ARCHI4MOM: Using Tracing Information to Extract the Architecture of Microservice-Based Systems from Message-Oriented Middleware

Snigdha Singh(✉), Dominik Werle, and Anne Koziolek

KASTEL – Institute of Information Security and Dependability, Karlsruhe Institute of Technology, Karlsruhe, Germany
{snigdha.singh,dominik.werle,koziolek}@kit.edu
https://mcse.kastel.kit.edu

Abstract. Microservice architectures that use Message-oriented Middleware (MOM) have recently seen considerable evolution regarding extensibility, re-usability and maintainability. Of particular interest are systems that are distributed and deployed with mixed-technologies. On the one hand, such MOM-based microservice systems improve flexibility through their messaging middleware. On the other hand, configuration for the above systems has to quickly adapt to required changes because of the continuous development process. Architecture reconstruction methods from dynamic data can keep architecture documentation and models in synchrony with the implemented architecture for such systems. However, the existing dynamic analysis methods for architecture reconstruction do not support the extraction for MOM-based microservice systems. The main challenge here is to understand and capture the asynchronous sender-receiver communication via the messaging middleware and to reconstruct the architecture model from it. In our work, we provide the ARCHI4MOM approach to automate the architecture extraction process. We instrument the sender-receiver and messaging services, collect run time data, analyse the trace data and construct the model from it. Architects can use the extracted architecture model for system refactoring and analysis of MOM-based systems. Thus, it reduces the cost and time required for manual architecture extraction process. We evaluate the accuracy of the approach by comparing the extracted model components to a manually crafted baseline model for a case study system.

Keywords: MOM-based system · Reverse Engineering · Architecture Extraction · Performance Prediction · Dynamic Analysis

1 Introduction and Motivation

Today's software systems are extensively using Message-oriented Middleware (MOM) to communicate in distributed microservice environments [9]. MOM

enables message-based communication between the components in order to achieve loosely-coupled and asynchronous communication. This provides benefits in terms of service maintainability and extendability, since microservices can be tested and deployed separately. MOM provides an intermediate layer between the message sender and receiver to decouple them from each other. In order to meet various requirements, there exists technologies like RabbitMQ[1], Kafka[2] to enable MOM-based communication. Such technologies use MQTT[3] or AMQP[4] protocols for message exchange between the messaging middleware.

During the continuous software development process, the architecture of MOM changes when the system changes. When a new requirement demands the addition of new components, the system's developers have to understand how it will change the architecture of the system, for example, by requiring changes in the configuration of the MOM or the introduction of additional communication channels. This continuous development, evolution, and software maintenance creates the problem of architecture erosion in the existing software when the knowledge of the system's architecture is not adapted after changes to the system [14].

The loss of architecture knowledge makes it difficult for the system architect of software systems to understand and refactor the system. At the same time, the refactoring process is time-consuming and costly. One solution to this general problem of erosion is the approach of architecture reconstruction through reverse engineering (RE) of an existing software system. Despite the importance of architecture knowledge recovery for MOM-based systems, only few studies exist on RE for architecture recovery in MOM-based microservice systems [1,13].

To build the component-based architecture model, we have to identify the components and their communication behaviour. Static analysis approaches for RE make it possible to extract the models from the existing source code [11]. Such techniques sometimes fail to provide all the required information about the components. Hence, it is good to collect data during run time to gather the information not covered via static analysis.

In case of synchronous HTTP communication, there is a direct communication (i.e., messages are sent directly to a specific receiver) where all the microservices have to be available during their message exchange. In such communication, the sender microservice has information about the receiver microservice because HTTP follows the send and acknowledgement principle. In case of MOM-based microservice system, microservices (components) communicate via messaging middleware asynchronously. Unlike HTTP communication, sender and receiver components do not know each other. Often, this information is hard to collect from the available architecture documents.

The first problem is to identify the sender and the receiver components. The second challenge is to understand and capture these components and model the

[1] HTTPs://www.rabbitmq.com/.

[2] HTTPs://www.confluent.io/Kafka-summit-lon19/Kafka-vs-integration-middleware/.

[3] https://mqtt.org/.

[4] https://www.amqp.org/.

system's architecture. However, the state-of-the art approaches do not support the architecture reconstruction of the asynchronous MOM-based communication which is typically found in modern microservice systems [6,17].

To handle the above problem, we propose the method ARCHI4MOM, which supports the architecture extraction of modern MOM-based microservice systems. With our work, we present an approach for the automatic extraction of architectural models from MOM-based microservice applications based on tracing information. We introduce an implementation of these concepts on the basis of a flexible and extensible architecture. An overview of the steps of the approach is shown in Fig. 1.

We leverage existing frameworks and library standards for OpenTracing, which is a method for the distributed tracing of data preparation and processing [5]. For our work, we re-use parts of the existing builder pattern architecture of Performance Model Extractor (PMX) [8,17] allowing the extraction of architecture models that are instances of the Palladio Component Model (PCM) [2]. To evaluate the approach, we use a community example system, flowing retail[5], which implements a system where microservices communicate with each other via MOM. We analyse the extracted model elements and compare them with a manually created model to evaluate the accuracy of our approach.

The rest of the paper is organized as follows: Sect. 2, gives the foundations, Sect. 3, describes the complete architecture design of ARCHI4MOM, Sect. 4, presents the technical details of the architecture recovery approach along with the implementation methods. In Sect. 5, we evaluate our approach with the case study scenario. Section 6, classifies our work with respect to the state-of-the art literature. Section 7, provides the concluding remark with possible future ideas.

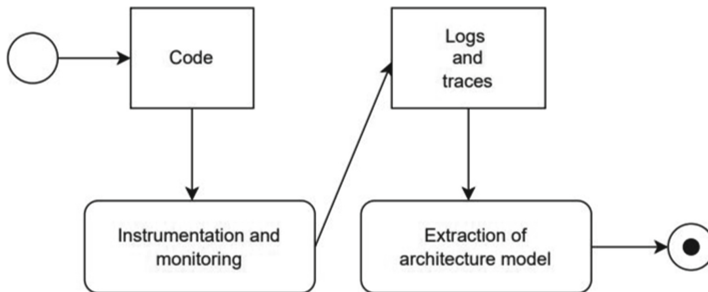

Fig. 1. Overview of the Approach.

2 Foundation

First, we introduce different concepts and techniques our approach uses and highlight the specific challenges of MOM-based systems for architecture reconstruction.

[5] HTTPs://github.com/berndruecker/flowing-retail.

2.1 PMX

In the proposed ARCHI4MOM approach, we extract architecture models for MOM-based microservice applications from tracing data by extending the PMX approach. PMX is used to collect and analyse [17] tracing data and extract a PCM architecture model out of it.

We discuss existing PMX challenges and our motivation to extend it further [15]. First, the current implementation of PMX only considers synchronous communication between microservices. It does not support the architecture extraction of MOM-based microservices, where an asynchronous sender-receiver communication happens. Second, PMX depends on the monitoring tool Kieker for tracing data collection. Kieker does not comply with current standards in the field of distributed tracing, and therefore PMX cannot analyse data collected from modern MOM-based systems. Third, PMX extracts PCM models that do not contain concepts for modelling messaging communication via middleware. To meet the above limitations, we extend parts of PMX as shown in Fig. 2 and make it compatible for MOM-based systems. We discuss the extension in details in Sect. 3.1.

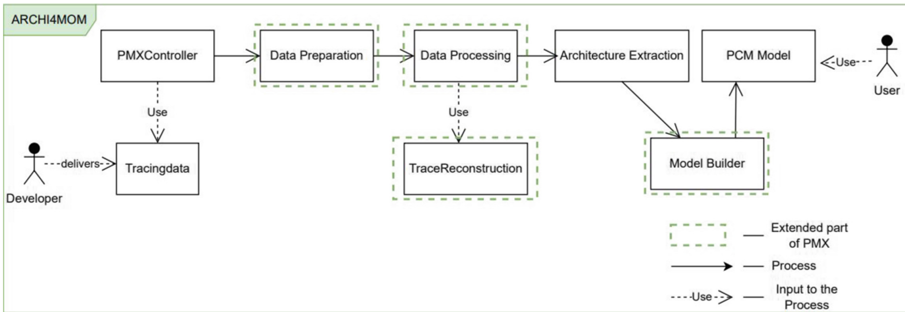

Fig. 2. Extension of PMX for ARCHI4MOM.

2.2 Palladio Component Model (PCM)

PCM is an architectural model that supports to explore the quality of the software systems at design and run time. PCM allows modelling components of the system, and these models of components are reusable. It consists of different views of the system, the *Repository model*, *System model*, *Resource Environment model*, *Allocation model* and *Usage model*. In our architecture extraction approach, we extract the *Repository model* and the *System model* from the trace information. The *Repository model*, contains data types, components, and interfaces. Each component provides at least one *ProvidedInterface* and an arbitrary number of *RequiredInterfaces*. The *ProvidedInterfaces* define the services provided by the component, the *RequiredInterfaces* define the services required to operate. This happens with *ProvidedRole* and *RequiredRole*.

PMX uses PCM modelling language for generation of architecture model. These PCM model elements do not model asynchronous message-based communication. Therefore, the existing PMX fails to support the architecture extraction of MOM-based microservice systems. We integrate new PCM modelling elements into ARCHI4MOM to enable the extraction for the MOM-based systems.

2.3 Message-Oriented Middleware (MOM)

To understand model extraction, it is required to recognize different possible communication types in MOM-based systems and represent it in the model. We focus the message exchange mechanism via "topics" which facilitates publish-subscribe method. The messages are published to the "topics" and then the subscribers receive all the messages published to the topics they subscribed to. Figure4 shows the example application that shows how the "topics" connect to a MOM middleware. The sender microservice (component A) sends messages to Topic T and the topic forwards the message to the receiver microservice (component B) subscribed to the MOM. The existing architecture extraction methods including PMX do not consider the extraction of sender-receiver communication via the messaging middleware as explained here.

2.4 Flowing Retail Case Study System

We use Flowing-Retail (FL) as our case-study system, which simulates a production process where goods are retrieved, fetched and shipped after being ordered and paid by the customer. In this application, the microservices communicate through an asynchronous messaging method. For our purpose, we use the variant of the system that is based on Kafka messaging middleware with the Spring messaging framework. The system consists of six microservices: *Checkout, Order, Payment, Inventory, Shipping, Monitor* and one "topic" *flowing-retail* to communicate between these microservices. The internal communications between the microservices, take place via message channel. For external communication, with other microservices and third party libraries, FL uses an external Kafka messaging broker to store and forward external messages to all subscribed services. We only focus on the communication between microservices using the Kafka message broker and ignore the internal asynchronous communication at the moment.

3 ARCHI4MOM Structure

ARCHI4MOM is mainly guided by the following research question: How to generate the architecture model for MOM-based microservice systems from traced information, which is collected dynamically by instrumenting the source code of a software system? ARCHI4MOM provides a generic framework for describing asynchronous communication and implementations for specific frameworks. These frameworks are implemented by concrete classes who handle the details of the chosen standard or language. We show the parts we extend in existing

PMX in order to support the architecture extraction in ARCHI4MOM. This overcomes the challenges of existing PMX described earlier. In the following, we describe the different parts of the approach shown in Fig. 2.


```
{
    "data": [
        {
            "traceID": "532de50458050082",
            "spans": [
                ...,
                {
                    "traceID": "532de50458050082",
                    "spanID": "015891c441167ca9",
                    "flags": 1,
                    "operationName": "receive:input",
                    "references": [
                        {
                            "refType": "FOLLOWS_FROM",
                            "traceID": "532de50458050082",
                            "spanID": "c00c953c439352cf"
                        }
```

```
    "data": [
        {
            "traceID": "f34f282ae9b7a6df",
            "spans": [
                ...,
                {
                    "traceID": "f34f282ae9b7a6df",
                    "spanID": "d47b4a01e709635a",
                    "flags": 1,
                    "operationName": "findOwner",
                    "references": [
                        {
                            "refType": "CHILD_OF",
                            "traceID": "f34f282ae9b7a6df",
                            "spanID": "f34f282ae9b7a6df"
                        }
```


Fig. 3. Comparison of Traces Structure between Asynchronous ARCHI4MOM old Synchronous PMX.

3.1 PMXController

PMXController is the first entry point for the extraction process, which enables the extraction process to start independently and allows the easy integration of the process to the continuous development pipeline.

3.2 Data Preparation

Before starting the extraction, a preparation of its input is necessary. In this step, we instrument the source code with the Jaeger[6] tracing tool and collect tracing data. We use the OpenTracing API which supports Kafka-based messaging systems. The use of the OpenTracing standards solve the first limitation of existing PMX and extend the usability to support modern MOM-based systems. This dependency enables the auto-instrumentation of systems with Spring and Kafka. We add the dependency to all microservices to generate the trace in each service. The trace data consists of several spans composed of tags, logs and other information. The asynchronous trace data introduces a set of new information for MOM-based asynchronous microservices from OpenTracing which is not present earlier. We transform the *Trace*, *Span* collected from OpenTracing, to the internal trace structure of ARCHI4MOM, which are called *ExecutionTrace*, *MessagingExecution*. Like wise, other important mappings are represented in Table 1. We map the information to recognizes the messaging spans from normal spans. We later use it in the *traceReconstructionService*. The next step after instrumentation is the collection of tracing data. We collect them in the form of JavaScript Object Notation (JSON) files. This becomes the input to Data Processing phase.

[6] https://www.jaegertracing.io/.

Table 1. Mapping of new OpenTracing data to ARCHI4MOM structure.

OpenTracing	**ARCHI4MOM**
Span	MessagingExecution
Trace	ExecutionTrace
Operation	MessagingOperation
TraceID+SessionID	TraceInformation
AsynchronousCall	AsynchronousCallMessage
AsynchronousCallReply	AsynchronousReplyMessage

3.3 Data Processing

In the Data Processing phase, we analyse the structure of the traces. In synchronous methods, the sender and receiver are present in one span, which makes it easier to track the behaviour. In asynchronous communication, the information is not present in the same span because they communicate through the middleware. Therefore, the execution of methods in different components needs to match based on the tagged information. We require finding this information from the trace span and collect them for data analysis. This is the novelty in our approach.

With the span reference relationships, we extract the communication pattern. We determine the nature of the current span for synchronous or asynchronous communication. For our work, we focus on the tags and logs since this provides relevant information about the communication. For example, in case of synchronous communication it is CHILD-OF and in case of asynchronous communication it is FOLLOWS-FROM. Figure 3 shows the difference in the communication traces collected from old PMX with synchronous Spring-based application in comparison with the asynchronous MOM-based application in case of FL. We notice that the structure of the trace widely varies in both types of communication. This result is because of changing the communication type from synchronous to asynchronous.

After the trace reconstruction, we find all the spans of send operation do not have information about the *topics*, they send to or receive from. Often, this information is missing in collected spans. From the traces, we manually search for the topic name. Then, we iterate over all the sending spans with a *FOLLOWS-FROM* or *message-bus* relation tag and add the *topic* name to all the receiving spans from a given sending span, that have no topic set in their tags or logs. Thereby, we identify the message type that has been sent to the receiver. For example, the *message-bus* tag is used for identification of topic names in case of Spring-Kafka messaging middleware. Asynchronous MOM-based applications have more operation-related data which we identify in this step and integrate in architecture extraction step, for example identification of the *topic name*, *components* and corresponding *interfaces*. In ARCHI4MOM, all these new information about asynchronous communication is integrated into PMX, which handles the existing issues. We use this information later to generate *DataInterfaces*.

3.4 Architecture Extraction

In order to extract the complete PCM architecture model that supports the messaging behaviour of MOM-based systems, we require new model elements. For this reason, we combine recent additions to the PCM which support asynchronous communication. It introduces[7] additional model elements for representing asynchronous communication and also provides a simulation for this type of communication. We integrate new model elements to existing PMX and enable the PCM to represent messaging middleware and sender-receiver relation. These new components are:

- *DataChannel* responsible for data transfer providing a *DataSinkRole* and requiring a *DataSourceRole*.
- *DataInterface* determines the type of data transfer and has exactly one signature.
- *DataSinkRole* is a *ProvidedRole* and describes which data is received by the *DataChannel*.
- *DataSourceRole* is a *RequiredRole* and describes which data is send to the *DataChannel*.

3.5 Model Builder

In the Model Builder phase, we use the extracted data to create a model instance. There is no logic available in PMX to construct the model for asynchronous communication. We introduce the logic to generate the new PCM model elements to support model building for messaging communication.

We illustrate the extended implementation logic, we adapt to build the PCM model, with an example. Assuming microservice C communicates with a *DataChannel* D through two different *DataSinkRole*, we require two *DataInterface*s DI1, DI2 and two *DataSinkRole*s R1 and R2 with the respective *DataInterface*s D1-R1 and DI2-R2 to architect the communication. With two different roles, C can process the messages received from D differently, depending on *DataInterface*. We transform the knowledge into model generation.

FL case study component *Monitor* microservice always receives messages. So it has several *DataSinkRole*s depending on the *DataInterface*s. This relation is represented in the architecture construction logic. For a sending operation, the corresponding component has a *DataSourceRole* and for a receiving operation, the component has a *DataSinkRole*. The number of sink roles of *DataChannel* depends on the number of different types of messages it receives, and hence on the number of *DataInterface*s. Sender microservice components send the messages via *DataSourceRole* to the *DataChannel* and receiver microservice consumes the message via *DataSinkRole* from the *DataChannel*.

[7] HTTPs://github.com/PalladioSimulator/Palladio-Addons-Indirections/tree/master/bundles/org.palladiosimulator.indirections/model.

As a result, in the case of the *DataChannel*, we have two possibilities to represent the sending roles. Both are illustrated in Fig. 5. In the first case, we see a single source role for each *DataInterfaces* in the *DataChannel*, where the message is sent several times from the *DataChannel* and received by all the components in the *DataChannel*. In the second, we see several source roles for a single *DataInterface* in a *DataChannel*. In our work we choose the second alternative, because in FL case study every message is sent simultaneously to at least two other microservice components, which is easily captured by the second possibility.

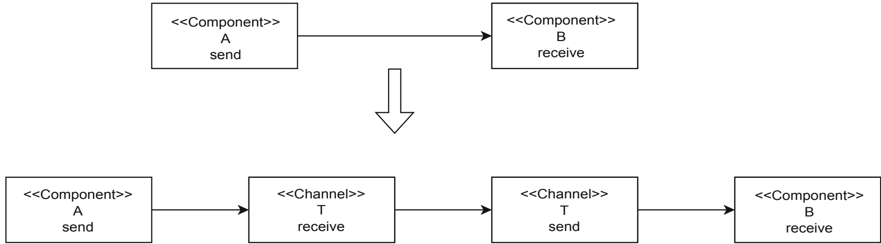

Fig. 4. Message Communication with Topic T.

4 Implementation of ARCHI4MOM

In this section, we discuss the implementation of our approach with reference to the FL case study. Our description is structured according to Sect. 3. We adjust the OpenTracing structure, by adding logic for new tags and logs pairs, discussed in Sect. 3.3. ARCHI4MOM adds new logs which not only supports and recognizes messaging spans but also adjusts them before and after the trace reconstruction.

In the case of FL system, all the microservices are subscribed to the "topic" *flowing-retail*. When a microservice sends a message, all other microservice components except *Checkout* receive it. However, not all microservices process it further. All microservices communicate through Kafka messaging, which makes it difficult to figure out the behaviour of the communication. The communication between the sender and receive is hard to capture since they do not talk directly.

We face the problem in identifying the data type and hence the *DataInterface* to transform it into model elements from the trace information. In order to tackle this, we consider the microservice that processes a given message further is a *DataInterface* and put it in the architecture model. Thereby, we extract 6 *DataInterfaces* in FL case study. Considering the messaging communication from sender to Kafka middleware and Kafka middleware to receiver, we model the Kafka middleware as a*DataChannel*.

When a component sends a message to the other component, the message goes through a messaging middleware. It looks like two sending operations: first,

sending from the component to the messaging middleware and then from the message broker to the receiving component. For example, we have an operation O, where a component A sends a message M to a component B through topic T, we will then have two operations O1 and O2. O1 is then a sending operation from A to T and O2 from T to B. But, in our observation, each message sending operation is represented by two spans, a sending span and a receiving span. In each sending span, we extract three spans, the first span is the sending of a message from a component to a message broker, the second is the receiving of that message by the messaging middleware and the third is the sending of the message from the messaging middleware to the receiving component. In order to avoid consistency problems, we assign the new tags *FOLLOW-FROM* discussed earlier, and the third span gets the identifier of the original span. The receiving spans refers to the third sending span and connects the sending operation. This process is illustrated in Fig. 4.

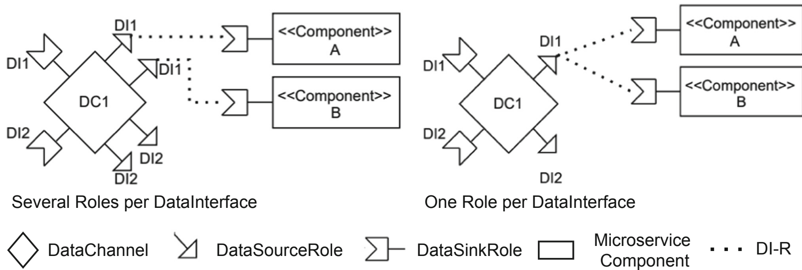

Fig. 5. Roles per DataInterfaces.

We process the spans and transform them in the ARCHI4MOM internal structure. For example, the microservice *Order-Camunda* from the case study FL sends a message to another microservice. The first span is the sending operation from the *Order-Camunda* service followed by the receiving operation from the "topic" *flowing-retail*. We use this analysis to extract the control flow of the architecture model extraction.

The next step is the creation of an architecture model. In our case, *DataChannel*s and *DataInterface*s are created as a part of PCM repository model to represent the MOM-based communication. In PCM, every component obtains a corresponding interface to communicate with each other. As already discussed, we model source and sink roles, which are characterized by a *DataInterface* to represent the data type. We can realize the above implementation in the extracted architecture of FL case study described in evaluation section.

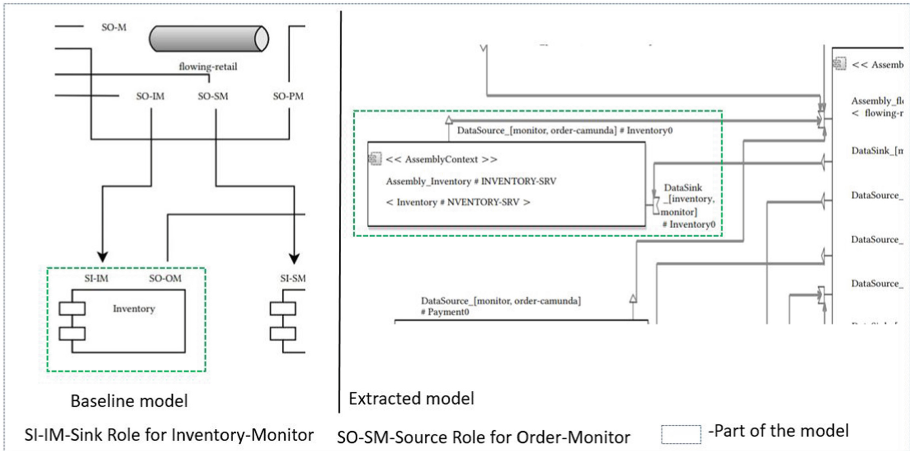

Fig. 6. Excerpts of Extracted and Manual Model.

5 Evaluation

In order to evaluate the extracted model, we compare it to a baseline (manual) model. We use the available reference architecture description of FL[8] and create the PCM baseline model from it. In order to ensure the ground truth, we validated the manual model by 3 developers. The manual PCM model considers new model elements described in Sect. 3.4, in order to support modelling of asynchronous message-based communication. The manual model contains 6 microservice components, 1 topic, 17 *DataSourceRoles*, 17 *DataSinkRoles* model elements [4].

We collect the traces after 20 iterations. We collect the traces for Order creation, Payment and successful Delivery. The longest trace we collect have 116 traces with all the services. We use Jaeger UI in order to view and collect the traces. We ensure to include all 6 services of FL in our evaluation. We search and locate the model elements in the extracted architecture model. The actual extracted model contains 18 *DataSourceRoles*, 14 *DataSinkRoles*, 6 *DataInterfaces* and 1 *DataChannel*. We observe, ARCHI4MOM identifies 1 more *DataSource* which we cannot identify from the architecture description. Also, there are 3 less *DataSinkRoles* compared to manual model. This is because, in the System model, 4 *DataSourceRoles* use only 1 *DataSinkRole*. We can observe the difference between the extracted model elements and manual model elemnts in the last row of the Table 3. The first number 18 represents the extracted model element and second number 17 represents the corresponding manual model elements in 18/17 notation. We share all the relevant diagrams and source code[9].

[8] https://github.com/berndruecker/flowing-retail/tree/master/kafka/java.
[9] https://doi.org/10.5281/zenodo.6778977.

Since the full model is too large to discuss in this article, we explain the main ideas of the extraction using an excerpt of the model, as shown in Fig. 6. The excerpt shows the *Inventory* component and the components it communicates with. We show the baseline model on the left with the *DataSinkRole* to Inventory-Monitor component (SI-IM) and *DataSourceRole* to Order-Monitor (SO-OM) component. We use SI for SinkRole and SO for SourceRole here. As we see on the right of Fig. 6, ARCHI4MOM extracts the *DataSinkRole* and *DataSourceRole* correctly. In addition, ARCHI4MOM extracts "topic" as *flowing-retail* and Kafka messaging as *DataChannel*.

Using only the source code of FL, it is not easy to identify the above-mentioned model elements, but ARCHI4MOM automatically extracts 38 model elements from the tracing data.

To verify the achievement of this objective, we structure the evaluation using a Goal Question Metric (GQM) plan as described by [16], which is presented in Table 2. For all model elements that are relevant for the asynchronous communication, we observe the model elements created by the extraction approach and in the baseline by the expert. Both sets are compared using Precision, Recall and F1 score. Overall, the results of the evaluation show that the extraction of MOM-based microservices based on dynamic tracing data is possible for systems communicating asynchronously to achieve (100%)*Precision*, (95.65%)*Recall* and (97.8%)*F1-score* and is shown in Table 3.

5.1 Threats to Validity

In this section, we address threats to validity for case-study-based research in software engineering.

Internal Validity. Addresses whether all implementation possibilities of asynchronous communication have been considered. In our case study, we analysed publish-subscribe based communication with Kafka and extracted the architecture for the same. We evaluated the results with the baseline model. One factor that is hard to eliminate is the expertise of the person modelling the case study architecture. We consider this factor by creating a baseline that is as accurate as possible to avoid unfairness in our evaluation approach.

External Validity. Addresses whether the findings of the case study can be generalized to other cases of interest. We can not say, at this point, if the approach can be successfully applied to the industrial set up with more than 100 microservices. We aim to increase the external validity by focusing on a case description that comes from the research community. Furthermore, we consider the case study system which is used by most researcher and uses popular middleware for MOM like Kafka and RabbitMQ.

Table 2. The GQM-Plan for Evaluation.

Goal	Purpose	Achieve
	Issue	Complete extraction
	Object	Architecture extraction of MOM-based microservices
	Viewpoint	Software architects
Questions	Q	Are all *DataSourceRole*, *DataSinkRole*, *DataInterfaces* are extracted?
Metrics	M1	Precision
	M2	Recall
	M3	F1-score

6 Related Work

In our observation, we categorize the state-of-the-art literature for architecture extraction of MOM-based systems into three main categories. First, based on the type of input used by several approaches. If the input used by the approaches are the artefacts, documents, and source code of the system, we categorize it as static analysis for architecture extraction. Otherwise, if the approach use inputs such as logs, spans, traces collected dynamically from the system for architecture extraction, we categorize it as dynamic analysis. There exists some approaches which combine both the approaches for more accurate architecture extraction, and we categorize it as hybrid approach. Second classification is based on what kind of microservice systems are taken into consideration for the architecture extraction. For example, whether the microservice systems communicating synchronously with each other or they communicate asynchronously via messaging middleware. Third, whether the outcome of the approaches focuses on architecture extraction, behavioural extraction or performance model extraction. Based on the discussed categorization of the state-of-the-art literature, we place our work in the category of dynamic analysis for architecture extraction for the microservice systems which communicate explicitly via messaging middleware. Therefore, we narrow down our discussion focusing to the related work relevant to our work.

Table 3. Extracted model elements for Flowing-retail.

	Microservices	DataSourceRole	DataSinkRole	DataInterface
	Checkout	1	1	1
	Order-Camunda	8	5	1
	Payment	2	1	1
	Inventory	1	1	1
	Shipping	1	1	1
	Monitor	5	5	1
Total	6\6	18\17	14\17	6\6

Granchelli et al. [7] present an approach (MicroART) which takes system's service descriptor as input for static analysis and container communication logs for dynamic analysis to generates the model for messaging systems. The main limitation of MicroART approach is manual refinement of the generated model. It needs a software architect to manually resolve the sender-to-message broker and message broker-to-receiver interactions into sender-receiver interactions before the final architecture is generated, which makes the recovery process slow and prone to error. In our approach, we automatically extract the relation for message-based systems and transform it into an architecture model.

Alshuqayran et al. [1] propose the MiSAR approach for architecture recovery of microservices systems with hybrid approach. This approach provides manual mapping rules to identify the microservices as an infrastructure component and hence not as a component for modelling the basic messaging behaviour. The approach lacks to capture the asynchronous dependencies between sender-receiver communication via messaging middleware, which is the main focus in our approach.

Kleehaus et al. present Microlyze [10], which analyses the system statically and dynamically to extract the architecture semiautomatically. The Microlyze discovers the microservices using the service discovery Eureka, and then it finds the communication between the microservices using distributed tracing technology Zipkin. However, the discovery process ignores the detection of microservices and the communication among each other and with the messaging middleware. Therefore, the architecture is not suitable for MOM-based microservice systems.

Brosig et al. [3] propose a method to automatically extract the architecture and performance models of distributed microservices. Their approach uses run-time monitoring data in order to extract system's architecture and performance model parameters. Their work is based only on Enterprise Java Beans, Servlets, Java Server Pages, therefore fails to support microservice communication via messaging middleware.

Mayer and Weinreich [12] aim to recover the architecture of REST-based microservice systems. The approach combines a hybrid approach to automatically extract relevant information from the system and recover the architecture from the information. The metamodel of this approach only supports REST-based systems, but not asynchronous MOM-based microservice systems.

7 Conclusion and Future Work

In our work, we capture the asynchronous communication related information between MOM components and other components in MOM-based microservice systems and transfer them into an architecture model. We mainly focus on the sender-receiver message exchange via state-of-the art messaging middleware. ARCHI4MOM approach introduces an automated and flexible architecture extraction method to support modern mixed-technology systems. In order to precisely fit the extracted architecture models to MOM-based systems, we build upon the data model of OpenTracing standards and libraries. In addition,

our data preparation phase provides an extension point to import data from different tracing standards other than OpenTracing. This adds necessary flexibility for the model preparation and generation phase.

In the future work, we plan to test our approach with other middleware systems communicating with more topics. Currently, we extract the repository and part of system model and hence plan to extract the usage model for complete performance model extraction. We evaluated our approach with an academic-oriented case study, but in reality there could exist systems which use the asynchronous as well as synchronous communication between its components. Considering this, we think that it is important in the future to merge our implementation with the extraction's approach for synchronous communication to be able to model such mixed-technology microservice systems. Although we have successfully applied our approach to MOM-based microservice application and extracted the architecture, we want to further extend our approach with the above-mentioned variations to make it more general and useful for the user.

Acknowledgement. This work was supported by the German Research Foundation (DFG) Research Training Group GRK 2153: Energy Status Data - Informatics Methods for its Collection, Analysis and Exploitation and by KASTEL Security Research Labs. We thank Fatma Chebbi for implementing and evaluating the approach as a part of her Bachelor's thesis [4].

References

1. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: 2016 IEEE 9th International Conference on Service-oriented Computing and Applications (SOCA), pp. 44–51. IEEE (2016)
2. Becker, S., Koziolek, H., Reussner, R.: The palladio component model for model-driven performance prediction. J. Syst. Softw. **82**(1), 3–22 (2009)
3. Brosig, F., Huber, N., Kounev, S.: Automated extraction of architecture-level performance models of distributed component-based systems. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 183–192. IEEE (2011)
4. Chebbi, F.: Architecture extraction for message-based systems from dynamic analysis. Bachelor's thesis, Department of Informatics, Karlsruhe Institute of Technology (KIT) (2021)
5. Cinque, M., Della Corte, R., Pecchia, A.: Advancing monitoring in microservices systems. In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 122–123. IEEE (2019)
6. Di Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE (2017)
7. Granchelli, G., Cardarelli, M., Di Francesco, P., Malavolta, I., Iovino, L., Di Salle, A.: Microart: a software architecture recovery tool for maintaining microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 298–302. IEEE (2017)
8. Heinrich, R.: Architectural runtime models for integrating runtime observations and component-based models. J. Syst. Softw. **169**, 110722 (2020)

9. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional, Boston (2004)
10. Kleehaus, M., Uludağ, Ö., Schäfer, P., Matthes, F.: MICROLYZE: a framework for recovering the software architecture in microservice-based environments. In: Mendling, J., Mouratidis, H. (eds.) CAiSE 2018. LNBIP, vol. 317, pp. 148–162. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92901-9_14
11. Krogmann, K.: Reconstruction of Software Component Architectures and Behaviour Models Using Static and Dynamic Analysis, vol. 4. KIT Scientific Publishing, Amsterdam (2012)
12. Mayer, B., Weinreich, R.: An approach to extract the architecture of microservice-based software systems. In: 2018 IEEE Symposium on Service-oriented System Engineering (SOSE), pp. 21–30. IEEE (2018)
13. Singh, S., Kirschner, Y.R., Koziolek, A.: Towards extraction of message-based communication in mixed-technology architectures for performance model. In: Companion of the ACM/SPEC International Conference on Performance Engineering, pp. 133–138 (2021)
14. Terra, R., Valente, M.T., Czarnecki, K., Bigonha, R.S.: Recommending refactorings to reverse software architecture erosion. In: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 335–340. IEEE (2012)
15. Treyer, P.: Extraction of Performance Models from Microservice Applications based on Tracing Information. Master's thesis, Department of Informatics, Karlsruhe Institute of Technology (KIT) (2020)
16. Van Solingen, R., Basili, V., Caldiera, G., Rombach, H.D.: Goal question metric (GQM) approach. Encyclopedia of software engineering (2002)
17. Walter, J., Stier, C., Koziolek, H., Kounev, S.: An expandable extraction framework for architectural performance models. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, pp. 165–170 (2017)