UNIVERSIDAD
NACIONAL
DE COLOMBIA

Formal verification of the high availability quality attribute for software systems based on microservices architectures

Camilo Andres Dajer Piñerez

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D. C., Colombia
2023

Formal verification of the high availability quality attribute for software systems based on microservices architectures

Camilo Andres Dajer Piñerez

Thesis presented as a partial requirement for the degree of:
Master in Systems Engineering and Computer Science
«Magíster en Ingeniería - Ingeniería de Sistemas y Computación»

Advised by:
Jeisson Andrés Vergara Vargas, M.Sc.

Research Field:
Software Enginnering
Research Group:
ColSWE - Software Enginnering

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D. C., Colombia
2023

Dedicated to my family.

Acknowledgments

Firstly, I would like to express my gratitude to the Universidad Nacional de Colombia for their efforts in topics related to software engineering. I am especially grateful to all the professors who played a vital role in my training process, particularly Professor Jeisson Vergara for his invaluable teaching of software architecture and his advice in completing this thesis.

I would also like to acknowledge my appreciation for my family, especially my mother, who encouraged and supported me throughout the completion of this thesis. Additionally, I extend my gratitude to all my friends who provided me with their unwavering support and guidance throughout this process.

Finally, I would like to express my thanks to all the authors cited in this document. Their work and contributions were critical to the successful completion of this thesis. Without them, this project would not have been possible.

Title in English

Formal verification of the high availability quality attribute for software systems based on microservices architectures.

Título en español

Verificación formal del atributo de calidad de alta disponibilidad para sistemas de software basados en arquitecturas de microservicios.

Abstract

The decisions made by software architects can have significant repercussions if they are made incorrectly. Therefore, it is crucial to base such decisions on concrete evidence rather than solely relying on heuristic experiences to determine if the architectural design decisions were correct. This approach can help avoid incurring refactoring or reengineering costs in the future.

This work proposes a general model for the formal verification of software architectures and presents a case study of an architecture based on microservices for ensuring high availability. Through this thesis, the different steps of the model will be presented, and it will be demonstrated how they should be applied to carry out a formal verification. This verification can guide software architects in making decisions based on formal evidence using automatic verification tools.

Keywords: Software architecture, quality attributes, formal methods, architectural tactics, formal verification, microservices architecture, high availability.

Resumen

Las decisiones tomadas por los arquitectos de software pueden tener graves repercusiones si se toman de manera incorrecta. Por lo tanto, es fundamental basar dichas decisiones en evidencias concretas en lugar de depender únicamente de experiencias heurísticas para determinar si las decisiones de diseño arquitectónico fueron correctas. Este enfoque puede ayudar a evitar costos de refactorización o reingeniería en el futuro.

Este trabajo propone un modelo general de verificación formal para arquitecturas de software y presenta un caso de estudio de una arquitectura basada en microservicios para asegurar el atributo de calidad de disponibilidad. A lo largo de esta tesis se expondrán los diferentes pasos necesarios para llevar a cabo una verificación formal y se explicará cómo deben aplicarse para guiar a los arquitectos de software en la toma de decisiones basadas en evidencias formales, utilizando herramientas de verificación automáticas.

Palabras clave: Arquitectura de software, atributos de calidad, métodos formales, tácticas arquitectónicas, verificación formal, arquitectura de microservicios, alta disponibilidad.

This master's thesis was defended on April 25, 2023 at 2:00 p.m.,
and was evaluated by the following juries:

Jorge Eliécer Camargo Mendoza, Ph.D.
Universidad Nacional de Colombia, Facultad de Ingeniería

Henry Roberto Umaña Acosta, M.Sc.
Universidad Nacional de Colombia, Facultad de Ingeniería

Contents

Acknowledgments vii

Abstract x

List of Figures xv

List of Tables 1

1 Introduction 2

2 Background 4
- 2.1 Software Architecture 4
- 2.2 Microservices Architecture Style 4
- 2.3 Availability Quality Attribute 5
  - 2.3.1 Availability Generalities 6
  - 2.3.2 Architectural Tactics in HA 7
- 2.4 Formal Verification 7

3 Related Work 10
- 3.1 High Availability in Microservices Architectures 10
- 3.2 Formal Methods in Software Architecture 10
- 3.3 Gaps in Previous Work 12

4 Analysis of Availability Tactics 14
- 4.1 Monitor 15
- 4.2 Retry 15
- 4.3 Functional redundancy 15
- 4.4 Replication 17
- 4.5 Exception handling 17
- 4.6 Escalating restart 18
- 4.7 Time stamp 19
- 4.8 Transactions 19
- 4.9 Ignore faulty behavior 19
- 4.10 Self-test 21

5 Verification Model **22**
5.1 System Specification 22
5.2 Scenario 23
5.3 System Properties 24
5.4 Structural Definition 24
5.5 Behavioural Definition 25
5.6 Implementation 26
5.7 Verification 26

6 Case Study **27**
6.1 Base Case Study 27
6.1.1 System Specification 27
6.1.2 Scenario 28
6.1.3 System Properties 28
6.1.4 Structural Definition 29
6.1.5 Behavioural Definition 31
6.1.6 Implementation 31
6.1.7 Verification 36
6.2 Study Case with Architectural Tactics 37
6.2.1 System Specification 37
6.2.2 Scenario 37
6.2.3 System Properties 38
6.2.4 Structural definition 39
6.2.5 Behavioural definition 41
6.2.6 Implementation 41
6.2.7 Verification 44
6.3 Results Comparison 46

7 Conclusions and Future Work **48**
7.1 Conclusions 48
7.2 Future Work 48

Bibliography **50**

List of Figures

4-1 C&C view of base model. 14
4-2 C&C view of monitor tactic. 15
4-3 C&C view of retry tactic. 16
4-4 C&C view of functional redundancy tactic. 16
4-5 C&C view of replication tactic. 17
4-6 C&C view of exception handling tactic. 18
4-7 C&C view of escalating restart tactic. 18
4-8 C&C view of timestamp tactic. 19
4-9 C&C view of transactions tactic. 20
4-10 C&C view of ignore faulty behavior tactic. 20
4-11 C&C view of self-test tactic. 21

5-1 Verification model. 22
5-2 Components & Connectors View. 23

6-1 Case study - Components & Connectors View. 27
6-2 Case study - Sequence diagram 29
6-3 Case study - Standard declarations - CPN Tools. 33
6-4 Case study - Implemented Model - CPN Tools. 35
6-5 Case study - Verification results. 36
6-6 Case study with tactics - Components & Connectors View. 37
6-7 Case study with tactics - Sequence diagram 39
6-8 Case study with tactics - Standard declarations - CPN Tools. 42
6-9 Case study with tactics - Implemented Model - CPN Tools. 44
6-10 Case study with tactics- Verification results. 45
6-11 Case study with tactics- Liveness results. 46

List of Tables

3-1 Study categorization. 12

1 Introduction

The fourth industrial revolution has resulted in software systems becoming increasingly large and complex. For this reason, carrying out a comprehensive design of the blueprints of the software systems before implementing them has become an essential part of maintaining harmony between the system's components. This process is known as software architecture [13].

Due to the complexity of software systems and the criticality of their quality, both industry and academia have dedicated significant effort to finding solutions to recurrent problems in the design process. As a result, we can find different architectural patterns and styles in software architecture that can be applied to the architectural design process [19].

Despite the inherent characteristics of architectural styles and patterns, it is necessary to be certain about the architectural decision and the impact that is being made when choosing Microservices Architecture (MSA) as the architectural style to be implemented in a system. For this, there are methodologies for making decisions on the architecture of a software system, however, authors such as Li [25] and Salama [37] agree that it is necessary to carry out validations on the candidate software architecture to determine if the architectural design decisions were correct, and not incur refactoring or reengineering costs in the future.

Currently, software architects use a heuristic method and their experiences to make architectural decisions by applying informal checks on their software systems [34]. This means that architects sometimes have to carry out refactoring or reengineering processes as they are not sure that the architectural decision made meets the non-functional needs they need. With the current trend of microservices architecture, some architects are migrating their systems to this architecture without being clear about the advantages and disadvantages that this entails. For this reason, it is necessary to have verifiers with formal methods [25] [37], which allow robust architectural decision-making and be able to carry out the implementation safely. For this reason, the research question posed is the following: How to formally verify the quality attribute of high availability in a microservices architecture?

The main objective of this thesis is to discuss the work that has been carried out in the area of implementing formal verifications on software architecture, with a particular emphasis on the microservices style. To achieve this, the thesis will cover the main components of MSA, the quality attributes associated with this architectural style, and the formal verification methods

that have been applied, with a special emphasis on Model Checking. By doing so, it will be possible to select a set of mathematical elements to define the high availability quality attribute and verify it in a microservices software architecture that has been designed specifically for this thesis.

For this reason, the objective achieved is be to apply a formal verification method for the quality attribute of high availability in software systems based on microservice architectures, through the following specific objectives:

- To classify a set of high availability architectural tactics for microservices architectures.
- To define a set of elements that allow to formally specify software systems based on microservices from the point of view of high availability.
- To formally specify a software system as a case study, based on a microservices architecture.
- To design a formal description model for the selected software system.
- To validate the formal model defined through formal verification processes.

Chapter 2 will provide the necessary background to understand the thesis development, including topics such as software architecture, architectural styles and patterns, with a focus on microservice architectures. Additionally, it will cover quality attributes and quality attribute assurance tactics, as well as formal verification.

In chapter 3, an exploration will be made of the different works carried out to ensure quality attributes through formal verifications focused on microservice architectures.

In chapter 4, a classification of architectural tactics used to ensure the high availability quality attribute will be carried out, focused on which of them structurally modify a software system using a base scenario.

Chapter 5 will propose the general verification model, and in Chapter 6, it will be applied to a case study in two phases. The first phase will not implement any architectural tactics, and the second phase will implement architectural tactics to ensure high availability. The formal verification will then be applied to the case studies using the state space method.

Finally, in chapter 7 conclusions and future work will be presented.

2 Background

2.1 Software Architecture

A system's architecture is the set of principal design decisions made during it's development and any subsequent evolution [43], this implies a high-level abstraction that allows to describe the elements and all the interactions of the system, including hardware and software [36]. Due to this, the architecture is present in all the systems, including of course those which were not done consciously, because it will have, whether contemplated or not, a structure.

2.2 Microservices Architecture Style

In software architecture we can find patterns and styles: An architectural style can be defined as a set of reuse design decisions that can be applied to a software system design [7]. On the other hand, an architectural pattern is a specific, well-established solution to a common design problem that appears in particular design contexts [4]. It means that an architectural style is a broad approach to solving architectural problems, while an architectural pattern is a specific solution to a specific problem within that architectural style.

Choosing the wrong architectural pattern or style can result in unnecessary time and costs since the process of changing it is known as the reengineering process. During the reengineering process, the tech team may need to re-implement software components if the software system is still under development or has already been finalized. Therefore, it is crucial to ensure that the correct decision is made to avoid such additional costs and efforts.

One important architectural style is Microservices, which have received special attention from researchers and private companies that have decided to implement it due to its characteristics [1].

A microservices architecture (MSA) according to the authors Di Francesco [11], Magableh [27], Zdun [50] y Newman [30] is a set of small services, each executing its own process and communicating by light mechanisms with each other.

From an architectural perspective, the elements and relationships that compose MSA are microservices and connectors REST respectively. For this reason, in an abstract level, an architecture can be taken as a graph, where each component is a node, and it's communication methods the connectors of the graph [49]. Likewise, it is possible to describe the structure of a system that implements this architecture, using the components and connectors view [5].

In other hand, like all the architectural styles, MSA has some main characteristics, that compose the main fundamentals, these are:

- Single responsibility principle: Each microservice should be restricted to performing the specific tasks it was designed for, and should carry out these tasks in an effective and efficient manner [30].
- Independent and cohesive process: The microservices should be designed to operate independently from one another, so that in the event of changes, only the microservice responsible for the specific functionality would need to be modified [10].
- Heterogeneous technology: Even though each microservice is independent, it is possible to select different technologies for each of them, based on their advantages and their ability to meet the desired quality standards [30].

Despite these characteristics, developers have the freedom to implement the architecture applying different design decisions, trying to reinforce the robustness of their system, although these decisions may not always be the best, in this way, many times they can fall into the called "bad smells" [29], which are bad practices applied in making decisions about the implementation of a software architecture. Among the most common bad practices in the use of MSA, the literature highlights the following:

- Do not use the Api Gateway component to perform orchestration on architecture components.
- Forced use of static IP addresses.
- "Woobly" interactions, which imply that, in the interaction of microservices, an error in microservice-1 produces an error in microservice-2 [29].

This is why it is very important to follow the MSA standards, which will ensure the basic quality attributes that are inherent to the architectural style.

2.3 Availability Quality Attribute

The quality attributes are properties that the software system has, through which a stakeholder can rate the quality of the system [36]. The authors Tekinerdogan, Ozca [44], Magableh [27] and

Kee Wook Rim [33] agree that the quality attributes that can be highlighted are interoperability, performance, availability, recovery before failures, security, portability, usability and modifiability. In this way, it means that a software system that does not have any quality attribute, even though it functionally fulfills everything required, is low-quality software that may not function properly. As an illustrative example, a bank information system can be taken, which performs all reconciliation tasks correctly, but if the system remains down, is too delayed or insecure, it may not fulfill what the bank needs or is expecting. to have. For this reason, it can be stated that there is a significant relationship between the quality metrics of a product and the quality metrics of a software architecture [17], and these must be built together when the system is designed, do not analyze them at a later stage [41] [18].

As well as the different architectural styles and patterns, the microservices architecture, due to its characteristics, allows ensuring quality attributes associated with low coupling, interoperability and the continuous provision of services [28]. The quality attribute to be work in this thesis is high availability:

2.3.1 Availability Generalities

Availability can be defined as the system readiness to provide correct service. It corresponds to the probability that the system is working within its specifications at a given instant. From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement. In availability, the cause of a failure is called a fault. A fault can be either internal or external to the system under consideration. Intermediate states between the occurrence of a fault and the occurrence of a failure are called errors. A failure is the deviation of the system from its specification, where that deviation is externally visible. Determining that a failure has occurred requires some external observer in the environment. For that reason, by Bass [4], availability is about minimizing service outage time by mitigating faults. Failure implies visibility to a system or human observer in the environment. One of the most demanding tasks in building a high-availability, fault-tolerant system is to understand the nature of the failures that can arise during operation. Once those are understood, mitigation strategies can be designed into the software.

The goal of availibility is to avoid single points of failure (SPOF), that is, eliminate all elements, that if they fall, the system will not work correctly or even the entire system will fall. Due to that, the availability of a system can be calculated as the probability that it will provide the specified services within required bounds over a specified time interval. When referring to this quality attribute, there is a well-known expression used to derive steady-state availability:

$$\frac{MTBF}{MTBF + MTTR}$$

where:
MTBF: Refers to the mean time between failures
MTTR: Refers to the mean time to repair

Due to this, when a system should operate continuously, with minimal downtime, and to quickly recover from failures is named High availability (HA). In other words, high availability is a level of availability that ensures a system is always operational and accessible to users, with minimal interruption.

2.3.2 Architectural Tactics in HA

An architectural tactic is defined as a characterization of an architectural decision that is necessary to ensure a quality attribute [45] [24].

According to Bass, classification of architectural tactics to ensure high availability are based on the moment where it is used, this classification is:

- Prevention: Due to availability want to avoid is the occurrence of a failure, these tactics are focused on acting in such a way that failures are minimized.
- Detection: When a failure occurs, it is necessary that it be detected in such a way that action can be taken, for this reason these tactics are focused on the early detection of system failures.
- Recovery: Once the fault is detected, it is necessary to take action to solve it in the most automatic way possible and in this way reduce the time of failure, these tactics are focused on acting in case a fault occurs in order to eliminate it.

2.4 Formal Verification

Formal method is a method in which all its constituent techniques and tools are formal that is given by a precise mathematical meaning, and the use of the techniques can be defined and justified formally [35].

Formal verification is a part of formal methods, and refers to the process of validate a design satisfies some requirements or properties [20]. In this way, to validate or verify a software design, it must be in a verifiable format, for that reason is important to define a language that is a

set of operators and variables which can be expressed in a verification model.

Due to this, the first step in a verification process is the specification process, which consists of obtaining a Finite State Machine description of the system. Finite State Machine is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM changes from one state to another in response to some inputs; the change from one state to another is called a transition. An FSM is defined by a set of states, and the inputs that trigger each transition [40]. The verification process is only one of the multiple sections of the formal methods.

In the revision of the literature, two principal resources were identified. Petri Nets and Model Checking. In this way, both can be used in the same process, so a software system can be defined in Petri Nets, and you can verify the model using Model Checking.

Petri nets are a formal language to specify and verify concurrent system, that uses a mathematical representation of a discrete event system [6]. This language can be used to specify control, function and time problems [15]. Through automata theory, these networks allow the specification of a distributed system in order to perform verifications on the structure based on a Bayesian probabilistic model, which allows the dynamic study of the behavior of complex systems [42].

As for Model Checking, this has become one of the most used formal verifiers in the analysis of complex system designs, because it is a fully automatic process [38], this being a significant advantage over the others checkers. Users who want to apply Model Checking must build a model of the software system and specify which property to check, and then the checker will perform property analysis on the proposed architecture [51]. Model Checking is an influential method for verifying complex, concurrent, and distributed system interactions, because it builds a behavior model of a system using formal concepts such as operations, states, events, and actions.

The state space method is a mathematical approach that is used to analyze and understand the behavior of systems. It involves representing the system as a set of states, and the transitions between those states. The state space of a system is a representation of all possible states that the system can be in, as well as the transitions between those states.

The state space method can be used to analyze a wide range of systems, including dynamical systems, control systems, and computer systems. It is a powerful tool for understanding the behavior of systems over time, and can be used to analyze both deterministic and probabilistic systems.

In the state space method, the states of the system are represented as points in a state space, and the transitions between states are represented as arrows between those points. The state

space method can be used to analyze both the steady-state behavior of a system and its transient behavior, and can provide insights into the stability, convergence, and other properties of the system.

3 Related Work

3.1 High Availability in Microservices Architectures

Microservices architectures have been studied from different perspectives, such as security, scalability and among others quality attributes, trying to compare the problems that other architectures have in search of looking at what happens in those same situations in microservices architectures. For example, in monolithic applications all modules run within hardware and execution process, a bug in server or the execution process, such as a memory leak, can affect the availability of all system [31], for that reason some authors like Torvekar [46] affirms that some companies are migrating their systems to microservices architectures to improve their system's availability and meet the service-level agreement (SLAs). On the other hand, microservices can increase complexity in system monitoring and detect faults for the different components [52].

In 2021, Shanshan Li [26], categorized availability tactics in MSA into four types, which are:

- Fault Monitor
- Service Registry
- Circuit Breaker
- Inconsistency Handler

3.2 Formal Methods in Software Architecture

There are some methodologies that are used to improve assertiveness in architectural decision making, the authors Svahnberg, Wohlin (2003) and Kazman (2005) mention the ATAM methodology, which is a methodology to evaluate Software Architectures based on the attributes of quality specified for the system [23][41]. After the definition of these methodologies, the authors Ferrari and Madjavhi [14] conducted a study in 2008 in which they show that groups of architects that use methodologies tend to perform better than those architects that do not.

Despite using these methodologies for decision making, both authors Li [25] and Salama [37] agree that it is necessary to perform validations on the candidate software architecture to determine if the architectural design decisions were correct and not incur refactoring or reengineering

costs in the future.

In 2005 the authors Kazman, Bass and Klein [23] mentioned that there are two verification methods: formal and informal. Formal methods have a fairly strong mathematical foundation, however, the authors mention that there are few formal methods to determine if a software architecture is correct.

Throughout the 2000s, different authors such as Dobrica & Niemelä [12], Babar and Gorton [2], Zhang, Pengcheng Muccini, Henry Li and Bixin [51] have mentioned the importance of software architecture analysis methods and have carried out works comparing different methods to determine their most important characteristics, similarities and differences, including the specific objectives, the evaluation method, the quality attributes evaluated by each method and the validation method. However, the authors Henry Li and Bixin focus on the techniques applied with Model Checking, where they analyzed 16 techniques to evaluate software architectures.

The authors Pavel Parizek and Frantisek Plasil [32] mention that performing model checking of isolated software components is not possible because the components do not form a complete system with a specific starting point. In order to get around that pitfall, it is typically necessary to create a component environment for the system on which you are trying to apply model checking [32].

In 1999 Giannakopou [16] proposed a model checking approach to analyze the behavior of concurrent systems using FSP, later in 2001, Suzanne Barber and Thomas Graser [3] proposed Arcade, an approach that allows timely feedback from the architecture designed through simulation and model checking, these authors assure that Model checking is a very robust technique for validating the quality attributes of availability and security.

In order to apply model checking in a software architecture, it is necessary to define the input that the analyzer will receive, for this it is broken down into two main entities: the architecture specifications, and the quality attributes. Different specification languages can be used for this process, both formal and informal [51]. The most widely used specification is formal Architecture Description Language (ADLs), although, with the CHARMY approach, UML can be used as a notation to specify the architecture to be evaluated, since it is based on models [21]. Once the architecture is specified, the quality attributes must be specified, for this, each of the approaches uses its own tool, in which the CHARMY tool is highlighted, which provides a graphical tool [51].

Once the input parameters are defined, the model checking engine must be chosen. The authors mention different engines like FDR, SPIN, PoliMC and SMV [47][51][8].

In this way, it is timely to emphasize the importance that the authors have given to the formal

verifications of software architectures.

3.3 Gaps in Previous Work

A classification of the most relevant works for this thesis was carried out, where the categories worked by each one of them are cataloged, the results were the following:

Table 3-1: Study categorization.

Study	Software Architecture	Architecture Evaluation	Quality attributes	Microservice Architecture	Availability attribute	Formal Methods
Ahmad, A., & Babar, M. A. (2016). Software architectures for robotic systems: A systematic mapping study. Journal of Systems and Software, 122, 16–39. https://doi.org/10.1016/j.jss.2016.08.039	x					
Babar, M. A., Kitchenham, B., Zhu, L., Gorton, I., & Jeffery, R. (2006). An empirical study of groupware support for distributed software architecture evaluation process. Journal of Systems and Software, 79(7), 912–925. https://doi.org/10.1016/j.jss.2005.06.043	x	x				
BARBER, K. S. ; GRASER, Thomas ; HOLT, Jim: Providing early feedback in the development cycle through automated application of model checking to software architectures	x	x				x
Bass, L., Clements, P., & Kazman, R. (2013). Software Architecture in Practice Second Edition Third Edition. In Communication. https://www.oreilly.com/library/view/software-architecture-in/9780132942799/	x		x		x	
Brito, P. H. S., De Lemos, R., Rubira, C. M. F., & Martins, E. (2009). Architecting fault tolerance with exception handling: Verification and validation. Journal of Computer Science and Technology, 24(2), 212–237. https://doi.org/10.1007/s11390-009-9219-2	x	x	x			x
Camilli, M., Gargantini, A., Scandurra, P., & Bellettini, C. (2017). Event-based runtime verification of temporal properties using time basic Petri nets. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10227 LNCS, 115–130. https://doi.org/10.1007/978-3-319-57288-8_8			x			x
Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003). Documenting software architectures: Views and beyond. Proceedings - International Conference on Software Engineering, 131, 740–741. https://doi.org/10.1109/icse.2003.1201264	x		x		x	
Czepa, C., Tran, H., Zdun, U., Tran, T., Weiss, E., & Ruhsam, C. (2017). Reduction techniques for efficient behavioral model checking in adaptive case management. Proceedings of the ACM Symposium on Applied Computing, Part F1280, 719–726. https://doi.org/10.1145/3019612.3019617	x	x	x			x
Dehkordi, Z. S., Harounabadi, A., & Parsa, S. (2013). Evaluation of software architecture using fuzzy color Petri net. Management Science Letters, 3(2), 555–562. https://doi.org/10.5267/J.MSL.2012.12.016	x	x	x			x
Demirli, E., & Tekinerdogan, B. (2011). Software language engineering of architectural viewpoints. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6903 LNCS, 336–343. https://doi.org/10.1007/978-3-642-23798-0_36	x					
Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. Journal of Systems and Software, 150, 77–97. https://doi.org/10.1016/j.jss.2019.01.001	x		x	x		
Dobrica, L., & Niemelá, E. (2002). A survey on software architecture analysis methods. In IEEE Transactions on Software Engineering (Vol. 28, Issue 7, pp. 638–653). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/TSE.2002.1019479	x	x				
Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In Present and Ulterior Software Engineering (pp. 195–216). Springer International Publishing. https://doi.org/10.1007/978-3-319-67425-4_12	x		x	x	x	
Ferrari, R., & Madhavji, N. H. (2008). Software architecting without requirements knowledge and experience: What are the repercussions? Journal of Systems and Software, 81(9), 1470–1490. https://doi.org/10.1016/j.jss.2007.12.764	x	x				
Ghezzi, C., Mandrioli, D., Morasca, S., & Pezze, M. (1991). A unified high-level Petri net formalism for time-critical systems. IEEE Transactions on Software Engineering, 17(2), 160–172. https://doi.org/10.1109/32.67597		x				x
Giannakopoulou, D. (1998). Model Checking for Concurrent Software Architecutures. Department of Computing, January. http://www.doc.ic.ac.uk/∼dg1/tracta/papers/thesis.pdf	x	x	x			x
Hansen, K. M., Jonasson, K., & Neukirchen, H. (2011). An empirical study of software architectures' effect on product quality. Journal of Systems and Software, 84(7), 1233–1243. https://doi.org/10.1016/j.jss.2011.02.037	x	x	x			
Haoues, M., Sellami, A., Ben-Abdallah, H., & Cheikhi, L. (2017). A guideline for software architecture selection based on ISO 25010 quality related characteristics. International Journal of System Assurance Engineering and Management, 8, 886–909. https://doi.org/10.1007/s13198-016-0546-8	x	x	x		x	x
Zhang, P., Muccini, H., & Li, B. (2010). A classification and comparison of model checking software architecture techniques. Journal of Systems and Software, 83(5), 723–744. https://doi.org/10.1016/j.jss.2009.11.709	x	x	x			x
Li, J. J., & Horgan, J. R. (2000). Applying formal description techniques to software architectural design. Computer Communications, 23(12), 1169–1178. https://doi.org/10.1016/S0140-3664(99)00244-3	x	x	x			x
Salama, M., & Bahsoon, R. (2017). Analysing and modelling runtime architectural stability for self-adaptive software. Journal of Systems and Software, 133, 95–112. https://doi.org/10.1016/j.jss.2017.07.041	x	x	x			

After reviewing the previous work, it was found that some authors have contributed through formal techniques for the assurance of quality attributes but focused on types of systems such

as IOT or aerospace, however, given the important boom that microservices architectures have had in the construction of systems, the opportunity was found to contribute to the scientific community in the application of formal methods to ensure the high availability attribute in microservices architectures, and in this way help software architects to make decisions based on formal results.

4 Analysis of Availability Tactics

To better illustrate the applicability of the proposed model, it is important to consider not only architectural tactics that are based on the behavior of the system but also those that make structural modifications to the system. Since the focus of this thesis is on high availability, an analysis of 10 architectural tactics is carried out to validate whether the tactic modifies the system structurally.

Therefore, the selected architectural tactic will be applied to a system that has already been designed, where the views of components and connectors have been established. This approach will enable a comparison of the system before and after the tactic is applied to confirm that the views of components and connectors remain unchanged.

The base system for the modification is based on an MSA that utilizes the orchester pattern:

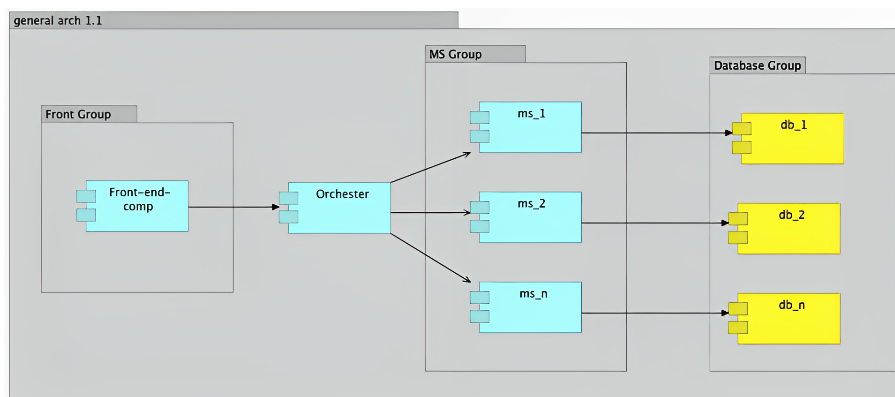

Figure 4-1: C&C view of base model.

As can be seen in graph 4-1, the system will be made up of 8 components, of which **Front-end-comp** corresponds to the component in charge of interacting with the user, **Orchester** to perform the system orchestration, **ms_1**, **ms_2** and **ms_3** will have specific logic system, and **db_1**, **db_2** and **db_3** responsible for system persistence.

4.1 Monitor

Monitor is a external component that is used to keep track of the functioning of various components within a system, such as processors, processes, input/output, memory, and so on. A system monitor can identify issues or high levels of use in the network or other shared resources, such as in the case of a denial-of-service attack [4].

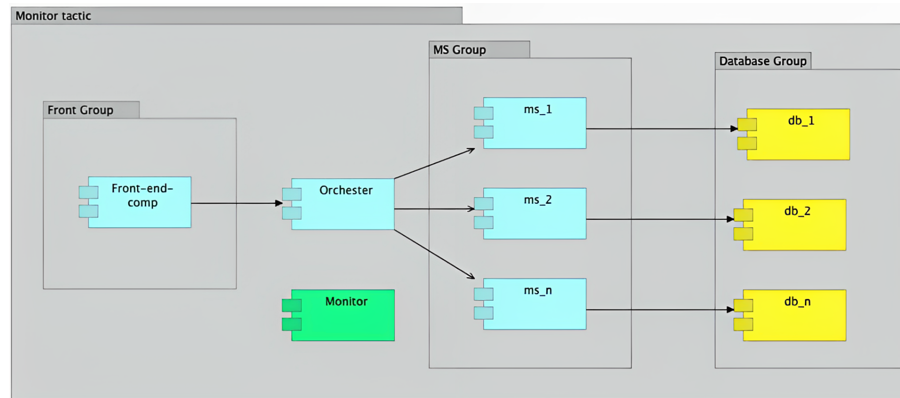

Figure 4-2: C&C view of monitor tactic.

As can be seen, the monitor tactic introduces a new component in the system in charge of carrying out the analysis of the system variables, therefore it does structurally modify the system.

4.2 Retry

The retry tactic is based on the assumption that temporary issues are the cause of failures, and that repeating the operation may result in success. This tactic is often used in networks and server farms, where failures may be expected and frequent. It is important to set a limit on the number of times an operation is retried before it is considered a permanent failure [4].
As can be seen in figure 4-3, the retry tactic does not introduce a new component into the system, since retries only modify the behavior of the previously described components, therefore it does not structurally modify the system.

4.3 Functional redundancy

In this tactic, the components are required to consistently provide the same output when given the same input, even though they have been designed and implemented in different ways [4].
The functional replication tactic introduces new components into the system, since it is necessary to implement the current components in different ways and deploy them to work together, thus structurally modifying the system.

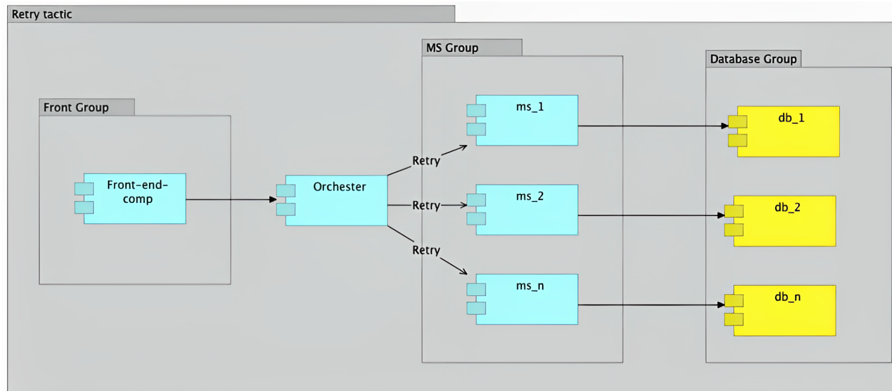

Figure 4-3: C&C view of retry tactic.

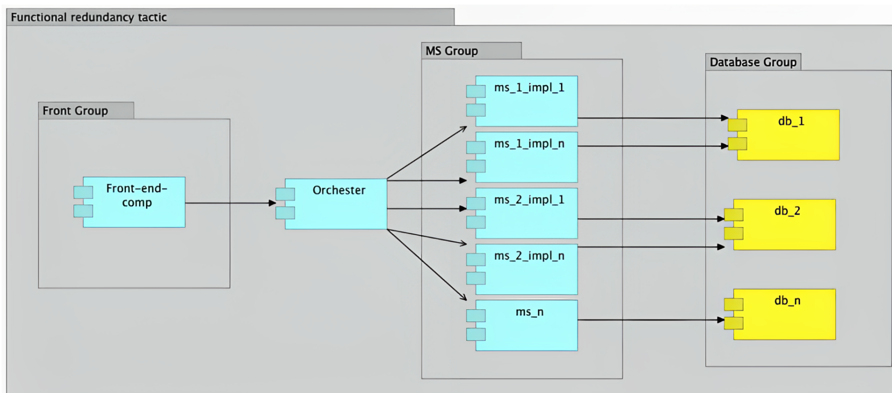

Figure 4-4: C&C view of functional redundancy tactic.

4.4 Replication

This tactic involves using multiple copies of identical components. This approach can be effective in protecting against random hardware failures, but it cannot protect against design or implementation errors in hardware or software because there is no diversity among the components [4].

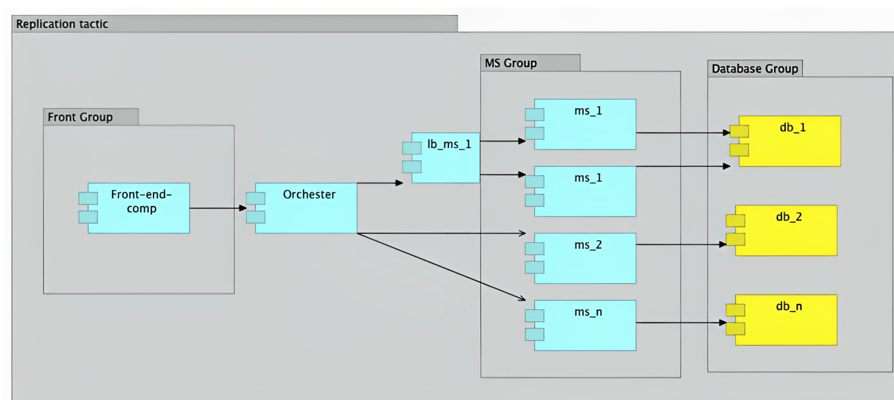

Figure 4-5: C&C view of replication tactic.

From component and connector view theory, multiple instances should not represent a new component within the view. However, since it modifies the system behaviorally, it is necessary to specify it for the proposed verification model, where additionally the new component known as load balancer (lb) must be specified, in charge of balancing the requests between the different instances of the component.
The replication tactic introduces new components into the system since it is necessary to deploy the current components several times, in this way not leaving each component as a single instance, and additionally it is necessary to deploy a new component called a load balancer that allows distribute the loads depending on different factors such as the load, and the availability of each one of them, therefore it structurally modifies the system.

4.5 Exception handling

When an exception is detected, the system must respond in some way. The simplest option would be to crash, but this is not a good solution in terms QA. Instead, there are more productive approaches that can be taken. The mechanism used for exception handling depends on the programming environment being used, and can range from simple error codes returned by functions to the use of exception classes that contain information about the exception, such as its name, origin, and cause. This information can then be used by the software to address the fault, often by correcting the cause of the exception and retrying the operation [4].

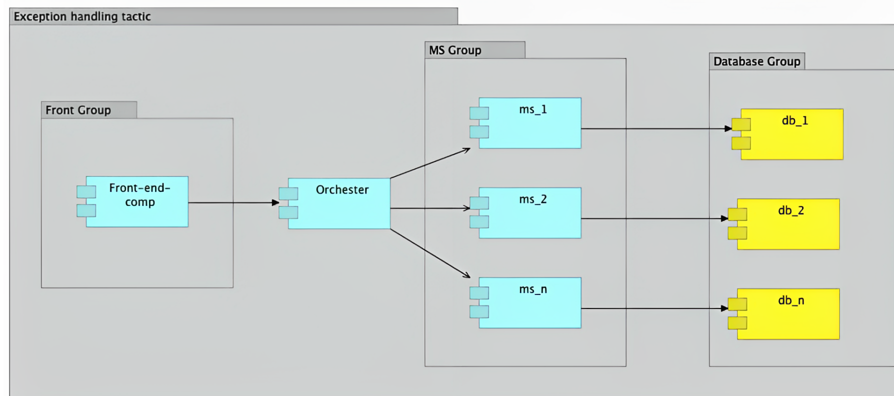

Figure 4-6: C&C view of exception handling tactic.

The exception handling tactic does not introduce a new component into the system as the exception handling behavior is implemented within the components, such as the orchestrator, the microservice, or the component where the error occurred, therefore it does not structurally modify the system.

4.6 Escalating restart

The Escalating restart tactic is a method of recovery that allows the system to recover from faults by starting again at different levels of granularity and minimizing the impact on service [4].

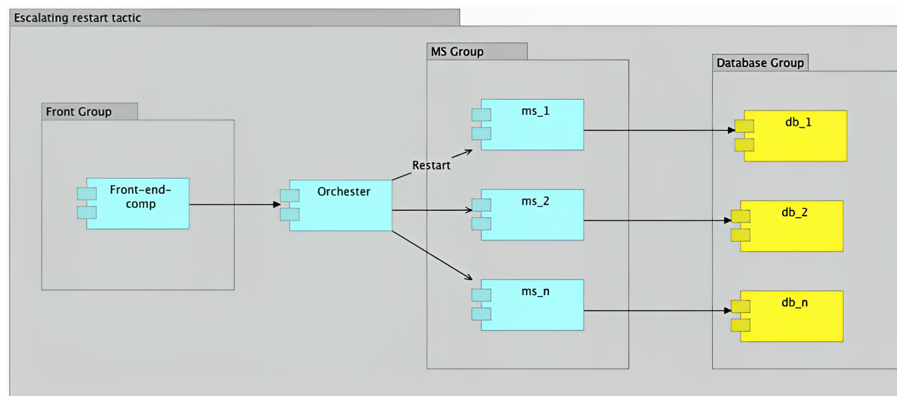

Figure 4-7: C&C view of escalating restart tactic.

This tactic is unique because it enables the execution of escalating restart behavior either by the current components or a separate component. The first scenario, where this function is assumed by the current components, is considered in this thesis. In this way, it is determined that this tactic does not structurally modify the system.

4.7 Time stamp

This tactic is used to identify incorrect sequences of events, particularly in distributed systems that rely on message passing. The time stamp of an event can be recorded by assigning the state of a local clock to the event immediately after it occurs [4].

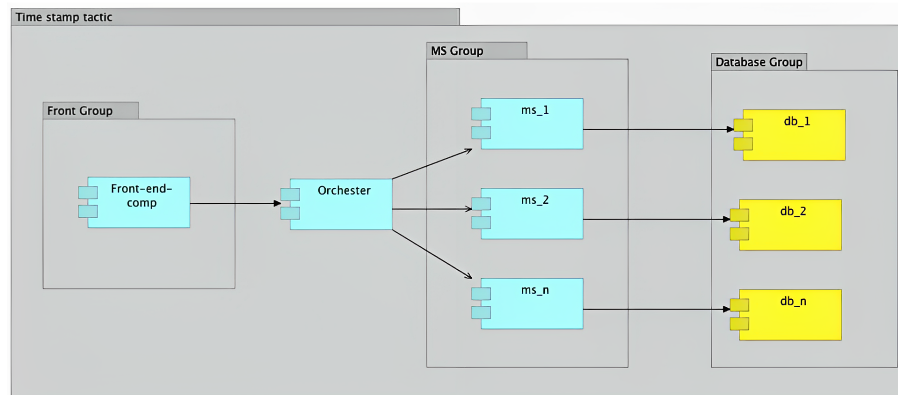

Figure 4-8: C&C view of timestamp tactic.

The timestamp tactic is implemented under the same components and the same connectors, since it is not necessary to add new calls, instead it is necessary to add the data to each of the requests in order to apply it. Therefore this tactic does not structurally modify the system.

4.8 Transactions

Systems that aim to provide high-availability services use transactional semantics to ensure that the messages exchanged between distributed components are atomic, consistent, isolated, and durable. These four properties, known as the "ACID properties," help to prevent race conditions caused by two processes attempting to update the same data item at the same time [4].
The transaction tactic is implemented under the same components, and requires a logical implementation in order to achieve the ACID principles for each operation. Therefore this tactic does not structurally modify the system.

4.9 Ignore faulty behavior

This tactic involves disregarding messages that are sent from a specific source when it is determined that they are not valid. For example, this approach could be used to ignore messages from an external component attempting to launch a denial-of-service attack by implementing Access Control List (ACL) filters [4].

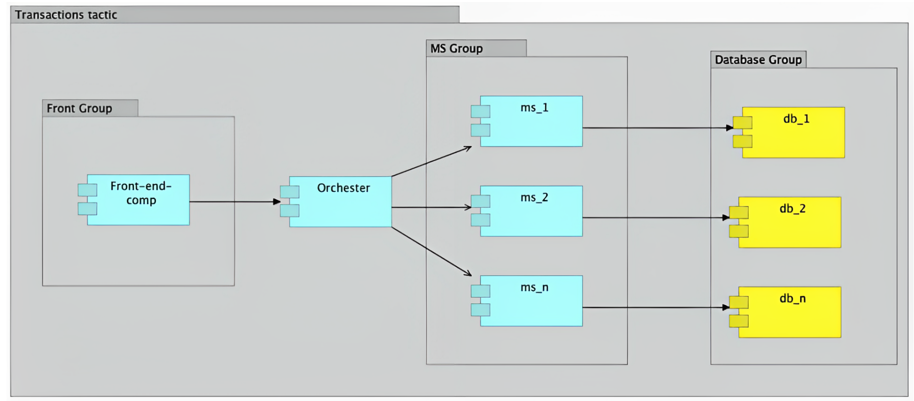

Figure 4-9: C&C view of transactions tactic.

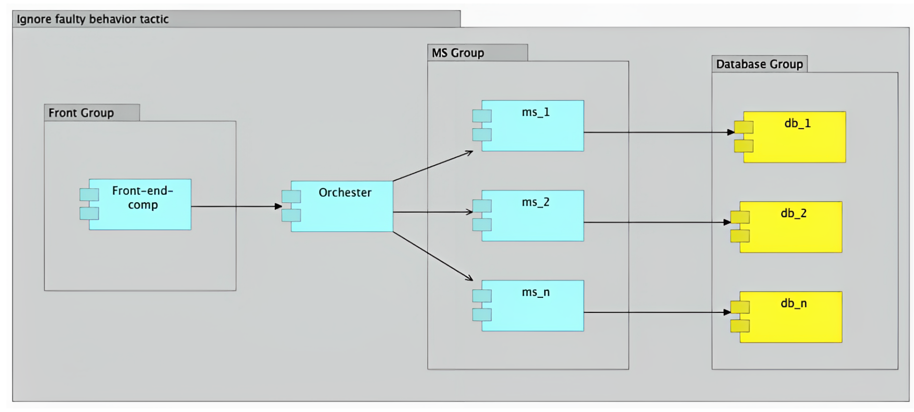

Figure 4-10: C&C view of ignore faulty behavior tactic.

The ignore faulty behavior tactic is implemented under the same components, and requires an implementation logic in order to achieve the behavior. Therefore this tactic does not structurally modify the system.

4.10 Self-test

Components (or, more likely, entire subsystems) can run tests to verify that they are functioning correctly. These self-test procedures can be initiated by the component itself or run periodically by a system monitor. These tests may include techniques similar to those used in condition monitoring [4].

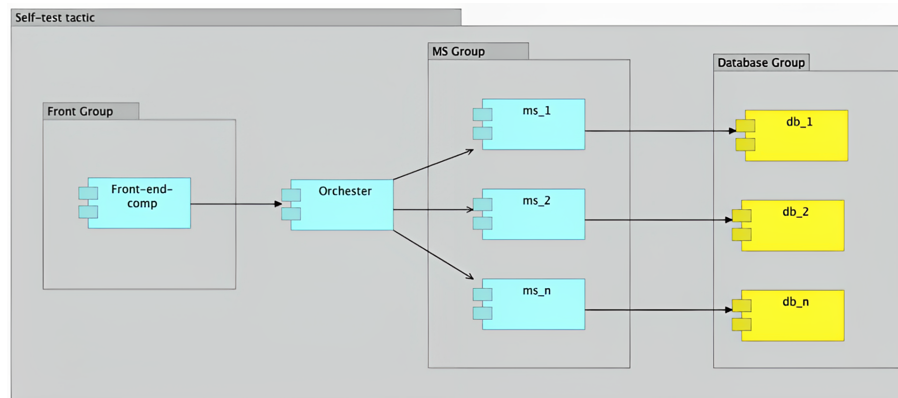

Figure 4-11: C&C view of self-test tactic.

The self-test tactic is implemented under the same components, and requires a logical implementation to be able to achieve the necessary tests of each one of the components. Therefore this tactic does not structurally modify the system.

5 Verification Model

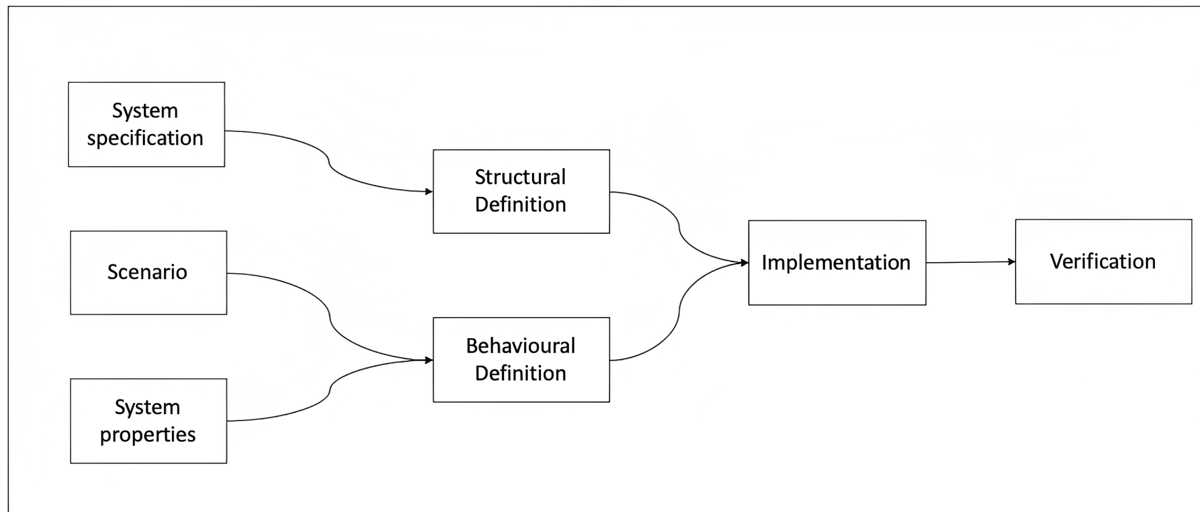

Figure 5-1: Verification model.

The general verification model is important, because it allows determining the factors that must be specified in order to later formally verify a software system. In the case of this study, it was decided to use a model based on author Zhang [51]. For this, three initial components are described to be defined in any system that wishes to use the formal modeling and verification process proposed in this study. These components are:

5.1 System Specification

For the topological description of a software system, it is decided to use components and connectors (C&C) view proposed by Clements [7], because it allows the abstraction of the architectural structure of the system at runtime enough to be formally modeled later.

The Components and Connectors view are composed by:

- Components: Logical units in charge of process and data stores. It interacts with other components using connectors.
- Connectors: Communication channel between components.

For this need to be take into account:

- Determine the parts of the system: Begin by identifying the various components that make up the system. A component is a standalone unit of functionality that performs a particular task or fulfills a specific role within the system.
- Establish the connections between components: Next, determine the relationships between the components. This can include dependencies, in which one component depends on another component to function, or collaborations, in which two or more components work together to accomplish a shared objective.

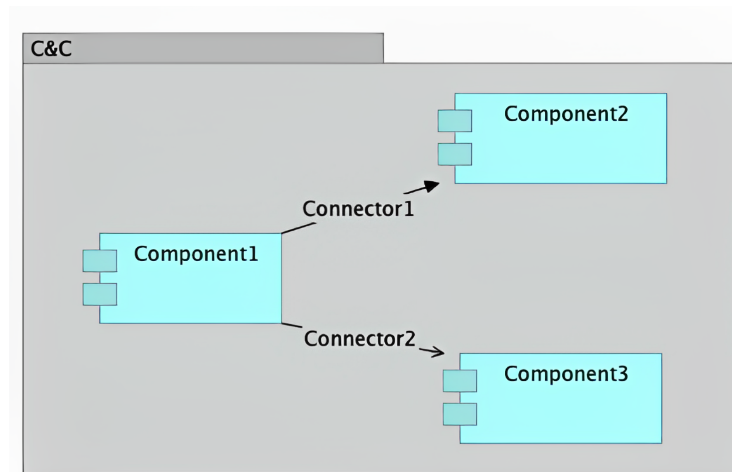

Figure 5-2: Components & Connectors View.

Similarly, it is not necessary to make low-level specifications in the diagram, since these can be specified in each of the scenarios as long as they have an effect on the variables that influence the system. That is, if the programming language used has an influence on the number of errors that appear, this will be a parameter to be defined in section 5.3.

Output: Components & connectors view of the system.

5.2 Scenario

The scenario is related to the different variables that can change during the simulation that is carried out later. In this way, it refers to the different variables that can infer during the execution environment of the software system, such as: maximum repair time, number of requests per second, probability of failure of a component, among others. They can be seen as assumptions during the modeling process.

It is necessary to make a list of the parameters that they want taken into account. Once this is done, the number of scenarios to be tested must be created, assigning a quantitative or qualitative value to each parameter described in the list made for each of them.

This approach allows us to be as specific in the variables as needed. For instance, for a specific case, it may be enough to specify the probability error of a component. However, for other cases, it may be necessary to specify more details of system parameters as long as they have an impact on its behavior. This is crucial as it affects the results delivered by the verifier.

It is important to note that the type of verifier used can be influenced by these variables. Therefore, it is crucial to select a verifier that is capable of defining the scenario as required and not limited by the same verifier. It is therefore important to follow the order of the verification model proposed in this thesis.

Output: List of scenarios with its respective value for each parameter & constraints to be evaluated.

5.3 System Properties

They refer to the system's ability to act in a certain way, which in turn can affect each of the system's behaviors. They are generally defined as system restrictions, and can allow to ensure or worsen the behavior of the system to maximize a quality attribute of the system.

In this way, in order to be defined, it must be clearly specified what the system is capable of depending on each of the situations that occur. An example may be the way in which the system behaves when a system failure occurs.

These must be described sequentially according to the operation of the described system. The interaction of each component with another component, and the operation of each of them at runtime.

Output: Sequence diagram of the system.
With these 3 initial components it is now possible to make a more formal definition of the software system and its properties.

5.4 Structural Definition

Input: Components & connectors view of the system.

The system specification refers to the structure of the system, and to achieve this, the following factors must be taken into account:

- The components and connectors must be translated into a specific modeling language that allows for the definition of each of the components and connectors.
- There is no restriction on the selected language as long as it allows for the syntactic and semantic definition of the system in a previously defined structural way.

Output: Model language of list of components and their connections.

5.5 Behavioural Definition

Input: Sequence diagram of the system.

The behavior of the system will be defined by the different flows and actions that occur in the system, as well as specific events that can or should occur in the system. In this way, it is important to detail all the actions that can occur so that later the results are the most accurate possible, in the same way, this does not imply that some events may not be defined and even then they will appear in the results, it all depends of the verifier selected later. For this reason, the system operation flow should be listed together with the possible responses to be returned.

In the case of this thesis, the functions will be listed as follows:

$$functionName(component1, component2, object) = return(result_1, result_2, result_n)$$

where:

- **functionName**: It is the name of the function that is executed.
- **component1**: It is the component that makes the request.
- **component2**: It is the component that handles the request.
- **object**: It is the sent object that has the request information.
- **result**: It is the list that represents each of the possible objects that will return the response of the request.

Output: Model language of system flow with its possible states.

5.6 Implementation

Input: Components & Components view, list of scenarios with its respective value for each parameter and sequence diagram of the system.

Each component and connector must be converted to the chosen modeling language that allows defining each of them independently and which allow to model complex structures using the types defined previously.

Likewise, it must be possible to implement the different previously defined flows in the selected modeler in order to obtain in the results each one of them, the number of occurrences and the necessary feedback to carry out the correct formal verification later. In chapter 6 an implementation will be carried out in a case study that will detail this process more.

Output: Mathematical representation of the system & requirement that want to check.

5.7 Verification

Input: Mathematical representation of the system & requirement that want to check.

Ensuring that a formal model meets certain properties or requirements involves verifying the model. Use a verification tool to check whether the model satisfies the property. The verifier will either return a result indicating that the property holds, then analyze the results, if the verifier returns a counterexample, analyze the counterexample to understand why the property is violated and how to fix the issue. If the property holds, the verification process is complete.

Output: Results.

6 Case Study

6.1 Base Case Study

For the Case study, a basic transactional system will be defined that contains all the necessary elements that can be implemented in real life. In this way, it is decided to use a system based on the architectural style of microservices that contains a component in charge of the interaction of the users, a component that performs the orchestration of the system, some components that contain the specific logic of the business and finally some components responsible for managing and storing information.

6.1.1 System Specification

The system will be defined according to the following view of components and connectors:

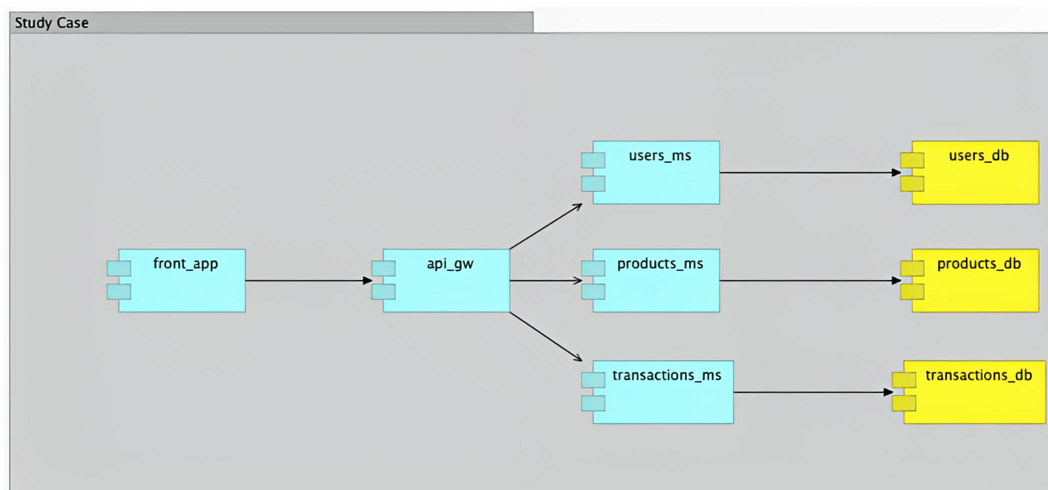

Figure 6-1: Case study - Components & Connectors View.

As shown in Graph 6-1, the system will consist of eight components, of which **front_app** corresponds to the component in charge of interacting with the user, **api_gw** to perform the system orchestration, **users_ms**, **products_ms** and **transactions_ms** will have specific logic system, and

users_db, **product_db** and **transactions_db** responsible for system persistence.

6.1.2 Scenario

In this study, variables that directly affect the availability of the system will be taken into account. As it is a basic transactional system, it will consist of requests that must be resolved correctly. Therefore, the variables that will be considered in this case study are as follows:

- **Number of requests**: These are the number of requests that the system should attend to.
- **Probability of failure in users_ms**: It is the probability that a request is not serviced correctly by the users_ms microservice.
- **Probability of failure in auth_ms**: It is the probability that a request is not handled correctly by the auth_ms microservice.
- **Probability of failure in transactions_ms**: It is the probability that a request is not serviced correctly by the transactions_ms microservice.

For illustrative purposes, a single scenario will be carried out to be verified by the proposed model, the above with the objective of emphasizing the results. Thus, the scenario to be tested is the following:

Scenario 1:
- Number of requests: 5
- Probability of failure in users_ms: 25%.
- Probability of failure in products_ms: 0%.
- Probability of failure in transactions_ms: 0%.

Constraints to validate:
- Failure transaction always at 0.

6.1.3 System Properties

The system will allow you to carry out a transaction using the following flow:

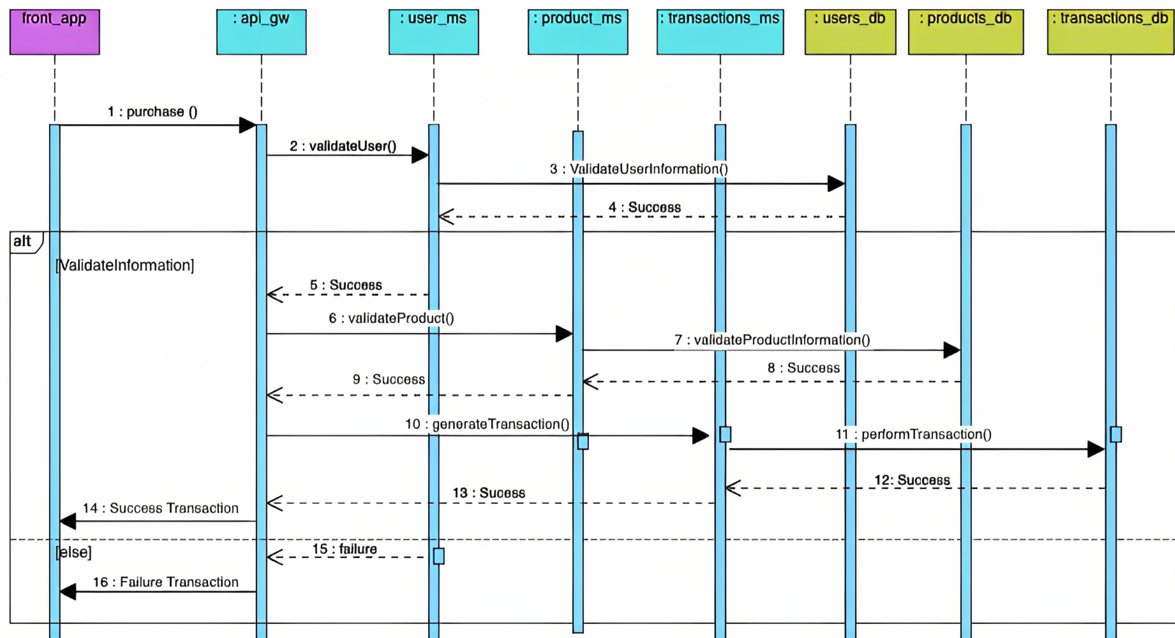

Figure 6-2: Case study - Sequence diagram

As can be seen in image 6.2, the functionality in user_ms can fail generating a complete failure of the operation, therefore, the desired state is that all requests are resolved correctly.

6.1.4 Structural Definition

The system components are as follows:

- **front_app**: Will be the component in charge of making the requests to the system, because it is the component through which the users of the system make use of it.
- **api_gw**: Will be in charge of managing the orchestration of each of the system requests between the different microservices.
- **users_ms**: Will be in charge of managing user information.
- **product_ms**: Will be in charge of managing product information.
- **transactions_ms**: It will be in charge of carrying out the transactional operations of the system.
- **users_db**: Will be in charge of the persistence of system user data.
- **products_db**: Will be in charge of the persistence of system product data.

- **transactions_db**: It will be in charge of the persistence of the transactions carried out in the system.

It has been decided to use the Sarch language [48], due to it allows the structural definition of a software system from different architectural views. However, in the case of this thesis, it will be carried out as mentioned in the previous chapter through the definition of the components and connectors of the system. Therefore, this is the implementation of the selected software system:

```
architecture {
  software_system : CaseStudy;
  author : CamiloDajer;
  architectural_views{
    component_and_connector_view ::
    elements {
      component_types{
        front; orchestrator; logical; db;
      }
      component front front_ms ;
      component orchestrator api_gw ;
      component logical users_ms ;
      component logical products_ms ;
      component logical transactions_ms ;
      component db users_db ;
      component db products_db ;
      component db transactions_db ;
      connector purchase ;
      connector validateUser ;
      connector validateProduct ;
      connector generateTransaction ;
      connector validateUserInformation ;
      connector validateProductInformation ;
      connector performTransaction ;
    }
    relations {
      attachment( purchase:front_ms,api_gw );
      attachment( validateUser:api_gw,users_ms );
      attachment( validateProduct:api_gw,products_ms );
      attachment( generateTransaction:api_gw,transactions_ms );
      attachment( validateUserInformation:users_ms,users_db );
      attachment( validateProductInformation:products_ms,products_db );
      attachment( performTransaction:transactions_ms,transactions_db );
    }::
```

```
    }
}
```

6.1.5 Behavioural Definition

The defined functions of the behavior of the system are as follows:

- $purchase(front_app, api_gw, req) = return(success_transaction, failure_transaction)$
- $validateUser(api_gw, users_ms, req) = return(success, failure)$
- $validateUserInformation(users_ms, users_db, req) = return(success)$
- $validateProduct(api_gw, products_ms, req) = return(success)$
- $validateProductInformation(products_ms, products_ms, req) = return(success)$
- $generateTransaction(api_hw, transactions_ms, req) = return(success)$
- $performTransaction(transactions_ms, transactions_db, req) = return(success)$

6.1.6 Implementation

6.1.6.1 Petri Nets in MSA

Just like all the standar in concurrent applications [39], microservices architectures allow design concurrent systems due to their advantages in receiving and process request in high scalable environments. For that reason, Petri nets are chosen to model a microservices architecture for the easy model process of places and transitions, allowing represent the different components in an actual microservices architectures using api gateway pattern such as:

1. Load balancers
2. Security components (WAFs, IPS/IDS, etc)
3. Api Gateway
4. Microservices and their communication with Api Gateway
5. Database
6. Storage components

As a result, in this thesis Colored Petri will be used, due to Coloured Petri nets allow model different behaviors in transitions where contrary to Petri net this allows defining variables and data types that are very useful in the case of modeling. For that reason is possible model quality attributes in software systems, such as high scalability, thanks to the functions that allow set constraints when the model is evaluated.

6.1.6.2 CPN Tools

CPN Tools is an important tool that allow modeling and verifying petri nets [22]. CPN Tools makes it possible to study the behavior of the modeled system using a simulation to verify properties by means of state space methods and model checking, and to conduct a simulation-based performance analysis [9].

For the purpose of this thesis, the CPN Tools is being used as a modeling and verification tool for the case study.

6.1.6.3 Case Study Implementation

As previously mentioned, in the case of this thesis, Petri Nets is being used to model the case study, in this way, the components and connectors will be mapped as states and transitions of the Petri Nets respectively. Additionally, 2 new states will be generated, *successful_transaction* and *failure_transaction*, which will correspond to the 2 final states of each of the requests that will be sent.

Petri Nets also allow the creation of tokens, which are those that allow a transition to be activated as long as its requirements are met. For our case, these tokens will be the requests that will travel through the states and transitions to finally end up in one of the two final states.

Therefore, we proceed to create the petri nets in the selected program called CPN Tools, implementing the following declarations in the system:

```
▼Declarations
  ▶Standard priorities
  ▼Standard declarations
    ▼colset BOOL = bool;
    ▼colset INT = int;
    ▼colset INTINF = intinf;
    ▼colset TIME = time;
    ▼colset REAL = real;
    ▼colset STRING = string;
    ▼colset REQUEST = int with 1..10 ;
    ▼var req : REQUEST;
    ▼colset RESPONSE = with success | failure ;
    ▼fun validateUser(req) =  1`1;
    ▼fun validateProduct (req) = 1`3;
    ▼fun generateTransaction (req) = 1`5;
    ▼fun validateUserInformation (req) = 1`1;
    ▼fun validateProductInformation(req) = 1`3;
    ▼fun performTransaction(req) = 1`5;
    ▼fun performdb(req) =  1`req;
    ▼fun db(req) = 1`req;
    ▼fun response_users_db(req) = 1`2;
    ▼fun response_products_db(req) = 1`4;
    ▼fun response_transactions_db(req) = 1`6;
    ▼fun purchase(req) = 1`1;
    ▼fun perform(req) = 1`1;
    ▼fun users_response(req) = if discrete(1,4) = 1 then
      1`9
     else
      1`3;
    ▼fun products_response(req) = 1`5;
    ▼fun transactions_response(req) = 1`8;
    ▼fun success_transaction(req) =  1`8;
    ▼fun failure_transaction(req) =  1`9;
```

Figure 6-3: Case study - Standard declarations - CPN Tools.

In this way, the types, variables, and functions are declared and supported in a colored Petri net.

6.1.6.4 Types:

- **REQUEST**: It will be composed of a set of integers from 1 to 10, in this way each number will represent the state of the request and who will be the next component to process the request.
- **RESPONSE**: It will be composed of 2 types, success and failure, which will represent the final status of the request.

6.1.6.5 Variables:

- **req**: Represents each of the requests made by the system.

6.1.6.6 Functions:

- **validateUser**(**req**): It will be the function in charge of validating that the request must be processed by users_ms, therefore it will send the token to the corresponding state.
- **validateProduct** (**req**): It will be the function in charge of validating that the request must be processed by products_ms, therefore it will send the token to the corresponding state.
- **generateTransaction** (**req**): It will be the function in charge of validating that the request must be processed by transactions_ms, therefore it will send the token to the corresponding state.
- **validateUserInformation** (**req**): It will be the function in charge of validating that the request must be processed by users_db, therefore it will send the token to the corresponding state.
- **validateProductInformation**(**req**): It will be the function in charge of validating that the request must be processed by products_db, therefore it will send the token to the corresponding state.
- **performTransaction**(**req**): It will be the function in charge of validating that the request must be processed by transactions_db, therefore it will send the token to the corresponding state.
- **users_response**(**req**): It will be the function in charge of returning the response from users_ms to the api_gw, therefore it will send the token to the corresponding state.
- **products_response**(**req**): It will be the function in charge of returning the response from products_ms to api_gw, therefore it will send the token to the corresponding state.

- **transactions_response(req)**: It will be the function in charge of returning the response from transactions_ms to the api_gw, therefore it will send the token to the corresponding state.
- **success_transaction(req)**: It will be the function in charge of validating that the request is completed successfully and triggers the transition to the success_transaction state.
- **failure_transaction(req)**: It will be the function in charge of validating that the request has been completed in a failed manner and triggers the transition to the failure_transaction state.

As can be seen, all the functions receive as a parameter the *req* variable that represents the current value of the request in order to execute the corresponding logic.

Therefore, the finally implemented model can be seen in Figure 6-4:

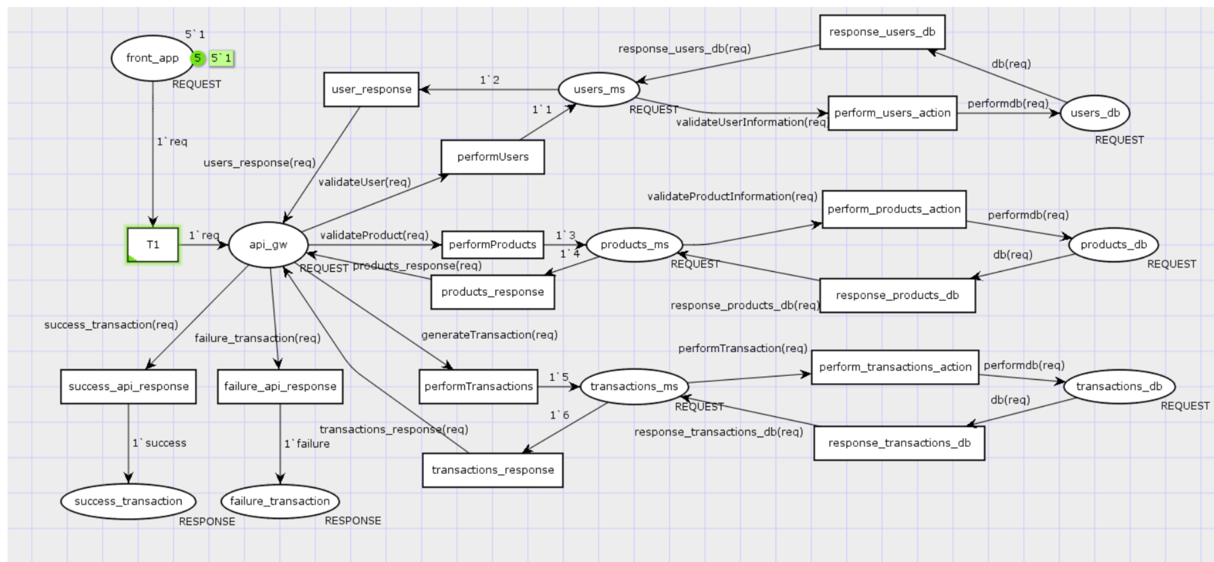

Figure 6-4: Case study - Implemented Model - CPN Tools.

6.1.7 Verification

The verification used is the one provided by the CPN Tools tool called State Space. CPN Tools provides the complete execution performing verification through the possible states of the system.

In our case, the validation was executed for a period of 2 hours, and the maximum number reached by the *failure_transaction* status was also determined. In this way, the ideal scenario is that the number of times that state is reached is 0, if it is at least 1, it implies that one of the requests was not answered correctly.

When carrying out the execution, the following results were obtained:

```
------------------------------------------------------------

  State Space
     Nodes:  80786
     Arcs:   868212
     Secs:   7200
     Status: Partial

  Scc Graph
     Nodes:  80786
     Arcs:   868212
     Secs:   28


 Boundedness Properties
------------------------------------------------------------

  Best Integer Bounds
                            Upper      Lower
     StudyCase'api_gw 1      5          0
     StudyCase'failure_transaction 1
                            3          0
     StudyCase'front_app 1   5          0
     StudyCase'products_db 1 2          0
     StudyCase'products_ms 1 3          0
     StudyCase'success_transaction 1
                            1          0
     StudyCase'transactions_db 1
                            1          0
     StudyCase'transactions_ms 1
                            2          0
     StudyCase'users_db 1    5          0
     StudyCase'users_ms 1    5          0
```

Figure 6-5: Case study - Verification results.

As can be seen in the results, the state of *failure_transaction* reached a maximum of 3 requests in its state, therefore it can be determined that with the error that occurs in user_ms it is affecting the availability of the system, therefore **the proposed architecture, in the proposed scenarios, the quality attribute of high availability is not ensured**.

6.2 Study Case with Architectural Tactics

In this case, it is decided to use the same base case study, applying architectural tactics in order to validate the results and compare them to demonstrate the effectiveness of the applied tactics. In this way, it is necessary to perform the same steps illustrated in the verification model with the proposed new software system.

In this case, 2 architectural tactics will be applied: **Replication** and **Retry**.

6.2.1 System Specification

The system will be defined according to the following view of components and connectors:

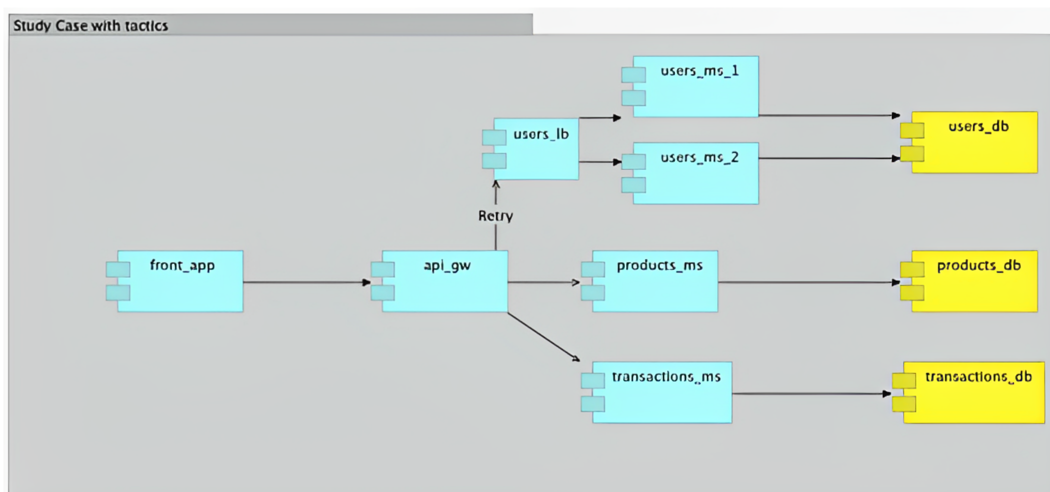

Figure 6-6: Case study with tactics - Components & Connectors View.

As shown in Graph 6-6, the system will consist of eight components, **front_app** corresponds to the component in charge of interacting with the user, **api_gw** to perform the system orchestration, **users_ms**, **products_ms** and **transactions_ms** will have specific logic system, **users_db**, **product_db** and **transactions_db** responsible for system persistence and **users_lb** in charge of balancing the loads of **users_ms**.

6.2.2 Scenario

In the case of this study, variables that are directly carried out with the availability of the system will be taken into account. For this, being a basic transactional system, it will be composed of requests that must be resolved correctly, in this way the variables that will be taken into account

in this case study are the following:

- **Number of requests**: These are the number of requests that the system should attend to.
- **Probability of failure in users_ms**: It is the probability that a request is not serviced correctly by the users_ms microservice.
- **Probability of failure in auth_ms**: It is the probability that a request is not handled correctly by the auth_ms microservice.
- **Probability of failure in transactions_ms**: It is the probability that a request is not serviced correctly by the transactions_ms microservice.
- **Number of retries**: It is the number of times that the orchester will resend the requests to the microservices when they are not resolved correctly.

For illustrative purposes, a single scenario will be carried out to be verified by the proposed model, the above with the objective of emphasizing the results. Thus, the scenario to be tested is the following:

Scenario 1:

- Number of requests: 5
- Probability of failure in users_ms_1: 25%.
- Probability of failure in users_ms_2: 0%.
- Probability of failure in products_ms: 0%.
- Probability of failure in transactions_ms: 0%.
- Number of retries: n

where **n** implies an unlimited number.

Constraints to validate:

- failure transaction always at 0.

6.2.3 System Properties

In the event of a failure, the **api_gw** component will forward the request again to **users_ms** in order to be processed correctly.

The system will allow you to carry out a transaction using the following flow:

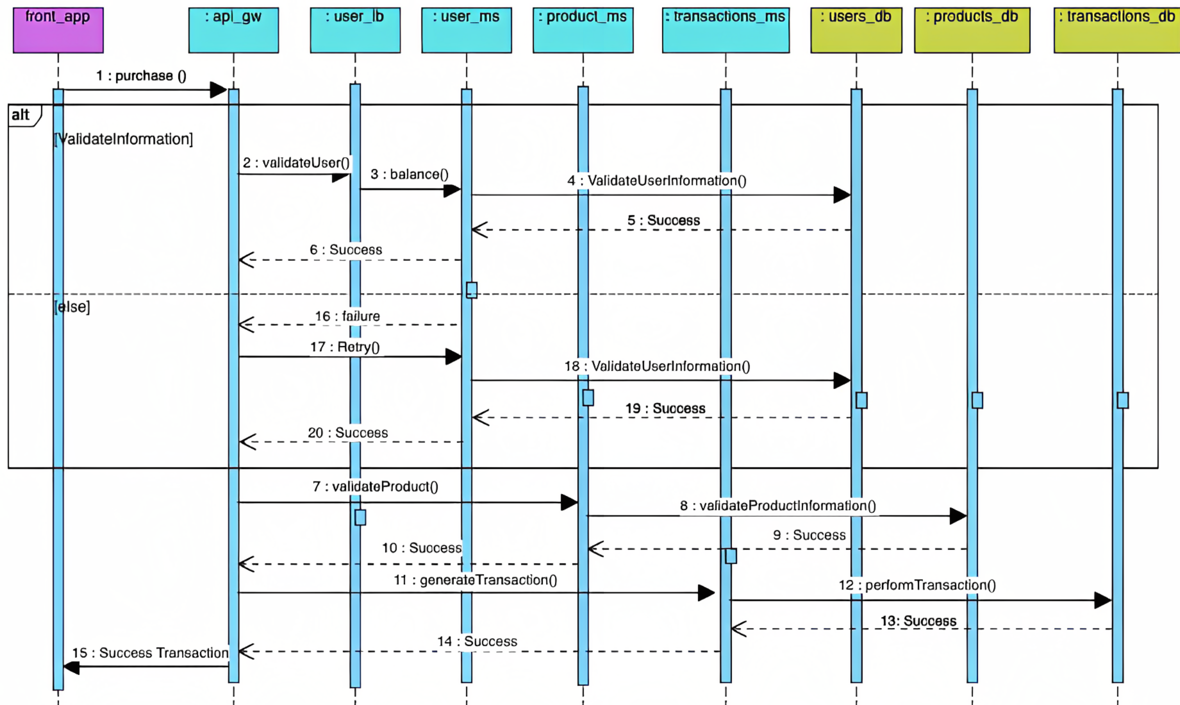

Figure 6-7: Case study with tactics - Sequence diagram

6.2.4 Structural definition

The system components are as follows:

- **front_app**: Will be the component in charge of making the requests to the system, because it is the component through which the users of the system make use of it.
- **api_gw**: Will be in charge of managing the orchestration of each of the system requests between the different microservices.
- **users_ms**: Will be in charge of managing user information.
- **users_lb**: It will be in charge of balancing the loads of **users_ms**.
- **product_ms**: Will be in charge of managing product information.
- **transactions_ms**: It will be in charge of carrying out the transactional operations of the system.
- **users_db**: Will be in charge of the persistence of system user data.
- **products_db**: Will be in charge of the persistence of system product data.

- **transactions_db**: It will be in charge of the persistence of the transactions carried out in the system.

The implementation of the system in **Sarch** would be the following:

```
architecture {
  software_system : CaseStudyWithTactics;
  author : CamiloDajer;
  architectural_views{
    component_and_connector_view ::
    elements {
      component_types{
        front; orchestrator; logical; db; lb;
      }
      component front front_ms ;
      component orchestrator api_gw ;
      component logical users_ms ;
      component logical products_ms ;
      component logical transactions_ms ;
      component db users_db ;
      component db products_db ;
      component db transactions_db ;
      component lb users_lb;
      connector purchase ;
      connector validateUser ;
      connector validateProduct ;
      connector generateTransaction ;
      connector validateUserInformation ;
      connector validateProductInformation ;
      connector performTransaction ;
      connector balance ;
    }
    relations {
      attachment( purchase:front_ms,api_gw );
      attachment( validateUser:api_gw,users_lb );
      attachment( validateProduct:api_gw,products_ms );
      attachment( generateTransaction:api_gw,transactions_ms );
      attachment( validateUserInformation:users_ms,users_db );
      attachment( validateProductInformation:products_ms,products_db );
      attachment( performTransaction:transactions_ms,transactions_db );
      attachment( balance:users_lb,users_ms );
    }::
```

```
  }
}
```

6.2.5 Behavioural definition

The defined functions of the behavior of the system will be the following:

- $purchase(front_app, api_gw, req) = return(success_transaction, failure_transaction)$
- $validateUser(api_gw, users_lb, req) = return(success, failure)$
- $validateUserInformation(users_ms, users_db, req) = return(success)$
- $validateProduct(api_gw, products_ms, req) = return(success)$
- $validateProductInformation(products_ms, products_ms, req) = return(success)$
- $generateTransaction(api_hw, transactions_ms, req) = return(success)$
- $performTransaction(transactions_ms, transactions_db, req) = return(success)$
- $balance(users_lb, users_ms, req) = return(success, failure)$
- $retry(api_gw, users_lb, req) = return(success)$

6.2.6 Implementation

We proceed to create the petri nets in CPN Tools, implementing the following declarations in the system:

```
▸Standard priorities
▾Standard declarations
  ▾colset BOOL = bool;
  ▾colset INT = int;
  ▾colset INTINF = intinf;
  ▾colset TIME = time;
  ▾colset REAL = real;
  ▾colset STRING = string;
  ▾colset REQUEST = int with 1..12 ;
  ▾var req : REQUEST;
  ▾colset RESPONSE = with success | failure ;
  ▾fun validateUser(req) =  1`1;
  ▾fun validateProduct (req) = 1`3;
  ▾fun generateTransaction (req) = 1`5;
  ▾fun validateUserInformation (req) = 1`1;
  ▾fun validateProductInformation(req) = 1`3;
  ▾fun performTransaction(req) = 1`5;
  ▾fun performdb(req) =  1`req;
  ▾fun db(req) = 1`req;
  ▾fun response_users_db(req) = if req = 1 then
    1`2
    else empty;
  ▾fun response_users_db_2(req) = if req = 10 then
    1`11
    else empty;
  ▾fun response_products_db(req) = 1`4;
  ▾fun response_transactions_db(req) = 1`6;
  ▾fun purchase(req) = 1`1;
  ▾fun perform(req) = 1`1;
  ▾fun users_response(req) = if discrete(1,4) = 1 then
     1`10
    else
     1`3;
  ▾fun products_response(req) = 1`5;
  ▾fun transactions_response(req) = 1`8;
  ▾fun success_transaction(req) =  1`8;
  ▾fun failure_transaction(req) =  1`9;
  ▾fun retry(req) =  1`10;
  ▾fun balance(req) =  if discrete(1,2) = 1 then
    1`1
    else
    1`10;
  ▾fun lb(req) =  if req = 1 then
    1`1
    else
    empty;
  ▾fun lb2(req) =  if req = 10 then
    1`10
    else
    empty;
```

Figure 6-8: Case study with tactics - Standard declarations - CPN Tools.

In this way, the types, variables, and functions are declared and supported in a colored Petri net.

6.2.6.1 Types:

- **REQUEST**: It will be composed of a set of integers from 1 to 10, in this way each number will represent the state of the request and who will be the next component to process the request.

- **RESPONSE**: It will be composed of 2 types, success and failure, which will represent the final status of the request.

6.2.6.2 Variables:

- **req**: Represents each of the requests made by the system.

6.2.6.3 Functions:

- **validateUser(req)**: It will be the function in charge of validating that the request must be processed by users_ms, therefore it will send the token to the corresponding state.
- **validateProduct (req)**: It will be the function in charge of validating that the request must be processed by products_ms, therefore it will send the token to the corresponding state.
- **generateTransaction (req)**: It will be the function in charge of validating that the request must be processed by transactions_ms, therefore it will send the token to the corresponding state.
- **validateUserInformation (req)**: It will be the function in charge of validating that the request must be processed by users_db, therefore it will send the token to the corresponding state.
- **validateProductInformation(req)**: It will be the function in charge of validating that the request must be processed by products_db, therefore it will send the token to the corresponding state.
- **performTransaction(req)**: It will be the function in charge of validating that the request must be processed by transactions_db, therefore it will send the token to the corresponding state.
- **users_response(req)**: It will be the function in charge of returning the response from users_ms to the api_gw, therefore it will send the token to the corresponding state.
- **products_response(req)**: It will be the function in charge of returning the response from products_ms to api_gw, therefore it will send the token to the corresponding state.
- **transactions_response(req)**: It will be the function in charge of returning the response from transactions_ms to the api_gw, therefore it will send the token to the corresponding state.
- **success_transaction(req)**: It will be the function in charge of validating that the request is completed successfully and triggers the transition to the success_transaction state.
- **failure_transaction(req)**: It will be the function in charge of validating that the request has been completed in a failed wat and triggers the transition to the failure_transaction state.

- **balance(req)**: It will be the function in charge of balancing the requests between the 2 instances of the **users_ms** component.
- **retry(req)**: Will be the function in charge of forwarding the negative requests from**users_-ms_1** to **users_ms_2**.

Finally implemented model can be seen in Figure 6-9:

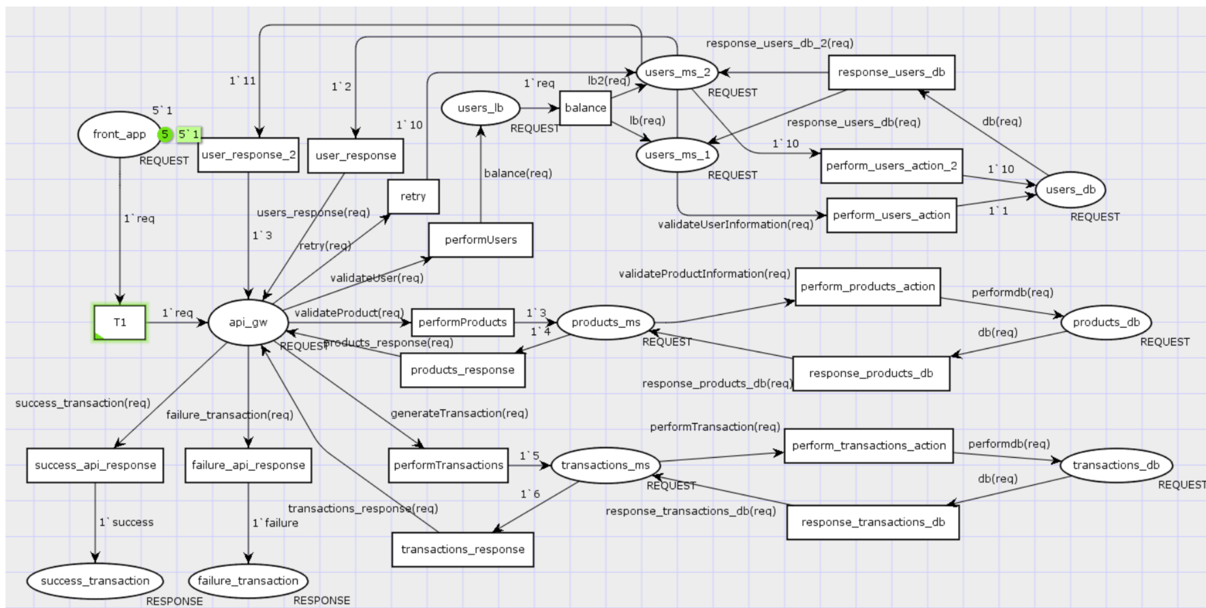

Figure 6-9: Case study with tactics - Implemented Model - CPN Tools.

6.2.7 Verification

The verification used will be the one provided by the CPN Tools tool called State Space. CPN Tools provides the complete execution performing verification through the possible states of the system.

In our case, the validation was executed for a period of 2 hours, and the maximum number reached by the *failure_transaction* status was also determined. In this way, the ideal scenario is that the number of times that state is reached is 0, if it is at least 1, it implies that one of the requests was not answered correctly breaching the restriction.

When carrying out the execution, the following results were obtained:

As can be seen in the results, the state of *failure_transaction* reached a maximum of 0 requests in its state. Additionally, the analysis of state spaces allows us to determine transitions that cannot

```
------------------------------------------------------------------------

 State Space
    Nodes:  73143
    Arcs:   1043816
    Secs:   7320
    Status: Partial

 Scc Graph
    Nodes:  73143
    Arcs:   1043816
    Secs:   41


Boundedness Properties
------------------------------------------------------------------------

 Best Integer Bounds
                             Upper      Lower
    StudyCase'api_gw 1       5          0
    StudyCase'failure_transaction 1
                             0          0
    StudyCase'front_app 1    5          0
    StudyCase'products_db 1  3          0
    StudyCase'products_ms 1  3          0
    StudyCase'success_transaction 1
                             1          0
    StudyCase'transactions_db 1
                             2          0
    StudyCase'transactions_ms 1
                             2          0
    StudyCase'users_db 1     5          0
    StudyCase'users_lb 1     5          0
    StudyCase'users_ms_1 1   5          0
    StudyCase'users_ms_2 1   5          0
```

Figure 6-10: Case study with tactics- Verification results.

be reached no matter how many iterations are carried out. For this verification, the results were the following:

```
Liveness Properties
------------------------------------------------

 Dead Transition Instances
    StudyCase'failure_api_response 1
```

Figure 6-11: Case study with tactics- Liveness results.

In this way, **it can be determined that the implemented architectural tactics will not allow a request to reach the failure_transaction state, ensuring the high availability quality attribute under the assumed scenario and properties.**

6.3 Results Comparison

The comparison of verification results can be observed in Table 6-1:

	Base Study Case	**Study Case with Architectural Tactics**
Scenario		
Number of Request	5	5
Probability of failure in users_ms	25%	25%
Probability of failure in products_ms:	0%	0%
Probability of failure in transactions_ms:	0%	0%
Number of retries	0	n
Results		
Max Failure Requests	3	0
Dead Transaction Instances	None	StudyCase'failure_api_response
Ensure High Availability	**No**	**Yes**

The results obtained by the verification through the state space graph allowed to determine the maximum number of requests that were incorrectly resulted in the software system proposed under similar conditions, where one had not implemented high availability architectural tactics and the other implemented 2 tactics: Replication and Retry.

Both scenarios had a probability of failure in the **users_ms** component of 25%. The first proposed architecture did not handle the error and that is why in one of the scenarios simulated

by the tool, the maximum number of failed requests was 3. On the other hand, the architecture proposed with the architectural tactics of ensuring high availability allowed not only to have a replica of the microservice that presented failures, but also to retry the failed requests and thus resolve all of them correctly. For this reason it can be observed that in the simulation executed the maximum number of failed requests in the explored spaces was 0.

In this degree, it is possible to ensure that the architecture with architectural tactics presents a more suitable structure, behavior and properties than the architecture without architectural tactics for the assurance of the quality attribute of high availability.

Finally, the liveness property through Dead Transition Instances allows us to be sure about the results delivered because it allows us to determine those transitions that will not be executed and in this way to have, through simulation and formal verification, support for architectural decision-making based on formal results.

7 Conclusions and Future Work

7.1 Conclusions

Based on this work and related works, a formal verification approach was proposed for the assurance of quality attributes in software architectures. First, an analysis approach was carried out for the availability tactics that structurally modify the system to determine the architectural implications of each tactic when implemented. On the other hand, a formal verification approach was proposed and tested by implementing a case study in a software system that employed a microservices architecture. The architecture used some tactics to ensure high availability in specific scenarios. Through Coloured Petri Nets, formal verification was achieved through an exploration of state spaces. This way, it was determined that a condition of a software system is correctly satisfied.

Therefore, this thesis, through the proposed model, allows for:

- Define a set of elements that allow to formally specify software systems.
- Formally specify a software system.
- Design a formal description model.
- Validate the formal model defined through formal verification processes.

This model is extensible beyond MSA because the steps for generating the model do not have any specific restrictions on the architecture. Any architecture can be defined through a view of components and connectors. However, the validation of quality attributes may be limited if the quality attributes are subjective properties. Thus, a model is provided that allows software systems to be defined using formal resources and their formal verification. This way, it can contribute to architectural decision-making through formal support.

7.2 Future Work

As future work, there is the possibility of working on different sections through what is presented in this thesis. First, the implementation of models and case studies can be carried out where the assurance of other quality attributes can be validated. Second, the use of petri nets in

other contexts is software architecture that allows the analysis and verification of properties of different architectural styles and patterns. Third, an extension of the Sarch language is proposed to be able to support formal verifications, and from the structural and behavioral definition, analyze software architectures from a perspective of quality attributes, or specific properties of the software system.

Bibliography

[1] Ahmad, Aakash ; Babar, Muhammad A.: Software architectures for robotic systems: A systematic mapping study. In: *Journal of Systems and Software* 122 (2016), dec, S. 16–39. http://dx.doi.org/10.1016/j.jss.2016.08.039. – DOI 10.1016/j.jss.2016.08.039. – ISSN 01641212

[2] Babar, Muhammad A. ; Kitchenham, Barbara ; Zhu, Liming ; Gorton, Ian ; Jeffery, Ross: An empirical study of groupware support for distributed software architecture evaluation process. In: *Journal of Systems and Software* 79 (2006), jul, Nr. 7, S. 912–925. http://dx.doi.org/10.1016/j.jss.2005.06.043. – DOI 10.1016/j.jss.2005.06.043. – ISSN 01641212

[3] Barber, K. S. ; Graser, Thomas ; Holt, Jim: Providing early feedback in the development cycle through automated application of model checking to software architectures. In: *Proceedings - 16th Annual International Conference on Automated Software Engineering, ASE 2001*, Institute of Electrical and Electronics Engineers (IEEE), aug 2001. – ISBN 076951426X, S. 341–345

[4] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software Architecture in Practice Second Edition Third Edition*. 2013. – 576 S. https://www.oreilly.com/library/view/software-architecture-in/9780132942799/. – ISBN 0321815736

[5] Brito, Patrick H. ; De Lemos, Rogério ; Rubira, Cecília M.F. ; Martins, Eliane: Architecting fault tolerance with exception handling: Verification and validation. In: *Journal of Computer Science and Technology* 24 (2009), mar, Nr. 2, S. 212–237. http://dx.doi.org/10.1007/s11390-009-9219-2. – DOI 10.1007/s11390-009-9219-2. – ISSN 10009000

[6] Camilli, Matteo ; Gargantini, Angelo ; Scandurra, Patrizia ; Bellettini, Carlo: Event-based runtime verification of temporal properties using time basic Petri nets. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 10227 LNCS, Springer Verlag, 2017. – ISBN 9783319572871, S. 115–130

[7] Clements, Paul ; Garlan, David ; Little, Reed ; Nord, Robert ; Stafford, Judith: Documenting software architectures: Views and beyond. In: *Proceedings - International Conference on Software Engineering*, 2003. – ISBN 978-1-491-95035-7, 740–741

[8] Czepa, Christoph ; Tran, Huy ; Zdun, Uwe ; Tran, Thanh ; Weiss, Erhard ; Ruhsam, Christoph: Reduction techniques for efficient behavioral model checking in adaptive case

management. In: *Proceedings of the ACM Symposium on Applied Computing* Bd. Part F1280, Association for Computing Machinery, apr 2017. – ISBN 9781450344869, S. 719–726

[9] Dehkordi, Zohreh S. ; Harounabadi, Ali ; Parsa, Saeed: Evaluation of software architecture using fuzzy color Petri net. In: *Management Science Letters* 3 (2013), feb, Nr. 2, S. 555–562. http://dx.doi.org/10.5267/J.MSL.2012.12.016. – DOI 10.5267/J.MSL.2012.12.016. – ISSN 19239335

[10] Demirli, Elif ; Tekinerdogan, Bedir: Software language engineering of architectural viewpoints. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 6903 LNCS, AMSE Press, mar 2011. – ISBN 9783642237973, S. 336–343

[11] Di Francesco, Paolo ; Lago, Patricia ; Malavolta, Ivano: Architecting with microservices: A systematic mapping study. In: *Journal of Systems and Software* 150 (2019), apr, S. 77–97. http://dx.doi.org/10.1016/j.jss.2019.01.001. – DOI 10.1016/j.jss.2019.01.001. – ISSN 01641212

[12] Dobrica, Liliana ; Niemelá, Eila: *A survey on software architecture analysis methods*

[13] Dragoni, Nicola ; Giallorenzo, Saverio ; Lafuente, Alberto L. ; Mazzara, Manuel ; Montesi, Fabrizio ; Mustafin, Ruslan ; Safina, Larisa: Microservices: Yesterday, today, and tomorrow. Version: nov 2017. http://dx.doi.org/10.1007/978-3-319-67425-4_12. In: *Present and Ulterior Software Engineering*. Springer International Publishing, nov 2017. – DOI 10.1007/978-3-319-67425-4_12. – ISBN 9783319674254, S. 195–216

[14] Ferrari, Remo ; Madhavji, Nazim H.: Software architecting without requirements knowledge and experience: What are the repercussions? In: *Journal of Systems and Software* 81 (2008), sep, Nr. 9, S. 1470–1490. http://dx.doi.org/10.1016/j.jss.2007.12.764. – DOI 10.1016/j.jss.2007.12.764. – ISSN 01641212

[15] Ghezzi, Carlo ; Mandrioli, Dino ; Morasca, Sandro ; Pezze, Maura: A unified high-level Petri net formalism for time-critical systems. In: *IEEE Transactions on Software Engineering* 17 (1991), feb, Nr. 2, S. 160–172. http://dx.doi.org/10.1109/32.67597. – DOI 10.1109/32.67597. – ISSN 00985589

[16] Giannakopoulou, Dimitra: Model Checking for Concurrent Software Architecutures. In: *Department of Computing* (1998), Nr. January. http://www.doc.ic.ac.uk/{~}dg1/tracta/papers/thesis.pdf

[17] Hansen, Klaus M. ; Jonasson, Kristjan ; Neukirchen, Helmut: An empirical study of software architectures' effect on product quality. In: *Journal of Systems and Software* 84 (2011), jul, Nr. 7, S. 1233–1243. http://dx.doi.org/10.1016/j.jss.2011.02.037. – DOI 10.1016/j.jss.2011.02.037. – ISSN 01641212

[18] Haoues, Mariem ; Sellami, Asma ; Ben-Abdallah, Hanêne ; Cheikhi, Laila: A guideline for software architecture selection based on ISO 25010 quality related characteristics. In: *International Journal of System Assurance Engineering and Management* 8 (2017), nov, S. 886–909. http://dx.doi.org/10.1007/s13198-016-0546-8. – DOI 10.1007/s13198–016–0546–8. – ISSN 09764348

[19] Harrison, Neil B. ; Avgeriou, Paris: How do architecture patterns and tactics interact? A model and annotation. In: *Journal of Systems and Software* 83 (2010), oct, Nr. 10, S. 1735–1758. http://dx.doi.org/10.1016/j.jss.2010.04.067. – DOI 10.1016/j.jss.2010.04.067. – ISSN 01641212

[20] Hocking, Ashlie B. ; Knight, John C. ; Aiello, M. A. ; Shiraishi, Shin'Ichi: Formal Verification in Model Based Development. In: *SAE Technical Papers* 2015-April (2015), apr, Nr. April. http://dx.doi.org/10.4271/2015-01-0260. – DOI 10.4271/2015–01–0260. – ISSN 01487191

[21] Inverardi, Paola ; Muccini, Henry ; Pelliccione, Patrizio: Automated check of architectural models consistency using SPIN. In: *Proceedings - 16th Annual International Conference on Automated Software Engineering, ASE 2001*, Institute of Electrical and Electronics Engineers (IEEE), aug 2001. – ISBN 076951426X, S. 346–349

[22] Jensen, Kurt ; Kristensen, Lars M.: Coloured Petri Nets: Modelling and validation of concurrent systems. In: *Coloured Petri Nets: Modelling and Validation of Concurrent Systems* (2009), 1–384. http://dx.doi.org/10.1007/B95112. – DOI 10.1007/B95112. ISBN 9783642002830

[23] Kazman, Rick ; Bass, Len ; Klein, Mark ; Lattanze, Tony ; Northrop, Linda: *A basis for analyzing software architecture analysis methods*

[24] Lewis, Grace ; Lago, Patricia ; Echeverría, Sebastián ; Simoens, Pieter: A tale of three systems: Case studies on the application of architectural tactics for cyber-foraging. In: *Future Generation Computer Systems* 96 (2019), jul, S. 119–147. http://dx.doi.org/10.1016/j.future.2019.01.052. – DOI 10.1016/j.future.2019.01.052. – ISSN 0167739X

[25] Li, J. J. ; Horgan, J. R.: Applying formal description techniques to software architectural design. In: *Computer Communications* 23 (2000), jul, Nr. 12, S. 1169–1178. http://dx.doi.org/10.1016/S0140-3664(99)00244-3. – DOI 10.1016/S0140–3664(99)00244–3. – ISSN 01403664

[26] Li, Shanshan ; Zhang, He ; Jia, Zijia ; Zhong, Chenxing ; Zhang, Cheng ; Shan, Zhihao ; Shen, Jinfeng ; Babar, Muhammad A.: *Understanding and addressing quality attributes of microservices architecture: A Systematic literature review*

[27] Magableh, Basel ; Almiani, Muder: A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster. In: *Advances in Intelligent Systems and Computing* Bd. 926, Springer Verlag, 2020. – ISBN 9783030150310, S. 846–858

[28] Mohsin, Ahmad ; Janjua, Naeem K.: A review and future directions of SOA-based software architecture modeling approaches for System of Systems. In: *Service Oriented Computing and Applications* Bd. 12, Springer London, dec 2018. – ISSN 18632394, S. 183–200

[29] Neri, Davide ; Soldani, Jacopo ; Zimmermann, Olaf ; Brogi, Antonio: Design principles, architectural smells and refactorings for microservices: a multivocal review. In: *Software-Intensive Cyber-Physical Systems* Bd. 35, Springer Science and Business Media LLC, sep 2020. – ISSN 25248529, S. 3–15

[30] Newman, Sam ; .: Building microservices: Designing fine-grained systems (second edition). (2021), S. 1–10. ISBN 9781492034025

[31] Noyle, Brian (DTS A. ; Bouwman, Dave (DTS A.: From Design to Deployment. In: *ArcUser* (2010), S. 45–47

[32] Parizek, Pavel ; Plasil, Frantisek: Specification and Generation of Environment for Model Checking of Software Components. In: *Electronic Notes in Theoretical Computer Science* 176 (2007), may, Nr. 2, S. 143–154. `http://dx.doi.org/10.1016/j.entcs.2006.02.036`. – DOI 10.1016/j.entcs.2006.02.036. – ISSN 15710661

[33] Rim, Kee W. ; Min, Byoung J. ; Shin, Sang S.: An Architecture for High Availability Multi-user Systems. In: *Computer Communications* 20 (1997), may, Nr. 3, S. 197–205. `http://dx.doi.org/10.1016/S0140-3664(97)00007-8`. – DOI 10.1016/S0140-3664(97)00007-8. – ISSN 01403664

[34] Rodano, Matthew ; Giammarco, Kristin: A formal method for evaluation of a modeled system architecture. In: *Procedia Computer Science* Bd. 20, Elsevier, 2013. – ISSN 18770509, S. 210–215

[35] Roggenbach, Markus ; Cerone, Antonio ; Schlingloff, Bernd-Holger ; Schneider, Gerardo ; Shaikh, Siraj A.: Formal methods for software engineering : languages, methods, application domains. ISBN 3030387992

[36] Sabry, Ahmed E.: Decision Model for Software Architectural Tactics Selection Based on Quality Attributes Requirements. In: *Procedia Computer Science* Bd. 65, Elsevier, 2015. – ISSN 18770509, S. 422–431

[37] Salama, Maria ; Bahsoon, Rami: Analysing and modelling runtime architectural stability for self-adaptive software. In: *Journal of Systems and Software* 133 (2017), nov, S. 95–112. `http://dx.doi.org/10.1016/j.jss.2017.07.041`. – DOI 10.1016/j.jss.2017.07.041. – ISSN 01641212

[38] Siddiqui, Junaid H. ; Rauf, Affan ; Ghafoor, Maryam A.: Advances in Software Model Checking. Version: jan 2018. http://dx.doi.org/10.1016/bs.adcom.2017.11.001. In: *Advances in Computers* Bd. 108. Academic Press Inc., jan 2018. – DOI 10.1016/bs.adcom.2017.11.001. – ISBN 9780128151198, S. 59–89

[39] Siegel, Stephen F. ; Avrunin, George S.: MODELING MPI PROGRAMS FOR VERIFICATION.

[40] Sipser, Michael: Introduction to the Theory of Computation. In: *ACM SIGACT News* 27 (1996), Nr. 1, S. 27–29. http://dx.doi.org/10.1145/230514.571645. – DOI 10.1145/230514.571645. – ISBN 9788131525296

[41] Svahnberg, Mikael ; Wohlin, Claes ; Lundberg, Lars ; Mattsson, Michael: A quality-driven decision-support method for identifying software architecture candidates. In: *International Journal of Software Engineering and Knowledge Engineering* 13 (2003), oct, Nr. 5, S. 547–573. http://dx.doi.org/10.1142/S0218194003001421. – DOI 10.1142/S0218194003001421. – ISSN 02181940

[42] Taleb-Berrouane, Mohammed ; Khan, Faisal ; Amyotte, Paul: Bayesian Stochastic Petri Nets (BSPN) - A new modelling tool for dynamic safety and reliability analysis. In: *Reliability Engineering and System Safety* 193 (2020), jan, S. 106587. http://dx.doi.org/10.1016/j.ress.2019.106587. – DOI 10.1016/j.ress.2019.106587. – ISSN 09518320

[43] Tarullo, Michael: Software architecture theory and practice. In: *CrossTalk* 24 (2011), Nr. 6, S. 11–15. ISBN 0470167742

[44] Tekinerdogan, B. ; Ozcan, O.: Architectural Perspective for Design and Analysis of Scalable Software as a Service Architectures. Version: 2017. http://dx.doi.org/10.1016/B978-0-12-802855-1.00010-1. In: *Managing Trade-offs in Adaptable Software Architectures*. Elsevier, 2017. – DOI 10.1016/B978–0–12–802855–1.00010–1. – ISBN 9780128028551, S. 223–245

[45] Tekinerdogan, Bedir ; Sozer, Hasan ; Aksit, Mehmet: Software architecture reliability analysis using failure scenarios. In: *Journal of Systems and Software* 81 (2008), apr, Nr. 4, S. 558–575. http://dx.doi.org/10.1016/j.jss.2007.10.029. – DOI 10.1016/j.jss.2007.10.029. – ISSN 01641212

[46] Torvekar, Nupura ; Game, Pravin S.: Microservices and Its Applications An Overview. In: *International Journal of Computer Sciences and Engineering* 7 (2019), apr, Nr. 4, S. 803–809. http://dx.doi.org/10.26438/ijcse/v7i4.803809. – DOI 10.26438/ijcse/v7i4.803809

[47] ul Muram, Faiz ; Tran, Huy ; Zdun, Uwe: Supporting automated containment checking of software behavioural models using model transformations and model checking. In: *Science of Computer Programming* 174 (2019), apr, S. 38–71. http://dx.doi.org/10.1016/j.scico.2019.01.005. – DOI 10.1016/j.scico.2019.01.005. – ISSN 01676423

[48] Vergara-Vargas, Jeisson ; Umana-Acosta, Henry: A model-driven deployment approach for scaling distributed software architectures on a cloud computing platform. In: *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS* 2017-November (2018), apr, S. 99–103. http://dx.doi.org/10.1109/ICSESS.2017.8342873. – DOI 10.1109/ICSESS.2017.8342873. – ISBN 9781538645703

[49] Zdun, Uwe ; Navarro, Elena ; Leymann, Frank: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 10601 LNCS, Springer Verlag, 2017. – ISBN 9783319690346, S. 411–429

[50] Zdun, Uwe ; Stocker, Mirko ; Zimmermann, Olaf ; Pautasso, Cesare ; Lübke, Daniel: Guiding architectural decision making on quality aspects in microservice APIs. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 11236 LNCS, Springer Verlag, 2018. – ISBN 9783030035952, S. 73–89

[51] Zhang, Pengcheng ; Muccini, Henry ; Li, Bixin: A classification and comparison of model checking software architecture techniques. In: *Journal of Systems and Software* 83 (2010), may, Nr. 5, S. 723–744. http://dx.doi.org/10.1016/j.jss.2009.11.709. – DOI 10.1016/j.jss.2009.11.709. – ISSN 01641212

[52] Zhou, Xin ; Li, Shanshan ; Cao, Lingli ; Zhang, He ; Jia, Zijia ; Zhong, Chenxing ; Shan, Zhihao ; Babar, Muhammad A.: Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. In: *Journal of Systems and Software* 195 (2023), 111521. http://dx.doi.org/10.1016/j.jss.2022.111521. – DOI 10.1016/j.jss.2022.111521. – ISSN 01641212