PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
Ministry of Higher Education and Scientific Research
University of Mohamed Khider – BISKRA
Faculty of Exact Sciences, Science of Nature and Life
Computer Science Department

Ordre N° : GLSD07/M2/2022

Thesis

Submitted in fulfilment of the requirements for the Masters degree in

Computer science

Option : **software engineering and distributed system**

SAGA distributed transactions verification using Maude

By :
Djerou Manel

Session Defended on 27/06/2022

Members of the jury :

Tigane Samir	MCB	President
Okba Tibermacine	MCA	Supervisor
Mohammedi Amira	MAA	Examiner

2021/2022

Acknowledgement

All our thanks and gratitude for ***ALLAH*** the Almighty who gave us all the courage and the will to go till the end of this project.

I would like to express my special thanks to my supervisor **Prof. *Okba Tibermacine*** to him I am deeply grateful for his help and guidance, patience, availability and support, my deep respect and thanks for his commitment to giving me the opportunity to carry out this project.

I am most grateful and I would like to express my special thanks to my teacher of english **Mr. *Seyf Eddine Messast*** for his help and support.

I am most grateful to my Teachers :

Mr. Babahenini Mohamed Chaouki.

Mr. Fodile Cherif

Mr. Kardoudi Lamine

Mr . Bachir Abd El Malik

Mr. Djeffal Abd El Hamid

Mr . Kahloul Laid

Miss. Bahi Naima

Miss. Somia Sahraoui

Mme Boughtitiche Amina

Miss. Roumani Yamina

Finally, I would like to thank all the ***Software Engineering and Distributed Systems*** teachers for their overall insights in this field and sharing their expertise.

Thank you.

Dedications

Thank you **God** for guiding me to accomplish this work; This work is dedicated :

To my dear Grandmother ***Sidehoum Khadidja*** (**MAYMA**) may God have mercy on her, She always showed me nothing but love and warmth; her love and support are what made me what I am now; thank you from The bottom of my heart;

To the most precious blessing I ever had and I am thankful for, “my parents” whose prayers light up my life;
To dear mother ***" BOUSLOUGUIA EL TAOUS "*** , a person full of love and caring from whom I learnt patience, thank you for everything;
To dear father ***" LHAJ ABD EL KRIM"***, who taught me to value knowledge, thank you for the encouragement and support I got throughout my whole career;
To my beloved Aunty ***Pr.*** **DJEROU LEILA** she was my role-model and my supporter also I am eternally grateful to Her daughter ***Dr.*** **KHELIL SARAH** .
To my lovely sisters " ***AMEL*** and ***DIKRA"*** for their help, advice and patience;
To my brothers “***HAMLAOUI*** and ***YACINE*** ” for their support;
To all my friends and classmates who helped me with their advice
A special dedication goes to my lovely friends:
My Best Friend " ***TOUMI YOUSRA***".
Extended gratitude to all my family members and To all who love me and always have wanted to see me successfully accomplish this work.;

Thank you

Djerou Manel

Abstract

Microservices is an architectural style that allows building software applications as a suite of small services which guarantees independent maintainability, deployability and scalability. *Database per service* is a pattern that is dedicated to build microservice architecture and manages efficiently the relationship between service business logic and data interpreted by the presence of a local database for each service. In fact, complex business tasks require the collaboration of many service, thus, a distributed transactions have to take place to effectively accomplish those tasks. Classical distributed transaction protocols such as Two Phase Commit protocol are not a good solution for this pattern. This is due to the fact that these protocols require all participants in a transaction to commit or roll back before the transaction can proceed. However some participant implementations, such as NoSQL databases and message brokering, don't support this model. Alternatively, saga is an efficient mechanism that goes along the 'Databae per Service' pattern and which ensures distributed transactions that span many services. The Saga pattern provides transaction management using a sequence of local transactions. The later is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction which guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed. In this work, we provide a formal verification of Saga using the Maude language. We present an elegant way to specify the Saga protocol by orchestration, as well as we present an efficient way to check their correctness.

Keywords: Software, microservices, two phase commit Protocol, Saga, Maude language.

Résumé

Les microservices sont considérés comme un style architectural permettant de créer des applications logicielles sous la forme d'une suite de petits services garantissant une maintenabilité, une capacité de déploiement et une évolutivité indépendantes. *Data base par service* est un pattern dédié à la construction d'une architecture à base microservices qui gère efficacement la relation entre la logique métier du service et les données, qui est interprétée par la présence d'une base de données locale pour chaque service. En fait, les processus métiers complexes nécessitent la collaboration de nombreux services. Par conséquent, des transactions distribuées doivent avoir lieu pour accomplir efficacement ces tâches. Les protocoles de transaction distribués classiques tels que le protocole Two Phase Commit ne sont pas une bonne solution pour ce modèle. Cela est dû au fait que ces protocoles exigent que tous les participants à une transaction s'engagent ou annulent simultanément avant que la transaction puisse se poursuivre. Cependant, certaines implémentations de participants, telles que les bases de données NoSQL et le courtage de messages, ne prennent pas en charge ce modèle de transaction. Alternativement, saga est un mécanisme efficace qui est souhaitable pour le modèle "Databae par Service" et qui garantit des transactions distribuées couvrant de nombreux services. Le modèle Saga fournit une gestion des transactions à l'aide d'une séquence de transactions locales. Ce dernier est l'unité de travail effectuée par un participant Saga. Chaque opération faisant partie de la saga peut être annulée par une transaction de compensation qui garantit que toutes les opérations se terminent avec succès ou que les transactions de compensation correspondantes sont exécutées pour annuler le travail précédemment effectué. Dans ce travail, nous fournissons une vérification formelle de Saga en utilisant le langage Maude. Nous présentons une manière élégante de spécifier le protocole Saga par orchestration, ainsi nous présentons une approche efficace qui vérifier leur exactitude.

Mots clés : Logiciel, microservices, protocole de commit à deux phases, Saga, langage Maude.

List of Figures

List of Tables

Contents

General Introduction

Software plays an important role in our daily life. over the least decade, software development has known a lot of progress in design, production, maintainability and deployment of software applications. Microservices is an architectural style that was proposed as a practical implementation of SOA (Service Oriented Architecture) that allows building software applications by the mean of a suite of small independent software units called services. The primary goal of this architectural style is to guarantee independent maintainability, deployability and scalability which let applications evolve and become more and more complex shifting the presence of classical problems.

Database per service, is a design pattern used in microservice architectural that lets the service manage domain data independently on a data store that best suites its data types and schema [12]. Further, it also lets the service scale its data stores on demand and insulates it from the failures of other services. Thankfully to this model, we can choose the technology stack per service. For example, we can decide to use a relational database for one service and a NoSQL database for a second service.

In fact, complex business logic require the collaboration of many service, thus, a distributed transactions have to take place to effectively accomplish those tasks. Classical distributed transaction protocols such as the Two Phase Commit protocol are not a good solution for this pattern. This is due to the fact that these protocols require all participants in a transaction to commit or roll back before the transaction can proceed. However some participant implementations, such as NoSQL databases and message brokering, don't support this model. Alternatively, saga is an efficient mechanism that goes along the 'Databae per Service' pattern and which ensures distributed transactions that span many services. The Saga pattern provides transaction management using a sequence of local transactions. The later is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction which guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed. In this work, we provide a formal verification of Saga using the Maude language. We present an elegant way to specify the Saga protocol by orchestration, as well as we present an efficient way to check their correctness using the executable specifications generated by the Maude language and their verification capabilities.

The present manuscript is organized in 5 chapters including general introduction and conclusion:

- The first chapter describes the microservices architecture style, the distributed databases, Distributed transactions Concepts and a pattern in microservices architecture.
- The second chapter covers and explains all that is related to Software, verification and validation, formal methods, rewriting logic and Maude language. The chapter is subtitled " Rewriting logic, Maude and Formal Verification".
- The third chapter is dedicated to the formal verification of Saga using Maude language. First we present a real case study. Then we specify each element of the orchestration (Namely, services, messages, message brokers and compensation tasks) using rewriting theories. Then, we present the mechanisms that Maude supports to verify Saga.
- Finally, the manuscript is finished by a general conclusion.

Chapter 1

Background: Distributed Database and Microservices

1.1 *Introduction*

Microservices is an architectural style that is used to develop software systems. This architecture offers a lot of benefits compared to monolithic architecture. often the business logic implemented by microservices require data serialization and storing in databases. In fact, database design is one of the biggest challenges of microservices where the database per service organization represents a distributed database that needs a certain pattern to manage data within microservices. In this chapter, we present an overview about microservice architecture and Distributed databases focusing on protocols to ensure consistency using distributed transactions.

1.2 *Microservice Architecture*

1.2.1 *Microservices definition*

Microservices depend on containers as a design element for building a distributed application. They are named so because each function of the application operates independently as a service. This architecture design allows each service to be scaled or updated independently .

This framework creates a flexibly manageable system which avoids the bottlenecks of central database improving business capabilities, among those abilities is enabling continuous delivery and deployment . [29]

1.2.2 *Microservice Architecture definition*

According to ***Martin Fowler*** " The **Microservice architectural** style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [20].

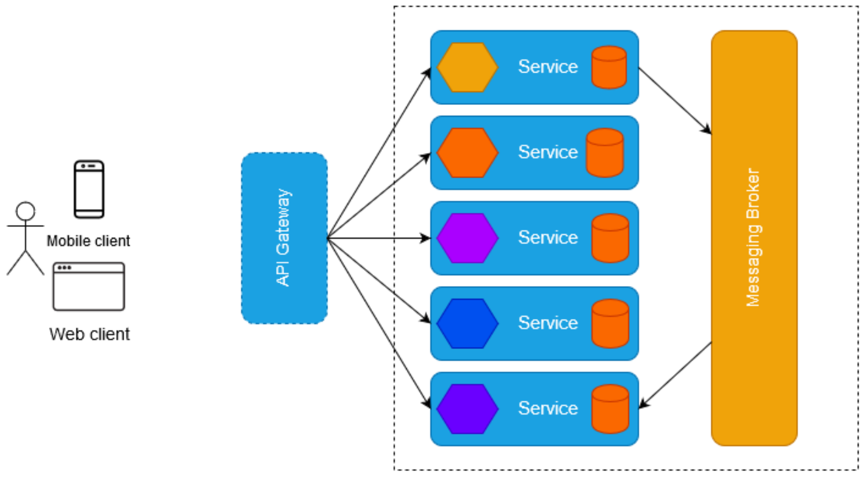


Figure 1.1: Microservice Architecture

[27]

These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. [20]"

1.2.3 Who uses microservice architecture ?

Among the most prominent users of microservice architecture are the following :

- eBay
- Amazon

- Twitter
- PayPal
- SoundCloud
- Netflix
- Facebook [18]

Figure 1.2: Companies Using Microservices

[13]

1.2.4 Benefits

Microservices deal with the problem of complexity by decomposing huge monolithic applications into sets of services without changing the function. Each service has its own boundary could ARPC (remote procedure call).

This level of simplicity and modularity is unachievable in any monolithic application which makes microservices easier and faster to understand, develop and maintain.

Microservice architecture allows each service independence in development and freedom of the developing team which enable them to choose the appropriate technology to the task.

Been independent allows microservices to deploy independently from each other, developers don't need to coordinate the deployment or changes and can apply them locally without widespread system shutdown. [26]

Microservices depend on the efforts of small specialized teams which tend to be more productive and agile and which also means it requires smaller base code. In the case of a microservices crashing the rest of application is not affected, that is because they implement circuit breaking .

Services can be scaled independently, upgrading and changing is done easily and without scaling out the entire application. The developer can pack a higher density of services onto a single host, which allows for more efficient utilization of resources and more freedom and less cost of upgrade. [21]

1.2.5 Drawbacks

One of the most proponent disadvantages of microservices is the complexity within the system. This complexity arises from the fact that microservices are a distributed system which needs a communication mechanism using RPC and also developers need to maintain and handle any failures locally because of its inter-process communication.

In microservice system testing poses a challenge because of the multiple interactions happening between services. The aspect of asynchronicity adds to the difficulty of testing due to the very dynamic ecosystem it creates. With the new concern over privacy and trust, a microservice system is built by deferent teams and managing trust between them and even with other organizations is challenging.

Coding a microservices application can be costly both in resources and time, it also requires a different approach than writing a traditional monolithic or layered application.

Different microservices require different languages and frameworks so this can create problems in regard to working standards .

With each microservice responsible for its own data persistence, As a result, data consistency can be a challenge. Embrace eventual consistency where possible.

In order to successfully manage microservices system the team needs a high level of skills and experience. [21]

1.2.6 Building a Microservices Architecture

The monolith is dividable into independent microservices. The connection between these microservices creates the application architecture .

Using this architecture offers great agility for enterprises to scale and develop at will without the risk of other microservices being affected. [17]

1.2.6.1 Understand the monolith

Examine how a monolithic application works and identify the functions and services of the components it executes. [17]

1.2.6.2 Develop the microservices

Develop every application function as a stand-alone microservice, operating independently. These are typically executed in a container on a cloud server. Each microservice serves only one purpose: search, dispatch, payment, accounting, payroll, etc.[17]

1.2.6.3 Integrate the larger application

Freely integrate microservices through API Gateway to make them work together to form the largest application. An iPaaS like DreamWorks can play a key role in this.[17]

1.2.6.4 Allocate system resources

Use container orchestration tools like Kubernetes to manage how system resources are allocated for every microservice.[17]

1.3 Distributed database (DDB)

1.3.1 Databases (DB)

1.3.1.1 Definition

A database is a systematic collection of related, logically coherent data, stored and manipulated electronically enabling the access and retrieval of information as needed. [25, 15, 11]

All the above is done using a database management system (DBMS).

Both the management system and the database and any related applications are addressed as DATABASE. [23]

1.3.1.2 Types of Databases

Here are some popular types of databases. [23, 25]

- Distributed databases(DDB);
- Relational databases;
- Object-Oriented databases ;
- Centralized databases(CDB);
- Open – source databases;
- Cloud databases ;
- NoSQL databases;
- Graph databases;
- Personal databases;
- Hierarchical database;
- Network DBMS;

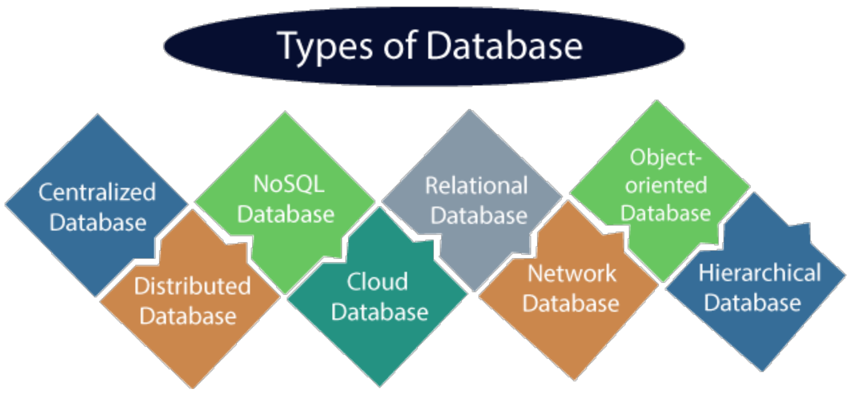


Figure 1.3: Types of Database

[10]

1.3.2 Why opt for DDB over CDB?

DDB was chosen based on the fact that it is more expandable meaning that in the case of the need to add new locations or units to the database it can be done easily and without much disruption in the service . Another reason to choose DDB is the reliability it offers as the system can still function in the event of the failure of one component of the database . DDB also offers faster response times (to queries) which facilitates the user experience.

There are many more factors that played a role in choosing DDB over CDB. [3]

1.3.3 Distributed Database definition

A distributed database (BDD) is a set of multiple, logically interrelated databases distributed across a computer network. It is operated by a DBMS that is the software that handles the DDB and provides an access mechanism that makes this distribution transparent to users.

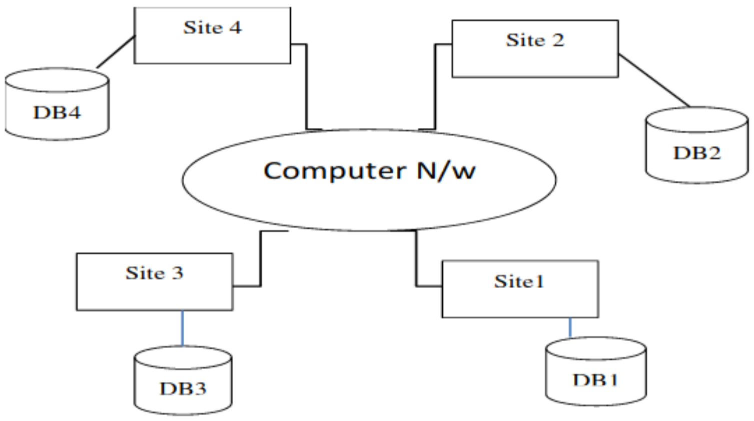


Figure 1.4: Distributed Database System[28]

1.3.4 Types of Distributed Databases' Architectures

There are two types distributed of architecture :

- Homogeneous Distributed Database System [9]
- Heterogeneous Distributed Database System [9]

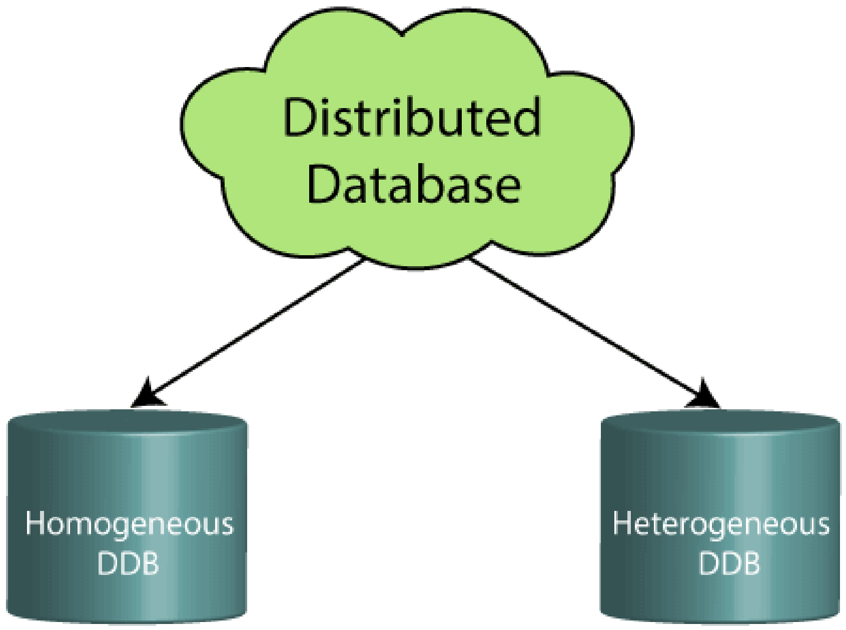


Figure 1.5: Distributed Database System

[5]

1.3.4.1 Homogeneous Distributed Database (Peer-to-Peer Architecture)

Is a web of identically designed and operated data bases stored on multiple sites. All the sites share the same operating system DBMS and data structure which makes managing data bases and maintaining them easier.

They enable users seamless access to the data and this approach allows for growth and improved performance. [8]

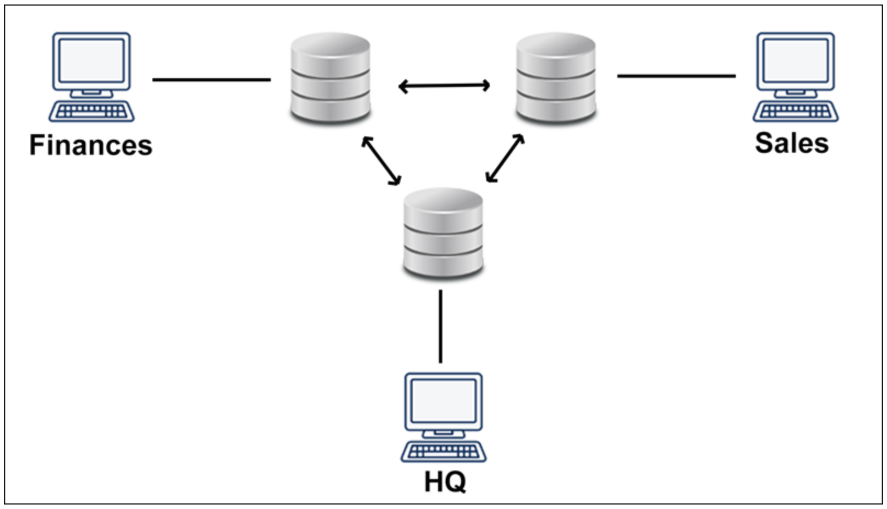


Figure 1.6: Homogeneous Distributed Database System

[8]

1.3.4.2 Heterogeneous Distributed Database(Multi-DBMS Architecture)

It is employ different designs operated system, DBMS and different data management models .

This means that particular site can stay an effected by the changes done on other sites. Translation is needed to allow the use of deferent hard ware and different DBMS. [8]

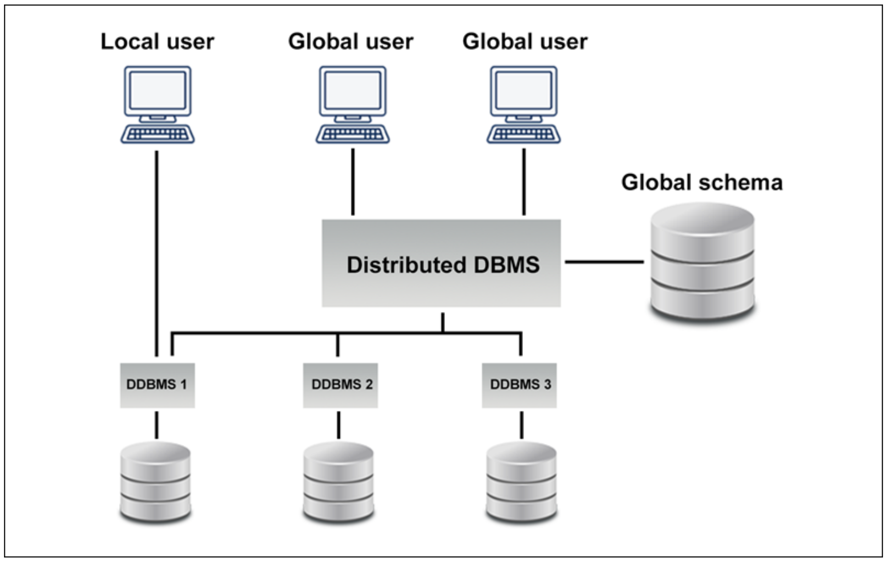


Figure 1.7: Heterogeneous Distributed Database

[8]

1.3.5 Advantages Vs disadvantages

Below are some key advantages and disadvantages of distributed databases :

1.3.5.1 Advantages

+ Flexibility : Growth is easier. We don't need to interrupt sites in operation to introduce (add) a new site. Hence, the expansion of the whole system is easier. Site removal is also does not cause much problems. [2]

+ Reliability : A distributed database system is resilient to failure to a certain extent. Hence, it is reliable in comparison to a centralized database system. [2]

+ Lower maintenance : The data is distributed in such a way as to be available near to the location where it is most needed. This reduces the cost of communicating much more compared to a centralized system. [2]

+ Better user experience : Much of the data is local and in close proximity to where it's needed. Hence, the requests can be responded to rapidly compared to a centralized system. [2]

1.3.5.2 Disadvantages

- Processor-Demanding : a distributed database demands constant attention to different processes which makes it CPU intensive. [2]
- Risk of security breach: Data integrity becomes complex. Too much network resources may be used.[2]
- Improper data distribution : In some cases, we may not distinguish site failure, network partition, and link failure. [2]
- Reliant on network : in poor-infrastructured countries relying on network can be unpredictable . [2]

1.3.6 Distributed Transactions Concepts

1.3.6.1 Distributed Transactions Definition

A distributed transaction is composed of one or more statements that, individually or as a group, update data over two or more different nodes of a distributed database. [4]
The operation known as a “two-phase commit” (2PC) is a distributed type of transaction. "XA transactions" are transactions using the XA Protocol, which is an implementation of a two-phase commit operation. A distributed transaction spans multiple databases , ensuring the integrity of the data. [1]

1.3.6.2 Two phase commit protocol (2PC)

Two-phase commit is a protocol that insures the integrity of distributed data base transaction in the case of site communication failure or falter errors guaranteeing atomicity of transactions . [30]

Phase1 : when all the participant nodes signal the coordinator that the transaction involving all of them has finished.

The coordinator publishes a message to all the nodes to prepare for commitment.

Phase 2 : if the nodes complete their local recovery sequence successfully they will send "ready to commit " or " OK to coordinator" . If even one of the nodes is not prepared to commit the whole transaction is rolled back.

If the coordinator does not receive a response from any of a nodes, it assumes a "not ready "response.

After receiving the ok message from all participants nodes the coordinator publishes a commit signal the transaction, the nodes write a commit entry and permanently update the database.

If the coordinator or one of nodes fail the transaction is rolled back through an UNDO message to all nodes. [22]

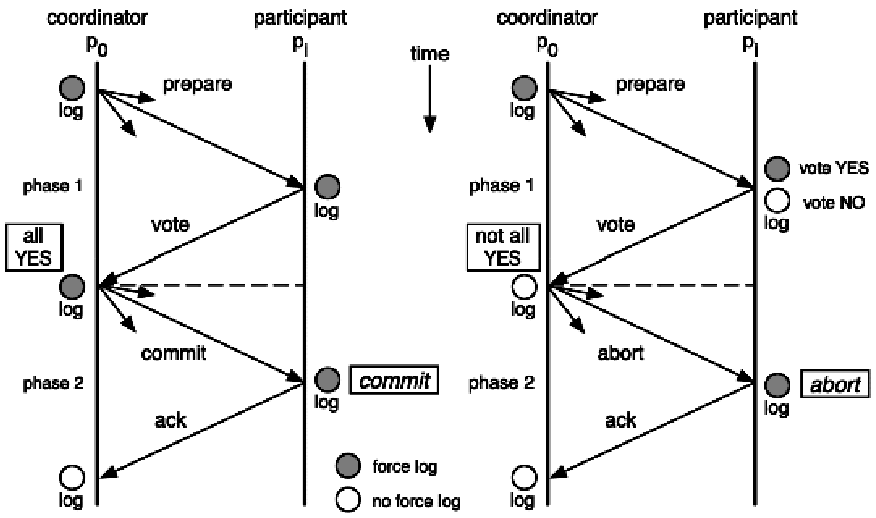


Figure 1.8: Two Phase Commit Protocol

[7]

1.4 Pattern in microservices architectures

1.4.1 Database per service

The loose coupling of services represents One of the core characteristics of the microservices architecture. For that reason every service must have its own databases, it can be polyglot persistence among microservices. [14] The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

Each service's persistent data can only be accessed via Rest APIs. So database per microservice provides many benefits, especially for evolving rapidly and supporting massive scale systems.

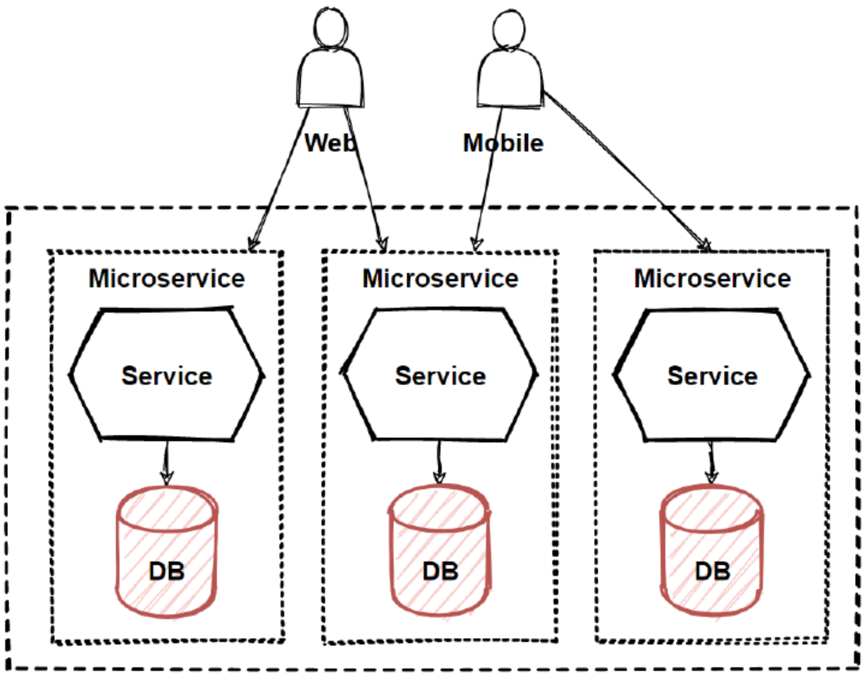


Figure 1.9: Database Per Service

[24]

1.4.1.1 Benefits

- Helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services. [14]
- Each service can use the type of database that is best suited to its needs. [14]

1.4.1.2 Drawbacks

- implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them. [14]
- Implementing queries that join data that is now in multiple databases is challenging. [14]

- The complexity of managing multiple SQL and NoSQL databases. [14]

1.4.2 Saga pattern

1.4.2.1 Definition

A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

1.4.2.2 Function

Using microservice architecture allows a degree of separation between the application and its database. [16]

The saga pattern can manage transactions resulting from multiple microservices using a sequence of local transactions. [16]

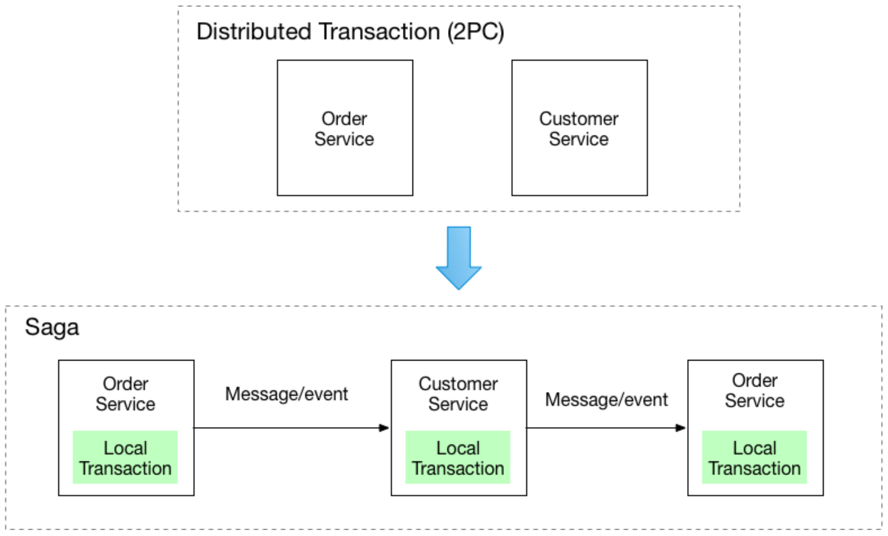


Figure 1.10: SAGA

[14]

For the sake of performance, two-phase commit is not a good idea for many applications.

There are two ways of coordination sagas:

A)Choreography: each local transaction publishes domain events that trigger local transactions in other services. [14]

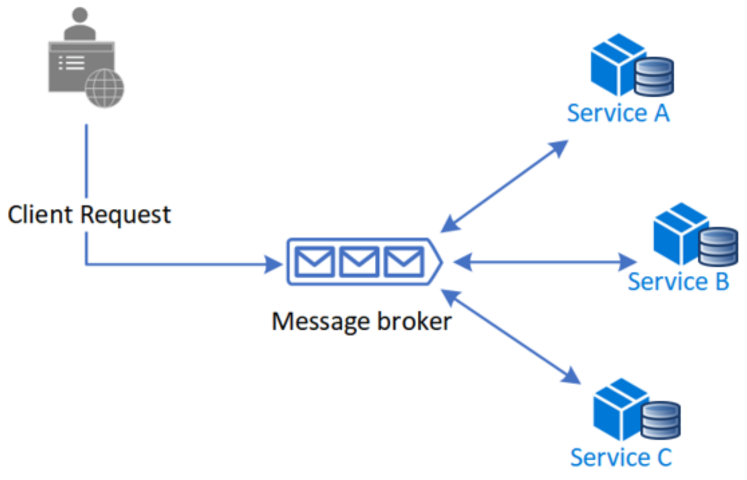


Figure 1.11: Choreography-Saga

[6]

A.1) Benefits: Choreography-based sagas have many benefits[14]:

- Good for simple workflows that require little participation and no coordination logic. [6]
- No need for further implementation and maintenance. [6]
- Does not present a single point of failure, because the responsibilities are distributed through the participants of the saga. [6]
- Simplicity — Services publish events when they produce, update, or cancel business objects[14]
- Loose connection — The participants subscribe to events and don't have direct knowledge of each other. [14]

A.2) Drawbacks: And there are some drawbacks:

- The workflow can become confusing when adding new steps because it is difficult to know which participants are listening to which commands. [6]

- There is a risk of cyclical dependence between saga participants because they have to consume one another's commandments. [6]

- Integration testing is challenging because all services have to work to simulate a transaction. [6]

B) Orchestration: an orchestrator (object) tells the participants what local transactions to execute. [14]

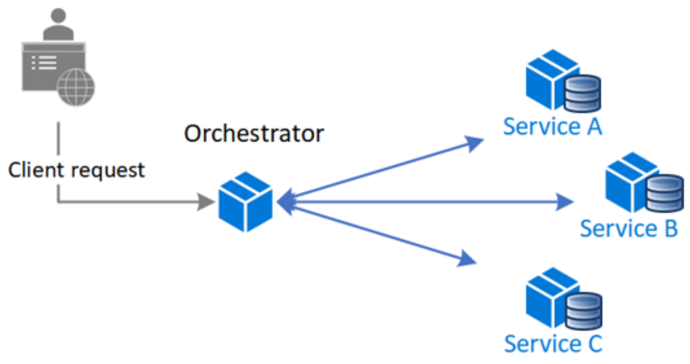


Figure 1.12: Orchestration-Saga

[6]

B.1) Benefits:

- Suitable for complex workflows with multiple participants or newly added participants over time. [6]

- good when there's control over every participant in the operation, and control over the inflow of conditioning. [6]

- Does not present cyclical reliance, because the orchestrator unilaterally depends on the saga participants. [6]

- Saga participants do not want to know about commands for other participants. Clear separation of interests simplifies business sense. [6]

B.2) Drawbacks:

- As the design becomes more complex, you need to implement tuning logic. [6]

- There are additional obstacles as Orchestrator manages the entire workflow. [6]

1.5 Conclusion

Through this chapter, we presented outlines about microservices architecture and distributed databases where we showed the relation between them. Then, we introduced the two phase commit protocol that is used to manage distributed database transactions and the SAGA pattern that is replacing it in a database per service. Later we will discuss the SAGA pattern and its relevance in managing database per service transactions compared to Two phase commit protocol.

Chapter 2

Rewriting logic, Maude and formal verification

2.1 INTRODUCTION

Verification is the process which ensures that a software program accomplishes its objective without any defects. This process has various techniques, in this chapter, we focused on the presentation of formal verification methods.

2.1.1 Software verification and validation

2.1.1.1 Software

Definition 1:

A software is the set of instructions and algorithms that command and control a computer to perform a certain task it contains the programs, procedures and routines related to the functioning of computerized system.

This term was coined to distinguish from the term hardware (the physical component of computer system).

Definition 2:

According to Len Bass and colleagues at the Software Engineering Institute (www.sei.cmu.edu), they define software architecture as follows: “The software architecture of a computing system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both." Documenting Software Architectures by Bass et al.

2.1.1.2 Types of software

A) New software: is software written specially for the application it is intended to be used with.

B) Existing accessible software: is software related to a similar application to the user's requirement, the documentation is accessible and can be recycled.

C) Existing proprietary software: it is existing software that meets all the requirements but the documentation is unavailable because it is most a likely a commercial product.

D) Configurable software: typically, software that already exists but is configured for the specific application using data or an application specific input language.

2.1.1.3 Verification and validation in software development life cycles

To test the adequacy of software during its life cycle the software must undergo testing as a verification technique.

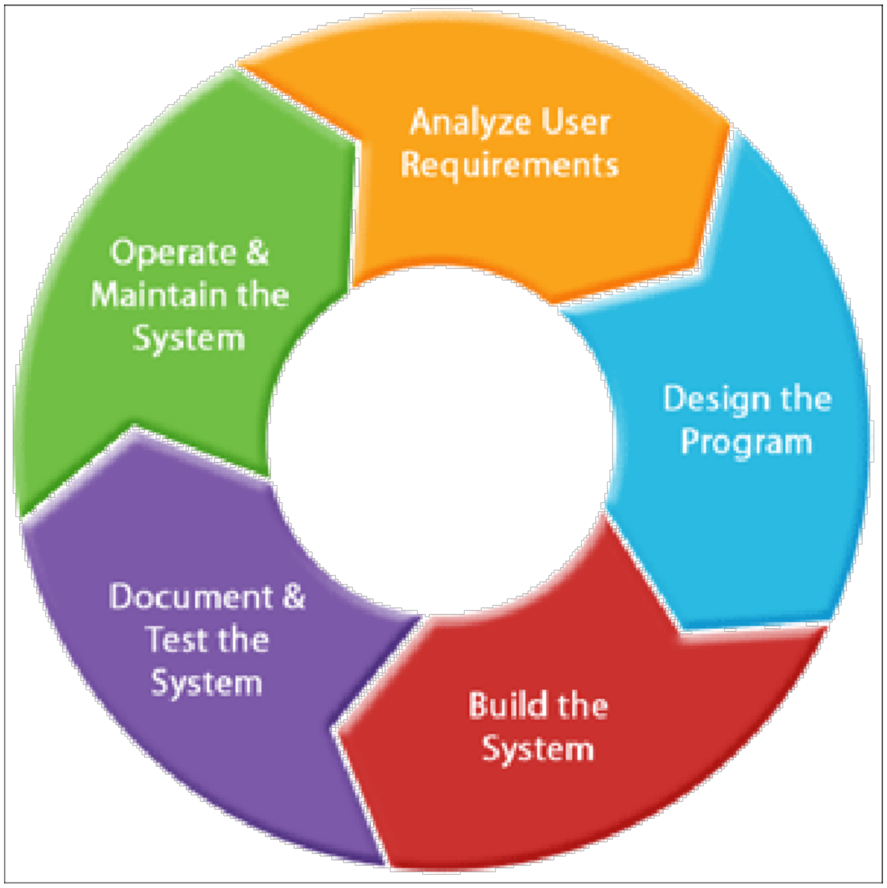

Analyze User Requirements
Design the Program
Build the System
Document & Test the System
Operate & Maintain the System

Figure 2.1: Software development lifecycle

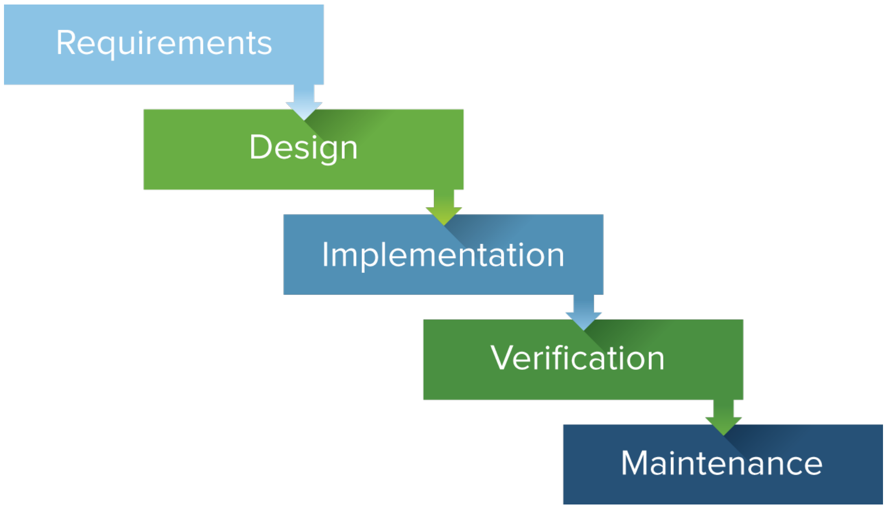


Figure 2.2: Software development lifecycle

A)Verification and validation for the requirements phase:

A.1) Requirement analysis phase:

The objective of this phase is to make sure that requirements are stated clearly and unambiguously and it starts as soon as the requirements are agreed upon.

A.2) Specification requirements phase:

It aims at inspecting the software requirement specification (SRS) compared to the user's requirements.

SRS can be either informal or very formal.

B) Verification and validation for the design phase:

It aims at the assessment of correctness, consistency and adequacy of the design based on the requirements.

This phase starts when the design team designs the software.

C) Verification and validation for the coding phase:

It aims at confirming that the source code complies with the organization's coding standards, implements the required functionality, satisfies performance and security requirements.

C.1) Testing phase:

During this phase the system is integrated with other systems to test its functionality compared to the requirements defined earlier.

The verification and validation teams are responsible for the integration of the software to ensure the system falls under the constraints stated in the requirements specifications and the user's needs.

D) Verification and validation for operation and maintenance:

No system is perfect and while using it the software can exhibit defects or the users would like some modifications to make the software easier to use.

As an example to this, a user could ask for a design rework to make their work flow easier by offering shortcuts to specific parts of the system.

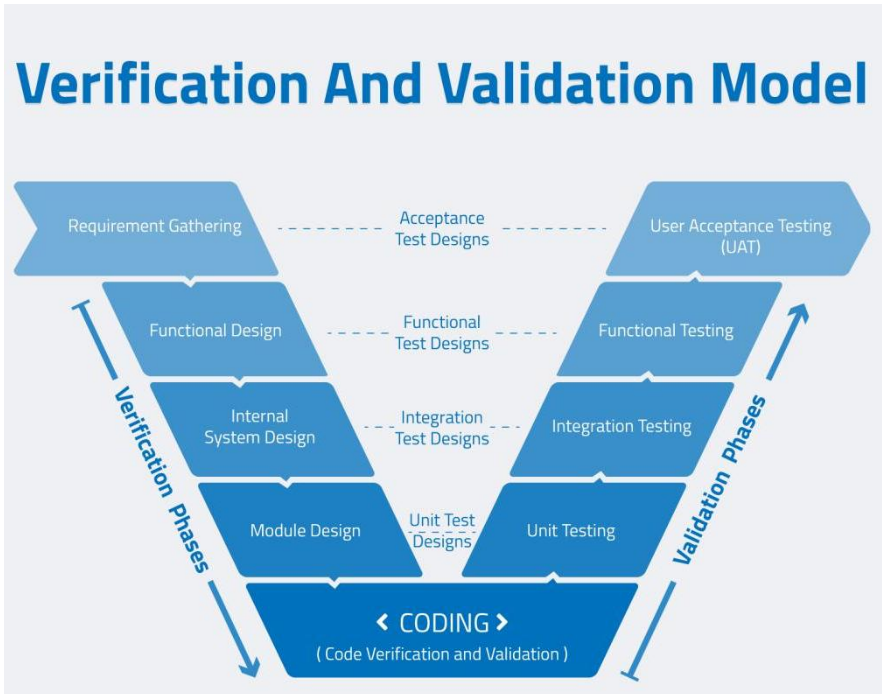


Figure 2.3: Verification and Validation Model

2.1.1.4 Verification and Validation terms

4.1. Verification definition:

Verification is the process of evaluating our system or parts of it to find out whether it corresponds with the conditions we have set.

The verifications can look for correctness, completeness, consistency and accuracy in the outputted information from the software.

Verification can be performed at any phase in the software development cycle.

It is considered important step in the software development process.

4.2. Validation definition:

Is providing evidence that a product a service or a system is complete and corresponds to the user's expectations and does what is needed of it.

4.3. Software verification and validation:

The purpose of V and V is to build quality checking mechanisms into the software development process ensuring the system will meet the requirements put forward by the developer and the user.

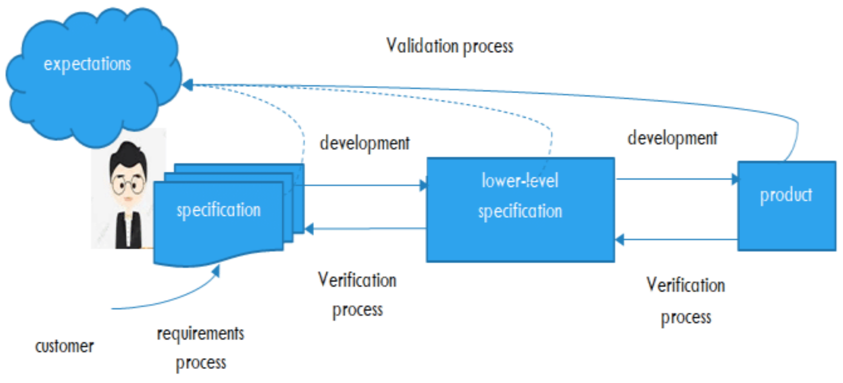


Figure 2.4: A continuous VV process in software system development life Cycle. (Software verification and Validation)

2.1.1.5 Differences And relation between V and V

Table 2.1: **Difference Between V and V**

Verification	**Validation**
1)Static process (unchanged process)	1) Dynamic process (changeable process)
2)Checks program design, document, code (parts of software)	2) Checks the finished product (complete software)
3) Techniques used in verification are reviews, walkthroughs, inspections and desk-checking.	3) Techniques used in validation are Black Box Testing, White Box Testing and non-functional testing.
4) It confirms whether the product meets the specifications or not.	4) It confirms whether the product meets the user's expectations and requirements.
5)Verification is concerned about the correctness of the process.	5)Validation is concerned about the correctness of the product
6)Discovers bugs throughout the development process.	6) Only discovers bugs related to final execution
7) Final code is not yet executed	7) Final code is executed

Relation between verification and validation is:

Verification is followed by Validation.

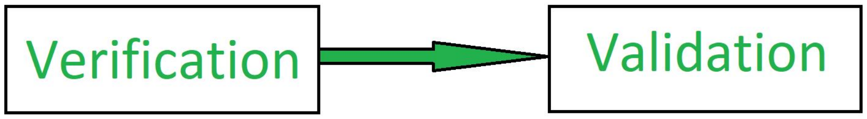


Figure 2.5: Relation Between V and V

2.1.1.6 Methods of V and V

Methods of verification:

A) Review: it is a process of the general overview of the product detecting and noting defects and errors by compering to predetermined specifications and guidelines.

B) Walkthrough:

Walkthroughs are essential parts of the early evaluation of documents, models, designs and code in the software development phase.

Walkthroughs aim to evaluate a specific part of software detecting its problems and suggesting solutions and offering the developer learning experience throughout the process.

C) Inspection: it is the process of checking how the conceptual model to the executable one which lays the road for an action plan to solve the problems.

D) Audit: it is a review to establish how a product matches the guidelines it was built upon and that all requirements have been met.

Methods of validation:

A) Unit testing: is testing on a unitary (each unit is tested individually) it also makes sure each unit is ready for integration with other unites. It is usually performed by the developer by using white box testing method.
What is a white box testing?

White box testing aims to check the internals of a software.

the tester can see the code working hence the name" glass box test" through the use of debugging tools.

Prior preparation is required for the test as improvisation is not recommended.

White box testing is concerned with the details of the inner working of an application and it goes deeper than the results and outputs of the software.

B) Integrated testing: is testing systematically the software components integrated with each other and test the link between unit blocks. It is usually performed by the developer as well as an independent test team.

C) System testing: it is overall testing of the whole system to insure functional and structural integrity. It is usually performed by the independent test team by using black box testing method.

What is a black box testing?

This testing approach examines the application's accessible inputs as well as the expected outputs that each input should produce.

It is unconcerned with the program's inner workings, the procedure it uses to create a certain result, or any other internal feature of the application that may be involved in the conversion of an input into an output.
Most black-box testing solutions rely on picture recognition or coordinate-based interaction with the application's graphical user interface (GUI).

A search engine is a perfect example of a black-box system.

You type the text you're looking for into the search field, hit "Search," and the results appear. In this scenario, you don't know or observe the particular procedure used to generate your search results in this example; all you see is that you give an input – a search word – and you get an output - your search results.

D) User acceptance testing: the final testing phase performed by the user to check the complaints with hardware and software requirements and the acceptance for delivery to the customer. It is performed after system testing and before making the system available to the user.

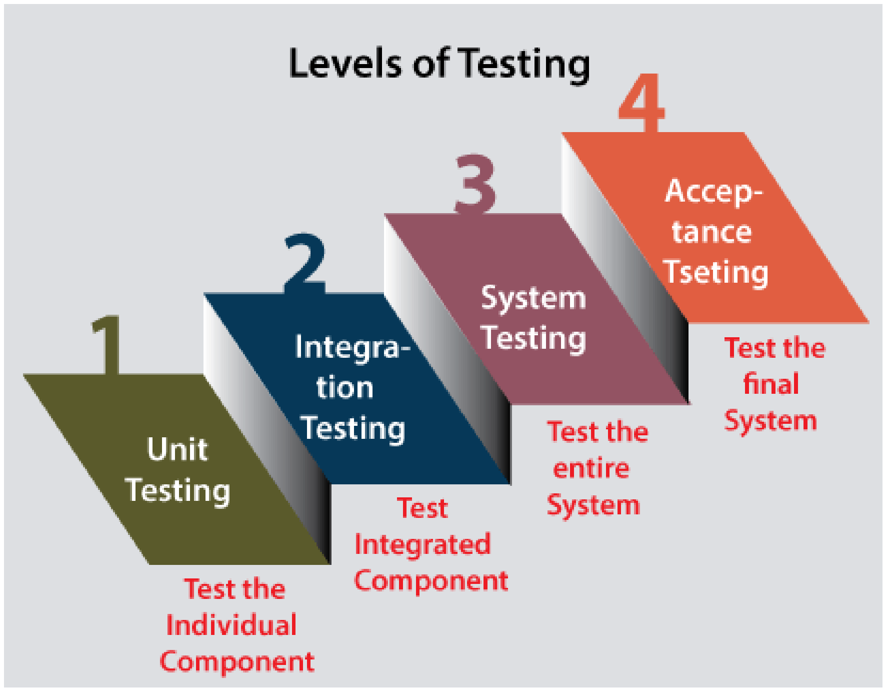


Figure 2.6: Methods of Validation (Tester's Life Cycle)

2.1.2 Formal methods (FM)

2.1.2.1 Definition

Formal methods are the application of mathematical and modeling techniques on the software specification, design and verification and validation. They can be applied to general aspect of the system as well as more specific ones[19].

They are mostly used in development of the most important and highly crucial software that requires high levels of security, safety and efficiency

2.1.2.2 Advantages FM

+ By forcing the system developer to think critically about the specification formal methods insures the proper engineering approach to specification.

+ FM insures fewer bugs and errors from the design stage on world.

2.1.2.3 Disadvantages FM

- FM is costly economically and resources wise and it also requires more time.
- Many FMs exist and they are not compatible with each other.
- Applying FMs does not guarantee the functionality of the hole system. It only works to ensure no part of the software is left out.

2.1.2.4 Levels of FM

Formal specification:

Is a general view of a system's properties and behaviors it is based on the mathematical logic describing what the system is expected to do[19].

A) Abstraction: Abstraction entails excluding non-developmental features and displaying just those that are required. During the software development cycle's requirement analysis stage a specification is developed.

The requirements for the software system to be created should be specified in this specification. A strong requirements definition explains what the system should perform in terms of some key features, rather than how those properties should be achieved.

It should concentrate on the most important features and leave the more complicated implementation details for later. A specification may have many implementations as a result of abstraction.

The act of contemplating a less thorough characterization of observable system behaviors is referred to as abstraction.

For example, instead of evaluating all of the intermediate programs of execution, one may focus on the end outcome of a program's execution.

A concrete (more exact) model of execution is defined as abstraction. However, while not always right, abstractions should be precise. That is even if the abstraction just yields an undecidable conclusion, it should be feasible to derive exact responses from them.

We discover similarities between items, events, or processes and address them through abstraction, which is a key technique of dealing with complexity.
Specification is a formal term that refers to the process of defining a system abstractly.

It Is the omission of non-relevant details from the development.
Only essential details are kept for later developmental phases [19].

B) Refinement:

Refinement is the technique used to specify the specification process and get more and more precise each and every step (correct according to specification)[19].

Formal verification:

Formal verification is the process of checking if a system meets the specification determined for it (compeer to initial requirements)[19].

A) Type of logic:

Propositional Logic:

This reasoning is comparable to Boolean Algebra semantics. Variable [0,1] is a topic in algebra. Conjunctions, disjunctions, negations, implications, reductions, and equivalences are the six types of compound sentences used in propositional logic[19].

Propositional logic is full and determinable. A compound statement in propositional logic, for example:

$$(\mathrm{p} \wedge q) \Rightarrow \neg r$$

First order predicate logic:

First order Predicate logic is concerned with the concepts of predicate and quantification. It employs the universal quantifier (for all) and the existential quantifier (there is).
Other logical symbols, such as brackets and the equality symbol, are the same as those used in propositional logic. It is definable, but not exhaustive.
For example: no one is superior than another[19].

$$\forall m, n. person(m) \wedge person(n) \Rightarrow \neg superior(m, n)$$

High order logic:

It incorporates quantification reasoning across collections and predicates. If a logic permits sets to be quantified or if collections can be elements of other collections, it is termed higher order logic.
Individual variables that we can quantify over predicate variables, function variables, and so on are called higher order variables.
term does not use implications or universal quantification; instead, it employs unique substitution to prove a goal from a collection of specific sentences[19].

$$\lambda x(person(x) \wedge \forall y(child(x, y) \supset doctor(y)))$$

B) Theorem Proving:

Theorem proving signifies building a mathematical proof which proves that a mathematical declaration is true. If the action results in a proof, the statement is considered real and is called a theorem. If there is no evidence, the declaration cannot be said to be false.

It may be that the statement is false, or it may be that the statement is true, but the person or evidence system used to find the evidence is not strong enough to find it.

The inference rules will be applied to the specification through the prover theorem, which will enable us to develop new system properties. Theorem proving tools have a robust set of different reasoning steps that can be used to reduce the proof goal to simpler subgoals that the prover can automatically offload using the original proof steps[19].

Formal proof:
A proof is a logical demonstration that a given theory is consequential of the relies applied. The proving process is done using computer-oriented language and note natural language limiting the possibility of errors and reducing the risks of misunderstandings[19].

Proof Checker:

Proof Checker are software component that can run a check and analyses on a proof to check if it is indeed the correct proof for the correct theorem.

The system is responsible for conformation and validation of the proof's logical integrity.
Automated theorem provers:
Automated theorem provers are software component automatically and autonomously search for valid proof of a specific theorem. They are very advanced and require expert supervision and huge computational power.

These tools are so difficult to make because of their advanced and complex nature[19].

C) Model Checking:
Model checking techniques are based on models identifying the system's expected behavior and

features in a clear and exact manner.

This is done using mathematical techniques, and is usually expressed through finite-state automata.

Abstractions omit irrelevant details.

A model checker is a computer-oriented tool that performs model checking. There exists many model checkers each of them with its own language for development and expression of properties and algorithms[19].

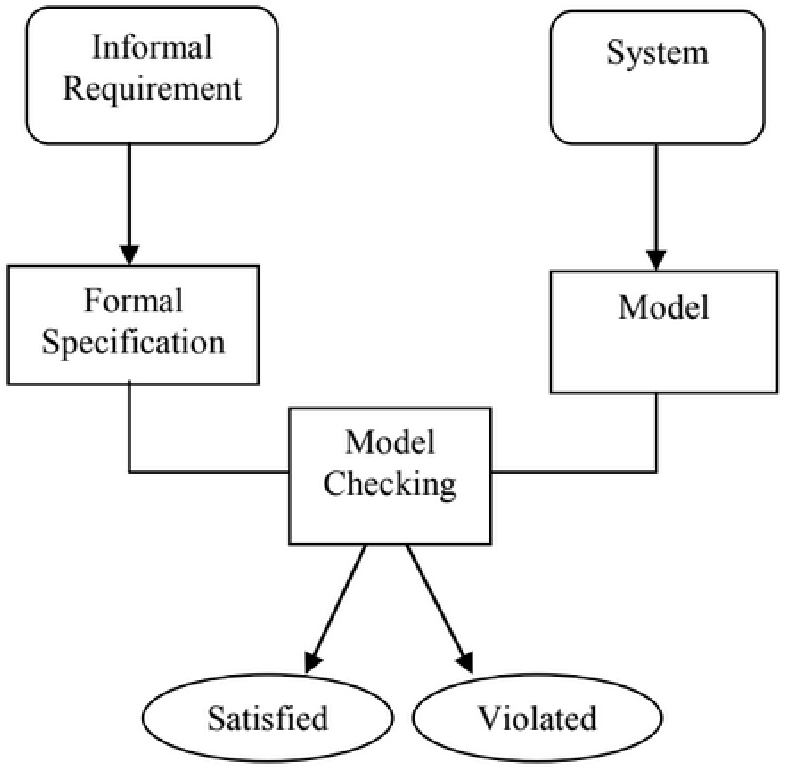


Figure 2.7: Schematic view of the model-checking approach

1) Phases in model checking:

- Checking and assessment phase done quickly to the model and after that, performing simulations to insure the proper specification language.
- Runing phase the model checker is ran to validate the property in the system model, the model needs the appropriate settings and directives.
- Analysis phase determines if the specific property is valid or not.

When all the phases are done and the system is confirmed to have all required properties as an examples of model checkers:

PROB[ ], SPIN[ ], NuSMV[ ] and SAL[ ].

2) Methods of model checking

In model checking, there are fundamentally two ways that differ in how the intended behavior, for instance, the requirement specification is stated.

1- Logic-based or heterogeneous method: the intended system behavior is described in this technique by specifying a set of attributes in an appropriate logic, often temporal or modal logic.

2- Behavior-based or homogeneous method: In this technique, both intended and possible behavior are expressed in the same notation, such as an automaton, and equivalence relations or pre-orders are employed as a correctness criterion.

3) The Advantages of Model Checking General approach to hardware assessment, software engineering, multiagent devices, communication protocols, embedded systems and so on.

Case studies have demonstrated that including model checking into the design process does not cause any more delays than simulation and testing.
Model checking is built on solid and intriguing mathematical foundations such as modeling semantics, currency theory, logic and automata theory, data structures, graph algorithms, and so

on.
Allows for partial verification: This can lead to increased efficiency since one can limit validation to only testing the most important criteria while disregarding less important but potentially computationally costly requirements.

4)Model checking's constraints Finding relevant abstractions, like the system model and acceptable attributes in temporal logic, necessitates considerable knowledge.
Suitable for control-intensive applications with component-to-component communication. It is less suitable to data-intensive applications because data processing typically introduces finite state spaces
Model checking's applicability is susceptible to definability concerns in some circumstances, including as most kinds of intestate systems, where model checking is not effectively computable. Formal verification, on the other hand is in theory applicable to such systems.
Only specified requirements are examined; there is no guarantee that required qualities are complete.

D) Model Checking versus Theorem proving: Model checking is advantageous only in its full automation Vs theorem proving as it is easier and faster to perform formal verification.

Model checking on the other hand is disadvantageous in regard to its feasibility to check very large systems and that is due to the state space explosion problem (when the number of states exceeds the computer's memory).

More advanced techniques have been developed to handle large systems (up to 108 or 109 states).
Also, model checking cannot be used to check generalizations in contrast theorem proving can be used to check the generic software's validation. [19]

2.1.2.5 Formal specifications languages

Specification languages come in all shapes and forms and each one is suitable to a specific kind of system.

One of the main differences is that each language has its own mathematical logical framework, this contains software models, statements, formulas used in expressing the properties of such models. [19]

A. Model Oriented Languages:

1) B:

B is a way for describing, developing, and coding software systems in a formal manner. It was created in 1980 by Jean-Raymond Abrial.

The abstract state machine is the underlying mechanism of this method. Data and operations are stored in each machine.

The data is always obtained through the machine's actions. A formal approach known as Event-B was just devised.

B is seen as an enlargement of Event-B. There are a number of sophisticated commercially available tools that assist in the writing of B specifications. PROB and AtlierB are two examples[19].

2) VDM:

In the 1970s, IBM's Vienna Laboratory established the Vienna Development Method (VDM). It includes of a formal specification language known as VDM-SL, rules of the language for data and operation refinement which, allow for the establishment of linkages between abstract requirements specifications and concrete design specifications at the implementation level, and a proof theory in which rigorous argumentation about the attributes of defined systems and the accuracy of specification that is mandated in design may be done. VDM++, a more advanced version, is used to model parallel and object-oriented systems. SpecBox, Overture, and VDM tools are all available for VDM[19].

3) Z:

Z has been created by the Oxford University Programming Research Group (PRG). In Z, we separate the specification into a collection of states of an abstract data type that is specified by a schema, which has typically the same name as the data type itself.

A schema is used to describe a system's static and dynamic behavior.

There are a number of available commercially tools that can help us to write Z specifications. such as include Z Word and ProZ[19].

B. Property Oriented Specification language:

A software engineering technique known as algebraic specification is used to formally specify system behavior and attributes.

To interpret information systems, these languages use methods drawn from abstract algebra or category theory.

The algebraic technique was created to define the interface of abstract data types. The type operation is used to define the type, not the type representation. CASL and CafeOBJ are two examples. [19]

1) CASL:

As defined by the Common Algebraic Specification Language (CASL) is a formal specification language. It employs first-order logic. It was created in 1997 by the Common Framework Initiative.

This powerful expressive specification language with easy-to-understand semantics, is a language that can be used to express requirements and design software packages.

CASL is primarily concerned with basic, organized architectural standards and specification libraries.

They were created to be easily combined, so that basic specifications may be used in structured specifications, which can then be used in architectural specifications, and both structured and architectural specifications can be collected into libraries. HETS and CASL are CASL tools[19].

2) CafeOBJ:

CafeOBJ is a Modern successor to OBJ that combines and integrates various new algebraic specification paradigms into an executable industrial algebraic specification language.

It is primarily meant for formal system specification, validation, and verification of specifications, as well as quick prototyping[19].

C. Process Oriented Language:

Process oriented formal specification language is used to describe or specify parallel systems. These languages are built on a special implicit parallelism architecture.

Expressions and elemental expressions, which define particularly simple operations via operations that join processes to give new potentially more complicated processes are used to point and construct processes in these languages.

Communicating Sequential Processes is an example of this (CSP). [19]

1) Communicating sequential Processes (CSP):

C. A. R. Hoare was the first to create it in 1978. CSP has been used to explicitly describe and test the concurrent characteristics and features of several systems in practice.

CSP specifies systems in terms of component processes that function independently and communicate with one another via a message-passing communication mechanism.

FDR and ProBE are two tools for CSP. Model-oriented, property-oriented, and process-oriented specifications are all combined in certain languages, RAISE, for example.[19]

RAISE :

In the 1990s, Dines Bjrner developed RAISE (Rigorous Approach to Industrial Software Engineering) as part of the European ESPRIT II LaCoS project.

VDM, Z, CSP, B, and OBJ were all designed with the purpose of delivering a unified enhancement over formal approaches.

The RAISE Specification Language (RSL) and the RAISE method are two of the services provided by RAISE.

The RAISE method describes how to use Raise Specification Language in different stages of the software development life cycle.

It also contains methods for explicitly and carefully testing specification properties[19].

2.1.2.6 Best Practices FMs:

a. Precision: using a mathematical notation allows the designer to be very precise about the specification it also facilitate the detection of problems.

b. Conciseness: the formal method is one the most conciseness a language program among the others which make site easer to comprehend.

c. Abstraction: it allows the writer more focus on the essential features of the system.

d. Reasoning: ones a formal specification is available, mathematical reasoning is possible to help in the validation.

2.2 Rewriting logic and Language Maude

2.2.1 Language Maude

Maude is a high level and high-performance language and system supporting both equational and rewriting logic programming and computation.
Maude is a formal specification and declarative programming language based on a rewriting logic mathematical theory. Jose Meseguar and his team at SRI International's laboratory created the rewriting logic and the Maude language. Maude is a clear, expressive, and powerful language.
It is regarded as one of the best languages for algebraic specification and the modeling of parallel systems .
Maude specifies theories of rewriting logic; data types are defined algebraically by equations and the dynamic behavior of the system is defined by rewrite rules.
Maude also allows object-oriented programming and asynchronous communication via message passing.
Maude's team has also concentrated on performance, since the present version of the interpreter can handle millions of rewrites per second. In terms of efficiency, Maude competes with high-level languages.
Maude also has socket support for network programming, allowing not just the modeling, simulation, and analysis of concurrent computer systems, but also their programming.
Maude provides three sorts of modules for specifying a concurrent system:
As a result, three types of modules are distinguished: Equational theories are implemented us-

ing functional modules.
System modules implement rewriting theories and determine a system's dynamic behavior.
Object-oriented modules (which can be simplified to system modules) implement object-oriented rewriting theories .

2.2.2 Rewriting logic

Rewriting logic is an ever-changing logic that can naturally deal with real time computations. It supports object-oriented computation. It is a good semantic and logical framework.

Being metalogic, it is able to include many other logics in representation and execution.
A parallel system is defined in rewriting logic by a rewriting theory R = (, E, L, R) where (, E) is the signature (an equational theory) describing the specific algebraic structure of the system states (multiset, tree, binary tree...) that are distributed according to this same structure.

The named rewriting rules R explain the system's dynamic structure (L is a set of labels of these rules).

The rewriting rules indicate which elementary and local transitions are feasible in the current state of the concurrent system.

Each rule (denoted [t] [t']) represents an activity that can occur concurrently with other actions. in conjunction with other acts.

Thus, rewrite logic is a logic that clearly represents a system's competing change.

2.2.2.1 Definition (Labelled Rewriting Theory)

A rewrite theory R is a 4-uplet (, E, L, R) such that:

1. is a set of function symbols, and sorts.

2. E a set of -equations (the set of equations between -terms).

3. L is a set of labels.

4. R is a set of rewriting rules, defined as follows:

Each rule is a pair of elements, the first is a label, the second is a pair of equivalence classes of terms T ,E(X) on the signature (,E), modulo the equations E, with

X = x1, ... xn , ... an infinite and countable set of variables [Mes04].

For a rewrite rule of the type r([t], [t'], C1,..., Ck), the following notation is used: r: [t] [t'] if C1... Ck, where a rule r specifies that the equivalence class containing the term t can be rewritten to the equivalence class containing the term t' if the condition of rule C1... Ck is verified.

The latter is known as the rule condition and may be shortened by rule and can be abbreviated by the letter C, and the rewrite rule is said to be conditional in this situation.

The conditional part of a rule can be empty, in which case the rules are called unconditional called unconditional rewrite rules and are denoted by r: $[t] \rightarrow [t']$ A rewrite rule can be parameterized by a set of variables x1, ..., xn which appear either in t, t' or C, and we write:
r: $[t(x1, ..., xn)] \rightarrow [t'(x1, ..., xn)]$ if C(x1, ..., xn) [Mes92].

2.2.3 Functional modules

These modules specify the data types and operations utilized by the equations.

In Maude, an equational specification is a functional module represented by the following syntax: fmod MODULENAME is ***(le nom du module) BODY ***(les déclarations de sortes, d'opérations, de variables, D'équations, d'axiomes d'appartenance et de commentaires) Endfm.

Where MODULENAME is the name of the introduced module, and BODY is the set of declarations of sorts, operations, variables, equations, membership axioms and comments.

Comments start with '***' or '—' and ends with the end of the current line or they start with *** (or — (and end with the occurrence of with the occurrence of")".

The body of the module specifies a theory (, E U A,) in the logic equational logic of membership. The signature includes sorts (indicated by the keyword sort), subsorts (specified by the keyword subsort) and operators (introduced with the keyword op).

The syntax of the operators is specified by the users by identifying the location of the parameters with the symbol (_).

The set E represents the equations and the membership tests (which may be conditional) and A is a set of equational axioms introduced as attributes of certain operators in the signature such as the axioms of associativity (specified by the keyword assoc), commutativity (specified by the keyword comm) or identity (specified by the keyword id). The latter are defined in such a way that equational deductions are made modulo the axioms of A.

Equations are specified with the keyword eq or the keyword ceq (for conditional equations) and membership tests are introduced with the keywords mb or cmb (for conditional membership tests). A condition related to an equation or a membership test can be formed by a conjunction of unconditional equations and membership tests of unconditional membership equations and tests.

Variables can be declared in modules with the keywords var or vars, or introduced directly into equations and membership tests, in the form of a var of a var expression: sort .
An example of a functional module of natural numbers is the following:

```
fmod BASIC-NAT is ***(le nom du module est : BASIC-NAT)  \\
sort Nat .  ***(declaration de sorte naturelle)  \\
op 0 : -> Nat [ ctor ] . *** (declaration  d une  op ration  \\
<math matiquement : fonction> qui donne z ro)  \\
op s_ : Nat -> Nat [ ctor ] .***(d claration  d une  op ration)  \\
op _+_ : Nat Nat -> Nat .  \\
op max : Nat Nat -> Nat .  \\
vars N M : Nat . ***(d claration des variables naturelles N et M)  \\
eq 0 + N = N .***(  quation  )  \\
eq s M + N = s (M + N) .  \\
eq max(0, M) = M .  \\
eq max(N, 0) = N .  \\
eq max(s N, s M) = s max(N, M) .  \\
endfm  \\
```

2.2.4 Rewriting system modules

Rewriting logic module always results with a single value. to reach this value each step is replacement of equals by equals. The rewrite paths can lead to infinite chains of rewriting. Or it can have divergent paths.

Rewriting logic is a logic of concurrent state change.

If we add rewriting rules to functional modules, they become a system module. This process enables us to model concurrent systems as conditional rewrite theories.
A system module is a description of a rewriting theory that comprises types, operations, variables, equations, membership axioms (conditional and unconditional), and conditional and unconditional rewriting rules.

A rewrite rule is executed when its left side matches a portion in the global state of the system and with the satisfaction of the condition in case of a conditional rule [Mcc03].

In Maude, the following syntax represents the system module:

mod MODULENAME is

BODY

endm.

R rewriting rules are introduced using the terms rl or crl. They are specified in Maude using the syntax:

crl [l] : t => t' if cond .

If the rule is unconditional, the crl term is changed with rl, and the "if cond" clause is removed [Cla02].

The following is an example about how these modules can be used:

```
mod CHOICE-INT is***(CHOICE-INT est le nom du module) \\
including INT ***(importation du module INT (le module pour les entiers)
   \\
pour utiliser ses op rateurs) \\
op _?_ : Int Int -> Int***(d claration  d un  op rateur) \\
vars I J : Int .***(d claration de deux entiers I et J) \\
rl [choose_first] : I ? J => I . \\
rl [choose_second] : I ? J => J. \\
endm \\
```

2.2.5 Object oriented modules

Object-oriented modules are supported by the Full Maude system .

Note that Full Maude is an extension of Maude (Core Maude) whose code is written in Maude, which enriches the Maude language with a very powerful and extensible module. These object-oriented modules are introduced by the keywords:

(omod MODULNAME is ..... Endom)

Object-oriented modules, as comparison to system modules, provide a more appropriate syntax for specifying the fundamental elements of the object paradigm, such as classes, objects, messages, and configuration .

The concurrent state of a concurrent object-oriented system, referred to as a configuration (line 1), is often structured as a multi-set of objects and messages (line 2).

As a result, we can see the creation of configurations by a binary operator of union of multi-sets, which we can represent by the syntax of (line 5), where the union operator of multi-sets is declared to satisfy the structural laws of associativity and commutativity, and to have commutativity and null as its identity.
The third line (line 3) indicates that the object and message states are singleton configurations, according to the subsort declaration.

```
1 sort Configuration . *** [1] \\
2 sorts Object Msg . *** [2] \\
3 subsorts Object Msg < Configuration . *** [3] \\
4 op null : -> Configuration . *** [4] \\
5 op _ _ : Configuration Configuration -> Configuration [ assoc comm id :
     null ] . *** [5] \\
6 An object in a given state is represented as a term: \\
7
8 < O : C | a1 : v1 ,..., an : vn > \\
```

Where O is the object's name or identifier, C is its class, an are the object's attributes and vn are the corresponding values.

- Each class is declared by the syntax:

class C| a1 : S1, ..., an : Sn .

Where C: name of the class, an of the attributes, Sn: type of each attribute.

- The declaration of a message is defined by the syntax:

msg m : p1 , ..., pn -> Msg .

Where m: message name, pn: kinds of associated attributes.

- Messages of the same kind can be declared using the keyword msgs.

- Maude also allows class inheritance: a subclass declaration is defined by the syntax: subclass C' < C.

The structure and behavior of the subclass C are determined by the attributes, messages, and rules of the superclass C, as well as the newly specified attributes, messages, and rules of the subclass.

Maude is a proponent of multiple inheritance.

2.2.6 Full Maude

Full Maude is a system extension that supports the full syntax of Maude. It is writing in Maude and uses it is reflective capabilities.

Full Maude supports all the object-oriented features and also parameterized modules, theories, views and module expressions.

All of full Maude including grammar, user Interface and internal functionality has already been defined in Maude using Maude's reflective capabilities.

Full Maude sures is of use and maintenance and modification and even extension to new features and operations. Its competitive performance makes the user interaction reasonable.
All full-maude Modules must be included in parentheses.

omod ... endom for object-oriented modules.

(omod X is ... endom) .

2.2.7 Execution and searching with Maude

The rewriting rules are the fundamental units of execution in a Maude module. They interpret the simulated system's local activities and can be run indefinitely and continuously (at any moment).

Maude allows us to simulate the execution of such rewrites (through rewriting rules) or equational rewrites (by equations) in a M module by using the following two commands:

reduce and rewrite.

By applying the membership equations and axioms in a particular module, the reduction command (abbreviated as red) can decrease an initial term.

The syntax is as follows:

Reduce in module : term .

The rewrite (abbreviated as rew) and frewrite (abbreviated as frew) commands perform a single rewrite sequence (among numerous potential sequences) from an initially provided word according to the syntax:

rewrite in module : term .
frewrite in module : term .

The maude rew [n] command performs a maximum of n rewriting steps,

The frew command is similar to the previous one (but the application of the equation is not fair)
These commands allow you to rewrite an initial term using the supplied module's rules, equations, and membership axioms .

2.2.7.1 Formal analysis

Formal model checking on systems specified in Maude is carried out using tools available around the Maude system. Among these tools, we distinguish a tool for accessibility analysis (the search command).
Searching and model-checking to Analyse all possible all possible behaviors from an initial state, This command searches whether states corresponding to given patterns and satisfying certain conditions can be accessed from t0.
The execution of this command performs a deep traversal of the computational tree (reachability tree), generated during this search, in order to detect invariant violations in infinite state systems.

search startterm arrow pattern [such that cond] .

Where arrow is one of :

=>1 (reachable in one step)

=>* (reachable in 0 or more steps)

=>+ (reachable in 1 or more steps)

=>! ("Final state")

And pattern is a term (possible, with variables) and cond is a rewrite condition.

Limit the number of results: serach [n] and also limit the number of rewriting steps: search [n,d] .

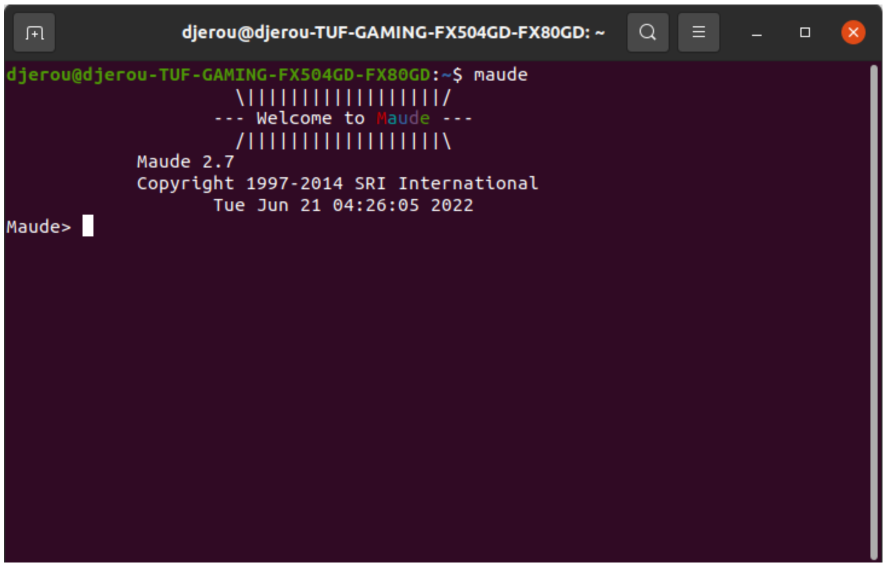
Show path n . Prints the path in state number n in the previous search,

and if we only need the list of the names of the rules in the path, we can use the command show path labels n .

2.2.7.2 Execution of Maude

We can start a session with Maude on UBUNTU 20.04 by clicking on the terminal after opening.

We run the maude command to display the main Maude window, which opens as shown in Figure:



```
djerou@djerou-TUF-GAMING-FX504GD-FX80GD:~$ maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.7
            Copyright 1997-2014 SRI International
                   Tue Jun 21 04:26:05 2022
Maude> 
```

Figure 2.8: Execution of Maude

The Maude system is fully operational and ready to take orders or modules.

During a session, the user interacts with the system by typing a request into the "Maude prompt." For example, if one desires to quit Maude:
Maude>quit

‘q’ can be used as an abbreviation for the command ’quit’. You may also insert modules and use other commands.

It is not practicable to input a module into the prompt; instead, the user can create one or more modules in a text file and have the file loaded into the system using the ’in’ or ’load’ commands .

Assuming the nat.maude file containing the BASIC-NAT module described as follows:

[My program]

```
fmod BASIC-NAT is   \\

  sort Nat .  \\

  op 0 : -> Nat [ctor] .  \\

  op s_ : Nat -> Nat [ctor] .  \\

  op _ + _ : Nat Nat -> Nat .  \\

  op max : Nat Nat -> Nat .  \\

  vars N M : Nat .  \\

  eq 0 + N = N .   \\

  eq s M + N = s (M+N) .  \\

  eq max(0, M) = M .  \\

  eq max(N, 0) = N .  \\

  eq max(s N, s M) = s max(N, M) .  \\

Endfm  \\
```

We can introduce it to the system by doing the following command:

```
Maude> in nat.maude
```

.

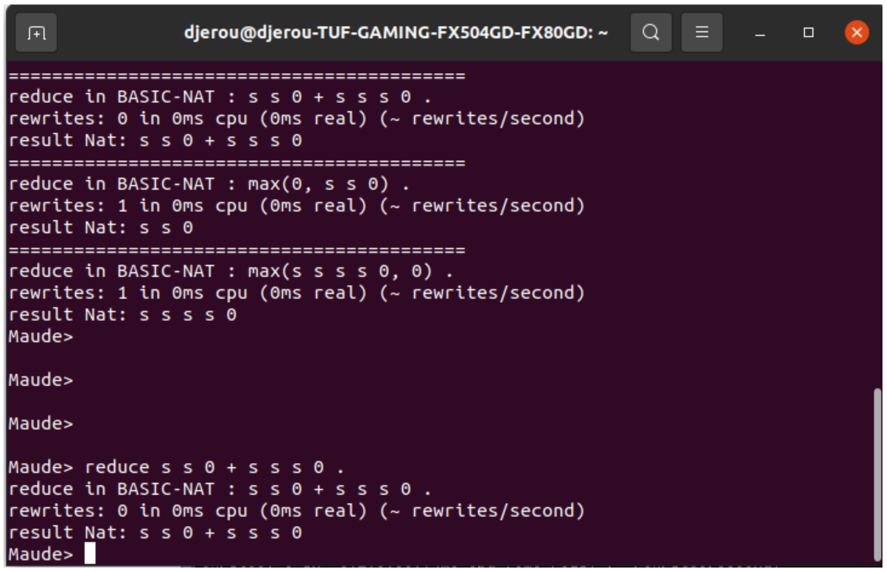
After entering the NAT module we can make an example, Running and validating the 're-duce' command returns a result as follows:

```
Maude>reduce in BASIC-NAT : s s 0 + s s s 0 .

rewrites: 0 in 0ms cpu (0ms real) (  rewrites/second)

result Nat: s s 0 + s s s 0
```

It is not required to state specifically the name of the module in which the term is shortened. The phrase is being shortened.

All commands that a module requires relate to the current module by default, otherwise the module by default, otherwise the module name must be explicitly specified.

The most recent module added or utilized is normally the current one, otherwise we may use the reduce command.



```
==========================================
reduce in BASIC-NAT : s s 0 + s s s s 0 .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s 0 + s s s s 0
==========================================
reduce in BASIC-NAT : max(0, s s 0) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s 0
==========================================
reduce in BASIC-NAT : max(s s s s 0, 0) .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s s s 0
Maude>

Maude>

Maude>

Maude> reduce s s 0 + s s s s 0 .
reduce in BASIC-NAT : s s 0 + s s s s 0 .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s s 0 + s s s s 0
Maude>
```

Figure 2.9: Executing the reduce command

All full-maude Modules must be included in parentheses if executed them.

(rew ....)

(search ....)

2.3 Conclusion

Throughout this chapter we presented the main concept of verification and validation, where we are concerned with the formal verification methods. Also, we defined Maude which is a formal specification and declarative programming language based on rewriting logic mathematical theory. In the next chapter, we will introduce the use of the Maude language in our implementation

Chapter 3

Specification and Implementation of Sagas using the Maude language

3.1 INTRODUCTION

As we mentioned in the first chapter, that Saga pattern is used to manage database transactions in microservices architecture.
In this chapter we aim to verify the Saga Orchestration Pattern with Maude language where we used a real example of an existing application.

3.2 FTGO Application Description

Food To Go, or briefly ***FTGO*** is an enterprise distributed applications that offers functions to facilitate Ordreding, ticketing, payment and delivery of food orders requested by online users using their PCs, tablets or Mobile phones.

The main purpose of the application is linking between Consumers, local restaurants and couriers. After a consumer place an order through the ***FTGO*** application in local restaurants, the order is coordinated between the restaurants and the couriers finishing by the coordination of the payment to both restaurants and couriers [Chris R][14].

Moreover, the application offers a secure medium to perform payments through payment methods and payments services. Architecturally, the applications is built upon the following components / services as it's depicted in **Figure 1.**:

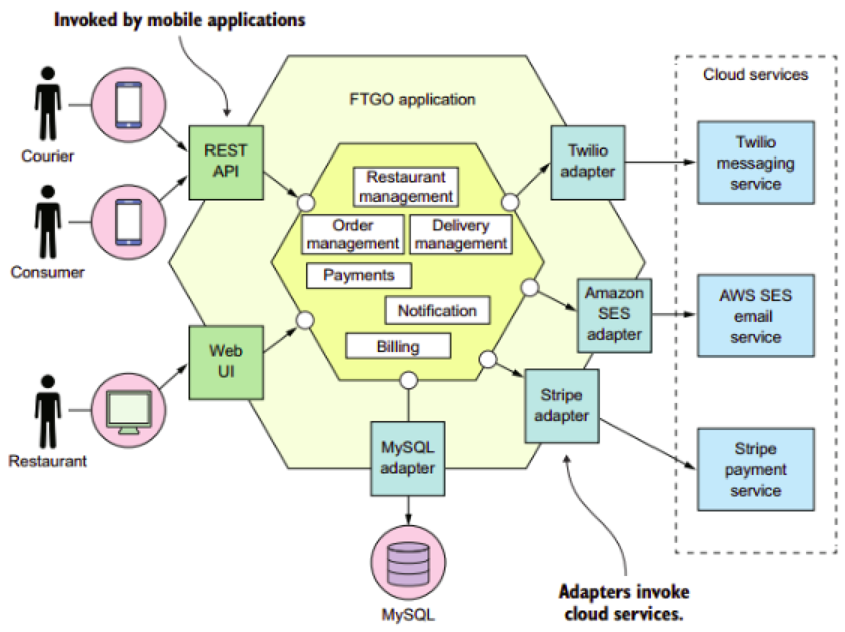


Figure 3.1: Component of the FTGO application

[14]

The microservice version of the application is presented in Figure 2 and the function of each service is described briefly below:

1. **Consumer service:** it is responsible for the creation of a pending order that has not been approved yet. [14]

2. **Order Service:** it is responsible for checking that the consumer is able to place this order. [14]

3. **Restaurant Service:** it is responsible of validation of the order and its details in order to create a pending ticket. [14]

4. **Kitchen service:** it is responsible for the authorization of the payment from the consumer's credit card. [14]

5. **Delivery Service:** it is responsible for switching the ticket state from pending to awaiting acceptance. [14]

6. **Notification service :** it is responsible of changing the order to approved state as a final step. [14]

7. **Accounting Service:** it is responsible for the authorization of the payment from the con-

sumer's credit card. [14]

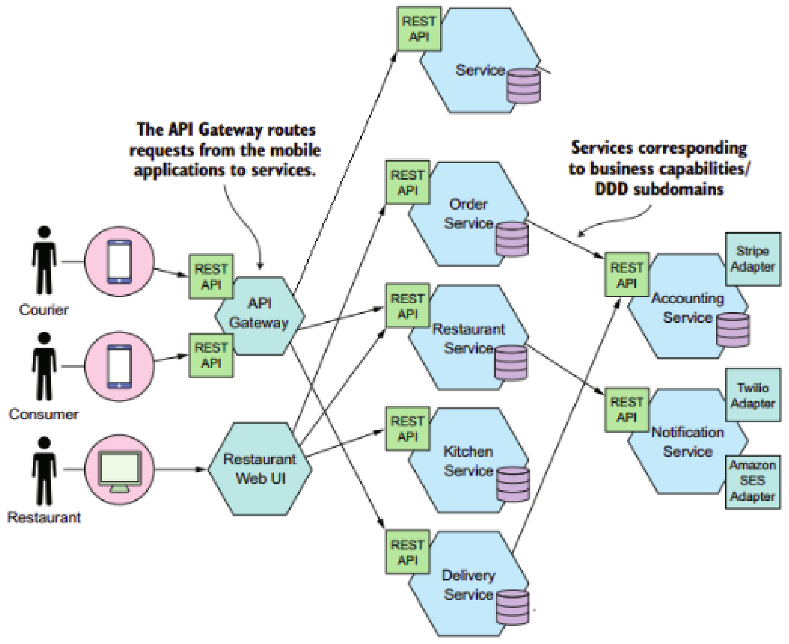


Figure 3.2: Microservice architecture of the FTGO Application.

[14]

In our work, we focus on the verification of Saga using Maude System. Thus, we present in the next section is a real example of Saga through the Create order function.

3.3 Create Order Saga:

The created Order Saga is depicted in Figure 3 [Chris Ridecharson] [14]. The Order Service realize the createOrder() operation using this saga.

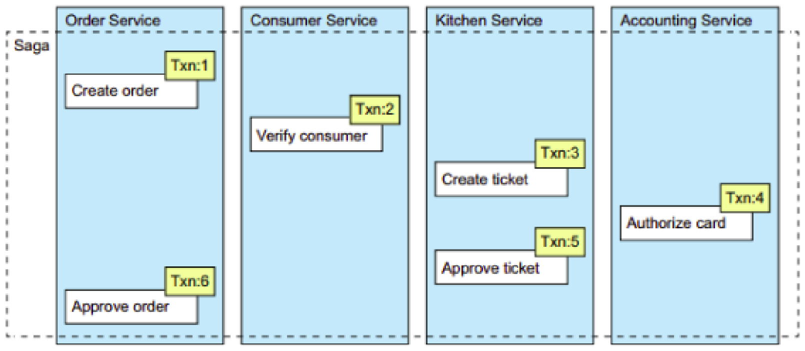


Figure 3.3: Create Order Saga: each operation is implemented by a saga that consists of local transactions in several services.

[14]

First, the local transaction is initiated by an external request. **Second,** the other local transactions **(subsequent operations)** are initiated in different services and are triggered by completion of the previous one forming a chain of executions.

3.4 Orchestration-based SAGA

Figure.4 represents the overall design of the create order saga in its orchestration based type.

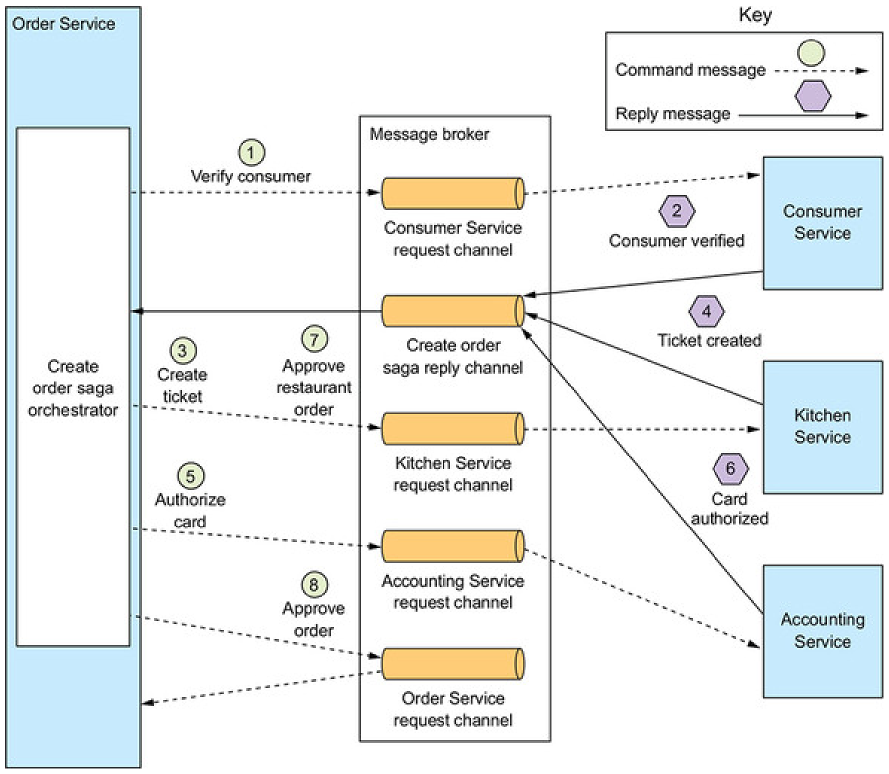


Figure 3.4: Implementing the Create Order Saga using orchestration. Order Service implements a saga orchestrator, which invokes the saga participants using asynchronous request/response. [14]

implements a saga orchestrator, which invokes the saga participants using asynchronous request/response.

Starting from figure.4 a sequence of events has been designed describing the mechanism of creating an order. Figure.5 demonstrates the case of all services approving the order.

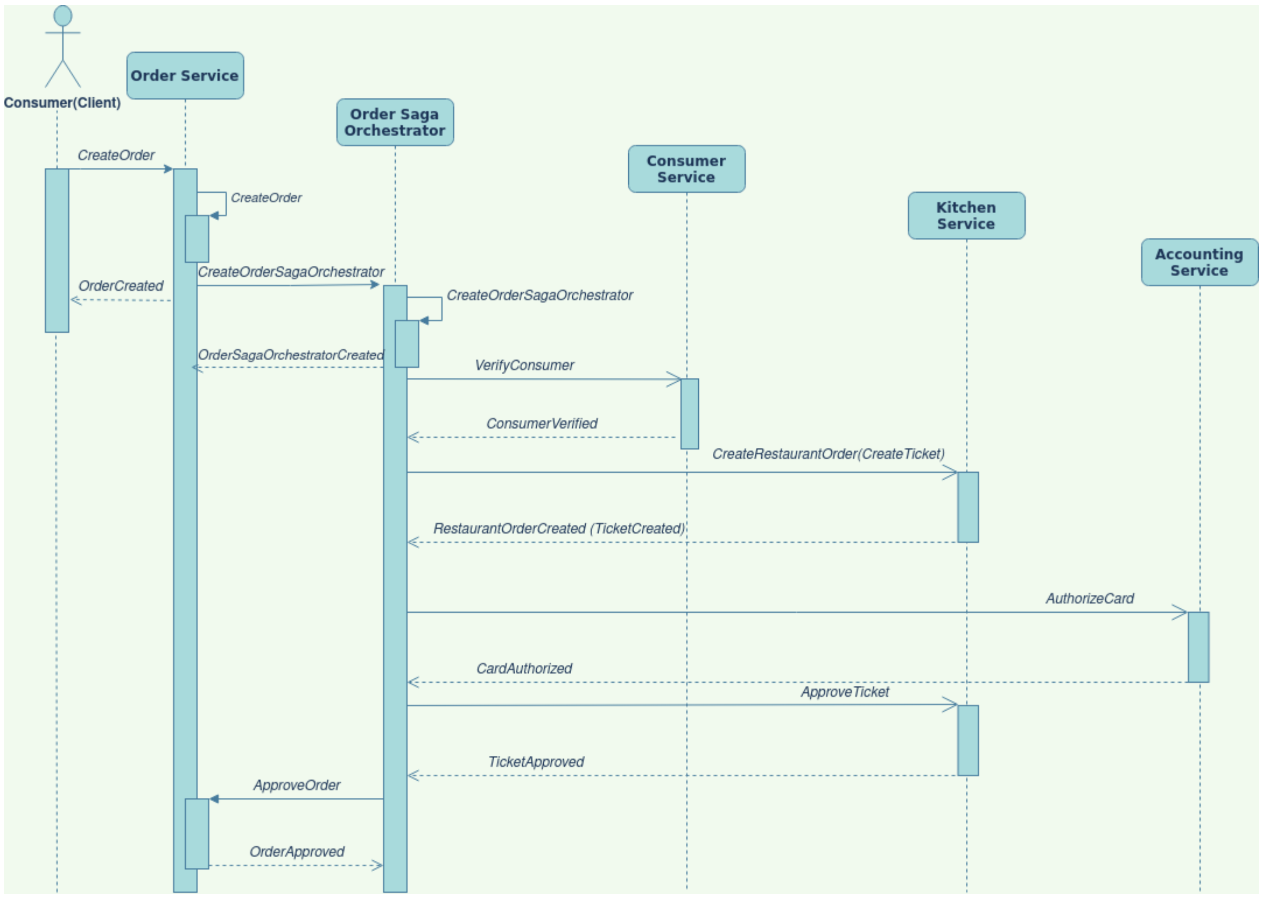


Figure 3.5: Sequence Diagram represents the sequence of events leading to the approval of the order service to order saga orchestrators.

Figure.6 tries to shed light on the case of one of the services failing and the system's response to that.

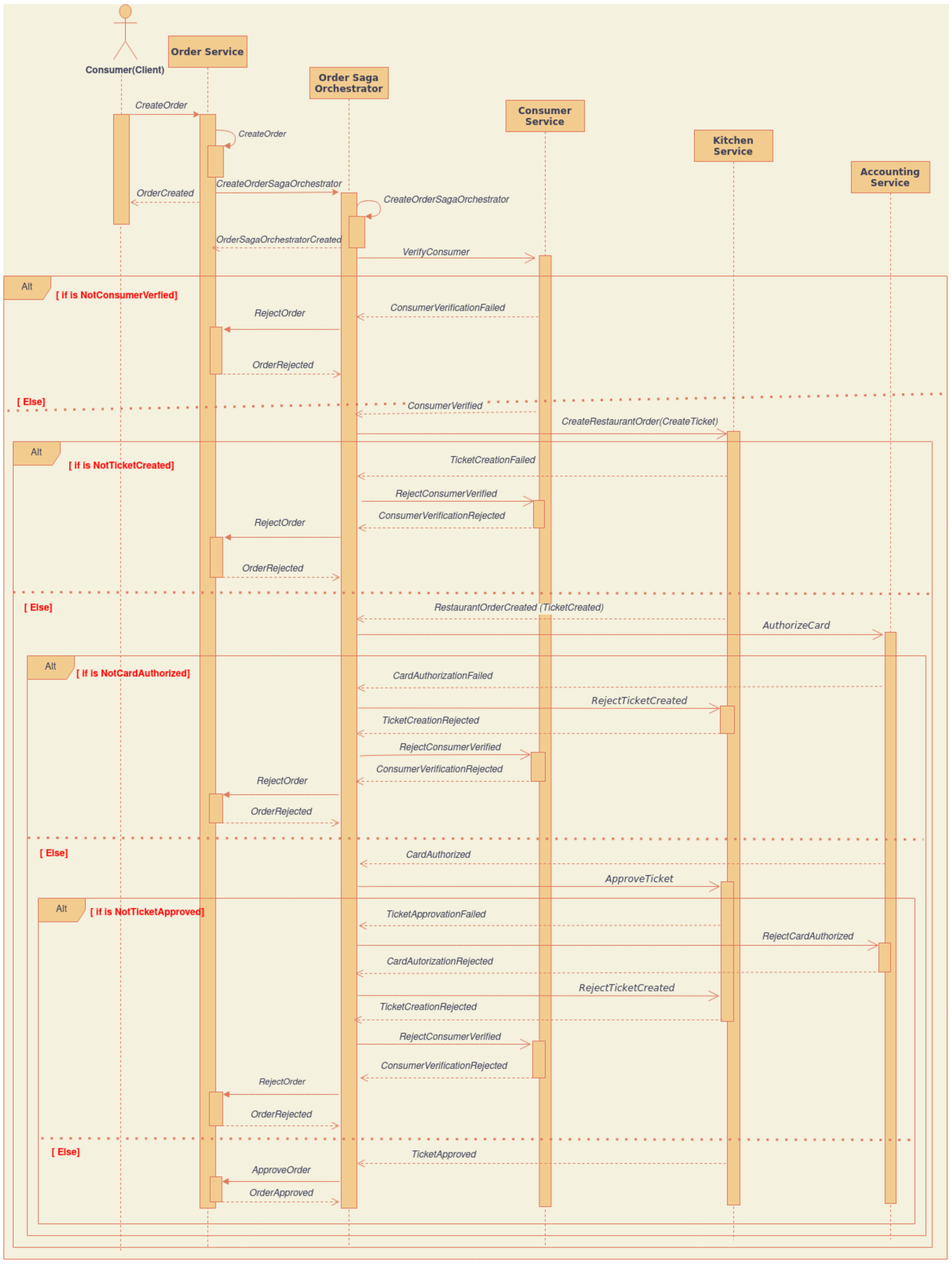


Figure 3.6: Sequence diagram represents the possibilities related to the failure of one of services and subsequent rollback order.

3.5 Maude Specification

We have four modules.

3.5.1 Module Service

Firstly, the services module is defined as **SERVICE-DEFINITION**. It is composed of class **SERVICE**, this class is formed of attributes, values, operations, sorts and messages.

```
(omod SERVICE-DEFINITION is
including MESSAGE-WRAPPER .
including MESSAGE-BROKER .

--- OidSet definition : pour definir un ensemble d'objets
sort OidSet . subsort Oid < OidSet .
op none : -> OidSet [ctor] .
op _;_ : OidSet OidSet -> OidSet [ctor assoc comm id: none] .

--- Message List declaration
sort MsgList .
subsort Msg < MsgList .
op noMsg : -> MsgList [ctor] .
op _//_ : MsgList MsgList -> MsgList [ctor assoc id: noMsg] .
sorts State ComponstionMode .
```

In the SERVICE-DEFINITION we needed both other modules like MESSAGE-WRAPPER, and MESSAGE-BROKER for including. We have defined MESSAGE-WRAPPER in this form :

```
--- Message wrapper
(omod MESSAGE-WRAPPER is including MESSAGE-CONTENT .
op msg_from_to_ : MsgContent Oid Oid -> Msg .
op msg_to_ : MsgContent Oid -> Msg .
op msg_from_ : MsgContent Oid -> Msg .
endom)
```

By this declaration form , sending any message is easier to identify the sender and receiver and the content of the message. In the MESSAGE-WRAPPER we needed other module named MESSAGE-CONTENT for including.

```
(mod MESSAGE-CONTENT is
sort MsgContent . --- message content, application-specific
endm)
```

3.5.1.1 Service Class Modeling

We have defined the class service with its attributes (isOrchestrator, approved, otherSERVICE, state, mode, OrchestrationStack, RollbackStack, recievedMsgs, MessageToSend and anyServFailed) and sorts (Bool, OidSet, State, ComponstionMode, MsgList and Msg) where the definition in the model is:

```
--- Definition of service Class that represents a microservice
class SERVICE | isOrchestrator : Bool, approved : Bool, otherSERVICE :
    OidSet, state : State,
mode : ComponstionMode, OrchestrationStack : MsgList, RollbackStack :
    MsgList,   recievedMsgs : MsgList, anyServFailed : Bool .
```

3.5.1.2 Attributes in SERVICE class

In this parts we have defined all the operators (sorts) and messages that the SERVICE class needs and uses.

```
---All the operators and messages needs and uses SERVICE class.
ops initial sended ready willRespond available done failed rollbacked : ->
     State [ctor] .
ops waiting : Oid -> State .
ops VerifyConsumer CreateTicket AuthorizeCard ApproveTicket ApproveOrder
    RejectOrder : -> MsgContent [ctor] . ---All the messages the
    Orchestrator can be sending to another service.
msgs ack noAck rollback : Msg -> Msg .   ---Answering Messages
ops normal  abnormal : -> ComponstionMode [ctor] .  ---Mode
ops ConsumerServ KitchenServ AccountingServ OrderServ : -> Oid [ctor] .
    ---Services Name
op orchSaga : -> Oid [ctor] .  ---Orchestrator Name

msg CreateOrder : Oid -> Msg .
```

we have defined the meaning of service class attributes in short as :

approved : “**main attribute**”, true if database components are approved.
isOrchestrator : true if a service is Orchestrator.
otherSERVICE : all other services in the system.
state : state in Saga Orchestrator.
mode: functions as normal automatically, but changes to abnormal if any service fails.
OrchestrationStack : this attribute handles all the requests sent by the orchestrator.

RollbackStack : the initial operations is null and it is specified to Orchestrator Saga, add message rollback if the service responds by Ack message, and use the messages adding if any service fails. The orchestrator sends a message rollback to services and responded with ack to reject any operation .
recievedMsgs : any messages the service received.
MessageToSend : all the reply messages that any service other than orchestrator can send for response to Orchestrator saga.
anyServFailed : true (in the orchestrator) if any service Failed.

3.5.2 Brokers Modeling

In this part, we are modeling the links of communication between services by creating class MESSAGE-BROKER where the only attribute in this class is MsgContentList :

```
--- Module Broker( Links)
(omod MESSAGE-BROKER is including MESSAGE-CONTENT .
sorts MsgContentList .
subsort MsgContent < MsgContentList .
subsort Msg < MsgContentList .
op nil : -> MsgContentList [ctor] .
op _::_ : MsgContentList MsgContentList -> MsgContentList [ctor assoc id:
   nil] .
op _to_ : Oid Oid -> Oid [ctor] . --- link names
op allto_ : Oid -> Oid [ctor] .
class MBroker | content : MsgContentList .
endom)
```

3.5.3 Messages Modeling

The third module is specified as Messages represents how sequence events happen into services. This Message is represented by rewriting logic rules.

```
 --- Modeling sending message from orchestrator to a service

vars O O'  : Oid .    --- O and O' represented a services with Oid type.
var OS : OidSet .    --- OS represented all of service (other Services).
var S  S' : State .      --- S and S' represented state of the service.
vars ML MLC MLC' : MsgList .        --- ML: Message list of attribute
   RollbackStack , MLC and MLC' represented msg list content.
```

```
var MC : MsgContent .
vars MCL MCL' : MsgContentList .

--- rewriting rules
rl [ Starting-Orch] :
CreateOrder(O)  < O : SERVICE | isOrchestrator : true, state : initial >
=>
< O : SERVICE | isOrchestrator : true, state : ready > .

```

In the first rewriting rule we have showed how the Orchestrator started and changed your state from initial to ready .

```
rl [ Sending-Message] :
< O : SERVICE | isOrchestrator : true, approved : false, otherSERVICE : OS
     , state : ready , mode : normal ,
OrchestrationStack : (msg MC to O') // ML , RollbackStack : noMsg ,
    anyServFailed : false >
< O to O' : MBroker | content : MCL >
=>
< O to O' : MBroker | content : MCL :: (msg MC to O') >
< O : SERVICE | isOrchestrator : true, approved : false, otherSERVICE : OS
     , state : waiting(O') ,
mode : normal , OrchestrationStack : ML, RollbackStack : noMsg,
    anyServFailed : false > .

```

After the Orchestrator starting. The saga Orchestrator sends a request message by label named (Sending-Message) to another service .

```
rl [ Read-Message] :
< O' : SERVICE | isOrchestrator : false, state : initial, recievedMsgs :
    ML >
< O to O' : MBroker | content : ( msg MC to O' ) :: MCL >
=>
< O to O' : MBroker | content : MCL >
< O' : SERVICE | isOrchestrator : false, state : willRespond, recievedMsgs
     : ML // (msg MC from O) > .

```

After that, the destination service receives and readies the order via a message broker from service Orchestrator. This rule contains a label named (Read-Message).

After the destination service reads the order message , it replies by response message to the orchestrator service.

3.5.3.1 Answering to Sending Message

In this rule, a service answers to a Message- request with ack, ack answer represents a rule containing a label named (Ack$_{T}o_{R}eception$).

```
--- Responding with Ok to the Orchestrator
rl [ Ack_To_Reception] :
< O' : SERVICE | isOrchestrator : false , state : willRespond , recievedMsgs
     : ML // (msg MC from O) >
< allto O : MBroker | content : MCL >
=>
< allto O : MBroker | content : MCL :: ack((msg MC to O')) >
< O' : SERVICE | isOrchestrator : false , state : done , recievedMsgs : ML
    // (msg MC from O) > .
```

But, in this rule, a service answers to a Message- request with NotAck, NotAck answer represents a rule containing a label named (NotAck$_{T}o_{R}eception$).

```
--- Responding with NotOk to the Orchestrator
rl [ NotAck_To_Reception] :
< O' : SERVICE | isOrchestrator : false , state : willRespond , recievedMsgs
     : ML // (msg MC from O) >
< allto O : MBroker | content : MCL >
=>
< allto O : MBroker | content : MCL :: noAck((msg MC to O')) >
< O' : SERVICE | isOrchestrator : false , state : done , recievedMsgs : ML
    // (msg MC from O) >
```

3.5.3.2 Receiving the Answers Message

Saga Orchestrator receives answers to his request :

```
--- Orechestrator receives the ack
rl [ Ack_Reception] :
< allto O : MBroker | content : ack(msg MC to O') :: MCL >
< O : SERVICE | isOrchestrator : true , approved : false , state : waiting(O
    ') , mode : normal ,
OrchestrationStack : MLC', RollbackStack : ML , anyServFailed : false >
=>
< allto O : MBroker | content : MCL >
< O : SERVICE | isOrchestrator : true, approved : false, state : ready ,
    mode : normal ,
OrchestrationStack : MLC', RollbackStack : rollback((msg MC to O')) // ML,
     anyServFailed : false > .
```

In this part we have found the Orchestrator received Ack response. If saga orchestrator receives answer with Ack from another service, it completes sending the orders message to the rest of services like as sequence events with mode normal .

```
--- if orchestrator recieves a noAck then it balances to the Abnormal mode
rl [ NoAck_Reception] :
< allto O : MBroker | content : noAck(msg MC to O') :: MCL >
< O : SERVICE | isOrchestrator : true, approved : false, state : waiting(O
    ') , mode : normal ,
OrchestrationStack : MLC', RollbackStack : ML, anyServFailed : false >
=>
< allto O : MBroker | content : MCL >
< O : SERVICE | isOrchestrator : true, approved : false, state : ready ,
    mode : abnormal ,
OrchestrationStack : MLC', RollbackStack : rollback((msg MC to O')) // ML,
     anyServFailed : true > .
```

But this parts represent the Orchestrator recieved NotAck response. If saga orchestrator receives answer NotAck from another service, it changes the mode from normal to abnormal and it sends a reject message (Rollback message) to services which answered with ack until it sends a reject message to Order Service.

```
rl [RollBack_Sending] :
---Send the rollback message
< O : SERVICE | isOrchestrator : true, approved : false, otherSERVICE : OS
     , state : ready , mode : abnormal ,
RollbackStack : rollback((msg MC to O')) // ML, anyServFailed : false >
< O to O' : MBroker | content : MCL >
=>
< O to O' : MBroker | content : MCL :: rollback((msg MC to O')) >
< O : SERVICE | isOrchestrator : true, approved : false, otherSERVICE : OS
     , state : ready , mode : abnormal ,
RollbackStack : noMsg, anyServFailed : true > .
```

This rule represents Rollback Message when sending from Orchestrator to sercice which responded with Ack Message.

```
--- Recieve the rollback action
rl [ RollBack_Reception ] :
```

```
< O' : SERVICE | isOrchestrator : false, state : done, recievedMsgs : ML >
< O to O' : MBroker | content :  rollback((msg MC to O')) :: MCL  >
=>
< O to O' : MBroker | content :  MCL  >
< O' : SERVICE | isOrchestrator : false, state : rollbacked, recievedMsgs
   : ML // rollback((msg MC from O)) > .
```

By this rule we have represented how to receive the rollback action from another service to Orchestrator.

We need a conditional rule for finishing all the steps in normal or abnormal mode , in this rule we have represented :

```
--- Finishing in normal mode
ceq < O : SERVICE | isOrchestrator : true, approved : false, state : S ,
    mode : normal,
OrchestrationStack : ML , anyServFailed : false > =
< O : SERVICE | isOrchestrator : true, approved : false, state : done ,
    mode : normal,
OrchestrationStack : ML , anyServFailed : false > if ML == noMsg .
```

For the finishing in normal mode rule we have found just the state variable S to change to done state and anyServFailed had false value .

```
--- Finishing in Abnormal mode
ceq < O : SERVICE | isOrchestrator : true, approved : false, state : S ,
    mode : abnormal,
RollbackStack : ML , anyServFailed : true > =
< O : SERVICE | isOrchestrator : true, approved : false, state : done ,
    mode : abnormal,
RollbackStack : ML , anyServFailed : true > if ML = noMsg .
```

But for the finishing in abnormal mode rule we have found the state S to change to done state too but for anyServFailed had true value.

3.5.4 Initial Configuration

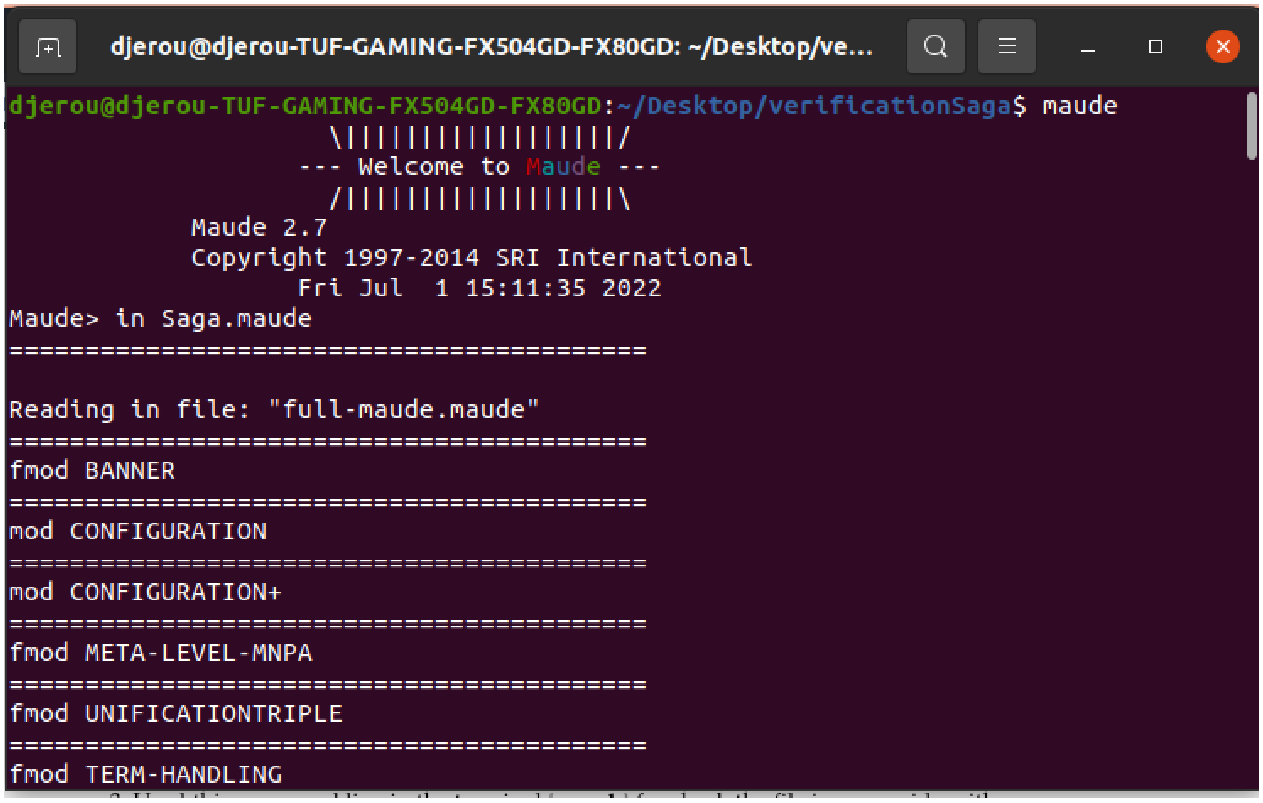
This module of the Configuration is represented like an example allowing testing our programs . In this part exactly, we used and tested rewriting rules .Also, in this part of configuration we are starting with part initialization of declaration of a services in initial state . We have four services defining (" ConsumerServ ; KitchenServ ; AccountingServ ; OrderServ") and service

Orchestrating names orchSaga. For linking between any service and saga orchsetrator, we used MBroker class.

```
op initState : -> Configuration .  ---this is operator represents a
   Configuration.
eq initState = CreateOrder(orchSaga) < orchSaga : SERVICE | isOrchestrator
    : true, approved : false,
otherSERVICE : ( ConsumerServ ; KitchenServ ; AccountingServ ; OrderServ )
   , state : initial, mode : normal,
OrchestrationStack : (msg (VerifyConsumer) to (ConsumerServ))//(msg (
   CreateTicket) to (KitchenServ))//(msg (AuthorizeCard) to (
   AccountingServ))//(msg (ApproveTicket) to (KitchenServ))//(msg (
   ApproveOrder) to (OrderServ)), RollbackStack : noMsg, recievedMsgs :
   noMsg, anyServFailed : false >
--- Consumer Service
< ConsumerServ : SERVICE | isOrchestrator : false, approved : false,
   otherSERVICE : ( orchSaga ; KitchenServ ; AccountingServ ; OrderServ ),
    state : initial, mode : normal, OrchestrationStack : noMsg ,
   RollbackStack : noMsg, recievedMsgs : noMsg, anyServFailed : false >
--- Kitechen Service
< KitchenServ : SERVICE | isOrchestrator : false, approved : false,
   otherSERVICE : ( orchSaga ; ConsumerServ ; AccountingServ ; OrderServ )
   , state : initial, mode : normal, OrchestrationStack : noMsg ,
   RollbackStack : noMsg, recievedMsgs : noMsg, anyServFailed : false >
--- Accounting Service
< AccountingServ : SERVICE | isOrchestrator : false, approved : false,
otherSERVICE : ( orchSaga ; KitchenServ ; ConsumerServ ; OrderServ ),
   state : initial, mode : normal,
OrchestrationStack : noMsg , RollbackStack : noMsg, recievedMsgs : noMsg,
   anyServFailed : false >
--- Order Service
< OrderServ : SERVICE | isOrchestrator : false, approved : false,
   otherSERVICE : ( orchSaga ; ConsumerServ ; AccountingServ ; KitchenServ
    ), state : initial, mode : normal, OrchestrationStack : noMsg ,
   RollbackStack : noMsg, recievedMsgs : noMsg, anyServFailed : false >
--- brokers
< ((orchSaga) to (ConsumerServ)) : MBroker | content : nil >   --- broker
   between Orchestrator and Consumer Service
< ((orchSaga) to (KitchenServ)) : MBroker | content : nil >    --- broker
   between Orchestrator and Kitchen Service
< ((orchSaga) to (AccountingServ)) : MBroker | content : nil >  --- broker
    between Orchestrator and Accounting Service
< ( allto (orchSaga)) : MBroker | content : nil > .             --- broker
    from any service to the Orechestrator
endom)
```

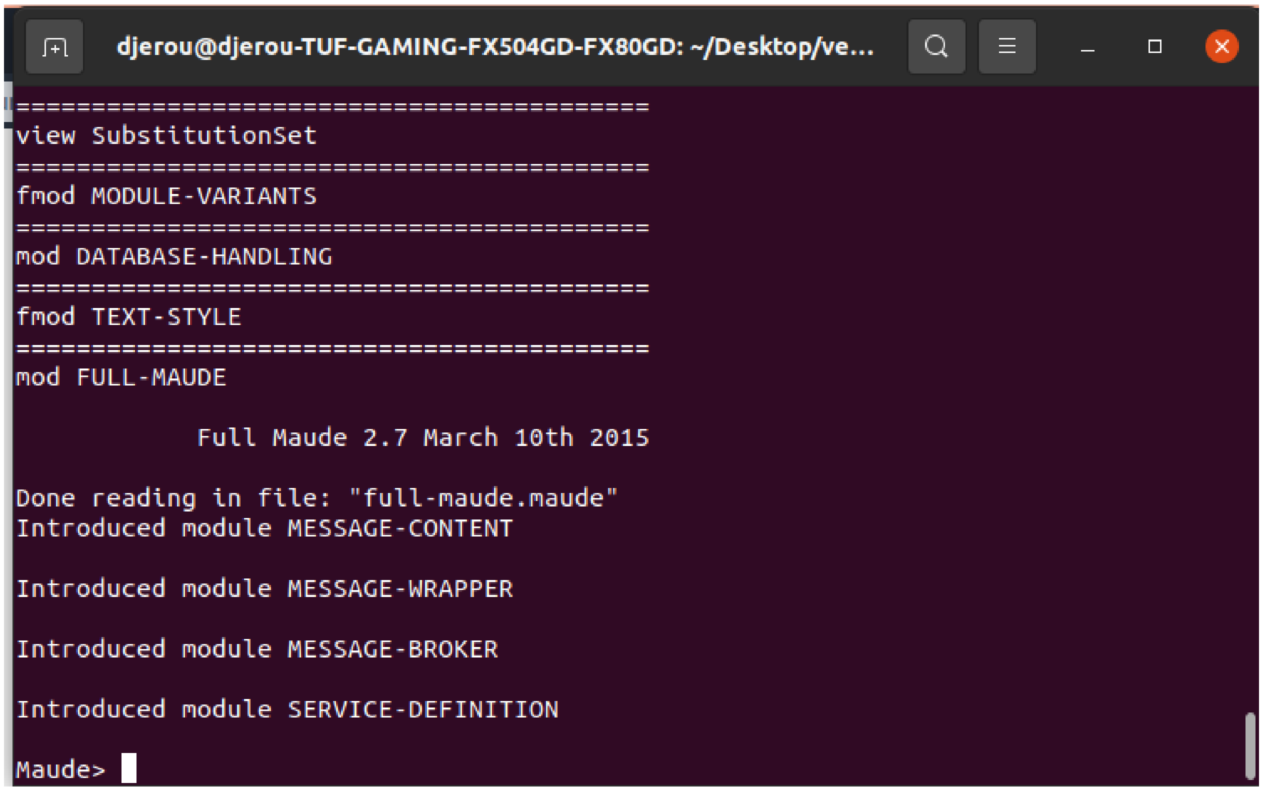
3.5.5 Execution Commands and Results

1. For executing this model , we used **in nameFile.maude** . This command line is used for reading the file in maude and verify the syntax of file .



```
djerou@djerou-TUF-GAMING-FX504GD-FX80GD: ~/Desktop/ve...
djerou@djerou-TUF-GAMING-FX504GD-FX80GD:~/Desktop/verificationSaga$ maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.7
            Copyright 1997-2014 SRI International
                   Fri Jul  1 15:11:35 2022
Maude> in Saga.maude
==========================================

Reading in file: "full-maude.maude"
==========================================
fmod BANNER
==========================================
mod CONFIGURATION
==========================================
mod CONFIGURATION+
==========================================
fmod META-LEVEL-MNPA
==========================================
fmod UNIFICATIONTRIPLE
==========================================
fmod TERM-HANDLING
```

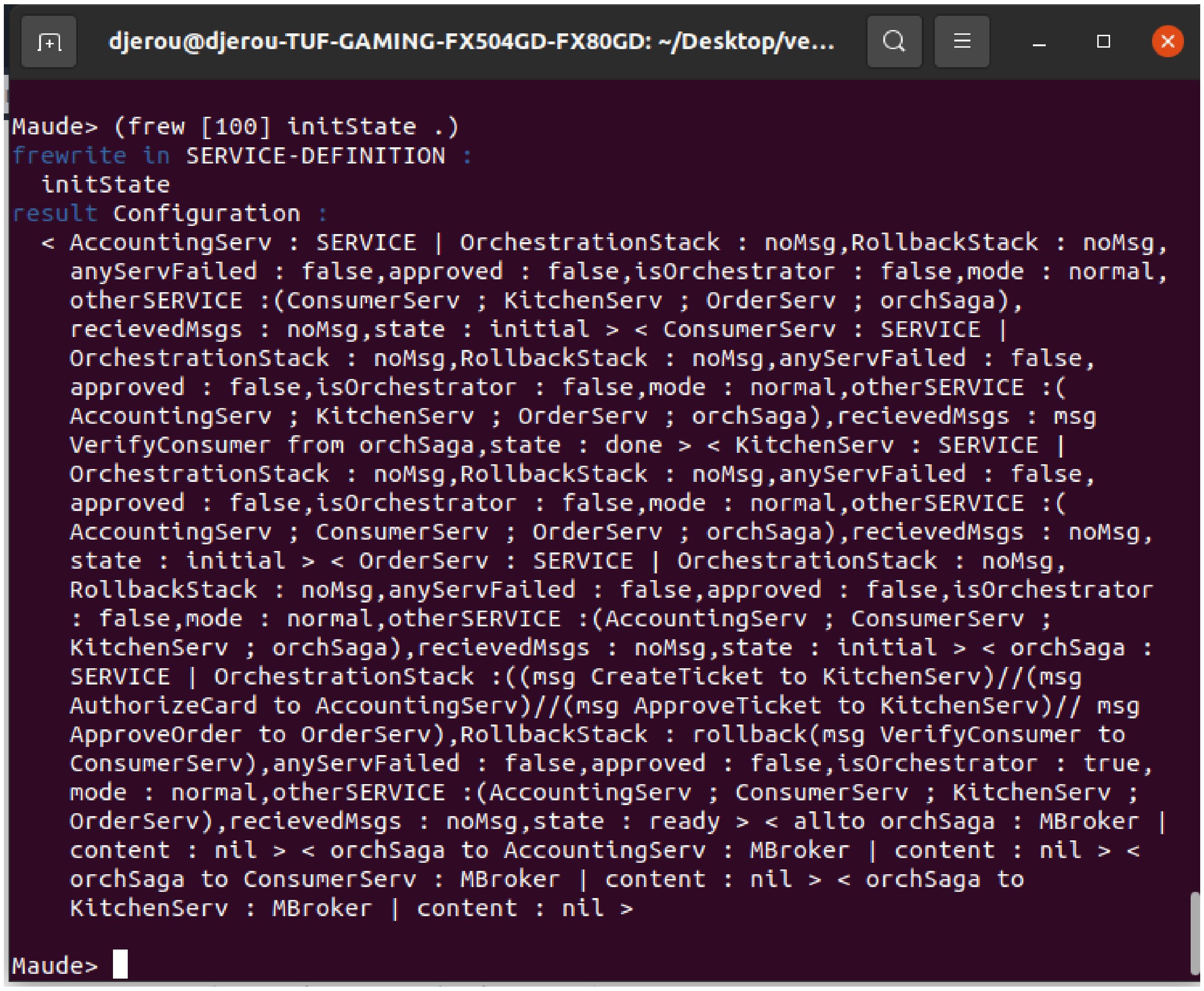
Figure 3.7: Reading of the Saga file maude by command in



```
==========================================
view SubstitutionSet
==========================================
fmod MODULE-VARIANTS
==========================================
mod DATABASE-HANDLING
==========================================
fmod TEXT-STYLE
==========================================
mod FULL-MAUDE

            Full Maude 2.7 March 10th 2015

Done reading in file: "full-maude.maude"
Introduced module MESSAGE-CONTENT

Introduced module MESSAGE-WRAPPER

Introduced module MESSAGE-BROKER

Introduced module SERVICE-DEFINITION

Maude>
```

Figure 3.8: Result of the execution in command

2. After that, we used **(frew [100] initState .)** command line in the terminal to try the execution of the model.

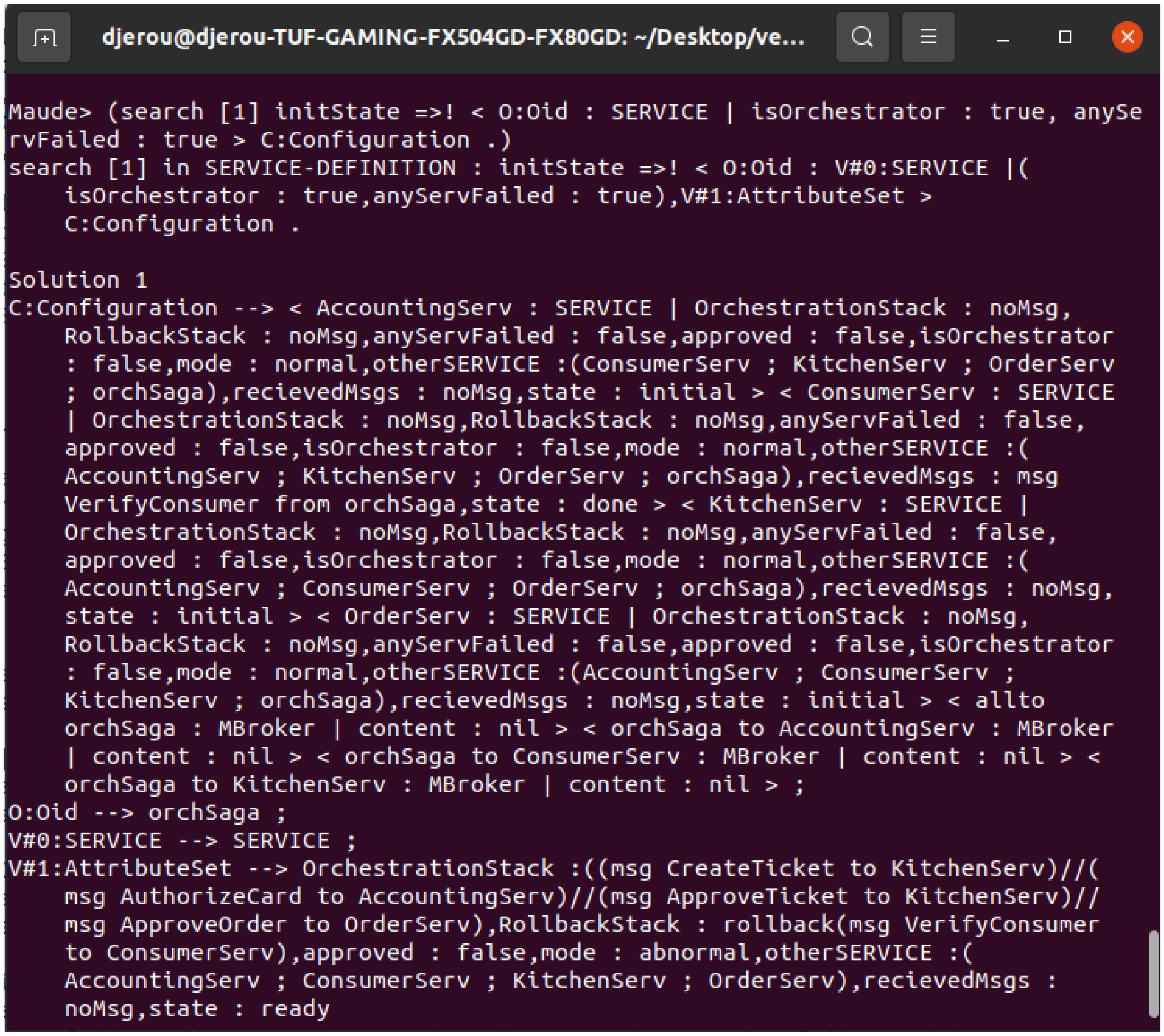


```
Maude> (frew [100] initState .)
frewrite in SERVICE-DEFINITION :
  initState
result Configuration :
  < AccountingServ : SERVICE | OrchestrationStack : noMsg,RollbackStack : noMsg,
    anyServFailed : false,approved : false,isOrchestrator : false,mode : normal,
    otherSERVICE :(ConsumerServ ; KitchenServ ; OrderServ ; orchSaga),
    recievedMsgs : noMsg,state : initial > < ConsumerServ : SERVICE |
    OrchestrationStack : noMsg,RollbackStack : noMsg,anyServFailed : false,
    approved : false,isOrchestrator : false,mode : normal,otherSERVICE :(
    AccountingServ ; KitchenServ ; OrderServ ; orchSaga),recievedMsgs : msg
    VerifyConsumer from orchSaga,state : done > < KitchenServ : SERVICE |
    OrchestrationStack : noMsg,RollbackStack : noMsg,anyServFailed : false,
    approved : false,isOrchestrator : false,mode : normal,otherSERVICE :(
    AccountingServ ; ConsumerServ ; OrderServ ; orchSaga),recievedMsgs : noMsg,
    state : initial > < OrderServ : SERVICE | OrchestrationStack : noMsg,
    RollbackStack : noMsg,anyServFailed : false,approved : false,isOrchestrator
    : false,mode : normal,otherSERVICE :(AccountingServ ; ConsumerServ ;
    KitchenServ ; orchSaga),recievedMsgs : noMsg,state : initial > < orchSaga :
    SERVICE | OrchestrationStack :((msg CreateTicket to KitchenServ)//(msg
    AuthorizeCard to AccountingServ)//(msg ApproveTicket to KitchenServ)// msg
    ApproveOrder to OrderServ),RollbackStack : rollback(msg VerifyConsumer to
    ConsumerServ),anyServFailed : false,approved : false,isOrchestrator : true,
    mode : normal,otherSERVICE :(AccountingServ ; ConsumerServ ; KitchenServ ;
    OrderServ),recievedMsgs : noMsg,state : ready > < allto orchSaga : MBroker |
    content : nil > < orchSaga to AccountingServ : MBroker | content : nil > <
    orchSaga to ConsumerServ : MBroker | content : nil > < orchSaga to
    KitchenServ : MBroker | content : nil >

Maude>
```

Figure 3.9: Result of the execution frew command

3. Used this command line in the terminal **search** for check the file in many side with many ways . In the first example of the execution we used this command **search** this way for Check if we can arrive to a compensation case from the initial one .

```
(search [1] initState =>! < O:Oid : SERVICE | isOrchestrator : true,
    anyServFailed : true > C:Configuration .)     --- search a fault

```

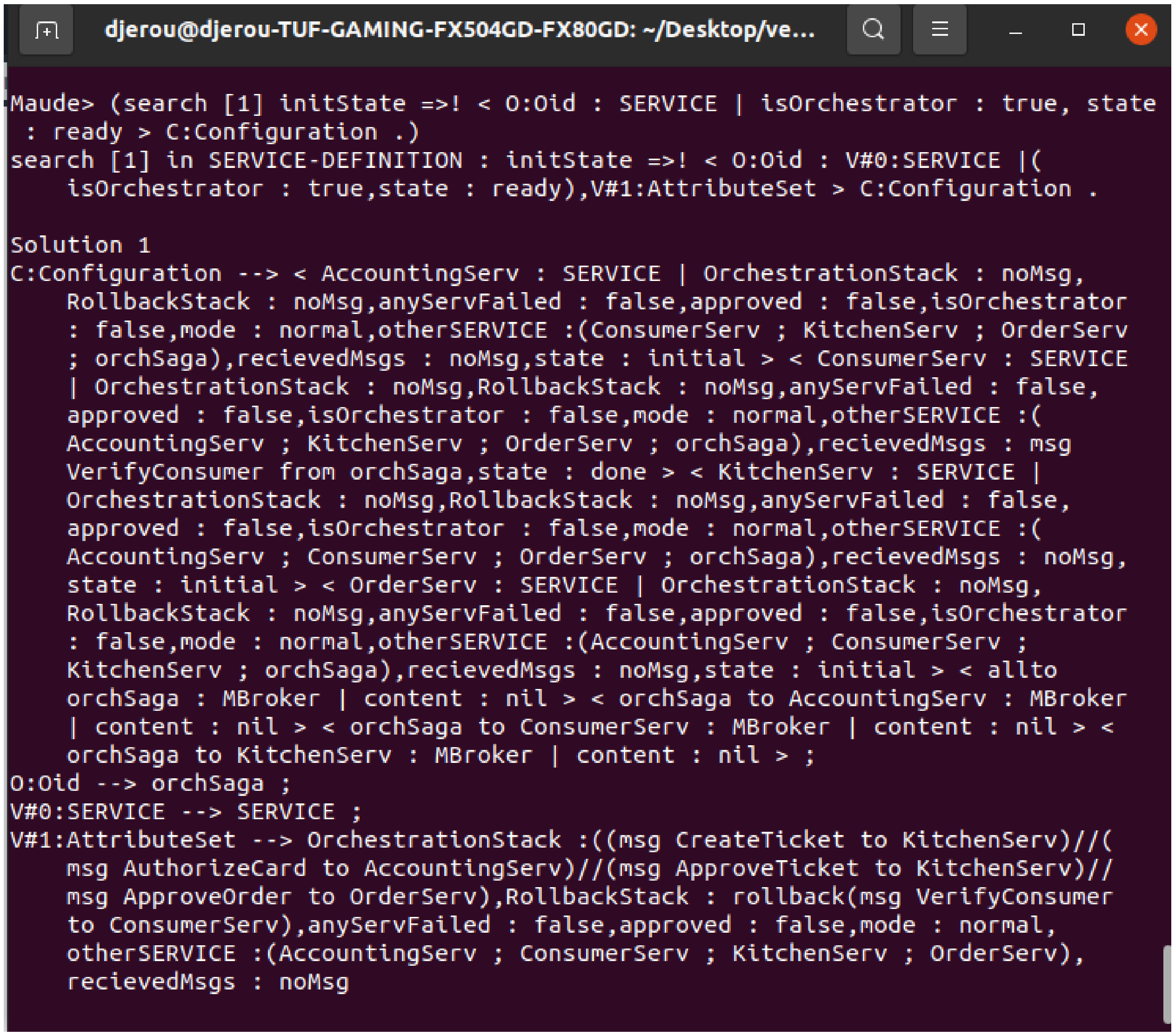


```
Maude> (search [1] initState =>! < O:Oid : SERVICE | isOrchestrator : true, anySe
rvFailed : true > C:Configuration .)
search [1] in SERVICE-DEFINITION : initState =>! < O:Oid : V#0:SERVICE |(
    isOrchestrator : true,anyServFailed : true),V#1:AttributeSet >
    C:Configuration .

Solution 1
C:Configuration --> < AccountingServ : SERVICE | OrchestrationStack : noMsg,
    RollbackStack : noMsg,anyServFailed : false,approved : false,isOrchestrator
    : false,mode : normal,otherSERVICE :(ConsumerServ ; KitchenServ ; OrderServ
    ; orchSaga),recievedMsgs : noMsg,state : initial > < ConsumerServ : SERVICE
    | OrchestrationStack : noMsg,RollbackStack : noMsg,anyServFailed : false,
    approved : false,isOrchestrator : false,mode : normal,otherSERVICE :(
    AccountingServ ; KitchenServ ; OrderServ ; orchSaga),recievedMsgs : msg
    VerifyConsumer from orchSaga,state : done > < KitchenServ : SERVICE |
    OrchestrationStack : noMsg,RollbackStack : noMsg,anyServFailed : false,
    approved : false,isOrchestrator : false,mode : normal,otherSERVICE :(
    AccountingServ ; ConsumerServ ; OrderServ ; orchSaga),recievedMsgs : noMsg,
    state : initial > < OrderServ : SERVICE | OrchestrationStack : noMsg,
    RollbackStack : noMsg,anyServFailed : false,approved : false,isOrchestrator
    : false,mode : normal,otherSERVICE :(AccountingServ ; ConsumerServ ;
    KitchenServ ; orchSaga),recievedMsgs : noMsg,state : initial > < allto
    orchSaga : MBroker | content : nil > < orchSaga to AccountingServ : MBroker
    | content : nil > < orchSaga to ConsumerServ : MBroker | content : nil > <
    orchSaga to KitchenServ : MBroker | content : nil > ;
O:Oid --> orchSaga ;
V#0:SERVICE --> SERVICE ;
V#1:AttributeSet --> OrchestrationStack :((msg CreateTicket to KitchenServ)//(
    msg AuthorizeCard to AccountingServ)//(msg ApproveTicket to KitchenServ)//
    msg ApproveOrder to OrderServ),RollbackStack : rollback(msg VerifyConsumer
    to ConsumerServ),approved : false,mode : abnormal,otherSERVICE :(
    AccountingServ ; ConsumerServ ; KitchenServ ; OrderServ),recievedMsgs :
    noMsg,state : ready
```

Figure 3.10: Result of the execution search command for Check if we can arrive to a compensation case from the initial one .

And in the second example of the execution we used **search** command with another way for Check if all.

```
(search [1] initState =>! < O:Oid : SERVICE | isOrchestrator : true, state
    : ready > C:Configuration .)    --- search if service orch change to
  state ready
```

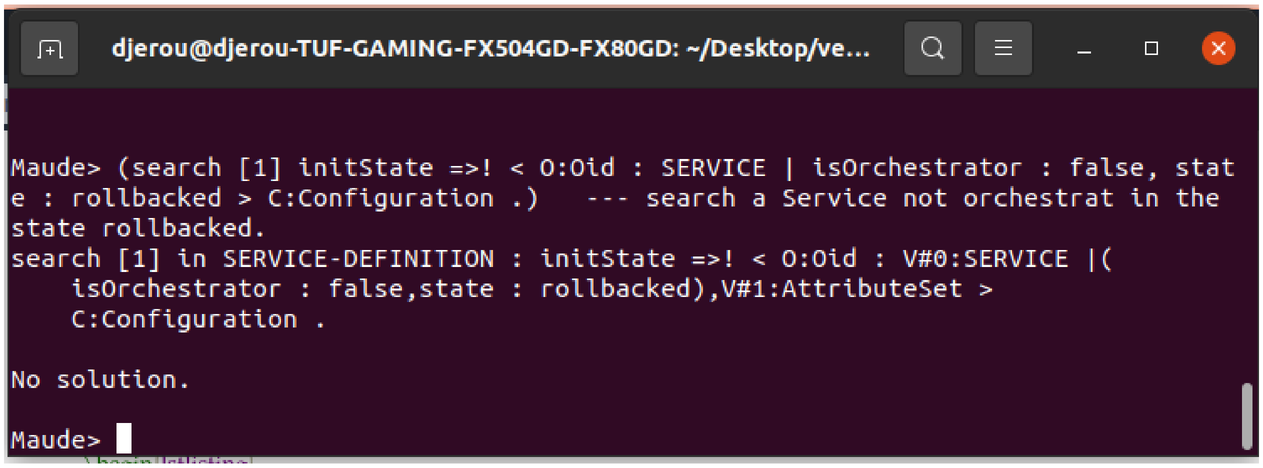


```
Maude> (search [1] initState =>! < O:Oid : SERVICE | isOrchestrator : true, state
 : ready > C:Configuration .)
search [1] in SERVICE-DEFINITION : initState =>! < O:Oid : V#0:SERVICE |(
    isOrchestrator : true,state : ready),V#1:AttributeSet > C:Configuration .

Solution 1
C:Configuration --> < AccountingServ : SERVICE | OrchestrationStack : noMsg,
    RollbackStack : noMsg,anyServFailed : false,approved : false,isOrchestrator
    : false,mode : normal,otherSERVICE :(ConsumerServ ; KitchenServ ; OrderServ
    ; orchSaga),recievedMsgs : noMsg,state : initial > < ConsumerServ : SERVICE
    | OrchestrationStack : noMsg,RollbackStack : noMsg,anyServFailed : false,
    approved : false,isOrchestrator : false,mode : normal,otherSERVICE :(
    AccountingServ ; KitchenServ ; OrderServ ; orchSaga),recievedMsgs : msg
    VerifyConsumer from orchSaga,state : done > < KitchenServ : SERVICE |
    OrchestrationStack : noMsg,RollbackStack : noMsg,anyServFailed : false,
    approved : false,isOrchestrator : false,mode : normal,otherSERVICE :(
    AccountingServ ; ConsumerServ ; OrderServ ; orchSaga),recievedMsgs : noMsg,
    state : initial > < OrderServ : SERVICE | OrchestrationStack : noMsg,
    RollbackStack : noMsg,anyServFailed : false,approved : false,isOrchestrator
    : false,mode : normal,otherSERVICE :(AccountingServ ; ConsumerServ ;
    KitchenServ ; orchSaga),recievedMsgs : noMsg,state : initial > < allto
    orchSaga : MBroker | content : nil > < orchSaga to AccountingServ : MBroker
    | content : nil > < orchSaga to ConsumerServ : MBroker | content : nil > <
    orchSaga to KitchenServ : MBroker | content : nil > ;
O:Oid --> orchSaga ;
V#0:SERVICE --> SERVICE ;
V#1:AttributeSet --> OrchestrationStack :((msg CreateTicket to KitchenServ)//(
    msg AuthorizeCard to AccountingServ)//(msg ApproveTicket to KitchenServ)//
    msg ApproveOrder to OrderServ),RollbackStack : rollback(msg VerifyConsumer
    to ConsumerServ),anyServFailed : false,approved : false,mode : normal,
    otherSERVICE :(AccountingServ ; ConsumerServ ; KitchenServ ; OrderServ),
    recievedMsgs : noMsg
```

Figure 3.11: Result of the execution search command for Check if service orchestrate change to state ready .

we used this mark ' ! ' in examples for **search** to arrive final state . In the third example we checked if any service other than the orchestrator in the state rollbacked .

```
(search [1] initState =>! < O:Oid : SERVICE | isOrchestrator : false,
   state : rollbacked > C:Configuration .)   --- search a Service not
   orchestrat in the state rollbacked.
```



```
Maude> (search [1] initState =>! < O:Oid : SERVICE | isOrchestrator : false, stat
e : rollbacked > C:Configuration .)   --- search a Service not orchestrat in the
state rollbacked.
search [1] in SERVICE-DEFINITION : initState =>! < O:Oid : V#0:SERVICE |(
    isOrchestrator : false,state : rollbacked),V#1:AttributeSet >
    C:Configuration .

No solution.

Maude>
```

Figure 3.12: Result of the execution search command for Check if any service not orchestrate in the state rollbacked .

We can use too all of command lines execution and checking in the same file of code source. after that we used **in** or **load** commands in the terminal for reading the file.maude .

3.6 CONCLUSION

During this chapter, we described the realization of our verification, in which we presented the application using ***FTGO*** and the create order scenario, then we introduced this scenario with Orchestration-based SAGA, after that we presented our solution of the scenario verification using Maude language and the obtained results.

General Conclusion

The microservice based architecture embodies the concept of a distributed system since that it includes small services which communicate with each other using network. This architecture provides several benefits, but it contains several challenges.

One of the most difficult tasks in microservices is handling the data transactions between multiple services. Where database per service enhances the concept of independablitity of services, but it requires a distributed transaction management. This last one must ensure the correctness of a transaction where all microservices must complete the individual local transactions.

The widely used pattern to implement distributed transactions is Two-Phase Commit which can be employed in a micro-service architecture to manage distributed transactions. But this protocol has limitation such as the use of coordinator that might lead to performance issues in a micro-service-based architecture involving multiple services. The saga pattern is emerged to manage data consistency across micro-services where this pattern produces a set of transactions that update microservices sequentially and locally.

In this work, we have been interested in the verification of the saga pattern and in particular the orchestration-based saga where we used the Maude language to verify this pattern and to confirm the correctness achievement of a given transaction across multiple micro-services.

Bibliography

[1] What is a distributed transaction? URL https://hazelcast.com/glossary/distributed-transaction/.

[2] Advant and disadv distributed database. URL https://www.exploredatabase.com/2014/08/advantages-and-disadvantages-of-distributed-databases.html.

[3] Distributed dbms - distributed databases. URL https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_databases.htm.

[4] *Databases Distributed Database.*

[5] Types of arch distributed database. URL https://www.javatpoint.com/types-of-databases.

[6] Saga distributed transactions pattern. URL https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga.

[7] two phase commit. URL https://arabicprogrammer.com/article/1421700130/.

[8] what is a distributed database, . URL https://phoenixnap.com/kb/distributed-database.

[9] Types of architectures in distributed database, . URL https://tutsmaster.org/types-of-architectures-in-distributed-database/.

[10] Types database, . URL https://www.w3cschoool.com/types-of-databases.

[11] *Databases.* Cengage Learning, 2015.

[12] baeldung. Saga pattern in microservices. URL https://www.baeldung.com/cs/saga-pattern-microservices.

[13] Rashmit Chawla. Companies using microservices. URL https://www.qaoncloud.com/microservices-test-automation/.

[14] Richardson Chris. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[15] S. Parker Deborah Morley, Charles. *Databases and Database Management Systems*. Cengage Learning, 2015.

[16] Chameera Dulanga. How to use saga pattern in microservices. *bitsrc.io*. URL https://blog.bitsrc.io/how-to-use-saga-pattern-in-microservices-9eaadde79748.

[17] Jeremy H. 4 microservices examples: Amazon, netflix, uber, and etsy. 2021. URL https://blog.dreamfactory.com/microservices-examples/.

[18] Laura Shiff Jonathan Johnson. Examples of microservices architecture. URL https://www.bmc.com/blogs/microservices-architecture/.

[19] Subodh Kumar et al. Techniques and languages for software developement. *International Journal Engg Sci Adv Research*, pages 35–42, 2015.

[20] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. *MartinFowler. com*, 25:14–26, 2014.

[21] Microsoft. Benefits and drawbacks of microservices. URL https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.

[22] Peter Csaba Ölveczky. *Designing Reliable Distributed Systems*. Springer, 2017.

[23] Oracle. What is a database? URL https://www.oracle.com/database/what-is-database/.

[24] Mehmet Ozkaya. Database per service. *medium.com*. URL https://medium.com/design-microservices-architecture-with-patterns/the-database-per-service-pattern-9d511b882425.

[25] Richard Peterson. What is a database? definition, meaning, types with example. *Guru99.com*. URL https://www.guru99.com/introduction-to-database-sql.html.

[26] C Richardson and F Smith. Microservices: From design to deployment, nginx inc, san francisco, ca, usa, 2016.

[27] Feras Taleb. figure of microservice. URL https://feras.blog/microservices-architecture-to-be-or-not-to-be/.

[28] Parul Tomar et al. An overview of distributed databases. *International Journal of Information and Computation Technology*, 4(2):207–214, 2014.

[29] VMware. Microservices definition. URL https://avinetworks.com/glossary/microservice/.

[30] George Samaras Yousef J. Al-houmaily. two phase commit protocol. *Institute of Public Administration, Riyadh,Saudi Arabia, University of Cyprus, Nicosia, Cyprus*, 2015.