PADERBORN UNIVERSITY
The University for the Information Society

Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften“ (Dr. rer. nat.)

Brokerage as a Service

Efficient Creation of Service Mashups

Simon Schwichtenberg

February 5, 2024

Für Bahar und Nick.

Danksagung

Als ich mit meiner Promotion begann, war ich motiviert etwas Wertvolles zu kreieren. Es hat mich gereizt neue Ansätze für reale Probleme zu entwickeln. Der Weg vom Anfang bis zum Ende meiner Promotion war gespickt von Höhen und Tiefen. Nun kann ich rückblickend konstatieren, dass sich der weite Weg gelohnt hat und mich das Ergebnis mit Stolz erfüllt. Viele Menschen haben mich auf meinem Weg begleitet und haben zum Erfolg meiner Promotion beigetragen. Dafür möchte ich mich herzlich bedanken.

Zunächst möchte ich mich bei meinem Doktorvater Gregor Engels bedanken. Gregor, meine Promotion hat mich sowohl fachlich als auch menschlich bedeutend vorangebracht. Deine Sichtweise habe ich stets als Bereicherung angesehen. Vielen Dank für Dein Vertrauen und Deine Ausdauer.

Ganz herzlich bedanken möchte ich mich bei Christian Gerth. Christian, Deinen Überzeugungskünsten ist es überhaupt zu verdanken, dass ich mit der Promotion begonnen habe. In meiner Masterarbeit, die Du mit betreut hast, wurden bereits die Grundsteine für meine Promotion gelegt. Unsere gemeinsame Projektzeit hat mir sehr viel Freude bereitet. Deine positive Art und Unterstützung hat mich stets motiviert und unser fachlicher Austausch war stets fruchtbar.

Dank gebührt auch meinen Kollegen der Forschungsgruppe Datenbanken und Informationssysteme, die mich mit konstruktiven, fachlichen Diskussionen in meiner Dissertation weiter vorangebracht haben.

Lieber Dank geht auch an meine Familie mit Irene, Wilhelm und Eva, Melanie und Tobias. Ihr hab den Mensch mit geprägt, der ich heute bin. Ihr habt mich immer unterstützt und an mich geglaubt. Meiner Frau Bahar und meinem Sohn Nick gebührt besonderer Dank. Danke Dir Bahar für Deine Hingabe und Liebe. Ich kann mir diesen Moment des Erfolgs nur schwer ohne Deine Unterstützung vorstellen. Nick, Du hast mich enorm motiviert meine Promotion erfolgreich zum Ende zu bringen. Du erinnerst mich täglich an die wirklich wichtigen Dinge des Lebens.

Abstract

A predominant kind of software application is mashups of third-party web services. The third-party services are usually black-boxes that have private source code and public Application Programming Interfaces (APIs) with operations and parameters. Requesters of such black-box services are developers who compose services into mashups.

Today, the creation of mashups is manual and thereby inefficient because the services requested and offered are insufficiently specified. Specifically, three main problems make the mashup creation inefficient: (1) Finding suitable services is cumbersome because request and API specification often mismatch due to terminological heterogeneity. This makes search algorithms ineffective in finding relevant APIs. (2) API specifications lack API protocols, which makes it impossible for requesters to determine all operation call sequences that are required for their mashup. Operations often are not used in isolation, as they have control or data flow interdependencies. (3) Enabling data communication inside the mashups is laborious for the requesters because different APIs use different parameter names and have incompatible data types and formats.

This dissertation introduces *Brokerage as a Service (BaaS)* that pursues these objectives: (1) Resolving the terminological heterogeneity between requests and API specifications by linking them to a global ontology. Terminological normalization improves the effectiveness of finding relevant APIs. A systematic method to choose the most effective techniques to link APIs and ontologies is presented. (2) Deriving operation dependencies by mining API protocols from call-logs. The languages OWL-S, BPMN, and WS-BPEL are examined to identify control constructs needed to describe API protocols and which of these control constructs can be discovered through process mining. It is analyzed which mining algorithms are suitable for deriving API protocols. (3) Making APIs interoperable by generating glue code from parameter mappings. The code generator emits executable program code that makes API calls, extracts relevant parameters from requests and responses, and translates the data. In summary, this dissertation shows how to facilitate mashup creation by adding missing information to requests and API specifications and how to make APIs interoperable.

Zusammenfassung

Softwareapplikationen sind heutzutage oft ein Zusammenschluss von Webservices verschiedener Drittanbieter. Deren Programmcode ist für Entwickler von Mashups nicht zugänglich, während die Schnittstellen (APIs) öffentlich sind. In API Spezifikationen sind Operationen und deren Eingabe- und Ausgabeparameter beschrieben, die Entwickler nutzen um die APIs in Mashups zu integrieren.

Das Erstellen von Mashups ist heutzutage vorwiegend manuell und ineffizient. Dafür sind drei Hauptprobleme verantwortlich: (1) Das Auffinden von relevanten APIs für ein Mashup per Suchalgorithmen ist ineffektiv, weil Suchanfragen und API Spezifikationen aufgrund ihrer terminologischen Heterogenität nicht zusammenpassen. (2) API Spezifikationen enthalten keine API Protokolle was es den Entwicklern von Mashups unmöglich macht alle Operationen zu ermitteln, die für das Mashup benötigt werden. Denn einzelne Operationen können oft nicht in Isolation benutzt werden, da sie Datenflussabhängigkeiten zu anderen Operationen besitzen. (3) Die Kommunikation zwischen verschiedenen APIs innerhalb eines Mashup herzustellen ist aufwändig, da ihre Datenmodelle zueinander inkompatibel sind.

In dieser Dissertation wird Brokerage as a Service eingeführt, welche die zuvor genannten Hauptprobleme auf folgende Weise adressiert: (1) Die terminologische Heterogenität von Suchanfragen und API Spezifikationen wird aufgelöst indem beide mit einer globalen Ontologie verknüpft werden. Es wird gezeigt, dass die dadurch stattfindende terminologische Normalisierung die Effektivität von Suchalgorithmen verbessert. (2) API Protokolle werden automatisch mittels Process Mining aus Aufzeichnungen von API Aufrufen erzeugt. Dazu werden die Kontrollflusskonstrukte aus den Sprachen OWL-S, BPMN, WS-BPEL klassifiziert und den Kontrollflusskonstrukten gegenübergestellt, die per Process Mining erkannt werden können. Es wird analysiert welche Mining Algorithmen sich zum Ableiten von API Protokollen eignen. (3) Interoperabilität zwischen den APIs eines Mashups wird durch das automatische Erzeugen von Glue Code aus korrespondierenden Parametern erzielt. Der Glue Code kann einzelne, für das Mashup relevante Parameter aus komplexen Datenstrukturen extrahieren, die Parameterwerte konvertieren, um damit nachfolgenden API-Aufrufe durchzuführen. Mithilfe dieser drei Lösungen ermöglicht es BaaS API Spezifikationen mit fehlenden Informationen anzureichern und APIs miteinander interoperabel zu machen.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In modern software development, a predominant kind of software application is service mashups that are an interplay of different web services provided by various independent third-party service providers. The single services realize just a small task, are self-contained and work in isolation. Combining web services into mashups can solve more complex tasks than individual services can do alone. To solve a complex task, services need to interact with each other and exchange data.

While there are also other competing technologies such as GraphlQL[1] or gRPC[2], the REST architectural style [2] has prevailed as the de-facto standard for realizing web services. Such RESTful web services can be consumed through a programming-language neutral Application Programming Interface (API). The API of a RESTful web service is a REST API. Developers of mashups are often dependent on certain data that are processed in the mashup. Often a practice-proven way to get such data is via an API from a third party. Thus, these APIs have to be integrated into the mashup.

1.1 Application Scenario

This section introduces a real-world scenario for mashup creation which is depicted in Figure 1.1. A service requester is a software developer that searches APIs to combine them into a mashup. In the scenario, the requester wants to create a mashup for travel arrangements. A travel arrangement consists of a flight, rental car, and hotel room reservation that are connected: The day of the flight arrival, the pickup of the car, and the hotel check-in are identical. Also, the location of the arrival airport, the rental office, and the hotel are

[1] https://graphql.org/
[2] https://grpc.io/

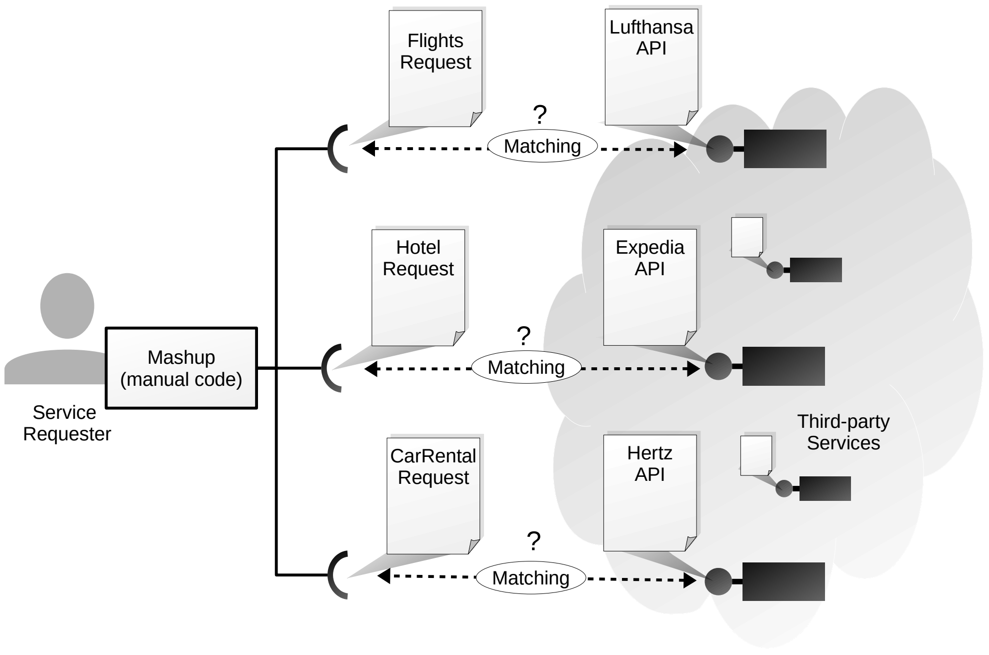


Figure 1.1: Matching requests and third-party APIs for mashup creation

in same the geographic region. To realize this mashup, the requester needs to find three individual APIs for flight, car rental, and hotel room reservation and to combine them. He needs to find all available third-party APIs that match his requirements best possible. In this case, the Lufthansa API, Hertz API, and Expedia API provide what the requester is looking for.

Every time service requesters compose third-party APIs into mashups, the same questions keep coming up: (1) How to find suitable APIs among thousands of third-party APIs? (2) Which operations of the third-party APIs are needed for the mashup, what are the interdependencies of these operations, and in which order they have to be called? (3) Which parameters have to be exchanged between the APIs and how can they be exchanged although they have incompatible data types and formats?

Finding the answers to these questions is mainly a manual process that makes service composition nowadays inefficient as described in the following problem analysis.

1.2 Problem Analysis

This section generalizes the problematic aspects of inefficient mashup creation that are shown in terms of the application scenario in Section 1.1. In practice, service requests and API specifications are incomplete and heterogeneous, i.e., ontological and behavioral semantics of an API is usually barely specified. Based on insufficient information, requesters can not decide whether a certain API is relevant for their mashup and how that API can be composed with others. In particular, the creation of mashups is inefficient because of the following three problems:

(1) Requests and APIs are terminologically heterogeneous which makes searching relevant APIs ineffective.

(2) API protocols are unspecified but necessary to determine which operations are needed in a mashup and in which order the operations have to be called.

(3) Incompatibility of APIs makes it difficult to share data between APIs and to write the glue code such that the APIs are able to exchange the data.

All three problems are shown in Figure 1.2. In the following, the individual three problems are described in detail and opposed to related works.

(1) Request–API Terminological Heterogeneity Today, service requesters use general search engines like Google, Bing, etc. to search APIs. General search engines search through trillions of any kind of web content. Thus, a service requester that uses such a general search engine to search for API specifications may find irrelevant web content, making this method inefficient.

Alternatively, service providers publish their API specifications on public API registries, where requesters can search them. For example, there are public API registries such as RapidAPI[3], SmartAPI[4], APIs.guru[5], etc. The content of public service registries is limited to API specifications per se, which makes the discovery of APIs more effective than with general search engines.

On RapidAPI, service requesters can browse the APIs by categories or use a keyword-based search function to find relevant APIs. Search functions that solely base on simple string similarity tend to be ineffective, because

[3]`https://rapidapi.com/`
[4]`http://smart-api.info/registry`
[5]`https://apis.guru`

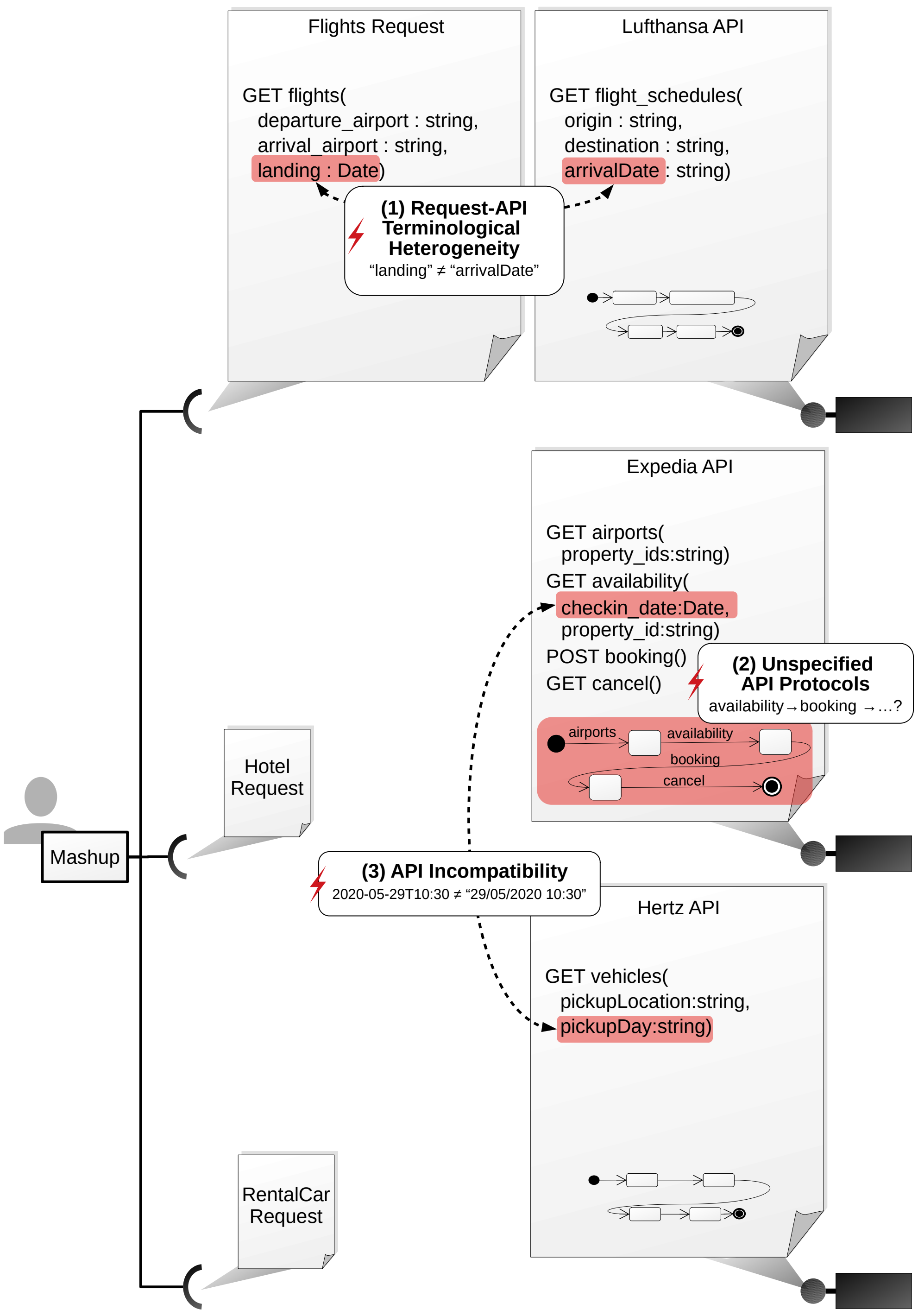


Figure 1.2: Problems related to inefficient mashup creation

requesters and providers may use different terms, e.g., synonyms to describe the same concepts. Synonyms are not string-similar at all so that string-based techniques fail to work. It is also state-of-the-art technology to look up synonyms in a digital dictionary like WordNet [3]. The problem with this approach is that words can have different meanings such that disambiguation of words depending on the context can be difficult.

This problem of terminological heterogeneity is illustrated in Figure 1.2: The Flights Request requires an operation *GET flights* that returns all flight connections from *departure_airport* to *arrival_airport* on a given *landing* date. The Lufthansa API has an operation *GET flight_schedules* that provides the required functionality, but it is described using different terminology. In particular, the parameter *landing* from the *Flights Request* corresponds with the parameter *arrivalDate* from the Lufthansa API. Because of this terminological heterogeneity between the request and the APIs, the requester is not able to find the Lufthansa API.

More advanced service discovery approaches like those presented in [4] mitigate the problem of terminological heterogeneity by combining string-based and semantic techniques. Semantic techniques require that requests and API specifications are linked with ontologies that model concepts and their relations in certain areas of concern. Semantic service discovery can exploit these ontology links to determine the semantic relations between requests and APIs.

Semantic approaches have not been adopted in practice on a large scale until today. Consequently, real-world requests and API specifications are not linked to ontologies, so that semantic techniques cannot unfold their full potential and ultimately fall back on string-based techniques. For example, not a single of the 1,986 specifications on RapidAPI contains links to ontologies.

What would requesters and providers have to do to add those links? How are they able to find relevant ontological concepts in ontologies that have thousands of concepts? It needs a great deal of manual effort to find ontological concepts that are relevant. The efficiency requesters gain by a more effective service discovery through semantic links should not come at the expense of service providers who have greater effort to link their API specifications to ontologies. Several approaches have been proposed to link requests/API specifications to ontologies. These approaches are discussed in the remainder.

In her work, Huma [5] assumes that requests and API specifications already use local ontologies and that local ontologies from requesters and providers are terminologically heterogeneous. To normalize the local ontologies of the requesters and providers they are aligned with a global ontology. In practice, however, API specifications do not contain local ontologies but data models containing only named types. Huma's approach is useful to create new web services from scratch but it is incompatible with existing REST APIs as they usually do not base on ontologies. As a result, the vast amount of REST APIs remains inaccessible for efficient mashup creation with this approach.

The *METEOR-S Web Service Annotation Framework (MWSAF)* [6] allows to automatically categorize APIs by assigning them to ontological classes. Categorizing APIs on the level of ontologies does not make the search for relevant APIs much more efficient: There might be still many APIs assigned to the same ontology which barely narrows down the number of relevant APIs such that requesters still have to inspect many irrelevant APIs.

Other approaches are on a more fine-grained level. Instead of mapping whole APIs to an ontology, they map single API parameters to ontological concepts: Oliveira et al. [7] propose *OntoGenesis* which calculates a similarity score between API parameters and ontological concepts. The similarity of a parameter and a concept bases on the number of shared values. This technique is suitable only for finite value ranges, but APIs have also parameters with infinite value ranges. Even if the value ranges are finite, the complete value ranges are publicly unspecified. In practice, only a few sample values of the API parameters and ontological concepts are known while the complete value range is much bigger. To reliably determine the similarity of a parameter and an ontological concept, according to the law of large numbers, it is not sufficient to infer the generality from a few cases. In addition, parameter values, e.g. string literals can also be terminologically heterogeneous so that the number of shared values may be inaccurate, falsifying the search results.

Maleshkova et al. [8] propose *SWEET* that extracts parameter names from API specifications and uses the external semantic search engine *Watson* [9] to find suitable ontological concepts. Cremaschi et al. [10] propose *AutomAPIc* that uses the Stanford CoreNLP [11] natural language entity recognition to extract concepts from textual API specifications. Hess et al. [12] propose *ASSAM* which uses a combination of different string matching techniques to determine the similarity of API parameters and ontological concepts. *SWEET*, *AutomAPIc*, and *ASSAM* use different techniques to match API specifications with ontologies. The problem with these approaches is that they do not scale with a large number of APIs: The approaches are either too precise but fail to find many relevant ontological concepts, or they are imprecise, but find too many irrelevant ontological concepts.

(2) Unspecified API Protocols Once suitable APIs have been found, it comes to their integration into an executable mashup. This includes the identification of those operations that are needed for a mashup. APIs usually have many operations, but mashups usually do not need the full range of operations, but just a fraction. Requesters have to find out which of the operations their mashup has to call to complete its purpose. On the other hand, single operations may have control and data flow dependencies on other operations such that they cannot be called in isolation. This means that dependent operations have to be called in a certain order because otherwise, the mashup is dysfunctional.

An example can be seen in Figure 1.2. The Expedia API has multiple operations, but only a few are required for the mashup and the requester has to decide which operations are needed. The operation *GET availability* returns the free capacity of a given hotel identified by a Expedia-specific *property_id* on a given *checkin_date*. To obtain a valid *property_id*, the operation *GET airports* must be called before. Requesters have to laboriously find out these operation dependencies. It would have been better if the Expedia API explicitly specified its API protocol that prescribes which operation have to be called in which order.

To detect such operation dependencies computer-assisted, Bertolino et al. [13] propose to recognize data flow dependencies between operations: A dependency between two operations is detected if the first returns an output with data type T and the second consumes an input of type T. Bertolino et al. address Remote Procedure Call (RPC) style web services and their approach is not applicable to RESTful web services, where the data types are JSON formats that are often unique per operation, which means there is often no such type T that is shared between two operations.

Dustdar and Gombotz [14, 15] propose to discover interaction dependencies of APIs composed in mashups using process mining techniques. The approach is applied *after* the composition to analyze if the actual behavior of the mashup conforms to the expected behavior. In their work, only the execution paths that are taken in an existing mashup are in the focus. Opposed to the work of Dustdar and Gombotz, the goal of this dissertation is rather to mine all operation dependencies covering all possible execution paths that may be taken in future mashups. The entirety of all possible execution paths is the API protocol. The approach presented in this dissertation is applied *before* APIs are composed into mashups. Without a comprehensive API specification including its protocol, the requester cannot operate an API correctly since it is not known in which order operations must be called.

(3) API Incompatibility APIs that are composed into a mashup need to exchange data when they are interacting with one another. For example, an output parameter of API X may serve as an input parameter of another API Y. APIs may have dozens or hundreds of parameters, while not all parameters are required in a mashup. The requester has to find out which input and output parameters are critical and have to be exchanged in a mashup. Finding corresponding parameters of different APIs is very time-consuming and therefore inefficient since the requester has to compare dozen of parameters pairwise to decide which are semantically similar and syntactically compatible.

The terminological mismatch between the APIs to be composed in the exemplary travel arrangement mashup can be seen in Figure 1.2. When a travel arrangement is planned with the help of the mashup, the arrival date of the flight, the pick-up date of the rental car, and the check-in date are all on the same day. Thus, this time-based data needs to be shared between the APIs. For example, the parameters *pickupDay* and *checkin_date* have to be shared between the Hertz API and the Expedia API. By just considering the parameter names it is not obvious that they are corresponding with each other. This terminological heterogeneity between the APIs makes it difficult for the requester to find parameters that need to be shared across APIs.

Izquierdo et al. [16] present a set of guiding rules to identify corresponding parameters of two given APIs. The correspondences of parameters are determined based on their names and types, which is not effective when APIs are strongly heterogeneous.

AutomAPIc [10] assists with linking APIs to ontologies and also with composing them. On the basis of ontological links and a set of predefined rules, *AutomAPIc* automatically detects which pairs of APIs are generally compatible, independent from a concrete service request. *AutomAPIc* already considers two APIs as compatible when there exists at least one parameter which is linked to the same semantic type. This definition of compatibility is too weak for practice because mapping one parameter is usually not sufficient as APIs often have multiple mandatory parameters.

In addition to the challenge that the parameter names of different APIs are terminologically heterogeneous, there is also the challenge that parameter values are not compatible because they have different types and formats. In practice, the requester writes the glue code that translates one parameter value into another. Programming the glue code to convert the data requires a great deal of manual effort.

An example can be seen in Figure 1.2: The parameter *pickupDay* of the Hertz API is a *string*-encoded date and the parameter *checkin_date* of the Expedia API is of type *Date*. The first uses the date format *YYYY-MM-DDTHH:mm* while the latter uses the date format *DD/MM/YYYY HH:mm*. Hence, the values of these parameters cannot be exchanged directly. In general, data exchanged between APIs need to be preprocessed, filtered, converted, supplemented, corrected, etc. before it can be exchanged across APIs, which, in practice, is done manually. The process of combining different APIs requires a great deal of manual effort and is one of the reasons why API composition is inefficient until today.

Hess et al. [12] propose to generate complex transformations to transform local to global types and vice versa. So-called lifting transformations transform a local type of API X to a global type. Lowering transformations transform the global type to another local type of API Y. Hess et al. do not consider that the values of both local types of API X and API Y may have incompatible values even though they are mapped to the same global type.

Burstein et al. [17] present an approach that can handle complex transformations for the glue code generation. The translation problem is cast as solving higher-order functional equations. The problem with this approach is that most developers are not familiar with lambda calculus and it is very time-consuming to describe an entire API in lambda calculus. This makes the approach impractical in real-world scenarios.

The approaches [16, 10, 12] only offer limited support for making APIs interoperable and do not address complex parameter mappings with is necessary to achieve executable mashups. Therefore, these approaches do not scale in practice. The approach [17] addresses complex mappings but requires describing APIs with functional equations. The assumption that requesters and providers are proficient in writing functional equations is unrealistic which restricts the learn-ability of the approach.

To summarize all three problems (1-3), it can be seen that today, the creation of mashups is inefficient, because requesters do not find relevant APIs due to terminological heterogeneity, requesters do not know how to use an API correctly, because of missing API protocol specifications, and requesters have to write glue code manually to make APIs compatible. The following section formulates the requirements of an approach that addresses the remaining problems.

1.3 Requirements

In the previous section the shortcomings of state-of-the-art approaches for efficient mashup creation with respect to terminological heterogeneity between requests and API specifications, the lack of API protocols, and the incompatibility of APIs are discussed. Derived from these shortcomings, this section describes the cross-cutting requirements of a new approach that resolves terminological heterogeneity to find relevant APIs, creates API protocol specifications, and generates the glue code that makes APIs compatible.

1.3.1 Upward Compatibility

The enormous plethora of existing RESTful web services brings a lot of ready-to-use functionality that, in its current shape, cannot be efficiently reused in mashups. Unfortunately, due to the lack of sufficient specifications, RESTful web services are not very suitable to be composed efficiently. If complete new kinds of services would be developed especially for efficient composition, the existing infrastructure could not be used anymore and a new infrastructure of services needed to be built up. For this reason, the approach presented in this dissertation targets REST APIs and builds upon the current infrastructure. Therefore, it is one requirement that no changes to RESTful web service implementations or their REST APIs shall be required.

1.3.2 Learnability

Any additional skills that requesters and providers must learn to use the approach should be minimized. Skills that need to be learned represent a possible barrier to acceptance of the approach and its adoption. The approach should be easy to learn and use for both requester and provider which is important to become accepted and adopted in practice. It is beneficial for the acceptance of an approach when it is easy to learn or builds on the existing knowledge and skills of the requesters and providers. REST is an approved technology [18] and there are already specification formats especially like OpenAPI that are widely adopted in practice. Therefore, mashup creation should build upon established technologies like REST and OpenAPI that many requesters and providers are already proficient in.

1.3.3 Effectiveness

There is an enormous amount of third-party APIs that is publicly available. It is important that the efficiency of the API search scales with the huge number of APIs. The complete creation of mashups from request to executable software is very complex that can hardly be fully automated. A first step in this direction reduces manual work. The greater the amount of third-party APIs, the more important are effective techniques to find relevant APIs: The large set of APIs needs to be narrowed down to those APIs that are truly relevant for the requester which reduces the manual work to review, assess, and select relevant APIs that can be reused in a mashup.

1.3.4 Comprehensiveness

API specifications need to be comprehensive such that requesters can decide whether an API provides the desired functionality and can be used in a mashup. For this purpose, the ontological and behavioral semantics of the APIs must be specified. Ontological semantics describe the concepts and their relations in the APIs area of concern. One important aspect of an API's behavioral semantics is the dependencies of its operations.

1.3.5 Interoperability

APIs of third-party services are usually syntactically incompatible and cannot be directly used together in a mashup. First, they must be made interoperable. APIs are interoperable if they can communicate, cooperate, and exchange data via a shared set of exchange types and formats without the need for separate arrangements.

In summary, an approach for semi-automatic mashup creation needs to be upward-compatible with existing REST APIs so that these APIs can be used for mashups, be easy to learn to get acceptance from requesters and providers, scale for the large amount of APIs that exist today, output comprehensive specifications to supply all information requesters need, and achieve interoperability of APIs so that they can be used together in a mashup.

The following sections present a solution that addresses the problems stated above and meets the requirements.

1.4 Solution Approach

In this dissertation, I propose a novel IT service called *Brokerage as a Service (BaaS)* that substantially supports providers and requesters to create mashups

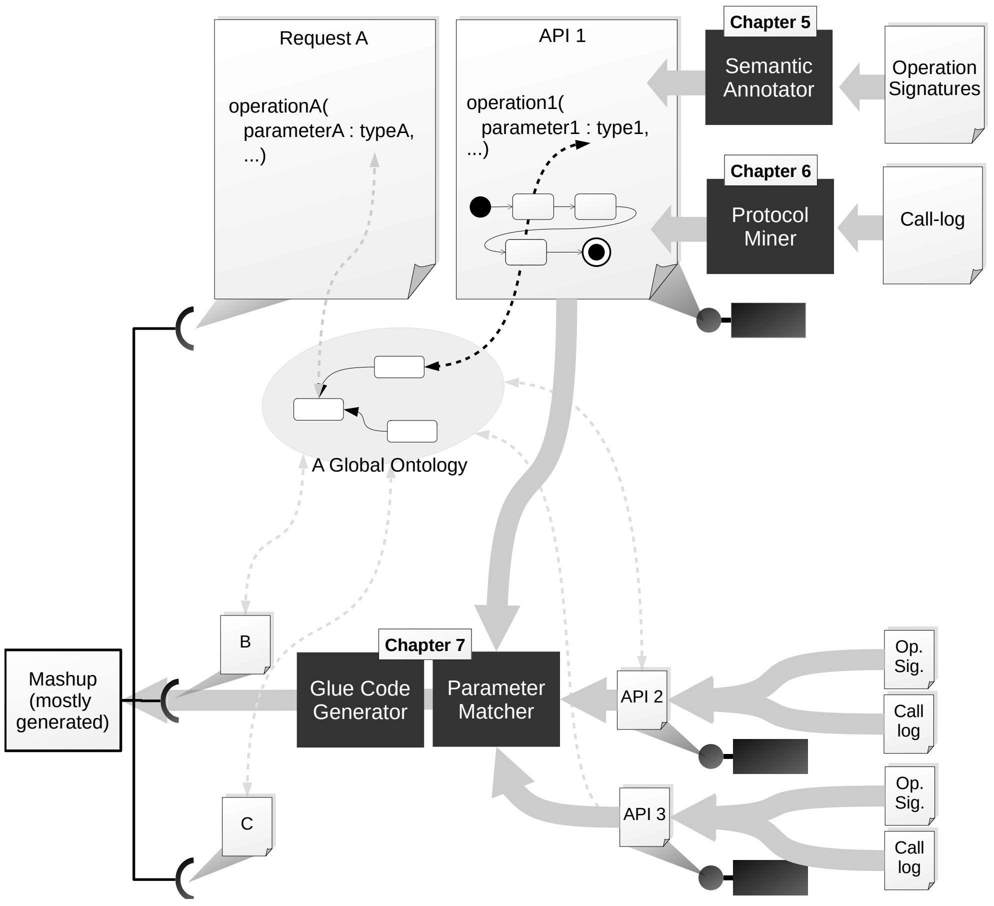


Figure 1.3: Solution overview

efficiently. On one hand, this includes support for the requester to find relevant REST APIs effectively, to find the necessary operations of that REST APIs, to find which parameters need to be exchanged (produced and consumed) between the REST APIs, and to generate the glue code that translates incompatible parameter values to make REST APIs interoperable. The approach presented in this dissertation primarily addresses REST, but can essentially also be adapted for other technologies such as GraphQL and gRPC. Besides the substantial support for requesters, BaaS also includes includes techniques for service providers to link their API specifications with an ontology and to derive API protocol specifications.

Figure 1.3 shows the three main components of BaaS, the *Semantic Anno-*

tator, the *Protocol Miner*, and the *Glue Code Generator* and the connections between them. The components are explained in detail in the following paragraphs.

(1) Semantic Annotator In this dissertation, I propose to normalize service requests and API specifications by aligning them to a common global ontology. Global ontologies such as schema.org are available today and are also widely used in certain areas of application, for example in the semantic annotation of web content [19]. When requesters and providers use a common set of ontological concepts, the terminological heterogeneity between them is resolved so that relevant APIs can be found effectively. The *Semantic Annotator* supports substantial semi-automatic support for requesters and providers to link their API specifications and service requests to concepts from a shared global ontology.

The effectiveness of the search depends on two main factors: (1) Which string matching techniques are used for the search and (2) what contextual information is included in the search keywords. The search is effective when it has high precision and high recall. Precision is the fraction of relevant concepts among those that have been found. Recall is the fraction of relevant concepts among all relevant concepts (even those that have not been found).

Similar to the approaches [8, 6], the *Semantic Annotator* internally uses combinations of different matching techniques. Which techniques are combined and how they are configured has a major effect on their effectiveness. The challenge with linking API specifications to ontologies is to select the matching techniques and add contextual information in such a way that precision and recall are maximized. In this dissertation, I propose a systematic approach to decide which matching techniques and their configurations are most effective.

Another factor that influences effectiveness is how search keywords are extracted from the API specifications. Maleshkova et al. [8] extract search keywords from parameter names, Oldham et al. [6] from type names, and Cremashi et al. [10] from textual parameter descriptions. For example, in Figure 1.4 search keywords are extracted from the parameter name, i.e., *destination*. Searching the global ontology schema.org for the word *destination* yields the semantic type *TouristDestination* which are both irrelevant in this context and therefore false positives. On the other hand, the concept *Airport*

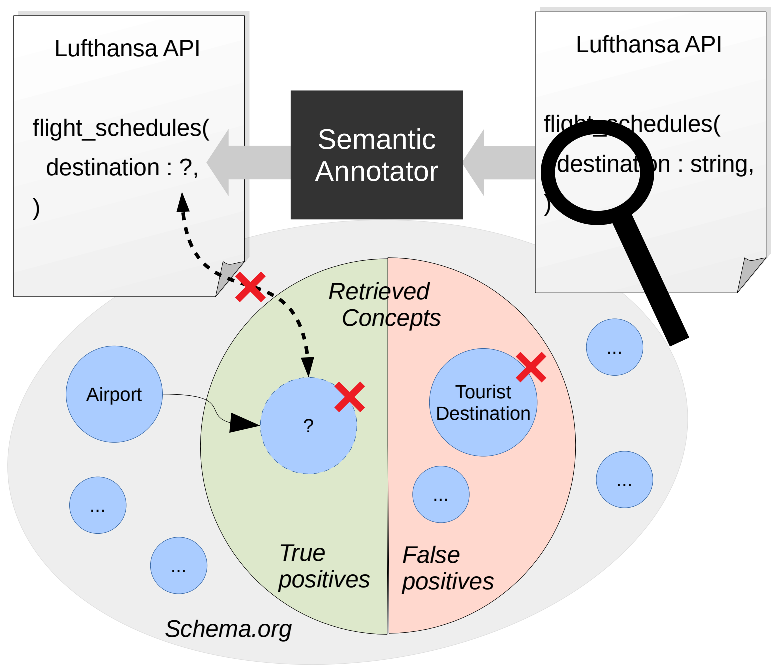


Figure 1.4: Terminological heterogeneity causing ineffective search

is a false negative, which is relevant but not found as its name does not contain the word "destination".

Instead of using just the parameter names for the search, other contextual information of the parameter can be included in the search keywords, e.g., the parameter description, the enclosing operation name, etc. For example, the parameter *destination* belongs to the operation named *flight_schedules* and has the textual description "3-letter IATA airport code" where additional search keywords can be extracted from. Including these search keywords now yields the semantic type *Airport* because "airport" is contained in the textual description of the parameter *destination*. On the other hand, considering search keywords from the operation name also yields the irrelevant semantic types *Schedule*, *Flight*, etc. Thus, adding contextual information to the search keywords can improve recall at the expense of degrading precision.

In this dissertation, the effectiveness of the *Semantic Annotator* is evaluated on a large set of real-world API specifications from RapidAPI that are matched with concepts from the ontology schema.org. It is also shown that the effectiveness of the service discovery system OWLS-MX3 [20] is improved by 61% in terms of mean average precision when API parameters are linked to ontological concepts.

(2) Protocol Miner In my dissertation, I propose to discover API protocol specifications from call-logs using process mining techniques. Service providers use the semi-automatic *Protocol Miner* to derive API protocol specifications from call-logs using process mining. Process mining is a method to extract behavioral models by observing the interaction with a system.

Thousands of REST APIs that are available today are already in use by existing applications. These applications know the API protocol, but this knowledge is not available for other service requesters. Every time such an application calls an operation of an API, it leaves traces in a call-log. Call-logs can be obtained by recording network traffic between applications and the API.

I propose to use process mining techniques to discover API protocol specifications from call-logs. In practice, call-logs are noisy, because, e.g., API calls are made from multiple interfering process instances that happen virtually at random. Noise causes mined protocols to be inaccurate. Inaccurate protocols would falsely suggest future requesters to make operation calls in a wrong order such that the mashup runs into an error state and becomes dysfunctional. Figure 1.5 shows an erroneous protocol mined from a call-log with two interleaving process instances. In this dissertation, I am proposing methods for noise reduction in call-logs to retrieve accurate protocols.

API protocols can have complex control flow with loops, branches, sequences, etc. On the other hand, existing process mining algorithms are limited in which control flows they are capable to discover. In my dissertation, I oppose the control constructs that are generally needed to describe API protocols and the control constructs that can be generally discovered with different process mining algorithms.

(3) Parameter Matcher and Glue Code Generator In this dissertation, I propose a semi-automatic approach for parameter mapping and to generate executable glue code from parameter mappings to establish data communication between incompatible APIs. The *Glue Code Generator* is used by requesters to find corresponding parameters across different APIs and to generate the glue code that establishes API compatibility.

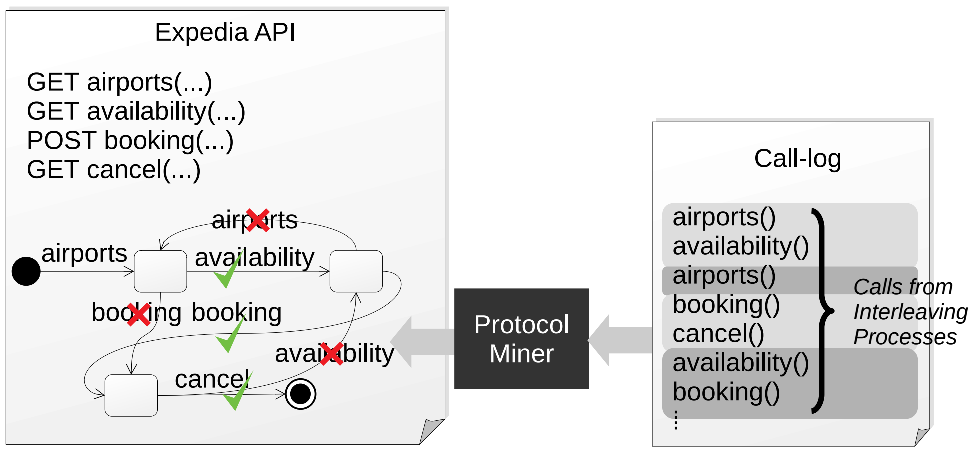


Figure 1.5: Noisy call-logs causing mined protocols to be inaccurate

This approach is shown in Figure 1.6 and works as follows: The parameters are matched pairwise and a similarity score is calculated for every pair of parameters. This similarity score is an aggregation of different metrics for the name, data type, and ontological similarity. The ontological similarity is determined with the help of the ontology links produced by the *Semantic Annotator*: The similarity of two parameters is determined by the ontological distance of the concepts with which the parameters are annotated. The ontological distance between two concepts is the number of concepts that lie between the two concepts in the ontological structure. The closer these concepts are located in the ontology, the higher the ontological similarity. Guided by these similarity scores, requesters can work faster to identify related parameters that have to be exchanged between APIs.

Until now, a decision was made *which* parameters have to be exchanged, but not *how* they have to be exchanged. This is done in the code generation step: The glue code is generated from the set of parameter mappings where the first parameter is translated into the other. The code generator inserts program instructions to invoke the API operations, to extract critical input and output parameters from request and response messages, and to predict and insert a pre-built translation function. A heuristic based on the data types and formats of the involved parameters is used to select translation functions.

The challenge with the code generation is to insert appropriate translation functions. The translation function must not only ensure syntactic type com-

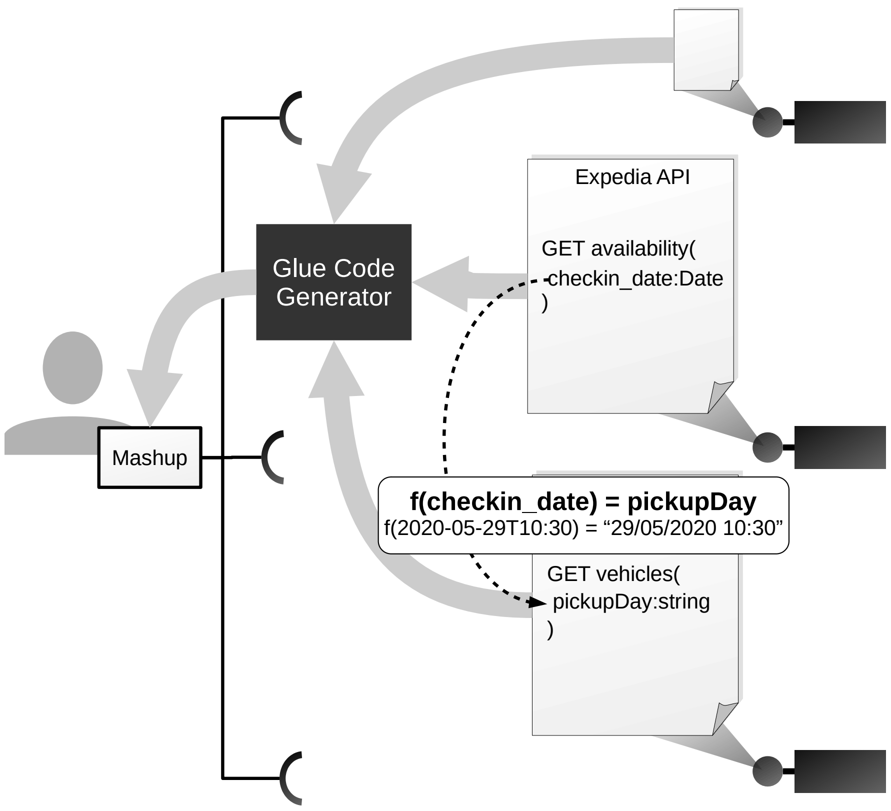


Figure 1.6: Glue code containing translation function to translate from parameter *checkin_date* to *pickupDay*

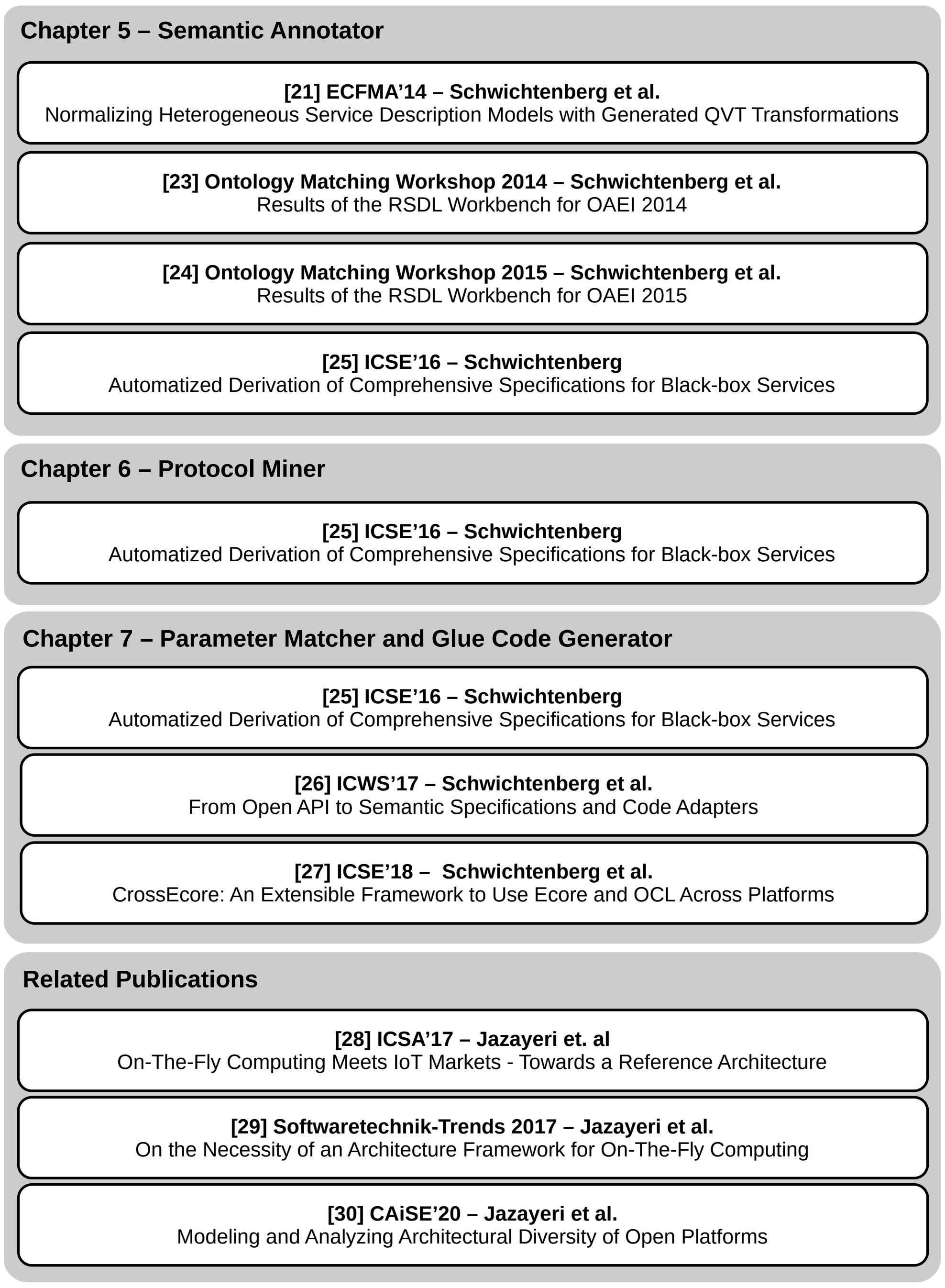
patibility but also correctly recognize and convert data formats if necessary. A predicted translation function may be inaccurate. In case a wrong translation function is automatically inserted, the requester has to replace it manually with a correct translation function, which makes the approach semi-automatic.

Figure 1.6 shows how the *Glue Code Generator* generates the program logic from the parameter mapping *checkin_date* → *pickupDay* and the translation function f that translates values of *checkin_date* into values of *pickupDay.*

The building blocks of the approach presented in this thesis are also part of my scientific publications, as explained in more detail in the following section.

1.5 Publication Overview

This section explains the relationships between my scientific publications and my dissertation. Figure 1.7 shows a list of my publications grouped by the



Chapter 5 – Semantic Annotator

[21] ECFMA'14 – Schwichtenberg et al.
Normalizing Heterogeneous Service Description Models with Generated QVT Transformations

[23] Ontology Matching Workshop 2014 – Schwichtenberg et al.
Results of the RSDL Workbench for OAEI 2014

[24] Ontology Matching Workshop 2015 – Schwichtenberg et al.
Results of the RSDL Workbench for OAEI 2015

[25] ICSE'16 – Schwichtenberg
Automatized Derivation of Comprehensive Specifications for Black-box Services

Chapter 6 – Protocol Miner

[25] ICSE'16 – Schwichtenberg
Automatized Derivation of Comprehensive Specifications for Black-box Services

Chapter 7 – Parameter Matcher and Glue Code Generator

[25] ICSE'16 – Schwichtenberg
Automatized Derivation of Comprehensive Specifications for Black-box Services

[26] ICWS'17 – Schwichtenberg et al.
From Open API to Semantic Specifications and Code Adapters

[27] ICSE'18 – Schwichtenberg et al.
CrossEcore: An Extensible Framework to Use Ecore and OCL Across Platforms

Related Publications

[28] ICSA'17 – Jazayeri et. al
On-The-Fly Computing Meets IoT Markets - Towards a Reference Architecture

[29] Softwaretechnik-Trends 2017 – Jazayeri et al.
On the Necessity of an Architecture Framework for On-The-Fly Computing

[30] CAiSE'20 – Jazayeri et al.
Modeling and Analyzing Architectural Diversity of Open Platforms

Figure 1.7: Publication overview

problem they address.

In [21], an approach to normalize heterogeneous service requests and API specifications written in the Rich Service Description Language (RSDL) [5] which bases on the Unified Modeling Language (UML) is presented. An RSDL specification consists of (1) a UML class model, describing the local ontology of the service's business objects, (2) operation signatures, (3) a UML sequence model describing the service protocol, (4) and the pre- and postconditions of the operations in the shape of visual contracts [22], i.e., transformation rules over UML object models. In [21], the local ontologies from requests and API specifications are aligned using ontology matching techniques and visual contracts are retyped over a common class model which is a preliminary step for service discovery after Huma [5]. A disadvantage of this approach is that it does not scale for a large number of services because the reconciliation needs to be done with all pairs of requests and API specifications (cf. [5]). In addition, the publication [21] bases on the assumption that the ontological and behavioral semantics of the web services are fully specified (in RSDL). This is in contrast to the more realistic assumption made in this dissertation that API specifications and requests are not comprehensively specified.

The works [23, 24], present the results of the participation in the campaigns of the Ontology Alignment Evaluation Initiative (OAEI) in the years 2014 and 2015. The goal of the OAEI is to assess the strengths and weaknesses of ontology matchers. The OAEI campaign consists of several tracks with various ontology matching tasks. This dissertation addresses a very specific ontology matching task that is not covered by the OAEI, namely to align local data models of APIs with an ontology. Data models are also a kind of ontology and therefore ontology matching techniques are highly related.

In [25], an early version of the BaaS vision that includes an approach for a semi-automatized derivation of fully-fledged comprehensive service specifications is presented. Comprehensive service specifications include descriptions of the ontological and behavioral semantics. The approach includes automatic learning of ad-hoc local ontologies from input/output parameter specifications. The classes and class properties (attributes) contained in the ad-hoc ontologies are linked to a global ontology using ontology matching techniques. The idea is to eliminate the terminological heterogeneity between service providers and requesters by linking all local ad-hoc ontologies to the global ontology. In this dissertation, I abandon the approach to learn ad-hoc ontologies from API specifications for the following reason: Output parameters of REST APIs are organized in a tree structure, e.g., in a JSON format. In practice, the way how the information is structured in a tree structure does not reflect ontological knowledge which is usually organized as a graph. This makes it difficult to obtain meaningful ad-hoc ontologies. In [25], it is also proposed to discover API protocol specifications from call-logs to capture the behavioral semantics of an API. In this dissertation, I further pursue this idea and evaluate if it is feasible to mine protocol specifications from call-logs based on a real-world example. My experiences from this investigation allow me to pinpoint the limitations of this approach.

In [26], it is proposed to use a global ontology as a global data model to establish interoperability of various incompatible third-party RESTful web services so that they can be used in a mashup. Adapter code is generated from the correspondences of local and global data types. This adapter code translates the data between the global and local data model and vice versa: API consumers can pass data conforming to the global data model to the adapter, the adapter converts it to the corresponding local data types, passes it to the original REST API, translates its response back so that it conforms to the global data model, and returns the result to the consumer. However, this approach is invasive, because it requires adapter APIs and has therefore limited upward compatibility with the existing corpus of REST APIs. In contrast, the approach proposed in this dissertation is not invasive: The generated glue code directly translates between the APIs without the need for adapter APIs.

In [27], the modeling framework *CrossEcore* is presented. *CrossEcore* includes a code generator that allows generating advanced C#, TypeScript (JavaScript), Swift, and Java source code from ontologies. With *CrossEcore*, it is also possible to generate API specifications from ontologies. From these generated API specifications, in turn, fully-functional RESTful web services can be generated. Using *CrossEcore*, a forward-engineering variant of BaaS can be implemented where new APIs are built around ontologies. The disadvantage of this forward-engineering approach is that it is invasive and upward-incompatible as existing web services need to be rebuilt using *CrossEcore*. In this dissertation, I follow a reverse-engineering approach that builds on the corpus of APIs that exist today.

In our collaborative works [28, 29, 30], we propose a reference architecture for designing open platforms. An open platform has public APIs which third-party developers use to develop services on top of the open platform. In future work, the BaaS architecture needs to be analyzed with the aid of the reference architecture to detect architectural deficiencies. BaaS platforms *are* a kind of open platforms in that sense that service providers can publish their API specifications and requesters can search them.

1.6 Thesis Structure

This thesis is structured as follows: Chapter 2 discusses the foundations of this dissertation. Related works addressing the three problems (1) request–API terminological heterogeneity, (2) unspecified API protocols, and (3) API incompatibility are presented in Chapter 3. A solution overview is given in Chapter 4. This includes a description of the Brokerage as a Service architecture as well as a description of how service requesters and service providers are working with BaaS to create mashups efficiently. Chapter 5 introduces a method to link API specifications with concepts from a global ontology. Furthermore, it is shown that these ontology links help to find relevant APIs more effectively. Chapter 6 presents a semi-automatic approach to derive API protocol specifications from call-logs using process mining. Chapter 7 introduces an approach for semi-automatic parameter matching. From critical parameter mappings, glue code is generated which translates the data from one API to another. Chapter 8 concludes this dissertation and presents prospects for future work.

2 Foundations

This chapter presents the foundations for the BaaS approach. Figure 2.1 maps the individual sections of this chapter to the relevant BaaS components and artifacts. The REST architecture is presented in Section 2.1 To ensure the highest possible pward compatibility of the BaaS approach, REST is chosen as the basic technology. This chapter introduces today's dominating architecture for web services: REST [2]. One basic prerequisite for mashup creation of REST APIs is that API descriptions are available because the services themselves are black-boxes.

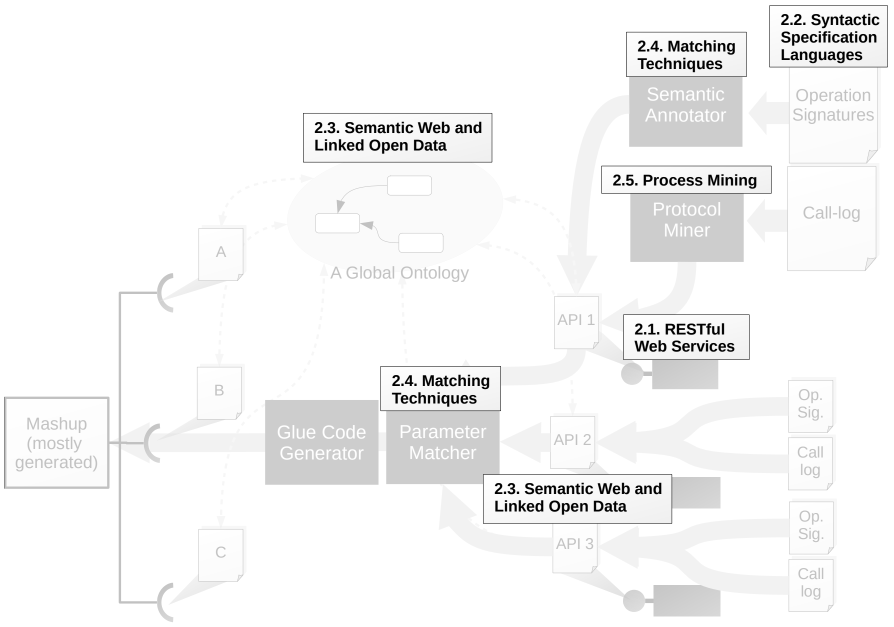


Figure 2.1: Overview foundations chapter

In this chapter, syntactic specification languages are discussed in Section 2.2 and their main elements necessary for mashup creation are identified. Creating mashups solely based on syntactic specifications tends to be inefficient, because of the syntactical and terminological heterogeneity of APIs.

This is the reason why this chapter presents the foundations of the Semantic Web and Open Linked Data in Section 2.3 which address these issues and aim to automate service discovery, service composition, and invocation. These tasks require semantic service specifications that describe the ontological and behavioral semantics (API protocols) of a service. Still, semantic and syntactic specifications are equally important for creating mashups: Semantic specifications for understanding the meaning of an API and syntactic specifications for the technical implementation via APIs.

The service grounding, which is explained in more detail in this chapter, connects the semantic and syntactic specifications and bridges the abstract service description and the concrete technical service realization. APIs are built upon local data models which is why they tend to be incompatible with each other. The BaaS approach suggests unifying the data models semi-automatically by mapping them to a global ontology.

State-of-the-art matching techniques to establish such a mapping are presented in Section 2.4. Later in this work, these techniques are evaluated for and used by the BaaS approach to establish connections between simple terms from syntactic service specifications and ontological concepts. The section is also relevant for the Parameter Matcher as it uses some of the matching techniques.

Equally important as an ontological description of the data model are API protocols for creating mashups. The BaaS approach employs process mining to derive API protocol semi-automatically. Section 2.5 introduces three main process mining algorithms that are later evaluated for and used by the BaaS approach to derive API protocols from API call-logs.

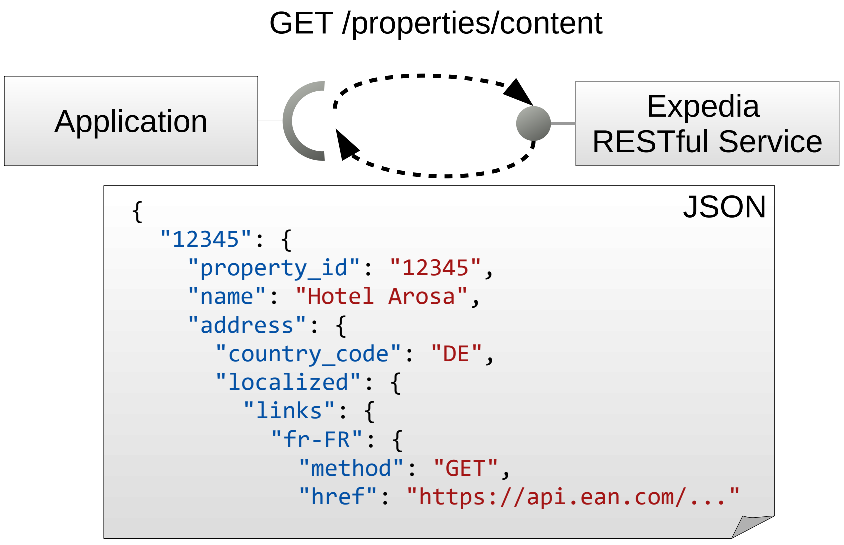
2.1 RESTful Web Services

REST [2] is an architectural style for IT services that is based on the classic client-server protocol. REST enjoys great popularity and has established itself as the de-facto standard: The state of API report 2023 [31] certifies REST an adoption rate of 91% among over 1100 manual testers, automation engineers, developers, consultants, QA managers, and analysts surveyed worldwide. Thus, the predominant position of REST has to be taken into account to meet the requirement of upward compatibility.

RESTful web services are realized based on the HTTP client-server architecture [32]. The REST architectural style comprises the following fundamental principles: Every resource can be uniquely addressed by a Uniform Resource Identifier (URI) [33]. Services are stateless, i.e., the messages contain all the data that is necessary to process the message. Neither the server nor the client stores state information between two messages. Such stateless communication is useful for load balancing which distributes client queries over several machines. Resources are self-explanatory, such that they contain enough information to describe how to process a message. In addition, resources have a uniform interface and a set of standard methods to facilitate their usage. In the context of web services, these standard HTTP methods are GET, POST, PUT, DELETE, etc. A resource may have different representations. For example, a resource can be represented in different formats like HTML [34], JSON [35], XML, etc.

Figure 2.2 shows an application calling the Expedia service through its REST API to get a list of accommodations. In the context of the Expedia API, accommodations are named *properties* as an inn, vacation home, etc. that is owned by a private person or a business. The API call consists of the HTTP GET method and the URI `https://api.ean.com/2.4/properties/content` where the Expedia service is hosted. The API responds with a JSON message that contains the accommodation details, e.g. the name of accommodation, the address, the amenities, etc.

In this dissertation, the term *method* is reserved for HTTP standard methods GET, POST, PUT, etc. as defined in the Hypertext Transfer Protocol (HTTP) [32]. An *operation* is a custom API operation defined by an HTTP method applied on a specific resource. Operations may have interdependencies with each other so that they have to be called in a certain order. The set



```
{
  "12345": {
    "property_id": "12345",
    "name": "Hotel Arosa",
    "address": {
      "country_code": "DE",
      "localized": {
        "links": {
          "fr-FR": {
            "method": "GET",
            "href": "https://api.ean.com/..."
```

Figure 2.2: Exemplary Expedia REST API call

of all valid operation call sequences is the API protocol. REST APIs include a mechanism that guides consumers of that API to obey its protocol. This mechanism is explained in the following section.

2.1.1 HATEOAS

Hypermedia as the Engine of Application State (HATEOAS) is an architectural component of REST: When a client application sends a request to a HATEOAS compliant REST API, this API replies with a message that contains all valid hyperlinks that lead to a subsequent valid call of the API. Client applications dynamically evaluate these links and choose one of them. The advantage of HATEOAS is that applications do not need to know how to interact with the API, but only need to know the entry points of an API, i.e., the starting URIs from which all possible call sequences can be reached. This ensures that the API protocol is followed. Because the links are evaluated at runtime and always on an ad-hoc basis, the client applications remain functional even if changes are made to the API protocol. If a protocol was hard-coded into a client application, the application can get into an invalid state if the API protocol changes.

HATEOAS is a runtime concept that is relevant in the execution of the service. For the creation of mashups, design-time concepts are needed to get the complete API protocol. If all REST APIs would implement HATEOAS, it could be feasible to automatically traverse all links of an API to discover its complete protocol. Unfortunately, the majority of REST APIs do not implement HATEOAS as explained in the following section.

2.1.2 Maturity Model

Strictly speaking, many APIs that are called REST APIs are not REST APIs at all, as they do not implement all of the properties of REST. Richardson defines a maturity model for RESTful web services that expresses what aspects of REST a web service implements [36]. The maturity model defines three levels: Services that do not fulfill any characteristics of REST (level 0), those that use different URIs for different resources (level 1), those that additionally use standard operations (HTTP verbs) for a certain purpose (level 2), and those that support HATEOAS (level 3).

To give an example, the maturity level of the Expedia API is analyzed in the following. The API defines multiple resources, each available under its own URI (level 1): The consumer must query the URI `https://api.ean.com/2.4/properties/content` to get a list of accommodations, `https://api.ean.com/2.4/itineraries` to get a list of bookings, etc. Operations on the same resource are differentiated by using the respective HTTP verbs (level 2). For better readability, the base path `https://api.ean.com/2.4` of the URIs is omitted in the following. To read the itinerary 123, the request `GET /itineraries/123` must be made and to delete it, `DELETE /itineraries/123` must be called accordingly. Figure 2.2 shows an excerpt of the JSON response message of `GET /itineraries/123` which contains information about available subsequent operation calls (level 3). In the example, the subsequent operations are to cancel or change the booking.

Unlike the Expedia API, many REST APIs do not implement HATEOAS and are therefore on level 2 at most. This dissertation addresses REST APIs that are on maturity level 2 and of which no information about the API protocol is available at all.

2.2 Syntactic Specification Languages

Typically, RESTful web services are black-boxes, which means that their source code is not publicly available. Thus, service requesters rely on a specification of the services they want to use. A syntactic specification describes the syntactic characteristics of a service. The essential elements that can be found in any kind of syntactic specification are:

1. Operation names,
2. Inputs of operations, i.e., the data an operation consumes,
3. Outputs of operations, i.e., the data an operation produces,
4. Types of in- and outputs, i.e., the data schema that prescribes all valid inputs and outputs.

The Web Service Description Language (WSDL) [37] is common to describe web services following the Remote Procedure Call (RPC) architectural style. Since version 2.0, WSDL was extended to support RESTful Web Services [2]. Another language to describe RESTful web services is the Web Application Description Language (WADL) [38]. However, these specification languages have rarely been adopted in industry for the specification of RESTful Services [8]. Instead, RESTful web services are often described by custom-made HTML pages that do not have a uniform structure. The hRESTS microformat [39] has been proposed to annotate parts of such custom-made HTML pages to markup certain syntactic elements of a web service.

Nowadays, the rising popularity of RESTful web services comes hand in hand with an increasing popularity of RESTful API specification languages and frameworks like RESTful API Modeling Language (RAML)[1], API Blueprint[2], OData[3], OpenAPI (Swagger)[4], etc.

The remainder of this section focuses on OpenAPI for illustration purposes. However, the ideas presented in this dissertation are not restricted to a specific specification language. Since all of the aforementioned syntactic specification

[1] `https://raml.org/`
[2] `https://apiblueprint.org/`
[3] `https://www.odata.org/`
[4] `https://www.openapis.org/`

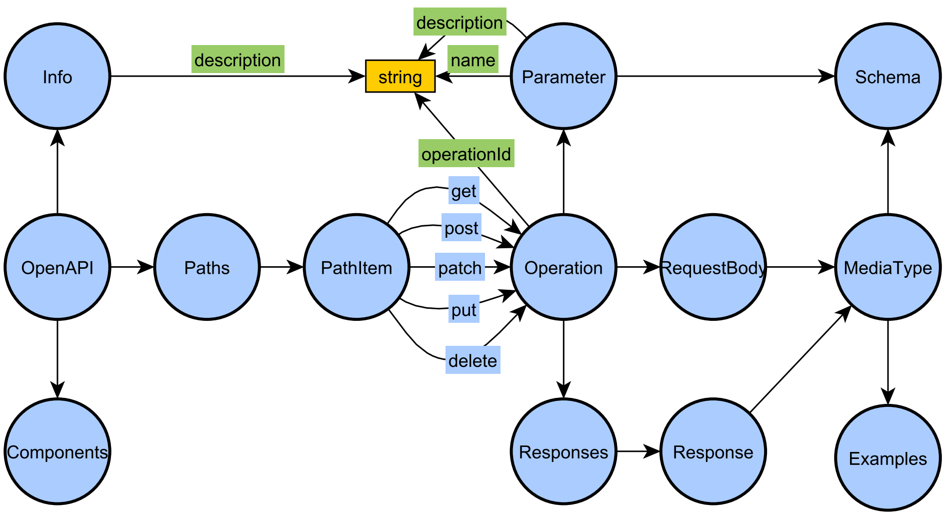


Figure 2.3: OpenAPI ontology (excerpt)

languages share the same essential parts, the approach described in this dissertation can be adopted for other syntactic specification languages.

OpenAPI is a framework for the development of RESTful web services and consists of a language specification [40] and software tools like specification editors and generators for documentation and code. Figure 2.3 shows an excerpt of the OpenAPI schema represented in VOWL.

The root of an OpenAPI specification is a `oas:OpenAPI` object. The `oas:OpenAPI` object declares all available `oas:Path` URLs of the REST interface. An `oas:PathItem` is a relative path to an individual endpoint. HTTP methods like GET, POST, PUT, DELETE, etc. are defined on every `oas:PathItem`. An `oas:Operation` has an `oas:operationId`. The inputs of an `oas:Operation` are `oas:Parameters`. Primitive types and complex types of a parameter are declared in `oas:Schema`. The outputs of an `oas:Operation` are `oas:Responses`. A single `oas:Response` is associated with `oas:Examples` and an `oas:Schema`.

Code generators for server-side and client-side code are available in many programming languages like Java, JavaScript, PHP, etc. The generators help service providers to keep their documentation consistent with their service implementation. Service requesters use the client code generator that creates an API in the programming language of their application which eases the integration of the web service into their application. For example, a generated

```yaml
swagger: "2.0"
host: api.ean.com
basePath: /2.4
paths:
  /properties/content:
    get:
      summary: "Property Content"
      description: |
        Search property content for active properties in the
         ↪  requested language.
      parameters:
      - in: query
        name: property_id
        description: |
          The ID of the property you want to search for.
        required: false
        type: array
        collectionFormat: multi
        items:
          type: string
      responses:
        200:
          schema:
            type: object
            description: An individual property object in the
             ↪  map of property objects.
            properties:
              property_id:
                type: string
                description: Unique Expedia property ID.
              name:
                type: string
                description: Property name.
```

Listing 1: OpenAPI specification of the Expedia API (excerpt)

Java client code encapsulates the web API as a Java API which can be used within a Java application. The generated Java API facilitates common tasks like the construction of HTTP requests and the (de-serialization) of JSON responses to Java objects and vice versa. Since version 3.0.0, OpenAPI includes a language construct, i.e., `oas:Links`, for the specification of HATEOAS.

Listing 1 is an excerpt of the Expedia OpenAPI specification[5]. The specification defines the operation `/properties/content` (line 5) which supports the HTTP method GET (line 6). The operation has an input parameter `property_id` (line 11-19). In the success case, the operation responds with an HTTP status code 200 (line 21) and returns a JSON message defined by a JSON schema [41] (line 22-31).

Searching for relevant APIs based on their purely syntactic specifications tends to be ineffective such that relevant APIs are not found by the service discovery. To overcome this problem, the Semantic Web has been introduced, which is explained in the following section.

2.3 Semantic Web and Linked Open Data

The Semantic Web was introduced by Tim Berners Lee et al. [42] and its core idea is to link data to a machine-readable description of their meaning. The aim is to make the data exchangeable between different web services without these systems having to make special agreements. The basic principles are to describe the meaning of the data in an unambiguous way, to connect the data, and to provide a set of rules to reason about the data. The formalism that is used to describe the meaning of data in the Semantic Web is ontologies. An ontology describes the entities and their relations of a certain domain of interest. Entities and relations are designated by a Unique Resource Identifier (URI) [33] that is globally unique. The entities and relations shape a graph of interconnected data. This graph is called Giant Global Graph or Linked Open Data. Any particular edge in the Linked Open Data graph is a subject-predicate-object triple of concept: The *subject* is the source concept where the edge starts. The *predicate* is a relation that describes the kind of the edge. The *object* is either the target concept where the edge ends or a literal, e.g. a string or a number. Ontologies play a major role in this dissertation as they are the formalism that is later used to capture and resolve heterogeneity between service offers and requests.

[5] `https://cdn.expediapartnersolutions.com/ean-rapid-site/documentation/rapid_2.4/swagger.yaml`

2.3.1 Ontologies

An ontology is a formal specification of a conceptualization and is used to represent knowledge of a particular domain of interest and serve the purpose to structure and exchange information. In computer science, ontologies occur in the shape of folksonomies, taxonomies, data schemas, etc. All these kinds of ontologies have different levels of expressiveness. A folksonomy is an unstructured collection of keywords. A taxonomy is a classification of terms in a hierarchy, which is a tree-like structure where the root is usually the most general term and the most specialized terms are the leaves. An ontology is defined as follows (cf. [43]):

An *Ontology* is a tuple $o = \langle C, I, R, T, V, \leq, \in, = \rangle$ where
C is the set of classes;
I is the set of individuals;
R is the set of relations;
T is the set of datatypes;
V is the set of values;
(C, I, R, T, V) being pairwise disjoint;
$\leq$ is a relation on $(C \times C) \cup (R \times R) \cup (T \times T)$ called specialization;
$\in$ is a relation over $(I \times C) \cup (V \times T)$ called instantiation;
$=$ is a relation over $I \times R \times (I \cup V)$ called assignment;

In a wider sense, a concept is an abstract idea in the human mind. In the context of this dissertation, a *concept* is defined as an ontological description of an entity (C), a relation (R), or an individual (I) designated by a globally unique URI. Concepts of the same kind can be grouped into classes. Members of a class are individuals (or instances). Classes have relations to each other. An individual can either be related to another individual or a literal value. Concrete literal values must comply with a data type, i.e. String or Integer. Specialization defines a type hierarchy of classes and relations. Classes may specialize multiple other superclasses, and relations multiple other relations.

Ontologies consist of the TBox ($C \cup T$) and the ABox ($I \cup V$). The TBox is like a meta-model that defines the type level, i.e., classes and relations. The ABox contains so-called *individuals*, i.e., concrete class instances. Common languages that can be used to describe ontologies are RDF Schema[44] (RDFS) and the Web Ontology Language (OWL) [45].

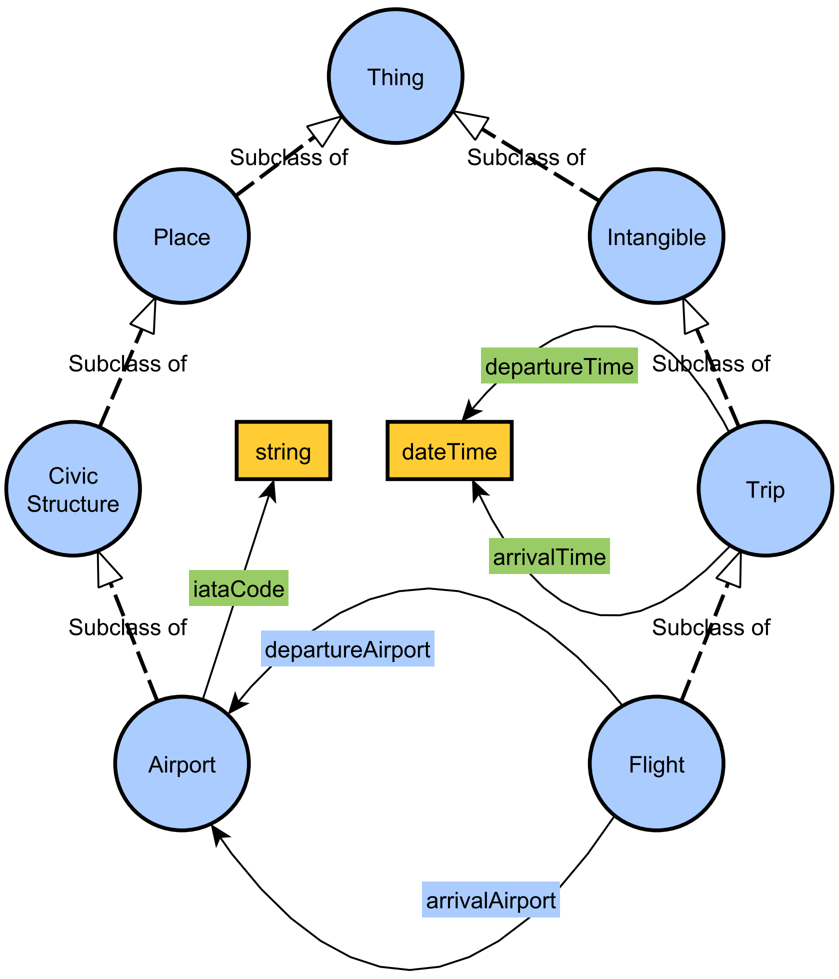


Figure 2.4: A domain ontology based on Schema.org

OWL distinguishes relations (R) into `DatatypeProperties` and `ObjectProperties`. While the latter describes relations between instances of classes the former describes relations between instances and values. Properties have a `Domain` that restricts the classes of subjects that may occur in a triple. Likewise, `Range` restricts the classes of objects that may occur in a triple. The range of ObjectProperties are classes (C) and the domain of DatatypeProperties are values (V).

VOWL is a visual notation for OWL Ontologies (VOWL) [46]. Figure 2.4 shows an ontology based on schema.org In VOWL, classes are denoted in light blue circles. An `Airport` can be related to a

`Flight` through the `ObjectProperty departureAirport`. Thus, the domain of `departureAirport` is `Flight` and its range is the class `Airport`. `ObjectProperties` have light blue labels in VOWL. `Flight` is a subclass of `Trip` and inherits its `DatatypeProperty departureTime`. `DatatypeProperties` have green labels in VOWL. Classes and properties from external ontologies are shown in dark blue.

2.3.2 Semantic Web Services

Semantic Web Services are web services that are enriched with machine-readable semantics in addition to their pure syntactic specification. A *service description ontology* is a meta-model for describing the concept of a web service itself. A *service model* describes a concrete web service instance, e.g., the British Airways API, by using the entities and relations from the service description ontology. Thus, a service model is the ABox that conforms to the TBox of the service description ontology. The input and output types used in a service model are usually defined in a separate *domain ontology.* In the case of the British Airways API, such a domain ontology contains entities like Airports, Aeroplanes, etc.

There are several approaches that all come with different service description ontologies to describe Semantic Web Services, e.g., Web Ontology Language for Web Services (OWL-S) [47], WSML [48], SA-REST [49], WSMO-Lite [50], Hydra [51], etc. [52] provides an overview of semantic service specification languages. Furthermore, there are extensions of OpenAPI that allow linking OpenAPI specifications with ontologies [52, 53]. In OWL-S, not only can the ontological semantics of a web service be described, but also behavioral semantics, especially the API protocol. Since the description of API protocols is an essential part of this dissertation, OWL-S is used consistently in the following examples.

Figure 2.5 shows an excerpt of the OWL-S service description ontology. In OWL-S, operations correspond to `AtomicProcesses`, which have `Inputs` and `Outputs`. Types of `Inputs` and `Outputs` are defined by `parameterType` that points to an URI of an ontological class or a data type. Ontological classes that are used as parameter types are also referred to as *semantic types.*

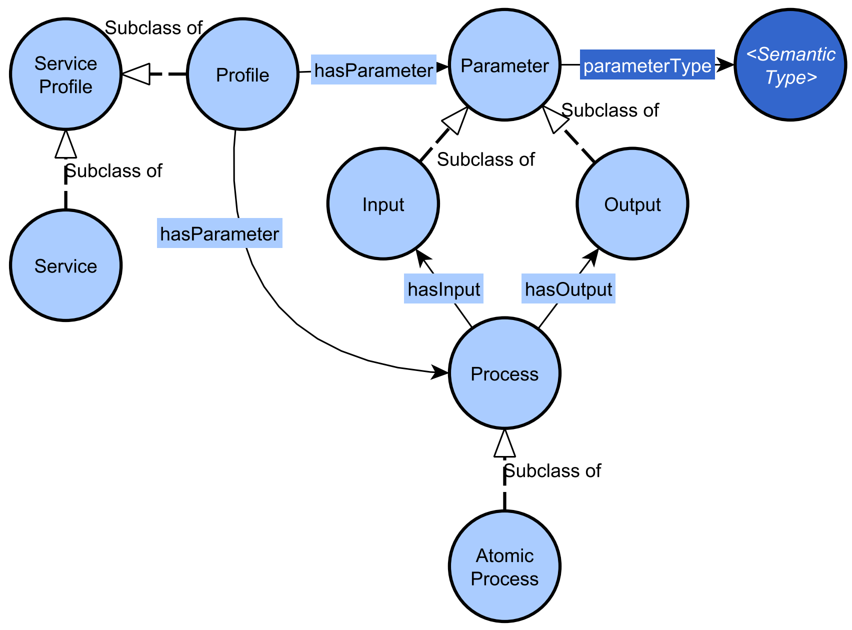


Figure 2.5: OWL-S service description ontology (excerpt)

API protocols are described through `CompositeProcesses`, which describe the control flow between `AtomicProcesses`. OWL-S defines control constructs like `Splits` for concurrent, `Loops` for repeated, `Branches` for the conditional execution, etc.

2.3.3 Semantic Service Discovery

Semantic service discovery is the process of locating web services that provide certain capabilities while adhering to some requester-specific constraints. Three roles participate in service discovery: service requesters, service providers, and a service registry. Service providers specify a service model that describes the capabilities of their services and publish it at the service registry. The service requester creates a service request which specifies a service model of the desired service and queries the service registry by that request.

A *service matchmaker* is a computer program that calculates the compliance of the service request and the service offers and recommends the most compliant service(s) to the requester. Existing service matchmakers usually return a confidence score ranging between 0 and 1, where zero means that request and offer are not similar at all, and 1 means that request and offer are equivalent. When the matchmaker returns its results, the service offers can be ranked by this score so that the requesters see the most relevant offers first. A survey of available semantic service matchmakers can be found in [4, 54]

2.3.4 Service Grounding

Regular web services that are not semantic web services in the first place can be elevated to Semantic Web Services by adding a semantic description retrospectively. That is done by replacing the data types in operation signatures with semantic types. Service matchmaking is done based on these semantically typed operation signatures.

Once the requester has found an appropriate service, it comes to the technical execution of such a web service, its original interfaces, and the original operation signatures must be used. The service grounding bridges the gap between the semantic specification and the technical realization of a service. The service grounding provides information on how to access a service and is a mapping between the abstract service model and the concrete technical realization. Such a service grounding is needed when it comes to the invocation of a service. For every semantic type that occurs as an input or output type in the service model, the grounding specifies how data actually is typed and structured. Like shown in Figure 2.6, service grounding consists of two kinds of transformations: the lifting and the lowering transformation. The lowering transformation converts the semantic types of the inputs into data types. Analogously, the lifting transformation converts output data types back into semantic types.

To give an example, OWL-S has built-in support to ground service models to elements of WSDL specifications. In particular, `AtomicProcesses` correspond to operations defined in the syntactic WSDL specification. The set of `Inputs` and `Outputs` correspond to parts of messages defined in WSDL. Semantic types of the inputs and outputs correspond to abstract types that are defined in an XSD schema [55]. The lifting and lowering transformations are realized as two separate XSLT [56] transformations.

The grounding mechanism of OWL-S is not limited to WSDL. However, the WSDL grounding is currently the only kind of grounding that is specified by the OWL-S standard.

Besides the OWL-S service grounding there are also other grounding ap-

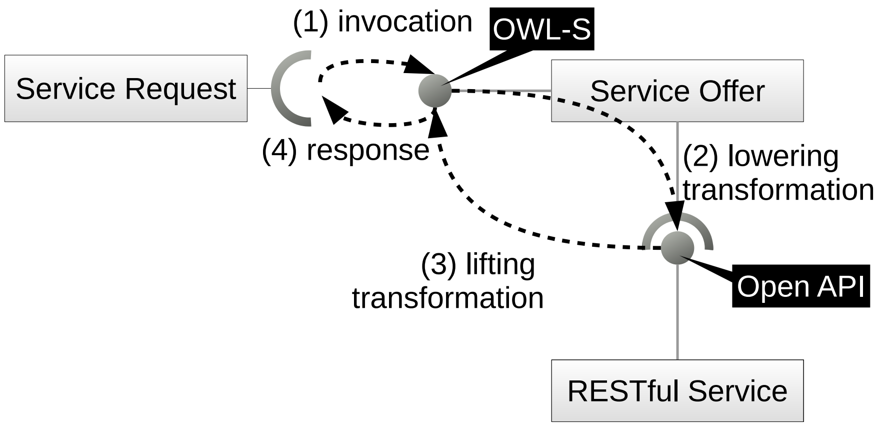


Figure 2.6: Service grounding

proaches: WSDL-S [57] and SAWSDL (Semantic Annotations for WSDL and XML Schema) [58] allow to annotate data types in a WSDL specification with semantic types from a domain ontology. Hydra-enabled web services [51] produce JSON responses that are annotated with links to domain ontologies [59].

2.3.5 Service Composition and Interoperability

It may happen that there is no single service offer that fully satisfies a request completely. In that case, multiple services may need to be combined in a mashup to fully satisfy the request. For example, it may be required to combine a flight booking, hotel reservation, and car rental service to satisfy the request for a travel arrangement service. The process on how to arrange different services so that they satisfy the request is called service composition. In general, third-party services combined in a mashup are not interoperable so they cannot immediately exchange their data, because the data may be structured in a different way and may have different types and formats. Thus, the data that is passed from one service to another has to be translated so that it conforms to the respective APIs.

2.3.6 Ontology Matching

Ontology matching is the process of finding correspondences between different ontologies and assessing their similarity. In this dissertation, ontology matching is relevant from two perspectives: First, to determine the relevance of a service offer for a given service request. The relevance of the offer is determined based on how similar the semantic types are that are used in the operation signatures. Second, to identify those semantic types of different services that are semantically similar, but cannot be exchanged with one another in a mashup because they are syntactically different.

This dissertation adopts the ontology matching terminology from Euzenat et al. [43]. A *correspondence* asserts that a certain relation holds between two concepts. There are different possible kinds of relations between two concepts, i.e. equivalence ($=$), disjointness ($\perp$), generalization ($\sqsupseteq$) relations. The set of all correspondences is an *ontology alignment*. An *ontology mapping* is the directed ontology alignment between a source and a target ontology. Like a mathematical mapping, such a mapping can be total, injective, surjective, or bijective. Accordingly, there can be one-to-one (1:1), one-to-many (1:n), many-to-one (n:1), or many-to-many (n:m) correspondences of concepts.

Ontology matchers are computer programs that perform ontology matching. Several surveys [60, 61, 62, 63, 64, 65, 43, 66] provide an overview of a large number of ontology matchers.

2.3.7 Types of Heterogeneity

In a real-world service discovery scenario, there can be thousands of service requesters and service providers. All of them are independent parties that are not aware of each other and are first acquainted with each other in the service discovery. Service providers and requesters are software developers that have individual interests, knowledge, beliefs, and habits. Based on their personal background they come up with different solution approaches for the same or a similar problem. This also applies to the design of IT services: Services requesters and service providers can describe the same service in completely different ways. This section discusses different types of heterogeneity and adopts the heterogeneity classification of [43] which defines syntactic, terminological, conceptual, and semiotic heterogeneity.

Syntactic Heterogeneity

Syntactic heterogeneity occurs when two parties use different syntaxes to specify a service. In the scope of mashup creation, syntactic heterogeneity occurs on three levels:

1. Two service partners use different specification languages, e.g., when

service requesters describe their requests in OWL-S and service providers describe their API specifications in SAWSDL. How to solve syntactic heterogeneity of specification languages is treated in [67].

2. Two APIs use different serialization formats such as JSON and XML. An API that emits JSON data cannot be directly combined in a mashup with another API that consumes XML data. The number of data exchange formats is limited so it is feasible to create converters that convert between the formats, which needs to be done only once. Often, such converters are already available for popular data exchange formats. For example, the Newtonsoft library[6] is capable to translate between JSON and XML and back. Because standard formats of data can be easily converted, this dissertation does not further discuss this kind of syntactic heterogeneity.

3. Two APIs use different types and formats for single data attributes. For example, a service might represent a concrete date as a string that is formatted according to the RFC 3339 [68] while another service might represent the same date as a Unix timestamp that counts the milliseconds passed starting from January 1st, 1970. This is exactly the kind of syntactic heterogeneity that is addressed in this dissertation.

Terminological Heterogeneity

Two services exhibit terminological heterogeneity when requesters/providers use different terminologies to name their operations, parameters, or types. In the simplest case, service partners might use different naming conventions. Common naming conventions are for example camel case and snake case: With the camel case convention, words are separated by capital letters, e.g., `BoardingPolicyType`. With snake case, words are separated by an underscore, e.g., `boarding_policy_type`.

In more complex cases of terminological heterogeneity, different parties can use synonymous terms to name their entities. Synonyms are words that have the same meaning. On the other hand, there are homonyms, that use the same word, but have different meanings. For example, the word `Plane` may describe a flying vehicle or a mathematical geometry.

[6]`https://www.newtonsoft.com/json/help/html/ConvertingJSONandXML.htm`

Conceptual Heterogeneity

Two ontologies exhibit conceptual heterogeneity when their concepts have a different logical structuring. Different ontologies might cover different domain areas, e.g. tourism, on a different granularity level, and from a different perspective which might be reasons for conceptual heterogeneity [69].

Semiotic Heterogeneity

Semiotic heterogeneity addresses the different ways of interpreting the same concept in different contexts [70]. Concepts that have the exact same meaning can be interpreted differently, depending on the context they are ultimately used in. For example, a flight can be seen as a transport connection between two locations or as a commercial offering of a business organization, i.e., an airline.

To make a service from one domain interoperable with another domain, it must be placed in the context of that domain. Therefore, to solve semiotic heterogeneity is especially crucial to combine services from different domains.

2.4 Matching Techniques

Multiple different matching techniques have been developed in order to resolve different types of heterogeneity. Typical state-of-the-art ontology matchers generate initial candidate correspondences of entities whose similarity is determined in terms of one or more similarity measures. Similarity is defined as follows [43, Chapter 4.1]:

A similarity $\sigma : o \times o \rightarrow \mathbb{R}$ is a function from a pair of entities to a real number expressing the similarity between two objects such that:

$$\begin{aligned} \forall x, y \in o, &\quad \sigma(x,y) \geq 0 && (\textit{positiveness}) \\ \forall x \in o, \forall y, z \in o, &\quad \sigma(x,x) \geq \sigma(y,z) && (\textit{maximality}) \\ \forall x, y \in o, &\quad \sigma(x,y) = \sigma(y,x) && (\textit{symmetry}) \end{aligned}$$

The remaining section describes different kinds of state-of-the-art matching techniques, i.e., string-based, language-based and structured-based techniques.

String-based

Names of concepts are strings and the intuition behind string-based techniques is that similar concepts have similar names. In the simplest case, it can be assumed that two concepts with the same name have equivalent semantics (which might be wrong when the names are homonyms). However, it is very unlikely that two parties use exactly the same names for equivalent concepts. Names appear in different variations, for example, according to different naming conventions.

A common technique to cope with different variations of strings is to normalize them before they are compared. How aggressive the normalization is depends on the fact whether different variations of a string are meaningful for the search result. *Case normalization* converts all alphanumeric characters to their lowercase counterpart. *Diacritics suppressions* removes signs that indicate pronunciation or accentuation of letters and replaces the characters with their standard form. Furthermore, digits and punctuations can be removed from strings. These different methods can be used in conjunction.

More advanced string-based techniques take the sequence of characters into account. Many similarity measures have been proposed that calculate a similarity score of two strings. Among their well-known representatives of string-based techniques belong n-gram, Levenshtein [71], Jaro-Winkler [72] similarity.

Besides string-based techniques, there are also token-based techniques that view strings as sets of words rather than sequences of characters. *Tokenization* is the process of segmenting a string into a set of tokens, disregarding the order and multiplicity of the tokens. A special form is the bag-of-words model that disregards the order but keeps the multiplicity of the tokens. Common token-based techniques are cosine similarity and term frequency-inverse document frequency (TF-IDF).

Term Frequency Inverse Document Frequency is defined as follows:

$$tfidf(t, D) = tf(t, D) \cdot idf(t)$$

$$tf(t, D) = \frac{\#(t, D)}{max_{t' \in D \#(t', D)}}$$

$$idf(t, D) = \log \frac{N}{\sum_{D : t \in D} 1}$$

where $tf(t, D)$ is the term frequency of the term t in document D. The inverse document frequency relates the number of all documents N to the number of documents that contain t.

Because name-based techniques do not take the semantics of the entities into account, their capabilities to resolve terminological heterogeneity is limited. The context of the entities in their ontological structure is also not considered which is why name-based techniques are not suited to resolve conceptual or semiotic heterogeneity.

Language-based

Inflection is the grammatical modification of words regarding tense, case, voice, aspect, person, number, gender, and mood. Language-based techniques like stemming take such phenomena of natural languages into account to normalize words to be matched. Other techniques look up words in dictionaries. Dictionary entries provide additional (contextual) information of words, such as their relations to other words or a glossary description. Such language-based techniques are explained in the following sections.

Stemming Often words occur in morphological variants, like in their grammatically declined or conjugated form. In information retrieval, it is often desired to consider different morphological variants of a string as a match. The different variants complicate the task of finding string matches algorithmically.

Stemming or lemmatization is a technique to eliminate morphological phenomena, i.e., to reduce tokens to their linguistic root form. This technique goes beyond plain string normalization as it takes background knowledge about the grammatical rules of a natural language into account.

There are specialized stemmers for the different natural languages according to the language's specific morphological rules. The English Minimal Stemmer [73] is a plural stemmer for English. The English Possessive Stemmer stems English possessive S's. The Porter Stemmer [74] has a list of shortening rules to reduce words to their stem. KStem [75] removes suffixes from words until the reduced word is found in a dictionary. The different stemmers differ in their stemming degree, i.e., how aggressively they generalize words to a stem. Overstemming is an error where two words have been incorrectly reduced to the same root (false positive). Analogously, understemming is an error where two words have not been reduced to the same root, although they should have been.

Stop word elimination Stop words are the most common words in a language and play a minor role to grasp the meaning of a sentence. Typical stop words of the English language are for example indefinite and definite articles ("a", "the"), conjunctions ("and"), pronouns ("our"), etc. Stop words occur in almost any English sentence but they do not allow to make conclusions about the contents of a sentence. Therefore, stop words are usually ignored as they do not contribute to the relevance of the search results in general.

External resources Other language-based techniques use external resources to measure the similarity of two entities. Such techniques take the semantics of the words and their relations into account. There are different types of external resources: lexicons, thesauri, etc. Lexicons define certain keywords by a natural language description. Thesauri additionally contain the relations between words like synonyms, hypernyms, hyponyms, etc. A *hypernym* is a term that classifies a certain set of other words, i.e. it is more general than the other words. Analogously, a *hyponym* is a term that is more specific than another word.

WordNet [3] is a digital lexical database of the English language. For every word, WordNet stores its part of speech (noun, verb, adjective, adverb), its morphologic root, a brief definition in natural language, optionally one or more short example sentences, and a set of its synonyms. The linguistic relations among the synonym sets are also available in WordNet. WordNet contains relations like: hyperonomy/hyponomy (is-a), meronymy (part-whole), antonymy (opposite).

Structure-based

Contrary to string-based and language-based techniques that determine the similarity of concepts alone by their name, structure-based techniques take the concept's context within the ontological structure into account. The intuition is that similar concepts are located nearby within a given taxonomy. Taxonomies define the class hierarchy in a tree structure. The most general class is at the root of the taxonomy, while the most specific classes are the leaves.

The upward cotopic similarity [76] is a more advanced metric that determines the similarity of two entities on the basis of their shared superclasses towards the root of the taxonomy. The Wu-Palmer similarity [77] also takes into account that classes close to the taxonomy root are more general while classes near the leaves are more specific. Thus, two adjacent concepts near the taxonomy root are less similar than other two adjacent concepts near the leafs. The upward cotopic similarity and the Wu-Palmer similarity can only be used when the corresponding classes are in the same taxonomy/ontology. They are therefore not suitable for Semantic Service Discovery, since it is unlikely that the service requesters and service providers use different ontologies.

2.5 Process Mining

The aim of this dissertation is not only to describe the ontological semantics of APIs but also their behavioral semantics in the form of API protocols. This dissertation examines the extent to which process mining techniques are suitable for semi-automatically deriving API protocols from call-logs in order to reduce the manual effort involved for specifying accurate API protocols.

Process mining is originally a technique for reconstructing business processes from digital traces of IT systems. Van der Aalst gives a comprehensive overview of the research area of process mining in [78]. The reconstructed models can then be analyzed to identify problems in the processes and to improve them. Event logs serve as input for process mining and are recorded by the IT system when users interact with the system. Each event in the event log refers to a particular process instance or *case*, an *activity*, and a *timestamp*. A trace is a sequence of events for a particular process instance. The output of process mining is a behavioral model such as Finite State Machines [79] or Petri Nets [80].

This dissertation interprets call-logs of web services as event logs. A call-log records all HTTP requests sent by clients and processed by the web service. A client session denotes all coherent API calls of a single client to complete a certain task. Thus, a case of an event log corresponds to a client session and an activity of an event log corresponds to a single API call. Like event log, call-logs contain timestamps. Therefore, call-logs contain the basic information that is required for process mining.

2.5.1 Mining Algorithms

This section introduces three important process mining algorithms: Alpha Miner [81], Heuristics Miner [81], and the Inductive Miner [82]. In addition, various quality properties of these algorithms are presented. The different mining algorithms have different quality characteristics. In the later course of this dissertation, these quality properties will be used to identify the algorithm that is best suited to mine API protocols for web services.

Alpha Miner The Alpha Miner [81] reads the event log and creates a list of events that follow one another directly. Two events x and y following one another are in a so-called direct succession relation, denoted as $x > y$. The Alpha Miner includes three rules that are used to derive more specific relations: causality $x \rightarrow y$, parallelism $x||y$, and choice $x\#y$.

$$x \rightarrow y \Leftrightarrow (x > y) \wedge \neg(y > x) \qquad (causality)$$

$$x \,||\, y \Leftrightarrow (x > y) \wedge (y > x) \qquad (parallelism)$$

$$x \,\#\, y \Leftrightarrow \neg(x > y) \wedge \neg(y > x) \qquad (choice)$$

These relations are mapped onto Petri net fragments. All the single fragments combined form the output Petri net. Traces are written in angle brackets. In the remainder, the aforementioned mining algorithm is explained by using the following event log as a running example, which is adapted from [83]. The first trace $\langle a, b, c, d, e, g\rangle^6$ means that events a, b, c, d, e, g occurred in that temporal sequence. This trace occurs six times in the event log.

$$\langle a, b, c, d, e, g\rangle^6$$
$$\langle a, b, c, d, f, g\rangle^{38}$$
$$\langle a, c, d, b, f, g\rangle^2$$
$$\langle a, b, d, c, e, g\rangle^{12}$$
$$\langle a, d, c, b, f, g\rangle^4$$

The *footprint matrix* shown in Table 2.1 captures all relations between events. Table 2.2 shows the three patterns the Alpha Miner applies on the footprint matrix. The resulting Petri net discovered from the example event log is shown in Figure 2.7.

	a	b	c	d	e	f	g
a	#	→	→	→	#	#	#
b	←	#	||	||	#	→	#
c	←	||	#	||	→	#	#
d	←	||	||	#	→	→	#
e	#	#	←	←	#	#	→
f	#	←	#	←	#	#	→
g	#	#	#	#	←	←	#

Table 2.1: Alpha Miner footprint matrix

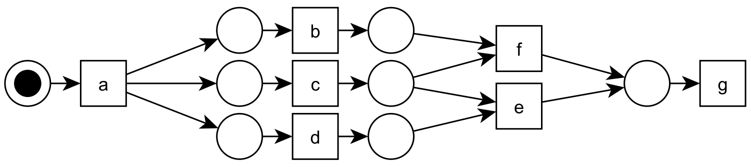


Figure 2.7: Petri net discovered by Alpha Miner

Heuristics Miner Unlike the Alpha Miner, the Heuristics Miner [81] is a process mining algorithm that also takes the frequency of events into account. Real-world event logs often contain noise, e.g. deviations from the normal order of events that only occur with certain exceptions. Process mining is about identifying general processes, where noise is a disruptive factor. The Heuristics Miner is less susceptible to noise. The idea of this algorithm is that if a sequence of two events is noise, then this sequence rarely appears in the traces which is reflected by a lower frequency of that sequences. Sequences

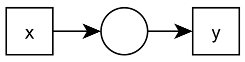
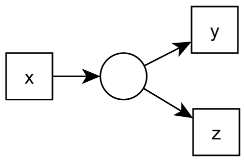
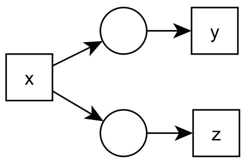
Pattern	Input Relations	Petri net Fragment
Sequence	$x \rightarrow y$	
XOR-split	$x \rightarrow y$, $x \rightarrow z$, $y \# z$	
AND-split	$x \rightarrow y$, $x \rightarrow z$, $y \| z$	

Table 2.2: Alpha Miner patterns

with low frequencies are then filtered out, eliminating noise. The challenge here is that it is fundamentally impossible to distinguish between unwanted noise and truly infrequent event patterns.

The Heuristics Miner works as described in the following: First, a so-called $n \times n$ directly-follows frequency matrix M is created over all n events. Each cell $m_{xy} \in M$ holds a value that counts how often event x is directly followed by event y. Table 2.3 shows the directly-follows frequency matrix created from the example event log that has been introduced in the previous section.

Based on these frequencies, an $n \times n$ dependency matrix D is created which represents the significance of each relation (Table 2.4). With the help of a frequency-based metric, the value of each cell $d_{xy} \in D$ is determined, which indicates how certain a dependency relationship between events x and y is. $x \Rightarrow_W y$ denotes the dependency relation between events x and y which is defined as:

$$x \Rightarrow_W y = \frac{|x >_W y| - |y >_W x|}{|x >_W y| + |y >_W x| + 1}$$

where W is an event log over events T and $x, y, z \in T$ and y and z are preceded by x. $|x > y|$ is the number of times the event x is directly followed by y. A value close to one means that there is certainly a dependency relation.

	a	b	c	d	e	f	g
a		56	2	4			
b			44	12		6	
c		4		46	12		
d		2	4		18	38	
e							18
f							44
g							

Table 2.3: Heuristics Miner: Directly-follows frequency matrix

	a	b	c	d	e	f	g
a		.98	.67	.8			
b	-.98		.82	.67		.86	
c	-.67	-.82		.9	.92		
d	-.8	-.67	-.9		.95	.97	
e			-.92	-.95			.95
f		-.86		-.97			.98
g					-.95	-.98	

Table 2.4: Heuristics Miner: Dependency matrix

Figure 2.8 shows the dependency graph for the dependency matrix. The Heuristics Miner uses the *dependency threshold* and the *relative-to-best threshold* to eliminate insignificant edges from the dependency graph: An edge (x, y) is discarded when d_{xy} is below the *dependency threshold* $\phi \in [0, 1]$. For example, let $\phi := 0.68$. Thus, the edge (b, d) is removed from the dependency graph (Figure 2.8), because $b \Rightarrow_W d = 0.67$.

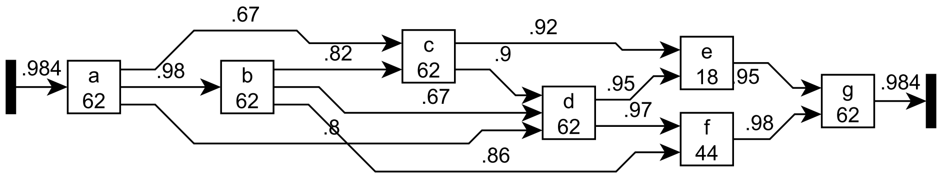


Figure 2.8: Dependency graph produced by Heuristics Miner

More insignificant edges are removed using the *relative-to-best threshold* $\chi \in [0, 1]$: The highest value $\overset{\bullet}{d_x}$ is identified in each row x of the dependency matrix, and all edges where the inequality $d_{xy}(1 - \chi) < \overset{\bullet}{d_x}$ holds are removed from that row. For example, let $\chi := 0.185$. The highest value in the first row is $d_{ab} = 0.98$. The edge (a, c) is removed, because $d_{ac} = 0.67$ is less then $d_{ab}(1 - \chi) = 0.7987$.

While the dependency threshold and the relative-to-best threshold are there to eliminate insignificant edges, there is also the *positive observation threshold* ψ which is used to recognize if two events are in an AND or XOR relation. The underlying idea is that if two events a and b are in an AND relation, then there are traces in the event log where b directly follows a. On the contrary, if a and b are in an XOR relation, there is no trace where b directly follows a. The following formula expresses this idea:

$$x \Rightarrow_W y \wedge z = \frac{|y >_W z| + |z >_W y|}{|x >_W y| + |x >_W z| + 1}$$

If $x \Rightarrow y \wedge z > \psi$ then y and z are in an AND relation, otherwise they are in an XOR relation. The final Petri net is depicted in Figure 2.9[7].

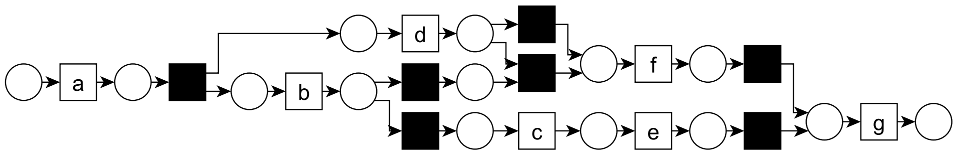


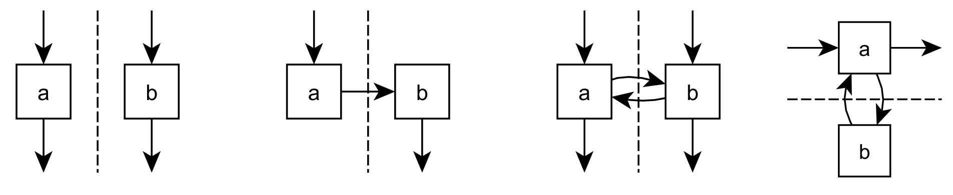
Figure 2.9: Petri net discovered by Heuristics Miner

[7]The black boxes are epsilon transitions that always fire when there is an input token.

Inductive Miner The Inductive Miner [82] is a process mining approach that obeys the divide-and-conquer principle. Divide-and-conquer algorithms strip complex problems down to smaller problems and assemble partial solutions of the easier problems to an overall solution of the original problem. The Inductive Miner firstly creates a directly follows graph, i.e. a directed graph that has a directed edge from event a to event b iff b occurs chronologically right after a. This directly follows graph is searched for a characteristic division of events and cut into disjoint sets.

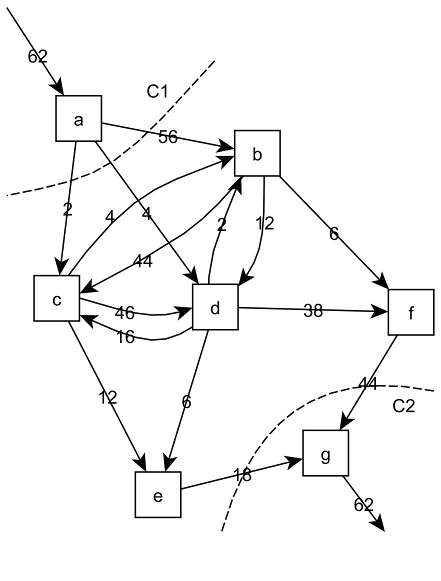
An operator that combines both sets is selected. There are four operators: exclusive or ($\times$), sequential composition ($\rightarrow$), interleaved parallel composition ($\wedge$), and loops ($\circlearrowleft$). Each of the four operators has a characteristic cut of the dependency graph shown in Figure 2.10. The algorithm is recursively rerun on the remaining subgraphs until no more characteristic cuts are possible. While the directly graph is cut into subgraphs, the Inductive Miner incrementally builds an internal *process tree*, where the nodes are operators and the leaves are events.

In the remainder, the algorithm is exercised based on the event log from the running example. The first cut $C1$ separates a from the rest of the graph and is a sequence cut, as it cuts through edges that are pointing in one direction (Figure 2.11a). The Inductive Miner adds the root node with the sequence operator and a as the first leaf is event a, followed by a placeholder representing the remaining graph. Similarly, $C2$ is a sequence cut that separates g (Figure 2.11b). Accordingly, the second leaf g is added below the root node in the process tree. Cut $C3$ is a sequence cut that splits the graph into a

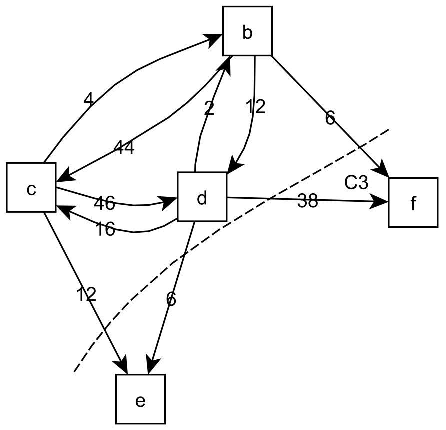


(a) $\times$ Cut (b) $\rightarrow$ Cut (c) $\wedge$ Cut (d) $\circlearrowleft$ Cut

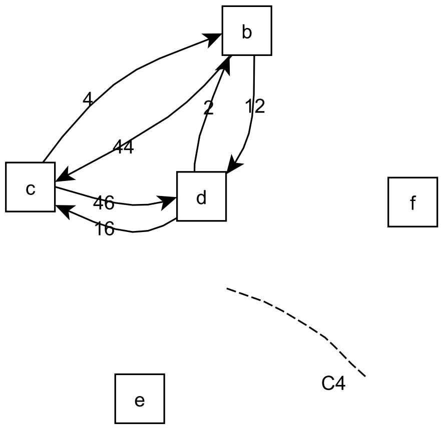
Figure 2.10: Inductive Miner cuts types



(a) → Cut



(b) → Cut



(c) × Cut

Figure 2.11: Inductive Miner cuts

disjoint set $\{f, e\}$ and the remaining graph (Figure 2.11c). The events f and e are not connected anymore, which is the reason why the $\times$ cut is applied. A new node is added below the root node labeled with the $\times$ operator. The leaves f and e are added below the new node. The remaining events b, c, d are all mutually connected in both directions, so that $\wedge$ cuts must be applied respectively. In the process tree, a new node with the $\wedge$ operator is added

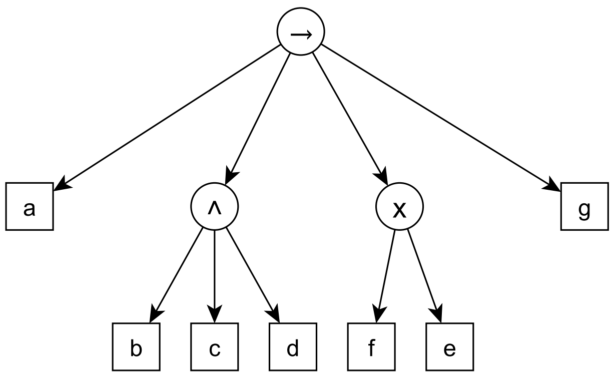


Figure 2.12: Process tree discovered by Inductive Miner

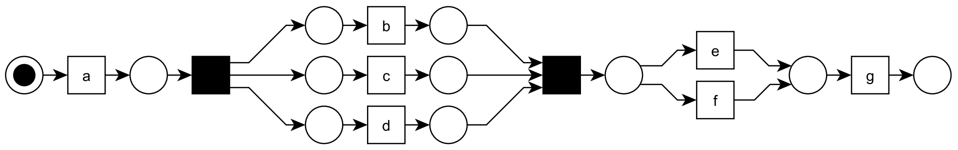


Figure 2.13: Petri net discovered by Inductive Miner

below the root node with the leaves b, c, d. Figure 2.12 shows the resulting process tree. Figure 2.13 shows the corresponding Petri net.

2.5.2 Quality Attributes

In process mining, a behavioral model is extracted from an event log. Ideally, all traces of the event log can be recreated by this behavioral model. In practice, this is not necessarily the case. The *fitness* of a behavioral model states whether a model can generate all traces seen in the event log.

An event log only contains a limited number of sample traces and is a snapshot of the IT system under consideration. It is possible that the event log coincidentally does not contain a certain trace, such that the mined model would not allow that trace. *Generalization* is the property of allowing certain traces even if these were not previously seen in the event log.

On the other hand, a behavioral model that overly generalizes possible traces is useless because no statements can be made about the actual behavior. *Precision* is the opposite of generalization and prohibits all traces that have not been seen in the event log. The problem with a behavioral model with high precision is that it is too specialized for the sample traces.

2.6 Summary

Upward compatibility is one important requirement for the BaaS approach. Today, REST is a predominant architectural style for creating web services and the main elements of REST are described in this chapter. REST APIs are described in different syntactic specification languages. Operations, inputs, outputs, and types are those elements that are critical for mashup creation and are available in existing specification languages. Mashup creation tends to be inefficient because the data models of the APIs and requests tend to be heterogeneous and incompatible. The BaaS approach aims to establish links to map heterogeneous data models to a unifying global ontology and this chapter presents state-of-the-art matching techniques to enable such mappings. In addition to the specification of the ontological semantics, the specification of the API protocols is essential for the efficient creation of mashups. In practice, they are unspecified which is why the BaaS approach aims to derive API protocols from call-logs using process mining. This chapter introduces three prominent process mining algorithms that are later evaluated for and used by the BaaS approach to derive API protocols from call-logs.

3 Related Work

In Section 1.2 three problems are identified which make mashup creation inefficient: (1) Request–API Terminological Heterogeneity, (2) Unspecified API Protocols, (3) API Incompatibility. The BaaS approach addresses these three problems with the components that are introduced in Section 1.4: (1) Semantic Annotator, (2) Protocol Miner, (3) Parameter Matcher and Glue Code Generator This chapter is structured around the thee identified problems. Each section presents related work addressing the respective problem.

The related works presented in this chapter are analyzed with respect to the requirements upward compatibility, learnability, effectiveness, comprehensiveness, and interoperability (cf. Section 1.3). Related approaches are assessed on a scale from fully satisfied (●), half satisfied (◑), and not satisfied (○). Upward compatibility is fully satisfied when the approach under consideration bases on REST APIs. Learnability is met when no new knowledge and skills need to be learned by the requester and provider or when the approach is fully automated. If an approach has already been evaluated for effectiveness, this requirement is fully met. Insofar as an approach takes into account both ontological semantics and behavioral semantics, the requirement of comprehensiveness is met. Approaches that are able to convert data between two APIs even with complex mapping fulfill the requirement of interoperability.

This dissertation is also related to On-The-Fly Computing[1], a novel programming paradigm for the automatic on-the-fly configuration and provision

[1] `https://sfb901.uni-paderborn.de`

of individual IT solutions made from base services that are available on a world-wide market. Related works in the research field of On-The-Fly Computing are also discussed in this chapter.

3.1 Request–API Terminological Heterogeneity

SWEET [8] is a tool that supports users to annotate the parts of HTML websites describing a REST API. Often REST APIs are described by regular HTML documents rather than in a particular specification language. These HTML documents do not have a uniform structure, but have custom formats. The plurality of custom formats complicates the task to extract information about a REST API automatically and to identify its operations, inputs, outputs, etc. within the custom HTML format. Therefore, such regular HTML documents are not well-suited for automatic service discovery, composition, and invocation.

SWEET parses HTML documents and SWEET users can link those HTML elements to corresponding concepts of a service description ontology, e.g., operations, inputs, outputs, etc. SWEET inserts annotations in hRESTS microformat [39] into the HTML document to markup these kinds of service properties. Once the inputs and outputs have been identified and marked up, they can be linked with semantic types from existing domain ontologies one by one. Semantic types being used for annotation can come from different ontologies. SWEET integrates the ontology search engine Watson [9] to search existing ontologies for adequate semantic types that can be linked with inputs and outputs. Watson constantly crawls the web and indexes the domain ontologies it encounters. Service specifications that have been semantically annotated with SWEET have not been evaluated to see if the effectiveness of service discovery improves.

To define the grounding of the semantic types to the technical data types, SWEET users need to define lifting and lowering transformations written in XSLT [56]. The definition of these XSLT transformations is out of the scope of SWEET and needs to be done manually. The readily annotated HTML document can be exported as an RDF [84] document which serves the purpose of a machine-readable service specification. Service specifications annotated by SWEET are more suitable for tasks like service discovery, composition, and invocation compared to the original HTML documents as they possess a uniform structure.

The existing infrastructure of REST APIs is upward compatible with SWEET. Adopters of SWEET need knowledge in XSLT with is why the learnability is reduced. The effectiveness of SWEET has not been evaluated. Because behavioral semantics are out of SWEET's scope, the comprehensiveness is limited. Interoperability and the exchange of data between different APIs is enabled through XSLT.

In contrast to SWEET, this dissertation assumes that the services are already described in a service specification language. The proliferation of frameworks like OpenAPI makes this a realistic assumption. While SWEET swaps out the task to establish the links of inputs and outputs to semantic types to Watson, how to establish these links is exactly in the focus of this dissertation.

The METEOR-S Web Service Annotation Framework (MWSAF) [6] is a framework for the semi-automatic annotation of WSDL specifications with ontologies. The annotations are intended to improve service discovery and service composition of heterogeneous third-party services. For this purpose, METEOR-S aims to add data semantics, functional semantics, execution semantics, and quality of service attributes to service specifications.

In WSDL, the types of operation inputs and outputs are defined in an XSD schema. MWSAF matches the XSD schema contained in a WSDL specification against a set of predefined domain ontologies. In a first step, the XSD schema and the domain ontologies are converted into a uniform, internal format to resolve the syntactic heterogeneity (cf. Section 2.3.7). Data types extracted from the XSD schema are compared pairwise with all concepts of all predefined domain ontologies. METEOR-S uses a series of common matching techniques, i.e., n-gram similarity, WordNet-based [3] synonym similarity, abbreviation expansion, stemming, tokenization, stop words removal (cf. Section 2.4) to calculate a matching score for each XSD/semantic type pair. The structural similarity of two XSD types is determined on the basis of the similarity of their subtypes. From these single similarity values, MWSAF calculates an overall score that assesses the similarity of the WSDL service and the respective domain ontology. Domain ontologies that reach a score above a threshold are presented to the user starting with the domain ontology with the highest score. MWSAF users have to decide if they approve the suggested domain ontologies. Approved domain ontologies are used to categorize the respective WSDL service. Additionally, the user can approve single mappings of XSD and semantic types of the accepted domain ontology. In doing so, the user is guided by the score of the single XSD and semantic type pairs. Service grounding including precise lifting and lowering transformations of fine-grained mappings of XSD types and semantic types is out of MWSAF's scope.

METEOR-S targets WSDL specifications and is therefore not suited for REST APIs. In [6], METEOR-S is evaluated on the basis of 24 public WSDL specifications: 15 ontologies from the geography domain and nine from the weather domain. One finding is that the domain ontologies do not contain enough classes to describe the ontological semantics of all XSD types found in the WSDL specifications. Therefore, only the classification accuracy is evaluated and METEOR-S has not been evaluated to see if the semantic annotations improve the effectiveness of service discovery.

Karma [85] is a data integration tool that allows integrating data from a variety of data sources including REST APIs. Data sources are integrated by aligning them to a domain ontology that is provided by the user. Karma has an interactive web-based user interface. To align a REST API to an ontology, users need to provide example request URLs of the API. Karma automatically invokes the service and extracts inputs and outputs. An initial service model is created by the tool which is then interactively refined in a dialog with the user. Karma comes with its own service description ontology to describe various characteristics of the service.

In the first step, the user selects a domain ontology to which the API's inputs and outputs are aligned. For every input and output, Karma suggests the top four most likely semantic types from the provided domain ontology. Suggestions are based on a machine-learning model, i.e., a conditional random field model [86] that learns from the mappings users have created before. With every new assignment of inputs/outputs, the random field model is retrained to improve its accuracy over time.

In a second step, the relevant relationships between the semantic types are identified. Not all possible relationships of the semantic types are meaningful for describing the REST API under consideration. To select meaningful relationships, Karma uses a heuristic: First, the solution space of all possible relations between the selected types is calculated. This solution space is a graph, where the nodes are semantic types and edges are relations between semantic types. The graph is then reduced to a minimal spanning tree which serves as a starting point to manually refine the relationships between semantic types. Once the user is satisfied with the resulting service model, it can be saved to a repository that can be accessed via a SPARQL [87] endpoint for service discovery. The lifting and lowering transformations that translate the SPARQL queries to API calls and vice versa do not require any specific technologies because the transformations realize one-to-one mappings of inputs and outputs to semantic types and vice versa. Such one-to-one mappings are usually not sufficient in practice where the linking of services' data types and semantic types require more complex mappings, which are addressed in this dissertation.

Karma is evaluated in paper [85]. The evaluation measures the time it takes the Karma authors to link APIs to ontologies using Karma. This is done using eleven operations of the same API. The effectiveness of Karma can thus hardly be shown, since the time required depends on the individual skills of the Karma users. Since the evaluation is carried out by the authors themselves, the results can hardly be transferred to inexperienced users. In addition, the terminological heterogeneity is hardly taken into account, since the examined operations all come from the same API.

AutomAPIc [10, 52, 88] is a tool for annotating service specifications and composing APIs. These annotations are intended to enhance the interoperability of the APIs and to improve their automatic composition. AutomAPIc generates sample input data that is in accordance with a given OpenAPI specification and invokes its operations with that input data. The response data is represented as a table, where every column corresponds to a single property of the response message and each cell in that column corresponds to a sample value of that property. Semantic Table Interpretation [89] is used to annotate the column headers with semantic types and values of the cells with semantic type instances. Relations between two named entity columns are represented as `owl:ObjectProperties`, while relations between named entity columns and literal columns are represented as `owl:DatatypeProperties`. AutomAPIc uses Semantic Table Interpretation for output parameters and natural language entity recognition (Stanford CoreNLP [11]) for input parameters to align parameters with semantic types. The ontological concepts used for the annotation come from the Linked Open Data cloud[2]. In the last step, the semantic type annotations are inserted into the original OpenAPI specification. While input parameters are annotated with class annotations only, the properties of response messages are annotated with classes and properties.

REST APIs are upward compatible with AutomAPIc. Existing literature does not evaluate how effectively AutomAPIc can add semantic annotations to APIs. The internal behavioral semantics of the APIs is out of AutomAPIc's scope, which is why comprehensiveness is limited. The authors of AutomAPIc also does not describe how concretely interoperability between different APIs can be achieved using the semantic type annotations.

OntoGenesis [7] is a tool to enrich the response messages of REST APIs with semantic annotations to enhance automatic service discovery, service composition, and interoperability. For this purpose, OntoGenesis associates the properties of the response messages with properties of existing, external ontologies. Since response messages and the ontologies are in different formats (e.g. JSON and OWL), an ad-hoc ontology is derived from the response message as a preliminary step to resolve syntactic heterogeneity (cf. Section 2.3.7). The structure of the ad-hoc ontology reflects the structure of the response message. OntoGenesis then determines semantic mappings between the properties of the ad-hoc and the external ontology. The similarity is determined by the Jaccard coefficient relating the number of shared values divided by the number of all values. The intuition is that the more values a concept of the ad-hoc ontology and the external ontology share, the more similar they are. The similarity of the values is determined based on the Levenshtein distance [71]. OntoGenesis also includes a so called *semantic adapter* that resides between the API caller and the original REST-API. The semantic adapter delegates the calls to the original REST-API, determines/updates semantic mappings, and inserts semantic annotations into the response message using the JSON-LD [59] format.

REST APIs are upward compatible with OntoGenesis. The requirement of comprehensiveness is not fully fulfilled as OntoGenesis focuses on the APIs' ontological semantics, while the behavioral semantics are out of its scope. The effectivness of OntoGenesis are evaluated in [7]. The semantic adapter of OntoGenesis adds semantic type annotations to the response messages, but

[2]`https://lod2.eu/`

this alone is not enough to establish interoperability between APIs.

ASSAM [12, 90] is a tool for annotating syntactic WSDL specifications with ontological semantics. ASSAM includes three built-in ontologies for web service categories, operations, and datatypes. Users of ASSAM browse the parts of the WSDL specification and assign proper concepts from these three ontologies. ASSAM helps the user to select the suitable concepts by making automatic suggestions. The suggestions are based on the semantic annotations of other WSDL specifications that had been annotated manually before. In particular, ASSAM has trained several machine learning classifiers on already annotated WSDL specifications. The classifiers are then used to predict the category and mappings to ontological concepts for previously unseen WSDL specifications. The readily annotated WSDL specification can be exported as a OWL-S specification. This OWL-S specification contains a profile, a process model, a grounding, and a domain ontology. The grounding includes XSLT lifting and lowering transformations.

REST APIs are not upward compatible with ASSAM as it targets WSDL services. The approach is highly automatized which contributes to learnability. Adopters of ASSAM still need knowledge of XSLT to adjust the transformations. The effectiveness of ASSAM to assign parameters to a data type is extensively studied in [12, 90]. Behavioral service semantics are not addressed by ASSAM. Interoperability is accomplished through XSLT mappings.

SmartAPI [91, 92] is a specification language and framework based on OpenAPI. It includes a specification editor that automatically suggests automatic semantic annotations of response messages: The response messages are recursively traversed to extract keypath/value pairs. A unique keypath addresses a particular value in the whole response message. The segments of the keypaths and the values of the key are matched with the names of concepts from Identifiers.org[3] to establish a link to semantic types. The matching concepts are suggested to the smartAPI user who needs to approve the suggested links to semantic types.

Existing REST APIs are upward compatible with the approach. The approach hides the complexity for adding semantic annotations from its users. The effectiveness of smartAPI has not been evaluated.

[3] `http://identifiers.org`

Wang et al. [93] present an API recommendation algorithm for creating mashups. To do this, a knowledge graph is first created in which the nodes represent categories, APIs, existing mashups, and the desired target mashup. The nodes are represented with a SkipGram [94] vector. The similarity of the nodes is based on the cosine similarity. The result is an API recommendation list that contains APIs most similar to the target mashup and the APIs used by most similar mashups.

Existing REST APIs are upward compatible with the approach. The API recommendation is highly automatized with is why learnability is generally given. Wang et al. focus on the service discovery task and the approach does not require annotating APIs with semantic types. Individual mappings between the mashup request and recommended APIs and how the individual APIs interact with each other in the mashup cannot be understood from the API recommendations produced by the approach of Wang et al. For these reasons the approach is excluded from the comparison.

Yao et al. [95] propose an approach to enrich JSON documents, e.g., the response messages of RESTful web services, with semantic types. The approach consists of four steps: JSON parsing, semantic mapping, semantic enrichment, and ontology merging. The semantic mapping parses the JSON document and translates it into an ad-hoc ontology. Semantic enrichment supplements the ad-hoc ontology with additional information. Ontology merging fuses the ad-hoc ontology with other ontologies to enable interoperability between them. In particular, this step is a key component of this dissertation, but the details on this are not presented in the work of Yao et al.

Existing REST APIs are upward compatible with the approach. The approach requires that its users are proficient in ontology languages which limits learnability. The effectiveness of the approach has not been evaluated. API protocols are not part of the approach such that the requirement of comprehensiveness is not fulfilled. A mechanism for service grounding is not included in the approach which is why the interoperability is limited.

Farrag et al. [96] propose a mapping algorithm that refines WSDL specifications with OWL-S annotations. WSDL types are annotated by corresponding semantic types from predefined ontologies. To find candidate matches of

WSDL types and semantic types, Farrag et al. employ tokenization, lemmatization using WordNet, stop word elimination, and term frequency-based matching (cf. Section 2.4). If no matches are found, the semantic search engine Swoogle [97] is used as a fallback. Structure-based similarity techniques are used to rank the concepts to support users to find proper semantic types for annotations.

The approach targets WSDL services and is not suited for REST APIs. Learnability is generally given as the approach mostly hides the technical complexity from the adopters of the approach. If the approach is effective in establishing links between WSDL services and ontologies has not been evaluated. Comprehensiveness is not fully fulfilled because the behavioral semantics of the APIs are not addressed. How interoperability can be achieved is not discussed in [96].

Karavisileiou et al. [98] present a service description ontology for OpenAPI and an automatic approach to automatically instantiate service models from OpenAPI specifications. Inputs and outputs have to be annotated manually with semantic types. With the help of this dissertation, the approach of Karavisileiou et al. can be extended to create semi-automatic links to semantic types. Because [98] just presents a service description ontology but no approach to link APIs to semantic types, it is excluded from the comparison.

Wenwen Gong et al. [99] present the DAWAR approach for diversity-aware API recommendation for mashup creation. Based on keywords from the service requester, DAWAR generates matching mashup candidates from compatible APIs. What is special about the approach is that certain APIs are not overly preferred. The problem of diversity is out of the scope of this dissertation. DAWAR considers API compatibility at the API level and defines APIs as compatible if they have been used together in a mashup in the past. Therefore, the disadvantage of this approach is that mashups cannot be created for new types of requests that have not be seen in the past. In contrast to the DAWAR approach that considers API compatibility on API level, this dissertation investigates API compatibility at the parameter level. For this reason, the approach is excluded from comparison.

Table 3.1 shows the comparison of the related works regarding the research problem of terminological heterogeneity of requests and APIs with respect to

	Upward Compatibility	Learnability	Effectiveness	Comprehensiveness	Interoperability	Source
SWEET	●	◑	n/a	◑	◑	[8]
METEOR-S	○	◑	n/a	◑	○	[6]
Karma	●	◑	n/a	◑	◑	[85]
AutomAPIc	●	◑	n/a	◑	◑	[10, 52, 88]
OntoGenesis	●	◑	●	◑	◑	[7]
ASSAM	○	◑	●	◑	●	[12, 90]
SmartAPI	●	●	n/a	◑	◑	[91, 92]
Yao et al.	●	◑	n/a	◑	◑	[95]
Farrag et al.	○	●	n/a	◑	○	[96]

Table 3.1: Comparison of related works addressing *Request–API Terminological Heterogeneity*

the requirements upward compatibility, learnability, effectiveness, comprehensiveness, and interoperability.

3.2 Unspecified API Protocols

Bertolino et al. [13] present an approach to automatically derive behavioral models through static analysis of WSDL service specifications. For this purpose, the data types of the inputs and outputs of the various operations are analyzed and data flow dependencies between them are identified. These dependencies are represented by an automaton. The conformance of that automaton with the actual service's behavior is validated through test cases that are automatically derived. Last, the automaton is transformed into a Business Process Execution Language (BPEL) [100] specification.

The approach is not suitable for RESTful web services, because the input parameters of RESTful web services tend to have primitive data types and the outputs are usually complex JSON documents with a schema that is often unique for each operation. Since there are no type definitions that are shared across operations, data flow dependencies cannot be determined this way.

Liskin et al. [101] propose an approach to make level 2 (cf. Section 2.1.2) REST APIs compliant with HATEOAS (cf. Section 2.1.1). That means that the API protocols are initially unspecified. The approach suggests manually modeling the API protocols as UML state machines. A generic service wrapper sits in front of the original REST API and exposes it as a HATEOAS compliant REST API to the clients. For this purpose, the service wrapper interprets the UML state machines, looks up the possible transitions, creates hyperlinks from the transitions, and inserts them into the response messages.

REST APIs are upward compatible with the approach of Liskin et al. The requirement of learnability is limited as the adopters of the approach need to be proficient in UML and have to create the state machines completely manually. On one hand, UML state machines describing the behavior of REST APIs are in fact API protocol specifications. On the other hand, the ontological semantics of the API are not in the focus of the approach. Therefore, comprehensiveness is not fully covered. This dissertation aims to derive API protocols semi-automatically and that derived API protocols would complement the work of Liskin et al.

Schur et al. [102] present ProCrawl: an approach to create behavioral models for the testing of enterprise web applications with HTML-based user interfaces. ProCrawl opens the landing page of the web application and observes the application state which is automatically determined by the HTML elements shown in the user interface. An initial behavioral model given in the shape of a finite state automaton is created. The nodes of the automaton represent application states and the transitions represent user actions. The behavioral model is refined in an iterative process by manually traversing the desired interaction paths and application states that need to be captured.

ProCrawl requires web applications with HTML-based user interfaces so that is not generally applicable to REST APIs. This approach does not scale to capture API protocol specification of a REST API: First, REST APIs do not have a user interface at all, which ProCrawl needs to determine application states. Second, capturing a complete API protocol specification would include traversing all possible interaction paths manually.

Dustdar and Gombotz [14, 15] introduce the principle of Web Service Interaction Mining. Web Service Interaction Mining performs process mining on the call-logs of web services. The call-logs are translated into event logs which are processed by the process mining tool ProM [103]. For process mining, the associated events must be assigned to a particular case. However, the case of an event can not always be determined from a call-log. Successive events in a call-log do not necessarily belong to the same case, as multiple user sessions may interfere. Dustdar and Gombotz propose several methods to reconstruct the session. A session identifier identifies the different cases.

The similarities of Dustdar's and Gombotz's work and this dissertation are that both works use process mining to derive behavioral models from call-logs. The work of Dustdar and Gombotz and this dissertation address different problems. While the focus of the work of Dustdar and Gombotz is on analyzing the interaction of multiple services in a mashup, the focus of this dissertation is to derive the API protocols of single web services. Dustdar and Gombotz do not explicitly investigate to what extent process mining is suitable for gaining API protocol specifications. In addition, the work of Dustdar and Gombotz aims at SOAP-based web services while RESTful web services are prevalent nowadays.

Ghezzi et al. [104] propose the BEAR approach which infers a set of probabilistic Markov models of the users' behavior from a web service interaction history given in the shape of a log file. The inferred models are used to verify quantitative properties by means of probabilistic model checking. This allows, for example, to determine the probability that a user, who enters the application from a certain link and navigates through the application by a certain path, will reach a given target page. The insights can be used to improve the application's navigational patterns according to the users' requirements. It has not been evaluated to what extent the BEAR approach is suitable to derive API protocol specifications. Because the work of Ghezzi et al. aims to mine user behavior and not API protocols, it is excluded from the comparison.

Van der Aalst and Pesic [105] propose the declarative language DecSerFlow which can be used to specify service flows. Traditionally, Petri nets or BPEL

	Upward Compatibility	Learnability	Effectiveness	Comprehensiveness	Interoperability	Source
Bertolino et al.	○	●	●	◑	○	[13]
Liskin et al.	●	○	n/a	◑	○	[101]
Schur et al.	○	●	n/a	◑	○	[102]
Dustdar and Gombotz	○	◑	n/a	○	○	[14, 15]

Table 3.2: Comparison of related works addressing *Unspecified API Protocols*

specifications are used to specify such service flows. DecSerFlow is intended to overcome the shortcomings of Petri nets and BPEL specifications that tend to over-specify the service flows. The approach to derive API protocol specification that is presented in this dissertation is not limited to a specific language but is designed so that it can be adopted for DecSerFlow. Because DecSerFlow is just a language but not an approach to derive API protocols, it is excluded from the comparison.

Thummalapenta et al. [106] introduce the PARSEWeb tool that programmers can use to find out in what order they need to call third-party API operations to get a destination type given a data source type. For this purpose, the approach uses static code analysis of Java program code and is therefore a white-box approach. This dissertation, on the other hand, is a black-box approach and uses dynamic analysis of data traffic. Because the approach of Thummalapenta et al. is a white-box approach, it is excluded from the comparison.

Table 3.2 shows the comparison of the related works regarding the research problem of unspecified API protocols.

3.3 API Incompatibility

Salvadori et al. [107, 108] address data integration of heterogeneous microservices and propose the Alignator framework that aims to achieve the semantic data-driven composition of microservices across different domains. The framework integrates external ontology matchmakers which match the entities of the microservices with the entities of a predefined set of existing ontologies. The approach expects that the microservices are already equipped with semantic annotations, which is usually not given in practice. One main objective of this dissertation is exactly to annotate API parameters with semantic types.

The corpus of existing REST APIs is upward compatible with the approach. It is not examined to what extent the approach effectively supports actually enabling data exchange between APIs. Alignator considers only ontological semantics, but no behavioral semantics of the APIs, which is why comprehensiveness is limited. No detailed information is provided on how exactly the semantic type annotations are used to enable the interoperability of different APIs.

Izquierdo et al. [16] present a discovery approach to extract ontologies from JSON-based web APIs. The domains are visualized as class diagrams to support web developers in understanding the APIs they want to integrate into their applications. Izquierdo et al. propose guiding rules on how the services can be composed. In addition, they propose simple heuristics to identify corresponding semantic types across the domain ontologies that rely on exact name and type match. Such exact techniques are ineffective when different terminologies are used. More elaborated techniques are presented in this dissertation.

The approach targets REST APIs and existing REST APIs are upward compatible with the approach. Indeed, the produced class diagrams help to better understand APIs but still leave a lot of manual work which is why adopters of the approach have to develop skills in how to work with these diagrams, such that the requirement of learnability is not fulfilled. Comprehensiveness is limited as behavioral semantics are not in the scope of the approach. How interoperability can be achieved is not presented in detail.

Serrano et al. [109] propose the Linked REST APIs (LRA) framework which maps the data exposed by REST services to ontologies. The LRA middleware implements lifting and lowering translations: SPARQL [87] queries are automatically translated into API calls and the response of the API calls is translated back to semantic types. How the mappings of the response messages and semantic types are established is out of the focus of the work of Serrano et al., but in the focus of this dissertation.

Existing REST APIs are upward compatible with the approach. The connection between semantic types and API parameters is established with SPARQL expressions. Even complex mappings can be realized with SPARQL, which is why interoperability can be realized. Reaching interoperability is up to the adopters of the approach who have to be proficient in SPARQL, which limits learnability. The effectiveness of the approach is exhaustively evaluated. As behavioral semantics are out of the scope, comprehensiveness is not fulfilled.

Klímek et al. [110] propose a method that can be used to map an external domain ontology like schema.org to an existing conceptual model. XSLT-based lifting and lowering transformations can be generated from this conceptual model. The method supports 1:1 element correspondences between the conceptual model and the ontology only. These correspondences have to be provided manually.

Upward compatibility with REST is not given as the approach requires that APIs are described by a conceptual model with is generally not the case for REST APIs. As adopters of the approach have to be able to work with conceptual models and XSLT, the learnability is not fulfilled. The effectiveness of the approach is not evaluated. Only ontological semantics are in the scope of the approach, which is why comprehensiveness is limited. Interoperability is realized through XSLT transformations but the approach considers only 1:1 element correspondences between the conceptual model and the ontology which is not sufficient for real-world applications.

Burstein et al. [17] propose the idea to automatically generate glue code that translates the heterogeneous data of web services that are supposed to communicate with each other. The authors cast this translation problem as solving higher-order functional equations.

Upward compatibility of the approach is not fulfilled, because it is not

	Upward Compatibility	Learnability	Effectiveness	Comprehensiveness	Interoperability	Source
Salvadori et al.	●	●	n/a	◑	◑	[107, 108]
Izquierdo et al.	●	○	n/a	◑	◑	[16]
Serrano et al.	●	○	●	◑	●	[109]
Klímek et al.	○	○	n/a	◑	◑	[110]
Burstein et al.	○	○	n/a	○	◑	[17]

Table 3.3: Comparison of related works addressing *API Incompatibility*

described how REST APIs are represented in lambda calculus. Learnability is also not given, as adopters of the approach have to be proficient in lambda calculus. Effectiveness has not been evaluated. Comprehensiveness is not fulfilled as behavioral semantics are out of the approach's scope. Interoperability is realized by the generated glue code.

Liu et al. [111] present an approach with which IFTTT mashup infrastructures can be generated automatically. An IFTTT service is a simple programming model consisting of a trigger, a rule, and an action. Triggers and actions are offered by different service providers and can be combined to form mashups. The approach of Liu et al. automatically creates the cloud infrastructure needed for mashups. Liu's approach does not deal with how triggers and content actions can be combined with one another and how, for example, the output data of the trigger can be used in the input parameters of the action. This compatibility on the parameter level is of particular interest of this dissertation. This dissertation also targets REST APIs, which represent a much more complex programming model, than simple IFTTT services. Because the work of Liu et al. targets a different research problem than this dissertation, it is excluded from the comparison.

JXML2OWL [112] is a tool that can be used to manually create mappings between an XML schema and an OWL ontology. These mappings can be exported as XSLT transformation scripts. The tool can also automatically generate XSLT transformations from these mappings. Because does not address REST APIs in particular, it is excluded from the comparison.

Table 3.3 shows the comparison of the related works regarding the research problem of API incompatibility.

3.4 On-The-Fly Computing

In the remainder of this section, this dissertation is related to works in the research field of On-the-Fly Computing. The vision of On-the-Fly Computing (OTF) is that service requesters only need to describe their requirements whereupon basic services are combined in such a way that they meet the desired functional and non-functional characteristics. The works presented in this section have links to this dissertation, but fundamentally addresses other research questions. Therefore, the approaches in this section are not analyzed in terms of requirements.

In an OTF market, basic services are provided by independent service providers. There are several service specification languages, such as OpenAPI, WSDL, OWL-S, that service providers can use to describe their services. Different services offered in an OTF market may be described in various specification languages. Which functional and non-functional service properties can be described in certain ways is different per specification language. This syntactic heterogeneity of the specification languages complicates their machine processing, e.g., during service discovery. Arifulina [67] introduces a core language that unifies various service specification languages. Furthermore, she presents an approach that learns from examples how to transform a service specification described in a proprietary language into this core language. Syntactic heterogeneity is not in the scope of this dissertation and to simplify the problem of syntactic heterogeneity, this dissertation uses OpenAPI as a unified basis. However, it is possible to use Arifulina's approach as a precursor to the BaaS approach to eliminate syntactic heterogeneity. In that case, the core language would take the place of OpenAPI.

In service discovery, service requests are matched with service specifications. If there is vagueness in the service requests or if the functional and non-functional properties of the provided services are not fully specified, traditional service discovery approaches tend to be ineffective. Platenius [113] proposes the concept of fuzzy matching to counteract the uncertainty due to the vagueness in service requests and the incompleteness of service specifications. This is done by informing service requesters about the fuzziness of service discovery. In contrast, my dissertation explores how the incompleteness of service specifications can be reduced by deriving and formalizing ontological and behavioral semantics of the services. It is also shown in my dissertation how the uncertainty regarding the ontological semantics between service requesters and service providers is resolved by linking service requests and service offers to a global ontology. The reduced fuzziness manifests in a more effective service discovery, which is shown in my dissertation.

Huma's work [5] focuses on how effective service discovery and composition can be enabled despite the multi-dimensional heterogeneity between service requesters and providers. Huma's approach is based on the fact that service requests and service offers are specified in a comprehensive service specification language — the rich service description language (RSDL). In RSDL, every specification defines its own local ontology which is also used as the service's data model. Since the data models are independent of each other, they are heterogeneous. Before RSDL requests and RSDL offers can be matched in service discovery, they need to be normalized. For this purpose, Huma suggests mapping the local ontologies to a unified, global ontology. The ontology matcher presented in this dissertation is an advancement of the ontology matcher used in Huma's approach [21, 114]. In Huma's work, the normalization of service offers takes place at the time of the service discovery, where a service request is matched pairwise will all available service offers. In practice, it is unrealistic that the heterogeneity can be resolved fully automatically. A manual consolidation process between service requesters and service would be necessary to completely resolve the heterogeneity. Such a consolidation process does not scale for a large number of service requests and offers. In contrast, in my approach, the normalization of service offers takes place before discovery time.

Besides the local ontologies, RSDL specifications also include API protocol specifications in the shape of UML sequence models. Such API protocol specifications are not provided in practice and how API protocol specifications are created is not addressed by Huma. In my dissertation, I present an approach to derive API protocol specifications from call-logs semi-automatically in order to facilitate the creation of API protocol specifications.

Huma proposes a service composition approach that bases on API protocol matching. The service composition takes place at the level of the normalized specifications that are typed over the global ontology. The specifications deviate from the actual implementation of the service offers that still use their local data models. To obtain executable mashups, the gap between the normalized specifications and the actual implementations needs to be closed by a service grounding, which is not addressed by Huma's work. In contrast, this dissertation also considers the execution level and how heterogeneous data can be shared between different services in a mashup.

3.5 Summary

Existing related work addresses only a few of the four problems stated in Section 1.2, i.e., insufficient specification of ontological semantics, insufficient specification of behavioral semantics, and incompatibility of third-party services in a mashup. There is no approach that addresses all four problems holistically. Consequently, there is no approach that retrieves comprehensive service specifications including ontological and behavioral semantics.

Although there are numerous approaches that enrich service specifications with ontological semantics, for none of these approaches it has been verified for a large number of real-world service requests and offers whether the semantic annotations help to improve the effectiveness of service discovery. Therefore, the question of the scalability of these approaches remains open.

In practice, interoperability between heterogeneous third-party services to be composed in a mashup requires often complex translation functions that translate the data from one data model into another. The proposed approaches consider only simplistic one-to-one correspondences that are often not sufficient in practice.

4 Solution Overview

Brokerage as a Service (BaaS) is a novel kind of IT service that substantially supports third-party service providers and requesters with semi-automatic tools which help them to better collaborate on creating mashups: (1) The terminological heterogeneity of service requests and service offers is resolved by annotating them with semantic types of a global ontology, enabling an effective service discovery. (2) API protocol specifications, extracted from call-logs using process mining techniques, help service requesters to understand in which order operations of third-party APIs have to be called. (3) The ontological annotations help to identify semantically related input and output parameters of different APIs that may be exchanged between APIs in a mashup. (4) Glue code that converts the data between the heterogeneous APIs is generated from fine-grained parameter mappings, making the APIs interoperable.

Service requesters, service providers, and BaaS vendors do benefit from BaaS likewise: Service requesters can find and compose existing APIs more effectively and are therefore able to reduce development time and costs for creating their mashups. Because of this, the service providers can reach a wider range of consumers that come with the new mashups. Vendors of cloud computing platforms that exist today can expand their platforms to BaaS platforms which is a unique selling point opposed to ordinary cloud computing platforms.

4.1 Architecture

Figure 4.1 shows the main components of a BaaS platform, i.e., API Registry, Semantic Annotator, API Gateway, API Protocol Miner, Service Discovery, Parameter Matcher, Glue Code Generator. The purpose and functionality of

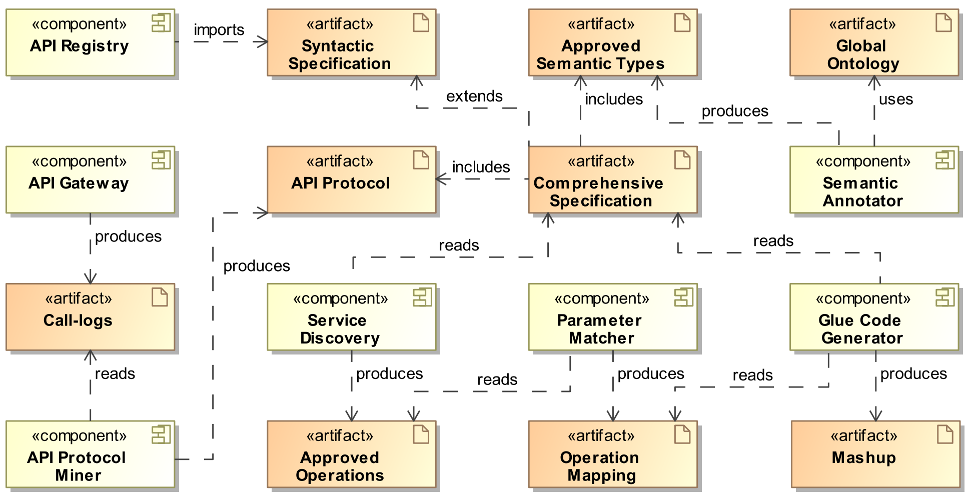

«component» API Registry
imports
«artifact» Syntactic Specification
«artifact» Approved Semantic Types
«artifact» Global Ontology
extends
includes
produces
uses
«component» API Gateway
«artifact» API Protocol
includes
«artifact» Comprehensive Specification
«component» Semantic Annotator
produces
produces
reads
reads
«artifact» Call-logs
«component» Service Discovery
«component» Parameter Matcher
«component» Glue Code Generator
reads
produces
reads
produces
reads
produces
«component» API Protocol Miner
«artifact» Approved Operations
«artifact» Operation Mapping
«artifact» Mashup

Figure 4.1: BaaS components and artifacts

each component are explained in the remainder. A prototype implementation of the BaaS components is publicly available on GitHub[1].

API Registry

The API registry stores comprehensive specifications of all API offers that are available on the BaaS platform. All these specifications are considered by the service discovery and for creating new mashups.

Providers can import their legacy syntactic specifications, i.e., OpenAPI specifications into the API Registry to upgrade them to comprehensive specifications. As described in the following, the providers use the Semantic Annotator and the API Protocol Miner to enrich these specifications with semantic types and API protocols to obtain comprehensive specifications.

4.1.1 Semantic Annotator

The Semantic Annotator links service requests and service offers with semantic types from the global ontology. This eliminates the terminological heterogeneity of service requests and service offers that makes Service Discovery and Parameter Matching ineffective. It is important that service requesters and service providers use semantic types that are somehow semantically connected such that the service matchmaker can reason about the similarity of

[1]`https://github.com/brokerage-as-a-service/baas`

a requested and provided data type and ultimately if a service offer satisfies a given service request. If service requesters and service providers would use isolated and unconnected semantic types (from different ontologies), i.e., semantic types that are not in an is-a or is-equivalent relationship, this would only shift the terminological heterogeneity from the level of service specifications to the level of ontologies instead of resolving the heterogeneity. The responsibility of the BaaS vendor is to provide that global ontology. Service requesters and service providers that are using a BaaS platform are restricted to using a shared set of semantic types provided through the global ontology whereby their terminological heterogeneity is resolved.

An API can be offered on multiple BaaS platforms, each using different global ontologies. Ideally, the global ontologies of different BaaS platforms are interconnected so that equivalent semantic types reference each other. Otherwise, the service providers have to annotate their APIs several times with the semantic types offered by each BaaS platform where they want to offer their API.

Comprehensive specifications exist only for effective service discovery. They only describe a fictitious variant of the actual API that has been expanded by semantic types. How the actual underlying API is called is still described in the syntactic specification. The advantage of this lightweight approach is that it allows an effective service discovery, while the actual API implementations and applications that use these APIs do not have to be changed.

The Semantic Annotator finds semantic types in the global ontology that are suitable to annotate service requests or service offers. In particular, it returns a list of semantic types from the global ontology ranked by their relevance.

The Semantic Annotator analyzes the syntactic specifications and automatically derives search queries which are used to search the global ontology for relevant semantic types to annotate the APIs' input and output parameters.

In contrast to common state-of-the-art approaches, the Semantic Annotator considers even more contextual information from the service specification than just the parameter names, e.g., the name of the enclosing operation. In general, adding more contextual information increases recall and decreases precision. The difficulty lies in selecting the contextual information so that the search for semantic types is generally effective for a large number of APIs.

API Gateway

Where do the call-logs required to mine API protocols come from? Usually, service providers offer their APIs via a web server that is publicly available on the Internet. Web servers run a web server software like Apache HTTP[2], Apache Tomcat[3], Nginx[4], Express[5], etc. Service providers are completely free in which web server software they use because the REST API exposed to the client application is the same in each case. All the web server software mentioned before is shipped with the capability to create call-logs[6]. All these web server implementations implement the HTTP/S transport protocol. What the call-logs produced by different web server implementations have in common is that they track meta-information that is transmitted in HTTP/S request and response messages between the clients and the server. In particular, any kind of data that is transmitted within an HTTP/S message [32] can be recorded. However, every web server implementation uses a different call-log format. Process mining by the BaaS cloud vendor would be complicated by the syntactic heterogeneity of the different log formats.

To counteract the problem of different call-log formats, BaaS uses an API Gateway: The API gateway sits in front of all APIs that are available on the BaaS platform. The clients send their request to the API Gateway that acts as a central proxy and forwards the HTTP/S requests to the respective APIs and returns their HTTP/S responses back to the client. Since all traffic between any clients and APIs goes through the API Gateway, it is possible to create call-logs in a uniform format for all APIs.

Figure 4.2 shows the relationship between clients, BaaS API Gateway, and the actual third-party APIs: Client A sends a request to the API at the domain `lufthansa.baas.com`. The API Gateway forwards the request to the original domain at `api.lufthansa.com` and returns the response message back to the client. Meta-information from the HTTP/S request and response messages are recorded in a call-log without Lufthansa having to change anything in their API.

Using API Gateways is already a common practice today. For example,

[2]`https://httpd.apache.org/`
[3]`http://tomcat.apache.org/`
[4]`https://www.nginx.com/`
[5]`https://expressjs.com/`
[6]In the context of web server software, call-logs are also known as *web server logs* or *access logs*

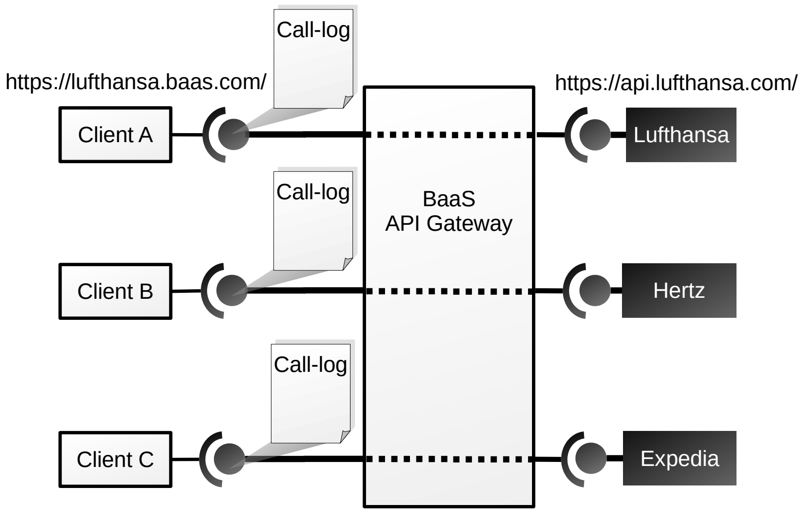


Figure 4.2: BaaS API gateway

the API management platform RapidAPI[7] uses API gateways. Infrastructure as a Platform cloud providers like Amazon Web Service (AWS) also offer the use of API Gateways[8]. The difference between the BaaS API Gateway and the RapidAPI or AWS API Gateway is that the latter is used for other tasks like authentication, billing, performance monitoring while the BaaS API Gateway is also used to record call-logs. The novelty is that BaaS uses API Gateways to create call logs in a uniform format in order to use them for process mining. Existing API gateways like the API gateway of RapidAPI can easily be extended to create call-logs for process mining.

4.1.2 API Protocol Miner

The API Protocol Miner assists service providers to create the API protocol specifications that service requesters require to understand the interdependencies of operations of an API they want to integrate into a mashup. The input of the API Protocol Miner is a call-log created by the API Gateway containing calls from all applications that have ever called this API. The output of the API Protocol Miner is an API protocol specification.

Mined API protocol specification can only describe operation sequences that have ever occurred in a call-log. To mine API protocol specifications that

[7]`https://rapidapi.com/`
[8]`https://aws.amazon.com/api-gateway/`

cover the complete behavior of APIs, it is a necessary criterion that all valid operation sequences have ever occurred and have been recorded in a call-log. In general, the completeness of mined API protocols improves over time the more traffic the API Gateway has seen.

When a BaaS platform is launched, call-logs are empty in the beginning. In order to generate the necessary traffic to fill the logs, in an initial start-up phase BaaS platforms can start as ordinary API management platforms. As soon as an API has seen some reasonable amount of traffic and the call-log is populated, the BaaS provider can publish the mined API protocol.

Even after the start-up phase, there might be still certain operation call sequences that are edge cases and have never been traced in a call-log. Consequently, the mined API protocols do not cover the complete behavior of an API. Ignoring infrequently occurring operation sequences is a compromise that the BaaS approach accepts as long as the essential behavior of an API is described. The hypothesis is that sequences that were rarely used in the past are likely to be rarely used in the future by new service requesters.

Service Discovery

The service discovery matches a service request with all service offers. The service requests and the service offers are specified in a specification format that supports semantic types, e.g., OWL-S. Other specification languages that support semantic type annotations are also possible, e.g. the extensions of OpenAPI presented in [52, 53].

There are dozens of existing service matchmakers supporting OWL-S that can be used. Because of the plenitude of existing service matchmakers, service discovery itself is not the focus of this dissertation. Service discovery is rather considered as a placeholder in the BaaS platform that can be equipped with any service matchmaker of the BaaS vendor's choice. State-of-the-art service matchmakers are able to take semantic type annotations into account to increase the effectiveness of the search for APIs. The input of the service discovery is an OWL-S request. For every operation in the request, a list of operations of all APIs ranked by their relevance is returned. The service requester inspects the retrieved operations and manually selects those operations to use in their mashup.

Parameter Matcher

The individual operations that are used together in a mashup consume/produce lots of inputs/outputs, but only certain inputs or outputs have to be exchanged between them in the specific context of the mashup. The inputs and outputs that need to be exchanged between APIs are usually highly incompatible: For example, the output o of an API needs to be passed as input parameter i to another API while o and i may use different data types and formats. Despite the heterogeneity of the parameters, the Parameter Matcher helps to determine related parameters that are likely to be interchangeable: It extracts all inputs and outputs of the operations to be composed in a mashup, matches them pairwise, and calculates a relevance score for each pair. The relevance score takes the semantic types of the inputs and outputs into account. This allows to effectively identify related parameters across the operation even though the APIs of the services are heterogeneous and incompatible. The output of the Parameter Matcher is a ranked list of parameter mappings.

4.1.3 Glue Code Generator

The relevance score of the parameter mappings just gives the service requesters a hint *which* inputs/outputs must be exchanged, but not *how* the data can be exchanged. Inputs and outputs of services from different third-party providers are using different data types and data formats that need to be made compatible before they can be exchanged within a mashup.

The Glue Code Generator takes a list of parameter mappings and generates the glue code, i.e., the program logic, that converts between the heterogeneous data types and formats which allows them to be exchanged across the operations. The generated glue code invokes the involved operations of the mashup, extracts the required inputs/outputs from request/response messages, and inserts translation functions to harmonize data types and formats. The glue code has an individual converter function for each parameter mapping. The converter functions are inserted using a heuristic. Since there is an individual translation function per parameter mapping, specific functions can be manually adjusted.

So far, the components of the BaaS have been explained. The following section explains how service requesters and service providers are using these components.

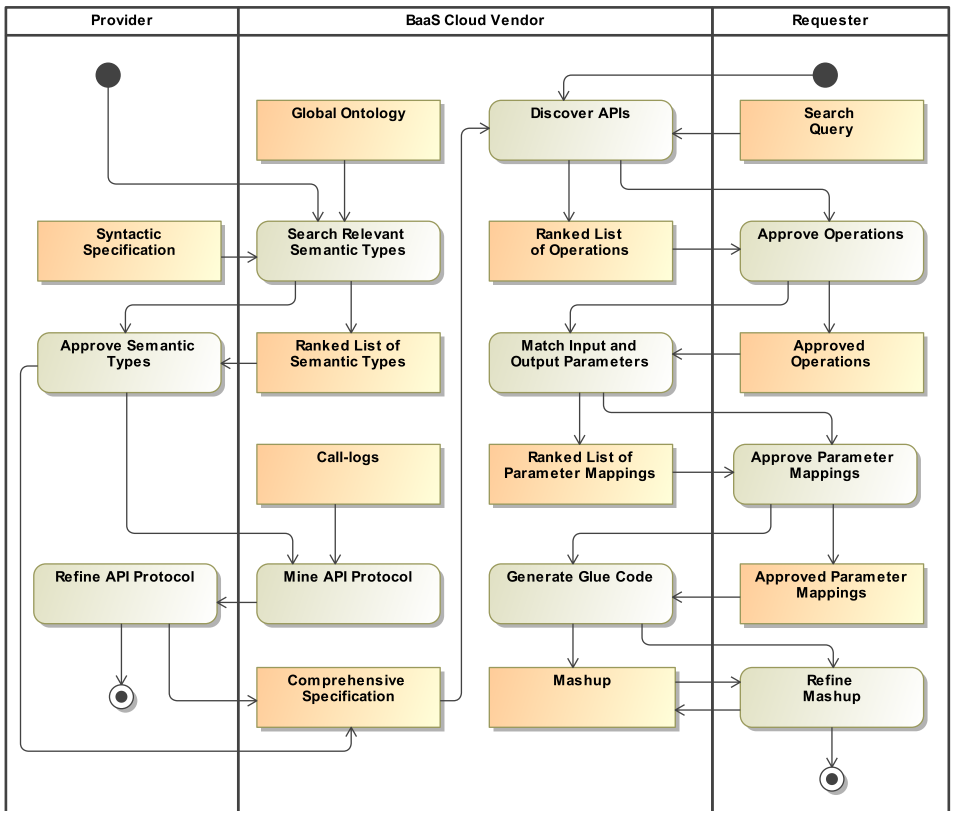


Figure 4.3: BaaS interaction processes of requesters and providers

4.2 Usage

Figure 4.3 shows how service requesters and service providers are interacting with the BaaS platform. Service providers publish syntactic specifications at the *API Registry*. (1) Using the Semantic Annotator, the service providers link the input and output parameters of their operations with semantic types from the Global Ontology. (2) They inspect the recommended semantic types and manually approve the ones they consider as accurate. If the Semantic Annotator is effective, service providers have to inspect only a small fraction of the first-ranked semantic types.

(3) Next, the service providers are using the Protocol Miner to extract API protocol specifications from call-logs, collected at the API Gateway. Due to noise in call-logs, extracted API protocols may be inaccurate. (4) The service providers inspect them and make manual corrections, if necessary. The

semantic type annotations and API protocol specifications are then part of a comprehensive service specification that captures the ontological and behavioral semantics of the APIs.

(5) Service requesters use the Service Discovery of the BaaS platform to discover relevant operations providing the desired functionality. The Service Discovery exploits the semantic type annotations of the service requests and service offers to overcome their terminological heterogeneity which results in finding relevant operations more effectively. The result of Service Discovery is a list of operations ordered by their relevance to the request. (6) Service requesters inspect the suggested operations and select those operations manually that they want to use in a mashup.

(7) The Parameter Matcher matches the input and output parameters of the operations that have been selected for mashup creation to determine corresponding parameters that must be exchanged in a mashup. To determine corresponding parameters, the Parameter Matcher uses semantic type annotations. (8) The service requesters inspect the corresponding parameters and approve the input/output mappings manually.

(9) From the list of approved parameter mappings, the Glue Code Generator generates the program code of a mashup that is executable. The program code contains the operations calls and the extraction of single input/output parameters from whole request/response messages. The generated code also contains converter functions that resolve syntactic heterogeneity. Thus, the generated program code already contains a lot of program logic that otherwise would have had to be programmed manually.

(10) In case the converter function predicted by the heuristic is inaccurate or incomplete, it still has to be adjusted manually by the service requester. These adjustments can be made specifically for each parameter mapping, as there is a separate converter function for each parameter mapping.

The following section explains why the BaaS approach meets the requirements defined in Section 1.3.

4.3 Meeting the Requirements

This section explains why the BaaS approach satisfies the requirements that have been raised in Section 1.3, i.e., upward compatibility, learnability, effectiveness, comprehensiveness, and interoperability.

4.3.1 Upward Compatibility

With BaaS, existing REST APIs can continue to be used. Adjustments to the APIs or service implementations are not necessary. BaaS builds on existing infrastructure with thousands of REST APIs that can be composed into new mashups. Because of this, BaaS is upward compatible.

4.3.2 Learnability

The BaaS approach allows service providers to continue using the technologies they are familiar with. Already created syntactic service specifications, e.g., legacy OpenAPI specifications can be reused. Changes to the implementations of existing APIs are not required. No new expertise has to be learned. Techniques such as process mining are used in the background so that it is largely transparent to providers and requesters: Call-logs, which are the input for process mining, are created automatically at the API Gateway without the intervention of the service providers. The selection of the process mining algorithm and its configuration is also handled by the BaaS platform. For these reasons, BaaS is accessible to a wide range of service providers and service requesters.

4.3.3 Effectiveness

Linking specifications to semantic types gives them a machine-accessible meaning. Service discovery can exploit the description of the ontological meaning and thereby reduce terminological heterogeneity which allows to narrow down the large set of all available operations to a small set of relevant API operations. Since service requesters only have to look at a smaller fraction of operations that are useful for their desired mashup, service discovery is more effective. Because the approach is effective, it also scales for large numbers of services.

4.3.4 Comprehensiveness

Understanding the ontological and behavioral semantics of APIs is crucial to integrating them into a mashup. By enriching syntactic service specifications with ontological semantics in the shape of semantic types and behavioral semantics in the shape of API protocol specifications, service requesters are now able to understand aspects of black-box services that are inaccessible solely based on purely syntactic service specifications.

4.3.5 Interoperability

The BaaS approach achieves interoperability on two levels: First, by linking the input and output parameters of the APIs to semantic types from a unified global ontology, the heterogeneity of the APIs is reduced. This makes it easier to identify the relations of input and output parameters of different APIs that might be used in a future mashup. Second, the glue code generated from parameter mappings allows wiring the inputs/outputs of the APIs together and converting their heterogeneous data. Both aspects make incompatible APIs of different service providers interoperable with each other such that they can be combined in a mashup.

4.4 Summary

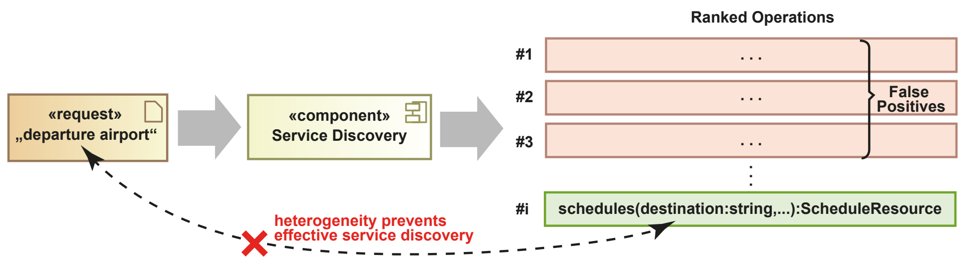
This chapter introduces Brokerage as a Service (BaaS): A novel IT service that provides semi-automatic tools for service requesters and service providers assisting them to better collaborate for the creation of mashups. It is a wide span for the requester to come from a vague service request to an executable mashup. The many decisions that need to be made along the way to obtain an executable mashup are difficult to automate and therefore require human interaction. This chapter introduces a process of individual small steps, always alternating between automatic and manual steps and helping service requesters and service providers in collaborating to create mashups: The Semantic Annotator assists service requesters and service providers with annotating service requests and service specifications with semantic types of a global ontology, eliminating their terminological heterogeneity. With the aid of the semantic types, the Service Discovery can find relevant operations for a service request more effectively. The Protocol Miner learns an API protocol specification from call-logs using process mining. Call-logs are automatically created by the API Gateway. Service requesters use the API protocol specification to comprehend the interdependencies of operations of an API to deciding which operations are needed in a mashup. The Parameter Matcher produces input/output parameter mappings that assist the service requester in identifying those inputs and outputs that must be exchanged between the operations involved in a mashup. From input/output parameter mappings, the Glue Code Generator generates the code that converts between the heterogeneous data types and formats of the heterogeneous APIs to make them interoperable. With this variety of semi-automated tools, BaaS makes creating mashups much more efficient.

5 Semantic Annotator

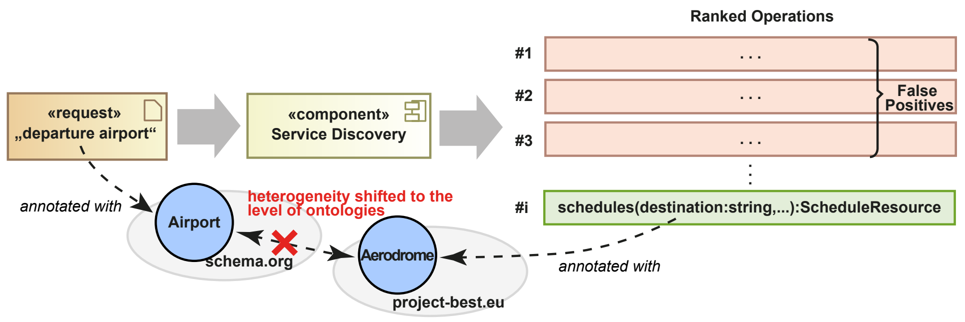
Today, service discovery tends to be ineffective because requesters and providers are independent and the data model and operation signatures of requests and offers are terminologically heterogeneous. Figure 5.1 shows three stages of heterogeneity. In Figure 5.1a, purely syntactic specifications are used, as is currently done on RapidAPI. RapidAPI uses string-based similarity measures for the search, which is why, for example, "destination" is not found when you search for "departure airport".

State-of-the-art semantic service matchmakers [4, 54] use even more similarity measures next to the string similarity of parameter names, for example the ontological proximity of the semantic types in a hierarchy (is-a) relationship. The latter requires that operation signatures use semantic types and that these semantic types are somehow related to each other. There are existing tools like SWEET [8] for annotating APIs with semantic types. The problem with these approaches is that they search multiple, unrelated ontologies for semantic types for annotation. The situation where requesters and providers use semantic types from unrelated ontologies is shown in Figure 5.1b. In this case, there is no relation between `Airport` and `Aerodrome` such that the calculation of the relevance score ultimately falls back on the string similarity of parameter names. Consequently, the terminological heterogeneity is only shifted from requests/offers to the level of ontologies. If, however, requesters and providers use semantic types from the same global ontology, the relations can be taken into account by existing semantic service matchmakers, which is shown in Figure 5.1c.

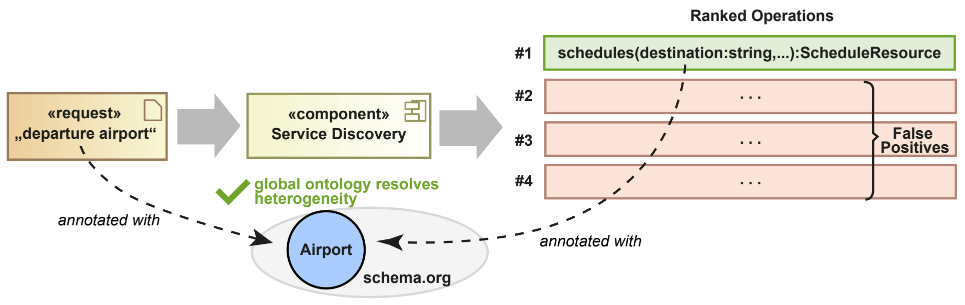
Brokerage as a Service annotates input and output parameters of offers and requests with semantic types of a global ontology. By using this global ontology, their heterogeneous data models are unified, making it more effective


Ranked Operations
#1
#2
#3
#i
. . .
False Positives
«request» „departure airport“
«component» Service Discovery
schedules(destination:string,...):ScheduleResource
heterogeneity prevents effective service discovery

(a) Heterogeneity between requests and offers: False negative match between parameters *departure airport* and *destination*.


Ranked Operations
#1
#2
#3
#i
. . .
False Positives
«request» „departure airport“
«component» Service Discovery
schedules(destination:string,...):ScheduleResource
annotated with
Airport
schema.org
heterogeneity shifted to the level of ontologies
Aerodrome
project-best.eu
annotated with

(b) Heterogeneity between ontologies: False negative match between classes `Airport` and `Aerodrome`.


Ranked Operations
#1
#2
#3
#4
schedules(destination:string,...):ScheduleResource
. . .
False Positives
«request» „departure airport“
«component» Service Discovery
global ontology resolves heterogeneity
annotated with
Airport
schema.org
annotated with

(c) Hypothesis: Annotating with semantic types from one global ontology resolves heterogeneity resulting in an effective service discovery.

Figure 5.1: Reducing Heterogeneity with Global Ontology

to identify similarities between them. The ultimate goal is to improve the effectiveness of service discovery by using semantic type annotations over the same ontology.

BaaS uses the Semantic Annotator to link parameters with semantic types, which is the matter of subject in this chapter. The Semantic Annotator automatically creates search queries from the syntactic API specifications and searches the global ontology for suitable semantic types. For this purpose, string-based matching techniques are used.

Improving the effectiveness of service discovery should not be at the expense of having a disproportionate overhead for adding semantic type annotations. Comparable approaches to semantic annotation of APIs like SWEET [8] are not effective, because they only consider a parameter and a semantic type as similar when their names are string-similar. As a result, search results produced by such kinds of approaches are precise but incomplete. Precision in this context is a metric of how many semantic types the discovery retrieves are relevant. For example when the semantic type `Airport` is correctly found when searching for the parameter name "departureAirport". The metric of incompleteness is called recall and measures how many relevant semantic types are not found, e.g., when searching for the keyword "accommodation" does not retrieve the relevant semantic type `Hotel`.

To counteract a low recall, the Semantic Annotator adds contextual information, so-called query parts, to the search query and so-called index fields to the search index. On one hand, adding contextual information to the search query and index improves the probability that there is a correct search hit, which would improve recall. On the other hand, the probability that many irrelevant semantic types are retrieved is increased, which would decrease precision.

In this chapter, the functionality of the Semantic Annotator is presented. It is shown which query parts and index fields should be used to achieve a high level of effectiveness. Furthermore, the effectiveness of different string-based and language-based matching techniques is tested. In this chapter, the Semantic Annotator is evaluated based on 100 REST API operations from RapidAPI that are aligned with the ontology schema.org. Furthermore, it is shown that the effectiveness of service discovery with the service matchmaker OWLS-MX3 is improved by 61% using semantic types.

5.1 Functionality of the Semantic Annotator

For the design of the Semantic Annotator, some preliminary considerations and design decisions must be made beforehand. An important design deci-

```
openapi: 3.0.1
paths:
  "/operations/schedules/{origin}/{destination}/{fromDateTime}":
    get:
      summary: Flight Schedules
      description: Scheduled flights between given airports on a given date.
      operationId: OperationsSchedulesFromDateTimeByOriginAndDestinationGet
      parameters:
      - name: origin
        in: path
        description: Departure airport. 3-letter IATA airport code (e.g.
        ↪  'ZRH')
        required: true
        schema:
          type: string
      - name: destination
        in: path
        description: Destination airport. 3-letter IATA airport code (e.g.
        ↪  'FRA')
        required: true
        schema:
          type: string
      - name: fromDateTime
        in: path
        description: Local departure date and optionally departure time
        ↪  (YYYY-MM-DD
          or YYYY-MM-DDTHH:mm). When not provided, time is assumed to be
          ↪  00:01
        required: true
        schema:
          type: string
          example: 2019-06-11T19:23
```

Listing 2: Lufthansa Open API

sion concerns the level of granularity of the annotations. Requests/offers can be annotated on different granularity levels: on the level of services, operations, or individual input/output parameters. Annotating at the level of APIs is sufficient when the number of relevant APIs needs to be narrowed down roughly and the deeper inspection for relevancy can be done manually by the requester. To enable highly automatized composition of executable mashups, fine-grained annotations on the level of parameters are necessary, because it is the parameter values that are shared between APIs in a mashup. This is the reason why requests/offers are annotated at the level of input and output parameters.

Input parameters of REST APIs often have primitive types such as String

```json
{
   "ScheduleResource": {
      "Schedule": [{
         "TotalJourney": {
            "Duration": "PT1H5M"
         },
         "Flight": {
            "Departure": {
               "AirportCode": "PAD",
               "ScheduledTimeLocal": {
                  "DateTime": "2019-08-12T09:25"
               }
            },
            "Arrival": {
               "AirportCode": "MUC",
               "ScheduledTimeLocal": {
                  "DateTime": "2019-08-12T10:30"
               },
               "Terminal": {
                  "Name": 2
               }
            },
            "OperatingCarrier": {
               "AirlineID": "CL"
            },
            "Equipment": {
               "AircraftCode": "CR9"
            }
         }
      }]
   }
}
```

Listing 3: Example JSON response of the Lufthansa OpenAPI

with a single value. In these cases, it is enough to annotate such parameters with a single semantic type to capture its semantics. On the contrary, especially output parameters usually have complex values, e.g., complex JSON or XML documents. The responses of REST APIs are typically complex JSON documents, consisting of nested attributes. In such cases, one needs multiple semantic types to express the semantics of the whole JSON document.

Mashups usually do not need the whole JSON document to process but only some isolated attributes. Therefore, the mashups need to query and extract the attributes from the JSON documents that they need. In a later phase of the BaaS, the Parameter Matcher must be able to match individual parameters from different APIs. Therefore, the annotations have to be added in this earlier phase by the Semantic Annotator.

Listing 2 shows the excerpt of the OpenAPI specification of the Lufthansa API, where the input parameters of the operation *OperationsSchedulesFromDateTimeByOriginAndDestinationGet* are defined. Listing 3 shows an excerpt of an example response of the same operation. This example response is also part of the OpenAPI specification.

Now, as the preliminary considerations have been made, the remainder of this section addresses the actual functionality of the Semantic Annotator. The Semantic Annotator casts the problem of finding relevant semantic types for certain input and output parameters as an information retrieval problem. A parameter has the following properties:

- name,
- direction (in or out),
- textual description,
- syntactic datatype (e.g. string),
- data format (e.g. an RFC 5322 [115] conforming email address),
- JSON path if it is part of a JSON response message,
- operation name

Once in the beginning, the Semantic Annotator indexes all semantic types from the global ontology. Subsequently, the provider/requester selects an input or output parameter they want to annotate, whereupon the Semantic Annotator automatically builds a corresponding search query to obtain relevant semantic types. Each input or output parameter is processed one at a time. The input of the Semantic Annotator is a single input or output parameter specification extracted from the offer/request. The output of the Semantic Annotator is a ranked list of `owl:Classes` ordered by their relevance. The relevance of the semantic types is determined based on TF-IDF (cf. Section 2.4). These semantic types recommended by the Semantic Annotator must be manually approved by the provider/requester.

The more search terms the search query and the index have in common, the higher the TF-IDF score. In general, adding more contextual information to the query or index usually increases recall, but comes at the expense of lower precision. Therefore, the terms that are used to build the search query and the terms that are indexed must be selected in a way such that precision and recall are maximized at the same time. In the following, the information that the Semantic Annotator includes in the index is called *index fields* and the information that is included in the query is called *query parts*. Candidate index fields and query parts are described in Section 5.1.2 and Section 5.1.3. The effectiveness of choosing different combinations of index fields and query parts for aligning offers from RapidAPI to the global ontology schema.org is evaluated in Section 5.2.1.

The TF-IDF score can be falsely low when the strings in the index fields and query parts vary slightly. For example, if the search query is “airports” and the index only contains the term “airport”, the TF-IDF score is zero. This problem can be avoided by preprocessing the terms of the search query and the terms being indexed to reduce them to a common stem (cf. Stemming in Section 2.4). However, if the stemming is too strong, unrelated terms may be wrongfully reduced to the same stem, which would again decrease precision. Therefore, the preprocessing must be configured in a way, such that recall and precision are balanced. Preprocessing of the Semantic Annotator is described in Section 5.1.1. The effectiveness of different preprocessing configurations is evaluated in Section 5.2.1 based on offers from RapidAPI and the global ontology schema.org.

5.1.1 Preprocessing

There are two kinds of index fields and query parts: Identifiers and textual descriptions. Identifiers and descriptive text have different requirements for a preprocessing pipeline which is why the Semantic Annotator has specialized pipelines for each.

These pipelines first split the input strings into tokens. Description texts mainly consist of whole natural-language sentences and contain punctuation. Since the individual words are separated by spaces, the space character and punctuation are usually used as the word delimiter for tokenization (cf. Section 2.4). In contrast to descriptive texts, identifiers contain fewer characters and have a higher information density than descriptive texts. As identifiers are used to instruct or communicate with computers, they conform to strict, standardized grammatical rules as defined in URI [33], JSON [35], Java, etc. For example, the OpenAPI standard recommends following the naming conventions of common programming languages for using identifiers [40, 4.7.10 Operation Object]. These naming specifications prohibit the use of spaces as a word delimiter, which is why other delimiters have to be used. Therefore, API providers often use the camel case or snake case naming convention for operation and parameter names, because this allows the individual words to be human-readable even without spaces or punctuation. With the camel case convention, delimiters are capital letters (e.g. "CamelCase") and with the snake case convention, the delimiter is the underscore (e.g. "snake_case"). Camel case or snake case naming conventions can be used for the tokenization of identifiers: In the case of camel case identifiers, strings can be split into tokens where the case of letters changes. In the case of snake case, non-alphanumeric delimiters can be used as token delimiters, e.g., the underscore character.

Later steps in the preprocessing pipeline may add, remove, or manipulate tokens. In particular, synonyms are added into the token stream, stop words are removed, and tokens are manipulated by stemming (cf. Section 2.4). It has to be analyzed which steps are effective.

5.1.2 Indexing Semantic Types

In a preparatory step, the Semantic Annotator first reads in all the `owl:Classes` from the global ontology and creates an index over all semantic types of the global ontology. A semantic type may have labels (`rdfs:label`) and a descriptive comment (`rdfs:comment`). Table 5.1 shows the raw contents of the index fields name, label, and comment for the semantic type `schema:Flight` next to tokens extracted by the preprocessing pipeline. Thus, using the label and the comment for indexing would mean that `schema:Flight` is found under the terms "flight" and "airline". The stop word "an" is an eliminated stop word and not added to the index.

5.1.3 Building Search Queries

As described earlier, the goal is to annotate single attributes in the JSON documents which is a requirement for the Parameter Matcher. A JSON document consists of a nested object structure, arrays, and the primitive data

Index Field	Sample Input	Indexed Tokens
Identifier	Flight	flight
Comment	An airline flight.	flight, airline

Table 5.1: Index fields and indexed tokens of the class `schema:Flight`

types String, Number, Boolean [35]. The Semantic Annotator breaks whole JSON documents down into a flat list of primitive typed attributes. All flattened attributes in a JSON document can then be annotated with a semantic type. The annotation on such a fine-grained level later allows the generation of program code that extracts only the individual data values from the JSON messages that need to be exchanged between the APIs.

To uniquely address single attributes of a complex JSON message and associate them to semantic types, the Semantic Annotator uses JSONPath[1]. JSONPath is a query language that can be used to access individual objects or values in a JSON document. The dollar operator ($) is used to access the root node of a JSON document. Dot notation (.) is used to access child nodes. Square brackets with integer indexes are used to access items in arrays. Using a wildcard (*) instead of an integer index returns all items of an array. Operators can be used to shape expressions to navigate over the JSON document. For example, evaluating the JSONPath *$.ScheduleResource.Schedule.[*].Flight.Departure.AirportCode* on the JSON response shown in Listing 3 yields *["PAD"].*

Table 5.2 shows the flat list of input and output parameters extracted from the JSON response shown in Listing 3. The input parameters shown in Table 5.2 have primitive types, which is why a JSONPath is not available.

Table 5.3 shows possible query parts that are investigated in this work. It has to be shown, which query parts are effective. The table also shows a JSONPath addressing the respective query part in an OpenAPI specification. The different query parts are described in the remainder.

Parameter Ids are the identifiers of the parameters. Occasionally, parameter IDs are abbreviations from which the original meaning of the parameter can be hardly understood. For example, the API endpoint graph.facebook.com/v15.0/{aid}/photos has the parameter *aid* that is an abbreviation for album id. This is why parameter ids are generally not sufficient

[1]`https://goessner.net/articles/JsonPath/`

Name	Direction	JSON Path
origin	Input	n/a
destination	Input	n/a
fromDateTime	Input	n/a
Duration	Output	$.ScheduleResource.Schedule.[*].TotalJourney.Duration
AirportCode	Output	$.ScheduleResource.Schedule.[*].Flight.Departure-.AirportCode
DateTime	Output	$.ScheduleResource.Schedule.[*].Flight.Departure.-ScheduledTimeLocal.DateTime
AirportCode	Output	$.ScheduleResource.Schedule.[*].Flight.Arrival.-AirportCode
DateTime	Output	$.ScheduleResource.Schedule.[*].Flight.Arrival.-ScheduledTimeLocal.DateTime
Name	Output	$.ScheduleResource.Schedule.[*].Flight.Arrival.Terminal-.Name
AirlineID	Output	$.ScheduleResource.Schedule.[*].Flight.OperatingCarrier.-AirlineID
AircraftCode	Output	$.ScheduleResource.Schedule.[*].Flight.Equipment.-AircraftCode

Table 5.2: List of extracted inputs and outputs from the Lufthansa API specification.

to use for searching relevant semantic types. Such abbreviations are sometimes written out in the query part *Parameter Description*.

The query part *JSONPath* uniquely identifies a parameter in a JSON document, e.g. a response message. The last segment of the *JSONPath* is identical to the *Parameter Id*. The intuition behind the *JSONPath* is that it may contain contextual information about the parameter under consideration to further improve the search for semantic types. This can be useful if the *Parameter Id* itself cannot be mapped to a semantic type, but another segment of the *JSONPath* can. The intuition behind using *Operation Id*, *Operation Description*, and *API Description* is just to add additional context from the API specification for the same reason as with the *JSONPath*.

Table 5.4 shows the query parts for the output parameter *AirportCode* of the Lufthansa OpenAPI specification. The query string created from the

Parameter Id	
Description	The identifier of an input or output parameter.
OpenAPI JSONPath	*$.paths.[*].[*].parameters.[*].name*
Parameter Description	
Description	The textual description of the input or output parameter.
OpenAPI JSONPath	*$.paths.[*].[*].parameters.[*].description*
JSONPath	
Description	The JSONPath of an input/output parameter if it is a complex JSON document and an example is given. Note that the last segment of the JSONPath is also the Parameter Id.
OpenAPI JSONPath	*$.paths.[*].[*].responses.[*].content.-['application/json'].example*
Operation Id	
Description	The identifier that encloses the input or output parameter.
OpenAPI JSONPath	*$.paths.[*].[*].operationId*
Operation Description	
Description	The description of the enclosing operation.
OpenAPI JSONPath	*$.paths.[*].[*].description*
API Description	
Description	The general textual description of the whole API that contains the respective input/output.
OpenAPI JSONPath	*$.info.description*

Table 5.3: Query parts

Parameter Id contains the tokens *airport* and *code*. The index created for the semantic type `schema:Airport` also contains the token *airport*. Consequently, the semantic type is correctly given a higher TF-IDF value. On the other side, the same holds for the irrelevant semantic type `schema:MedicalCode` because it shares the token *code*. This illustrates the difficulty of choosing index fields and query parts and the pipeline configuration in such a way that precision and recall are high at the same time. In the following evaluation, the effectiveness of index fields, query parts, and pipeline configurations is systematically tested.

Raw Input	Preprocessed Tokens
Parameter Id:	
AirportCode	airport, code
JSONPath:	
$.ScheduleResource.Schedule.[*].-Flight.Arrival.AirportCode	schedule, flight, code, resource, arrival, airport
Parameter Description:	
n/a	n/a
Operation Id:	
OperationsSchedulesFromDateTimeBy-OriginAndDestinationGet	date, schedule, origin, get, destination, time, operation
Operation Description:	
n/a	n/a
API Description:	
As for now, we do openly share: General aircraft, airport, airline, language, country information Operational information on schedules, flight status Lounge and seat map information Lufthansa Cargo order status and schedules	country, flight, lounge, aircraft, operational, language, airport, seat, general, schedule, lufthansa, openly, now, share, information, airline, cargo, map, status, order

Table 5.4: Query parts for the output parameter *AirportCode* of the Lufthansa API

5.2 Evaluation

In the following section, the effectiveness of the Semantic Annotator is first evaluated. The source code of the Semantic Annotator and the evaluation artifacts are publicly available on GitHub[2]. To this end, two experiments are performed. In the first experiment, different combinations of index fields and query parts are tested to determine which combination is most effective. In the second experiment, different configurations of the preprocessing pipeline are tested with different tokenizers, stemmers, and sources for synonyms. The second part of the evaluation is dedicated to the effectiveness of service dis-

[2]`https://github.com/brokerage-as-a-service/baas`

covery. It is examined whether the effectiveness of the service matchmaker OWLS-MX3 can be increased if semantic types are used instead of syntactic types. In short, the following three research questions are examined:

R1 Which combination of index fields and query parts is most effective?

R2 Which configuration of the preprocessing pipeline is most effective?

R3 Do semantic types improve the effectiveness of service discovery performed by OWLS-MX3?

5.2.1 Effectiveness of Semantic Annotator

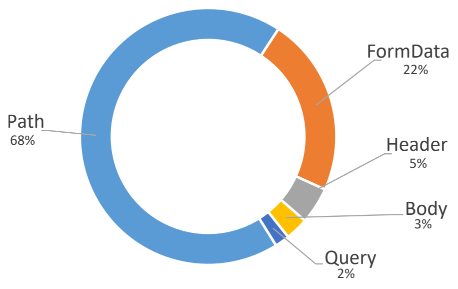
The following section shows how the effectiveness can be optimized by systematically testing different configurations of the Semantic Annotator. The section is divided into the preparation, the execution of the experiments, and the results.

Preparation

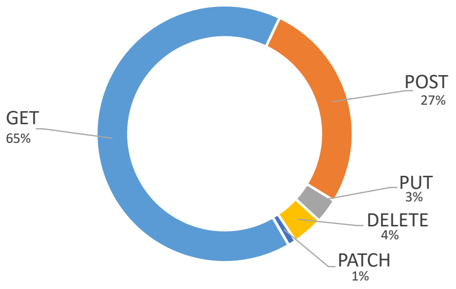
The data set used for the evaluation is based on real-world API specifications from RapidAPI and the ontology schema.org, whose semantic types are used for annotating APIs. Before the actual evaluation, the RapidAPI data set and the ontology schema.org are described in more detail below. Subsequently, the Mean Average Precision (MAP) is introduced as a metric to assess the effectiveness of the Semantic Annotator. To calculate MAP, a ground truth is required that defines which semantic types are actually relevant for a given parameter. In the remainder, the manual procedure for creating the ground truth is described.

RapidAPI Dataset As of November 2024, there are 1,862 public APIs registered at RapidAPI for which an API specification is available. These 1,862 APIs have 8,516 operations in total. All the operations have 20,212 inputs and 84,822 outputs in total.

An analysis of the parameters shows that some query parts are not available for all input and output parameters: Certain query parts are mandatory, e.g., the name of a parameter, others are optional. Using certain features of the specification as query parts is only useful if they are frequently used by the



(a) Input parameter locations



(b) HTTP methods

Figure 5.2: Dataset characteristics

providers. In the following, it is analyzed which of the query parts are frequently used by APIs found on RapidAPI: All of the 1,986 API specifications except one have an API Description. All 8,516 operations except for 19 have an Operation ID (99.8%).

Input parameters of APIs can be encoded in the URL path, as URL query parameters, in the HTTP request body (e.g., as form data), or in the HTTP request header [40, 4.8.12.1 Parameter Locations]. The kind of input parameter that is most frequently used on RapidAPI is the path parameter (cf. Figure 5.2a). Only 3% of the input parameters are complex JSON documents that are transmitted in the HTTP request body. Because a JSONPath is only available for JSON documents, this means that for the majority of input parameters, the query part JSONPath is not available.

APIs that implement REST correctly use the HTTP methods POST, GET, PUT, DELETE as CRUD operations, i.e., create, read, update, delete (cf. Section 2.1.2). According to REST, the GET operation is the only operation that is expected to return payload data. POST, PUT, and DELETE operations are not expected to return payload data. For this reason, GET operations are of particular interest for the composition of mashups because GET operations are used to read values from one API to pass them to another API. The majority of operations on RapidAPI are GET (cf. Figure 5.2b) operations.

For the API specification on RapidAPI, it is not possible to distinguish if an operation has no return value or if none has been specified. In fact, for 56.4% of all 8,516 operations, an example response message is given. 95.6% of all given example JSON response messages are well-formed according to RFC 4627 [35]. As a result, outputs can be extracted from JSON responses for more than half of the operations on RapidAPI. For all these outputs JSONPaths are available.

Usually, JSON attributes have a static key name and dynamic values, but it is also possible to have dynamic key names. In JSON schema [41], these kinds of object attributes are defined by so-called pattern properties. For example, the Open Aviation Data Service[3] encodes object identifiers into the JSON object attributes “SYN32”, “SYN33”, etc. In that case, the Semantic Annotator would match instance-level data with ontology-level data, which is not desired but cannot be prevented. A sampling check shows that pattern properties are rarely used.

Global Ontology Schema.org For the evaluation, the existing ontology schema.org is used to annotate APIs. An existing ontology is intentionally used to simulate the terminological heterogeneity between ontology and API specifications in a realistic BaaS scenario. A self-made ontology created specifically for this evaluation would have the bias that it is too tailored to the APIs on RapidAPI. This would then be expressed in exaggeratedly good values for precision and recall and would therefore reduce validity.

Schema.org is an initiative to create a standard ontology that can be used to structure data and was originally created by the biggest search engines Google, Bing, and Yahoo. An open community continuously improves schema.org. The original purpose of schema.org is to annotate contents on websites with schema.org concepts to improve the effectiveness of search engines. For its original purpose, schema.org is highly adopted in practice [116]. The schema.org core ontology is the foundation of schema.org, which consists of 607 classes and 861 properties from several domains like Creative Works, Health and Medical, Places, etc. In this chapter, it is evaluated if schema.org is also suitable to improve the effectiveness of service discovery.

Metrics A widely accepted measure to assess the effectiveness is the Mean Average Precision, which is defined as follows:

$$\text{MAP} := \frac{\sum_{r=1}^{|Q|} \text{AP}(r)}{|Q|} \tag{5.1}$$

[3]`https://github.com/brokerage-as-a-service/baas/blob/master/de.upb.dbis.baas/data/apis/openaviationdata_operational-aviation-data.json`

where Q is the set of requests and the average precision (AP) is defined as:

$$\text{AP(r)} := \frac{\sum_{k=1}^{n} P(k) \times \text{rel}(k)}{\text{number of relevant items}} \tag{5.2}$$

where k is the rank, n is the total number of retrieved items, $P(k)$ is the proportion of true positives from rank 1 to k, and rel(k) is a function equaling 1 if the result at rank k is relevant, zero otherwise.

In order to be able to calculate the Mean Average Precision, it must be known which semantic types are relevant for a parameter. Therefore, in a preparatory step, the input and output parameters extracted from the API specifications are first annotated manually with semantic types to obtain a ground truth. The proceeding of how the ground truth is created is described in the following section.

Creating the Ground Truth At first, 100 out of 8,516 operations are selected at random to reduce the huge data set. All inputs and outputs of these 100 operations are now manually annotated with semantic types from schema.org. The only tool to find relevant semantic types is the search function that is provided on the schema.org website. Every single schema.org concept has its own web page. This website shows the superordinate classes in the class hierarchy and a descriptive comment on the class. In addition, the web page contains a table with all properties including the properties inherited from superclasses. The `rdfs:ranges` of all properties are given. References to other classes are hyperlinked, which allows one to navigate through the ontological structure of schema.org.

The 100 randomly selected operations have 258 inputs and 472 outputs. 124 inputs (48%) and 250 outputs (53%) can be mapped to schema.org concepts. Table 5.5 shows the manually added annotations for the operation *Operations-SchedulesFromDateTimeByOriginAndDestinationGet* of the Lufthansa API (cf. Listing 2, Listing 3).

Experiment 1: Effective Index Fields and Query Parts

There are two index fields and six query parts. In total, there are $(2^2 - 1) \cdot (2^6 - 1) = 189$ combinations of index fields and query parts. The experiment is repeated 189 times with each combination. In each iteration, semantic types are queried for 258 inputs and 472 outputs. The returned list

Direction	Name/JSON Path	Syntactic Type	Semantic Type
Input	origin	string	Airport
Input	destination	string	Airport
Input	fromDateTime	string	DateTime
Output	$.ScheduleResource.Schedule.[*].-TotalJourney.Duration	string	Duration
Output	$.ScheduleResource.Schedule.[*].Flight.-Arrival.AirportCode	string	Airport
Output	$.ScheduleResource.Schedule.[*].Flight.-Arrival.ScheduledTimeLocal.DateTime	string	DateTime
Output	$.ScheduleResource.Schedule.[*].Flight.-OperatingCarrier.AirlineID	string	Airline

Table 5.5: Annotation of parameters

Label	Comment	Parameter Id	JSONPath	Parameter Descr.	Operation Id	Operation Descr.	API Descr.	MAP
✓		✓						.279
✓			✓					.23
✓				✓				.092
✓					✓			.15
✓						✓		.139
✓							✓	.092

Table 5.6: Effectiveness of single query parts and index field *Label*

of ranked semantic types is compared with the ground truth and the MAP is calculated accordingly.

Label	Comment	Parameter Id	JSONPath	Parameter Descr.	Operation Id	Operation Descr.	API Descr.	MAP
✓		✓	✓	✓	✓			.377
✓			✓	✓	✓			.374
✓	✓	✓	✓	✓	✓			.359
✓		✓		✓	✓			.358
✓	✓		✓	✓	✓			.355
✓		✓	✓	✓		✓		.353
✓		✓	✓		✓			.35
✓		✓	✓	✓				.344
✓	✓	✓		✓	✓			.343
✓			✓	✓		✓		.341

Table 5.7: Top ten index field and query part configurations in terms of mean average precision (MAP)

Results of Experiment 1

Table 5.6 shows the effectiveness of querying the index field *Label* with the individual query parts. Table 5.7 shows the ten best combinations in descending order with respect to MAP. All top ten combinations contain the index field *Label.* The index field *Comment* is rather unimportant. *Parameter Id* (MAP=0.279) is the top performing query part. In second place is the *JSONPath* with a MAP of 0.23, which is more of a degradation, because the *Parameter Id* slightly performs better as the *JSONPath*, while the *Parameter Id* is the last segment of the JSONPath. *API Description* is not part of any of the top ten configurations. It becomes apparent that the MAP can be increased by combining query parts instead of using a single query part: The MAP of the best query part *Parameter Id* can be increased from 0.279 to 0.377 by adding the *JSONPath*, *Parameter Description* and *Operation Id.*

Experiment 2: Effective Preprocessing Pipeline

In the second experiment, different pipeline configurations are systematically tested with respect to MAP. A pipeline consists of a Word Delimiter Tokenizer, a Lowercase Transformation (X), a Stemmer (Y), and a Synonym Dictionary (Z). For every individual pipeline step X-Z, different algorithms can be used or can be omitted completely. The configuration space of all possible pipelines is $X \times Y \times Z$. X, Y, Z are defined as follows:

$$\begin{aligned}
X &:= \{false, true\} \\
Y &:= \{\perp, EnglishMinimalStemmer, EnglishPossessiveStemmer, \\
&\quad\; PorterStemmer, KStem\} \\
Z &:= \{\perp, ConceptNetSynonyms, ConceptNetRelatedTo\}
\end{aligned}$$

This implies $|X \times Y \times Z| = 30$ configurations of the preprocessing pipeline.

The pipeline is either tested with the Lowercase Transformation or without ($\perp$). Furthermore, the pipeline is tested without a stemmer ($\perp$) or one of the following stemming algorithms: The English Minimal Stemmer [73], English Possessive Stemmer, Porter Stemmer [74], and KStem [75] (cf. Section 2.4)

The pipeline is tested without any synonyms ($\perp$) and with ConceptNet [117] synonyms (`conceptNet:/r/Synonyms`) and related terms `conceptNet:/r/RelatedTo`.

Results of Experiment 2

Table 5.8 shows the results of experiment 2. The best configuration uses the lowercase transformation and the English Minimal stemmer (MAP=.377). The lowercase transformation is the most effective part of the pipeline. Using the lowercase transformation alone without any other technique yields a MAP of 0.369. The additional gain from the English Minimal stemmer is therefore not significant. Adding synonyms or related words from ConceptNet only degrades the effectiveness of the lowercase transformation. This can be explained by the fact that adding synonyms to the token stream produces more false positives than true positives on average.

Lowercase	English Minimal Stemmer	English Possessive Stemmer	Porter Stemmer	KStemm	ConceptNet Synonyms	ConceptNet Related To	MAP
							.11
					✓		.11
						✓	.11
	✓						.114
	✓				✓		.114
	✓					✓	.114
		✓					.11
		✓			✓		.11
		✓				✓	.11
			✓				.114
			✓		✓		.114
			✓			✓	.114
				✓			.11
				✓	✓		.11
				✓		✓	.11
✓							.369
✓					✓		.32
✓						✓	.335
✓	✓						.377
✓	✓				✓		.332
✓	✓					✓	.345
✓		✓					.369
✓		✓			✓		.32
✓		✓				✓	.335
✓			✓				.362
✓			✓		✓		.284
✓			✓			✓	.352
✓				✓			.37
✓				✓	✓		.322
✓				✓		✓	.339

Table 5.8: Performance of various preprocessing pipeline configurations in terms of mean average precision

5.2.2 Effectiveness of Service Discovery

This section analyzes how the use of syntactic and semantic types affects the effectiveness of service discovery with OWLS-MX3. OWLS-MX3 is used for evaluation for the following two reasons: 1. OWLS-MX3 was developed independently from BaaS by third party authors and the source code of OWLS-MX3 is not publicly available. By using an independent system, the evaluation results with respect to effectiveness are more meaningful. 2. OWLS-MX3 comes in different variants, all using different matching techniques. It is thus possible to understand to what extent the semantic type annotations affect the various matching techniques. BaaS provides a component for Semantic Service Discovery, which can be implemented using OWLS-MX3, for example. The extent to which the effectiveness of the semantic type annotations affects the effectiveness of the service discovery is only shown here using the example of OWLS-MX3 to show the general validity of the approach. The section is divided into the preparation, the execution of the experiments, and the results.

Preparation

The section presents the service matchmaker OWLS-MX3 and the evaluation environment in detail. Furthermore, the procedure of creating the data set that is used for evaluation is described. The API specifications from RapidAPI are in OpenAPI format, but OWLS-MX3 does not support this format. Therefore, the OpenAPI specifications need to be translated into OWL-S, and that procedure is presented in this section.

Service Matchmaker OWLS-MX3 To test the effectiveness of service discovery with respect to syntactic and semantic types, the service matchmaker OWLS-MX3 [20] is used. OWLS-MX3 is a hybrid matchmaker that combines logic-based, text similarity-based, and ontological structure-based matching techniques. OWLS-MX3 comes in several variants: OWLS-MX0, OWLS-MX3 (Structure), OWLS-MX3 TextSim(Cos), OWLS-MX3 (M2), OWLS-MX3 (M3). Using the different variants of OWLS-MX3, the effects of the semantic types on individual matching techniques can also be measured. The different variants are briefly explained below.

OWLS-M0 is limited to logic-based techniques only. It defines the following logical matching degrees:

Exact match API S exactly matches request R iff all input types of S are exactly the same input types of request R and all output types of R are exactly the same output types as in S: $\forall in : C \in Input_S \, \exists in : C' \in Input_R : C \equiv C' \wedge \forall out : D \in Outputs_S \, \exists out : D' \in Output_R : D \equiv D'$.

Plug-in match API S plugs into request R iff for all input types of S there is an input type of R that is equal or more general and for all output types of R there is an output type of S that is a least specific class (direct child): $\forall in : C \in Input_S \, \exists in : C' \in Input_R : C' \sqsubseteq C \wedge \forall out : D \in Output_R \, \exists out : D' \in Outputs_S : D' \in LSC(D)$, where $LSC(D)$ is the set of least specific classes of D.

Subsumed-by match Request R is subsumed by API S iff for all input types of S there is an input type of R that is equal or more general and for all output types of S there is an output type of R that is equivalent or least generic concept (direct parent). $\forall in : C \in Input_S \, \exists in : C' \in Input_R : C' \sqsubseteq C \wedge \forall out : D \in Output_R \, \exists out : D' \in Outputs_S : D' \equiv D \vee D' \in LGC(D)$, where $LGC(D)$ is the set of least generic concepts of D.

Logical Fail OWLS-MX returns a logical fail iff none of the other rules applies.

OWLS-MX3 TextSim (Cos) uses string-based techniques only. To determine string similarity, text documents are first derived from the API and request specifications. On one hand, these text documents contain the names of the semantic types of the input parameters and the names of all superclasses (transitive). The same is done for the output types. In addition to the type names, the textual description `owls:serviceDescription` of the API is included. Three separate TF-IDF scores are calculated for inputs, outputs, and the API description. The three single TF-IDF scores are then aggregated by computing their average.

OWLS-MX3 (Structure) This OWLS-MX3 variant uses only the ontology structure-based matching technique proposed by [118] which is defined as follows:

$$sim_{dist}(C_R, C_S) := \begin{cases} e^{\alpha l} \cdot \frac{e^{\beta h} - e^{-\beta h}}{e^{\beta h} + e^{-\beta h}} & C_R \neq C_S \\ 1 & C_R = C_S \end{cases}$$

where l is the distance of the shortest path between class C_R and class C_S within their ontological structure. The variable h is the distance to their direct common subsumer class. The factors α and β are weights for l and h respectively. In OWLS-MX3, these factors are set to $\alpha = 0.2$ and $\beta = 0.6$, which was determined in previous experiments performed by the authors of OWLS-MX3.

To compute the similarity of whole sets of input types, the following formula is used:

$$sim_C(A, B) := \frac{1}{|A|} \cdot \sum_{a \in A} max\{sim_{dist}(a, b) \mid b \in B\}$$

where A is the set of input types of the API S and B is the set of input types of request R. Accordingly, $sim_C(A, B)$ is computed for the set of output types. Thus, the overall structural similarity sim_C of two sets of classes A, B is the average of the maximum distance from any class a in set A to any class b in set B.

Furthermore, the structural similarity metric also takes the number of inputs and outputs into account:

$$sim_M(A, B) := 1 - \frac{||A| - |B||}{max\{|A|, |B|\}}$$

These two similarity values $sim_C(A, B)$ and $sim_M(A, B)$ are aggregated separately for inputs and output types by using the following formulas:

$$\begin{aligned} sim_{S,in}(R, S) &:= \gamma \cdot sim_C(S_{in}, R_{in}) + (1 - \gamma) \cdot sim_M(S_{in}, R_{in}) \\ sim_{S,out}(R, S) &:= \gamma \cdot sim_C(S_{out}, R_{out}) + (1 - \gamma) \cdot sim_M(S_{out}, R_{out}) \end{aligned}$$

In OWLS-MX3, γ is set to 0.5, which was determined in previous experiments performed by the authors of OWLS-MX3.

This finally gives the overall structural similarity:

$$sim_{struct}(R, S) := \frac{sim_{S,in}(R, S) + sim_{S,out}(R, S)}{2}$$

OWLS-MX2 (M3) OWLS-MX2 is the predecessor of OWLS-MX3. OWLS-MX2 uses a fixed set of logic-based (OWLS-MX0) and text similarity-based matching techniques (OWLS-MX3 TextSim (Cos)).

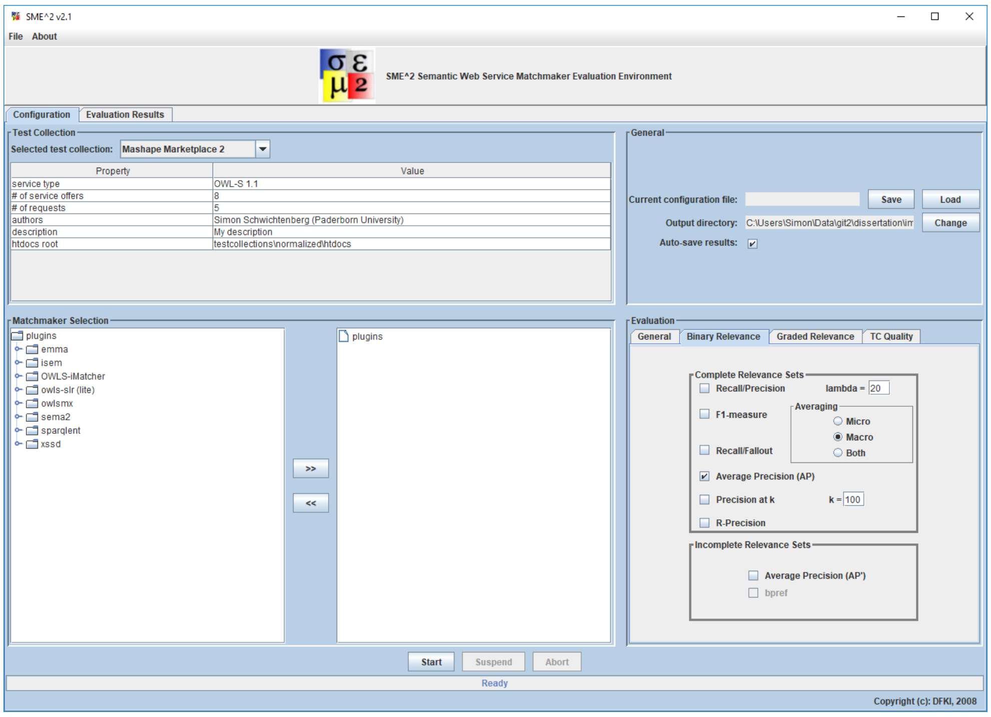
OWLS-MX3 (M3) OWLS-MX3 combines logic-based (OWLS-MX0), text similarity-based (OWLS-MX3 TextSim (Cos)), and ontological-structure matching techniques (OWLS-MX3 (Structure)). The output score values of the single matching techniques are encoded as a feature vector. A State Vector Machine (SVM) classifier is trained on a training set[4]. This classifier is used to determine if an API matches a request.

Semantic Service Matchmaker Evaluation Environment SME2 To assess the effectiveness of service discovery, the evaluation proceeding of the Semantic Service Selection (S3) contest [119] is adopted in this dissertation. While the goal of the S3 contest is to compare the retrieval performance of various service matchmakers, the goal of this evaluation is to investigate whether a single matchmaker, i.e., OWLS-MX3 is more efficient when semantic types are used.

The S3 contest uses the Semantic Service Matchmaker Evaluation Environment (SME2) to assess the performance of service matchmakers. The SME2 tool evaluates different measures, i.e., the retrieval performance, query response time, memory consumption, etc. The retrieval performance is the focus of this thesis, as it allows us to evaluate the impact of using semantic types on the effectiveness of service discovery. To evaluate the retrieval performance of the OWLS-MX3 service matchmaker, the mean average precision (cf. Equation 5.1) is used. Mean average precision is a widely accepted measure to assess the retrieval performance of service matchmakers.

SME2 requires a data set, a so-called test collection, which is used for the experiments. A test collection comprises a set of requests, a set of offers, and a binary relevance set that defines whether a specific offer is relevant to a request. The test collection used in the S3 contest is the OWL-TC[5].

[4]The training set is a 5% fraction of the OWLS-TC.
[5]`https://github.com/kmi/sws-test-collections`



Test Collection

Selected test collection: Mashape Marketplace 2

Property	Value
service type	OWL-S 1.1
# of service offers	8
# of requests	5
authors	Simon Schwichtenberg (Paderborn University)
description	My description
htdocs root	testcollections\normalized\htdocs

General

Current configuration file:

Output directory: C:\Users\Simon\Data\git2\dissertation\im

Auto-save results: ✓

Matchmaker Selection

plugins
- emma
- isem
- OWLS-iMatcher
- owls-slr (lite)
- owlsmx
- sema2
- sparqlent
- xssd

plugins

Evaluation

Complete Relevance Sets
- ☐ Recall/Precision lambda = 20
- ☐ F1-measure
- Averaging: ○ Micro [● Macro] ○ Both
- ☐ Recall/Fallout
- ✓ Average Precision (AP)
- ☐ Precision at k k = 100
- ☐ R-Precision

Incomplete Relevance Sets
- ☐ Average Precision (AP')
- ☐ bpref

Copyright (c): DFKI, 2008

Figure 5.3: SME2 user interface

Figure 5.3 shows the user interface of the SME2 tool. There are two tabs at the top. The tab *Configuration* is to configure the evaluation experiment. The test collection used for the evaluation experiment can be selected from a drop-down menu. The table below shows some characteristics of the selected test collection. The left part of the Matchmaker Selection shows the available service matchmakers. With the aid of the arrow buttons, service matchmakers can be selected or deselected for evaluation. Evaluation metrics can be selected in the section *Evaluation*. The start button initializes the evaluation process. A run of SME2 matches all requests with all offers pairwise.

RapidAPI Test Collection The OWL-TC test collection cannot be used in the course of this evaluation because it does not contain any offers with purely syntactic types. Therefore, the evaluation is conducted on the basis of real-world API specifications from RapidAPI. For experiment 1 and 2, 100 operations have been randomly sampled from all the 9,658 operations that are available on RapidAPI. On the basis of these 100 operations, two individual sets of offers are created: These sets contain 100 offers with 1 operation, cor-

Request:	distance
Signature:	$(schema : GeoCoordinates, schema : GeoCoordinates) \rightarrow schema : Distance$
Description:	Returns the distance between two geographic coordinates.
Request:	flight
Signature:	$(schema : Airport, schema : Airport) \rightarrow schema : Flight$
Description:	Returns flights connecting a departure airport and an arrival airport.
Request:	movie
Signature:	$schema : Person \rightarrow schema : Movie$
Description:	Returns the movies the given person (actor) plays a character in.
Request:	music
Signature:	$schema : MusicGroup \rightarrow schema : MusicAlbum$
Description:	Returns albums of a given music group.
Request:	sports
Signature:	$schema : SportsTeam \rightarrow schema : SportsEvent$
Description:	Returns matches of a given sports team.

Table 5.9: Five OWL-S requests used for the evaluation

responding to the 100 operations randomly selected from RapidAPI. In the first set O, parameters have syntactic types like in the original specification from RapidAPI. In the second set O', parameters have semantic types from schema.org according to the ground truth that was created before.

The set of requests R comprises five requests. Every request contains exactly one operation whose parameters have semantic types from schema.org. These requests are shown in Table 5.9. For every request-offer pair, the relevance set contains a Boolean value that indicates whether the offer is relevant to the request.

OpenAPI to OWL-S Transformation The offers from RapidAPI are specified in OpenAPI format, but OpenAPI is not supported by OWLS-MX3. Thus, the offers have to be syntactically translated from OpenAPI to OWL-S which is supported by OWLS-MX3. This section describes how the contents of the OpenAPI specifications are translated into OWL-S. Figure 2.3 and Figure 2.5 show the parts of OpenAPI and OWL-S that are affected by the transformation.

In OWL-S, a service is represented by an `owls:Service`. An `owls:Service` is presented by a `owls:ServiceProfile` which has an `owls:Process`. An `owls:AtomicProcess` is an `owls:Process` that executes in a single step and is directly invokable as it corresponds with the actual operation of the service implementation. An `owls:AtomicProcess` has `owls:Input` and `owls:Output` parameters. The parameter type of an `owls:Parameter` is the URL of a semantic type.

The transformation maps `oas:OpenAPI` to `owls:Service` and `oas:Operations` onto `owls:AtomicProcesses`. Operations in OpenAPI specifications have primitive data types. These primitive data types are translated into classes so they can be used in an OWL-S specification. The classes created from primitive types do not have any relation to other classes.

`oas:Parameters` with simple types are mapped to an `owls:Input` and corresponding semantic type. Complex JSON message types are decomposed into a flat list of primitive-typed parameters. For each of these primitive typed attributes, an `owls:Input` or `owls:Output` is created. This approach is related to those presented in [120] and [16]. Listing 4 shows an excerpt from the OWL-S specification that is the result of transforming the OpenAPI specification shown in Listing 2.

The second test collection is identical to the first one, except that the syntactic types are replaced by semantic types according to the ground truth. To translate the syntactic to semantic types, the ground truth from the first two experiments is used. Those parameters that could not be mapped to a corresponding semantic type keep their syntactic type.

Listing 4 shows an exemplary OWL-S specification for an offer from RapidAPI. In the first set of offers, the parameter *origin* has the data type `oas:string`. In the second set, the data type is replaced by the semantic type `schema:Airport`[6].

Experiment 3: Syntactic Types Versus Semantic Types

OWLS-MX3 comes in five variants, there are two sets of offers O and O', one set of requests R, and one relevance set. These can be combined in 10 different ways, resulting in 10 variations of the experiment. Table 5.10 shows the configuration of all 10 variations.

[6]During evaluation runs of SME2, all ontologies are served at and read from the local host

```
<process:AtomicProcess
↪  rdf:ID="OperationsSchedulesFromDateTimeByOriginAndDestinationGet">
    <service:describes rdf:resource="#Service" />
    <process:hasInput rdf:resource="#origin" />
    <process:hasInput rdf:resource="#destination" />
    <process:hasInput rdf:resource="#fromDateTime" />
    <process:hasOutput rdf:resource="#Duration" />
    <process:hasOutput rdf:resource="#AirportCode" />
    <process:Input rdf:ID="origin">
        <process:parameterType
        ↪  rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://127.0.0.1/ontology/openapi.owl#string
        </process:parameterType>
        <rdfs:label>origin</rdfs:label>
    </process:Input>
    <process:Output rdf:ID="AirportCode">
        <process:parameterType
        ↪  rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
            http://127.0.0.1/ontology/openapi.owl#string
        </process:parameterType>
        <rdfs:label>AirportCode</rdfs:label>
    </process:Output>
</process:AtomicProcess>
```

Listing 4: Lufthansa API in OWL-S and with syntactic types

#	OWLS-MX3 Variant	String	Structure	Logic	Types
1	OWLS-MX0			✓	Syntactic
2	OWLS-MX0			✓	Semantic
3	OWLS-MX3 TextSim (Cos)	✓			Syntactic
4	OWLS-MX3 TextSim (Cos)	✓			Semantic
5	OWLS-MX3 (Structure)		✓		Syntactic
6	OWLS-MX3 (Structure)		✓		Semantic
7	OWLS-MX2 (M3)	✓	✓	✓	Syntactic
8	OWLS-MX2 (M3)	✓	✓	✓	Semantic
9	OWLS-MX3 (M3)	✓	✓	✓	Syntactic
10	OWLS-MX3 (M3)	✓	✓	✓	Semantic

Table 5.10: OWLS-MX experiment configurations categorized according the classification scheme

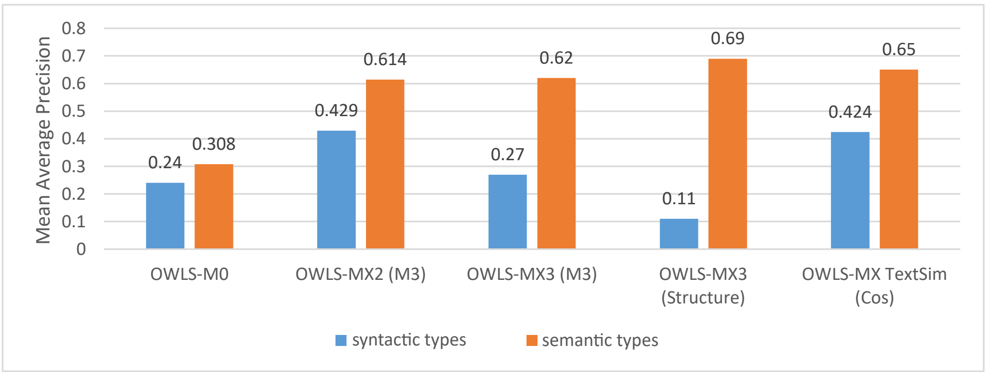


Figure 5.4: Comparison of OWLS-MX3's mean average precision

Results of Experiment 3

Figure 5.4 shows the effectiveness achieved in the 10 experiment variations. The y-axis shows the mean average precision (MAP) for each experiment. In the first experiment with syntactic types, the variant OWLS-MX2 (M3) exhibits the highest MAP (0.429). In contrast, variant OWLS-MX3 (Structure) exhibits the best performance with the operations associated with the semantic types from the global ontology (MAP = 0.69). This shows that the effectiveness of service discovery can be improved by 61% by using semantic types from a global ontology.

5.3 Discussion

Existing approaches for API annotation use only ontological classes, but no properties. In terms of this dissertation, it was also investigated if it is useful to use properties for annotating in addition to classes. Using the example of the operation *OperationSchedulesFromDateTimeByOriginAndDestinationGet* of the Lufthansa API and the ontology schema.org, the semantics of parameters can be described much more precisely if properties are also used for annotating. This operation returns flights between a departure airport and an arrival airport. The parameter for the departure airport is named *origin* and the parameter for the arrival airport is named *destination*. If parameters would only be annotated with a single ontological class from the ontology schema.org, both parameters would be annotated with the class *schema:Airport*. From

machine which is why the URLs of the ontologies start with the IP address 127.0.0.1.

the class annotations alone one cannot differentiate between both parameters. However, it may be crucial to differentiate between both parameters when they need to be wired in a mashup: A requester who creates a mashup of the Lufthansa and the Hertz rental car service must be able to recognize the difference between *origin* and *destination* because it is *destination* that must be wired with the input *pickupLocation* of the Hertz service.

Such differentiation is possible by annotating *origin* with the properties `schema:departureAirport` and *destination* with `schema:arrivalAirport` instead of annotating both with the class `schema:Airport`. The problem with using properties for annotating is that properties are ambiguous when their `rdfs:domains` or `rdfs:ranges` span multiple classes as is the case with schema.org.

To illustrate the problem, suppose that a triple with the predicate `schema:provider` is to be used to annotate a parameter *airlineID* in that sense that an airline is a provider of flights. The domain of `schema:provider` is `CreativeWork`, `Invoice`, `ParcelDelivery`, `Reservation`, `Service`, and `Trip`. Requesters that search for providers of `Flights` should not retrieve providers of `CreativeWorks`. To avoid such ambiguities and false positives matches, the specific domain and range must be given explicitly in the annotating triple.

`Flights` are also contained in the domain, because `Flight` is a subclass of `Trip`. The domain of `schema:provider` is `Organization` and `Person`. `Airlines` are also included in the range, because `Airline` is a subclass of `Organization`. The domains and ranges of `schema:provider` are shown in Figure 5.5.

Although one can capture the semantics much more precisely with triples, this approach is abandoned for the following reasons:

Incompatibility with Existing Service Matchmakers Existing service matchmakers like OWLS-MX3 do not support the annotation with triples and therefore cannot take advantage of the additional information. Although it would be possible to develop new approaches that can use the information from triples, this would require new similarity measures for triples, for example. Such new approaches also contradict the originally formulated requirement in Section 1.3.1 that BaaS should be upward compatible.

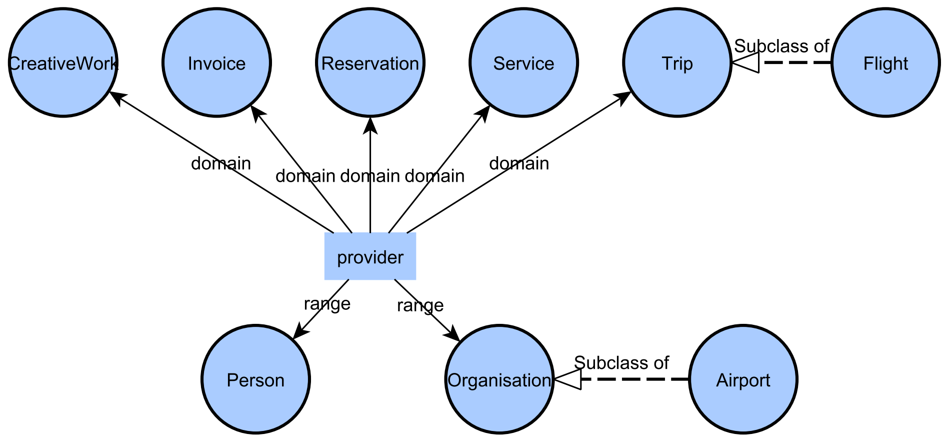


Figure 5.5: Domain and range of `schema:provider`

High Annotation Effort Annotating with triples is significantly more complex for the requesters and providers than annotating with a single ontological class since instead of one ontological class, three ontological concepts need to be used. Creating valid triples adds further overhead, since it always has to be checked whether the subject is in the domain of the predicate and the object is in its range.

Little Gain of Effectiveness Likewise, a precise description is not always desirable: The precise description can be helpful when it comes to being able to distinguish between a `departureAirport` and an `arrivalAirport`. However, since the various APIs and ontologies are so heterogeneous in terms of terminology, in practice it is rather unlikely that precise search results will be found in service discovery at such a fine-grained level.

5.4 Summary

In this chapter, the Semantic Annotator of the BaaS approach is presented. The BaaS approach first indexes the semantic types of the global ontology. Search queries are constructed from the API specifications with which the index is searched. As shown in the evaluation, the Semantic Annotator is most effective when indexing only the names of the semantic types and using the parameter names, the parameter description, and the operation name for building the search queries. The lowercase transformation is proven to be the most effective matching technique. Stemming and adding synonyms only is decreasing effectiveness. Furthermore, it is shown that the effectiveness of the service discovery can be increased by 61% using the service matchmaker OWLS-MX3 when semantic instead of syntactic types are used.

6 API Protocol Miner

When requesters integrate third-party APIs into mashups, they need to know their protocols such that they are able to call the operations in the right order. API specification languages like OpenAPI are not capable to specify API protocols. Instead, there are more sophisticated languages like OWL-S [47], WS-BPEL [100], and BPMN [121] that can describe complex control flows consisting of sequences, branches, loops, and other control flow constructs. In practice, API protocol specifications for REST APIs are virtually not existent at all. On the other hand, since REST APIs are usually black boxes for requesters, the requesters rely on the API protocols to integrate the APIs correctly into their mashups. Urging the API providers to specify API protocols is problematic because creating API protocols manually is time-consuming. Therefore, automatic assistance systems are desired to support the providers.

In general, there are two approaches for deriving API protocols semi-automatic: via static or dynamic analysis. Static analysis reads the API specification and discovers dependencies between operations. Dynamic analysis involves calling the REST API while it is running. This chapter compares the advantages and disadvantages of both approaches with each other.

The BaaS approach uses dynamic analysis to discover API protocols from call-logs using process mining [122, 123]. Every time an application calls an operation of an API, it leaves traces in a call-log. A call-log records the entire HTTP/S traffic between any client applications and an API. Recurring patterns of clients' usage behavior make it possible to draw conclusions about the underlying API protocol.

On the one hand there are the control flow constructs that can be described in languages like [47], WS-BPEL [100], and BPMN [121]. On the other side, there are process mining algorithms that are only capable to discover certain control flow constructs. In this chapter, the controls flow constructs needed to describe API protocols in OWL-S [47], WS-BPEL [100], and BPMN [121] are related to those control flow constructs that can be discovered by the process mining algorithms Alpha Miner [81], Heuristics Miner [124], and Inductive Miner [82]. All three algorithms are described in details in Section 2.5.

Under these restrictions of process mining, the functionality of the BaaS API Protocol Miner is introduced. Its main feature is to reprocess the call-logs so they can be used with process mining. This is required because there is a mismatch between the information provided through the call-log and the information that is required by process mining.

In the evaluation, it is shown which of the process mining algorithms Alpha Miner, Inductive Miner, and Heuristics Miner is best suited to discover API protocols from call-logs. Lastly, it is investigated how the quality of the mined API protocols compares to manually created API protocols.

This chapter is structured as follows: First, a real example is introduced, which is used in this chapter as a running example. Then the advantages and disadvantages of static or dynamic analysis are discussed. The control flow constructs of the specifications languages OWL-S, WS-BPEL, and BPMN are classified and compared with the control flow constructs that can be discovered using Process Mining. Afterward, the functionality of the API Protocol Miner is presented and in particular how call-logs can be prepared in such a way that they can be used for process mining. In the evaluation, it is shown that the Heuristics Miner is best suited for deriving API protocols from call-logs, as these API protocols are close to what a human would specify. This shows that process mining can be used to generate high-quality API protocols that require little post-processing from the API provider and thus save him a significant amount of time.

6.1 Running Example

The SIMPHERA API is used as a running example in this chapter and for evaluation. SIMPHERA[1] is a web-based, highly scalable solution for the simulation and validation of autonomous driving functions from the company dSPACE. Users of SIMPHERA can use an OpenAPI to integrate SIMPHERA into their development processes of their autonomous driving functions.

SIMPHERA can be deployed on various cloud providers such as Azure or AWS and runs in a Kubernetes[2] cluster. Nginx[3] is used as an ingress controller which accepts incoming traffic and load balances it to the internal components. By default, nginx creates call-logs for the incoming traffic. This shows that call-logs can be generated with little effort without making invasive changes to the application. Used in a BaaS scenario, SIMPHERA would also be operated behind a BaaS API gateway. In that case, the nginx ingress controller would be deployed on the API gateway.

The SIMPHERA API uses different URIs for different resources and uses standard HTTP operations. HATEOAS (cf. Section 2.1.1) is not supported which is the reason why the SIMPHERA API is on the REST maturity level 2 (cf. Section 2.1.2). The API is thus representative of the large class of REST APIs in which API protocols are not specified in any way. Expanding the API specification to include API protocols represents real added value here.

The order in which operations appear in the OpenAPI specification is up to the API provider and is quasi-random and does not reflect any information about the dependencies of the operations. Thus, a requester who wants to use the SIMPHERA API cannot know from the OpenAPI specification alone in which order to call its operations.

[1]`https://www.simphera.com/`
[2]`https://kubernetes.io/`
[3]`https://nginx.com/`

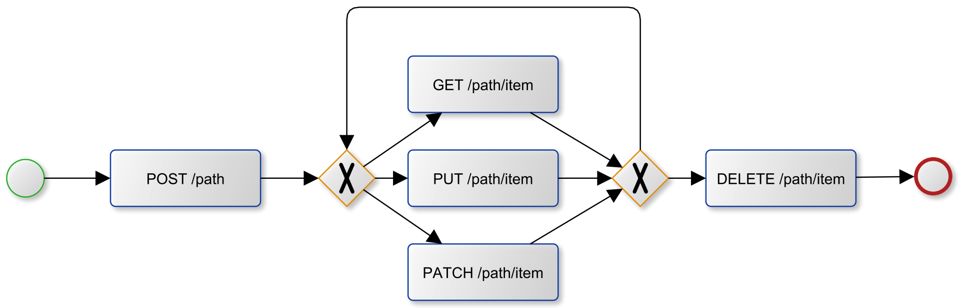


Figure 6.1: Logical dependencies between the methods POST, GET, PUT, PATCH, and DELETE

6.2 Static Versus Dynamic Analysis

In the context of this thesis, static analysis refers to the analysis of an (Open) API specification without actually calling the API, while dynamic analysis involves analyzing a REST API while it is running. Static and dynamic analysis can be used to derive the control and data flow between operations. In the remaining section, the advantages and disadvantages of static and dynamic analysis for deriving API protocols are analyzed.

6.2.1 Static Analysis

REST APIs use the POST, GET, PUT/PATCH, and DELETE methods to create, read, update, and delete resources. These CRUD (create, read, update, delete) operations have a logical call sequence: A resource can only be read, updated, or deleted after it has been created. Only once, a resource can be created or deleted. After a resource has been deleted, no further operations can be performed on it. A corresponding API protocol is shown in Figure 6.1.

Dependencies can also be derived from the URLs of REST APIs: URLs are structured hierarchically and this results in a logical sequence of dependencies. However, this method is not reliable because not all parameters are necessarily reflected in the URL, but are sent e.g. in the message body.

Bertolino et al. [13] present a more advanced information flow analysis for WSDL specifications. If Bertolino's approach is transferred to REST APIs, then there is a dependency between two operations a and b if a produces an output that b consumes. According to Bertolino et al. the output type of

```
1  paths:
2  "operation1":
3      get:
4        responses:
5          '200':
6            content:
7              application/json:
8                schema:
9                  "$ref": "#/components/schemas/MySchema"
10   "operation2":
11     get:
12       requestBody:
13         content:
14           application/json:
15             schema:
16               "$ref": "#/components/schemas/MySchema"
```

Listing 5: Shared JSON schema between operations

operation *a* and the input type of operation *b* need to be identical to detect a dependency between *a* and *b*. Transferred to REST APIs that means that the requests and responses of the operations must share the same JSON schema. Listing 5 illustrates the idea: The response of *operation1* and the request of *operation2* share the same JSON schema (cf. lines 9 and 16), which is why they are dependent. This pattern cannot be found either with SIMPHERA or with any other APIs on RapidAPI, because complex types are generally rarely defined for input parameters and the set of input types and output types are always disjoint.

A refinement of the method does not consider entire JSON schemas, but the individual attributes of the JSON messages. Thus, there would be a dependency between operation *a* and *b* if the output message of *a* has an attribute of the same name as an input of *b*. This pattern actually appears in the SIMPHERA API, but there are some problems with this approach: As soon as the names of the input and output parameters are not exactly the same, a dependency can no longer be created using this method. For example, some SIMPHERA operations use the parameter name *dataSetId* while others use the name *datasetId*. The different spellings can be normalized manually. In the above case, one could normalize the parameters *dataSetId* and *datasetId* to simply *datasetId* using the lowercase transformation.

```yaml
1  paths:
2    "/api/simphera/projects":
3      get:
4        responses:
5          '200':
6            description: Success
7            content:
8              application/json:
9                schema:
10                 type: array
11                 items:
12                   type: object
13                   properties:
14                     id:
15                       type: string
16                     description:
17                       type: string
18   "/api/simphera/projects/{projectId}":
19     get:
20       parameters:
21       - name: projectId
22         in: path
23         schema:
24           type: string
```

Listing 6: Shared inputs and outputs between operations

Listing 6 illustrates the principle on the basis of two SIMPHERA operations: The attribute *id* (cf. line 14) of the response message of the operation `GET /api/simphera/projects` needs to be normalized to *projectId* to make the data flow explicit. Figure 6.2 shows the resulting information flow dependencies of the SIMPHERA API operations. Every node represents an operation. Every edge represents the data flow of a single parameter. The resulting graph is very dense. The reason for this is that almost all SIMPHERA operations require the *projectId* as an input parameter and there are a lot of operations that return the *projectId* in their response messages.

In principle, it is possible to filter out certain parameters for the static analysis in order to simplify the analysis, such as the *projectID* in the SIMPHERA example. However, this approach hardly scales because the filtering is specific to each individual API and is therefore very time-consuming.

In addition, manual normalization of the parameter names is also very

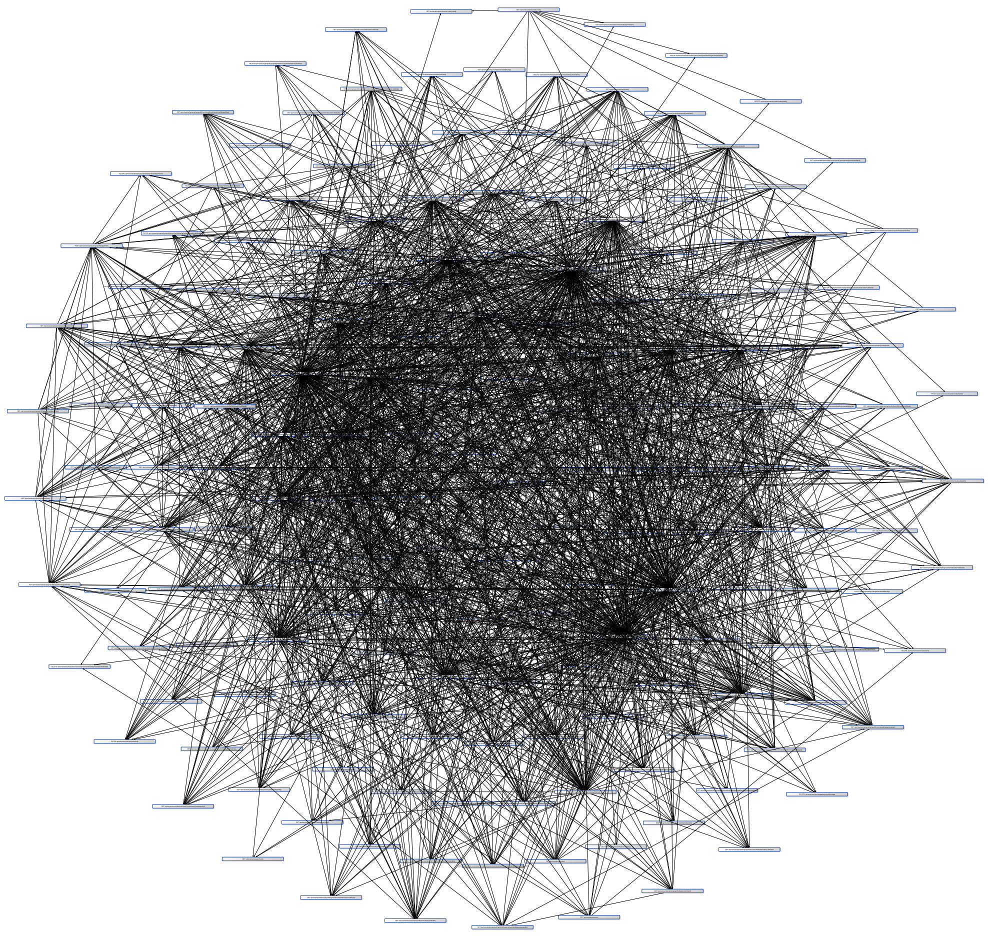


Figure 6.2: SIMPHERA API protocol produced by static analysis

time-consuming. REST APIs may have hundreds of parameters that need to be checked individually. A detailed understanding of the respective API is required to normalize the parameters. Otherwise, false dependencies between the operations can be modeled or actual dependencies can be missed at all. At most the API provider himself has a deep understanding of an API. However, it is impractical for service providers to perform normalization of their API and static information flow analysis to generate API protocol specifications because the process is too laborious.

6.2.2 Dynamic Analysis

Dynamic analysis considers the runtime behavior of an API. One way to perform dynamic analysis is through process mining. The input for the dynamic analysis is call-logs. Dynamic analysis has the advantage over static analysis in that parameter normalization is not required. However, the dynamic analysis requires that there is a client application implemented that uses the REST API whose API protocol is to be mined.

Process mining can only be used to discover process models if there are recurring patterns of behavior appearing in the call-logs. In practice, call-logs contain noise, i.e., random behavior. Noise is a disturbing source for process mining and may cause erroneous process models to be extracted. There are several sources of noise as explained in the following paragraphs.

Irregular Human Behavior REST APIs are used by computers or human users likewise. When the REST API is invoked via a graphical user interface, the call-log entries also reflect the behavior of the user. By using the user interface, certain activities can be carried out. Users can arbitrarily interrupt their current activity and continue with another activity. Consequently, the corresponding call-logs are very irregular and are less suitable for process mining. Conversely, when the API is called in a computer program, the calls are much more regular and generally better suited to process mining.

Deep Links Deep links allow you jumping directly to certain states of the client application while omitting the intermediate steps going to that state. Suppose the client application is a web application that runs in a web browser. Bookmarks can be created for individual dialogs in the web application. By opening the bookmark later, they are able to return to the dialog and application state. The moment when users open a bookmark marks a new process instance to be started. As a result, process mining would wrongfully recognize operation calls resulting from deep links as starting events.

Caching A cache is a fast buffer storage that prevents repeated read access on a slower medium of an output result. In the context of client-server architectures, the client caches HTTP responses in-memory on the client device. If the client calls the same operation with the same parameter

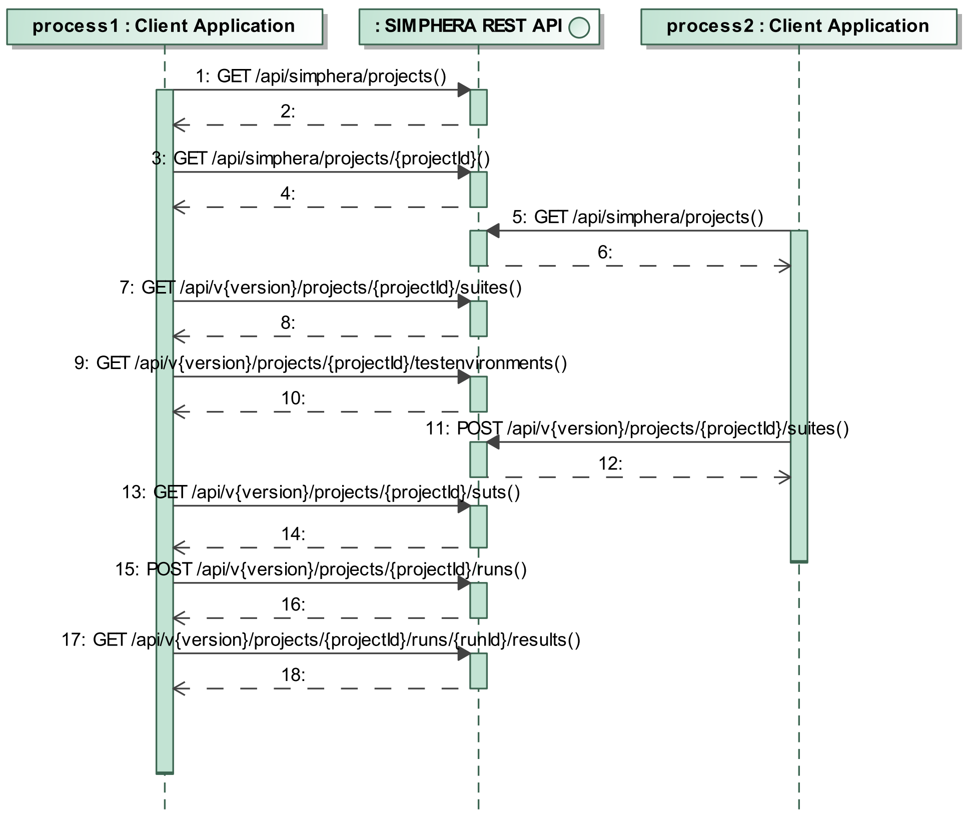

process1 : Client Application
: SIMPHERA REST API
process2 : Client Application
1: GET /api/simphera/projects()
2:
3: GET /api/simphera/projects/{projectId}()
4:
5: GET /api/simphera/projects()
6:
7: GET /api/v{version}/projects/{projectId}/suites()
8:
9: GET /api/v{version}/projects/{projectId}/testenvironments()
10:
11: POST /api/v{version}/projects/{projectId}/suites()
12:
13: GET /api/v{version}/projects/{projectId}/suts()
14:
15: POST /api/v{version}/projects/{projectId}/runs()
16:
17: GET /api/v{version}/projects/{projectId}/runs/{runId}/results()
18:

Figure 6.3: Interference of two process instances accessing the SIMPHERA API

values twice and it already has the response in the cache, the client reads the response from the cache instead of calling the operation again. Caching increases the performance of the client application: Every HTTP connection is afflicted with latency caused by different factors such as the bandwidth of the transmission medium, routing, storage delays, etc. This overhead can be eliminated when the response is read from local cache. In this case, only the client is aware of the operation call and no HTTP request is sent to the web server so that the call does not leave an event entry in a call-log on its way from the client application to the REST API.

Now that the sources of noise have been identified, the task is to eliminate the noise from call-logs. To reduce noise, it is essential that the calling process can be clearly assigned to each call in the call-log. Web servers are

capable to processes calls of multiple process instances concurrently. This means that calls from different processes may interfere and this interference generally occurs at random. Thus, two consecutive entries in the call-log are not necessarily created by the same process. To give an example, Figure 6.3 shows two interfering sessions that access the SIMPHERA API at the same time. The entries in the call-log are created at the time the requests arrive at the API. If the individual calls are not assigned to their calling process instance, it would look as if all calls are executed by the same process. In that case, process mining would try to generalize random behavior which would be useless.
To give an example, if process1 and process2 shown in Figure 6.3 cannot be distinguished in the call-log, process mining would wrongfully discover a dependency between `GET /api/v{version}/projects/{projectId}/testenvironments` and `POST /api/v{version}/projects/{projectId}/suites` which in reality does not exist.

Another disadvantage of dynamic analysis is that if applications use the API inefficiently, this will be reflected in the mined API protocols and may be adopted in the next mashups by future requesters. Furthermore, the dynamic analysis can only consider the behavior that can be seen in the call-logs. Consequently, only partial API protocols can be discovered. In literature, this issue is also referenced to as incomplete behavior [125]. Insofar as the applications do not use the full range of API operations, this unused part of the API protocol cannot be mined using process mining. As a result, the API logs mined using Process Mining are usually incomplete. Consequently, API protocols discovered through process mining are only approximations of the actual API protocols. The approximate API protocols could wrongfully forbid or allow certain operation call sequences. Wrongfully forbidden sequences could unnecessarily limit requesters when they use an API. Wrongfully allowed sequences lead to errors during the execution of the mashup. Table 6.1 compares the advantages and disadvantages of static and dynamic analysis.

Static Analysis	Dynamic Analysis
⊕ Complete API specifications are required and generally available	⊕ Existing applications are required and generally available
⊖ High effort to normalize parameter names	⊕ Low effort to gain call-logs
⊕ Covers all operations from specification	⊖ Covers only operations occurring in call-log

Table 6.1: Comparison between static and dynamic analysis of REST APIs

6.3 Discoverable Control Flow Constructs through Process Mining

API protocols may have complex control flows with sequences, branches, loops, etc. Specification languages that are capable to describe API protocols like OWL-S contain control flow constructs to specify complex control flows. On the other hand, the number of control flow constructs that process mining algorithms are able to discover is limited. This section opposes which kind of control flow constructs are present in specification languages and which can be discovered through process mining to identify the limitations of mining API protocols from call-logs.

6.3.1 Control Flow Constructs in Specification Languages

This section analyzes the specification languages OWL-S, WS-BPEL, BPMN regarding their control flow constructs. Following that, the differences and commonalities of the control flow constructs from different languages are analyzed and then generalized.

Control Flow Constructs in OWL-S

This paragraph describes how to specify API protocols in OWL-S. Figure 6.4 shows the language constructs of OWL-S that are relevant for specifying API protocols: `owls:AtomicProcesses` execute in a single step and are directly invocable as they correspond with the actual API operations. `owls:CompositeProcesses` are abstractions of multi-step processes that have

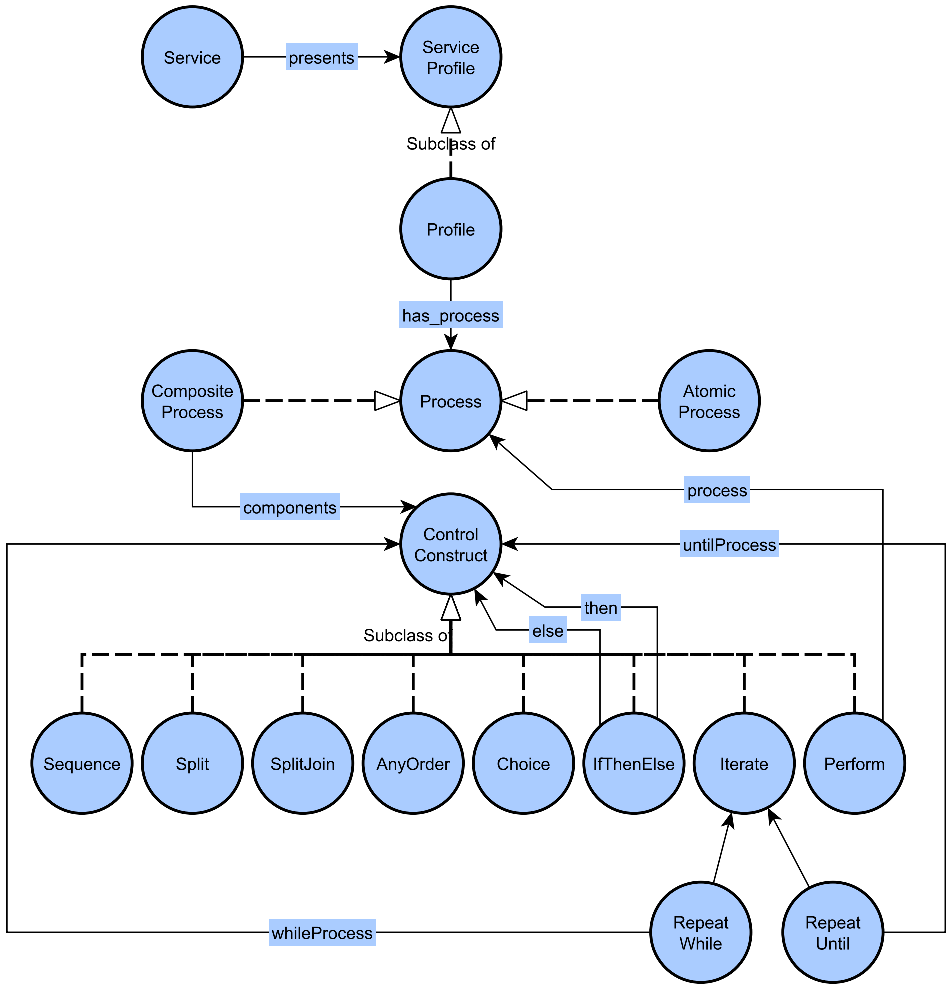


Figure 6.4: Control flow constructs in OWL-S

a complex control flow. Other processes can be referenced from the inner subprocesses of a `owls:CompositeProcess` using the `owls:Perform` control flow construct.

The abstract `owls:ControlConstruct` may have one or more nested components, i.e., subordinate `owls:ControlConstructs`. An `owls:Sequence` executes a list of components in a certain order. An `owls:Split` executes its components in several branches of control flows concurrently. An `owls:Split-Join` joins concurrent control flows via barrier synchronization when all its components have been completed. An `owls:Any-Order` executes components in an unspecified order sequentially, but not concurrently. The execution of all components is required. An `owls:Choice` executes exactly one of its components. An `owls:If-Then-Else` evaluates the condition *ifCondition*. If the condition evaluates to true, the `owls:ControlConstruct` *then* is executed, otherwise the `ControlConstruct` *else*. An `owls:If-Then-Else` has exactly one *ifCondition*, exactly one *then* statement, and at most one *else* statement. An `owls:Repeat-While` executes the `owls:ControlConstruct` *whileProcess* as long as a certain condition evaluates to true. An `owls:Repeat-Until` executes a `owls:ControlConstruct` *untilProcess* until a condition evaluates to true.

Control Flow Constructs in WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL) is an executable language to specify actions with web services within business processes. In WS-BPEL, operations of web services are specified in WSDL. On one hand, WS-BPEL has basic activities like a variable assignment (`bpel:assign`), function calls (`bpel:invoke`), etc. On the other hand, WS-BPEL defines structured programming constructions as shown in Figure 6.5: A `bpel:sequence` activity contains one or more activities that are performed sequentially. A `bpel:if` activity consists of an ordered list of one or more conditional branches. The first branch whose condition evaluates to true is taken. A `bpel:while` activity executes a contained activity while a condition holds. A `bpel:repeatUntil` activity executes a contained activity until a condition holds. A `bpel:foreach` activity executes a contained activity for a certain number of times concurrently or sequentially. A `bpel:flow` activity executes activities concurrently. A `bpel:pick` activity waits for the occurrence of exactly one event from a set of events, then executes the activity associated with that event.

Control Flow Constructs in BPMN

BPMN is a language to describe business process models. Figure 6.6 shows the BPMN ontology. A `bpmn:Task` is a named, single unit of work, e.g., the call of an operation. A `bpmn:ServiceTask` is a specialized `bpmn:Task` that references a `bpmn:Operation` of a service/API. A `bpmn:SequenceFlow` defines the order of `bpmn:FlowElements`. A `bpmn:ExclusiveGateway` is used to create alternative flows where only one path can be taken. All conditions of

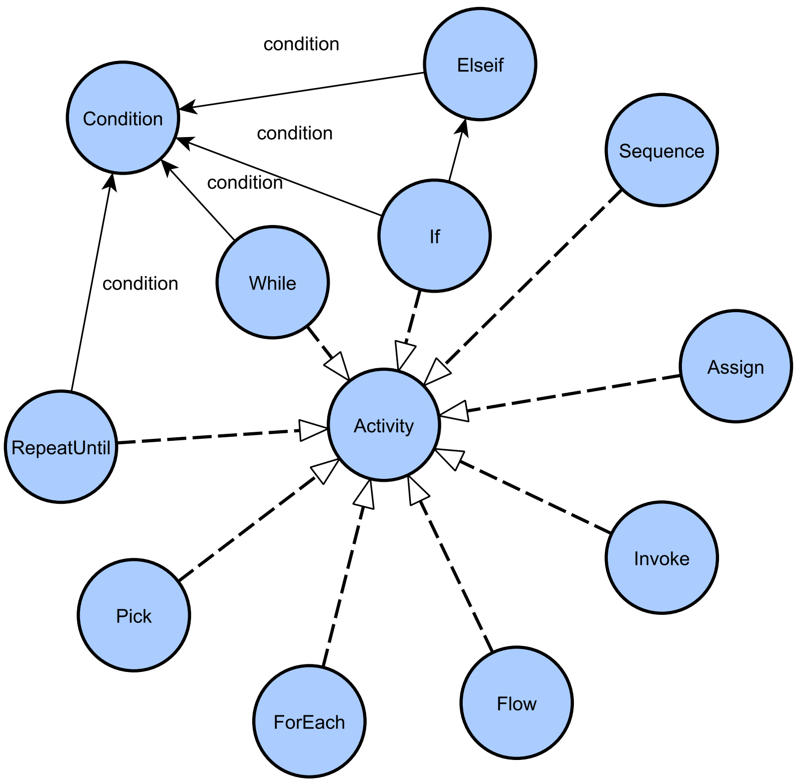


Figure 6.5: Control flow constructs in WS-BPEL

the outgoing paths are evaluated in order. A `bpmn:InclusiveGateway` is used to create alternative flows where all paths are evaluated and may be taken. The order in which the conditions of the outgoing path are evaluated is not specified. A `bpmn:ParallelGateway` is used to create flows that are taken in parallel.

Classifying Control Flow Constructs

In this section, the different control flow constructs from OWL-S, WS-BPEL, and BPMN are arranged in a classification scheme. The classification scheme defines seven abstract control flow constructs: A **Sequence** executes all contained activities sequentially in a certain order. A **Any-Order** executes all contained activities sequentially in any order. A **Branch** executes exactly one activity from a set of multiple possible activities. The further path of the control flow depends on the choice of the user. A **Conditional Branch**

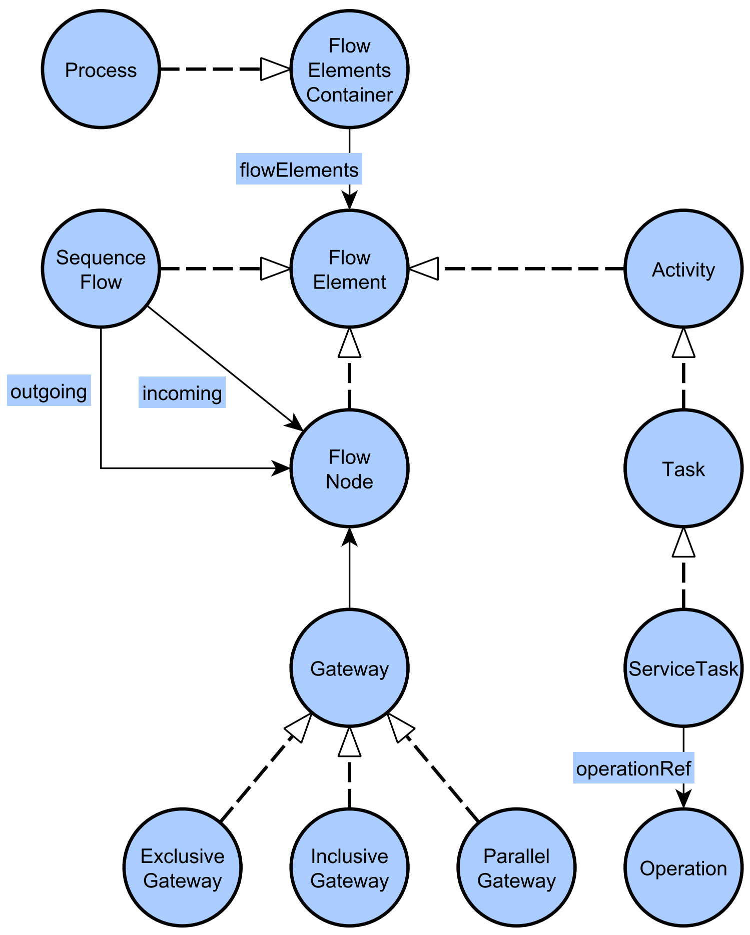


Figure 6.6: Control flow constructs in BPMN

consists of an ordered set of one or more branches. The further path of the control flow depends on an internal condition. The conditions of the branches are evaluated in order until a condition evaluates to true. The first branch whose condition evaluates to true is taken. An optional default branch may be taken if none condition evaluates to true. A **Conditional Multi-Choice Branch** is the same as the Conditional Branch, except that all conditions of the branches are evaluated and every branch whose condition evaluates to true is taken. A **Loop** is the repeated execution of an activity. A **Fork** is the parallel execution of activities. Table 6.2 shows the assignment of OWL-S,

Abstract Control Flow Construct	OWL-S	WS-BPEL	BPMN
Sequence	Sequence	Sequence	SequenceFlow
Any-Order	Any-Order	Flow	n/a
Branch	Choice	Pick	Event-based Gateways
Conditional Branch	If-Then-Else	If	ExclusiveGate-way
Conditional Multi-Choice Branch	n/a	Sequence + If	InclusiveGate-way
Loop	Repeat-While, Repeat-Until	While, RepeatUntil, ForEach	(SequenceFlows looping up/-downstream)
Fork	Split, Split-Join	Flow, ForEach	ParallelGate-way

Table 6.2: Classifying OWL-S, WS-BPEL, and BPMN control flow constructs

WS-BPEL, and BPMN control flow constructs into the classification scheme[4].

It should be noted that some control flow constructs are not present in some languages but can be simulated with other control flow constructs. A Conditional Multi-Choice Branch can be simulated by a sequence of Conditional Branches.

6.3.2 Relating Languages and Process Mining Algorithms

Process mining algorithms produce different kinds of output models. The Alpha Miner produces Petri nets, the Heuristic Miner produces causal nets, and the Inductive Miner produces process trees. Petri nets and causal nets are low-level process models. Control flow constructs are only implicitly visible and result from certain patterns of Places, Transitions, and Arcs. Different output models complicate a comparison with the control flow constructs of the

[4]The control flow construct `bpel:foreach` is classified as Loop and Fork, depending on whether the parallel flag is set to true or false.

Mining Algorithm	Output model	SequenceFlow	ParallelGateways	ExclusiveGateway	InclusiveGateway	Loops
Alpha Miner	Petri net	✓	✓	✓	✗	✗
Heuristic Miner	Causal net	✓	✓	✓	✓	✓
Inductive Miner	Process tree	✓	✓	✓	✓	✓

Table 6.3: Comparison of the mining algorithms [1]

specification languages. Kalenkova et al. [1] present algorithms to transform Petri nets, causal nets, and process trees into BPMN models. In the context of this section, the transformation is used to gain better comparability of the control flow constructs.

Now the question is which control flow constructs can in principle be discovered by process mining algorithms. Table 6.3 shows which BPMN control flow constructs can be recovered from the respective process models according to [1]. Furthermore, the control flow constructs Conditional Branch, Conditional Multi-Choice Branch, and Loop evaluate Boolean conditions to take specific control paths. The call-log does not show which conditions are responsible for certain control flow paths. Therefore, conditions of control flow constructs can not be reconstructed from the call-logs with the aid of the Alpha Miner, Inductive Miner, and Heuristics Miner alone. Since it is not visible whether a control flow construct has a condition, it is not possible to distinguish between a Branch that has no condition and a Conditional Branch that has a condition.

From the call-log is also not visible whether operations were called from concurrent processes or not. The sequence of calls generated by two operations op_1 and op_2 that are executed concurrently in a Fork are indistinguishable from the call sequences generated by op_1 and op_2 executed in Any-Order.

The ability of the Alpha Miner to discover loops is limited as it has been proven that the Alpha Miner is unable to discover short loops of size one and two [126]. These considerations lead to the following recommendations:

- The REST APIs on a BaaS platform must not implement the control flow construct Any-Order, so that the ambiguity between Any-Order and Fork does not arise in the first place.
- OWL-S is not suitable for the specification of API protocols because, in contrast to WS-BPEL and BPMN, it contains the Any-Order control flow construct.

6.4 Functionality of the API Protocol Miner

This section describes the functionality of the Protocol Miner. The input to the Protocol Miner is a call-log recorded at the API Gateway of the BaaS platform. When a new BaaS platform is launched, it initially contains no call-logs. Therefore, protocol mining cannot be offered from the start. To mitigate this problem, a BaaS platform should also be an API management platform like RapidAPI: API providers are attracted to use the platform, because they benefit from the features of the API management platform, such as API authorization, monitoring, billing, etc. Over time, the platform's user base grows and the APIs offered are increasingly used. The API Gateway of the BaaS processes the original HTTP/S requests, forwards the requests to the individual APIs and returns the API response to the client applications. Gradually, the call-logs get filled with entries so that protocol mining can be carried out. The call-logs are then used to mine API protocol specifications to help requesters creating mashups.

The approach is shown in Figure 6.7 which is explained in the following:

1. API providers publish their APIs at the BaaS platform (cf. Chapter 4).
2. API consumers create client applications that are using these APIs.
3. All calls from client applications go through the BaaS API Gateway and leave traces in the respective call-logs.

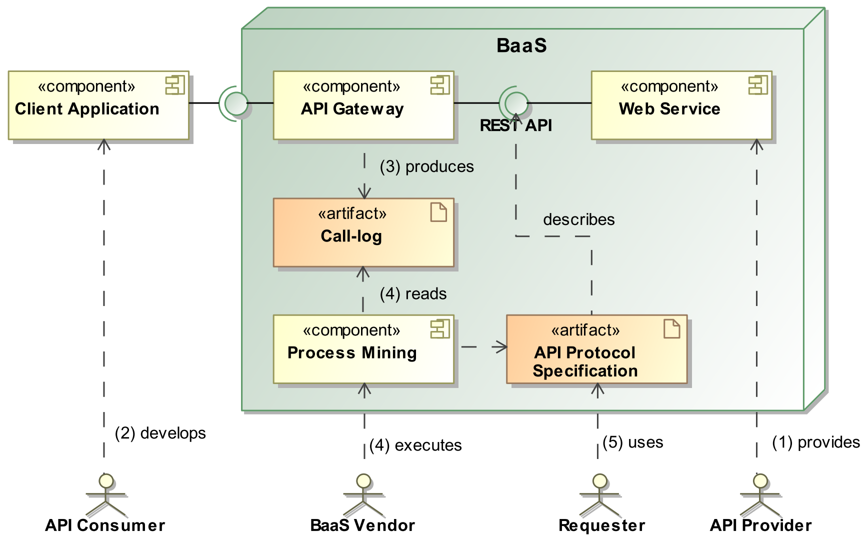


Figure 6.7: Mining API protocol specifications from call-logs

4. The BaaS vendor executes process mining to discover an API protocol specification from the call-log.
5. Requesters use this API protocol specification to learn in which order API operations have to be called which is necessary to integrate the API into their mashup.

The difference between API consumers and requesters is that API consumers only use the BaaS as an API management system. In order for API consumers to be able to create a client application that uses a REST API, they must already be familiar with the API protocol. Initially, API consumers cannot access API protocol specifications because they have to be mined first, which requires call-logs to be created first. Since API consumers do not have an API protocol specification, they must determine the protocol using inefficient methods such as reverse engineering, try and error, or by consulting the API provider. As soon as the client applications are used, call-logs are created and protocol mining becomes available. Protocol mining externalizes the implicit knowledge of API consumers about the API protocols. Future requesters or API consumers can then benefit when creating new mashups or client applications respectively.

Figure 6.8 is an activity diagram that shows how the approach is carried out

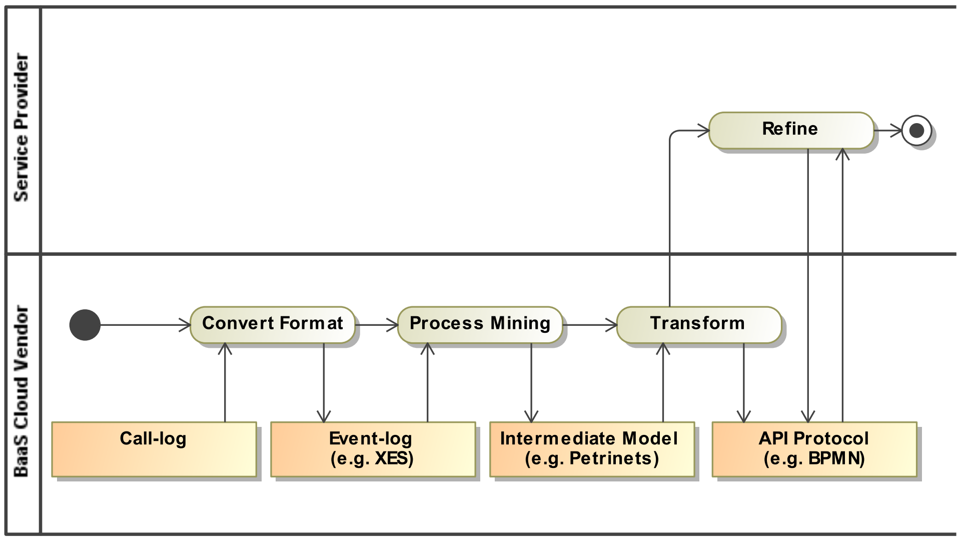


Figure 6.8: Using process mining to discover process models from a call-log

by the participating roles. The call-logs created at the API Gateway cannot be directly used by Process Mining algorithms. First, the BaaS Cloud vendor needs to convert the call-logs into event-logs, e.g., using the standardized XES format [103]. The Process Mining Algorithms produce different output models, e.g. Petri nets or Heuristic nets. These output models can be transformed to BPMN models [1]. BPMN is widespread and is often used to represent processes. In the remainder, BPMN diagrams are used to present and compare the results of the various mining algorithms in a uniform manner.

6.4.1 Converting Call-logs to Event-logs

Call-logs are not directly suited for process mining because process mining requires event-logs, where every event has at least a **name**, a **timestamp**, and a so-called **case**. Every event is assigned to a case, i.e., a session. The case allows distinguishing several process instances from another. The first and last events of a case define the start and the end of a single process instance. The standard for the representation of event-logs is the Extensible Event Stream (XES) format. Process Mining tools such as ProM [122] do not support call-logs, but event-logs in XES format. Thus, call-logs need to be translated to event-logs before they can be used for process mining.

Mapping Timestamps

In the context of this chapter, a name of an event and its timestamp corresponds to the name of the operation being invoked at a certain point in time. Timestamps from the call-logs can be directly used in the event-log.

Mapping Operation Calls to Event Names

Call-logs contain the invoked request URLs of the operations. In general, request URLs are not suited to be used as event names in event-logs because they can contain specific parameter values in the URL path. For example, the URL `/api/simphera/projects/e18f107b` contains a parameter *projectId* which is a random string (UUID) denoting a certain resource. If these request URLs would be used for process mining, the discovered API protocol would distinguish between concrete calls of the same operation, which is undesirable because the API protocol has to generalize from concrete calls. For this reason, these URLs with concrete values have to be replaced by an event name that uniquely identifies the called operation.

The API Protocol Miner matches the specific URLs from the call-log with the paths defined in the OpenAPI to determine the respective operation. The correspondence of a concrete URL in the call-log to an API operation is always unambiguous. If the corresponding operation is found, the concrete parameter values are replaced by the parameter names. The URLs path without concrete parameter values but with parameter names can then be used as event names in event-logs.

Mapping Sessions to Cases

Until now, the timestamps and request URLs from the call-log have been converted to timestamps and event names in the event-log. What remains is determining the cases. In the context of call-logs, cases are sessions, i.e., sequences of operation calls to complete a certain activity. Call-logs contain no explicit information about cases. Determining the case of certain calls is difficult because call-logs are noisy in practice and that noise disrupts discovering general API protocols through process mining.

Because there is noise in call-logs, it may happen that two consecutive calls in the call-log may belong to different interfering sessions. It is important to assign the calls to their sessions/cases because otherwise, process mining would discover useless API protocols that model random behavior. If requesters would use such a defective protocol specification they would call the operations in an incorrect order, resulting in a faulty execution of the API at run time.

The problem of assigning a single event in a call-log to a case is also known from another research field, e.g., in the area of Web Usage Mining [127, 128, 129, 130, 131, 132]. While the goal of Web Usage Mining is to learn user behavior on websites, the goal of the BaaS approach is to learn the API protocol. In Web Usage Mining, the process of assigning sessions to their individual cases is known as session reconstruction or sessionizing. The call-log entries for visiting a website and calling an API are identical in a call-log because websites and APIs both use the HTTP/S protocol. Thus, in general, session reconstruction techniques from the field of Web Usage Mining can also be applied to identify sessions in call-logs. In the following, two methods for session reconstructions are presented. Depending on the individual area of application of the BaaS platform, both methods can be considered.

Grouping by Client Host and User Agent A simple way to assign the call-log entries to a case is to group them by client IP address and user agent [32, Section 14.43]. A user agent is a software, e.g., a client application that acts on the behalf of the user, e.g., a web browser. If two different user agents access the same REST API from the same IP address from one computer, they can still be differentiated in the call-log and assigned to two different cases.

However, this method has limitations: For example, if several instances of the same user agent are running in parallel on one computer, the calls cannot be assigned to an instance using the call-log. Even if multiple user agent instances are running on different computers in the same private network and the SIMPHERA API is in a public network, it cannot distinguish between process instances.

```
10.255.204.170 [2022-12-11T23:48:52+00:00] "GET /api/simphera/projects
↪  HTTP/1.1" 200 "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
↪  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36"
```

Listing 7: Single entry from raw call-log including client IP address and user agent

Another limitation is due to the fact that SIMPHERA runs in a Kubernetes cluster. By default, Kubernetes obscures the original client IP and replaces it with an internal IP address, which is then identical in all incoming requests. This means that cases can no longer be distinguished based on the IP address. However, this behavior of Kubernetes can be switched off so that the original IP address is retained[5].

Listing 7 shows a single entry of the call-log for the SIMPHERA API. The entry comprises the timestamp of the HTTP request, the request URL, the HTTP method, the IP address of the client host (i.e. the caller), the HTTP response code, and the size of the response in bytes.

Grouping by Authorization Tokens Another possibility to assign calls to a case is using session ids or authentication tokens: When a client application authenticates at a REST API, it is typically assigned either a session id or an authentication token. Session ids are unique random strings and are used to store and load data across multiple subsequent requests on the server side. Authentication tokens like the JSON Web Token[6] contain data about the user, their permissions, etc. Both, session ids and authorization tokens are very suitable for identifying sessions: All process instances using an API need to authenticate against it and retrieve a new, unique session id or authentication token. Even multiple process instances running on the same machine can be distinguished in the call-log using the session id or authentication token. Once the session id or authentication token is issued, it needs to be sent with every subsequent call and it does not change between the calls. This allows tracking all the call-log entries back to its calling process instance.

[5] `https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/#preserving-the-client-source-ip`

[6] `https://jwt.io/`

```
[2022-12-11T23:48:52+00:00] "GET /api/simphera/projects HTTP/1.1" 200
↪  "Bearer eyJhb...UncA"
[2022-12-11T23:48:53+00:00] "GET /api/simphera/projects HTTP/1.1" 200
↪  "Bearer eyJhb...zqws"
[2022-12-11T23:48:54+00:00] "GET /api/simphera/projects/e18f107b HTTP/1.1"
↪  200 "Bearer eyJhb...UncA"
```

Listing 8: Single entry from raw call-log including authorization token

Session identification via authentication tokens fits particularly well with the BaaS architecture: As discussed earlier, BaaS platforms are also API management platforms and a common use case of API management platforms is authentication anyway.

Usually, session IDs and authorization tokens are not transmitted in the URL, but in the HTTP headers `Cookie` [133] and `Authorization` [32, Section 14.8]. By default, web servers do not log these header fields. The SIMPHERA API uses JSON Web Tokens for authentication which are transmitted in the `Authorization` header. By default, the nginx ingress controller that is used in SIMPHERA does not log the HTTP header field Authorization but can be configured accordingly[7]. Listing 8 shows an nginx call-log of the SIMPHERA API with authorization tokens. Listing 9 shows the event-log that is created from the call-log by grouping traces by the authorization token.

So far general considerations about whether it is feasible to discover API protocol specifications through process mining have been made. The following section examines how well the approach works in practice.

6.5 Evaluation

In this section, the API Protocol Miner is evaluated with three different process mining algorithms: Alpha Miner [81], Heuristics Miner [124], and Inductive Miner [82]. For the evaluation, call-logs from the SIMPHERA API are used. In order to evaluate the quality of the API protocols discovered by each algorithm, these are compared with a manually created reference API protocol that acts as the ground truth. The source code of the Protocol Miner and the material

[7] `https://nginx.org/en/docs/http/ngx_http_log_module.html`

```xml
<trace>
 <event>
  <date key="time:timestamp" value="2022-12-11T23:48:52+00:00" />
  <string key="concept:name" value="GET /api/simphera/projects" />
 </event>
 <event>
  <date key="time:timestamp" value="2022-12-11T23:48:54+00:00" />
  <string key="concept:name" value="GET /api/simphera/projects/{projectId}"
  ↪  />
 </event>
</trace>
<trace>
 <event>
  <date key="time:timestamp" value="2022-12-11T23:48:53+00:00" />
  <string key="concept:name" value="GET /api/simphera/projects" />
 </event>
</trace>
```

Listing 9: Translating a call-log into an XES event-log

used for evaluation is publicly available on GitHub[8].

Process models obtained from process mining are usually assessed regarding the quality dimensions simplicity, replay fitness, precision, and generalization [134]. Simplicity assesses the complexity of a process model. Generally, the more nodes and edges a process model has, the harder it is to understand. Replay fitness measures to what degree the event-log traces can be reproduced using a process model. Precision describes to what degree a process model forbids behavior that is not included in the event-log. A process model has perfect precision when it can reproduce exactly the traces contained in the event-log. Process models with perfect precision are overfitted because they allow no other traces as seen in the event-log. There may also be some valid traces that are not present in the event-log, but which should be allowed by the process model. Generalization measures the degree to which a process model generalizes the behavior seen in the event-log, i.e., how many traces are allowed in addition to those in the event-log. If a process model is too general, it is underfitted, which means that it allows too much behavior. A process model that is too general is not useful, because it allows disallowed behavior.

[8]`https://github.com/brokerage-as-a-service/baas`

In this evaluation, the assumption is made, that the manually created API protocol is optimal with respect to these quality criteria. The quality of the mined API protocols is judged relative to the manually created reference API protocol. Ideally, the API protocols produced by the process mining algorithms are identical to the manually created API protocol. Even if none of the mining algorithms is capable to reproduce the manually created process model, the question is whether the discovered process models are useful anyway. For this reason, this section investigates to what degree the discovered process models deviate from the manually created one and which algorithms generate the process model that comes as close as possible to it. The Protocol Miner automatically creates an initial API protocol, which can still contain errors. However, if the API provider only needs to make minor changes, the BaaS approach improves the efficiency of creating API protocols. These considerations bring up the following research questions:

R4 Which of the process mining algorithms Alpha Miner, Inductive Miner, and Heuristic Miner are most suitable for API protocol mining?

R5 How does the quality of a mined API protocol compares to an API protocol that is manually created?

6.5.1 Preparation

First, a reference API protocol is modeled manually which is shown in Figure 6.9 as a BPMN model. Since the entire SIMPHERA API protocol is too extensive for evaluation, the number of operations is reduced. The reference API protocol represents just a fraction of the entire SIMPHERA API protocol and is chosen in such a way that it covers the previously identified abstract control flow constructs *Sequence*, *Conditional Branch*, *Loop*, and *Fork*. A client application is then created that implements this reference API protocol. Using the client application, a real call-log is recorded which is the input for the API Protocol Miner. Several process instances of the client application are started to simulate process interference.

One disadvantage of evaluating with real call-logs is that a run of the SIMPHERA API protocol takes several minutes to complete and the effects of interference from multiple process instances and noise are not well reproducible. Therefore, a simulator is created that does not make real HTTP calls

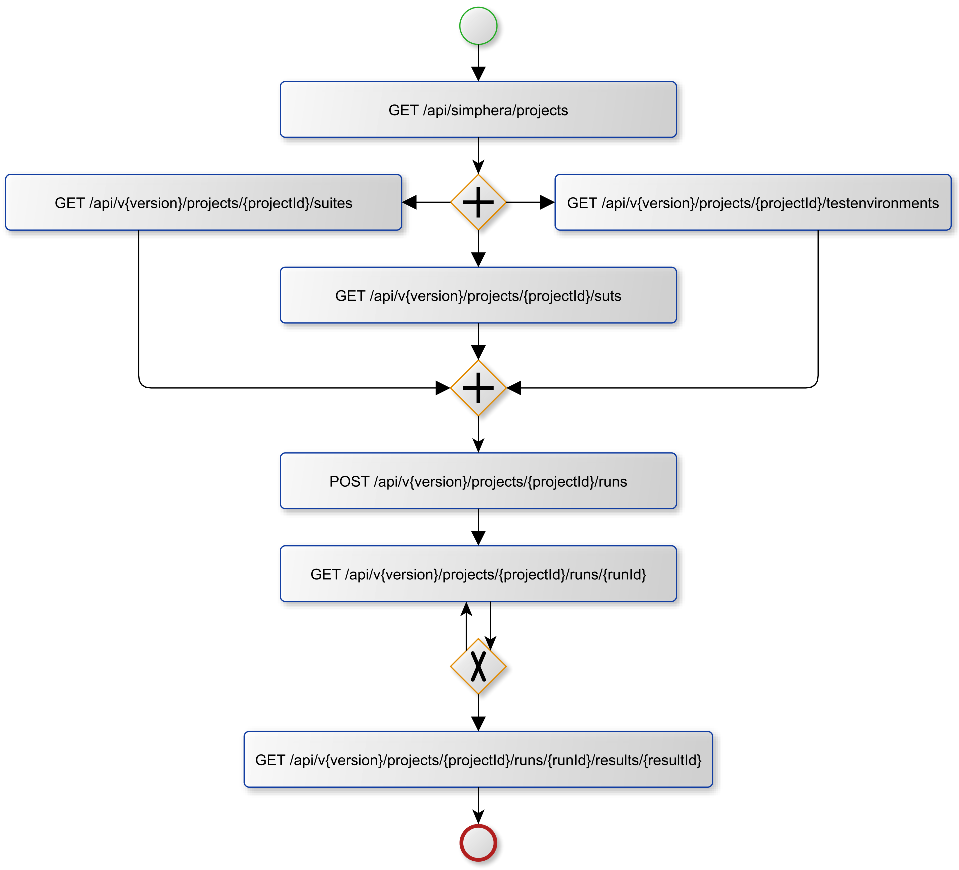

GET /api/simphera/projects
GET /api/v{version}/projects/{projectId}/suites
GET /api/v{version}/projects/{projectId}/testenvironments
GET /api/v{version}/projects/{projectId}/suts
POST /api/v{version}/projects/{projectId}/runs
GET /api/v{version}/projects/{projectId}/runs/{runId}
GET /api/v{version}/projects/{projectId}/runs/{runId}/results/{resultId}

Figure 6.9: Reference API protocol of the SIMPHERA API

but writes the call-log directly according to the real call-log. The simulator is run multiple times in parallel to simulate interference. To simulate noise, the simulator calls any random operations from the entire SIMPHERA API in a concurrent process.

To study the effects of noise on the mining algorithms under investigation, two call logs are created: One noise-free call-log and one call-log with noise. Each algorithm is run with the same two call-logs to maintain comparability.

6.5.2 Proceeding

The API Protocol Miner is executed using the Alpha Miner [81], Heuristics Miner [124], and Inductive Miner [82]. The Petri net discovered by the Alpha Miner, the Causal net discovered by the Heuristics Miner, and the Process Tree discovered by the Inductive Miner are transformed into BPMN models according to the approach presented by Kalenkova [1]. All algorithms are executed with a noise-free and with a noisy call-log.

6.5.3 Results

This section analyzes the BPMN models that are discovered through process mining and compares these models to the reference model.

Alpha Miner

Figure 6.10 shows the BPMN process model discovered by the Alpha Miner from the noise-free call-log. It has been correctly identified that the loop of operation `GET /api/v{version}/projects/{projectId}/runs/{runId}` exists. It is not recognized that `POST /api/v{version}/projects/{projectId}/runs` and GET `/api/v{version}/projects/{projectId}/runs/{runId}` are executed in a sequence. The parallel execution of operations is not correctly captured by a fork. The conditional branch of `GET /api/v{version}/projects/{projectId}/runs/{runId}` is not reflected in the BPMN model.

The BPMN model, which is generated from the noisy call-log using the Alpha Mining algorithm, is not shown here. The Alpha algorithm cannot deal with noise, which is why the random operation calls are also reflected in the model. This makes the model very cluttered so that it is of no use to the requester.

Inductive Miner

The process model created by the Inductive Miner is shown in Figure 6.11. In particular, the variant *Inductive Miner - infrequent IMf* is used with the noise threshold set to 0.2. The produced BPMN model is almost exactly

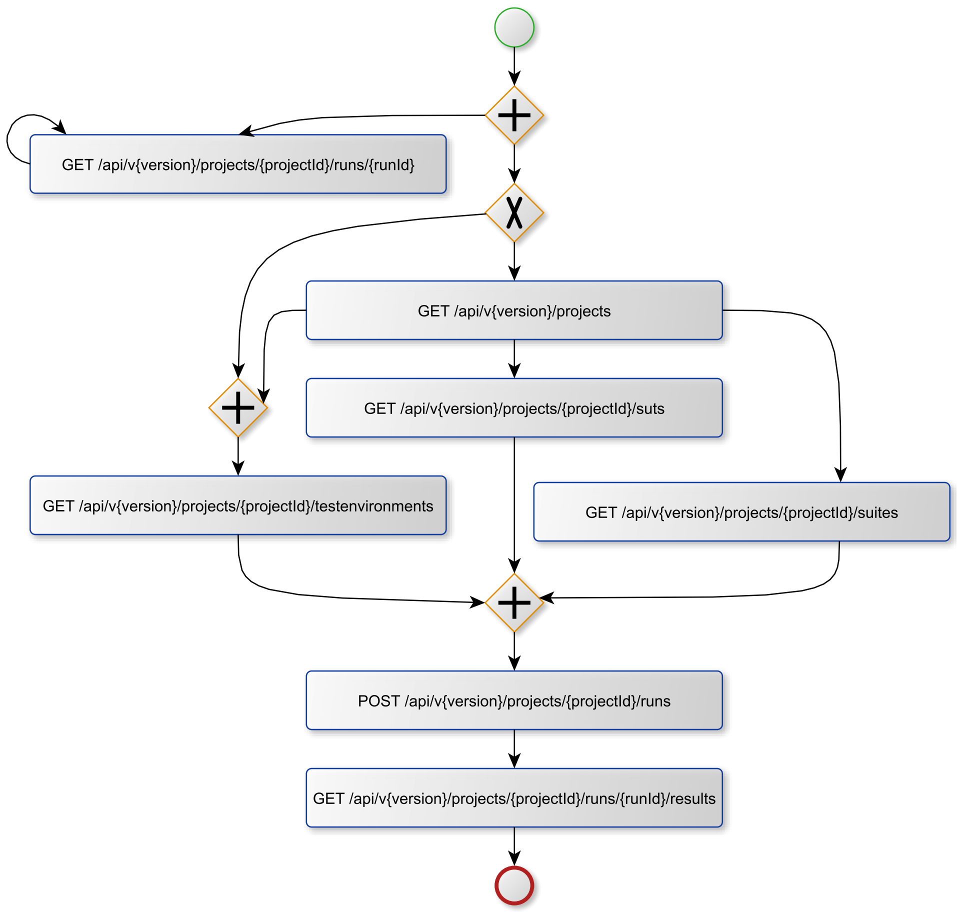

GET /api/v{version}/projects/{projectId}/runs/{runId}
GET /api/v{version}/projects
GET /api/v{version}/projects/{projectId}/suts
GET /api/v{version}/projects/{projectId}/testenvironments
GET /api/v{version}/projects/{projectId}/suites
POST /api/v{version}/projects/{projectId}/runs
GET /api/v{version}/projects/{projectId}/runs/{runId}/results

Figure 6.10: BPMN model produced by Alpha Miner from noise-free call-log

the reference model, except that one Parallel gateway is missing. All of the control flow constructs are captured correctly.

Figure 6.12 shows the BPMN model created by the Inductive Miner from the noisy call-log. The operation `GET /api/v{version}/projects` is entirely filtered out. Not all sequences have been identified correctly. For example, it is not recognized that `POST /api/v{version}/projects/{projectId}/runs` and GET `/api/v{version}/projects/{projectId}/runs/{runId}` are executed in a sequence.

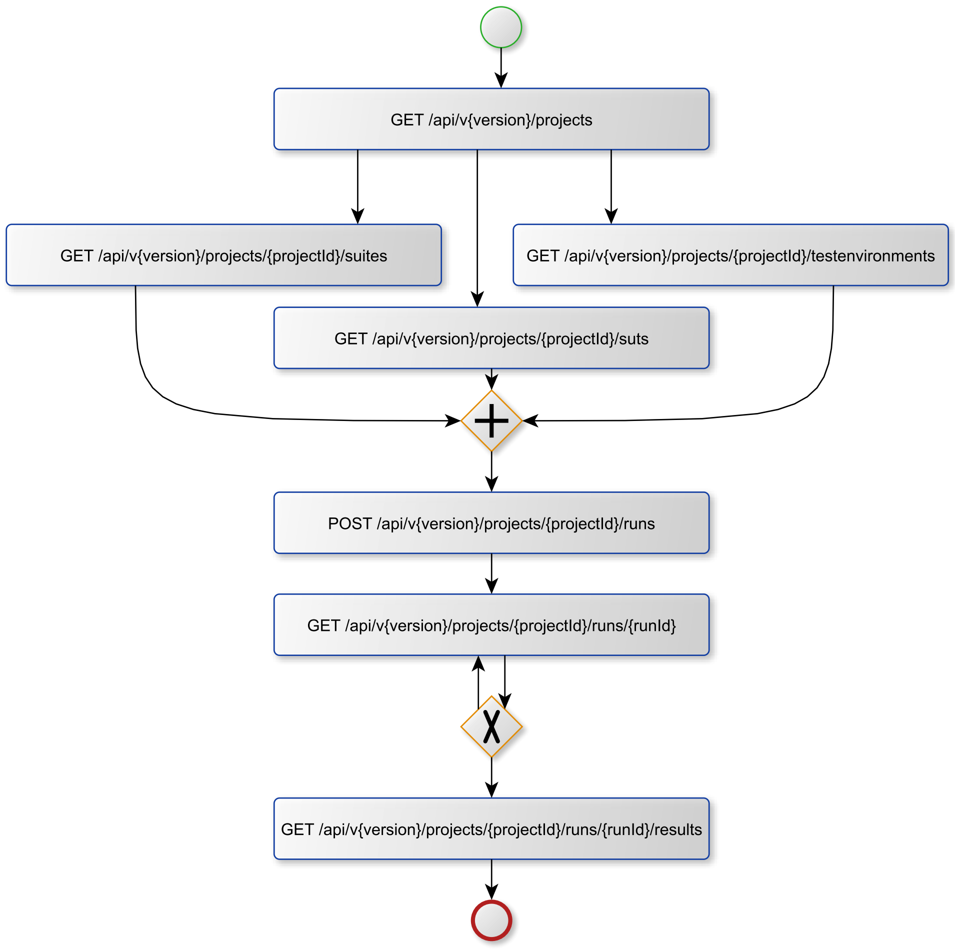

GET /api/v{version}/projects
GET /api/v{version}/projects/{projectId}/suites
GET /api/v{version}/projects/{projectId}/testenvironments
GET /api/v{version}/projects/{projectId}/suts
POST /api/v{version}/projects/{projectId}/runs
GET /api/v{version}/projects/{projectId}/runs/{runId}
GET /api/v{version}/projects/{projectId}/runs/{runId}/results

Figure 6.11: BPMN model produced by Inductive Miner and Heuristics Miner from noise-free call-log

Heuristic Miner

The BPMN model created by the Heuristic Miner is shown in Figure 6.11. The model is exactly the same as produced by the Inductive Miner from the noise-free call-log. In case of the noisy call-log, the produced BPMN model comes very close to the reference model. The BPMN model created by the Heuristics Miner from the noisy call-log is shown in Figure 6.13. There is only one misplaced Parallel gateway.

The Heuristics Miner shows the best results for the SIMPHERA example used in the evaluation. In general, the Heuristics Miner is best suited for

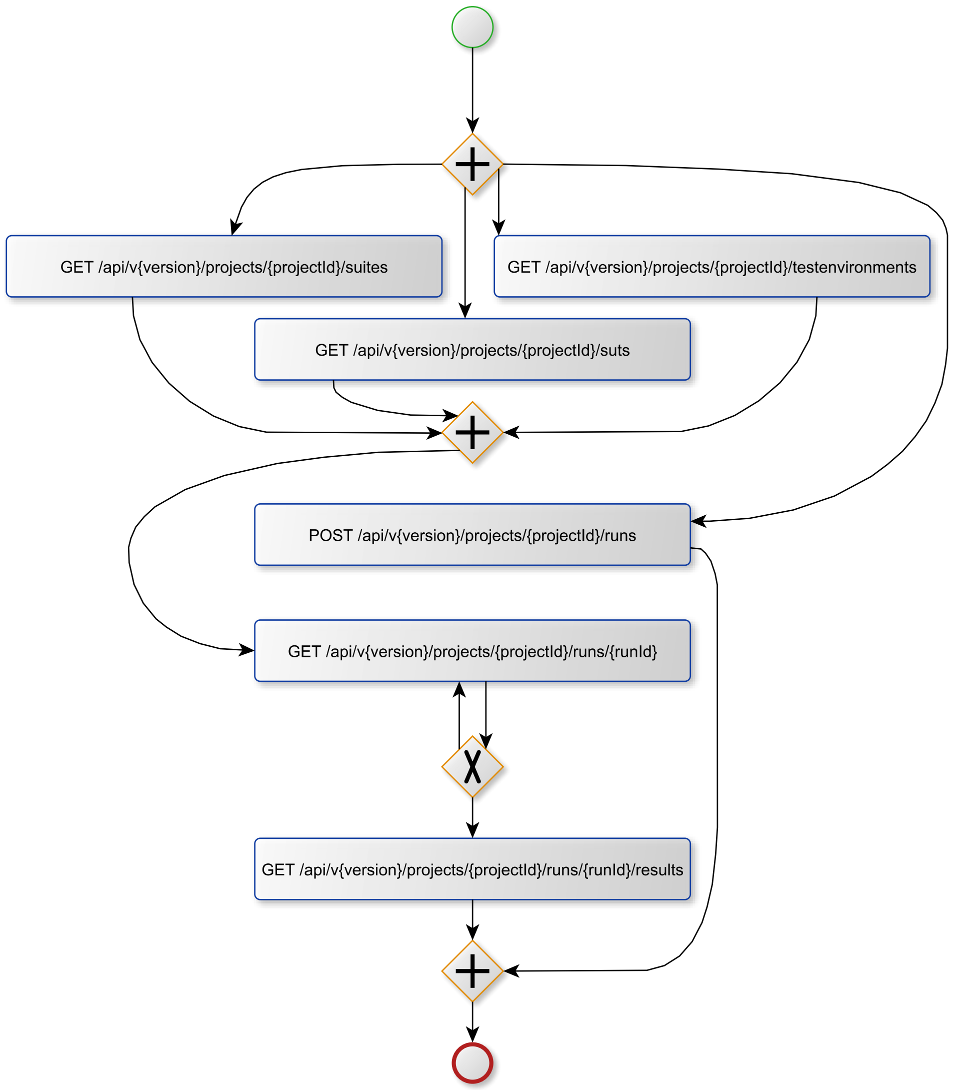

GET /api/v{version}/projects/{projectId}/suites
GET /api/v{version}/projects/{projectId}/testenvironments
GET /api/v{version}/projects/{projectId}/suts
POST /api/v{version}/projects/{projectId}/runs
GET /api/v{version}/projects/{projectId}/runs/{runId}
GET /api/v{version}/projects/{projectId}/runs/{runId}/results

Figure 6.12: BPMN model produced by Inductive Miner from noisy call-log

use with BaaS, since the algorithm is robust to noise. The degree of noise depends on the call log or the individual API. By adjusting the threshold values, the algorithm can be adapted to the individual circumstances of each API relatively easily. This is important for use on a BaaS platform, where the algorithm must provide adequate results for a large number of different APIs. For this reason, Heuristics Miner is best suited to mine API Protocols from call-logs in the sense of research question R4.

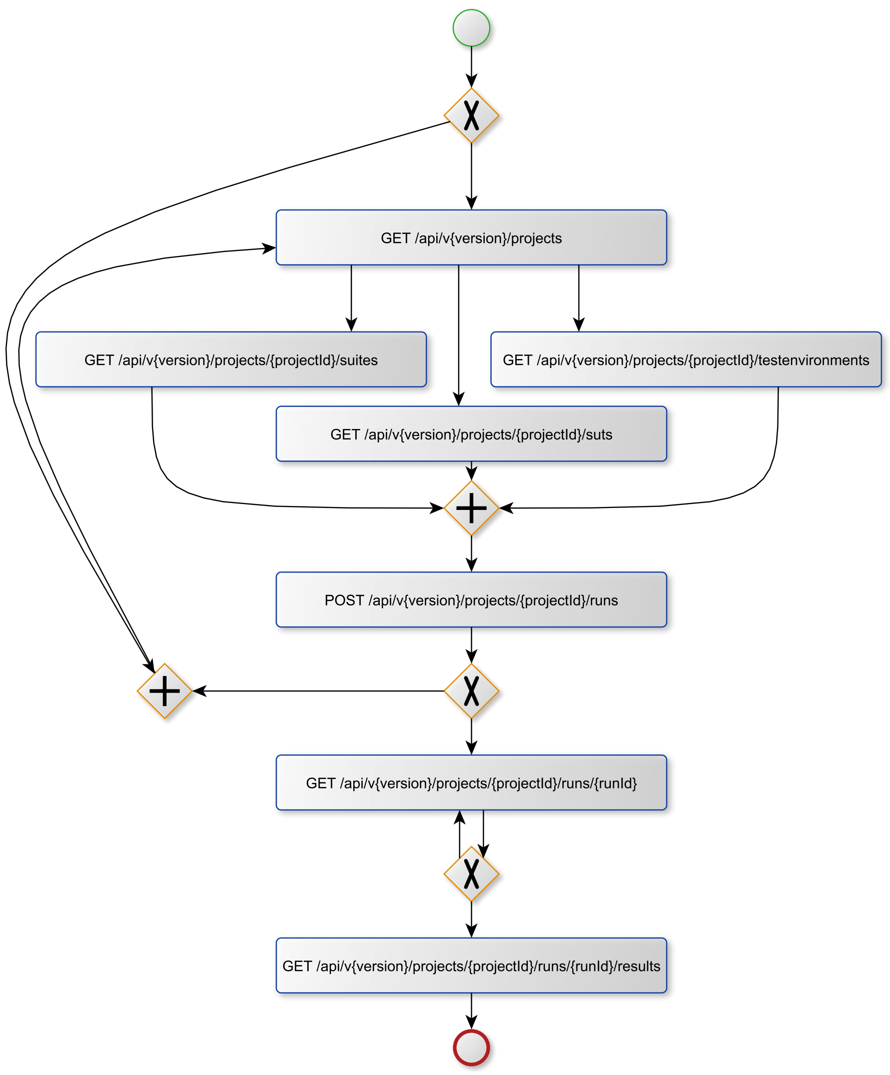

GET /api/v{version}/projects
GET /api/v{version}/projects/{projectId}/suites
GET /api/v{version}/projects/{projectId}/testenvironments
GET /api/v{version}/projects/{projectId}/suts
POST /api/v{version}/projects/{projectId}/runs
GET /api/v{version}/projects/{projectId}/runs/{runId}
GET /api/v{version}/projects/{projectId}/runs/{runId}/results

Figure 6.13: BPMN model produced by Heuristic Miner from noisy call-log

Regarding research question R5 it can be stated that the BPMN model produced by the Heuristics Miner comes very close to the manually created BPMN model that represents the API protocol. Just a little manual adjustment has to be done to correct the produced BPMN model. Compared to modeling an API protocol from scratch, the BaaS approach to mine API protocols from call-logs saves time and is therefore more efficient.

6.6 Summary

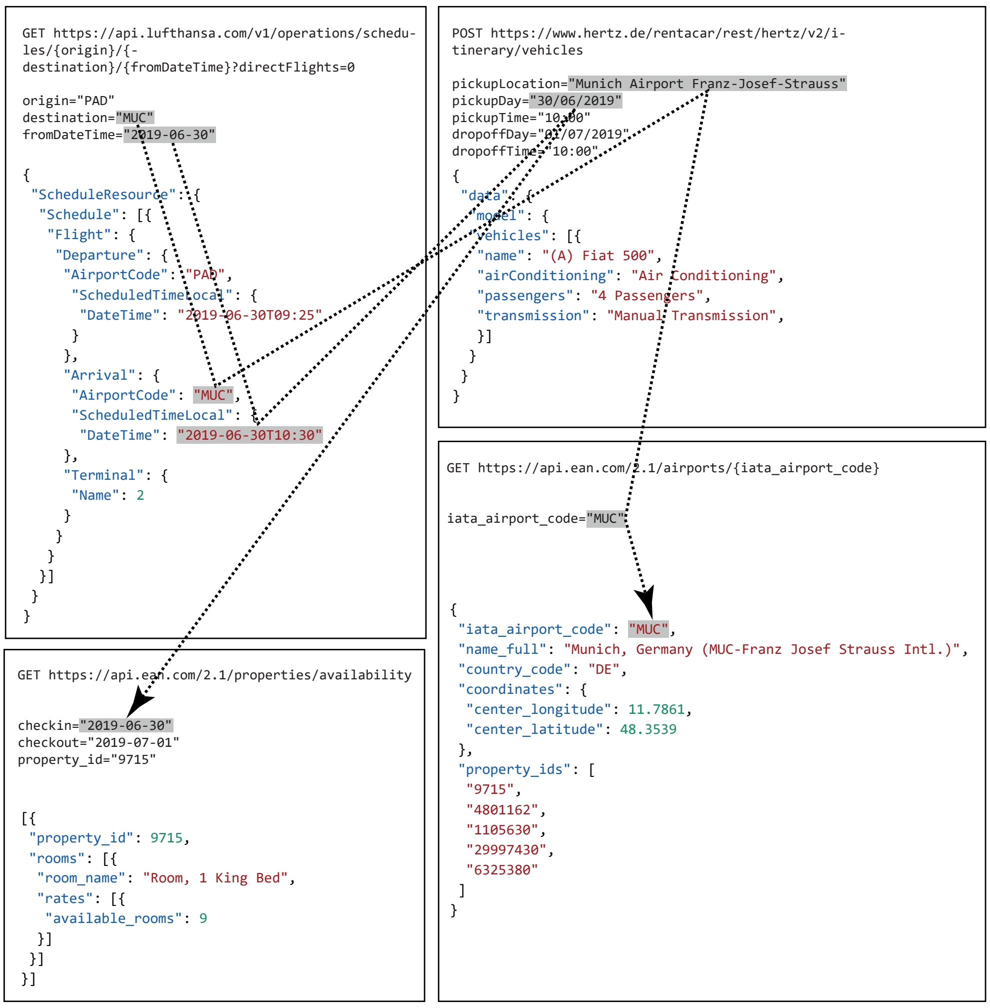
In this chapter, static and dynamic methods for deriving API protocols are discussed. Static methods often fail because the operations of an API often do not use uniform parameter names which makes it difficult to recognize the interconnections between operations. Dynamic methods have the advantage that the call-logs required for this can be obtained easily and no changes to the service implementation are necessary. A problem of dynamic analysis is noise in call-logs that promotes the creation of faulty API protocols. The BaaS approach uses process mining techniques for dynamic analysis. An analysis of the languages OWL-S, WS-BPEL, and BPMN identifies which general control flow constructs are necessary for the description of API protocols. This is opposed to the control flow constructs that can be discovered by the process mining algorithms Alpha Miner, Inductive Miner, and Heuristics Miner. The result is that process mining algorithms cannot distinguish between parallel and sequential control flows when the latter are executed in any order. Furthermore, the concrete conditions for branches in the control flow cannot be retrieved using any of the investigated algorithms. Another challenge is that call-logs cannot be used directly for process mining because process mining requires that each call can be assigned to a process instance which is generally not given in call-logs. This chapter presents two methods to identify a process instance of a call-log by either the IP/user agent or an authorization token/session ID. In the evaluation using a real example, it is shown that the Heuristics Miner generates API protocols even from noisy call-logs, which come very close to the API protocols that a human would specify. Since the automatically generated API protocol is already very accurate and requires little manual post-processing, the BaaS API Protocol Miner contributes to the efficient creation of mashups.

7 Parameter Matcher and Glue Code Generator

This chapter explores how multiple incompatible APIs can interact together in a mashup. Before several APIs can be wired together in a mashup, the relevant parameters that have to be exchanged have to be identified. Identifying input and output parameters that need to be exchanged between the APIs is difficult because the parameter names are terminologically heterogeneous and their values are syntactically incompatible. The syntactical incompatibility requires program logic that converts the data between the APIs. The program logic is called glue code in the remainder.

To give an example, let us consider a mashup for travel trip planning that combines Lufthansa API to book flights, the Expedia API to book a hotel room, and the Hertz API to book a rental car. In this case, the Lufthansa and Hertz API need to exchange data about location and time: The location of the flight arrival and the car pickup location is the same. Furthermore, the flight arrival date and the car pickup date are identical.

Figure 7.1 shows how the input and output parameters need to flow through the Lufthansa, Hertz, and Expedia API mashup. In the figure, every box contains an HTTP request and an HTTP response and consists of three parts: the request URL, the input parameter values, and a JSON response. The dashed lines show the relations between inputs/outputs across the different APIs. For example, the input *origin* of the Lufthansa API and *pickupLocation* of the Hertz API need to be exchanged in a mashup while both are terminologically heterogeneous. That *origin* and *pickupLocation* needs to be exchanged can be hardly understood by only looking at the



```
GET https://api.lufthansa.com/v1/operations/schedu-
les/{origin}/{-
destination}/{fromDateTime}?directFlights=0

origin="PAD"
destination="MUC"
fromDateTime="2019-06-30"

{
 "ScheduleResource": {
  "Schedule": [{
   "Flight": {
    "Departure": {
     "AirportCode": "PAD",
      "ScheduledTimeLocal": {
       "DateTime": "2019-06-30T09:25"
      }
     },
     "Arrival": {
      "AirportCode": "MUC",
      "ScheduledTimeLocal": {
       "DateTime": "2019-06-30T10:30"
     },
     "Terminal": {
      "Name": 2
     }
    }
   }
  }]
 }
}
```

```
POST https://www.hertz.de/rentacar/rest/hertz/v2/i-
tinerary/vehicles

pickupLocation="Munich Airport Franz-Josef-Strauss"
pickupDay="30/06/2019"
pickupTime="10:00"
dropoffDay="01/07/2019"
dropoffTime="10:00"
{
 "data": {
  "model": {
   "vehicles": [{
    "name": "(A) Fiat 500",
    "airConditioning": "Air Conditioning",
    "passengers": "4 Passengers",
    "transmission": "Manual Transmission",
   }]
  }
 }
}
```

```
GET https://api.ean.com/2.1/airports/{iata_airport_code}

iata_airport_code="MUC"


{
 "iata_airport_code": "MUC",
 "name_full": "Munich, Germany (MUC-Franz Josef Strauss Intl.)",
 "country_code": "DE",
 "coordinates": {
  "center_longitude": 11.7861,
  "center_latitude": 48.3539
 },
 "property_ids": [
  "9715",
  "4801162",
  "1105630",
  "29997430",
  "6325380"
 ]
}
```

```
GET https://api.ean.com/2.1/properties/availability


checkin="2019-06-30"
checkout="2019-07-01"
property_id="9715"


[{
 "property_id": 9715,
 "rooms": [{
  "room_name": "Room, 1 King Bed",
  "rates": [{
   "available_rooms": 9
  }]
 }]
}]
```

Figure 7.1: Heterogeneous data exchanged between APIs used in a mashup

parameter names. To recognize this correspondence, additional ontological background knowledge is required.

In the example, *destination* must be wired with *pickupLocation*. The output parameter *DateTime* or alternatively the input parameter *fromDateTime* from the Lufthansa API must be wired with *pickupDay* of the Hertz API. The data that needs to be exchanged is syntactically incompatible, i.e., *destination* is an airport code, while the Hertz API expects *pickupLocation* to be a Hertz-specific identifier. This means that the APIs are not interoperable immediately and cannot directly exchange data with each other. Therefore, it is necessary that the data is syntactically converted before being passed from one REST API to another. Writing the glue code to translate the data from one API to another is laborious which makes creating mashups inefficient.

This chapter presents a semi-automatic approach for assisting requesters to identify those parameters that need to be exchanged and to generate glue code from parameter mappings. First, the problem of parameter matching is analyzed in more detail, i.e., in which ways APIs exchange data with each other and what types of incompatibilities there are.

Next, the functionality of the BaaS Parameter Matcher is introduced. Between the concrete APIs that are to be combined in a mashup, the BaaS Parameter Matcher finds combinations of parameters for which there is a semantic relation. In this way, the requester is supported in identifying between dozens and hundreds of parameters that need to be exchanged between APIs. To find semantic relations between parameters across APIs, several similarity measures are used, which also take into account the semantic type annotations added by the Semantic Annotator.

Once the corresponding parameter mappings are defined, it comes to the task to create glue code for resolving the incompatibilities. Two approaches to how data types can be translated between APIs are discussed in this chapter: Translating the local types of the APIs to global types or translating between the local types directly. Local data types are data types used only within the scope of an API, while data types used by multiple APIs are global data types. In case of BaaS, global data types are defined in a global ontology. Based on the analysis of the two approaches, the functionality of the BaaS Code Generator is introduced. The goal is to pre-generate as much as possible of the glue code in order to relieve the requester of writing glue code by hand. Finally, the Parameter Matcher and Code Generator are evaluated on a real-world example based on the Lufthansa and Hertz API.

7.1 Parameter Matching

Parameter matching is the task to find related parameters of different APIs that are to be exchanged in a mashup. Parameters are either inputs or outputs and this section discusses which combinations of inputs and outputs are useful in mashups. In addition, parameters have different data types. This section defines different levels of compatibility and explains which combinations are useful in a mashup.

7.1.1 Combinations of Inputs and Outputs

Parameters have a direction: Operations consume input parameters and produce output parameters. Combining two parameters of different operations results in four possible combinations: output-input, input-input, input-output, and output-output. In the following, the use cases for the respective combinations are discussed.

Output-Input

For an output-input combination, the output of one operation is used as input of another operation. Cremashi et al. call this kind of combination *sequence composition* [10]. The input of the second operation depends on the output of the first operation. Since both parameters can have different data types and formats, a conversion may be necessary before a valid call to the second operation can be made.

Input-Input

For an input-input combination, two operations are called with the same input. Since the execution of one operation does not depend on the output of the other API, they can be executed in any order. It is possible that both input parameters have different data types and formats so they need to be converted. Cremashi et al. call this kind of combination *parallel composition* [10].

Output-Output

The output-output is useful to aggregate two parameters into a single value. The extent to which syntactic compatibility of the parameters is necessary depends on the individual case. An example of an aggregation is adding up the costs for the Lufthansa flight and the Hertz rental car to get the total price of the whole itinerary. In that case, it is required that the output values have the same data type, format, and unit.

Input		Output	Java Type
OpenAPI Type	OpenAPI Format	JSON Type	
integer	int32	-	java.math.BigDecimal
integer	int64	-	java.math.BigDecimal
number	float	-	java.lang.Float
number	double	number	java.lang.Double
string	-	string	java.lang.String
string	byte	-	byte[]
string	binary	-	java.io.File
boolean	-	boolean	java.lang.Boolean
string	date	-	java.time.LocalDate
string	date-time	-	java.time.OffsetDateTime

Table 7.1: OpenAPI data types and formats and corresponding Java types

Input-Output

In the case of the input-output combination, the input parameter of the first operation is related to an output parameter of the second operation. The input parameter does not have to fulfill the API contract of the second parameter. This combination can be used to perform integrity checks when the parameters are known or expected to have the same range of values.

7.1.2 Degrees of Compatibility

Parameters have a data type, a data format, and a unit. *Data types* like Integer, String, Boolean, etc. with a value range and operations defined on the range. A *data format* is a specific syntax to represent certain values from the value range. Numeric values usually also have a unit. The degree of compatibility between two parameters depends on the data type, the format, and the unit. Table 7.1 shows the OpenAPI data types and formats for input and output parameters and their corresponding Java types. Different cases of compatibility degrees are explained in the remainder. The first case starts with the highest and ends with the lowest compatibility degree.

Case 1: Identical Mapping

Parameters that have identical data types, data formats, and units can be shared directly between two APIs, because their value ranges are identical. To give an example, two parameters of type date-time can be shared between two APIs if they are using the same calendar, the same date format (e.g. RFC3339), and the same time zone.

Case 2: Injective Mapping

This case applies if two parameters differ in their type, format, or unit. Compatibility can be established if there is an injective function that maps the value of the first parameter to the value of the second parameter. This case splits into sub-cases, corresponding to whether the parameters differ in their type, format, or unit:

Case 2.A: Different Types Parameters with different data types may be compatible with each other. To give an example, let us consider two parameters of type *date-time* and *int64*. The int64 parameter is a Unix timestamp, i.e., an *int64* value that counts the milliseconds passed starting from January 1st, 1970. Therefore, both parameters take date values. Many standard libraries, e.g. the Java Runtime Environment, contain functions to convert between date objects and Unix timestamp.

Case 2.B: Different Formats Data formats are construction rules to produce valid values. The same data may have many representations encoded in different data formats. These data formats must first be aligned with each other before data can be shared between the APIs.

The amount of different data formats that are used in practice is limited. Date format standards like RFC3339 [68] are widespread. Different countries use different conventions, e.g., “MM/DD/YYYY” in the United States of America and “DD.MM.YYYY” in Europe. Most standard libraries like the Java Runtime Environment often support commonly used standard formats and the conversion between them.

Instead of specifying the value range by a construction rule, valid literals can also be simply enumerated to define the value range. To identify airports, for example, there are the standards of the International Air Transport Association (IATA)[1] and the International Civil Aviation Organization (ICAO)[2]. The airport Paderborn/Lippstadt is identified by the IATA code "PAD" and by the ICAO code "EDLP". IATA and ICAO identifiers are standardized worldwide and can be translated using a static translation table.

It is also possible that values only have a meaning in the context of a certain API. For example, identifiers of custom data objects cannot be exchanged between APIs, because they are only meaningful within the scope of one API. To translate a custom identifier from one API to another, there must be a mapping table that maps every value.

Case 2.C: Different Units Values of two parameters that measure the same quantity but in different units can be converted by a function. For example, if an operation returns a distance between two geographic coordinates in kilometers but another operation expects the distance in miles as input, then the value can be multiplied by a conversion factor before it is passed to the latter operation.

Case 3: Overlapping Value Ranges

For certain combinations of data types, there is no injective mapping function. Numeric data types have a fixed area allocated in memory. The size of this memory area defines the value range of the data type. The signed 32 bit Integer has a value range of $[-2^{31}, 2^{31} - 1]$ and the signed 64 bit Long has a value range of $[-2^{63}, 2^{63} - 1]$. In fact, there is an injective function that maps Integer values to Long (Case 2). Contrary, there is no injective function that translates all Long values to Integer. However, some values can be converted, as the value range of Long overlaps with the value range of Integer. The translation of a Long value that is outside of the value range of Integer is translated to Integer causing an overflow exception at run-time.

[1]`https://www.iata.org/`
[2]`https://www.icao.int/`

Case 4: Incompatibility

Combinations of parameters where none of the above cases apply are incompatible. For example, let a String parameter be mapped to an Integer parameter. Indeed, insofar as the string contains only numeric characters, the integer value can be parsed from the string. If the string contains only letters, the conversion is no longer possible. Whether a conversion is possible must be checked manually in individual cases.

7.2 Functionality of the Parameter Matcher

This section describes the functionality of the Parameter Matcher. The input of the Parameter Matcher is a list of operations signatures (from different APIs) to be combined in a mashup. Those operations have been selected through the service discovery earlier in the process. Parameters are matched pairwise.

The number of possible parameter mappings for a mashup explodes very quickly: For example, let us consider the operation *OperationSchedulesFromDateTimeByOriginAndDestinationGet* of the Lufthansa API which has 7 inputs and 17 outputs. The operation *vehicles* of the Hertz API has 54 inputs and 196 outputs. This results in $(7 \cdot 54 + 7 \cdot 196 + 17 \cdot 54 + 17 \cdot 196) \cdot 2 = 12000$ possible combinations. If the direction is neglected, the number is halved to 6000 combinations. The number of possible combinations is still too large such that it is very laborious for the requester to determine useful parameter mappings. To find the most relevant parameter mappings, the Parameter Matcher calculates a similarity score and ranks all pairs according to that score such that the request does not have to go through all of the mappings.

7.2.1 Matching Techniques

The Parameter Matcher uses several matching techniques to assess the semantic relevance of pairs of parameters. These similarity scores are determined on the basis of the parameter names, data types/formats, and semantic annotations using string-based, constraint-based, and structure-based similarity metrics respectively. All of the similarity metrics are symmetric such that holds $\sigma(\langle P_1, P_2 \rangle) = \sigma(\langle P_2, P_1 \rangle)$. Symmetry is necessary as the direction of the mapping is ignored.

String-Based

The intuition behind string-based techniques is that two parameters are similar when their names which are strings are similar. The string similarity σ_1 is the Levenshtein distance [71] of two parameter names:

$$\sigma_1(P_1, P_2) := levenshtein(name_1, name_2)$$

Constraint-Based

The intuition behind constraint-based techniques is that semantically similar parameters also have similar data types and formats. Syntactical incompatibility of parameter types and formats is an indicator of semantic incompatibility. The constraint-based similarity takes the type/format compatibility of parameters into account for determining the similarity of parameters. For example, a parameter with OpenAPI format *int64* is dissimilar from a *boolean* parameter. The value range of *int64* is 2^{63} times larger than the value range of *boolean*.

In OpenAPI, the data types and formats of inputs and outputs are defined in JSON schemas. Often, API providers do not specify data types and formats sufficiently and accurately, as is the case at RapidAPI. In general, this is acceptable for the original purpose of OpenAPI specifications which is to generate client and server code for the same API as long as the client and server use the same types/formats consistently. In the case of parameter matching, inaccuracies in the data type specification may falsify constraint-based similarity measures of parameters from different APIs. On the other way round, accurate data type and format specifications of parameters can be improved when constraint-based matching techniques are involved.

If no JSON schema is specified for a parameter, but there are examples, an auxiliary JSON schema can be derived from the example. There are multiple tools[3] available to derive a schema from JSON documents. Inferring the data types, e.g, number, boolean, and string of parameters in a JSON document is trivial: Strings are enclosed in quotation marks, boolean has only two literals true and false, and the rest are numbers.

[3] `https://jsonschema.net/`

```
{
    "Flight": {
        "Departure": {
            "AirportCode": "PAD",
            "ScheduledTimeLocal": {
                "DateTime": "2019-06-30T09:25"
            }
        }
    }
}
```

Listing 10: Example response from Lufthansa API

```
[
        {
                "nameAirport": "Anaa",
                "codeIataAirport": "AAA",
                "codeIcaoAirport": "NTGA",
                "latitudeAirport": "-17.05",
                "longitudeAirport": "-145.41667",
        }
]
```

Listing 11: Example response of the "IATA and ICAO" API using inappropriate types.

The data format is even more specific than the data type and can also be used to determine similarity. To give an example, Listing 10 shows an example JSON response that is returned by an operation of the Lufthansa API[4]. Indeed, the data type of the output *DateTime* is *string* and the data format is *date-time*. Here it can be seen that the data format is valuable for parameter matching because it allows distinguishing date-time parameters from ordinary string parameters so that pairs of date-time parameters can get a higher similarity score.

An analysis of the API specifications from RapidAPI shows that data types

[4]`https://rapidapi.com/lihcode/api/lufthansa-open?endpoint=5afca4b7e4b0547c247d9214`

Regular Expression	OpenAPI Type	OpenAPI Format
`^-?\\d\+\\.\\d+$`	number	double
`^-?\\d+$`	integer	int64
`^\\d{4}-\\d{2}-\\d{2}$`	string	date
`^\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}$`	string	date-time
`^(true|false)$`	boolean	-
`^(yes|no)$`	boolean	-

Table 7.2: Using regular expressions to determine the actual data type and format

and formats are sometimes correctly specified but wrongfully implemented. Listing 11 shows the response of the service *IATA and ICAO*[5]. The output *latitudeAirport* is of type string although it is declared as type/format *number/int64* in its OpenAPI specification.

Since data types and data formats may not be specified or are used incorrectly, the Parameter Matcher ignores the types and formats declared in the OpenAPI specification and uses regular expressions to infer accurate types and formats for the parameters. For this purpose, it tests a series of regular expressions against all available sample data for that parameter. These regular expressions are shown in Table 7.2. Finally, the constraint-based similarity score σ_2 of all OpenAPI data formats are determined according to the static similarity matrix M:

$$\sigma_2(P_1, P_2) := m_{format_1, format_2}, \quad m_{i,j} \in M$$

[5] `https://rapidapi.com/leopieters/api/iata-and-icao?endpoint=5a1c3020e4b0d45349f76c38`

$$M := \begin{array}{c|cccccccccc|}
 & boolean & byte & int32 & int64 & float & double & string & date & date-time & binary \\
\hline
boolean & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
byte & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 \\
int32 & 0 & 0 & 1 & 1/2 & 1/4 & 1/4 & 0 & 1/8 & 1/8 & 0 \\
int64 & 0 & 0 & 1/2 & 1 & 1/4 & 1/4 & 0 & 1/8 & 1/8 & 0 \\
float & 0 & 0 & 1/4 & 1/4 & 1 & 1/2 & 0 & 0 & 0 & 0 \\
double & 0 & 0 & 1/4 & 1/4 & 1/2 & 1 & 0 & 0 & 0 & 0 \\
string & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
date & 0 & 0 & 1/8 & 1/8 & 0 & 0 & 0 & 1 & 1/2 & 0 \\
date-time & 0 & 0 & 1/8 & 1/8 & 0 & 0 & 0 & 1/2 & 1 & 0 \\
binary & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

Structure-Based

The structure-based similarity measure of the Parameter Matcher determines the similarity of two parameters based on the proximity of the semantic types within the ontological structure. The semantic type annotation have been added by the Semantic Annotator earlier in the process. Parameters are annotated with a semantic type that is in the same structure, i.e., the global ontology. The Parameter Matcher uses these semantic types and their type hierarchy relations (*rdfs:subClassOf*) to calculate the upward cotopic similarity [76]. Using the upward cotopic similarity is only possible because all API providers use the same global ontology provided by the BaaS platform. This underscores the need for global ontology.

$$\sigma_3(P_1, P_2) \quad := \quad \frac{|UC(t_1, H) \cap UC(t_2, H)|}{|UC(t_1, H) \cup UC(t_2, H)|}$$

where $UC(t, H) = \{t' \mid \forall t, t' \in H \wedge t \sqsubseteq t'\}$ and $t \sqsubseteq t'$ means that t' is a subclass of t.

Similarity Aggregation

The Parameter Matcher ranks the mappings so that the requester is able to start inspecting the parameter pairs that are most similar. To determine the ranking, the individual similarity values of a parameter mapping are aggregated into a single value after which the parameter mappings are sorted in descending order. As an aggregation function, the average of the individual similarity measures is used:

$$\overline{\sigma} = \frac{1}{n}\sum_{i=1}^{n}\sigma_i$$

Example

The following example shows how the different similarity values are calculated for the mapping $\langle destination, pickupLocation\rangle$. The parameter *destination* is annotated with the semantic type *schema:Airport*. Likewise, the parameter *pickupLocation* is annotated with *schema:Place*.

$$\begin{aligned}
\sigma_1 &= levenshtein(\text{“}destination\text{''}, \text{“}pickupLocation\text{''}) = 0.357 \\
\sigma_2 &= m_{string,string} = 1.0 \\
\sigma_3 &= \frac{|\{Airport, CivicStructure, Place, Thing\} \cap \{Place, Thing\}|}{|\{Airport, CivicStructure, Place, Thing\} \cup \{Place, Thing\}|} \\
&= \frac{|\{Place, Thing\}|}{|\{Airport, CivicStructure, Place, Thing\}|} = \frac{2}{4} = 0.5 \\
\overline{\sigma} &= \frac{0.357 + 1.0 + 0.5}{3} = 0.619
\end{aligned}$$

The Parameter Matcher uses these similarity metrics to calculate a similarity score for all pairs of parameters. The output of the Parameter Matcher is a ranked list of parameter mappings. The requester manually selects appropriate parameter mappings before they are passed to the code generator.

7.3 Composing APIs in Mashups

After the parameter mappings have been created, it is now about how glue code can be created from them. This section introduces two approaches to when, where, and how to convert parameters between APIs in mashups to make them compatible. The first approach introduces adapter APIs that translate between the local syntactic data types of the APIs and the global semantic types of a global ontology and vice versa. In the second approach, the data conversion takes place in the mashup itself which comprises translation functions to translate between local data types of the individual APIs directly. Both approaches are described in more detail in the following.

7.3.1 Approach 1: Local-Global-Local Translation

The first approach uses adapters to resolve the incompatibility between APIs to be combined in a mashup. This is done by mapping the local data types of the APIs to the global semantic types which means that the semantic types are actually used in the implementation and not just for annotation. On the level of adapters, all APIs share the same types which are compatible and can be directly exchanged. Figure 7.2 is a component diagram of the Lufthansa-Hertz-Expedia mashup using adapter APIs.

In software engineering, the adapter pattern [135] is a structural design pattern that is used to translate from one programming interface into another. Adapters are often used to integrate different software components that have incompatible interfaces. This is especially useful to integrate third-party software components whose interface or implementation cannot be changed. This is also the case with mashups, where requesters are not able to change the REST APIs provided by the API providers. Adapter APIs are in front of the original APIs and have the same operations except that the local data types are replaced by the semantic types from the global ontology.

The adapter APIs are in front of the original APIs and implement the lifting and lowering transformations as presented in Section 2.3.4: The lifting translation converts the local types of the API into global types and the lowering transformation converts the global types into local types. The translations have to be implemented only once for each API.

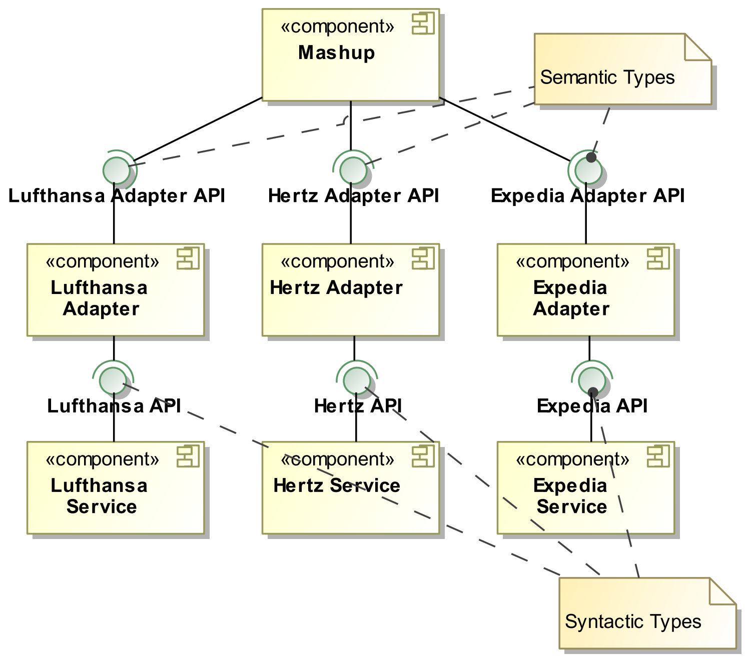
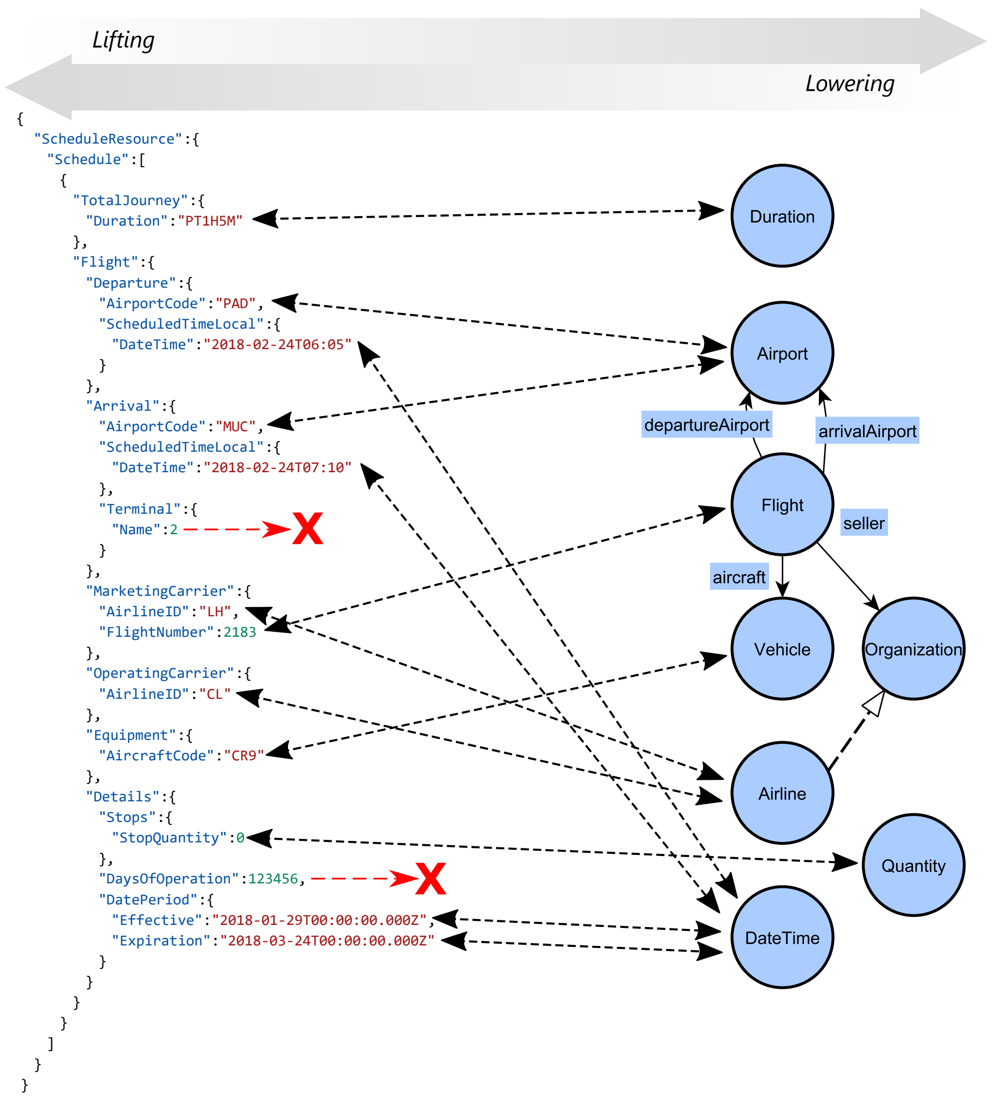

«component» Mashup
Semantic Types
Lufthansa Adapter API
Hertz Adapter API
Expedia Adapter API
«component» Lufthansa Adapter
«component» Hertz Adapter
«component» Expedia Adapter
Lufthansa API
Hertz API
Expedia API
«component» Lufthansa Service
«component» Hertz Service
«component» Expedia Service
Syntactic Types

Figure 7.2: Mashup with local-global-local Translation

One advantage of the approach is that the adapter APIs can be used by requesters as if they were native semantic web services. The local data models of the APIs are fully hidden from the requester that only sees the global semantic types. Thus, on the semantic level, terminological and syntactical heterogeneity is no longer a concern for the composition of the APIs. Once the adapter APIs are created and the local data types are mapped to global data types, the adapters can be reused in other mashups. Since global types are used in the requests and in the adapters, it is then easier to find suitable adapter APIs in the service discovery and also to integrate different adapter APIs with each other when they share some global types in their operation signatures.

While the approach sounds promising in theory, it is difficult to implement in practice. Figure 7.3 shows a mapping of the JSON Response of operation *operationsFlightstatusRouteDateByOriginAndDestination* with corresponding semantic types from schema.org. It can be seen that the semantic types do not



```
{
  "ScheduleResource":{
    "Schedule":[
     {
      "TotalJourney":{
        "Duration":"PT1H5M"
      },
      "Flight":{
        "Departure":{
          "AirportCode":"PAD",
          "ScheduledTimeLocal":{
            "DateTime":"2018-02-24T06:05"
          }
        },
        "Arrival":{
          "AirportCode":"MUC",
          "ScheduledTimeLocal":{
            "DateTime":"2018-02-24T07:10"
          },
          "Terminal":{
            "Name":2
          }
        },
        "MarketingCarrier":{
          "AirlineID":"LH",
          "FlightNumber":2183
        },
        "OperatingCarrier":{
          "AirlineID":"CL"
        },
        "Equipment":{
          "AircraftCode":"CR9"
        },
        "Details":{
          "Stops":{
            "StopQuantity":0
          },
          "DaysOfOperation":123456,
          "DatePeriod":{
            "Effective":"2018-01-29T00:00:00.000Z",
            "Expiration":"2018-03-24T00:00:00.000Z"
          }
        }
      }
     }
    ]
  }
}
```

Figure 7.3: Local datamodel of Lufthansa API cannot be mapped completely onto global semantic types

accurately capture the ontological semantics of the API. For example, there is no semantic type in schema.org that has the same meaning as the *StopQuantity* of the Lufthansa API. The most accurate semantic type in schema.org to represent the parameter *AircraftCode* is *Vehicle*, but this does not fully capture the same information.

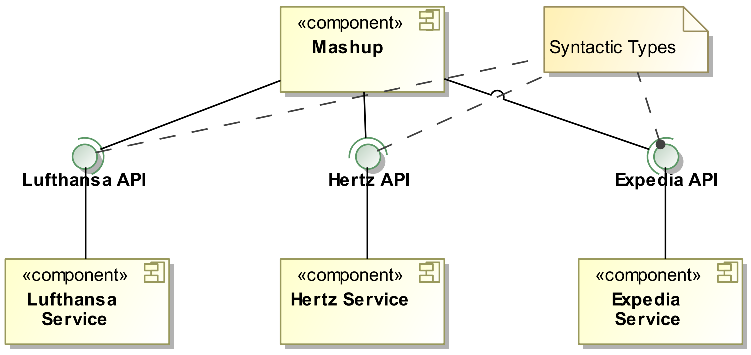


Figure 7.4: Mashup with local-local translation

7.3.2 Approach 2: Local-Local Translation

The second approach translates local types between the APIs directly to resolve the incompatibilities. Semantic type annotations are sorely used here to support service discovery. The program logic to translate the types is directly implemented in the mashup. Figure 7.4 is a component diagram of the Lufthansa-Hertz-Expedia mashup according to the second approach.

The type translations have to be implemented for all parameters that need to be exchanged between APIs and are tailor-made for a specific mashup. This is the reason why the type translation cannot be reused in other mashups. On the other hand, the glue code allows accurate translation of the data in the specific context of a concrete mashup. The effort involved in creating the translation functions is limited because only the data that is actually required has to be translated. The approach fulfills the requirements from Section 1.3:

Upward compatibility The approach is upward compatible because it incorporates the original APIs.

Learnability Requesters only need to be familiar with a programming language of their choice to implement the type translations.

Scalability The approach scales because the original APIs do not have to be changed at all and the amount of work to translate the local types is limited to what is actually needed.

Comprehensiveness The APIs are comprehensively described through ontological semantics by semantic type annotations and API protocol specifications.

Interoperability The glue code achieves compatibility between APIs.

Since the second approach meets the requirements and is also practicable, it is used in the BaaS approach. The next section explains how the code generator works in more detail.

7.4 Functionality of the Glue Code Generator

This section introduces a code generator that is capable to generate essential parts of the mashup program logic. What is novel about the code generator is that it can automatically insert the program logic to extract relevant parameters from complex JSON response messages, syntactically translate the parameter values, and call subsequent operations with the translated values.

The Parameter Matcher ignores the direction of the parameter mappings, but for code generation the direction matters because the direction defines if a parameter value is read or written. For this reason, the requesters have to define the direction of a parameter mapping when they are approving it.

The Glue Code Generator builds upon the standard OpenAPI code generator that can generate fully functional API clients from OpenAPI specifications for individual APIs. For example, Listing 12 shows the Java method that is generated from the operation *operationsSchedulesFromDateTimeByOriginAndDestinationGet* of the Lufthansa OpenAPI. This Java method has arguments for all mandatory and optional parameters declared in the specification. The generated Java functions construct HTTP request messages and send them to the server. The functionality that the BaaS Code Generator adds to the OpenAPI Code Generator is that it considers parameter mappings and wires the different API clients together in a mashup. Depending on the direction of the mapping, there is a repeating pattern of how parameters are extracted, converted, and read. For these patterns, the code generator uses templates, which are explained in more detail in the following.

```java
public Object operationsSchedulesFromDateTimeByOriginAndDestinationGet(
        String origin,
        String destination,
        String fromDateTime,
        String accept,
        Boolean directFlights,
        String limit,
        String offset) throws ApiException {
        //...
}
```

Listing 12: Java operation generated by default OpenAPI code generator

7.4.1 Code Generation Templates

Depending on how the input and output parameters are combined, the corresponding glue code results in recurring patterns. These patterns are reused in the form of code generation templates, with the consequence that glue code generation can be partially automated. This increases the efficiency of creating mashups.

Input-Input Template

Listing 13 shows a code generation template for the input-input mapping $\langle P_1, P_2 \rangle$. The individual three lines of code are explained below: At first, the operation $operation_1$ is called with all its inputs. Inputs are initialized with default values. Second, the translation function $P1\_to\_P2(P1)$ is called, where the value of $P1$ is mapped to a value from the value range of $P2$. For each parameter mapping, an individual translation function is generated. The result of the translation function is stored in a variable P_2. Third, $operation_2$ is called with the translated value as input. Any unbound input parameters of $operation_2$, i.e., input parameters of $operation_2$ that are not part of any other parameter mapping, are filled with default values.

```
$response_1 \leftarrow operation_1(\ldots)$
$P_2 \leftarrow P_1\_to\_P_2(P_1)$
$response_2 \leftarrow operation_2(\ldots, P_2, \ldots)$
```

Listing 13: Code generation template for input-input mappings

Output-Input Template

Listing 14 shows the code generation template for output-input mappings. First, *operation*$_1$ is called with all its inputs. The JSON response of the operation is stored in the variable *response*$_1$. Not all parameters from the JSON response are needed in the context of the mashup, but only those for which a parameter mapping exists. In line 2, calling the function *extract*() reads the value of parameter P_1 from the entire JSON response defined by its JSONPath. The return value of *extract* is passed to the translation function. The result of the translation function is written to the variable *name*$_2$. Finally, in the third and last line, the *operation*$_2$ is called with P_2 and all other inputs. Any unbound input parameters are filled with default values.

```
$response_1 \leftarrow operation_1(\ldots)$
$P_2 \leftarrow P_1\_to\_P_2(extract(jsonpath_1, response_1))$
$response_2 \leftarrow operation_2(\ldots, P_2, \ldots)$
```

Listing 14: Code generation template for output-input mappings

Output-Output Template

Listing 15 shows the code generation template for output-output mappings. The code calls the operations, extracts the parameter values from the respective JSON response messages, calls the aggregation function, and assigns the result to the variable *result*. The code generator just generates a stub for the aggregation function with an empty body. It is up to the requester to implement the function and to further use the aggregated value in the mashup.

```
$response_1 \leftarrow operation_1(\ldots)$
$P_1 \leftarrow extract(jsonpath_1, response_1)$
$response_2 \leftarrow operation_2(\ldots)$
$P_2 \leftarrow extract(jsonpath_2, response_2)$
$result \leftarrow aggregate\_P_1\_P_2(P_1, P_2)$
```

Listing 15: Code generation template for output-output parameter mappings

Input-Output Template

Listing 16 shows the code generation template for input-output mappings. At first, the $operation_1$ is called with input P_1 and the other inputs. Second, $operation_2$ is called with all its inputs. Third, the value of P_2 is extracted from the JSON response message and assigned to the variable $name_2$. This variable is passed to a translation function. Fourth, an assertion statement checks if the value of P_1 and P_2 are equal.

```
$response_1 \leftarrow operation_1(\ldots, P_1, \ldots)$
$response_2 \leftarrow operation_2(\ldots)$
$P_2 \leftarrow extract(jsonpath_2, response_2)$
assert $P_2 = P_1\_to\_P_2(P_1)$
```

Listing 16: Code generation template for input-output parameter mappings

7.4.2 Generating Translation Functions

This section describes the structure of the translation functions that have one argument, corresponding to P_1, and one return value, corresponding to P_2. For each OpenAPI data type/format pair there is a predefined translation function. The predefined method body is copied accordingly for each individual parameter mapping. OpenAPI defines 10 data formats, such that there are 100 possible combinations of formats. Thus, there are 100 pre-built transformation functions.

Some combinations of data type/format mapping are more common in mashups than others. The constraint-based similarity metric automatically prefers common combinations. Nevertheless, the Code Generator even allows uncommon combinations, e.g., a Boolean-Binary mapping so that the requester is not restricted.

The pre-built program logic that is inserted into a translation function may not be accurate for every specific parameter mapping. Even the highest compatibility degree is no guarantee that the pre-built program logic is accurate. For example, the Glue Code Generator does not consider the units of parameters. That the program logic is accurate must be checked by the requester manually. Ultimately, it is up to the requester to implement the translation function properly.

7.5 Evaluation

This section evaluates the Parameter Matcher and the Glue Code Generator. The source code of the Parameter Matcher and the Glue Code Generator and the evaluation artifacts are publicly available on GitHub[6]. The evaluation focuses on the mappings between the operations *operationsSchedulesFromDateTimeByOriginAndDestinationGet* of the Lufthansa API and the *vehicles* operation of the Hertz API. It is examined how the different similarity metrics perform and in particular how much the structure-based similarity metric based on semantic types contributes to finding relevant parameter mappings. In addition, it is examined which parts of the glue code can be generated and what still needs to be handwritten. This section evaluates the following research questions:

R6 Do semantic annotations improve the effectiveness of parameter matching?

R7 To what extent glue code generation can be automatized?

7.5.1 Effectiveness of Parameter Matcher

The section evaluates the effectiveness of the Parameter Matcher and the performance of string-, constraint-, and structure-based similarity measures with respect to average precision Equation 5.2).

Preparation

First, the inputs and outputs of the two operations are annotated with semantic types as much as possible. The Semantic Annotator supports this manual task. The ground truth consists of the parameter mappings $\langle destination, pickupLocation\rangle$, $\langle DateTime, pickupDay\rangle$, $\langle fromDateTime, pickupDay\rangle$ (cf. Figure 7.1). The last two mappings are alternatives: Either the value from input *fromDateTime* can be passed to *pickupDay* (input-input) or from output *DateTime* to *pickupDay* (output-input).

[6]`https://github.com/brokerage-as-a-service/baas`

	σ_1	σ_2	$\frac{\sigma_1+\sigma_2}{2}$	$\overline{\sigma}$
# ⟨destination, pickupLocation ⟩	134	174	84	18
# ⟨fromDateTime, pickupDay ⟩	5162	302	2423	69
# ⟨DateTime, pickupDay⟩	5460	1224	2556	71
Average Precision	.0026	.0033	0.0042	0.028

Table 7.3: Performance of similarity metrics: Ranks and average precision

Proceeding

The Parameter Matcher is executed with comprehensive API specifications including the semantic type annotations as inputs. The similarity measures and ranks for the mappings are calculated.

Results

Table 7.3 shows the ranks of the true positives and the average precision for string-based (σ_1); constraint-based (σ_2); combined string- and constraint-based ($\frac{\sigma_1+\sigma_2}{2}$); and combined string-, constraint-, structure-based ($\overline{\sigma}$)[7]. Table 7.4 shows the ranking of the 6000 parameter mappings according to $\overline{\sigma}$. It can be seen that string-based and constraint-based similarity measures perform better with respect to average precision when combined. Including the structure-based similarity measure further improves average precision. This shows how semantic type annotations can improve the accuracy of the ranking.

7.5.2 Glue Code Generation

This section analyzes how much of the glue code can be created automatically by the Glue Code Generator. This is an important criterion to increase the efficiency of creating mashups.

[7]Structure-based similarity (σ_3) alone is not listed, because not all parameters can be annotated with a semantic type so that parameters without a semantic type annotation appear at a random rank.

#	P_1	P_2	σ_1	σ_2	σ_3	$\overline{\sigma}$
1	Effective	dropoffDayStandard	.167	1.0	1.0	.722
2	Expiration	dropoffDayStandard	.167	1.0	1.0	.722
3	AircraftCode	selectedCarType	.133	1.0	1.0	.711
4	Expiration	pickupDayStandard	.118	1.0	1.0	.706
5	Expiration	pickupDay	.1	1.0	1.0	.7
6	Effective	pickupDayStandard	.059	1.0	1.0	.686
7	fromDateTime	dropoffTime	.5	1.0	.5	.667
8	Effective	dropoffDay	.0	1.0	1.0	.667
9	Effective	pickupDay	.0	1.0	1.0	.667
10	Expiration	dropoffDay	.0	1.0	1.0	.667
⋮	⋮	⋮	⋮	⋮	⋮	⋮
18	destination	pickupLocation	.357	1.0	.5	.619
⋮	⋮	⋮	⋮	⋮	⋮	⋮
6000	StopQuantity	fee	.0	.0	.0	.0

Table 7.4: Ranked parameter mappings

Preparation

Normally, the Glue Code Generator runs with the parameter mappings generated by the Parameter Matcher and manually selected by the requester. To simulate the selection of parameter mappings by a requester for this evaluation, the parameter mappings from the ground truth are taken for code generation. Using the ground truth mappings means that it is pretended that the requester has selected the correct mappings.

Proceeding

The Glue Code Generator is executed with the parameter mappings $\langle destination, pickupLocation\rangle$ and $\langle DateTime, pickupDay\rangle$ as inputs.

Results

Listing 17 is an excerpt from the generated glue code. The target programming language is Java. First, mandatory inputs of the Java method *operationsSchedulesFromDateTimeByOriginAndDestinationGet* are initialized with default values and optional inputs are initialized with the null value (Listing 17 line 1-7). Default values are not very meaningful but are sufficient to fulfill the API contract and to start testing the mashup. The requester just has to replace the default values with variables of the mashup.

Parts of the Lufthansa APIs response will serve as inputs for the Hertz API. To extract single inputs and outputs from JSON messages, the Java library *com.jayway.jsonpath*[8] is used.

The specification of the operation *operationsSchedulesFromDateTime-ByOriginAndDestinationGet* declares seven URL path/query parameters. Consequently, the generated Java method has seven arguments. The Hertz API, in contrast, expects all inputs to be structured in a single JSON object which is why the Java method *vehicles* has only one parameter (Listing 17 line 16, 26). The Code Generator initializes this JSON object with the example value from the Hertz OpenAPI specification, which immediately allows the requester to create syntactically valid API calls through the generated code. What still needs to be done manually is wiring the input JSON object with the rest of the program logic of the mashup.

Because $\langle DateTime, pickupDay\rangle$ is an output-input mapping, the value of *DateTime* is passed to *pickupDay* (cf. Listing 17 line 19). Both *DateTime* and *pickupDay* are declared as data type string. Using the heuristic from Table 7.2 detects the format *date-time* (*java.time.OffsetDateTime*) for *DateTime* and *date* (*java.time.LocalDate*) for *pickupDay*. The generated helper method *dateTime_as_offsetDateTime()* casts the string *DateTime* into its actual Java type *OffsetDateTime* (cf. Listing 17 line 29). The return value of *dateTime_as_offsetDateTime()* is the input for the translation function *dateTime_to_pickupDay()* which translates the type *OffsetDateTime* to *LocalDate*. In this case, the automatically generated translation function

[8] `https://github.com/json-path/JsonPath`

```java
public void compose(){
 String origin = "";
 String destination = "";
 String fromDateTime = "";
 //...

 Object response1 =
 ↪  lufthansa.operationsSchedulesFromDateTimeByOriginAndDestinationGet(
  origin, destination, fromDateTime, Accept, directFlights, limit, offset
 );

 String dateTime =
 ↪  com.jayway.jsonpath.JsonPath.parse(response1).read("$.ScheduleResource⌋
 ↪  .Schedule.[*].Flight.Arrival.ScheduledTimeLocal.DateTime",
 ↪  String.class);

 com.jayway.jsonpath.DocumentContext requestBody1 =
 ↪  com.jayway.jsonpath.JsonPath.using(com.jayway.jsonpath.Configuration.
   defaultConfiguration()).parse("{\"pickupDay\": \"30/06/2019\",
   ↪  \"pickupLocation\": \"Flughafen München, Franz-Josef-Straß\"}");

 requestBody1.set("$.pickupDay",
   dateTime_to_pickupDay(
   dateTime_as_offsetDateTime(dateTime)
  )
 );

 requestBody1.set("$.pickupLocation",
 ↪  destination_to_pickupLocation(destination));
 Object response2 = hertz.vehicles(requestBody1);
}

protected java.time.OffsetDateTime dateTime_as_offsetDateTime(String value)
{
 return java.time.OffsetDateTime.parse(value,
 ↪  java.time.format.DateTimeFormatter.ISO_OFFSET_DATE_TIME);
}

protected java.time.LocalDate
↪  dateTime_to_pickupDay(java.time.OffsetDateTime value) {
 java.time.LocalDate result;
 result = value.toLocalDate();
 return result;
}

protected String destination_to_pickupLocation(String value){
 String result;
 result = value.toString();
 return result;
}
```

Listing 17: Generated glue code from ⟨destination, pickupLocation⟩ and ⟨DateTime, pickupDay⟩

is accurate. This shows that the Code Generator can even handle incorrectly specified data types in the OpenAPI specification.

The method *destination_to_pickupLocation()* (Listing 17 line 39) translates *destination* into *pickupLocation*. Both parameters are of type string. The pre-built translation function is just copying the value. In this case, the predicted translation function is inaccurate, because *destination* contains an IATA code while *pickupLocation* expects a Hertz-specific identifier. An appropriate translation function cannot be generated completely automatically because it is specific to Lufthansa and Hertz API. Here, the requester has to manually implement the program logic of *destination_to_pickupLocation()*.

In total, the generated glue code for the mappings $\langle destination, pickupLocation\rangle$ and $\langle DateTime, pickupDay\rangle$ has 43 lines of code. Out of these 43 automatically generated lines only 5 need manual adjustments for the parameter initialization with meaningful values and the implementation of the translation function *destination_to_pickupLocation()*. This means that 88% of the generated glue code is accurate. Of course, the implementation of a mashup goes beyond the range of functions that the Glue Code Generator can generate. Nevertheless, it relieves the requester of a large part of the work, which contributes to the more efficient creation of mashups.

7.6 Summary

An important factor that makes creating mashups inefficient is identifying those parameters that need to be exchanged between APIs. Parameters have a direction and a syntactic data type. It is shown in this chapter that all combinations of directions and types can in principle be relevant for the creation of mashups, so the search for relevant parameter pairs cannot be simplified by excluding certain combinations in advance. This chapter introduces the Parameter Matcher, which helps requesters finding those parameters that need to be exchanged between APIs in a mashup. To do this, the Parameter Matcher calculates string-, constraint-, and structure-based similarity values. The structure-based similarity scores are computed using the semantic type annotations previously added by the Semantic Annotator. The evaluation shows that the structure-based similarity measures based on the semantic type annotations increase the average precision, which means that related parameters from different APIs can be found more efficiently. Another important factor that influences the efficiency of creating mashups is to program glue code that translates between the syntactically incompatible parameters of different APIs. This chapter also introduces the Glue Code Generator, which generates glue code from the parameter mappings supplied by the Parameter Matcher. What is novel about the Glue Code Generator is that it can generate a lot of the mashup's program logic, i.e., extracting the relevant parameters from JSON messages, syntactically translating the parameter values, and calling subsequent operations. In the evaluation, it is shown that 88% of the generated code does not need manual adjustments. The evaluation sample with three parameter mappings shows the validity of the approach. In future work, the sample must be enlarged in order to be able to make more general statements. Of particular interest is how often the generated translation functions can actually be used, as this has a significant impact on efficiency. Nevertheless, the Parameter Matcher and Glue Code Generator form a basis for an efficient creation of mashups.

8 Conclusion and Future Work

8.1 Summary

This dissertation introduces the novel IT service *Brokerage as a Service (BaaS)*, which substantially assists requesters and API providers in collaborating to create new mashups from existing APIs. This support comprises (1) annotating API specifications with semantic types from a global ontology to resolve terminological heterogeneity between requests and API specifications so that relevant API operations can be found effectively (2) deriving API protocols so that requesters can include all dependent operations calls required in their mashups, (3) identifying the input and output parameters that need to be exchanged between different APIs, and (4) generating the glue code from parameter mappings that makes operations from APIs interoperable. In particular, the contributions of this dissertation are the following:

1. **Effective API Annotation and Discovery:**

   BaaS includes the Semantic Annotator to help providers to find relevant semantic types of a global ontology to enrich their syntactic API specifications with ontological semantics. The terminological heterogeneity between API offers and requests that prevents effective service discovery is eliminated by annotating the requests and offers by a common set of semantic types, provided by a global ontology. The Semantic Annotator automatically derives search queries from the API specifications to search relevant semantic types. It is shown that using parameter names, the parameter description, and the operation name for building the search queries is the most effective. The lowercase transformation is proven to be the most effective matching technique. Stemming and adding synonyms only is decreasing effectiveness.

In this dissertation, it is shown that the effectiveness of the service discovery is increased when requests and API offers are annotated with the semantic types of a global ontology. It is shown in the evaluation that relevant operations from RapidAPI.com are found more effectively by the existing service matchmaker OWLS-MX3 [20] when operations are annotated with semantic types from the global ontology schema.org. Since RapidAPI.com, schema.org, and OWLS-MX3 are completely independent of each other, this evaluation simulates a realistic scenario.

2. **Mining API Protocols:**

   BaaS includes the API Protocol Miner that derives API protocols from call-logs using process mining. API protocols consist of control flow constructs and operation calls. Different kinds of control flow constructs from the specification languages OWL-S, WS-BPEL, and BPMN are categorized and then compared to the kinds of control flow constructs that can be discovered by the process mining algorithms Alpha Miner, Inductive Miner, and Heuristics Miner. The result is that process mining algorithms cannot distinguish between parallel and sequential control flows when the latter are executed in any order. Furthermore, the concrete conditions for branches in the control flow cannot be retrieved using any of the investigated algorithms. Call logs cannot be used directly for process mining because a process instance must be assigned to each call. In this dissertation, two methods are presented in which the process instance is assigned via the client host/user agent or an authorization token/sessionID. The evaluation shows that the API protocols discovered by the Heuristics Miner are very close to a manually created API protocol, even if the call-logs are noisy. Real-world call-logs are often noisy which is a challenge for process mining as noise promotes the creation of faulty API protocols. Between the Alpha Miner, the Inductive Miner, and the Heuristics Miner it is found that the Heuristics Miner is best suited for API protocol mining from call-logs, because it is robust against noise and can be adjusted easily to the different frequencies in the call-logs of different APIs. This can be achieved by adjusting the frequency thresholds. In the evaluation with the SIMPHERA API, the Heuristics Miner also showed the best results.

3. **Effective Parameter Matching:** The *Parameter Matcher* of the BaaS helps requesters to identify the necessary data flows between different APIs that are to be combined in a mashup. All input and output parameters of API operations involved in a mashup are matched pairwise and the similarity of each pair is assessed. The similarity of each parameter mapping is determined based on string-, constraint-, and structure-based similarity metrics. The structure-based similarity metric is calculated based on the type hierarchy of the semantic types added by the Semantic Annotator. Based on a real-world example it is shown that the effectiveness of the *Parameter Matcher* is considerably improved through structure-based similarity metrics.

4. **Glue Code Generation:** The *Glue Code Generator* helps the requesters to create the program code that makes the different APIs in a mashup interoperable. The glue code is generated from the parameter mappings produced by the *Parameter Matcher*. What is novel about the generated code is that it extracts relevant values from complex data structures and converts the data before it passes it to the subsequent operation. The Glue Code Generator predicts and injects proper conversion function into the program code using a heuristic. Based on a small sample, it is shown that 88% of the generated code is accurate, which shows the validity of the approach. Larger studies with larger samples are needed to confirm general validity. In this way, the Code Generator contributes in making mashup creation more efficient.

The BaaS approach (1) is *upward compatible* because it builds upon the existing corpus of REST APIs. (2) is *learnable* because providers and requesters can continue using the technologies they are familiar with and do not need special expertise, (3) is *scalable* because the approach implements effective search methods which are crucial to find relevant operations and parameter mappings in the large corpus of available APIs, (4) makes heterogeneous APIs *interoperable* as it generates the glue code that wires their inputs and outputs so they can be exchanged between each other, (5) makes API specifications *comprehensible* because it enriches them with ontological and behavioral semantics.

8.2 Future Work

Some aspects of the BaaS approach presented in this dissertation have to be addressed in future research. On one hand, the method for providers to link their API specifications to a global ontology plays a key role in this dissertation. On the other hand, semi-automatic support to link requests to the global ontology is out of the scope of this dissertation. Also, systematic methods to select an existing global ontology that is suitable to describe APIs or methods to develop and maintain new global ontologies need to be investigated in the future. Finally, the concepts that are proposed in this dissertation can also be used for other fields of applications: The API protocol specifications extracted from call-logs could be used to automatically make existing RESTful web services compliant with HATEOAS [2]. Additionally, the ontological annotations of the JSON response messages that are created using the *Semantic Annotator* can be used to automatically convert ordinary JSON response messages into the JSON-LD format to make ordinary RESTful web services usable for applications of the Semantic Web. These further research ideas are explained in detail below.

8.2.1 Linking Textual Requests with Ontologies

Terminological heterogeneity between requests and API offers prevents requesters from effectively finding relevant operations. In this dissertation, it is shown that annotating requests and API offers with semantic types from a global ontology eliminates heterogeneity and increases the effectiveness of service discovery. While the process of how providers annotate their API specification with semantic types is the focus of the thesis, the process of how untrained requesters can link textual service requests with semantic types has to be addressed in future works.

For this purpose, there are already approaches that are going in that direction: Bäumer [136] presents an approach for reducing ambiguity, incompleteness, and vagueness in textual requirement specifications, written in natural language. The approach includes the tool CORDULA which gives requesters feedback about the deficits in their requests and assists them in eliminating them. This particularly addresses resolving lexical ambiguity, which occurs when the meaning of the words used in a textual request is ambiguous.

Future work has to explore how lexical ambiguity can be eliminated by linking it to semantic types of a global ontology. On one hand, API providers would use the *Semantic Annotator* that is presented in this dissertation to link their API specifications with the global ontology. On the other hand, requesters would use an extended version of CORDULA that allows them to link single words in their textual requests with the global ontology. This way, even untrained requesters and providers would obtain sophisticated tools that help them to converge towards a common ontology.

8.2.2 Systematically Developing Global Ontologies

In this dissertation, schema.org is used as a global ontology to annotate APIs from RapidAPI.com just for evaluation. This evaluation shows that the semantic types from schema.org are insufficient to describe all the input and output parameters of APIs from RapidAPI. This is because global ontology and APIs have been created independently.

Another possibility for using independent off-the-shelf ontologies like schema.org is developing new global ontologies for a specific context. Huma [5] describes guidelines on how global ontologies are developed and maintained. However, no systematic, well-defined process to develop and maintain global ontologies has been proposed yet. For this purpose, a top-down and a bottom-up approach are conceivable: In the top-down approach, the global ontology is created independently from the APIs that are to be annotated. Developing a global ontology and APIs independently facilitates them being heterogeneous. In the bottom-up approach, however, existing API specifications are analyzed in order to learn a global ontology specifically for these existing APIs. Within this tailor-made global ontology, the semantic types derived from the respective specifications can then be linked together to resolve the heterogeneity between the various APIs. Because new APIs may be added all the time, the global ontology needs to be extended continuously. Future work will investigate whether the top-down or bottom-up approach is more practical. In addition, systematic processes for creating and maintaining global ontologies need to be explored.

8.2.3 Lifting the Maturity Level of Existing REST APIs

The Richardson maturity model [137] categorizes RESTful web services according to the degree to which they implement features of the REST architectural style. The maturity model defines four levels of maturity. RESTful web services that are on the fourth level implement HATEOAS. HATEOAS-conform web services send links to all possible states that are adjacent to the current state. HATEOAS decouples the client and the server applications so that they can evolve independently. However, many RESTful web services of today do not implement HATEOAS.

In this dissertation, a method is described on how API protocol specifications can be obtained from call-logs using process mining. Future work will examine whether these APIs protocols can also be used to automatically extend the response messages of RESTful web services by HATEOAS links. The idea here is to determine at the server side the current state in the API protocol on the basis of the recent API call. Based on the current state, all possible, adjacent hypermedia links are determined based on the transitions defined in the API protocol. The response messages are automatically expanded by the respective hypermedia links. This procedure could help providers in lifting their legacy RESTful web services to a higher maturity level.

Even though a RESTful web service is HATEOAS-conform, the data it produces can barely be used for many applications of the Semantic Web. This would require that the data it produces is in JSON-LD format [59]. The JSON-LD format is to link single attributes in a JSON response message to semantic types. However, many RESTful web services that exist today produce ordinary JSON and do not support JSON-LD. The idea is to use automatically enrich JSON response messages by semantic type annotations provided through the Semantic Annotator to make them utilizable for Semantic Web applications.

8.3 Final Remark

With the methods of Brokerage as a Service, requesters can find relevant APIs more effectively thanks to the semantic type annotations, use the APIs correctly through the API protocols derived from call-logs, identify the necessary data flows between the composed APIs more quickly, and pre-generate large parts of the program logic, to make the APIs interoperable with each other. The results of this dissertation thus make a substantial contribution to the efficient creation of mashups.

Bibliography

[1] A. A. Kalenkova, W. M. P. van der Aalst, I. A. Lomazova, and V. A. Rubin, "Process mining using BPMN: relating event logs and process models," in *Proceedings of the 19th International Conference on Model Driven Engineering Languages and Systems*, p. 123, 2016.

[2] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[3] G. A. Miller, "WordNet: A Lexical Database for English," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[4] M. Klusch, P. Kapahnke, S. Schulte, F. Lecue, and A. Bernstein, "Semantic web service search: A brief survey," *KI - Künstliche Intelligenz*, vol. 30, no. 2, pp. 139–147, 2016.

[5] Z. Huma, *Automatic Service Discovery and Composition for Heterogeneous Service Partners*. PhD thesis, Paderborn University, 2015.

[6] N. Oldham, C. Thomas, A. P. Sheth, and K. Verma, "METEOR-S web service annotation framework with machine learning classification," in *Proceedings of the International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pp. 137–146, 2004.

[7] B. C. N. Oliveira, A. Huf, I. L. Salvadori, and F. Siqueira, "Automatic semantic enrichment of data services," in *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pp. 415–424, 2017.

[8] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Supporting the creation of semantic RESTful service descriptions," in *Proceedings of 8th International Semantic Web Conference (ISWC)*, 2009.

[9] M. d'Aquin, M. Sabou, M. Dzbor, C. Baldassarre, L. Gridinoc, S. Angeletou, and E. Motta, "Watson: a gateway for the semantic web," *Proceedings of the 4th European Semantic Web Conference (ESWC)*, 2007.

[10] M. Cremaschi and F. D. Paoli, "A practical approach to services composition through light semantic descriptions," in *Proceedings of the 7th European Conference on Service-Oriented and Cloud Computing (ES-OCC)*, pp. 130–145, 2018.

[11] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, System Demonstrations*, pp. 55–60, 2014.

[12] A. Heß, E. Johnston, and N. Kushmerick, "ASSAM: A tool for semi-automatically annotating semantic web services," in *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, pp. 320–334, 2004.

[13] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable web-services," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, pp. 141–150, 2009.

[14] S. Dustdar and R. Gombotz, "Discovering web service workflows using web services interaction mining," *International Journal of Business Process Integration and Management*, vol. 1, no. 4, pp. 256–266, 2006.

[15] R. Gombotz and S. Dustdar, "On web services workflow mining," in *Proceedings of the Business Process Management Workshops (BPM)*, pp. 216–228, 2005.

[16] J. L. C. Izquierdo and J. Cabot, "Composing JSON-based web APIs," in *Proceedings of the 14th International Conference on Web Engineering (ICWE)*, pp. 390–399, 2014.

[17] M. H. Burstein, D. V. McDermott, D. R. Smith, and S. J. Westfold, "Derivation of glue code for agent interoperation," *Autonomous Agents and Multi-Agent Systems*, vol. 6, no. 3, pp. 265–286, 2003.

[18] F. Bülthoff and M. Maleshkova, "Restful or restless – current state of today's top web apis," in *Proceedings of the 11th Extended Semantic Web Conference (ESWC) – Satellite Events*, pp. 64–74, 2014.

[19] R. Meusel, C. Bizer, and H. Paulheim, "A web-scale study of the adoption and evolution of the schema.org vocabulary over time," in *Proceedings of the 5th International Conference on Web Intelligence, Mining and Semantics WIMS*, pp. 15:1–15:11, 2015.

[20] M. Klusch and P. Kapahnke, "OWLS-MX3: an adaptive hybrid semantic service matchmaker for OWL-S," in *Proceedings of the 3rd International Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web (SMR2)*, 2009.

[21] S. Schwichtenberg, C. Gerth, Z. Huma, and G. Engels, "Normalizing heterogeneous service description models with generated QVT transformations," in *Proceedings of the 10th European Conference on Modelling Foundations and Applications (ECMFA)*, pp. 180–195, 2014.

[22] G. Engels, B. Güldali, C. Soltenborn, and H. Wehrheim, "Assuring consistency of business process models and web services using visual contracts," in *Applications of Graph Transformations with Industrial Relevance*, pp. 17–31, 2008.

[23] S. Schwichtenberg, C. Gerth, and G. Engels, "RSDL workbench results for OAEI 2014," in *Proceedings of the 9th International Workshop on Ontology Matching collocated with (ISWC)*, pp. 155–162, 2014.

[24] S. Schwichtenberg and G. Engels, "RSDL workbench results for OAEI 2015," in *Proceedings of the 10th International Workshop on Ontology Matching collocated with (ISWC)*, pp. 192–199, 2015.

[25] S. Schwichtenberg and G. Engels, "Automatized derivation of comprehensive specifications for black-box services," in *Proceedings of the 38th International Conference on Software Engineering (ICSE) – Companion Volume*, pp. 815–818, 2016.

[26] S. Schwichtenberg, C. Gerth, and G. Engels, "From Open API to semantic specifications and code adapters," in *Proceedings of the 24th International Conference on Web Services (ICWS)*, pp. 484–491, 2017.

[27] S. Schwichtenberg, I. Jovanovikj, C. Gerth, and G. Engels, "CrossEcore: an extendible framework to use Ecore and OCL across platforms," in *Proceedings of the 40th International Conference on Software Engineering (ICSE) – Companion Volume*, pp. 292–293, ACM, 2018.

[28] B. Jazayeri and S. Schwichtenberg, "On-the-fly computing meets IoT markets - towards a reference architecture," in *Proceedings of the 14th International Conference on Software Architecture (ICSA) – Companion Volume*, pp. 120–127, 2017.

[29] B. Jazayeri and S. Schwichtenberg, "On the necessity of an architecture framework for on-the-fly computing," *Softwaretechnik-Trends*, vol. 37, no. 2, 2017.

[30] B. Jazayeri, S. Schwichtenberg, J. Küster, O. Zimmermann, and G. Engels, "Modeling and analyzing architectural diversity of open platforms," in *Proceedings of the 32nd International Conference on Advanced Information Systems Engineering (CAiSE)*, vol. 12127, pp. 36–53, 2020.

[31] SmartBear Software, "State of software quality API - latest trends & insights for 2023." `https://smartbear.com/state-of-software-quality/api/tools/`, 2023.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616: Hypertext transfer protocol – HTTP/1.1." `https://www.ietf.org/rfc/rfc2616.txt`, 1999.

[33] L. M. T. Berners-Lee, R. Fielding, "RFC 3986: Uniform resource identifier (uri): Generic syntax." `https://www.ietf.org/rfc/rfc3986.txt`, 2005.

[34] "Hypertext markup language HTML." `https://html.spec.whatwg.org/multipage/`, 2019.

[35] D. Crockford, "RFC 4627: The application/json media type for javascript object notation (json)." `https://www.ietf.org/rfc/rfc4627.txt`, 2006.

[36] J. Webber, S. Parastatidis, and I. Robinson, *REST in practice: Hypermedia and systems architecture.* O'Reilly Media, Inc., 2010.

[37] R. Chinnici, M. Gudgin, J.-J. Moreau, and S. Weerawarana, “Web services description language (WSDL) version 2.0 part 1: Core language.” `http://www.w3.org/TR/wsdl20/`, 2007.

[38] M. Hadley, “Web application description language.” `https://www.w3.org/Submission/wadl/`, 2009.

[39] J. Kopecký, K. Gomadam, and T. Vitvar, “hRESTS: An HTML microformat for describing RESTful web services,” in *Proceedings of the International Conference on Web Intelligence (WIC)*, pp. 619–625, 2008.

[40] “OpenAPI specification v3.1.0.” `https://spec.openapis.org/oas/v3.1.0.html`, 2021.

[41] “JSON schema.” `https://json-schema.org/specification.html`, 2019.

[42] T. Berners-Lee and M. Fischetti, *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. Turtleback, 2000.

[43] J. Euzenat and P. Shvaiko, *Ontology Matching*, vol. 18. Springer, 2007.

[44] “RDF schema 1.1.” `https://www.w3.org/TR/rdf-schema/`, 2014.

[45] “OWL 2 web ontology language structural specification and functional-style syntax.” `https://www.w3.org/TR/owl-syntax/`, 2012.

[46] S. Lohmann, S. Negru, F. Haag, and T. Ertl, “Visualizing ontologies with VOWL,” *Semantic Web*, vol. 7, no. 4, pp. 399–419, 2016.

[47] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, “OWL-S: Semantic markup for web services.” `http://www.w3.org/Submission/OWL-S/`, 2004.

[48] H. Lausen, J. de Bruijn, A. Polleres, and D. Fensel, “WSML – a language framework for semantic web services,” in *W3C Workshop on Rule Languages for Interoperability*, 2005.

[49] A. P. Sheth, K. Gomadam, and J. Lathem, “SA-REST: semantically interoperable and easier-to-use services and mashups,” *IEEE Internet Computing*, vol. 11, no. 6, pp. 91–94, 2007.

[50] D. Roman, J. Kopecký, T. Vitvar, J. Domingue, and D. Fensel, “WSMO-Lite and hRESTS: Lightweight semantic annotations for web services and RESTful APIs,” *Journal of Web Semantics*, vol. 31, pp. 39–58, 2015.

[51] M. Lanthaler and C. Guetl, “Hydra: A vocabulary for hypermedia-driven web APIs,” in *Proceedings of the 22nd International Conference on World Wide Web (WWW) – Companion Volume*, 2013.

[52] M. Cremaschi and F. De Paoli, “Toward automatic semantic API descriptions to support services composition,” in *Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, pp. 159–167, 2017.

[53] N. Mainas, E. G. M. Petrakis, and S. Sotiriadis, “Semantically enriched open API service descriptions in the cloud,” in *Proceedings of the 8th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 66–69, 2017.

[54] H. Dong, F. K. Hussain, and E. Chang, “Semantic web service matchmakers: state of the art and challenges,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 7, pp. 961–988, 2013.

[55] S. S. Gao, C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech, and M. Maloney, “W3C XML schema definition language (XSD) 1.1 part 1: Structures.” `http://www.w3.org/TR/xmlschema11-1/`, April 2012.

[56] “XSL transformations (XSLT) version 3.0.” `https://www.w3.org/TR/xslt-30/`, June 2017.

[57] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma., “Web service semantics – WSDL-S.” `http://www.w3.org/Submission/2005/SUBM-WSDL-S-20051107/`, 2005.

[58] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, “SAWSDL: semantic annotations for WSDL and XML schema,” *IEEE Internet Computing*, vol. 11, no. 6, pp. 60–67, 2007.

[59] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, “JSON-LD 1.1 – a JSON-based serialization for Linked Data.” `http://json-ld.org/spec/latest/json-ld/`, 2017.

[60] E. Rahm and P. A. Bernstein, “A survey of approaches to automatic schema matching,” *The International Journal on Very Large Data Bases*, vol. 10, no. 4, pp. 334–350, 2001.

[61] Y. Kalfoglou and M. Schorlemmer, “Ontology mapping: The state of the art,” *The Knowledge Engineering Review*, vol. 18, no. 1, pp. 1–31, 2003.

[62] N. F. Noy, “Semantic integration: A survey of ontology-based approaches,” *SIGMOD Record*, vol. 33, no. 4, pp. 65–70, 2004.

[63] P. Shvaiko and J. Euzenat, “A survey of schema-based matching approaches,” in *Journal on Data Semantics IV*, pp. 146–171, 2005.

[64] A. Doan and A. Y. Halevy, “Semantic integration research in the database community: A brief survey,” *AI magazine*, vol. 26, no. 1, p. 83, 2005.

[65] N. Choi, I.-Y. Song, and H. Han, “A survey on ontology mapping,” *SIGMOD Record*, vol. 35, no. 3, pp. 34–41, 2006.

[66] P. Shvaiko and J. Euzenat, “Ontology matching: State of the art and future challenges,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 1, pp. 158–176, 2013.

[67] S. Arifulina, *Solving Heterogeneity for a Successful Service Market.* PhD thesis, Paderborn University, 2016.

[68] G. Klyne and C. Newman, “RFC 3339: Date and time on the internet: Timestamps.” `https://www.ietf.org/rfc/rfc3339.txt`, 2002.

[69] M. Benerecetti, P. Bouquet, and C. Ghidini, “On the dimensions of context dependence: Partiality, approximation, and perspective,” in *Proceedings of the 3rd International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT)*, vol. 2116, pp. 59–72, 2001.

[70] P. Bouquet, J. Euzenat, E. Franconi, L. Serafini, G. Stamou, and S. Tessaris, “Specification of a common framework for characterizing alignment,” tech. rep., University of Trento, 2004.

[71] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, pp. 707–710, 1966.

[72] W. E. Winkler, “String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage,” in *Proceedings of the Section on Survey Research*, pp. 354–359, 1990.

[73] D. Harman, “How effective is suffixing?,” *Journal of the American Society for Information Science (JASIS)*, vol. 42, no. 1, pp. 7–15, 1991.

[74] M. F. Porter, “An algorithm for suffix stripping,” *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[75] R. Krovetz, “Viewing morphology as an inference process,” in *Proceedings of the 16th International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 191–202, 1993.

[76] A. Maedche and V. Zacharias, “Clustering ontology-based metadata in the semantic web,” in *Principles of Data Mining and Knowledge Discovery*, pp. 348–360, 2002.

[77] Z. Wu and M. S. Palmer, “Verb semantics and lexical selection,” in *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pp. 133–138, 1994.

[78] W. M. P. van der Aalst, “Service mining: Using process mining to discover, check, and improve service behavior,” *IEEE Transactions on Services Computing*, vol. 6, no. 4, pp. 525–535, 2013.

[79] E.F. Moore, “Gedanken-experiments on sequential machines,” in *Automata Studies*, pp. 129–153, 1956.

[80] C. A. Petri, *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[81] W. M. P. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[82] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from event logs containing infrequent behaviour," in *Proceedings of Business Process Management (BPM) Workshops*, pp. 66–78, 2013.

[83] E. Verbeek and J. Buijs, "Introduction to process mining with ProM." `https://www.futurelearn.com/info/courses/process-mining/`, 2016.

[84] "RDFa core 1.1 – third edition." `https://www.w3.org/TR/rdfa-core/`, Mar. 2015.

[85] M. Taheriyan, C. A. Knoblock, P. A. Szekely, and J. L. Ambite, "Rapidly integrating services into the linked data cloud," in *Proceedings of the 11th International Semantic Web Conference (ISWC)*, pp. 559–574, 2012.

[86] J. D. Lafferty, A. McCallum, and F. C. N. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the 18th International Conference on Machine Learning (ICML)*, pp. 282–289, 2001.

[87] "SPARQL 1.1 query language." `https://www.w3.org/TR/sparql11-query/`.

[88] M. N. Lucky, M. Cremaschi, B. Lodigiani, A. Menolascina, and F. D. Paoli, "Enriching API descriptions by adding API profiles through semantic annotation," in *Proceedings of the 14th International Conference on Service-Oriented Computing (ICSOC)*, pp. 780–794, 2016.

[89] Z. Zhang, "Effective and efficient semantic table interpretation using TableMiner(+)," *Semantic Web*, vol. 8, no. 6, pp. 921–957, 2017.

[90] A. Heß and N. Kushmerick, “Learning to attach semantic metadata to web services,” in *Proceeding of the 2nd International Semantic Web Conference (ISWC)*, pp. 258–273, 2003.

[91] A. Zaveri, S. Dastgheib, C. Wu, T. Whetzel, R. Verborgh, P. Avillach, G. Korodi, R. Terryn, K. M. Jagodnik, P. Assis, and M. Dumontier, “smartAPI: Towards a more intelligent network of web APIs,” in *Proceedings of the 14th Extended Semantic Web Conference (ESWC)*, pp. 154–169, 2017.

[92] S. Dastgheib, T. Whetzel, A. Zaveri, C. Afrasiabi, P. Assis, P. Avillach, K. M. Jagodnik, G. Korodi, M. Pilarczyk, J. de Pons, S. C. Schürer, R. Terryn, R. Verborgh, C. Wu, and M. Dumontier, “The SmartAPI ecosystem for making web APIs FAIR,” in *Proceedings of the 16th International Semantic Web Conference (ISWC) – Posters & Demonstrations and Industry Tracks*, 2017.

[93] X. Wang, X. Liu, J. Liu, X. Chen, and H. Wu, “A novel knowledge graph embedding based API recommendation method for mashup development,” *World Wide Web*, vol. 24, no. 3, pp. 869–894, 2021.

[94] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *Proceedings of the 1st International Conference on Learning Representations (ICLR)* (Y. Bengio and Y. LeCun, eds.), 2013.

[95] Y. Yao, H. Liu, J. Yi, H. Chen, X. Zhao, and X. Ma, “An automatic semantic extraction method for web data interchange,” in *Proceedings of the 6th International Conference on Computer Science and Information Technology (CSIT)*, pp. 148–152, March 2014.

[96] T. A. Farrag, A. I. Saleh, and H. A. Ali, “Toward SWSs discovery: Mapping from WSDL to OWL-S based on ontology search and standardization engine,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 5, pp. 1135–1147, 2013.

[97] T. W. Finin, L. Ding, R. Pan, A. Joshi, P. Kolari, A. Java, and Y. Peng, “Swoogle: Searching for knowledge on the semantic web,” in *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th*

Innovative Applications of Artificial Intelligence Conference, pp. 1682–1683, 2005.

[98] A. Karavisileiou, N. Mainas, F. Bouraimis, and E. G. M. Petrakis, "Automated ontology instantiation of OpenAPI REST service descriptions," in *Advances in Information and Communication*, pp. 945–962, 2021.

[99] W. Gong, X. Zhang, Y. Chen, Q. He, A. Beheshti, X. Xu, C. Yan, and L. Qi, "DAWAR: diversity-aware web APIs recommendation for mashup creation based on correlation graph," in *Proceedings of the 45th International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 395–404, 2022.

[100] "Web services business process execution language version 2.0." `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`, Apr. 2007.

[101] O. Liskin, L. Singer, and K. Schneider, "Teaching old services new tricks: adding HATEOAS support as an afterthought," in *Proceedings of the 2nd International Workshop on RESTful Design (WS-REST)*, pp. 3–10, 2011.

[102] M. Schur, A. Roth, and A. Zeller, "Mining behavior models from enterprise web applications," in *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 422–432, 2013.

[103] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "XES, XESame, and ProM 6," in *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE Forum)*, pp. 60–75, 2010.

[104] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli, "Mining behavior models from user-intensive web applications," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 277–287, 2014.

[105] W. M. P. van der Aalst and M. Pesic, "Specifying and monitoring service flows: Making web services process-aware," in *Test and Analysis of Web Services*, pp. 11–55, 2007.

[106] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE)*, pp. 204–213, 2007.

[107] I. L. Salvadori, B. C. N. Oliveira, A. Huf, E. C. Inacio, and F. Siqueira, "An ontology alignment framework for data-driven microservices," in *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pp. 425–433, 2017.

[108] I. L. Salvadori, A. Huf, B. C. N. Oliveira, R. dos Santos Mello, and F. Siqueira, "Improving entity linking with ontology alignment for semantic microservices composition," *International Journal of Web Information Systems*, vol. 13, no. 3, pp. 302–323, 2017.

[109] D. Serrano, E. Stroulia, D. H. Lau, and T. Ng, "Linked REST APIs: A middleware for semantic REST API integration," in *Proceedings of the 24th International Conference on Web Services (ICWS)*, pp. 138–145, 2017.

[110] J. Klímek and M. Necaský, "Generating lowering and lifting schema mappings for semantic web services," in *Proceedings of the 25th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pp. 29–34, 2011.

[111] L. Liu, M. Bahrami, and W. Chen, "Automatic generation of IFTTT mashup infrastructures," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, pp. 1179–1183, IEEE, 2020.

[112] T. Rodrigues, P. Rosa, and J. Cardoso, "Moving from syntactic to semantic organizations using JXML2OWL," *Computers in Industry*, vol. 59, no. 8, pp. 808 – 819, 2008.

[113] M. C. Platenius, *Fuzzy Matching of Comprehensive Service Specifications*. PhD thesis, Paderborn University, 2016.

[114] S. Schwichtenberg, "Ontology-based normalization and matching of rich service descriptions," Master's thesis, Paderborn University, 2013.

[115] "RFC 5322: Internet message format." `https://datatracker.ietf.org/doc/html/rfc5322`, 2008.

[116] C. Bizer, R. Meusel, and A. Primpeli, "Web data commons – RDFa, microdata, and microformat data sets." `http://webdatacommons.org/structureddata/`.

[117] H. Liu and P. Singh, "ConceptNet – a practical commonsense reasoning tool-kit," *BT Technology Journal*, vol. 22, no. 4, pp. 211–226, 2004.

[118] Y. Li, Z. Bandar, and D. McLean, "An approach for measuring semantic similarity between words using multiple information sources," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 4, pp. 871–882, 2003.

[119] M. Klusch, "Overview of the s3 contest: Performance evaluation of semantic service matchmakers," in *Semantic Web Services: Advancement through Evaluation*, pp. 17–34, 2012.

[120] J. Kopeckỳ, D. Roman, M. Moran, and D. Fensel, "Semantic web services grounding," in *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW)*, pp. 127–127, IEEE, 2006.

[121] "Business process model and notation 2.0.2." `https://www.omg.org/spec/BPMN/2.0.2/PDF`, Jan. 2014.

[122] W. Van Der Aalst, "Data science in action," in *Process Mining*, pp. 3–23, Springer, 2016.

[123] W. Van Der Aalst, *Process mining: discovery, conformance and enhancement of business processes*, vol. 2. Springer, 2011.

[124] A. J. M. M. Weijters and J. T. S. Ribeiro, "Flexible heuristics miner (FHM)," in *Proceedings of the Symposium on Computational Intelligence and Data Mining (CIDM)*, pp. 310–317, 2011.

[125] S. J. Leemans, *Robust Process Mining with Guarantees.* PhD thesis, Technische Universiteit Eindhoven, 2017.

[126] A. K. A. de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters, "Workflow mining: Current status and future directions," in *Proceedings of the Confederated International Conferences, CoopIS, DOA, and ODBASE*, pp. 389–406, 2003.

[127] D. Bayomie, I. M. A. Helal, A. Awad, E. Ezat, and A. E. Bastawissi, "Deducing case ids for unlabeled event logs," in *Proceedings of the 13th International Workshops on Business Process Management (BPM)*, pp. 242–254, 2015.

[128] M. A. Bayir, I. H. Toroslu, A. Cosar, and G. Fidan, "Smart miner: a new framework for mining large scale web usage data," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pp. 161–170, 2009.

[129] D. R. Ferreira and D. Gillblad, "Discovering process models from unlabelled event logs," in *Proceedings of the 7th International Conference on Business Process Management (BPM)*, pp. 143–158, 2009.

[130] R. F. Dell, P. E. Román, and J. D. Velásquez, "Web user session reconstruction using integer programming," in *Proceedings of the International Conference on Web Intelligence*, pp. 385–388, 2008.

[131] M. Spiliopoulou, B. Mobasher, B. Berendt, and M. Nakagawa, "A framework for the evaluation of session reconstruction heuristics in web-usage analysis," *INFORMS Journal on Computing*, vol. 15, no. 2, pp. 171–190, 2003.

[132] B. Berendt, B. Mobasher, M. Nakagawa, and M. Spiliopoulou, "The impact of site structure and user environment on session reconstruction in web usage analysis," in *Proceedings of the 4th International Workshop on MiningWeb Data for Discovering Usage Patterns and Profiles (WEBKDD)*, pp. 159–179, 2002.

[133] "RFC 2109: HTTP state management mechanism." `https://www.rfc-editor.org/rfc/rfc2109`, 1997.

[134] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "On the role of fitness, precision, generalization and simplicity in process dis-

covery," in *Proceedings of the Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE*, pp. 305–322, 2012.

[135] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design patterns: Elements of reusable object-oriented software*, vol. 49. 1995.

[136] F. S. Bäumer, *Indikatorbasierte Erkennung und Kompensation von ungenauen und unvollständig beschriebenen Softwareanforderungen.* PhD thesis, Paderborn University, 2017.

[137] M. Fowler, "Richardson maturity model." `https://martinfowler.com/articles/richardsonMaturityModel.html`, 2010.