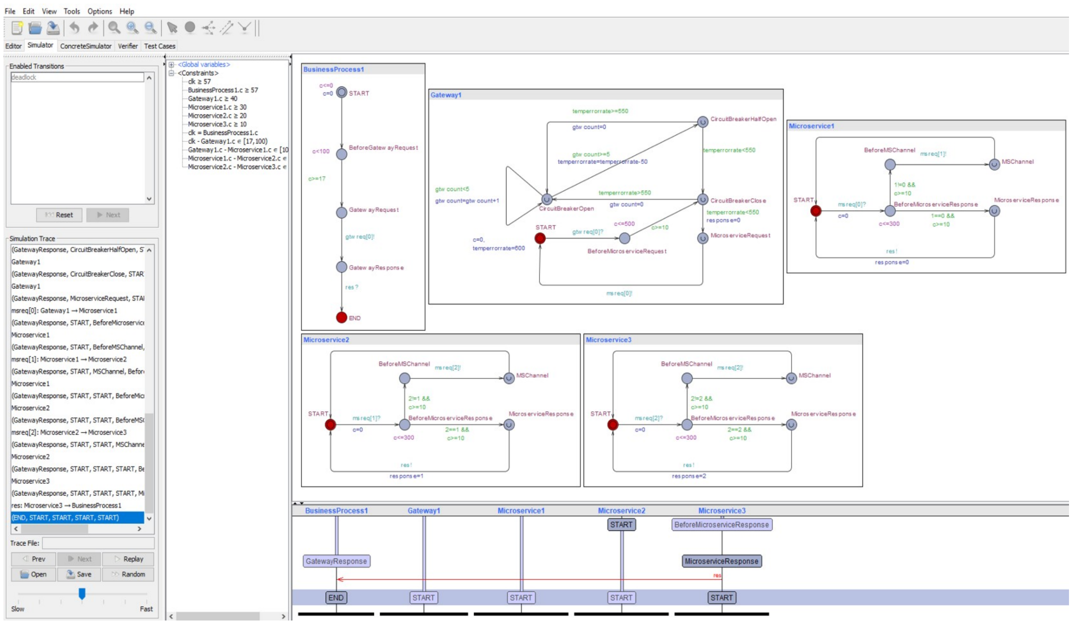
As a result, in order to know that the request to our model has been successful, the BusinessProcess1 state must be in the "END" location, and when it is in this location, the "clk" value must be less than or equal to 2500. Thus, the verify result we get from this query shows that the model we created works correctly (*Fig. 18*).

4.3. Simulating the Model

We examined the model in detail by simulating the model we created (*Fig. 19*).The Simulator tab consists of five sections.

These;

- Enabled Transitions,
- Simulation Trace,
- Variables and Constraints,
- Simulation Trace Screen
- Simulation Screen

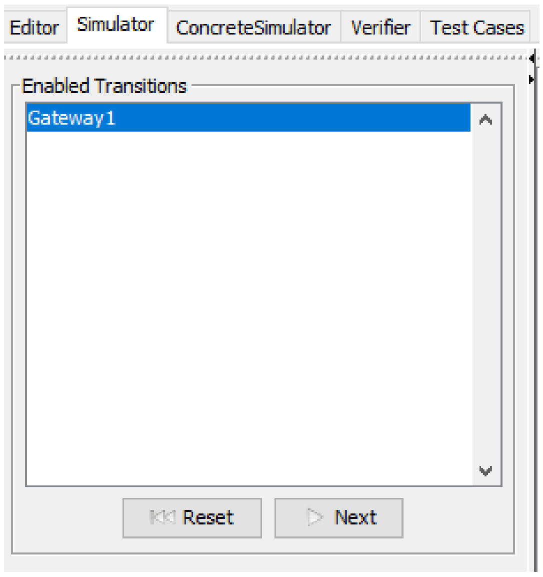


```
<Global variables>
<Constraints>
  dk ≥ 57
  BusinessProcess1.c ≥ 57
  Gateway1.c ≥ 40
  Microservice1.c ≥ 30
  Microservice2.c ≥ 20
  Microservice3.c ≥ 10
  dk = BusinessProcess1.c
  dk - Gateway1.c ∊ [17,100)
  Gateway1.c - Microservice1.c ∊ [10
  Microservice1.c - Microservice2.c ∊
  Microservice2.c - Microservice3.c ∊
```

```
(GatewayResponse, CircuitBreakerHalfOpen, S
Gateway1
(GatewayResponse, CircuitBreakerClose, STAR
Gateway1
(GatewayResponse, MicroserviceRequest, STA
msreq[0]: Gateway1 → Microservice1
(GatewayResponse, START, BeforeMicroservic
Microservice1
(GatewayResponse, START, BeforeMSChannel,
msreq[1]: Microservice1 → Microservice2
(GatewayResponse, START, MSChannel, Befor
Microservice1
(GatewayResponse, START, START, BeforeMic
Microservice2
(GatewayResponse, START, START, BeforeMS
msreq[2]: Microservice2 → Microservice3
(GatewayResponse, START, START, MSChanne
Microservice2
(GatewayResponse, START, START, START, Be
Microservice3
(GatewayResponse, START, START, START, Mi
res: Microservice3 → BusinessProcess1
(END, START, START, START, START)
```

Figure 19. Uppaal simulator tab.

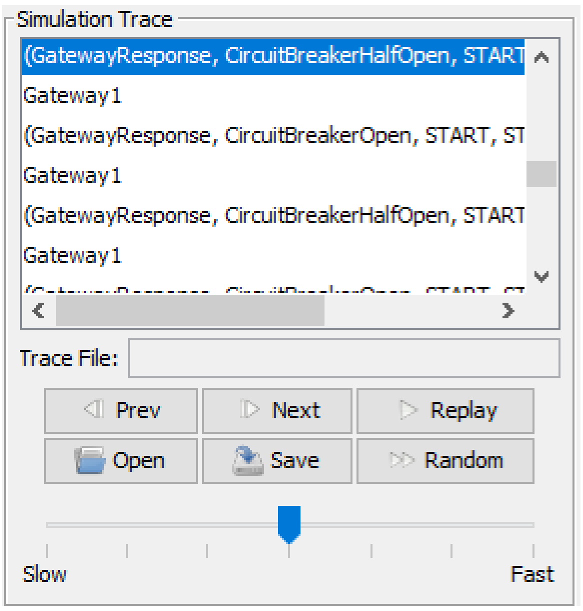
Enabled Transitions: This area allows us to proceed step by step in the simulation. It shows the next step or steps that the model can go through (*Fig. 20*).



Gateway1

Figure 20. Enabled transitions.

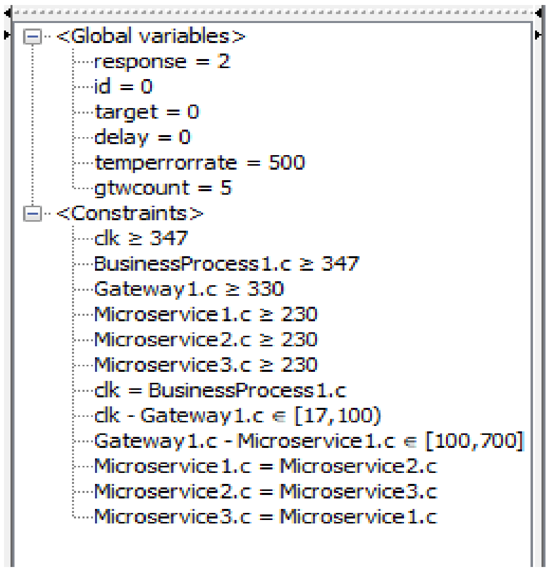
Simulation Trace: This area shows all steps traced in the simulation. By navigating through these steps, you gain information about variables, constraints and transitions. Additionally, if we have a trace file, we can read it or save the existing trace file (*Fig. 21*).



(GatewayResponse, CircuitBreakerHalfOpen, START
Gateway1
(GatewayResponse, CircuitBreakerOpen, START, ST
Gateway1
(GatewayResponse, CircuitBreakerHalfOpen, START
Gateway1

Figure 21. Simulation trace.

Variables and Constraints: This area holds the variables and constraints we defined in the simulation. Thus, when we run the simulations step by step, we can see the values of variables and constraints changing in the stages (*Fig. 22*).



```
<Global variables>
    response = 2
    id = 0
    target = 0
    delay = 0
    temperrorrate = 500
    gtwcount = 5
<Constraints>
    clk ≥ 347
    BusinessProcess1.c ≥ 347
    Gateway1.c ≥ 330
    Microservice1.c ≥ 230
    Microservice2.c ≥ 230
    Microservice3.c ≥ 230
    clk = BusinessProcess1.c
    clk - Gateway1.c ∈ [17,100)
    Gateway1.c - Microservice1.c ∈ [100,700]
    Microservice1.c = Microservice2.c
    Microservice2.c = Microservice3.c
    Microservice3.c = Microservice1.c
```

Figure 22. Variables and constraints.

Simulation Trace Screen: This area visualizes the trace of the model we created. In this way, we can see which transaction we went from which template (*Fig. 23*).

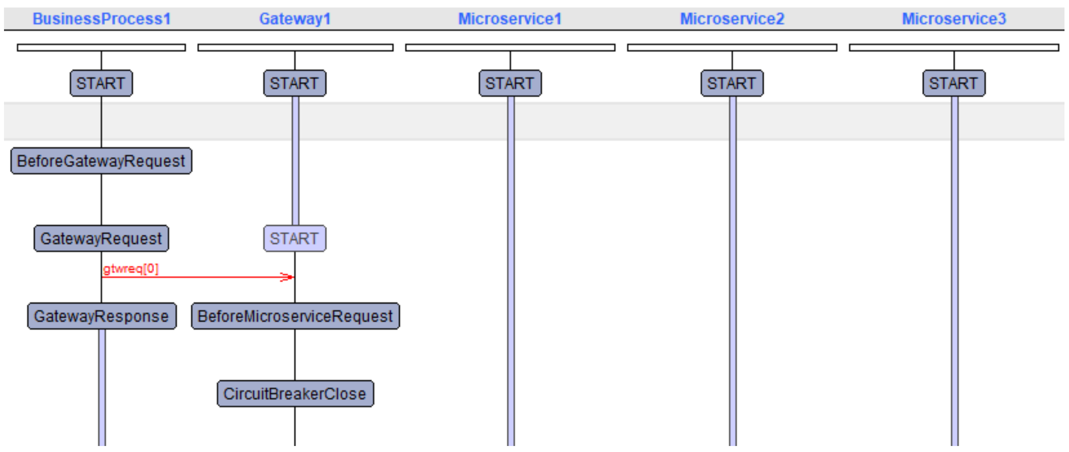


Figure 23. Simulation trace screen

Simulation Screen: This area visually presents us with the simulation of the model we created. Thus, we see the entire simulated model and examine the flow through the visual (*Fig. 24*).

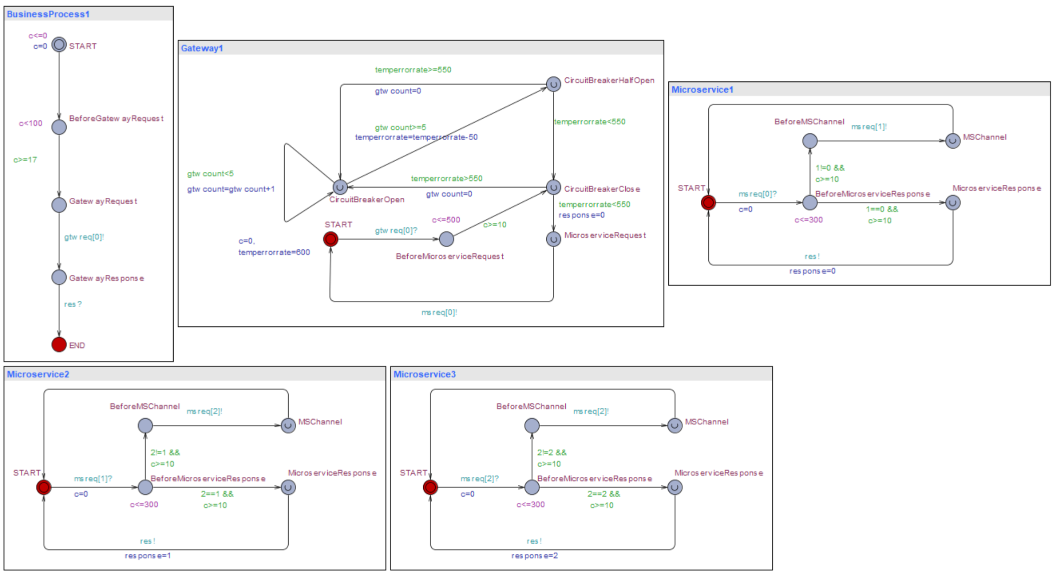


Figure 24. Simulation screen.

4.4. Creating Test Cases

We chose the mutation method to create test cases of the model we created. We used our previously prepared model to apply the mutation method. Because the model we prepared resembles the structures of applications used in real life and is designed with the microservice architecture we want to test. The situations we call transitions in our model represent the communication between services or functions in the structures we use in real life. We used transitions in the model, service connections and transitions in critical situations for us. In this way, we aimed to quickly detect which service or critical situation had an error in our testing processes. Therefore, if we delete the transition points, we have mutated our model that works correctly. Thus, we performed tests for our application and module by creating test cases based on the transitions we deleted. Thus, we tested how it could affect our system in case of any communication breakdown. Another example of mutation is updating guard and transition values. In this case, these values include the transitions of the communication parts in our application at certain time intervals and under certain conditions. In other words, we mutated by updating the conditions required to move from one state to another. Thus, we tested how our application would react according to the arrival times of the data in the services and the conditions that may occur. Using these two mutation examples in our model, we created test cases for scenarios that may occur in application by mutating our model in desired time, value ranges and possible conditions. Based on the results of the test cases we create, we can learn how our application will react, and if it gives an error, we can determine where it originates from.

We developed functions using the python programming language to automate the processes performed while creating test cases of the model we created. Among the functions we developed, we wrote functions that mutate our model and turn into test cases. We chose two ways to mutate our model.

These;

- Deleting transitions,
- Updating guard and transition values

4.4.1 Deleting transitions

In order to create test cases of the model we created, we had to create scenarios by mutating. Therefore, we started by deleting transitions to create mutations. We export out the .xml file of the model we created in Uppaal. We first parsed the xml file using the python programming language. Then, we navigated through each template, deleted the transitions and saved the model.

The point we pay attention to here is that in each scenario we create, we delete and save only one transition of our main model. The transaction in the red circled section of Microservice1 shown in Figure 25 is an example of this situation. There are three microservices in the study. For this reason, this study was conducted for all three microservices. As seen in Mikroservice2 and Mikroservice3, the specified transaction has not been deleted. Because in the example shown, only Microservice1 is derived from the “MicroserviceFail” template, while the others are derived from the “Microservice” template.

As a result, we derived whichever microservice we wanted to test from the "MicroserviceFail" template. Figure 25 shows the scenario derived from Microservice1. However, this work has also been done in other microservices, respectively.

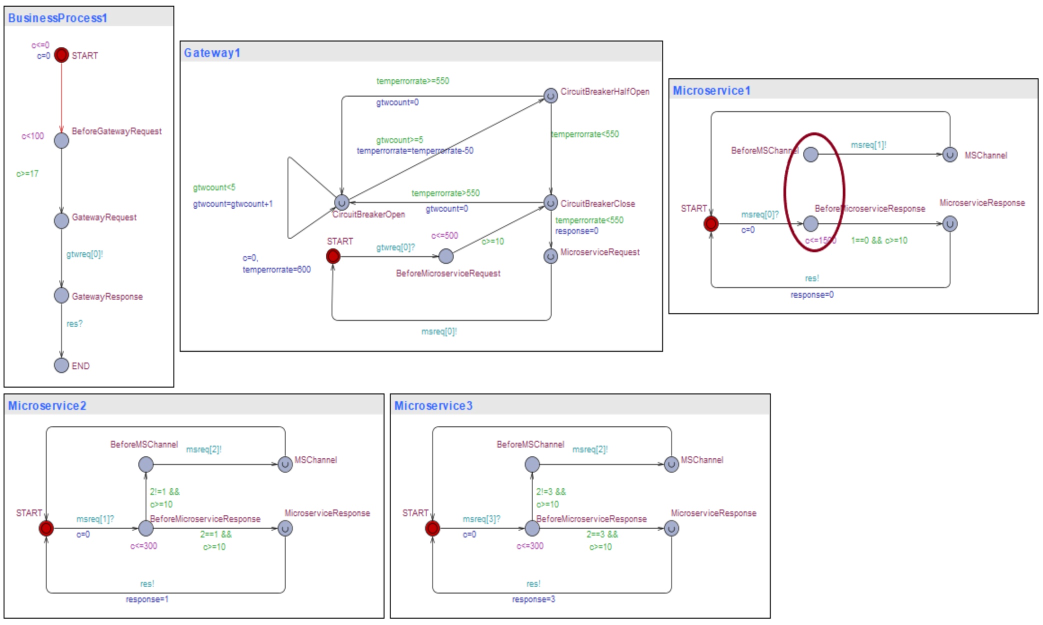
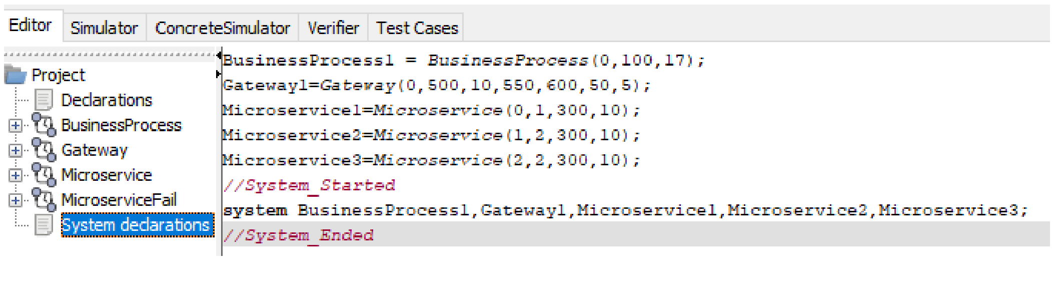


Figure 25. Transitions deleted model.

4.4.2 Updating Guard and Transition Values

We mutated our models by updating the "Guard" and "Transitions" values of the model we created. While creating our model, we created it depending on the parameters. Because we wanted to automatically generate new scenarios using values within a certain range, instead of trying test cases manually. In this way, our test cases would be created.



```
BusinessProcess1 = BusinessProcess(0,100,17);
Gateway1=Gateway(0,500,10,550,600,50,5);
Microservice1=Microservice(0,1,300,10);
Microservice2=Microservice(1,2,300,10);
Microservice3=Microservice(2,2,300,10);
//System_Started
system BusinessProcess1,Gateway1,Microservice1,Microservice2,Microservice3;
//System_Ended
```

Figure 26. Model sytem declarations section.

We updated the "System Declarations" section of the model to create mutation models. This is the part that controls the templates of our model. Since we create templates according to parameters, the changes we make in the "System Declarations" section will affect the "Guard", "Transition" and structure of our model.

What's in the System Declarations section;

- BusinessProcess1 = BusinessProcess(0,100,17);
- Gateway1=Gateway(0,500,10,550,600,50,5);
- Microservice1=MicroserviceFail(0,1,300,10);
- Microservice2=Microservice(1,2,300,10);
- Microservice3=Microservice(2,0,300,10);
- system BusinessProcess1, Gateway1, Microservice1, Microservice2, Microservice3;

BusinessProcess1=BusinessProcess(0,100,17) => In this section, we create an object named “BusinessProcess1” from the “BusinessProcess” template (*Fig. 27*).

The values taken by BusinessProcess1 are respectively;

- 0 = It is the "id" number of the object.
- 100 = It is the "delay" value of the object. It was defined as the maximum waiting time in the defined location.
- 17 = It is the “startclk” value of the object. The transition in which it is defined is also defined as the minimum waiting time.

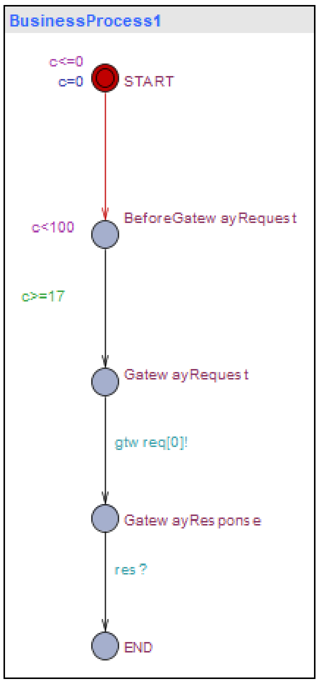


Figure 27. BusinessProcess1 object.

Gateway1 = Gateway(0,500,10,550,600,50,5) => In this part, we create an object named "Gateway1" from the "Gateway" template (*Fig. 28*).

The values taken by Gateway1 are respectively;

- 0 = It is the "id" number of the object.
- 500 = It is the "delay" value of the object. It was defined as the maximum waiting time in the defined location.
- 10 = It is the "startclk" value of the object. The transition in which it is defined is also defined as the minimum waiting time.
- 550 = It is the "threshold" value of the object. It determines the status of the Circuit Breaker in the Gateway together with the "error rate".
- 600 = It is the "errorrate" value of the object. It determines the Circuit Breaker status in the Gateway together with the "threshold".
- 50 = It is the "rate" value of the object. The "errorcount" value of the Circuit Breaker in the Gateway counts as the value we specify when changing from open to half-open. Then we subtract the "rate" value from the "errorrate" value. Thus, the system determines which state it will enter.
- 5 = It is the "errorcount" value of the object. It represents how many requests Circuit Breaker expects in the open state.

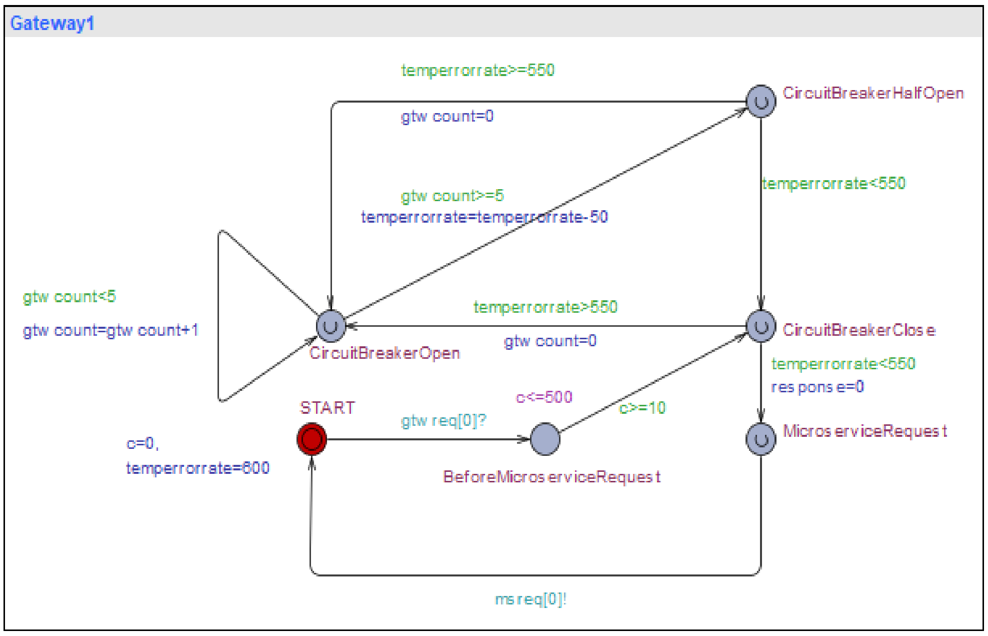


Figure 28. Gateway1 object.

Microservice1 = MicroserviceFail(0,1,300,10) => In this section, we create an object named "Microservice1" from the "MicroserviceFail" template (*Fig. 29*).

The values taken by Microservice1 are respectively;

- 0 = It is the "id" number of the object.
- 1 = It is the "id" number of the microservice it will go to.
- 300 = It is the "delay" value of the object. It was defined as the maximum waiting time in the defined location.
- 10 = It is the "startclk" value of the object. The transition in which it is defined is also defined as the minimum waiting time.

Microservice2=Microservice(1,2,300,10) => Bu kısımda "Microservice" template den "Microservice2" adında bir nesne üretiyoruz (*Fig. 29*).

The values taken by Microservice2 are respectively;

- 1 = It is the "id" number of the object.
- 2 = It is the "id" number of the microservice it will go to.
- 300 = It is the "delay" value of the object. It was defined as the maximum waiting time in the defined location.
- 10 = It is the "startclk" value of the object. The transition in which it is defined is defined as the minimum waiting time.

Microservice3=Microservice(2,0,300,10) => In this section, we create an object named "Microservice3" from the "Microservice" template (*Fig. 29*).

The values taken by Microservice3 are respectively;

- 2 = It is the "id" number of the object.
- 0 = It is the "id" number of the microservice it will go to.
- 300 = It is the "delay" value of the object. It was defined as the maximum waiting time in the defined location.
- 10 = It is the "startclk" value of the object. The transition in which it is defined is also defined as the minimum waiting time.

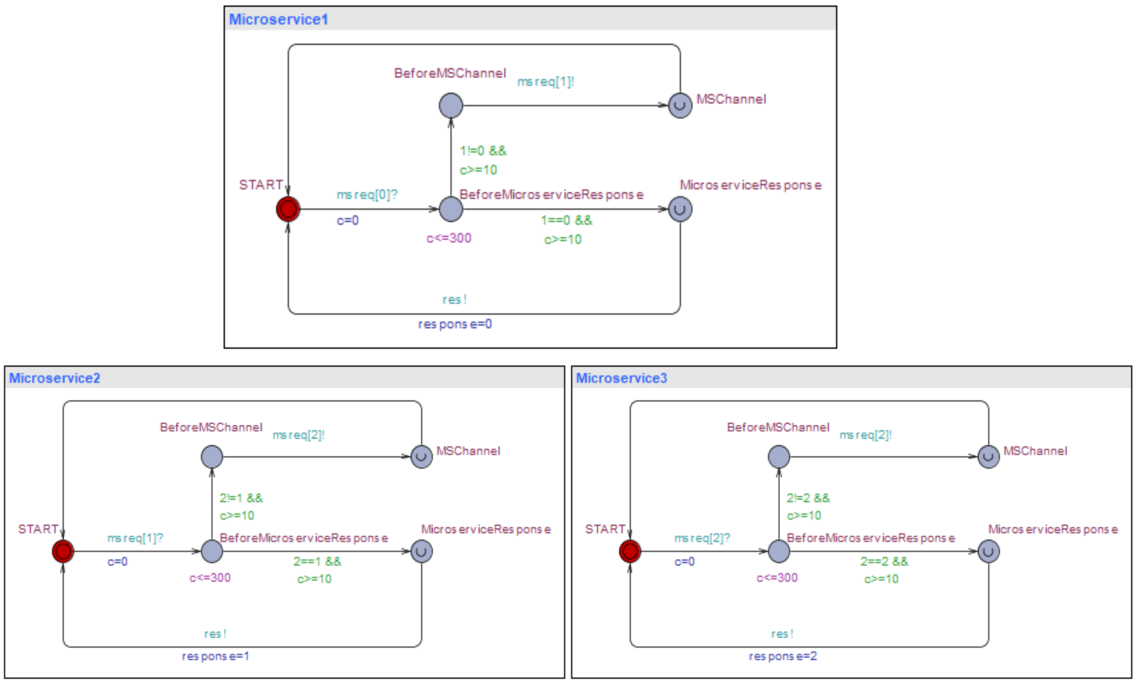


Figure 29. Microservice1, Microservice2 and Microservice3 object.

4.5. Cleaning Test Scenarios

We carried out a cleaning effort in the scenarios in case similar ones to the scenarios we created occurred. The reason why similar scenarios occur is that the microservices we derived from the "MicroserviceFail" theme are common objects. For example, Figure 31 is an example of a fail scenario for Microservice1, and Figure 31 is an example of a fail scenario for Microservice2. However, the BusinessProcess1 object, which is common to both, does not have a transition in the red circle. Therefore, this scenario needs to be deduplicated.

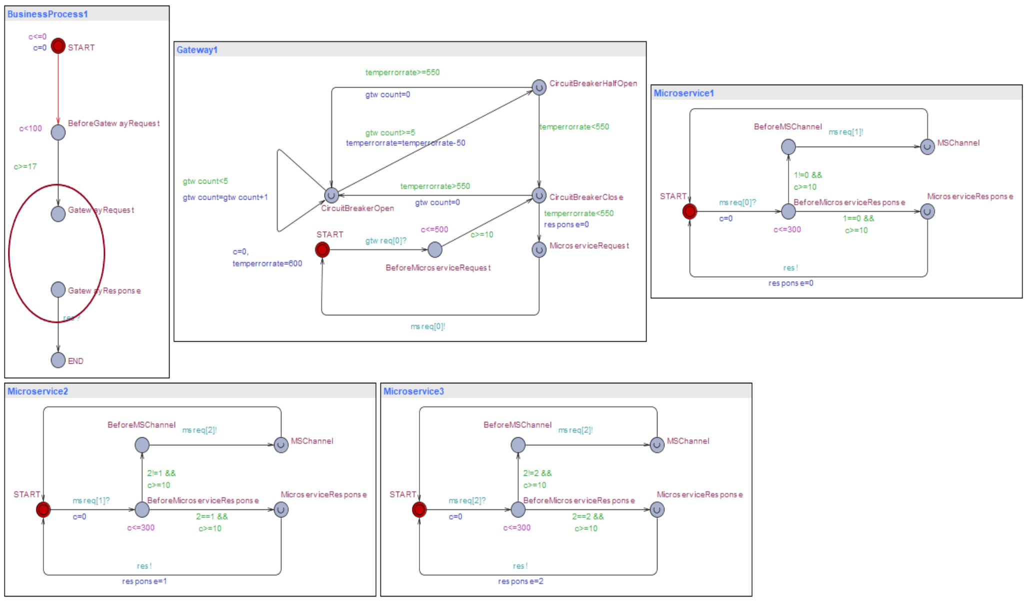


Figure 30. Fail scenario example for Microservice1

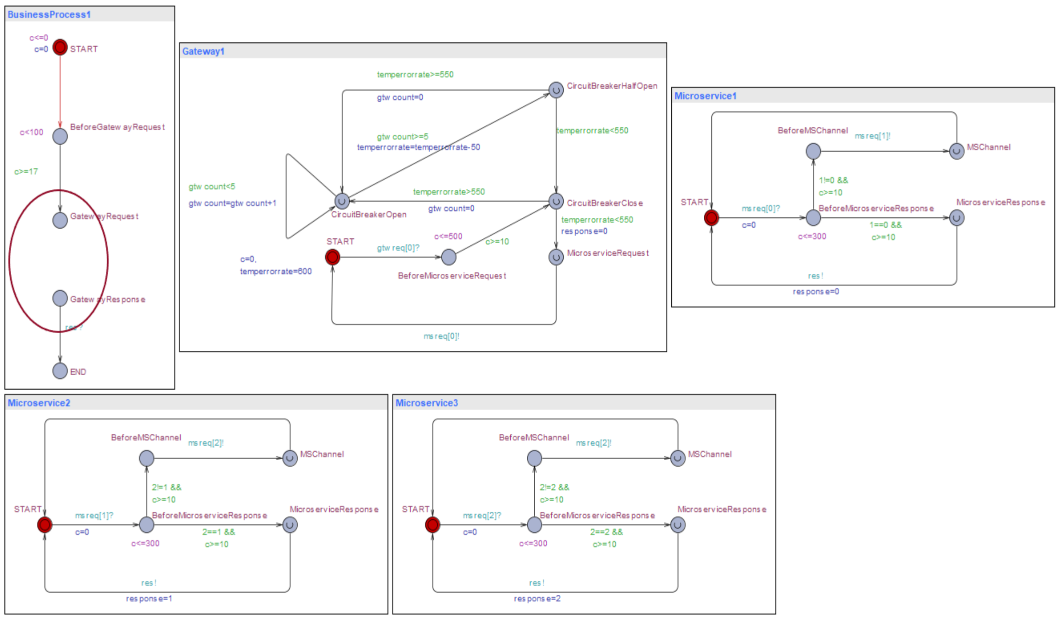


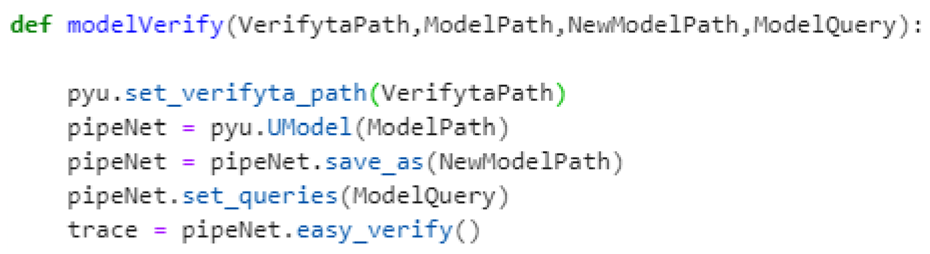
Figure 31. Fail scenario example for Microservice2

4.6. Verifying Test Models

After cleaning the test scenarios we created, we verified the scenarios. Since one of our goals was to optimize the testing processes, we carried out the verify process using the "pyuppaal" library in the python programming language (*15*). The function we created takes four parameters (*Fig. 32*).

These;

- VerifytaPath = We are giving the verify path of Uppaal application.
- ModelPath = We give the path to the model we will verify.
- ModelQuery = We write which query will verify the model.
- NewModelPath = We determine the path to save the model created with Query.



```python
def modelVerify(VerifytaPath,ModelPath,NewModelPath,ModelQuery):

    pyu.set_verifyta_path(VerifytaPath)
    pipeNet = pyu.UModel(ModelPath)
    pipeNet = pipeNet.save_as(NewModelPath)
    pipeNet.set_queries(ModelQuery)
    trace = pipeNet.easy_verify()
```

Figure 32. Verify function.

In our study, we determined two methods to create test scenarios. One of these methods is "Deleting Transitions" and the other is "Updating Guard and Transition Values", so we prepared queries to verify the models.

We prepared the following queries to reverse our model.

Prepared queries;

- A[]deadlock
- A<>(BusinessProcess1.END)
- A<>(BusinessProcess1.END and clk<=2500)

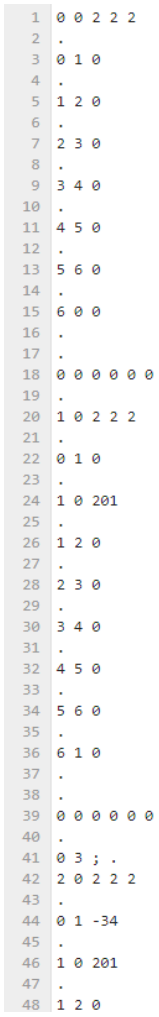
A[]deadlock => It ensures that the request coming to the model does not enter an infinite loop within the model and thus verifies the model.

A<>(BusinessProcess1.END) => It verifies whether the request to the model is successful.

A<>(BusinessProcess1.END and clk<=2500) => It verifies the response time of the request that must be received by the model.

4.7. Making Sense of Verify Output of Test Models

In our study, we verified the test scenarios. We create the outputs of the verified scenarios in the .xtr extension file format (*Fig. 33*). The specified file format is not a readable file format. That's why we worked to convert it into a readable file format. We functionalized it to automate our work.



```
0 0 2 2 2
.
0 1 0
.
1 2 0
.
2 3 0
.
3 4 0
.
4 5 0
.
5 6 0
.
6 0 0
.
.
0 0 0 0 0 0
.
1 0 2 2 2
.
0 1 0
.
1 0 201
.
1 2 0
.
2 3 0
.
3 4 0
.
4 5 0
.
5 6 0
.
6 1 0
.
.
0 0 0 0 0 0
.
0 3 ; .
2 0 2 2 2
.
0 1 -34
.
1 0 201
.
1 2 0
```

Figure 33. XTR file format example.

We showed the data in our work to make the XTR file format readable. At first glance, we learn in which location, at which clock value or in which transaction it is left.

In this way, when the model gets an error, we can see at which stage we got an error.

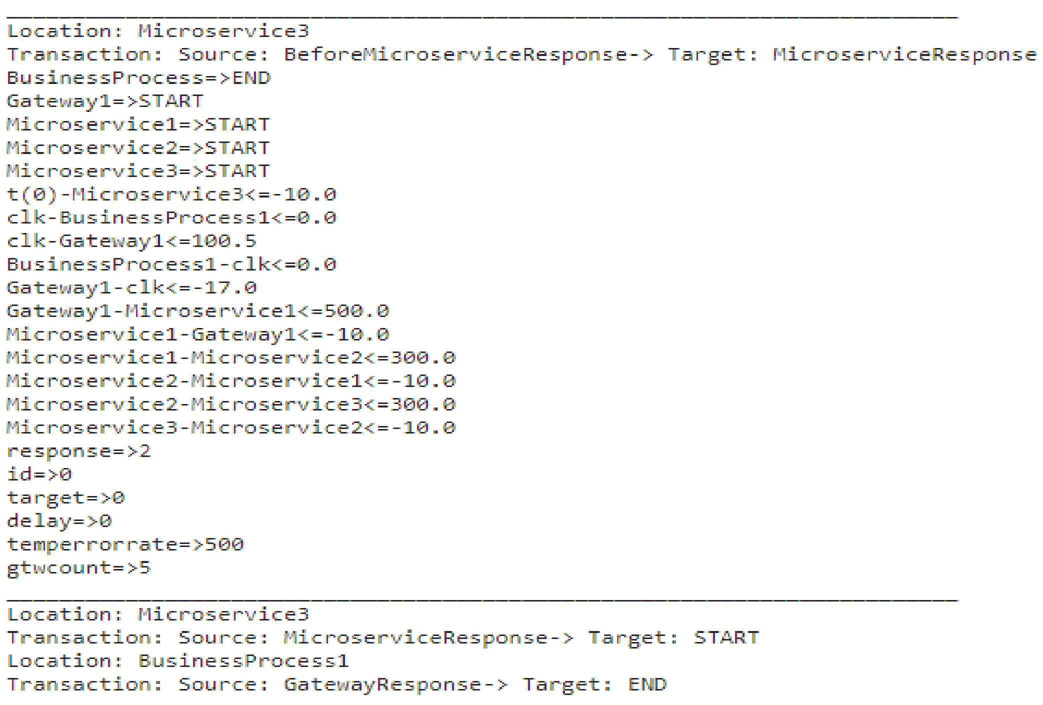

```
________________________________________________________________________
Location: Microservice3
Transaction: Source: BeforeMicroserviceResponse-> Target: MicroserviceResponse
BusinessProcess=>END
Gateway1=>START
Microservice1=>START
Microservice2=>START
Microservice3=>START
t(0)-Microservice3<=-10.0
clk-BusinessProcess1<=0.0
clk-Gateway1<=100.5
BusinessProcess1-clk<=0.0
Gateway1-clk<=-17.0
Gateway1-Microservice1<=500.0
Microservice1-Gateway1<=-10.0
Microservice1-Microservice2<=300.0
Microservice2-Microservice1<=-10.0
Microservice2-Microservice3<=300.0
Microservice3-Microservice2<=-10.0
response=>2
id=>0
target=>0
delay=>0
temperrorrate=>500
gtwcount=>5
________________________________________________________________________
Location: Microservice3
Transaction: Source: MicroserviceResponse-> Target: START
Location: BusinessProcess1
Transaction: Source: GatewayResponse-> Target: END
```

Figure 34. A certain part of the expansion of the XTR file format

As in Figure 34, we see a certain part of the .xtr file format. We can examine the part taken as a sample from the file as follows:

Our model is now in Microservice3, as a transition from "BeforeMicroserviceResponse" to "MicroserviceResponse" state, the other objects, BusinessProcess1, is in "END" state, Gateway1 is in "START" state, Microservice1 and Microservice2 are in "START" state, and Microservice3 is in "START" state. The rest of the file shows us the clock, invariant, global variable and guard values. In time period t(0), Microservice3>=10, BusinessProcess1<=0, clk-Gateway1.c ∈ [17,100], Gateway1.c- Microservice1.c ∈ [10,500], Microservice1.c-Microservice2.c ∈ [10,300], Microservice2.c - Microservice3.c ∈ [10,300]. We see that the response, id, target, delay values are “0”, temperature = 500 and gtwcount = 5.

We compared the results with the Uppaal tool to verify that the algorithm we prepared made the xtr file extension format readable (*Fig. 35*).

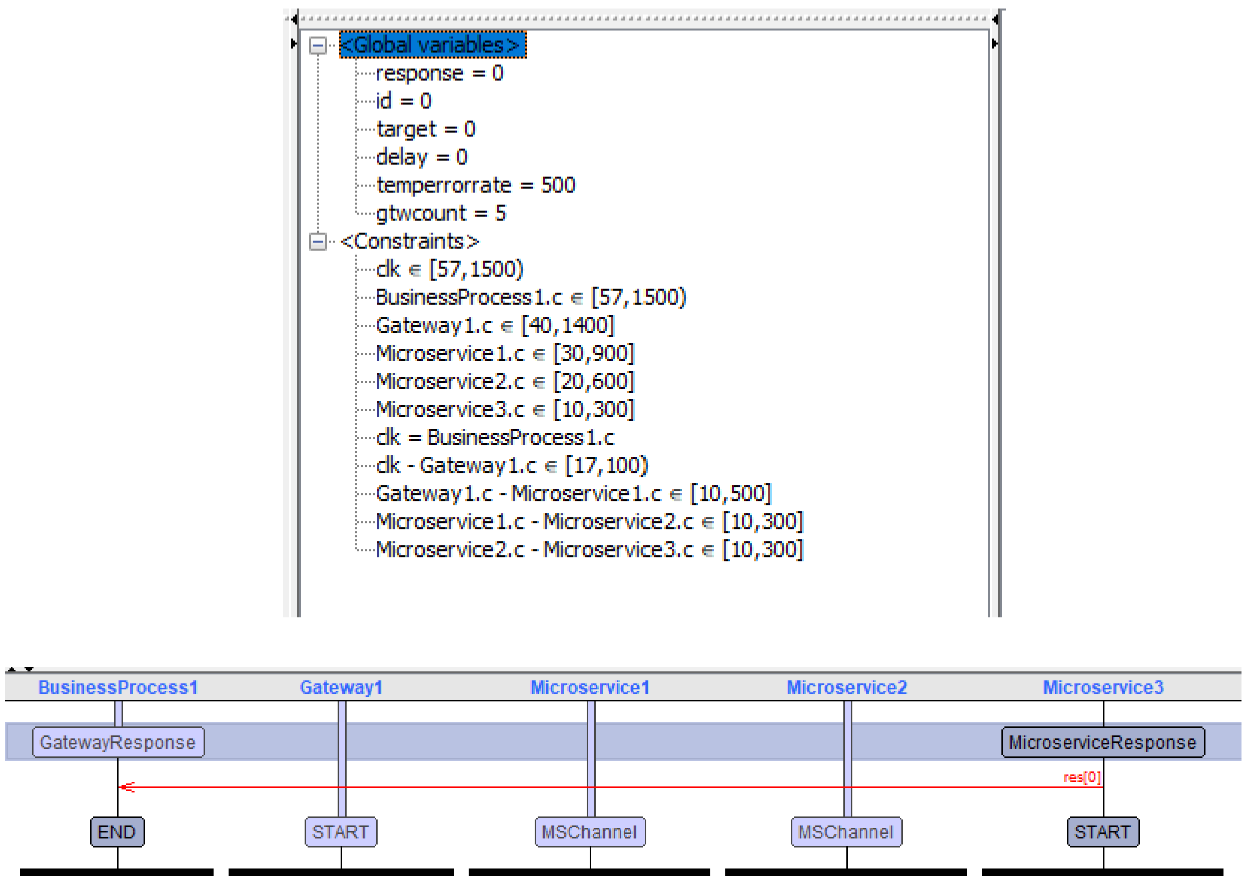

```
<Global variables>
    response = 0
    id = 0
    target = 0
    delay = 0
    temperrorrate = 500
    gtwcount = 5
<Constraints>
    clk ∈ [57,1500)
    BusinessProcess1.c ∈ [57,1500)
    Gateway1.c ∈ [40,1400]
    Microservice1.c ∈ [30,900]
    Microservice2.c ∈ [20,600]
    Microservice3.c ∈ [10,300]
    clk = BusinessProcess1.c
    clk - Gateway1.c ∈ [17,100)
    Gateway1.c - Microservice1.c ∈ [10,500]
    Microservice1.c - Microservice2.c ∈ [10,300]
    Microservice2.c - Microservice3.c ∈ [10,300]
```

Figure 35. Uppaal visual version of Figure 34.

CHAPTER 5

DISCUSSIONS AND RESULT

We carried out the testing processes in microservice applications with the model checker-based method. There are two important concepts in this study. One of them is microservice architecture and the other is model checker-based testing method.

Our work can be used in applications that use microservice architecture. While testing the developed application, the communication and dependency between the services are important for the testers. Therefore, this study eliminates this problem. Since we have created the model of the application, we can determine which services the request will pass from the starting point to the end point, through which parts of these services and in what time interval.

With the method we recommend, you can create a topology of the entire application and determine the time intervals of the critical points you see. This way, you can write tests for these critical points, regardless of internal or external services.

The person who will use the Model-Checker-Based testing method in microservices must have technical experience in more than one field.

First of all, she/he must have knowledge in the field of software. It is necessary to know what stages an application goes through while it is being developed and what kind of problems may arise at these stages. If these are known, it should be known what to pay attention to during the testing processes and what kind of model to prepare. If this competence is not possessed, the model will not be able to be built well.

Secondly, it is necessary to have knowledge about application architectures. Each application is developed according to different needs and different usage areas. Therefore, different architectures can be preferred. It may be found suitable for implementing microservice architecture, but may not be suitable for using a gateway. For this reason, when an application is running, the architecture needs to know what kind of life circle it should be.

The person who will use the method must know time-automata and Uppaal tool.

With this study, we wanted to contribute to the testing processes of applications implemented in microservice architecture. While carrying out this process, it was important to automate the test processes and quickly create and verify test cases. The computer specifications in the study were carried out on a computer with Intel(R) Core(TM) i5-4460 CPU 3.20GHz, 16.0GB RAM and 64-bit operating system.

There is a BusinessProcess part, a Gateway part and three Microservice parts in our model. As an approach, we have determined two methods for creating test cases, as explained in Chapter 4.

First of all, we created test cases that we could create using the transition deletion method. There are a total of thirty-one transactions in our model. This means we can create a total of thirty-one different scenarios. For this reason, we ran the transaction deletion method that we wrote in python programming language for each section. And thirty-one scenarios were created for each part. However, since there were similar scenarios, we simplified our test scenarios by clearing similar scenarios. As a result, we have as many test cases as the number of transactions of each section (*Table 1*).

	Number of Transitions in the Model	Number of Scenarios Created	Total creation time of the scenarios	Number of Scenarios Cleaned	Total Number of Scenarios
BusinessProcess	4	31	66	27	4
Gateway	9	31	84	22	9
Microservice1	6	31	77	25	6
Microservice2	6	31	77	25	6
Microservice3	6	31	77	25	6

Table 1. Transitions creation data of deleted scenarios.

To avoid an endless loop between the templates in our model, we first ran the **"A[]deadlock"** query. By seeing that not all scenarios can be verified, we have seen that they are not infinite loops. Then, we ran the **"A<>(BusinessProcess1.END)"** query to see if the model was still working despite the broken transactions. However, according to the stable working model we created, we saw that if the transaction breaks in the BusinessProcess and Gateway sections, a total of 13 scenarios fail. However, we saw that out of a total of 18 scenarios, 7 scenarios were successful in the broken transactions in Microservice1, Microservice2, Microservice3 sections (*Table 2*).

	Total Number of Scenarios	A[] deadlock not Verify	A[] deadlock Verify The time (ms)	A<>(Business Process1.END) Clock Verify	Clock Verify The time (ms)
BusinessProcess	4	4	438	0	445
Gateway	9	9	962	0	977
Microservice1	6	6	647	2	632
Microservice2	6	6	683	2	681
Microservice3	6	6	701	3	658

Table 2. Transitions query test data of deleted scenarios.

Secondly, we applied the method of updating Guard and Transition values. In this method, we updated the "delay" and "startclk" values of the parts in our model with the function we wrote. Our function produces two values, one of which represents the “delay” value and the other represents the “startclk” value. Since the “delay” value is an invariant value, we made it larger than the “startclk” value. For this reason, the "delay" value starts from 10 and increases 50 by 50 until it reaches the value "3000". The "startclk" value starts from 100 and progresses 50 by 50 up to 3000. Thus, we showed the number of scenarios that occurred for each part and the times during which they occurred in Table 3.

	Number of Scenarios Created	Total creation time of the scenarios
BusinessProcess	1769	3196
Gateway	1769	3156
Microservice1	1769	3211
Microservice2	1769	3135
Microservice3	1769	3278

Table 3. Data of scenarios created by updating Transitions and Guard values.

We performed the verification processes of the resulting scenarios and showed the results in Table 4. Of the total 8845 scenarios we created, 1260 were verified. The **A<>(BusinessProcess1.END and clk<=2500)** query tests that the request we send responds in at least one journey and only in one time period, less than or equal to 2500 milliseconds.

	Total Number of Scenarios	A<>(BusinessProcess1. END and clk<=2500) Clock Verify	Clock Verify The time (ms)
BusinessProcess	1769	252	189211
Gateway	1769	252	202754
Microservice1	1769	252	204746
Microservice2	1769	252	207235
Microservice3	1769	252	208664

Table 4. Query test data created by updating Transitions and Guard values.

As a result, if we can model our applications correctly, we can carry out testing processes quickly. As we showed in section 5.7, we can analyze the results in more detail and fix any errors or problems.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this study, we presented a new approach for using applications developed in microservice architecture in testing processes. With this approach, we carried out a study in which we can automatically create some scenarios that we have difficulty creating in our projects or that sometimes escape our attention even if we create them, and test the resulting scenarios. In this way, we can automatically create our scenarios in our projects and test which of the scenarios we create will be successful and which will fail. We can also see at what points the unsuccessful scenarios fail. To automate our work, we just used the uppaal tool to create the model of our application. Thus, we were able to create and test test scenarios by mutating our application model. We implemented the software of the processes in the Python programming language to create test scenarios, clean these scenarios, perform the verify operation and make the output files meaningful, and perform all these processes automatically (*16*). Thus, we have automated all scenario processes. With this approach, we automatically produced test scenarios of important points regarding communication and time in the applications we use today. Since we prepared the model parametrically, we can add a new microservice to our model in seconds and create the scenarios of this added service in seconds. In this way, our scenarios are produced both automatically and quickly, as we mentioned in the Result and Discussions section. We quickly verified the scenarios we produced, and in the unverified scenarios, we determined where the problem was. In this way, we can determine in which time intervals the communication points that are critical for us in our real-life applications and other points that are important in terms of time should work, or whether the application will work or not if communication is broken at which points. When our applications receive updates, they can see how the time and communication costs incurred in our application will affect our application and its processes. Thus, when we test our applications with the approach we offer, we can transition to the production environment without any problems in time and communication.

In future studies, the entire process can be done using an interface. In this way, the processes can be carried out faster and there may be no need for various competency levels of the people who will carry out the testing processes. Architectures and patterns can be given as templates in the created tools.

CHAPTER 7

REFERENCES

(1) Ünlü, H.; BiLgiN, B.; DemiRörs, O. A Survey on Organizational Choices for Microservice-Based Software Architectures. *Turkish Journal of Electrical Engineering and Computer Sciences* (2022), *30* (4), 1187–1203. https://doi.org/10.55730/1300-0632.3843.

(2) Ünlü, H.; Tenekeci, S.; Yıldız, A.; Demirörs, O. Event Oriented vs Object Oriented Analysis for Microservice Architecture: An Exploratory Case Study. *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2021). https://doi.org/10.1109/seaa53835.2021.00038.

(3) Ghani, I.; Wan-Kadir, W. M.; Mustafa, A.; & Imran Babir, M. Microservice Testing Approaches: A Systematic Literature Review. International Journal of Integrated Engineering, 11(8), 65–80 (2019).

(4) Ma, S.; Fan, C.-Y.; Chuang, Y.; Liu, I.; Lan, C.-W. Graph-Based and Scenario-Driven Microservice Analysis, Retrieval, and Testing. *Future Generation Computer Systems* (2019), 100, 724–735. https://doi.org/10.1016/j.future.2019.05.048.

(5) Waseem, M.; Liang, P.; Shahin, M.; Di Salle, A.; Márquez, G. Design, Monitoring, and Testing of Microservices Systems: The Practitioners' Perspective. *Journal of Systems and Software* (2021), 182, 111061. https://doi.org/10.1016/j.jss.2021.111061.

(6) Savchenko, D.; Radchenko, G.; Hynninen, T.; & Taipale, O. Microservice Test Process: Design and Implementation. *International Journal on Information Technologies & Security*, (2018)

(7) Fraser, G.; Wotawa, F. Mutant Minimization for Model-Checker Based Test-Case Generation. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)* (2007). https://doi.org/10.1109/taic.part.2007.30.

(8) Fraser, G.; Wotawa, F. Using LTL Rewriting to Improve the Performance of Model-Checker Based Test-Case Generation. *A-MOST '07: Proceedings of the 3rd International Workshop on Advances in Model-Based Testing* (2007). https://doi.org/10.1145/1291535.1291542.

(9) Fraser, G.; Wotawa, F. Ordering Coverage Goals in Model Checker Based Testing. *2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (2008). https://doi.org/10.1109/icstw.2008.31.

(10) UPPAAL.(2023) [online] Available at: https://uppaal.org. (accessed 2023-08-20)

(11) Hessel, A.; Larsen, K. G.; Mikučionis, M.; Nielsen, B.; Pettersson, P.; Skou, A. Testing Real-Time Systems Using UPPAAL. In *Springer eBooks*; (2008); pp 77–117. https://doi.org/10.1007/978-3-540-78917-8_3.

(12) Gong, X.; Ma, J.; Li, Q.; Zhang, J. Automatic Model Building and Verification of Embedded Software with UPPAAL. *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications* (2011). https://doi.org/10.1109/trustcom.2011.152.

(13) Alur, R. Timed Automata. In Lecture Notes in Computer Science; 1999; pp 8–22. https://doi.org/10.1007/3-540-48683-6_3.

(14) Wikipedia. (2023). Timed Automaton. [online] Available at: https://en.wikipedia.org/wiki/Timed_automaton. (accessed 2023-08-08)

(15) PypUppaal (2023) A Python Interface to UPPAAL [online] Available at: https://pypi.org/project/pyuppaal/ (accessed 2023-05-11)

(16) GitHub. (2023). Model Checker-Based Testing with UPPAAL. [online] Available at: https://github.com/Ozgur-OZTURK/Model-Checker-Based-Testing-Uppaal. (accessed 2023-12-01)