Towards Recovering the Software Architecture of Microservice-based Systems

Giona Granchelli†, Mario Cardarelli†, Paolo Di Francesco‡,
Ivano Malavolta‡, Ludovico Iovino‡, Amleto Di Salle†

{giona.granchelli | mario.cardarelli}@student.univaq.it, amleto.disalle@univaq.it
†Gran Sasso Science Institute, L’Aquila, Italy - {paolo.di francesco | ludovico.iovino}@gssi.it
‡Vrije Universiteit Amsterdam, The Netherlands - i.malavolta@vu.nl

Abstract—Today the microservice architectural style is being adopted by many key technological players such as Netflix, Amazon, The Guardian. A microservice architecture is composed of a large set of small services, each running in its own process and communicating with lightweight mechanisms (often via REST APIs). If on one side having a large set of independently developed services helps in terms of developer productivity, scalability, maintainability, on the other side it is very difficult to have a clear understanding of the overall architecture of a microservice-based software system, specially when the deployment and operation of the involved microservices evolves at run-time.

In this paper we present MicroART, an architecture recovery approach for microservice-based systems. By using Model-Driven Engineering techniques, we leverage a suitably defined domain-specific language for representing the key aspects of the architecture of a microservice-based system and provide a toolchain for automatically extracting architecture models of the system. The only inputs of MicroART are: (i) a GitHub repository containing the source code of the system and (ii) a reference to the container engine managing it. We validated MicroART on a publicly available benchmark system, with promising results.

Index Terms—Microservices, Architecture recovery, Model-Driven Engineering.

I. INTRODUCTION

Lewis and Fowler define the microservice architectural (MSA) style as an approach to developing a single system as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API [10]. This style puts emphasis on the design and development of highly maintainable and scalable software where large systems are decomposed into independent services [18]. Services are small in size with respect to Service-Oriented Architecture (SOA) design, independent from each other, and designed using bounded contexts to combine together related functionalities [9]. Alshuqayran et al. [1] state that commonly agreed on benefits of this style are multiple, e.g., increase in agility, developer productivity, resilience, scalability, reliability, maintainability, separation of concerns, and ease of deployment. However, a set of challenges have significant impact on this style, as services discovering over the network, security management, communication optimization, data sharing and performance [1].

Model-Driven Engineering (MDE) promotes models as first-class entities and leverages the abstraction of software development from coding to modeling [5] [21]. The main benefit is the intention to better manage the increasing complexity of modern software while preserving the values of quality attributes of code-centric techniques. A model is a high-level representation of aspects of a system usually employed by software engineers to precisely specify concepts and relationships before the development phase starts. Modeling tools can be sophisticated and they can even generate the skeleton or all of the code without employing explicit programming techniques that can result particularly error-prone.

Reverse engineering is the process of analyzing and comprehending the software and producing a representation of it at a high level of abstraction [3, 4]. In particular, Model-driven Reverse Engineering (MDRE) applies reverse engineering techniques in combination with modeling technologies, to overcome the maintenance problem. Reverse engineering methodologies have many applications and purposes, e.g., design recovery, program comprehension of legacy systems aimed to support the evolution of the system, and description reconstruction of poorly documented systems [4] [17]. One of the main tasks in reverse engineering is design recovery. It reduces the complexity of software systems by leveraging the software architecture abstraction instead of dealing directly with the source code [19]. In detail this task recreates design abstractions from a combination of code, existing design documentation when available, personal experience, and general knowledge about application domains [2]. When the process of reverse engineering produces an explicit architecture representation is usually described as reverse software architec-
ting process [13]. Reverse Engineering techniques have been largely applied in literature for architecture recovery and change dependency analysis [17] [22], but recent studies confirmed that in the microservice architectural style area little investigation is being performed [11] [8].

In this paper we present an architecture recovery approach for microservice-based systems to tackle the problem of the complexity of microservice architecture. The proposed approach is implemented as a prototype named MicroART [11]. More specifically, our approach is able to automatically extract the deployment architecture of a microservice-based system starting from (i) a GitHub repository containing its Docker-based source code and (ii) a reference to the Docker container.
engineering managing it; then, the approach allows software ar-
discovery. Large microservice-based architectures can grow up to hun-
determine the number of services. Organizing and managing a high number of services can be challenging and
refined architectural model is ready to be finalized. In the
The approach is specially tailored for microservice-based systems. Among the most important reasons why we
microservice-based system, confirming the
of the system, and finally decide when the refined architectural model is ready to be finalized. In the
microservice-based systems, e.g., removing unnecessary details, perform model analysis, architectural change impact analysis, overall understanding of the system, and finally decide when
A first assessment of the approach and its prototype validation have been performed on the Acme Air system, an open-source benchmark microservice-based system, confirming the
The rest of the paper is organized as follows. Section II presents the proposed approach and Section III discusses the details of the MicroART DSL. Section IV presents the validation of MicroART with respect to a third-party benchmark system. Section V presents the implementation of our approach, whereas Section VI discusses related work. Section VII closes the paper and discusses future work.

II. RECOVERING MICROSERVICE-BASED ARCHITECTURES

The MicroART approach for microservice architecture recovery is composed of two phases, namely architecture recovery labelled with A and architecture refinement labelled with B, as shown in Figure 1.

The architecture recovery phase deals with all the activities necessary to extract an architectural model of the system starting from its source code repository. From the source code repository we expect to find information about: system (e.g., system and service names), deployment (e.g., service descriptors), product management (e.g., teams, developers, releases). This makes the approach specially tailored for microservice-based systems. Among the most important reasons why we have built our approach on the analysis of repositories there are: (i) they enable automation of the process, by providing access to the source code but also system’s configuration files, (ii) they provide access to the history of the evolution of the system, and (iii) they are widely adopted in industry. For architecture model we mean a model representing the deployment architecture of the system, which is composed by all the elements of the architecture.

The architecture refinement phase aims to refine the initial architecture model into one or more refined architectural models by means of model refinement incremental steps. The software architect can decide to enhance the generated architectural model in order to recover an architecture more suitable for its needs, e.g., removing unnecessary details, perform model analysis, architectural change impact analysis, overall understanding of the system, and finally decide when

A. Architecture Recovery

In the architecture recovery phase, we recover the system architecture by analyzing the system’s source code repository. It is based on the extract-abstract-present paradigm [14], depicted in Figure 2. Generally, in the Extraction phase, the information is extracted from artefacts as the system’s source code, documentation, history, or the architect knowledge. Abstraction is about grouping and filtering information to obtain a meaningful and focused set of information. Presentation is about organizing the information in a way that is familiar to the targeted readers [13].

Figure 3 provides a graphical representation of the architecture Extraction activities. The extraction phase is divided in two major activities: static analysis and dynamic analysis.

In the static analysis, the following information is retrieved by analysing the repository given as input: (i) service descriptors, (ii) system name, and (iii) developers. Service descriptors are simple text files describing properties and configurations of each service in order to efficiently package and run each service and its dependencies. These files contain all the necessary information related to the deployment of each service in the target environment, as the name of both the service and its container, input and output ports of containers, and the build path. Examples of service descriptors are the Docker and Vagrant files, respectively for the Docker container engine and the Vagrant platform. The system name and the information about each developer which has contributed to the repository are collected, specifically name, username, and email.

Once the static analysis is completed, the dynamic analysis activity begins. The dynamic analysis aims to extract the following information: (i) containers information, and (ii) the communication logs. Since this information is not available statically, this operation must be performed at runtime. Two main steps are involved in the dynamic analysis: (i) query the runtime environment in order to identify each container IP address and the network interface used by microservices for their

---

https://www.docker.com

https://www.vagrantup.com
communication, and (ii) the creation of the communication logs. Container engines typically provide runtime environment information by running specific commands (e.g., in Docker the command is `inspect`). By means of these commands, the containers information is extracted and the specific network interface used for microservice communications is discovered. By using a monitoring tool (in our implementation `TcpDump`) this interface can be monitored, and the communication logs can be written to `Log files`.

The Abstraction phase is about grouping and filtering the information gathered in the extraction phase in order to rework the collected information in the desired manner. The extracted information is given as input to a `model factory` that creates the architecture model, which is the final output of the architecture recovery phase as shown in Figure 2. The model generated at this stage conforms to MicroART-DL, our domain-specific modeling language for microservice-based architectures. The concepts and relationships within the MicroART-DL language are presented in Section III.

The abstraction phase includes the mapping procedure that associates the gathered data from the previous phase (see Figure 3) to the MicroART-DL concepts (see Figure 4). As shown in Table I, for each extracted data, a direct mapping to the DSL concept is matched and the result of this association is presented in Section IV.

The first step of the mapping procedure creates an instance of the root metaclass `Product` by setting the extracted system name. For each service descriptor a new `MicroService` class is created, and the corresponding attributes are mapped. For each MicroService class, a new instance of `Team` class will be attached, because a single team is associated to a single microservice. A `Developer` class is created for each developer that has committed to the system repository. On the basis of the commit history and the commit paths, if a match is found with a specific microservice build path, then the `Developer` is assigned to the microservice’s team. On the basis of the communication logs, for each microservice communication an `Interface` class is created on both the service provider and the service consumer side. A source interface is introduced in order to keep track of the source service request, instead the target interface will have more detailed information about the resource needed, i.e., endpoint, protocol and method. For each communication, a new translation into a `Link` class will be applied, representing the connection between two interfaces. The creation and the utilization of the `Cluster` metaclass is a classification, not involved in the translation mapping, that can be used to group microservices under specific characteristics, as also discussed in section III. `Cluster` provides a logical division of the system.

The Presentation phase is related to render the obtained architecture model making the modelled concepts, extracted and mapped in the previous phase available and especially exploitable to the software architect. The architecture model extracted can be rendered with a visual editor, or in a text editor developed and distributed with a concrete syntax part of the infrastructure developed in MicroART, like the editors showing the models rendered in Section IV in Figure 6a and 6b.

B. Architecture Refinement

This phase, labelled in Figure 1 with B is semi-automatic, since it requires the intervention and supervision of the software architect. Initially, a model of the architecture is automatically created, as presented in phase A. After that, refinements are applied on the model, leading thus to the final microservice architecture representation. We define refinement such as the process of modification of the architecture model in a new model, in this case, filtered of some of the contained elements that the architect can easily spot. For this phase the architect interaction is needed since it has to select one or more components in the architecture model in order to filter or resolve that specific components from the system representation. The purpose of this phase is produce another architecture model which the architect considers more significant for its purposes. This new architecture is referred to as refined architecture model, as shown in Figure 1.

The advantages of adopting refined architecture models are several, and could be defined considering the architect needs. Indeed, a refined architecture model could be used for analysis, or for obtaining different views of the system architecture customized on specific components. The first refinement we have applied to our approach is the Service Discovery Resolution, and other refinements can be further defined and integrated in the current platform.

Service Discovery Resolution is the first architectural refinement considered in the approach and its purpose is to resolve the service discovery services in order to reveal the dependencies among microservices. Microservice-based
TABLE I: Mapping between the extracted information and the MicroART-DSL

<table>
<thead>
<tr>
<th>Analysis type</th>
<th>Information</th>
<th>Concept</th>
<th>DSL concept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extracted information</td>
<td>Concept</td>
<td>Metaclass</td>
</tr>
<tr>
<td>Static analysis</td>
<td>GitHub metadata</td>
<td>System Name</td>
<td>Product (name)</td>
</tr>
<tr>
<td></td>
<td>Developers</td>
<td>Developer (Name, Username, Email)</td>
<td></td>
</tr>
<tr>
<td>Service Descriptor</td>
<td>Service name</td>
<td>Team (Name)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Container name</td>
<td>Microservice (Name)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Input ports</td>
<td>Interface (Port)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Output ports</td>
<td>Interface (Port)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Build path</td>
<td>Microservice (Build)</td>
<td></td>
</tr>
<tr>
<td>Dynamic analysis</td>
<td>Containers</td>
<td>Identifiers</td>
<td>Microservice</td>
</tr>
<tr>
<td></td>
<td>IP address</td>
<td>Microservice (Host)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP source address</td>
<td>Link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP target address</td>
<td>Link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>Interface (EndPoint)</td>
<td></td>
</tr>
</tbody>
</table>

architectures adopt service discovery mechanisms in order to keep the microservices dependencies loosely coupled. Since microservices might change their status dynamically (e.g., IP address) for several reasons (e.g., upgrades, autoscaling or failures), the service discovery mechanisms are used to allow services to find each other in the network dynamically. Despite service discovery mechanisms are fundamental to simplify the discovery of services at run-time, they mask the real resource dependencies among microservices in the system. The software architect can use the MicroArt graphical editor for identifying and selecting the service discovery services in the architecture model of the system and obtain a new refined architecture model where dependencies are represented.

The architecture model is refined into the refined architecture model by following three steps: (i) removing all the existing Links among microservices, (ii) removing the service discovery MicroServices identified by the software architect, and (iii) creating new Links by identifying the actual connections between microservices. This last step is feasible because it is now possible to track each request from the service consumer to the targeted service provider.

III. THE MICROART DOMAIN-SPECIFIC LANGUAGE

In this section we present the conceived domain-specific language (DSL) for microservice-based systems and discuss its underlying metamodel, as depicted in Figure 4. The metamodel is composed of seven metaclasses where Product is the root node of the system being designed. MicroService represents the microservices composing the system and its attributes are: host, which is the assigned IP address, and type, which is the type assigned to the microservice. According to the service type classification discussed by Richards [20], we have classified in our DSL the microservices as either functional or infrastructural. The set of possible values of the ServiceType enumeration allows to define a service as functional by assigning to the service the value functional, or implicitly define the service as infrastructural by assigning to it one of the remaining possible infrastructural value, which were extracted according to the classification provided in [8]. Interface represents a communication endpoint and it is attached to specific microservices, for which it represents either an input or output port. Link connects two interfaces together, thus representing the communication among them. Team is composed of one or more developers. Each microservice is owned by one and only one team. The metaclass Developer represents a software developer that participates to the development of the system. Cluster is a logical abstraction for grouping together specific microservices.

[Fig. 4: DSL Metamodel for microservice-based systems]

Our DSL has been developed around the microservice needs and characteristics [10], and it is kept minimal in order to support the design and description of multiple microservice-based systems.

Typical aspects of the MSA style represented in the DSL. First, the notion of products, not project[3] where cross-functional teams are responsible for building and operating each product has been realized by tying together the metaclasses Product, Microservice, Team and Developer. Second,
the Cluster metaclass allows to group together microservice in specific categories, as for example functional and infrastructural services \[15\]. Third, the metaclasses Interface and Link allow to define lightweight communication protocols, by specifying for their representation only the required basic information.

Every model generated by MicroART is an instance of the MicroART-DSL metamodel; examples of model instances are discussed in Section IV.

IV. Validation

The presented approach and its prototype tool have been applied on a publicly available open-source system called Acme Air\[4\]. Acme Air is a microservice-based system of a fictitious airline system implemented in NodeJS, and runs on top of the Docker platform\[5\]. The given input to the MicroART tool is the Acme Air GitHub repository URL and the architecture recovery activity \(A\) starts. The details of the extraction phase are shown in Figure 5, where the used artefacts are shown simply instantiating the extraction phase presented in Figure 3.

![Fig. 5: Acme Air extraction phase](image)

The static analysis extracts the service descriptors, specifically represented by the DockerFiles and the Docker-compose file. The system name (i.e., Acme Air), and the developers information are extracted. The dynamic analysis starts by running the \textit{inspect} command at runtime, from which the container information and the network interface (i.e., Docker Network Bridge) are obtained. Using this information, the running TcpDump monitoring tool detects the communication among services and stores them into log files, while the system usage is simulated.

![https://www.docker.com](https://www.docker.com)
The architecture refinement phase provides to the software architect the possibility to reason, analyse, and refine the architecture model, for instance for performing change impact analysis at the architectural level or having a deep understanding of the overall architecture of the system.

V. TOOL SUPPORT

For supporting the approach presented in Section II, we developed a tool named MicroART in order to guide the architecture recovery of microservice-based systems. This tool first extracts the information from the given repository then it generates an architecture model of the system. This model is then refined by MicroART into a refined architecture model by application of the service discovery resolution refinement.

MicroART has been realized using Model-Driven Engineering tools and development principles [7], working with model-based representations of the microservice architectures, and the tool is publicly available for download [6]. The MicroART tool has been developed on top of Eclipse and in particular the Eclipse Modeling Platform (EMF) in combination with the Spring Framework [7] as can be seen in the bottom layer of Figure [7]. The Extraction Layer is composed of a Repository Analyzer, used for connecting and extracting information from GitHub repositories. The Dynamic Analyzer extracts infor-

---

[Image of architecture models]

Fig. 6: Acme Air architectural models

---

https://github.com/microart/microART-Tool
https://projects.spring.io/spring-framework/
mation from the running system, e.g., Acme Air, and with the Information Aggregator composes the information to be mapped into the MicroART-DSL concepts. This translation is done by the DSL Manager in the Abstraction Layer that creates the architecture models to be rendered with the DSL Visual Editor developed and part of the Presentation Layer, built on top of the GMF tool\cite{https://www.eclipse.org/modeling/gmp}. The DSL manager communicates also with the Plug-in Manager that allows the Developers to integrate their resolution plugin, like the one presented in this paper, i.e., Service Discovery Resolution plug-in. The architect can work with the UI in order to manipulate the models and obtain the refined model version we described in section\ref{sec:refinement}. We outline that the MicroART tool architecture is extensible and it will be improved in the future to manage also other resolution patterns and increase the level of automation in the refinement phase, when allowed. Currently, the MicroART prototype implementation depends on GitHub repositories that use Docker as container engine, and thus MicroART applicability is restricted to projects based on these technologies.

VI. RELATED WORK

The Architecture Reconstruction and MINing (ARMIN) is a tool to reconstruct deployment architectures from the source code presented in\cite{20}. ARMIN starts the recovery process with the source information extraction, where a set of elements and relations are extracted from the system. By using the elements different views of the system’s architecture are generated. The ARMIN approach differs from our because they consider only the source code extraction without considering the dynamic analysis.

Several architecture recovery techniques and tools are presented in\cite{21}. They provide different recovery approaches and give an overview of the state of the art of this field. A framework called Mitre is introduced, and it consists of three components: the architectural representation, the source code recognition engine and “bird’s-eye” program-overview capability. The Mitre’s approach is based on an abstract syntax tree. Compared to our tool despite some step of the recovery approach of Mitre are similar, Mitre is not specific for microservice based system. Also the authors present Architecture Reconstruction Method (ARM), a recovery technique that works in four major phases: development of a concrete pattern-recognition plan, extraction of a source mode, detection and evaluation of pattern instances, reconstruction and analysis of the architecture. Differently from our tool, ARM proposes a method that is aimed at systems developed with design patterns, MicroART has not this kind of limitation. Another techniques is the software architecture reconstruction (SAR) method that is based on a relation partition algebra. This method employs five levels of architecture reconstruction: initial, described, redefined, managed, and optimized. Differently to our technique, which aims to recover the dependency of the system, SAR is oriented to a system information extraction.

A SOA-oriented architecture recovery process is presented in\cite{22}. Similarly to our approach, it is based on both a static and dynamic phases and it uses a set of tools, relying on UML to understand the system details.

In\cite{23} the authors define an ADL by introducing a UML profile that facilitates the incremental integration specification. The approach allows developers to specify and design microservice integration, and provide mechanisms with which to automatically obtain the implementation code for business logic and interoperation among microservices. The approach is generative and differently in our approach we focus on recovering the microservices architecture.

Another work has been considered concerning DSL implementation for architectural description\cite{24}. This approach supports full traceability between source code elements and architectural abstractions, and allows software architects to compare different versions of the generated UML model with each other. The focus is on filling the gap between the design and the implementation of a software system. Differently, in our approach we recover the microservices architecture to overcome the maintenance problem.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we presented an approach for semi-automatically recovering the architecture of microservice-based systems. The approach is based on MDE principles and is composed of two main steps: the first aims at recovering the deployment architecture of the system, while the second step has the goal to refine the obtained architecture. The considered architecture models conform to a dedicated DSL. The approach and the tool have been successfully applied to a third-party benchmark system called Acme Air.

Future work includes the definition of new plugins related to the component resolution functionality, such as those for resolving third-party data stores, load balancers, logging services. Also, we are investigating on identifying and defining a set of metrics for automatically evaluating key aspects of the system, such as the coupling among microservices within the system, their cohesion, their evolvability. Those metrics will live in a shared ecosystem, where third-party actors can reuse and execute them on specific microservice-based
systems, and even aggregate them in order to create new ones. In this context, we are planning to extend the proposed approach to support the incremental application of the metrics at run-time, during the whole lifetime of the system; software architects will have a better guidance with respect to the evolution of the system, e.g., in terms of evolution effort estimation. Finally, we are planning to extend MicroART to support components deployed in the public cloud or in other deployment platforms (e.g., Vagrant), with additional logging and rendering tools, and to investigate the application of MicroART to other microservice technologies (e.g., AWS Lambda serverless Function-as-a-Service). Additional aspects that we are considering are related to data contracts, message exchange patterns and formats, protocols, and integration with DevOps provisioning tools.

ACKNOWLEDGEMENT

This research has been supported by the European Union’s H2020 Programme under grant agreement number 644178 (project CHOReVOLUTION - Automated Synthesis of Dynamic and Secured Choreographies for the Future Internet), and by the Ministry of Economy and Finance, Cipe resolution n. 135/2012 (project INCIPICT - INnovating City Planning through Information and Communication Technologies).

REFERENCES


