COOP - automatiC validatiOn of evOlving microservice comPositions

Olga Groh, Harun Baraki, Alexander Jahl, and Kurt Geihs

Distributed Systems Group
University of Kassel
Kassel, Germany
{groh, baraki, jahl, geihs}@vs.uni-kassel.de

CEUR-WS.org/Vol-2510/sattose2019_paper_4.pdf

Abstract

Current change management solutions apply versionized interfaces to ensure coherent evolution in service environments. This leads to several drawbacks such as an increasing number of service versions and time-consuming testing of backward compatibilities in case of drastic changes. In this paper, we present an approach with continuous integration that automatically validates interfaces for compatibility. Developers are relieved from keeping track of multiple versions of different services. This guarantees that dependent parties coevolve accordingly to the interface changes. Our solution is based on the automatic generation of declarative logic programs for validating compatibilities and extracting changes.

1 Introduction

After the first mentioning of the term Microservice in 2011 [LF14], the new style of architecture emerged quickly. Instead of creating single layered systems, the development shifted towards a composition of Microservices, connected through well-defined interfaces.

Microservices are developed independently from each other and, thus, allow separate development cycles. However, this may cause conflicts due to service changes. A typical measure is to apply service versioning. This requires to preserve all previous versions and leads to an increasing number of services, which still have to be maintained. In case of a model change, the old versions need to be adapted as well to adhere to updates. With an increasing number of services, the complexity of version management rises drastically [Bya13]. This is especially true for projects with short development cycles. In other words, scalability of change management is not fostered.

We present in this paper our framework COOP that provides management for semantic, syntactic and protocol changes. Syntactic changes are changes in the interface structure while semantic changes consider changes in the meaning and context, e. g., of a return value or a parameter. The latter requires an ontology that is referenced by the interface.

By employing the declarative logic programming language ASP (Answer Set Programming) [GK14] instead of OWL (Web Ontology Language) [Bec09], not only changes with respect to the interface are possible, but also changes regarding the ontology definition.

In contrast to existing change management frameworks, such as [FS14] and [RP12], COOP is not bound to a specific protocol. Depending on the applied protocol and the used programming language, the framework generates the client stubs accordingly.

Furthermore, COOP provides a notification mechanism for informing service providers whether prior versions have any active clients. This prevents interruptions and failures during runtime and reduces the resource consumption since unused versions are identified and deleted safely.

The paper is organized as follows. COOP and its components are presented in Section 2. The underlying architecture comprises a central repository and the three main modules Service Description, Client Integration, and Change Detection. Section 3 describes an example workflow for our framework. Relevant related works are summarized in Section 4. Finally, the main findings of this paper are concluded in Section 5.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In: Anne Etien (eds.): Proceedings of the 12th Seminar on Advanced Techniques Tools for Software Evolution, Bolzano, Italy, July 8-10 2019, published at http://ceur-ws.org

2 Architecture of COOP

COOP is composed of a central repository and the three modules Service Description, Client Integration, and Change Detection. While the Service Description module extracts interface details to capture the semantic and syntactic facets of the service, the Client Integration module utilizes the description through the central repository to generate a client stub accordingly and to check for new versions regularly. In case of an interface change, the Client Integration module triggers the Change Detection module to identify incompatibilities. By applying COOP, developers gain a threefold benefit. Firstly, service can be enriched with semantic information by annotating the respective methods in the implementation. The annotation is resolved automatically by means of incooperated ontologies. Secondly, ontologies and interfaces are represented in ASP, which is partcularly siuted for dynamic environments. In contrast to OWL, ASP ontologies allow adaptations of existing knowledge during runtime by employing defaults and externals [GK14]. Thirdly, COOP checks automatically for semantic and syntactic of different versions during development time.

While the Service Description module has to be integrated in the service implementation, the Client Integration and Change Detection modules are included in the client application. However, all modules are available on both, the server and the client side, as both may consume or provide further services. It is thus applicable for Microservice compositions. The framework is depicted in Figure 1. The following sections dive into each component and explain the interplay between them.

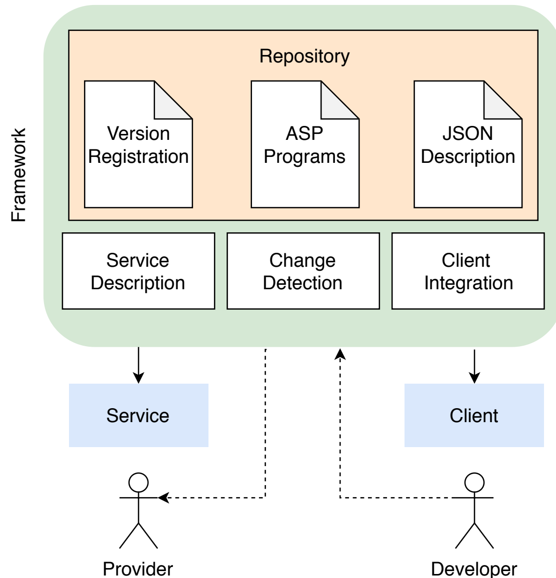

Figure 1: Framework Overview

2.1 Service Description Module

The Service Description module is platform-specific as it is responsible for extracting essential information regarding the syntactic, semantic and protocol aspects from a given service implementation. Thereafter, it creates an intermediate, comprehensive and platform-independent service description. Therefore, the module scans the service implementation and searches for interfaces with specific annotations. This means in particular that the developer has to follow certain implementation guidelines in order to flag the respective interfaces and to enrich them with additional semantic information.

As a clarifying example, the COOP Service Description module for Java shall be considered here. By means of the build tools Maven or Gradle it gets integrated into the build lifecycle of a service. The module searches for typical annotations for service interfaces in Java such as *@RestController* (Spring Framework[1]), *@Path* (Java EE[2]), *@WebService* (Java EE[2]) and others. The annotated interface is then analyzed further for additional details. Assuming that a *@RestController* annotation is present, the Service Description module looks out for annotations such as *@GetMapping* and *@PostMapping*. The attributes of these annotations reveal how a service method is accessed. In particular, this includes the used protocol, the path, and the data format. Further technical details relate to the syntax of parameters and return values. For both, the protocol and the syntax information, the Service Description module applies existing tools to transform the technical details to a JSON representation. If the interface contains complex data types, the module will map them as well. In case of a REST service, the generated JSON section adheres to the OpenAPI specification[3]. By using a widespread specification, the Client Integration module, in turn, is enabled to generate a client stub in different programming languages by third-party tools.

While the previous annotations were required by the respective framework to configure and provide the service, semantic annotations are specific to COOP. For each identified and extracted method, the Service Description module expects semantic information with respect to the parameters, the return value, and the method itself. These information are added by the developer by means of the *@SemanticContext* annotation. The semantic values used in the annotation have to be terms that are defined by the developer or another party in an ASP (Answer Set Programming) dictionary, which is also called an ASP program

[1] https://spring.io/
[2] https://javaee.github.io/javaee-spec/javadocs/
[3] https://github.com/OAI/OpenAPI-Specification

[GK14, BET11]. In contrast to OWL (Web Ontology Language) [Bec09], ASP programs are non-monotonic and, thus, can be extended and updated at runtime without discarding the current knowledge base. Furthermore, they support unique names, which is essential for deriving ASP code from the interface description with our Change Detection module in Section 2.3. Besides that, closed-world reasoning and *defaults* are supported. The latter allows to assume standard values, if certain details are currently missing. For example, we could express that temperature is measured in Celsius by default, but that Fahrenheit and Kelvin are allowed as well.

The developer associates, preferably, each method, parameter and return type with a semantic annotation. However, *defaults* in the ASP program may support the developer. A method annotated with *@SemanticContext("getWeather")* might assume the country code as default parameter and a temperature as response.

The Service Description module adds the semantic annotations and a reference to the ASP program to the JSON description of the service and submits it to the repository. The repository creates an entry in the version registry and provides then the JSON description to other developers.

2.2 Client Integration Module

A developer on client side will employ the Client Integration module to obtain the JSON description of the requested service. Subsequently, the module generates a client stub in the needed language. The stub is integrated in the project and used for development. Developers neither need to worry about the structure of the service used nor do they need to deal with the protocol. Here again, developers should integrate the module in the build lifecycle of their project. After each built, the module triggers a check against the newest version of the service registered at the Version Registration of the repository. If a new version is available, the Change Detection module is executed to determine the deltas between the used and the most current version and whether any incompatibilities exist.

2.3 Change Detection Module

The Change Detection module operates on the client side and is triggered whenever a new version of an involved service is available.

The JSON description will be translated into an ASP program. For example, the JSON structure in Listing 2 is translated to the ASP program in Listing 1. This program will be compared with the locally available version of the service description, which is likewise translated into ASP.

Listing 1: ASP example

```
1  service(holidayService).
2  protocol(holidayService,rest).
3  path(holidayService,"/api/holidays").
4  method(holidayService,getHolidays).
5  context(getHolidays,holiday).
6  return(getHolidays,"list<string>").
7  returnContext(getHolidays,date).
8  parameter(getHolidays,country,string).
9  context(getHolidays,country,countryName).
10 parameter(getHolidays,year,int).
11 context(getHolidays,year,date).
```

Since the JSON description contains syntactic, semantic and protocol information, the generated ASP program allows to identify changes in any of these dimensions. Syntactic information cover the data types of parameters and return values. Incorporating only the syntax, may lead to a successful build process, but omits crucial information with respect to the context. For example, the *HolidayService* in Listing 2 accepts the parameter country with the full country name as value. After an update, the parameter expects a two character country code as defined in ISO 3166 ALPHA-2. The value "Germany" would change to "DE". While the syntactic representation remains the same, i.e. a String, the semantic annotation would be changed accordingly by the developer to *@SemanticContext("iso3166ALPHA2")*. The context in Line 9 (Listing 1) would automatically be transformed to `context(getHolidays,country,iso3166ALPHA2)`.

Regarding the protocol, the Client Integration module generates the client stub. A change in the protocol would be catched automatically by generating a new stub. An alternative protocol would neither affect the syntax nor the semantic and, thus, would relieve developers from taking any further action.

2.4 Repository

The repository stores received ASP programs and JSON descriptions. Besides, it provides the Version Registration. The Service Description module announces new versions to the Version Registration, which, in turn, creates an entry to keep track of the number of registered clients per service version. A developer may trigger through the Client Integration module a registration for the used version. Whenever no more clients are associated to a previous version, the Version Registration informs the respective service provider. This allows the provider to delete the unused version safely.

3 Exemplary Workflow

A Microservice environment typically consists of several projects. Therefore, our solution operates on the complete composition of services. In the following, we provide an example scenario that contains the two separate Microservices *HolidayService* and *OrganizerService*. *HolidayService* provides the dates and names of holidays that are leveraged by the *OrganizerService* to enter them in a calendar.

The development cycles of both Microservices are depicted in Figure 2. After a first version of *HolidayService* is developed and deployed, COOP generates a JSON description of the Microservice, which is kept in a repository. Listing 2 shows an excerpt for *HolidayService* V1. This includes, inter alia, the used protocol, the provided interface and semantic information about the method and the communicated parameters and return values.

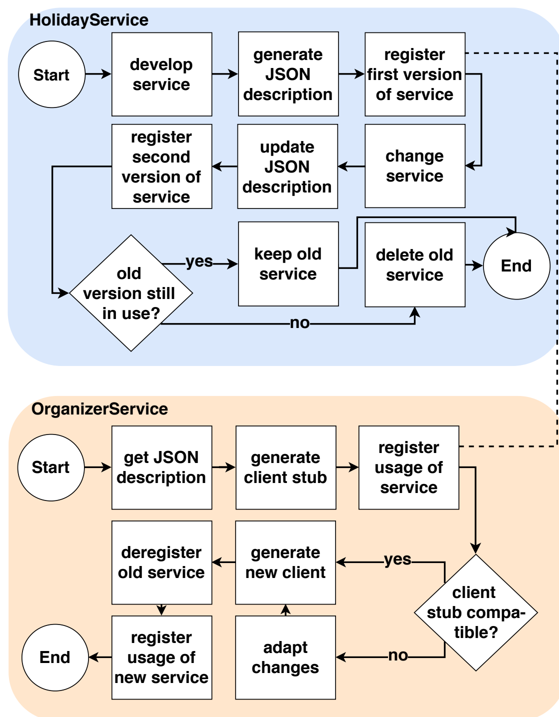

Figure 2: Development Cycle

At the *OrganizerService*, the Client Integration module of COOP fetches the JSON description to generate the client stub for *HolidayService* V1. The stub is automatically included in the dependencies and can be employed by the developer. At the same time, the Client Integration module registers the usage of the current version of *HolidayService*.

By using the provided JSON description of *HolidayService*, *OrganizerService* can detect changes of *HolidayService*. Let us assume, *HolidayService* is extended with the parameter *state* of type String. This change is reflected subsequently in the respective JSON description in the repository. Since the Client Integration module of *OrganizerService* registered for version V1 of *HolidayService*, both versions of *HolidayService* are kept.

Listing 2: JSON example

```
1  {"service": {
2   "name": "HolidayService",
3   "protocol": "REST",
4   "methods": [
5    {"path": "/api/holidays",
6     "name": "getHolidays",
7     "context": "holiday",
8     "returnType": "list<string>",
9     "returnTypeContext": "date",
10    "parameters": [
11     {"name": "country",
12      "type": "string",
13      "context": "countryName"},
14     {"name": "year",
15      "type": "int",
16      "context": "date"}]}]}}
```

On client side, the developer of the *OrganizerService* is informed about the update of *HolidayService*. COOP informs the developer about the type of change and whether the old client stub is compatible to *HolidayService* V2. In case of a compatible change, the developer can switch directly to *HolidayService* V2 by generating a new client stub through the JSON description from the repository. In parallel, the usage of *HolidayService* V1 is deregistered. Accordingly, the developers of *HolidayService* are informed about no registered clients and, hence, may delete V1 safely.

Assuming that the change is incompatible, *HolidayService* V1 would be kept as long as the client is registered for. The developers of *OrganizerService* first have to adapt before switching to V2. Again, the deletion of V1 can be executed when no further clients are registered.

4 Related Work

Providing automated tools for analyzing service behaviour and its changes during the application lifecycle is essential for both change analysis and change management. These tools may detect semantic, syntactic, and protocol changes and inform dependent clients in case of significant changes. This section reviews major approaches of service change detection.

Fokaefs et al. present in [FMT+11] VTracker, a tool for analysing WSDL interfaces. The application identifies differences between two WSDL service descriptions. In particular, the authors created an intermediate XML representation to reduce the verbosity of the WSDL specification. In [FS14], Fokaefs et al. propose the WSDarwin tool. The tool detects and ex-

tracts structural changes in the specification of a service by identifiying changes between different versions of WSDL descriptions. Unlike VTracker, the approach identifies differences between any pair of XML-based documents by comparing all elements with each other. This leads to the computation of the mapping based on their structural similarity. However, WSDarwin is a maintenance support tool that operates exclusively at development time and did not consider semantic service changes.

Similarly, Romano et al. [RP12] present in their work an application called WSDLDiff that recognises more fine-grained changes in WSDL descriptions, by comparing the subsequent versions of WSDL specifications. This approach takes into account the syntax of the WSDL file and the schema file XSD that is used to define the data types of the WSDL interface and enable the developer to analyse how a particular Web service evolves.

In [SAR+16, NKKM04], the authors compare the semantic service description of subsequent versions to extract differences between them. The client developers are relieved due to the automatic extraction of the semantic description of the service implementation. The presented framework assesses semantic change in terms of semantic concept drift using text and structural similarity methods to provide valuable insights. We provide therefore the Service Description Module.

Tran et al. discuss in [TBK+16] an agent-based notification mechanism that focuses on coordinated co-evolution within resource-constrained environments. It detects and informs dependent clients automatically about service changes by means of the analysis of ontology-based service descriptions. The paper mainly emphasizes the need for a notification mechanism, but does not dive into change detection and management.

5 Conclusion

In this paper, we present our COOP framework. COOP automatically validates service compositions with respect to their syntactic, semantic, and protocol compatibility. Therefore, it tracks service changes on behalf of the client and supports the version management of service providers. Furthermore, it enables that dependent parties co-evolve in case of a service change.

Currently, COOP comprises of the Service Description and Client Integration modules, while the first prototype implementation of the Change Detection and Repository are still ongoing. Although the framework is tailored for Microservices, the mechanism can be applied to service-oriented architectures in general.

Acknowledgment

This work is supported by the German Research Foundation (DFG) under the project PROSECCO, grant number 5534111. The authors would like to thank the DFG for supporting their participation in worldwide research networks.

References

[Bec09] Sean Bechhofer. Owl: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009.

[BET11] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[Bya13] Brandon Byars. Enterprise integration using rest. `http://martinfowler.com/articles/enterpriseREST.html#versioning`, Accessed: May 2018, 2013.

[FMT+11] Marios Fokaefs, Rimon Mikhaiel, Nikolaos Tsantalis, Eleni Stroulia, and Alex Lau. An empirical study on web service evolution. In *2011 IEEE International Conference on Web Services*, pages 49–56. IEEE, 2011.

[FS14] Marios Fokaefs and Eleni Stroulia. Wsdarwin: Studying the evolution of web service systems. In *Advanced Web Services*, pages 199–223. Springer, 2014.

[GK14] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach.* Cambridge University Press, 2014.

[LF14] James Lewis and Martin Fowler. Microservices - a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html#footnote-etymology`, Accessed: May 2018, 2014.

[NKKM04] Natalya F Noy, Sandhya Kunnatur, Michel Klein, and Mark A Musen. Tracking changes during ontology evolution. In *International Semantic Web Conference*, pages 259–273. Springer, 2004.

[RP12] Daniele Romano and Martin Pinzger. Analyzing the evolution of web services using fine-grained changes. In *2012 IEEE*

19th International Conference on Web Services, pages 392–399. IEEE, 2012.

[SAR+16] Thanos G Stavropoulos, Stelios Andreadis, Marina Riga, Efstratios Kontopoulos, Panagiotis Mitzias, and Ioannis Kompatsiaris. A framework for measuring semantic drift in ontologies. In *SEMANTiCS (Posters, Demos, SuCCESS)*, 2016.

[TBK+16] Huu Tam Tran, Harun Baraki, Ramaprasad Kuppili, Amir Taherkordi, and Kurt Geihs. A notification management architecture for service co-evolution in the internet of things. In *2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA)*, pages 9–15. IEEE, 2016.