An Approach and Toolset to Semi-Automatically Recover and Visualise Micro-Service Architecture

Nour Ali[1], Nuha Alshuqayran [2], Rana Fakeeh [3], Thoybur Rohman[1] and Carlos Solis[4]

[1] Brunel University London, UK nour.ali@brunel.ac.uk, 2026156@brunel.ac.uk

[2] Imam Mohammad Ibn Saud Islamic University, Saudi Arabia nshaqayran@imamu.edu.sa

[3] AIDA Geschäftsführungs-Organisations-Systeme GmbH Hauptstr. 11, 75391, Germany rfakeeh@aidaorga.de

[4] ION Group, UK carlos.solis@iongroup.com

Abstract. This paper presents the MicroService Architecture Recovery (MiSAR) toolset for software engineers (software architects and developers) that need to semi-automatically obtain as-implemented architectural models of existing microservice-based systems. The MiSAR approach has been designed following Model Driven Architecture, and a set of components have been developed to support the semi-automatic support of MiSAR. The toolset first parses microservice-based systems and generates a Platform-Specific Model, which is an abstract representation of the system using the technology. Then, a model transformation engine automatically generates a Platform Independent Model which represents the as-implemented microservice architectural mode of a system. To support the visualization of as-implemented architectural models, the Graphical Model Generator component of the toolset can be used. The Graphical Model Generator allows the software engineer to obtain quantitative metrics of the microservice architectural model and UML diagrams representing different views of the architecture.

Keywords: Microservice, architecture reconstruction, architecture recovery, architectural views, architecture visualization, model driven engineering, model driven architecture.

1 Introduction

Microservice architecture has become a popular architectural style [1]. Microservices are developed quickly and provide more agility to the system [2], which results in continuous architectural changes [3]. Therefore, it can be stated that not every system is built using a well-documented architecture, and often the documentation of the architecture is not kept up to date [4]. Keeping control of the overall architecture during development can be very difficult, especially when microservice-based systems are designed, developed and deployed by different stakeholders and teams. Moreover, these architectures follow evolutionary design, which is very hard to manage, and architectural constraints are difficult to track. Software engineers often have little knowledge of the as-implemented architecture of their systems, and often face the challenge of not knowing in detail the underlying structures of the software system architecture.

The above concerns can be solved by using software architecture recovery (reconstruction or reverse architecting) [5, 6] which is a technique that reverse engineers systems to obtain the actual (as-implemented) architectural structure and description from system artefacts such as source code.

This paper presents the MicroService Architecture Recovery (MiSAR) toolset, which aims to support the architecture recovery of microservice systems by allowing software engineers to obtain semi-automatically an up-to-date architecture of implemented microservice systems. This can be challenging to obtain manually as microservices are not first-class citizens in the software, microservice systems use different programming frameworks and technologies, and microservices are highly inter-dependent, making analysis and architecture abstraction and comprehension difficult. The MiSAR toolset, manuals, artefacts and its application to case studies can be found at [7]. The MiSAR toolset video demonstration is available at [8].

The paper is structured as follows: Section 2 gives an overview of the MiSAR approach. Section 3 presents the components of the MiSAR toolset. Section 4 describes how MiSAR toolset has been implemented. Section 5 presents a walkthrough of the toolset recovering the architecture of an open-source system. Section 6 evaluates the performance of the toolset. Section 7 presents related work to MiSAR and finally Section 8 concludes and discusses further work.

2 MiSAR Approach

MiSAR follows Model Driven Architecture (MDA) [9], to recover architectural models of existing microservice systems. The initial version of the MiSAR approach has been defined empirically in [10]. To define the MDA artefacts of MiSAR (metamodels and mapping rules), we selected microservice-based open-source systems and recovered their architectures manually. This allowed us to learn by example the architectural elements that need to be included in the metamodels and mapping rules.

The MiSAR approach analyses the microservice software artefacts and produces models at two abstraction levels (see Fig. 1). First, MiSAR analyses the source code of a microservice project represented in text files. Second, it automatically creates a Platform-Specific Model of the project. Third, MiSAR automatically creates a Platform Independent Model which represents the architectural model of the system. To support this, the MiSAR approach includes the following MDA artefacts found at [7]:

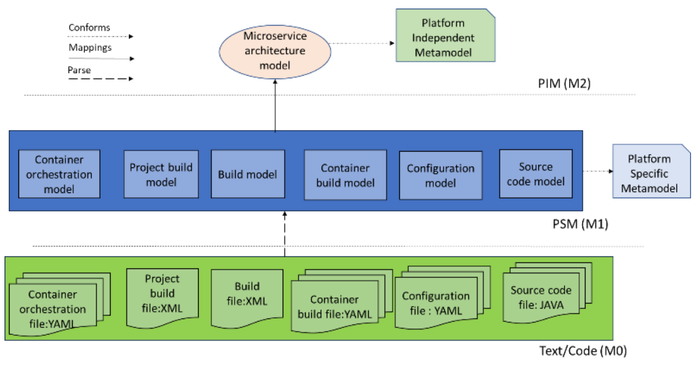
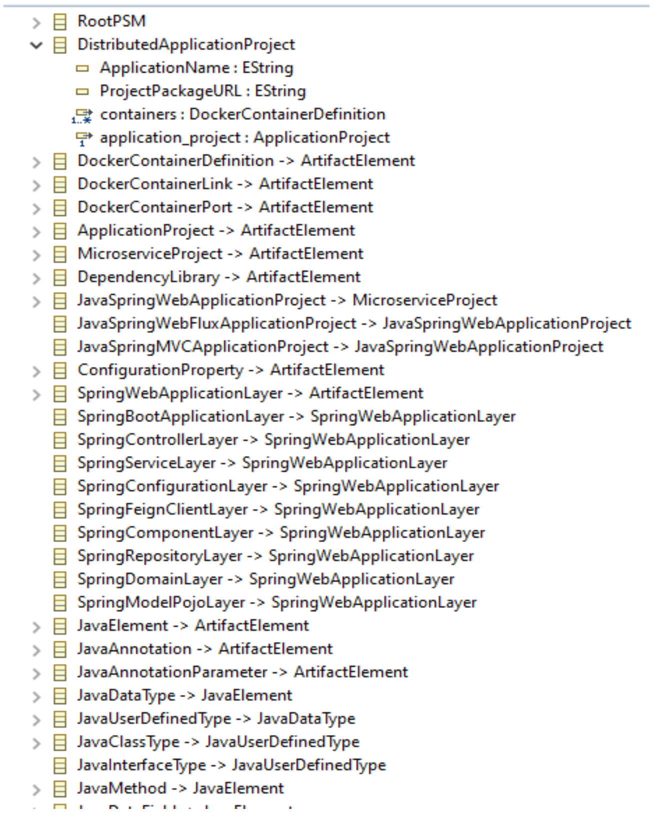

Conforms
Mappings
Parse
Microservice architecture model
Platform Independent Metamodel
PIM (M2)
Container orchestration model
Project build model
Build model
Container build model
Configuration model
Source code model
Platform Specific Metamodel
PSM (M1)
Container orchestration file:YAML
Project build file:XML
Build file:XML
Container build file:YAML
Configuration file : YAML
Source code file: JAVA
Text/Code (M0)

Fig. 1. MiSAR Model Driven Architecture abstraction levels

2.1 The Platform-Specific Metamodel

The Platform-Specific Metamodel defines the constructs which abstract microservice-based systems using the platforms and technologies (see Fig. 2). For each microservice-based system that needs to be recovered, a Platform-Specific Model (PSM) is generated conforming to the Platform-Specific Metamodel. The current platforms and technologies which are supported are the Java Language, Docker, and Spring boot framework and technologies which include Consul, Eureka, MongoDB, MySQL, Neo4j Graph database, OAuth2, and RabbitMQ.

Fig. 2 shows the main elements of the Platform-Specific Metamodel and the PSM.ecore file can be found at [7]. As it can be seen, every PSM of a microservice application has a **DistributedApplicationProject** instance, with an application name and its root repository URI (**ProjectPackageURL**). The **DistributedApplicationProject** is composed of the architecture's development artefacts which include a multi-module project (**ApplicationProject**) and Docker containers represented by the **DockerContainerDefinition**. The **DockerContainerDefinition** elements involved in the architecture are extracted from the Docker Compose and Dockerfile files. The **DockerContainerDefinition** captures **DockerContainerPort** and DockerContainerLink instances. An **ApplicationProject** represents one or many **MicroserviceProject** elements. A **MicroserviceProject** generalises a wide range of project artefacts implemented in any framework or language, including Java Spring Boot/Cloud. The **JavaSpringWebApplicationProject** element is a subtype of the **MicroserviceProject** element which reflects the specific characteristics of applications built with the Spring Boot/Cloud framework. Another characteristic of **JavaSpringWebApplicationProject** is that it aggregates multiple Java classes and/or Java interfaces with a means of annotation into **JavaSpringWebApplicationLayers**.

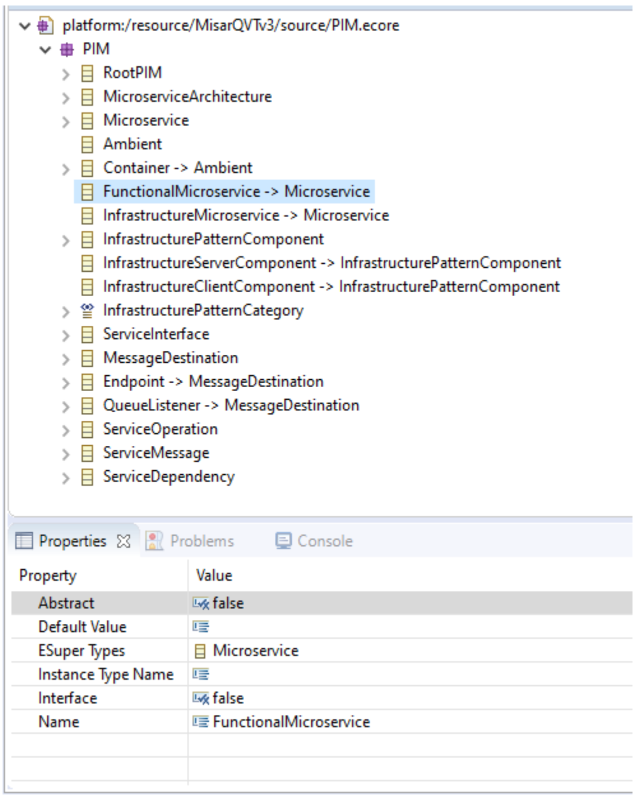


```
RootPSM
DistributedApplicationProject
    ApplicationName : EString
    ProjectPackageURL : EString
    containers : DockerContainerDefinition
    application_project : ApplicationProject
DockerContainerDefinition -> ArtifactElement
DockerContainerLink -> ArtifactElement
DockerContainerPort -> ArtifactElement
ApplicationProject -> ArtifactElement
MicroserviceProject -> ArtifactElement
DependencyLibrary -> ArtifactElement
JavaSpringWebApplicationProject -> MicroserviceProject
JavaSpringWebFluxApplicationProject -> JavaSpringWebApplicationProject
JavaSpringMVCApplicationProject -> JavaSpringWebApplicationProject
ConfigurationProperty -> ArtifactElement
SpringWebApplicationLayer -> ArtifactElement
SpringBootApplicationLayer -> SpringWebApplicationLayer
SpringControllerLayer -> SpringWebApplicationLayer
SpringServiceLayer -> SpringWebApplicationLayer
SpringConfigurationLayer -> SpringWebApplicationLayer
SpringFeignClientLayer -> SpringWebApplicationLayer
SpringComponentLayer -> SpringWebApplicationLayer
SpringRepositoryLayer -> SpringWebApplicationLayer
SpringDomainLayer -> SpringWebApplicationLayer
SpringModelPojoLayer -> SpringWebApplicationLayer
JavaElement -> ArtifactElement
JavaAnnotation -> ArtifactElement
JavaAnnotationParameter -> ArtifactElement
JavaDataType -> JavaElement
JavaUserDefinedType -> JavaDataType
JavaClassType -> JavaUserDefinedType
JavaInterfaceType -> JavaUserDefinedType
JavaMethod -> JavaElement
```

Fig. 2. MiSAR's Platform-Specific Metamodel in Ecore for the Java Language, Docker, and Spring boot

2.2 The Platform-Independent Metamodel

The Platform Independent Metamodel defines the microservice architectural elements that describe a microservice architecture in a technology independent way. The metamodel (see Fig. 3) includes 17 architectural element types. These include Microservices that can be classified into Functional Microservices, which realize the system's business capabilities, and Infrastructure Microservices, which realize infrastructural capabilities. Infrastructure Pattern Components which support the functionality of patterns. MessageDestination type which is an abstract element to represent communication and currently has two subtypes: Endpoints which are service URIs for synchronous remote calls and QueueListeners which are a kind of asynchronous communication. Service Dependencies which describe the communication between a consumer microservice and a provider microservice. Each architectural model recovered conforms to the Platform Independent Metamodel and is called a Platform Independent Model (PIM).



```
platform:/resource/MisarQVTv3/source/PIM.ecore
  PIM
    RootPIM
    MicroserviceArchitecture
    Microservice
    Ambient
    Container -> Ambient
    FunctionalMicroservice -> Microservice
    InfrastructureMicroservice -> Microservice
    InfrastructurePatternComponent
    InfrastructureServerComponent -> InfrastructurePatternComponent
    InfrastructureClientComponent -> InfrastructurePatternComponent
    InfrastructurePatternCategory
    ServiceInterface
    MessageDestination
    Endpoint -> MessageDestination
    QueueListener -> MessageDestination
    ServiceOperation
    ServiceMessage
    ServiceDependency
```

Property	Value
Abstract	false
Default Value	
ESuper Types	Microservice
Instance Type Name	
Interface	false
Name	FunctionalMicroservice

Fig. 3. MiSAR's Platform Independent Metamodel

2.3 Mapping Rules

Mapping Rules map elements of PSMs into PIMs. Each mapping rule is represented with a Left-Hand Side (L-H-S) and a Right-Hand Side (R-H-S). The L-H-S includes PSM elements structured in a tree and the R-H-S indicates targeted PIM elements. The L-H-S PSM elements are checked and if they exist in a PSM instance, then the R-H-S PSM elements are transformed into a group of target PIM elements. An example of a mapping rule is the one which identifies that a java method uses asynchronous communication:

[L-H-S] A Java Method with Element Identifier value: "convertAndSend" whose parent is a Java User Defined Type with Element Identifier value: "RabbitTemplate" or "AmqpTemplate", which has one Java Method Parameter with Parameter Order value: "2" and Field Value value: "[routing-key]" whose type is a Java Class Type with Element Identifier value: "String" such that there is a Queue Listener with Queue Name value that contains: "[routing-key]" and belongs to a Microservice with Microservice Name value: "[provider-name]" **indicates** [R-H-S] a Service Dependency with Provider Destination value: "QueueListener[QueueName:[queue-name]]".

In the above mapping rule a *Service Dependency* PIM element is created which has a QueueListener as a provider *MessageDestination*. MiSAR currently supports 275 mapping rules.

One of the benefits of MiSAR in following the MDA approach is the separation of concerns. Models can be reusable and independent of their graphical notation. As it can be noticed in the following sections, a recovered architectural model (PIM instance) can be obtained without a graphical notation. Consequently, an architectural model can be manipulated in other contexts and transformed into other forms.

Another advantage of MDA is obtaining and using models at different abstraction levels. The PSM is an abstraction that allowed MiSAR to have a structured reverse engineering process and therefore has enabled simple mapping rules (transformations) to generate an architectural model. The PSM allowed the reverse engineering process to first collect and extract which elements from the system and its technologies are needed to construct an architectural model and cluster them. The PSM instance can also be useful for users as it can allow them to trace back and identify which platform and technology elements participated in constructing a recovered architectural model.

3 Components of MiSAR Toolset

The MiSAR toolset is composed of four components which support a user to obtain an architectural model from a microservice system in a semi-automatic way. Each of the components, has as input and/or produces the MDA artefacts explained in Section 2. The components in the toolset are the following:

- AIO: The All In One (AIO) user interface appears when you launch MiSAR. If it is the first launch, it provides guidelines on how to install the toolset components and provides guidelines on using them.
- Parser: MiSAR includes a parser which statically analyses the source files of microservice-based software. The parser analyses these files, collects information from different artefacts, and clusters them into concepts of the PSM. The parser produces a Platform-Specific Model (PSM) of the system by instantiating the Platform-Specific Metamodel. For example, the parser to create a **JavaSpringWebApplicationProject** (explained in section 2.1) object, it analyses different POM files of a system which contain a list of dependency libraries. The parser deserializes each POM file from XML format into a Python dictionary, extracting only the 'parent' and 'dependencies' elements. Each child element within the 'dependencies' element is then converted into a 'DependencyLibrary' PSM object. The information from the source element is collected and organized in attributes within this object. Finally, all the 'DependencyLibrary' objects are clustered inside one parent 'JavaSpringMVCApplicationProject' object. Currently, the parser analyses the following files:
  - **Docker Compose Files** (*.yml*|*.yaml*): These files define services, networks, and volumes for Docker containers.
  - **POM Files** (*.xml*): Maven POM (Project Object Model) files, specify project information, dependencies, and build configurations.

  - **Configuration Files** (*.yml*|*.yaml*|*.properties*): Configuration files in YAML or properties format can be parsed. These files often contain settings, properties, or environment-specific configurations.
  - **Java Source Files**: For Java source files to be parsed, the project needs to have specific libraries in the POM/build.gradle file. Specifically, include either one of the following libraries: 1) org.springframework.boot: which indicates a Java Spring Boot project. 2) org.springframework.cloud: which indicates a Spring Cloud project.
- Model Transformation Engine: MiSAR implements bottom-up model-driven transformations to obtain architectural models. PSMs generated by the parser are fed into model transformations that automatically transform them into PIMs. The model transformations implement the mapping rules and automatically generate the as-implemented architecture model of a system.
- Graphical Model Generator: To improve the understandability of the PIMs, we have developed a Graphical Generator to enable users to visualize the PIM models of the recovered systems. For each PIM, the generator creates: 1) metrics of the PIMs (architectural models) in excel sheets, e.g., a table with the number of architectural elements in an architectural model such as the number of microservices, pattern components and service dependencies, 2) images with graphical UML diagrams of the models and 3) PlantUML [13] files of the models. We currently use the UML Component diagram to represent the microservice architecture. The architecture can also have different views at architecture level and microservice level.

4 Implementation of MiSAR

The Platform Independent and Platform-Specific Metamodels have been implemented as Ecore models using the Eclipse Modeling Framework (EMF) [11] (see Fig. 2 & Fig. 3) The MiSARParser is a python application that incorporates PyEcore, JavaLang, Yaml, XMLtoDict and other python libraries to parse YAML, XML and JAVA artefacts of a microservice-based application such as docker-compose.yml and pom.xml into a MiSAR PSM. The generated PSM is in Ecore (or XMI).

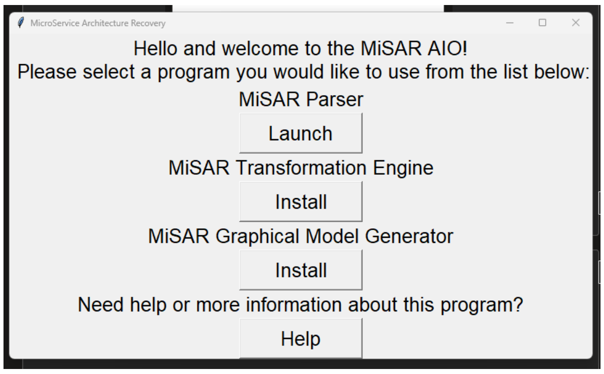
To automate the mapping rules, we have developed the model transformation engine using the Eclipse Model-to-Model Transformation (M2M) project. The 275 mapping rules of MiSAR are written using the operational QVT transformation language (QVTo) [12]. QVTo follows the structure of our mapping rules. The implementation of mapping rules into QVTo, implements the model transformation engine. The model transformation engine receives as input a PSM instance, executes the rules and then produces PIMs in Ecore (or XMI).

Finally, the graphical generator is a java application which navigates through PIMs and automatically translates them into UML graphical notations. The application uses the java Ecore implementations of MiSAR's Platform Independent Metamodel and translates each element into PlantUML textual language [13] to create the images with the diagrams. The java application also creates excel sheets with metrics of the models. We could have implemented a graphical editor by using frameworks such as Graphical

Modeling Projects [14] or EcoreViz [15] which can be integrated into Eclipse. However, we made the decision to be Eclipse independent as Eclipse is heavyweight and can change its versions making our approach obsolete in the future. Since, currently, MiSAR does not require the manipulation of diagrams generated (no human interaction), then this is sufficient. This could change in the future, if MiSAR is to be extended for further software engineering activities such as manipulating models to keep them consistent with the microservice implementation.

5 A Walkthrough of MiSAR

We will demonstrate the steps and artefacts produced using the MiSAR toolset to semi-automatically generate the as-implemented architectural model of a microservice-based system. To demonstrate MiSAR, we have selected a microservice project, which is an open-source project called the MicroCompany application [16]. MicroCompany is implemented using Java Spring Boot/Spring Cloud microservice-based application that consists of 11 microservices of which 4 are business-oriented. It utilises both synchronous and asynchronous inter-service communication.


MicroService Architecture Recovery

Hello and welcome to the MiSAR AIO!
Please select a program you would like to use from the list below:
MiSAR Parser

MiSAR Transformation Engine

MiSAR Graphical Model Generator

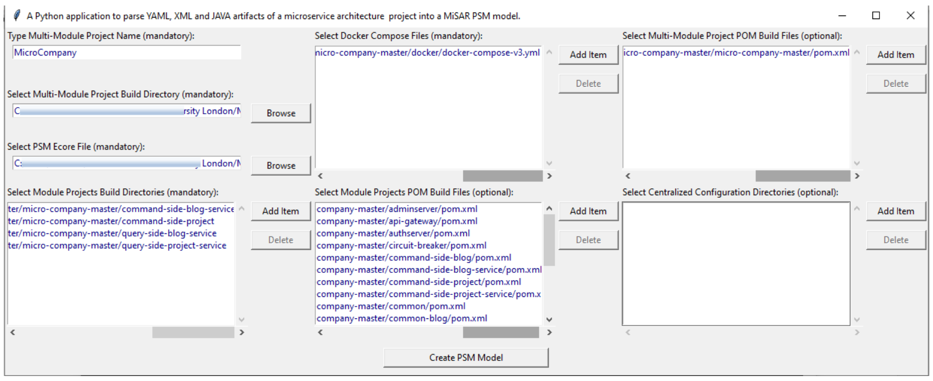
Need help or more information about this program?

Fig. 4. User interface of MiSAR AIO

Consider that the software team, after having developed the MicroCompany application, would need to get an up-to-date architecture of their application. The software team has to follow the installation instructions and manuals found on [17]. A user can use the AIO for installation guidelines as well. Fig. 4 shows the AIO when the user has already installed the parser and it is ready to be launched. To obtain the up-to-date architecture, the user follows the following steps:

Step 1- Parsing the Microservice System to Create a PSM instance: The files from the MicroCompany GitHub are first downloaded locally. Then, the required artefacts are collected and uploaded to the existing MiSAR parser, as illustrated in Fig. 5. The parser receives as input: the Project name, Build directory of the system (multi-module) project, Path of every Docker Compose file (yml), Build directory of every microservice (single-module) project, Path of build file (POM) of the system (multimodule) project and the Path of the build file (POM) for every microservice (single-module) project. Configuration and Java Source artefacts are collected automatically by the parser with the help of the build directory of every microservice project. When the user inputs the Build Directory, the parser asks the user if they would like to import all the files automatically or whether they would like to upload them manually.

The user has the option to delete or add uploaded files. This is to allow the user to control the parts of the system which they would like to recover. Users may want to recover the architecture of the entire microservice system, whereas other users may want to only recover specific parts of the system, e.g., specific microservices.



A Python application to parse YAML, XML and JAVA artifacts of a microservice architecture project into a MiSAR PSM model.

Type Multi-Module Project Name (mandatory):
MicroCompany

Select Multi-Module Project Build Directory (mandatory):
C[redacted]rsity London/N

Select PSM Ecore File (mandatory):
C:[redacted] London/N

Select Module Projects Build Directories (mandatory):
ter/micro-company-master/command-side-blog-service
ter/micro-company-master/command-side-project
ter/micro-company-master/query-side-blog-service
ter/micro-company-master/query-side-project-service

Select Docker Compose Files (mandatory):
nicro-company-master/docker/docker-compose-v3.yml

Select Module Projects POM Build Files (optional):
company-master/adminserver/pom.xml
company-master/api-gateway/pom.xml
company-master/authserver/pom.xml
company-master/circuit-breaker/pom.xml
company-master/command-side-blog/pom.xml
company-master/command-side-blog-service/pom.xml
company-master/command-side-project/pom.xml
company-master/command-side-project-service/pom.x
company-master/common/pom.xml
company-master/common-blog/pom.xml

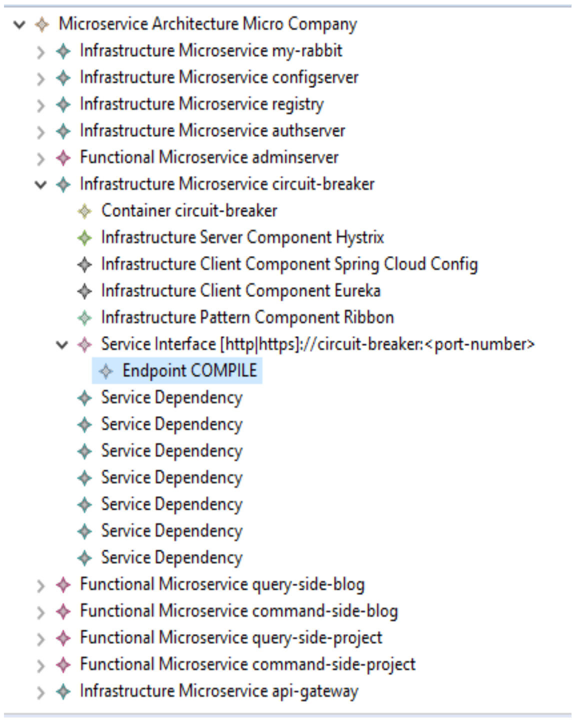
Select Multi-Module Project POM Build Files (optional):
icro-company-master/micro-company-master/pom.xml

Select Centralized Configuration Directories (optional):

Fig. 5. User interface of Parser used to create PSM of MicroCompany

The parser produces a PSM instance for the MicroCompany application. The PSM instance can be found at [18]. Even though the PSM is not the as-implemented architecture model, it is useful, as it provides backtracking support and allows the user to understand the elements that generated the PIM, by checking the specific lines in the artefact that generated those particular PSM elements.

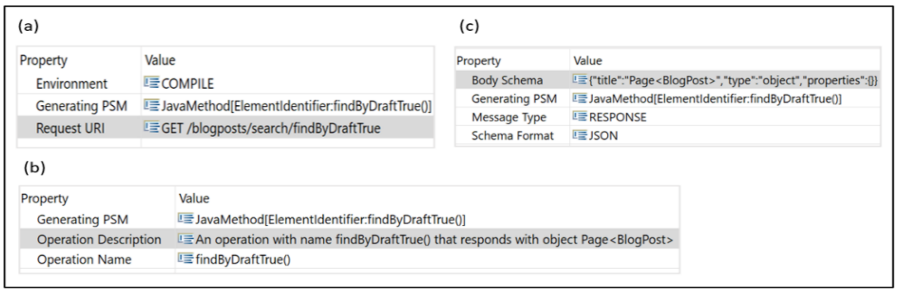
Step 2- Executing Model Transformations to Create the PIM instance: The PIM architectural model is obtained by running the Eclipse QVTo project. The PIM recovered is in XMI format and can be opened as a tree view with Sample Reflective Ecore Model editor provided by the Eclipse Modeling Framework (EMF). Fig. 6 shows the generated architecture of MicroCompany using EMF. It consists of 11 microservices: 6 Infrastructure microservices and 5 Functional microservices. The user can have a more detailed view of the microservices if they click on them. In Fig. 6, the user has clicked on the Infrastructure Microservice called circuit-breaker and can view its associated architectural elements: Container, Infrastructure Server Components, Infrastructure Client Components, Service Interface, Endpoint, and Service Dependencies.



```
Microservice Architecture Micro Company
  Infrastructure Microservice my-rabbit
  Infrastructure Microservice configserver
  Infrastructure Microservice registry
  Infrastructure Microservice authserver
  Functional Microservice adminserver
  Infrastructure Microservice circuit-breaker
    Container circuit-breaker
    Infrastructure Server Component Hystrix
    Infrastructure Client Component Spring Cloud Config
    Infrastructure Client Component Eureka
    Infrastructure Pattern Component Ribbon
    Service Interface [http|https]://circuit-breaker:<port-number>
      Endpoint COMPILE
    Service Dependency
    Service Dependency
    Service Dependency
    Service Dependency
    Service Dependency
    Service Dependency
    Service Dependency
  Functional Microservice query-side-blog
  Functional Microservice command-side-blog
  Functional Microservice query-side-project
  Functional Microservice command-side-project
  Infrastructure Microservice api-gateway
```

Fig. 6. Recovered PIM model for MicroCompany

In addition, the microservice view has the attributes for the microservices. Fig. 7 shows attributes for the recovered microservice called query-side-blog. For instance, (a) the "query-side-blog" microservice exposes an endpoint with request URI "GET /blogposts/search/findByDraftTrue" which is handled by (b) the service operation "findByDraftTrue()" and (c) returns a response service message of model "Page(Blog-Post)". As it can be noticed, one of the attributes is "Generating PSM" which indicates the element from the PSM that was used to generate the attribute. This feature provides traceability and backtracking support for the recovery.



(a)

Property	Value
Environment	COMPILE
Generating PSM	JavaMethod[ElementIdentifier:findByDraftTrue()]
Request URI	GET /blogposts/search/findByDraftTrue

(c)

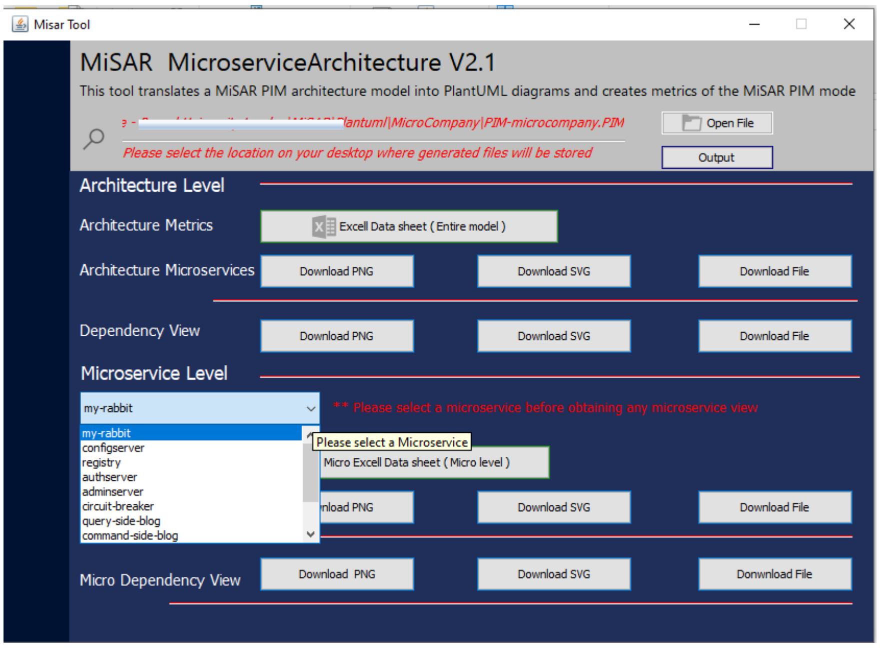
Property	Value
Body Schema	{"title":"Page<BlogPost>","type":"object","properties":{}}
Generating PSM	JavaMethod[ElementIdentifier:findByDraftTrue()]
Message Type	RESPONSE
Schema Format	JSON

(b)

Property	Value
Generating PSM	JavaMethod[ElementIdentifier:findByDraftTrue()]
Operation Description	An operation with name findByDraftTrue() that responds with object Page<BlogPost>
Operation Name	findByDraftTrue()

Fig. 7. Example of the recovered "query-side-blog" microservice attributes

Step 3- Transforming the PIM XMI into Graphical Architectural Diagrams: Once you have a PIM instance, you can explore it in XMI or by using EMF as explained in Step 2. However, if users are not experts in Ecore or they prefer to have an improved visualization experience, e.g., sharing diagrams with their teams, they can use the Graphical Model Generator. The user selects the PIM instance and indicates the location where the different images and excel sheets will be located once produced (see Fig. 8). Then, automatically a drop-down menu with all the microservices of the architectural model of the PIM instance will be visible under Microservice Level. The user can produce images with UML architecture diagrams and metrics at the architecture level or at a microservice level as follows:



This tool translates a MiSAR PIM architecture model into PlantUML diagrams and creates metrics of the MiSAR PIM mode

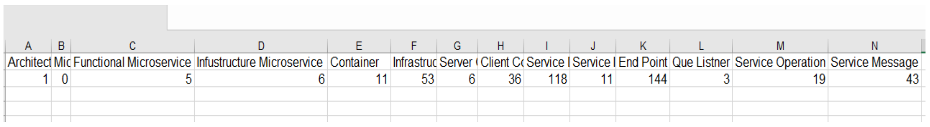
[redacted]lantuml\MicroCompany\PIM-microcompany.PIM

Please select the location on your desktop where generated files will be stored

** Please select a microservice before obtaining any microservice view

Fig. 8. Using the Graphical Model Generator for MicroCompany

At Architecture Level: If the user clicks on the Architecture Metrics Excel Datasheet, an excel sheet is produced that contains the number of architectural elements for every single architectural element type. Fig. 9 shows the excel sheet produced for MicroCompany. For example, there are 5 Functional Microservices and 6 Infrastructure Microservices in MicroCompany. In addition, the user can click under Dependency View and create an image (Download PNG and Download SVG buttons) or get the PlantUML file for the graphical UML diagram. Fig. 10 shows the dependency diagram for the architecture of MicroCompany. The diagram shows the microservices of the architecture and their dependencies. Blue components are Functional microservices and purple components are Infrastructure microservices.



A	B	C	D	E	F	G	H	I	J	K	L	M	N
Architect	Mic	Functional Microservice	Infustructure Microservice	Container	Infrastruc	Server (	Client Cc	Service I	Service I	End Point	Que Listner	Service Operation	Service Message
1	0	5	6	11	53	6	36	118	11	144	3	19	43

Fig. 9. Architecture Level Metrics of recovered MicroCompany

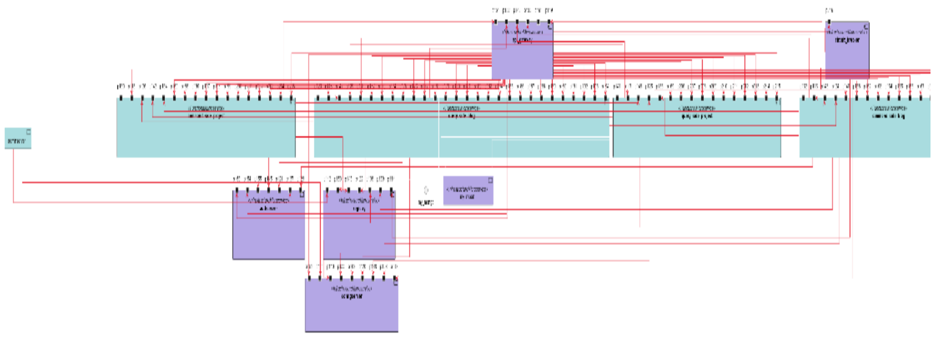
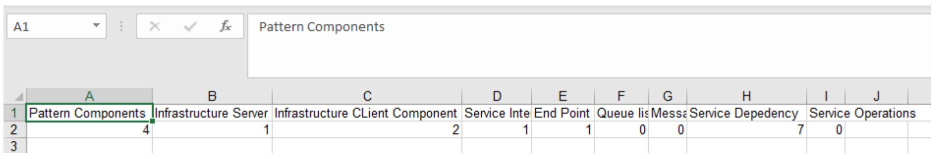


Fig. 10. Architecture Level Dependency View for recovered MicroCompany

At Microservice Level: As it can be noticed from Fig. 10, it is very hard to read the architectural diagram of a medium to large architectural model such as MicroCompany. Therefore, the tool allows the user to select from the top-down menu a specific microservice. Once a microservice is selected, they can create an excel with metrics for that microservice, a microservice view which shows the pattern components, endpoints and service interfaces and a microservice dependency view diagram which shows the microservice chosen and the service dependencies it has with others. Fig. 11 shows the excel sheet generated summarizing the metrics of Circuit-Breaker microservice: it has 4 Pattern Components, 1 Infrastructure Service Component, 2 Infrastructure Client components, 1 Service Interface, 1 endpoint and 7 Service Dependencies. Fig. 12 shows the microservice view for Circuit-Breaker showing that it has 4 InfrastructurePattern-Components (2 of type InfrastructureClientComponents and 1 InfrastructureServer-Component) and an endpoint. Fig. 13 shows the microservice dependency view diagram for Circuit-Breaker. Circuit-Breaker has 7 dependencies with other microservices.



A1 Pattern Components

	A	B	C	D	E	F	G	H	I	J
1	Pattern Components	Infrastructure Server	Infrastructure CLient Component	Service Inte	End Point	Queue lis	Messa	Service Depedency	Service Operations	
2	4	1	2	1	1	0	0	7	0	
3										

Fig. 11. Circuit-Breaker microservice metrics

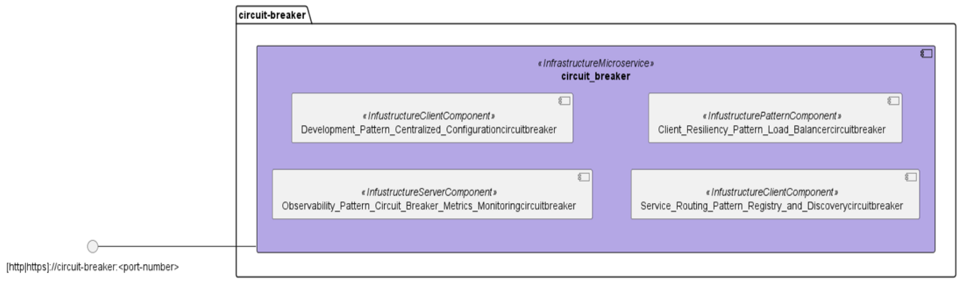

circuit-breaker
«InfrastructureMicroservice»
circuit_breaker
«InfustructureClientComponent»
Development_Pattern_Centralized_Configurationcircuitbreaker
«InfustructurePatternComponent»
Client_Resiliency_Pattern_Load_Balancercircuitbreaker
«InfustructureServerComponent»
Observability_Pattern_Circuit_Breaker_Metrics_Monitoringcircuitbreaker
«InfustructureClientComponent»
Service_Routing_Pattern_Registry_and_Discoverycircuitbreaker
[http|https]://circuit-breaker:<port-number>

Fig. 12. Circuit-Breaker microservice Dependency View

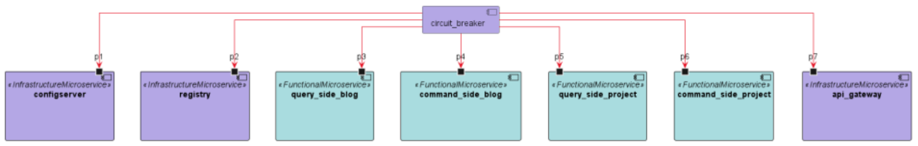


Fig. 13. Circuit-Breaker microservice Dependency View

6 Evaluation

In this section, we evaluate the performance of the tool's components to demonstrate the time it takes for MiSAR to generate the as-implemented architecture (the PIM instance) for three open-source projects. It is important to emphasize that the authors of the papers have not been involved in the development of these open-source projects. Table 1 shows the time it takes, for each toolset component, on an Intel Processor Core (TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s). The time for the Graphical Generator Component is not shown as this is instantaneous. It can be noticed that for a large project, such as TrainTicket, the parser takes most of the time of the recovery process. However, several days could have been taken if software engineers would want to recover the architecture manually. Manual architecture recovery typically requires the involvement of multiple stakeholders to gather knowledge about the system and its interpretation. It relies on the experience of these stakeholders and involves manual analysis of the system's source code [21]. Manual recovery could also produce an inaccurate architecture due to human errors or an architecture with not enough details.

This evaluation has several limitations. Firstly, it has only been conducted on three open-source systems, albeit including a large-scale one such as TrainTicket. Future evaluations could expand to encompass larger and industrial systems. Additionally, the evaluation did not consider the user experience of using the toolset.

Table 1. Time of MiSAR toolset to obtain as-implemented architecture models.

	LOC	Parser to generate PSM (sec)	Model Engine to Transform PSM to PIM (sec)	Total No. of Recovered elements in PIM
MicroCompany [16]	127.1K	9	3.89	490 including 11 microservices
TrainTicket [19]	507.2K	446	63.15	1341 including 69 microservices
MusicStore [20]	116.6K	1	1.07	107 including 9 microservices

7 Related Work

One of the few existing works related to ours is MicroART [22]. MicroART also uses model-driven engineering but does not follow MDA, e.g., it does not define a Platform-Specific Metamodel. In MiSAR, the architectural model is recovered automatically from the PSM, i.e., there is no human input, whereas in MicroART, a software architect needs to identify service discovery services. MiSAR produces architectural models that are richer than MicroART as MicroART only has 8 architectural concepts whereas MiSAR has 17. Therefore, the expressiveness of the MiSAR Platform Independent Metamodel has elements such as Infrastructure Pattern Components and Asynchronous communication which MicroART does not support.

MicroLyze [23] is another work which proposes an architecture recovery approach for microservices. MicroLyze, unlike MiSAR, does not adopt a model-driven approach. Instead, it utilises a distributed tracing component that dynamically monitors simulated user requests. In addition, the work of Wang et al. [24], present an automated recovery process using system source code to build a dependency graph. Like MiSAR, their approach is based on source code analysis. However, their approach does not employ model-driven architecture and does not recover many elements such as patterns.

Another approach that recovers microservice architecture is Kieker [25]. Kieker is a monitoring framework which uses dynamic analysis to discover the architecture of a system. The main elements it extracts (or recovers) are containers and methods. It does not explicitly provide a microservice as an architecture concept and infrastructure pattern components. Since Kieker uses dynamic analysis, the software engineer needs to add jar files next to docker files, execute the microservice systems and manipulate the docker files. In comparison to MiSAR which only statically analyses systems, the recovery of the architecture does not require manipulating any parts of the source code artefacts and does not require the microservice system to be executing. However, with dynamic analysis, the recovered architecture obtains dynamic information such as times of methods which are not recovered by MiSAR.

8 Conclusion

In this paper, we have introduced the MiSAR toolset that semi-automatically generates as-implemented architectural models from existing microservice systems implemented in diverse technologies. We have demonstrated how the MiSAR toolset components can be used to recover architectural models in Ecore (XMI) and if required they can be introduced to be visualized in UML Component diagrams in different views. We have also presented the evaluation of the time it takes for MiSAR to recover the architectures of 3 microservice projects.

Our further work includes improving the usability aspects of the toolset and the efficiency of the parser. As explained in the paper, currently MiSAR only supports Java Spring Boot Applications and/or Docker. We are currently working on a project to extend MiSAR to support the recovery of microservice-based systems partly (or fully) developed in Python. To do so, we need to extend the parser, the Platform-Specific Metamodel and the mapping rules. We will continue working on extending MiSAR to support its analysis of additional languages and technologies.

Furthermore, one existing limitation of our visually generated diagrams is their reliance on PlantUML, which generates static images. This restricts user manipulation of the graphical architecture models, and the layout of the diagrams cannot be controlled. To address this limitation, we plan on creating a diagramming tool. Additionally, we intend to evaluate our approach with practitioners and in industrial settings, rather than solely relying on open-source projects.

References

1. Newman, S., Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc, 2015.
2. Hasselbring, W., and Steinacker, G., Microservice architectures for scalability, agility and reliability in e-commerce, IEEE International Conference on Software Architecture Workshops (ICSAW), 2017, pp. 243–246.
3. Simioni, A. and Vardanega, t., In pursuit of architectural agility: experimenting with microservices, In 2018 IEEE International Conference on Services Computing (SCC), 2018, pp. 113–120.
4. Cerny, M., and Donahoo, T., and Trnka, M., Contextual understanding of microservice architecture, 2018, p. 29–45.
5. S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," IEEE Transactions on Software Engineering, vol. 35, no. 4, pp. 573–591, 2009.
6. Ali, N., Rosik, J., Buckley, J., Characterizing real-time reflexion based architecture recovery: an in-vivo multicase study, 8th international ACM SIGSOFT conference on Quality of software architectures. ACM, Jan. 2012, pp. 23–32.
7. MiSAR, available at https://github.com/MicroServiceArchitectureRecovery/misar
8. MiSAR Toolset video demo: https://youtu.be/sdRDkLesyS0
9. Brambilla, M., Cabot, J. and Wimmer, M, Model-Driven Software Engineering in Practice, 1st ed. Morgan & Claypool, 2012

10. Alshuqayran, N., Ali, N., Evans, R., Towards Micro Service Architecture Recovery: An Empirical Study, IEEE International Conference on Software Architecture (ICSA), 2018, pp. 47–4709.
11. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley 2008.
12. Barendrecht, P. J., Modeling transformations using QVT Operational Mappings, Research project report. Eindhoven: Eindhoven University of Technology Department of Mechanical Engineering Systems Engineering Group, 2010.
13. PlantUML Homepage, https://plantuml.com/, last accessed 2023/05/05.
14. Graphical Modeling Project, https://eclipse.dev/modeling/gmp/
15. Ecore visualization using KIELER, https://github.com/kieler/ecoreviz
16. Dugalic, I., "MicroCompany," Available: https://github.com/idugalic/micro-company, 2022, last accessed 26/04/2023.
17. MiSAR parser and Model Transformation Engine, available at https://github.com/MicroServiceArchitectureRecovery/MiSAR-Parser-and-Model-Transformation
18. MiSAR PSM and PIM instances of MicroComany: https://github.com/MicroServiceArchitectureRecovery/misar/tree/main/EmpiricalStudyReplication/EvaluationOfMiSAR/micro-company-PIM%26PIM
19. X. Zhou, X. Peng, T. Xie, C. J. C. Sun, J.and Xu, and W. Zhao, "Benchmarking microservice systems for software engineering research," in Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE, 2018, p. 323–324.
20. S. OSS, "MusicStore," Available at: https://github.com/SteeltoeOSS/Samples/tree/main/MusicStore, last accessed: 2023/07/06
21. A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen and C. Riva, "Symphony: view-driven software architecture reconstruction," Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), Oslo, Norway, 2004, pp. 122-132, doi: 10.1109/WICSA.2004.1310696.
22. G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, Microart: A software architecture recovery tool for maintaining microservice-based systems, IEEE International Conference on Software Architecture Workshops (ICSAW), 2017, pp. 298–302.
23. Kleehaus, M., Uludag, O., Sch¨afer, P., Matthes, F., MICROLYZE: A framework for recovering the software architecture in microservice based environments, International Conference on Advanced Information Systems Engineering. Springer, Cham., 2018, pp. 148–162.
24. Wang, L., Hu, P., Kong, X., Ouyang, W., Li, B., Xu, H. and Shao, T., Microservice architecture recovery based on intra-service and inter-service features, 2023, Journal of Systems and Software, p.111754
25. W. Hasselbring and A. van Hoorn, Kieker: A monitoring framework for software engineering research, Software Impacts 5(2020), 100019.