MSArch: A Graphical Modeling Language for MicroService Architecture Design

Abstract—**Microservices design is a crucial part of cloud-native application developments. It is not a trivial task in practice. It not only requires considering how to decompose a large and complex software system into high cohesion and low coupling components but also needs carefully designing the dynamic non-functional control policy of microservices such as traffic management, circuit breaker, and degradation to support high concurrency, security, and reliability. However, there is no modeling language for supporting the system architect to design those policies in microservice design, making it a great challenge for the developer to implement those policies directly into the code. In this paper, we propose a modeling language named MSArch for designing microservices architecture and dynamic control policy based on a model-driven approach. The meta-model and diagram of MSArch are proposed by investigating and referring to the advantages of state-of-the-art microservice models and the middleware of dynamic control such as Istio. We use two industrial case studies from the previous research to demonstrate the validity and usability of the proposed model. The results show that our proposed model has a lower learning curve, is easily understandable within ten mines of training, and it can cover an average of 98.24% of the case studies. The result is satisfactory. MSArch can be further extended and refined for modeling the microservice in the software industry.**

Index Terms—**Microservice, Microservice Architecture, Microservice Design, MSArch, MDA**

I. Introduction

As a system construction method, microservice has been widely used in recent years. It changes the situation that it is difficult to change once a single software architecture is established. Unlike the previous monolithic architecture system, microservice has advantages that cannot be ignored. In the past, the single architecture system was relatively easy in the initial stage of architecture design, but as time goes by, the system will become more and more complex due to the increase in requirements, which will be difficult to maintain. And because the system is a single construction, it will be difficult to expand, unable to use other technologies, and other problems in the later stage. Microservice architecture can use multiple technologies in a system. Breaking the system into small pieces can be easy to develop and maintain and has strong scalability.

Compared with the traditional single architecture system, the system after microservitization has the following advantages [13]: 1) The software sub-module is highly autonomous, and different programming languages and data storage technologies can be used for different application scenarios to give full play to the strengths of various technologies and realize complementary advantages; 2) Microservitization enables system sub-modules to achieve miniaturization and single responsibility, which makes it easier for the development teams to achieve miniaturization and agility, thus achieving efficient continuous development, integration, and deployment of projects, and ultimately reducing the total cost of software development and maintenance; 3) Microservitization enables the system to make full use of the elastic characteristics of on-demand distribution in the cloud computing environment to achieve high concurrency, high expansion, and high reliability of the software system, and then provide users with efficient and stable software services.

Systems that use microservice architectures often have more complex internal structures and underlying strategies to meet non-functional requirements, and modeling microservice architectures makes it easier for architects to conceive of the architecture. When collaborating with more than one person, using models makes it easier for architects to communicate the patchwork architecture clearly and quickly after they have designed their own parts and facilitated subsequent modifications. There are several models that can describe microservices. However, there is no universal microservice modeling language. Because of its principle of single responsibility, microservice architectures typically have more internal services and more complex structures. When designing architecture, software architects spend a considerable amount of time thinking about the dependencies between services and the structural descriptions within them. In the absence of a common microservice modeling language, architects often describe the architecture in their own comfortable way when designing it. As a result, designers need to spend a certain amount of time learning and understanding others' description habits when communicating, which increases the design time and reduces the development efficiency. This can also cause subsequent members to fail to fully understand the designer's architecture and make changes without written or verbal instructions.

The preferred architecture for cloud-native is the microservice architecture. Cloud-native encourages applications to be agile, reliable, resilient, scalable, and constantly updated. In microservices, the reliability of the system is generally improved by improving elasticity, but the existing microservice architecture models seldom pay attention to the elasticity strategy of microservices, which is often an important part of the user experience of the microservice system. The microservice architecture also needs to describe many concurrent or other microservice features, while the existing models cannot express the different policy relationships between each service and the characteristics of the microservice itself succinctly.

Many people believe that microservices have some similarities to Service-Oriented Architecture (SOA). Mazzara agrees that microservices as an architectural style derived from SOA. This new style differs from monolithic architecture and classic service-oriented architecture in that it emphasizes the scalability, independence, and semantic cohesion of each unit that makes up the system [6]. Zimmermann argues that microservices are not entirely new but qualify as "SOA done right" and contain an organic approach to SOA implementation [15]. Service-oriented architecture Modeling Language (SoaML)[1], promulgated by the OMG, provides a standard approach to architecture and modeling SOA solutions using the Unified Modeling Language (UML)[2]. The SoaML standard is extended based on UML methods and can be used with existing UML tools. There is no modeling language based on SoaML for microservice, so this paper proposes a microservice architecture model MSArch based on UML and SoaML.

In conducting this research, we faced several challenges. First of all, microservices will split the entire system into multiple services, the architecture will become complex, and the model will not be able to clearly describe the internal structure of microservices and the relationship between services. Second, in microservice architectures, one service often corresponds to multiple instances. As a modeling language to describe microservice architectures, it naturally requires the ability to describe high concurrency. At the same time, a service may correspond to multiple versions. For example, micro-service needs to control the direction of user traffic when dynamic update is used. However, it is difficult to describe specific traffic policies because of multiple mapping relationships between services, instances, and versions. Finally, as a unified software modeling language, UML has a wide range of modeling capabilities and a large audience. To accommodate the modeling habits of existing traditional modelers and make it easier for users to learn, the new modeling language MSArch will be extended based on UML and SoaML. Some elements in the UML metamodel are similar to the concept of microservices but do not mean exactly the same thing, and they need to be extended. In the UML metamodel, extensions start at the root element, and each extension adds new features to the original element and gives it new relationships. Although a parent element in a metamodel can also express its subclasses, this is not conducive to accurately describing the architecture. It is difficult to pick the elements in the UML metamodel suitable for microservices, extend them, and combine them according to relationships. At the same time, SoaML is a modeling language that describes SOA architecture. Although its expression seems to apply to microservices, the concept of elements in the metamodel is different from that of microservice elements and cannot be fully applied to microservices. After comparing the similarities and differences between SOA architecture and microservice architecture, it is difficult to extract and expand the parts suitable for microservice in SoaML.

The contributions of this paper are as follows:

- We propose a modeling language named MSArch for designing microservices architecture and dynamic control policy.
- We demonstrate the validity and usability of the proposed model through two industrial case studies.
- We evaluated the proposed MSArch model and proved that our proposed model has a lower learning curve and is easily understandable.

Due to space constraints, some of the results are presented on the github project[1]. The remains of this paper are organized as follows: Section 2 summarizes the relevant work at the present stage and compares the shortcomings of the existing work. Section 3 introduces the metamodel and graph characters of microservice architecture MSArch. Section 4 evaluates the model and gives the evaluation results of the model. Section 5 summarizes the work of this paper and looks into the future.

II. RELATED WORK

As microservices tools and technologies mature, more manufacturers will choose microservices architecture to develop their business. To facilitate the development and design of microservices, in recent years, some people have begun to construct the metamodel framework of microservices. This section focuses on efforts to describe the static and dynamic models of microservices.

A. Static Architecture Model

1) AjiML: Jonas and his team introduce the Aji Modeling Tool (AjiL)[12], a research tool aimed at simplifying the cumbersome and redundant handcrafting of Microservice Architecture implementation in the development process using Model-driven Engineering (MDE). AjiL employs AjiML, a modeling language that uses two hierarchical packages to separate conceptual microservice concerns from technical concerns that rely directly on the abstract syntax layer. AjiML focuses on the conceptual aspects of the AjiML Model, while AjiMLT extends the AjiML package to enrich the different microservice types and Interface elements with technical details. AjiL offers several advantages, such as cutting the Domain Model of the overall microservice, allowing each Functional Service to have an independent Domain Model and enabling the description of both Microservice and Infrastructure. Moreover, the metamodel has been implemented at the code level. However, AjiL also has some limitations. Firstly, the Infrastructure in the metamodel is an extension of Microservice, leading to confusion in the concept. Secondly, the Entity in the metaclass Domain Model only contains Relations and Attributes, while a Domain Model is a conceptual model and should contain attributes and operations. Finally, the metamodel divides Ability into six atomic-level operations and does not involve system-level operations.

2) LEMMA: LEMMA[10][9] is a model proposed by the authors with three views: Domain Data Modeling, Service Modeling, and Operation Modeling. Domain Data Modeling provides Domain Concepts and Primitive types that are utilized

[1] https://github.com/tryCdemo/models

TABLE I: Modeling Comparation

	AjiML	LEMMA	SoaML	Petrasch[8]	Log2MS	MSArch
System functional separation	✓	✓	×	-	-	✓
Data separation	✓	×	×	-	-	✓
Non-functional requirements description	×	✓	×	-	-	✓
Dynamic interaction policy description	×	×	×	✓	×	✓

in the other two models. Service Modeling provides microServices, Protocols, and data formats for Operation Modeling. Compared to AjiL, LEMMA expands to add the visibility attribute to Interface, Operation, and Microservice to indicate whether they are visible to other services. Additionally, Operation is not limited to adding, deleting, changing, checking and other operations, and the Import metaclass in the metamodel supports Operation reuse. LEMMA is suitable for engineering complex systems with many crosscutting concerns, where the modeling perspective breaks down the complexity of the entire system into specialized, concern-specific modeling tasks. Models from different perspectives can then be combined to infer consistent system parts or reuse modeled elements across model transformations. However, LEMMA also has some drawbacks, such as the lack of modeling for storage, no separate modeling for Infrastructure, and only the type attribute in the Microservice metaclass identifying the type of the current service (functional, Infrastructure, Utility). Furthermore, Operation in the metamodel is a direct Operation on the parameter, which should be the Operation on Entity in the Domain Model.

3) SoaML: SoaML is a canonical modeling language proposed by OMG. It meets the mandatory requirements of the UPMS (UML Profile and Metamodel for Services) RFP and covers extensions to UML2.1. SoaML provides a standard way to architect and model SOA solutions using UML. SoaML can show service refinement in ServicesArchitecture and ServiceContract diagrams, but it is not a standard to describe microservice architecture. It only focuses on service communication and does not model infrastructure and other aspects of microservices.

B. Dynamic Interaction Model

Microservices need to communicate through an inter-service communication mechanism. This leads to service integration, which requires consideration of messaging and event handling during the specification of microservices. Petrasch et al. applied enterprise integration Pattern (EIP) in the context of microservice architecture design[8]. They provide a model-based approach to microservices architecture design and service integration by using formal models of UML and UML profiles. Using the domain model (bounded context) as a starting point, extend the UML component diagram so that microservices can be modeled. Enterprise integration patterns are applied to microservice diagrams to provide precise microservice specifications that can be used for further simulation, transformation, or code generation tasks. In 2022, Bo Liu et al. introduced a method to transform a single system into a micro-service architecture through execution logs[5]. They defined microservice sequence diagrams to support modeling the structure and behavior of the MSA.

Retry and CircuitBreaker are two common elastic modes utilized in microservices to mitigate the impacts of partial outages. Elasticity is an important characteristic of microservices that demonstrates their ability to handle various system disturbances that can cause service degradation[4]. Kanglin Yin and his team have defined the concept of microservice elasticity and have proposed the Microservice Resource Measurement Model (MRMM) to measure it[7]. The Mendonca team has analyzed the elastic pattern and described the functions of Retry and Circuit Breaker in this context. In addition, there are many frameworks that describe microservices today, such as Istio, k8s, and SpringCloud, all of which have some success in describing the dynamic traffic of microservices.

C. Summary of Related Work

This section compares the above models based on the following dimensions: whether system function splitting, data splitting, non-functional requirement description, and standardization are supported. Table I shows how the existing model language compares to the MSArch mentioned in this article. AjiML is a model for describing microservices that supports the separation of functional and conceptual models to represent the relationship of each functional microservice to a separate domain model. Compared with AjiL, LEMMA represents the system functions, domain data, and infrastructure contained in the model from different perspectives and presents corresponding views for different stakeholders. In addition, LEMMA adds the expressiveness of microservice visibility to indicate whether microservices can be accessed by other microservices and external environments, the expressiveness of system-level operations and the ability to reuse representations. However, none of these models are derived from the UML standard model and do not support the non-functional description of the system. SoaML is a modeling language for service-oriented architecture based on UML extension. It mainly uses service contracts to conduct fine-grained modeling of the relationship between system submodules, but it does not care about the splitting and decoupling of system modules.

In general, the current microservice model has the following problems:

- There is no modeling language for supporting the system architect to design those policies in microservice design, making it a great challenge for the developer to implement those policies directly into the code.

- In the absence of a generic microservice modeling language, the architect designs the architecture and often use the method of his own habits to describe the architecture. This phenomenon leads to the need to spend a certain amount of time on the author to learn to understand the description habits of others, which increases the design time and reduces the efficiency of development.
- They only describe the services that traffic passes through at the time of the request. They do not involve policies and cannot describe the management and control of complex traffic policies.

III. MSArch Model

A. *MSArch Overview*

This section introduces MSArch. MSArch is divided into a static architecture model and a dynamic interaction model. MSArch has four diagrams in total, as shown in Fig. 1.

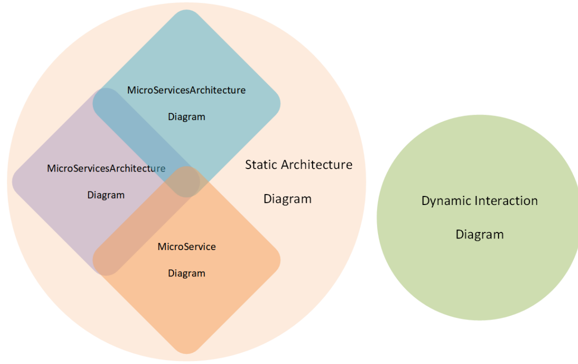

Fig. 1: Overview of MSArch

B. *MSArch Meta-Model*

1) Static Architecture Diagram Meta-Model: As a model for describing microservices, the static architecture model of MSArch is divided into three levels of view, namely MicroServicesArchitecture, MicroService, and MicroServiceContract. The metamodel is shown in Fig. 2. Red elements represent the original elements in SoaML, purple elements represent the original elements in UML, and yellow elements are extended or new elements. In this section, we introduce these three metamodels in turn.

- **MicroServicesArchitecture Meta-model:** MicroServicesArchitecture expands on the SoaML ServicesArchitecture element. MicroServicesArchitecture metamodel is shown in Fig. 3, it includes functional and non-functional microservices in the microservice architecture. Functional microservices are represented by Microservices and non-functional microservices are represented by Infrastructure in the metamodel. Infrastructure represents the essential services in microservices. It extends from the Participant in SoaML and describes the most basic common functionality in the microservice architecture. In the static architecture metamodel, we classify APIGateway, Security, and Discovery as non-functional services that are part of the infrastructure. MicroServicesArchitecture Diagram shows the microservice system as a whole. But it doesn't show all the information, like SoaML, just MicroService and the simple relational interfaces between them. This view helps users to have a rough understanding of the overall system architecture and quickly locate the services to be viewed in detail.

TABLE II: Graphical Notation of Static Architecture Diagram in MSArch

Diagram	Elements	Graphical Notation
MicroServices Architecture Diagram	MicroService	
	Provider Interface	
	Discovery	
	Security	
	APIGateway	
MicroService Diagram	ProvideInterface	
	RequestInterface	
	InterInterface	
	Operation	
	DomainModel	
	MSEntity	
	Relationships between MSEntity	
	Database	
MicroService Contract Diagram	Contract	
	Role	
	Relationships between Roles	

- **MicroService Meta-model:** MicroService is extended by the Participant in SoaML. It has the visibility property, which is used to indicate the observable scope of the service, INTERNAL or PUBLIC. As shown in Fig. 4, MicroService has three types of interfaces: RequestInterface, ServiceInterface, and InterInterface. RequestInterface and ServiceInterface represent interfaces required by MicroService and interfaces provided to others, extending from the Request and Service elements of SoaML. InterInterface is a self-used interface within MicroService that does not need to be exposed to other services. Each MicroService has its own domain model, which is broken down into individual services.
  The three types of interfaces have similar internal structures that operate on the entities MSEntity and attributes MSAttribute within DomainModel.
  In the MicroService Diagram, we can learn about the internal information of the service, including interfaces and entity objects.

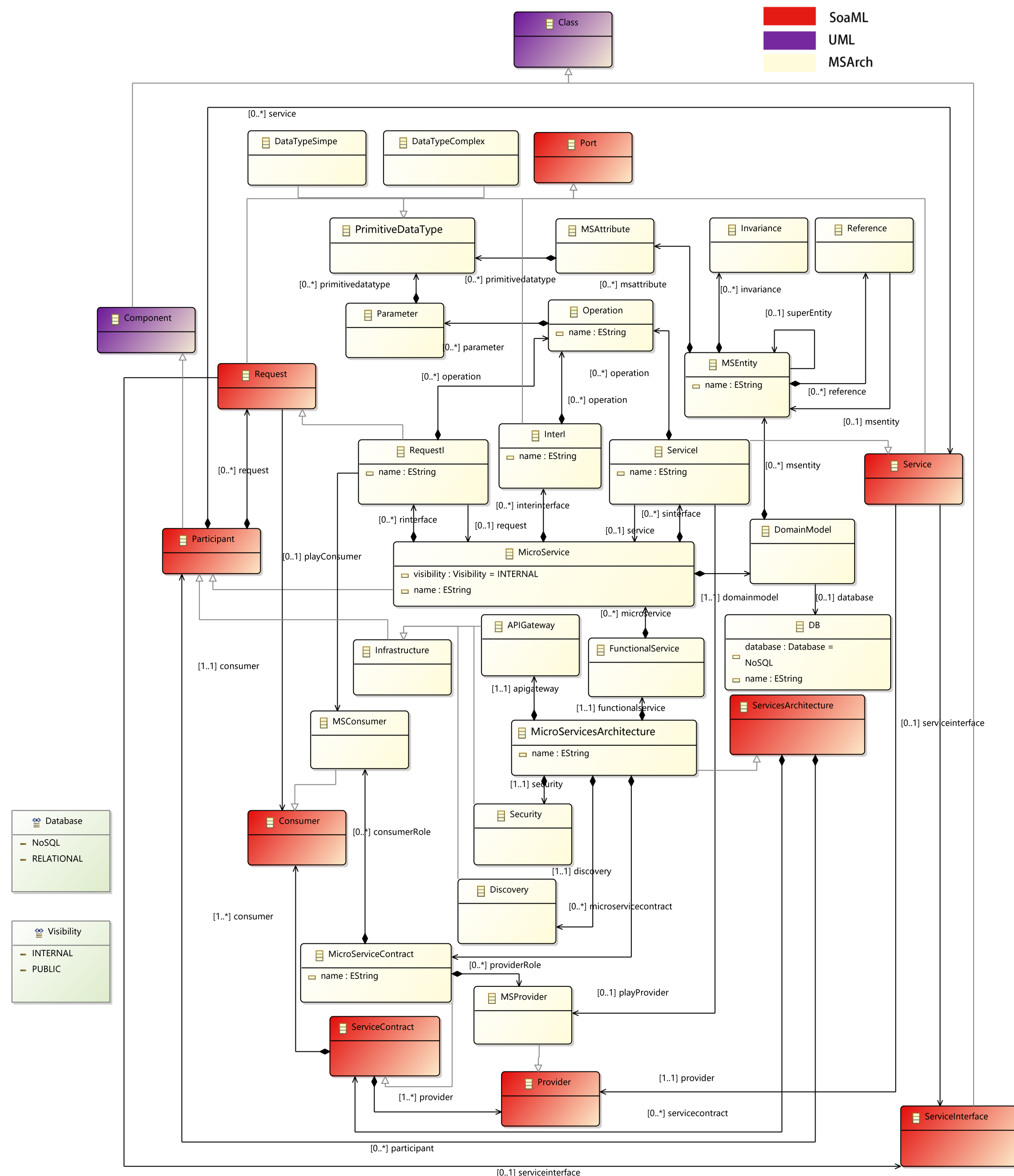

Fig. 2: Static Architecture Diagram Meta-Model

- **Contract Meta-model:** In SoaML, the service contract approach defines the roles each participant plays in the service (such as provider and consumer) and the interfaces they implement to play that role in that service. These interfaces are then the types of ports on the participant, which obligates the participant to be able to play that role in that service contract. The service contract represents an agreement between the parties on how the service is to be provided and consumed. This agreement includes the Interfaces, choreography, and any other terms and conditions. MicroServiceContract plays a similar role in MSArch. As shown in Fig.5, defines the role (interface service provider or consumer) that each MicroService plays in service communication. In the MicroServiceContract Diagram, you can see the roles of the two services as interface providers or consumers in this contract.

2) Dynamic Interaction Diagram Meta-Model: The meta-model of the dynamic interaction diagram is shown in Fig. 6. Purple elements are original UML elements, and yellow

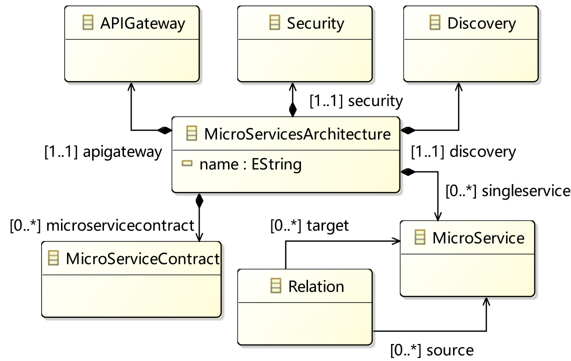

Fig. 3: Meta-model of MicroServicesArchitecture Diagram in MSArch

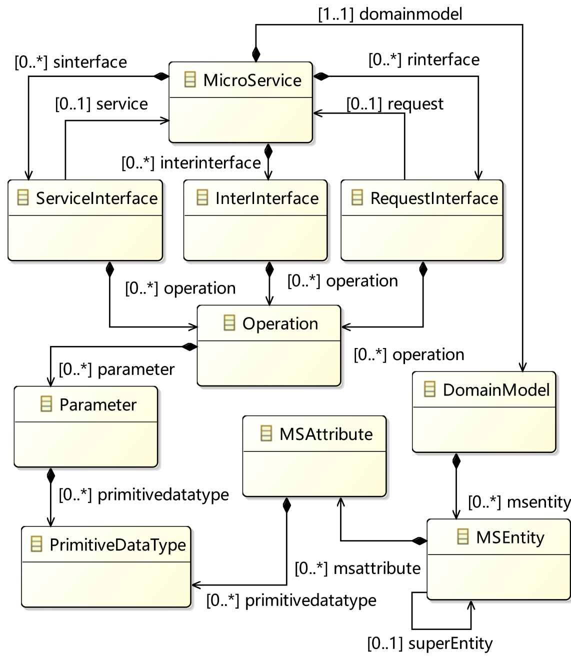

Fig. 4: Meta-model of MicroService Diagram in MSArch

elements are new elements or extensions of UML. Dynamic interaction diagrams, which are extended from UML activity diagrams and communication diagrams, describe dynamic information interactions between microservices. We summarize and extract the general traffic Settings from the existing microservice code framework, whose dynamic interaction diagram can express the traffic policies between related services when users or services make operation requests. In the diagram, users can set nodes to represent operations that control traffic. Common traffic operations such as retry, CircuitBreaker, TrafficForkNode, and TrafficDecisionNode are performed in the metamodel shown in Fig. 6. The RetryNode, Circuitbreaker, TrafficForknode, and TrafficDecisionNode are common traffic operations. To make it easier for the user to

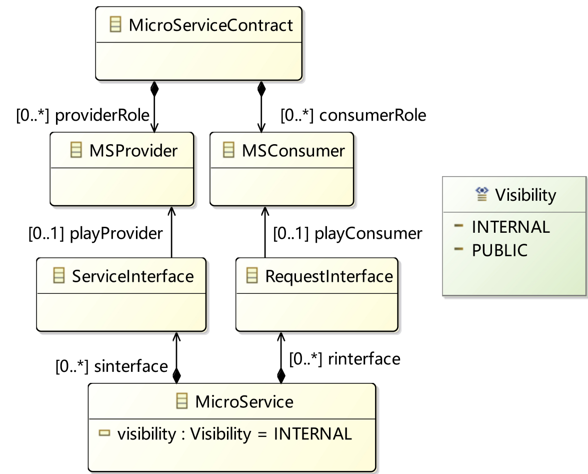

Fig. 5: Meta-model of MicroServiceContract Diagram in MSArch

draw the model, we set them up as separate nodes. These nodes extend from the ControlNode in UML. Some of the interaction policies between MicroServices are displayed on InteractionEdge. InteractionEdge contains controlledge and ObjectEdge. TrafficCondition is a condition displayed on InteractionEdge. It describes some other traffic-related policies between MicroServices, such as weights and timeouts.

C. MSArch Graphical Notation

To make the metamodel graphical, we design graphical notation for the basic elements in the metamodel, with each element corresponding to a symbol, as shown in Table II and III. The first column of the table contains the name of the element, and the second column displays the corresponding graph character.

TABLE III: Graphical Notation of Dynamic Interaction Diagram in MSArch

Diagram	Elements	Graphical Notation
MicroServices Dynamic Interaction Diagram	Actor	
	instance of services	
	TrafficForkNode	
	CircuitBreaker	
	TrafficDecisionNode	
	ControlEdge/ObjectEdge	

D. Example of MSArch

In this section, we will use examples to show how to use MSArch.

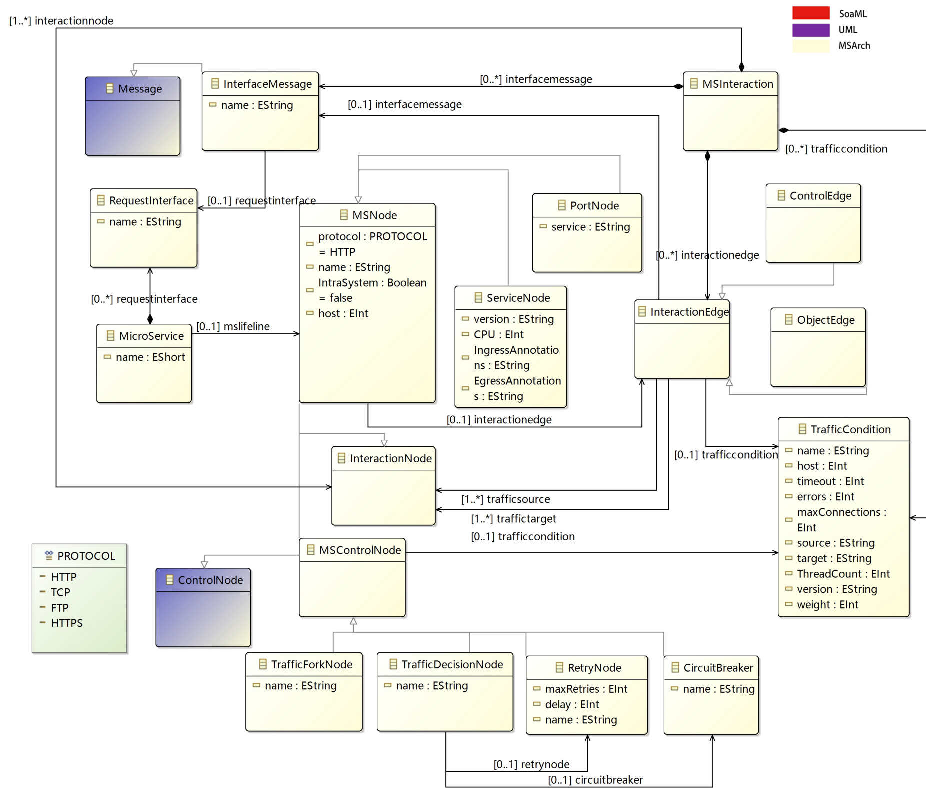

Fig. 6: Dynamic Interaction Diagram Meta-Model

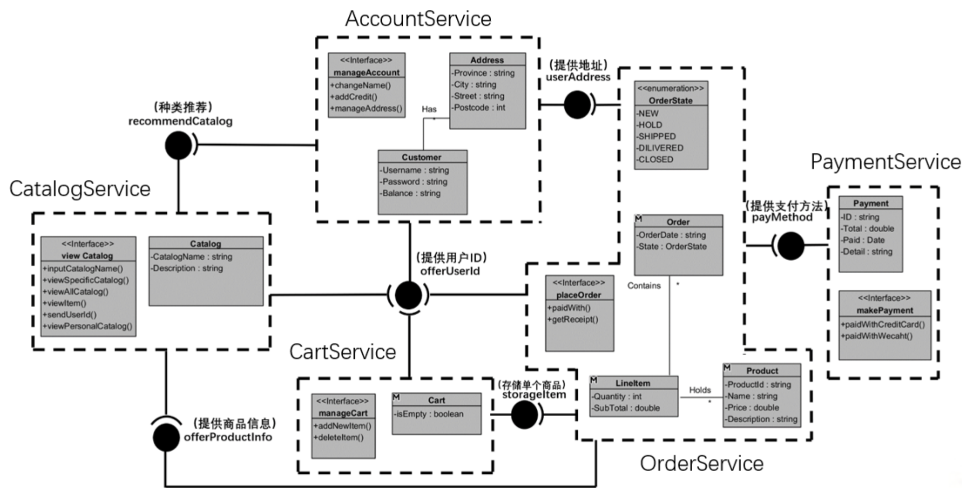

Fig. 7: Functional Division of the Online ordering system

1) Static Architecture Diagram: There is an Online ordering system, and this section attempts to represent it by using the MSArch's static architecture model. The microservice function division of the Online ordering system is shown in Fig. 7. Fig. 7 divides a complete system into five services, CartService, AccountService, PaymentService, OrderService, and CatalogService. The figure contains interface and entity information and information exchange between services such as AccountService, which provides user ID operations for CartService, OrderService, and CatalogService. MicroServicesArchitecture Diagram of the system is expressed in Fig. 9a in MSArch. The figure shows the online ordering system's overall architecture, specifically showing the provision and requirements among various services. The basic services in the microservices architecture, including Security, Discovery, and APIGateway, are shown on the right. In the left square are

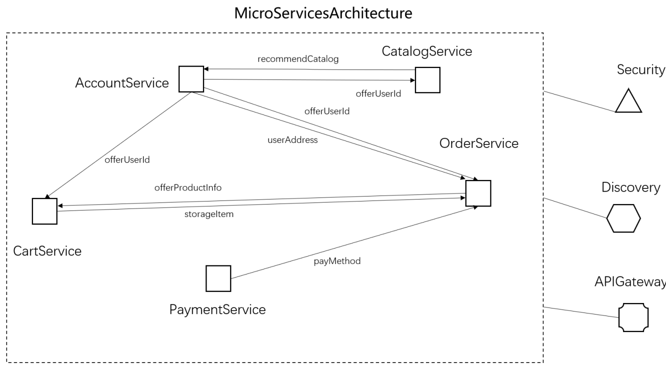

Fig. 8: MicroServicesArchitecture of the Online ordering system

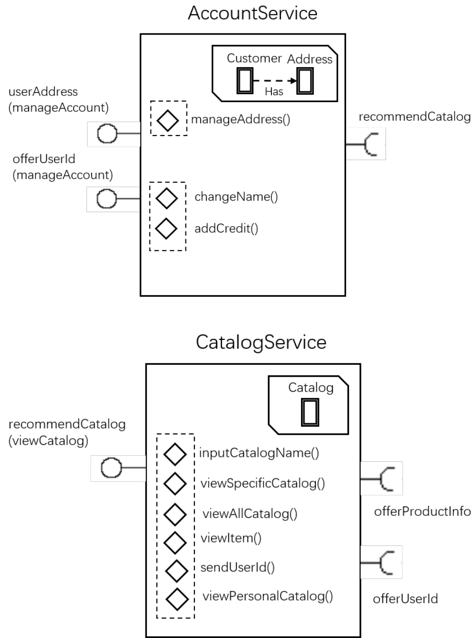

Fig. 9: MicroService Diagram of the Online ordering system

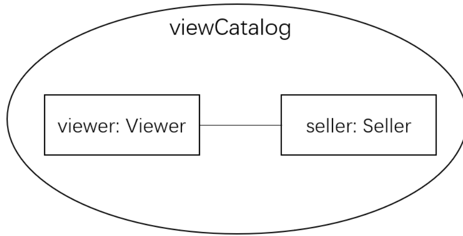

Fig. 10: MicroServiceContract of the Online ordering system

functional services in the microservices architecture, showing what services provide and require interfaces. The triangle end of the straight-arrow points to the service that needs the interface, and the other end is the service that provides the interface.

The MicroService Diagram for AccountService and CatalogService is shown in Fig. 9. The AccountService contains two entities, Customer and Address, where Customer has Address. The CatalogService contains only one Payment entity. CatalogService is the provider of the recommendCatalog interface, and AccountService is its consumer. The recommendCatalog interface contains six action functions. OfferProductInfo and offerUserId interfaces are also required for CatalogService. AccountService also provides userAddress and offerUserId interfaces. MicroServiceContract Diagram is shown in Fig. 10. This figure shows the information exchange between AccountService and CatalogService on the viewCatalog interface. In the MicroServiceContract, AccountService requires service and acts as the Consumer, while CatalogService is the Provider.

2) Dynamic Interaction Diagram: In this section, we use Bookinfo[2], the official example for Istio, to show the model. To better showcase our model language, we have added or modified some of Bookinfo's traffic policies. The Dynamic Interaction Diagram of the Bookinfo is shown in Fig. 11. In the figure, the black is the content of the model language, and the red is the annotation text. The strategy expressed is as follows:

- When service Productpage invokes getBookReviews() interface or method of service Reviews, traffic will be diverted to different versions of the service. The weight of traffic is 0 for v1, 80 for v2, and 20 for v3.
- Service Reviews with version v3 will retry the call request if the number of errors exceeds ten or the CPU usage exceeds 10% when calling getBookRatings() of Ratings with version v1.
- When service Reviews with version v3 call getBookRatings() of Ratings with version v1, the circuit breaker will be triggered when the request time exceeds the 20s.

IV. EVALUATION

This section evaluates the usefulness and complexity of the model and produces the results. A good practical model is generally of broad utility and low complexity for users to get started with.

A. Research Questions

For the MSArch model proposed in this paper, the following three research questions are proposed, and the evaluation results are expected to answer these questions.

RQ1: How well do static architecture models and dynamic models describe microservice in MSArch?

RQ2: Is MSArch easy to learn and use?

TABLE IV: Expressive Ability of MSArch

	TrainTicket	**Bookinfo**	**Average**
Static Architecture Model Coverage (%)	97	98	97.5
Dynamic Model Coverage (%)	98	100	99

[2]https://github.com/istio/istio/tree/master/samples/bookinfo

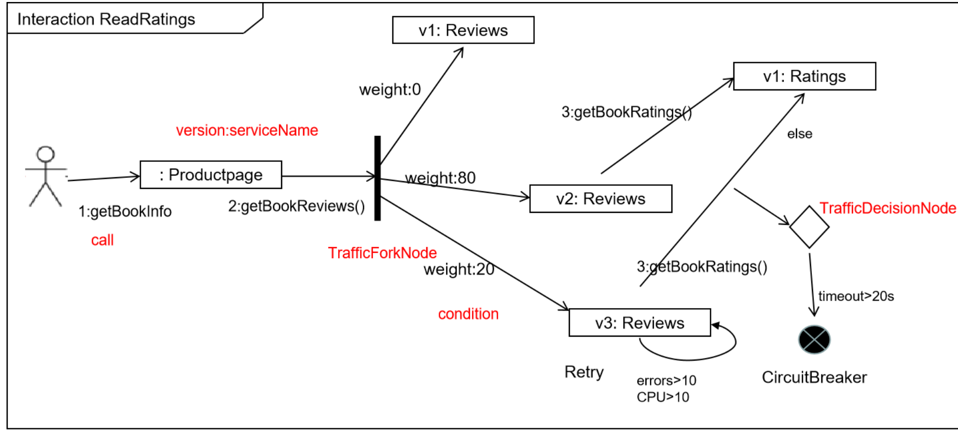

Fig. 11: Dynamic Interaction Diagram of the Bookinfo

TABLE V: MSArch Values for Independent Metrics

	Metric	MicroServicesArchitecture Diagram	MicroServiceContract Diagram	MicroService Diagram	Dynamic Interaction Diagram
1	$n(O_T)$	6	4	6	15
2	$n(R_T)$	2	2	3	17
3	$n(P_T)$	2	2	4	20
4	$\bar{P}_O(M_T)$	0.33	0.5	0.67	1.3
5	$\bar{P}_R(M_T)$	1	1	1.33	1.18
6	$\bar{R}_O(M_T)$	0.33	0.5	0.5	1.13
7	$\bar{C}(M_T)$	0.14	0.12	0.07	0.06
8	$C'(M_T)$	6.63	4.90	7.81	13.1
9	Total $n(O_T)$	31			
10	Total $n(R_T)$	24			
11	Total $n(P_T)$	28			
12	$\bar{C}(M)$	0.1			
13	$C'(M)$	8.33			

TABLE VI: Expressive Ability of MSArch

Name	**zy**	**zfl**	**ly**	**xpz**
Time to understand the model (min)	5	10	15	7
Accuracy rate of the questionnaire (%)	80	100	100	80
Accuracy rate of Static Architecture model (%)	97	98	98	99
Accuracy rate of dynamic model (%)	100	90	85	90

B. Case Studies

In this section, we will evaluate the model by drawing two microservice cases with MSArch. The TrainTicket[3] system was modeled to test MSArch's ability to model the static architecture of the microservice, and the Bookinfo system was modeled to test MSArch's ability to describe the dynamic interactions of the microservice. Two cases are detailed below.

1) TrainTicket System: This project is the research object of Xiang Zhou's team for fault analysis of microservice system [14]. The project is a train ticket booking system based on the microservice architecture, which contains 41 microservices.

[3]https://github.com/FudanSELab/train-ticket/

2) Bookinfo System: There is a Bookinfo system, and this section attempts to represent it by using the MSArch model. Bookinfo is a mock online bookstore application with a single page consisting of four sections: book listing, book details, reviews, and ratings, each implemented by a corresponding microservice. The Bookinfo system is the official example of Istio and has some recognition. To more fully demonstrate the capabilities of the MSArch model, we added some traffic rules to the system.

C. Evaluation Metrics

Kahraman et al. proposed that usability and expressiveness are important metrics for evaluating DSL[3]. Accordingly, we designed several evaluation indexes to evaluate MSArch.

1) Metrics of Expressiveness: This paper uses coverage to express the expressiveness of MSArch. We will use MSArch to model the three cases and calculate the proportion of the part that the model can express in the whole case to test the suitability of MSArch. Coverage is represented by C, N_{test} is the number of elements plus relationships in the model drawn by the volunteer, and N_{real} is the number of elements and

relationships described in the document or code.

$$C = \frac{N_{test}}{N_{real}}$$

2) Metrics of Learn and Use: Rossi et al. describe two sets of metrics that measure the complexity of the monograph technique and the full systems development approach. The proposed metrics provide a relatively quick and easy way to analyze the descriptive capability of a technology or method [11]. They defined 17 metrics to assess the complexity of a model. Technique Level Metrics are divided into three categories, including Independent Measures, Aggregate Metrics, and Method Level Metrics. The relative complexity of metamodel-based methods and techniques is important because it affects the learnability and ease of use of the methods. Next, we use this evaluation indicator to evaluate the complexity of MSArch, and the results are shown in Table IV and V. We invited four volunteers to do the experiment. First, we explain the meaning and usage of MSArch to the volunteers and record their study hours. They were then asked to complete a test questionnaire in which their accuracy was recorded. We then gave our volunteers a microservice document and asked them to model it in MSArch language, recording the accuracy of their final results.

3) Evaluation Results: With these two methods of evaluation, some conclusions can be drawn to answer the two research questions posed to the model.

RQ1: How well does MSArch's model describe microservices? The experimental results of the MSArch model expression ability are shown in Table IV. In MSArch, the average coverage rate of the static model is 97%, while that of the dynamic model is 98%.

RQ2: Is MSArch easy to learn and use? The identity background of volunteers is shown in github. Table V shows the values calculated according to the complexity calculation method proposed by the Rossi team. The experimental results of the learnability and usability of the MSArch model are shown in Table VI. The table includes the time required for volunteers to understand the model, the accuracy of answering the questionnaire, the accuracy of static modeling, and the accuracy of dynamic modeling. In the final data, the volunteers took an average of 10 minutes to understand the model and answered the questions correctly 90 percent of the time, 98 percent of the time on the static model, and 91 percent of the time on the dynamic model.

D. Discussion and Limitation

Combined with the above evaluation results, we found that MSArch has high Intuitive performance and ease of use, and people could easily use it based on their knowledge of UML and SoaML. In terms of its ability to describe microservices, people think highly of it. At the same time, compared with other models, MSArch has a lower average complexity. The MicroService Diagram is the most complex of the three MSArch diagrams. Because it needs to show the details inside the service, it has more relationship types and attribute types. UML is a general model, so it has many object types and attribute types. As MSArch is a model dedicated to describing microservices, its comprehensive complexity is higher than SoaML. However, its average complexity is lower than other models. MSArch has average low complexity and is easy for people to learn.

V. Conclusion and Future Work

In this paper, we introduce MSArch, a model language for describing microservices. Our evaluation demonstrates that MSArch is highly expressive and easy to learn. Establishing the structure is one of the primary tasks in developing a microservices project. With a well-defined structure, the development of microservice projects can be carried out smoothly throughout the process. Our future plan includes the development of a corresponding model drawing tool for MSArch. We intend to incorporate the ability to calculate the strengths and weaknesses of the microservice architecture into the tool, which would enable users to obtain the strengths and weaknesses of the microservice model after drawing it, providing a reference for architects to modify the microservice architecture.

References

[1] *About the Service Oriented Architecture Modeling Language Specification Version 1.0.1.* https://www.omg.org/spec/SoaML/1.0.1. 2012.

[2] *About the Unified Modeling Language Specification Version 2.5.1.* https://www.omg.org/spec/UML. 2017.

[3] Gökhan Kahraman and Semih Bilgen. "A framework for qualitative assessment of domain-specific languages". In: *Software & Systems Modeling* 14 (2015), pp. 1505–1526.

[4] Qingfeng Du Kanglin Yin. "On Representing Resilience Requirements of Microservice Architecture Systems". In: *International Journal of Software Engineering and Knowledge Engineering* 31 (2021), pp. 863–888.

[5] Bo Liu et al. "Log2MS: a framework for automated refactoring monolith into microservices using execution logs". In: *2022 IEEE International Conference on Web Services (ICWS)*. 2022, pp. 391–396.

[6] Manuel Mazzara et al. "Size matters: Microservices research and applications". In: *Microservices*. Springer, 2020, pp. 29–42.

[7] Nabor C. Mendonca et al. "Model-Based Analysis of Microservice Resiliency Patterns". In: *2020 IEEE International Conference on Software Architecture (ICSA)*. 2020, pp. 114–124. DOI: 10.1109/ICSA47634.2020.00019.

[8] Roland Petrasch. "Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication". In: *2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2017, pp. 1–4. DOI: 10.1109/JCSSE.2017.8025912.

[9] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. “Aspect-oriented modeling of technology heterogeneity in microservice architecture”. In: *2019 IEEE International conference on software architecture (ICSA)*. IEEE. 2019, pp. 21–30.

[10] Florian Rademacher et al. “Graphical and textual model-driven microservice development”. In: *Microservices*. Springer, 2020, pp. 147–179.

[11] Matti Rossi and Sjaak Brinkkemper. “Complexity metrics for systems development methods and techniques”. In: *Information Systems* 21.2 (1996), pp. 209–227.

[12] Jonas Sorgalla et al. “AjiL: enabling model-driven microservice development”. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. 2018, pp. 1–4.

[13] Simone Staffa et al. “Pangaea: Semi-automated Monolith Decomposition into Microservices”. In: *International Conference on Service-Oriented Computing*. Springer. 2021, pp. 830–838.

[14] Tao Xie Xiang Zhou Xin Peng. *TrainTicket*. https://github.com/FudanSELab/train-ticket/. Accessed April 4, 2010.

[15] O. Microservices Zimmermann. “Size matters: Microservices research and applications”. In: *Computer Science - Research and Development*. Springer, 2017, pp. 301–310.