RESTalk

A Visual and Textual DSL for Modelling RESTful Conversations

Ana Ivanchikj

RESTalk

A Visual and Textual DSL for Modelling RESTful Conversations

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Ana Ivanchikj

under the supervision of
Prof. Dr. Cesare Pautasso

January 2021

Dissertation Committee

Prof. Dr. Cinzia Cappiello	Politecnico di Milano - Milan, Italy
Prof. Dr. Mathias Weske	Hasso Plattner Institute - Potsdam, Germany
Prof. Dr. Walter Binder	Università della Svizzera italiana - Lugano, Switzerland
Prof. Dr. Matthias Hauswirth	Università della Svizzera italiana - Lugano, Switzerland

Dissertation accepted on 25 January 2021

Research Advisor	PhD Program Director
Prof. Dr. Cesare Pautasso	**Prof. Dr. Walter Binder, Prof. Dr. Silvia Santini**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Ana Ivanchikj
Lugano, 25 January 2021

To everyone who provided their selfless support during my PhD endeavour

In memoriam of Prof. Dr. Florian Daniel

One LEGO block by itself is no fun at all.

Jason Bloomberg

Abstract

Digitalization is all around us, even more so in pandemic times where substantial part of our lives has been moved online. One of the key enablers of digitalization are the Application Programming Interfaces (APIs) which enable the communication and exchange of data between different systems. They abstract from the implementation details of the underlying systems and allow for the monetization of digital assets paving the way to innovative services, bringing together different business partners, even in traditionally closed sectors such as banking.

In this dissertation we focus on a particular type of APIs which conform to the REpresentation State Transfer (REST) architectural style constraints due to their dominance in the API landscape. Although Fielding defined REST as architectural style back in 2000, our state of the art review has identified a gap when it comes to modeling the behaviour of REST APIs. Existing approaches are not domain specific and as such fail to emphasise important facets in RESTful interactions. Since APIs need to satisfy multiple clients, it is not always possible to provide them with operations dedicated to achieving the specific goals of individual clients. Instead, clients often need to perform multiple interactions in a specific order to achieve their goals. We call the set of possible sequences of interactions to achieve a goal a RESTful conversation.

Visualizing complex RESTful conversations in a domain specific model can help streamline their design by facilitating discussion and common knowledge sharing, but also by constraining the supported API behavior from the combinatorial set of all possible interaction sequences. This enables service providers to maintain and evolve their APIs. Visual models can also facilitate client developers' understanding of a given API, and speed up the identification of the correct sequence of calls to be made given a goal. Based on our study of the characteristics of RESTful conversations, we introduce the design of RESTalk: a Domain Specific Language (DSL) for modelling the behaviour of REST APIs. The language supports single client to single server conversations, but also multiple clients talking to the same server in order to achieve a common goal, as well as composite layered conversations. We have designed RESTalk iteratively with a

frequent feedback from users while modelling different use-cases. Since its targeted users are developers, we propose both a graphical and a textual syntax for the language. We take an innovative approach in the design of the textual grammar, which leverages on a mining algorithm to generate the alternative control flow of the RESTful conversation and on a graph layout algorithm to automatically produce the visual diagram. Thus, the textual DSL has a log-like form with a minimal use of keywords. It aims at decreasing the cognitive load of the modeler while increasing the efficiency of model generation. Further evaluations are necessary to verify the actual benefits of the textual DSL over the graphical DSL. We have evaluated the expressiveness of RESTalk with real-world examples of RESTful conversations and patterns, and performed controlled experiments to attempt to determine the effectiveness and efficiency of the visual diagrams in facilitating the understanding of a given API.

Acknowledgements

As the famous proverb says “it takes a village to raise a child”, it also takes a whole community to raise a PhD student, both an academic community and a personal community, although often the two begin to intertwine with time. I experienced some of the worst and some of the best moments personally and professionally during my PhD endeavour, and my advisor Prof. Cesare Pautasso was always there with his comprehensive nod, pushing me when needed and giving me freedom once I got back on the right track. I heard today a person being described as a volcano of ideas, and this is what Prof. Pautasso represents for me. He spent countless hours discussing with me, inspiring me with his new ideas, patiently explaining and waiting for me to fill up my knowledge gaps due to my different educational background. I always left his office with clarification of many doubts, and a creation of many new doubts and ideas. He supported me in trying a workflow performance research path, and gave me the courage to go back to the RESTful conversation modeling as research path when the other one came to a dead end. I will always remain profoundly grateful to this wise discreet man for his fatherly advises and support.

As I love teaching and put my heart to it, sometimes at an expense of my free time and sleep, in addition to Prof. Pautasso, I would also like to thank another great teaching reference, Prof. Matthias Hauswirth. I really enjoyed his mastery checks and his Informa platform while taking his Java course, and I learned so much about teaching itself from his PhD seminar on How Learning Works. His dedication to teaching is amazingly unselfish and creative.

I would also like to express my gratitude to Prof. Florian Daniel, a very positive person who I will always remember with a smile on his face. This thesis is in memoriam of his personality, his valuable constructive feedback on my thesis proposal work, the numerous discussions on broad topics we had over a coffee or a lunch. Words of gratitude also go to Prof. Cinzia Cappiello who accepted my request to join the committee at a last minute. I learned a lot about ERPs while assisting her courses, enjoyed our discussions regarding student projects and grading, and enjoyed our coffee breaks.

I would also like to thank Prof. Mathias Weske who I met at my very first conference as a PhD, the BPM 2015 conference in Innsbruck, where I showed him the first work on RESTalk, a nameless DSL at that point, which I was so nervous to present at the ECSA conference the week after. His feedback during the breaks at different BPM conferences as well as on my thesis proposal was very valuable for the final outcome of this work. My gratitude also goes to Prof. Walter Binder and his feedback during the thesis proposal presentation.

Another volcano of ideas which I spent countless hours discussing with, and who I learned a lot from, is Vincenzo Ferme. He was a co-supervisor of my master thesis, co-author of numerous publications and co-traveler to different conferences during my first years as a PhD. His ambition in life and passion for informatics is inspiring.

Of course these past 6 years would not have been fun if I were alone in the office, renamed to the Nerfds office in the Skype chat after Andrea appeared with 5 Nerf guns at the door as a new stress release method. Thank you Andrea for the guns, and for all the coffee breaks (thanks God you started drinking coffee when you became a PhD). The birthday muffins you prepared for me with Masiar will remain in history, as will the barbeques at Vassilis's place. I am sure the new members of the SW design office, Souhaila and Fabio will continue the good office vibe we created.

A warm thank you note goes to my close friends spread across many countries, and to "noi 7" my close group of friends here in Lugano, for all of their support when it was time to relax, but also when it was time to say no to their invitations due to an upcoming deadline. Special thanks to Miki, we have been through hell and back together in this period, but I would not change a thing about it.

Of course a huge life-long thank you goes to my family, for their unconditional love and support during all of my academic career. When I was on the crossroad after finishing my Master's degree they supported me into deciding for a PhD, and stood by me every step of the way. As my father said to me once "my role as a parent is to give you the means to learn so that you can fly away and succeed when the time comes". And last but not least, huge thanks to my husband to-be, Kevin. He stood by me in the last 4 years of my PhD and made everything more beautiful. He supported me and made me smile when I was nervous and stressed out, and pampered me with coffee and food during the thesis writing marathon. Thank you for all the love and patience.

Contents

Figures

Tables

Part I

Motivation and Context

Chapter 1

Introduction

1.1 Context

Web Application Programming Interfaces (hereinafter APIs) allow different systems, built on top of heterogeneous technology, to interact with each-other over the network, thus enabling remote access to services [173]. The main benefit is that these type of interactions do not require deep knowledge of the opposite system [114]. Emerging technologies, such as Cloud services [132], Service mashups [37], and Microservices [148] all make use of APIs, driving their improvement, not only in terms of design and performance, but also in terms of usability. APIs are used on daily basis in many different domains, from checking weather and traffic, to updating your social media status, sending messages, to even making payments. People working on digital transformation initiatives state that APIs play significant role in those initiatives [166]. In fact traditionally closed sectors, such as banking, are opening up their businesses through APIs [203], in what is known as the Open Banking initiative whose goal is to increase competition among financial institutions and boost innovation. This initiative is partially arising from the legislative changes in the Payment Services Directive (PSD2)[1], which came to force in 2018 and required banks to share customer data with regulated third-parties. As the years pass by this initiative is now seen as an opportunity to meet important client needs such as customized products and solutions, high-degree of automation and end-to-end integration, as well as hybrid digital-personal relationships [114]. "Innovative companies have discovered that APIs can be used as an interface to the business, allowing them to monetize digital assets, extend their value proposition with partner-delivered

[1] https://www.ecb.europa.eu/paym/intro/news/articles_2018/html/1803_revisedpsd.en.html

capabilities, and connect to customers across channels and devices" [159]. That said, APIs are becoming the center of ecosystems bringing together different business partners and disrupting existing markets (e.g., Uber). APIs are creating essentially a new form of business model innovation [145], where companies are monetizing their data and changing their value chain [96]. Depending on who is the target user, we can distinguish between internal APIs, partner APIs and external APIs. Internal APIs are used only inside the company who created them, partner APIs are only shared with integration partners, while external APIs are public APIs openly available to everyone. The results of recent industry survey [166] show dominance of the internal APIs (56.96%) in respect to partner APIs (26.75%) and external APIs (16.29%), but that might change in the future moving more towards external APIs. Although it is hard to tell the exact number of APIs in the world, the ProgrammableWeb[2], a popular API repository, counts almost 24'000 publicly available APIs in 2020, more than half of them being added in the last 6 years.

There has been a recent shift in software engineering towards API-driven development, where the key focus in the development is the API [71; 20]. Thus, the term "API First", which in a recent State of the API survey has been defined as "defining and designing APIs and schema before beginning development" [166], has been coined and has been gaining on popularity. The survey has discovered that respondents with more then 6 years of experience with API development are more likely to be "API First" leaders and as such are more likely to focus on external public APIs. Such an "API First" or "contract first" paradigm fosters an agile approach, not only in the service development, but also in the API development and allows for the creation of the API designer role, which might not always coincide with the API developer role. The *API designer* has to define the API interface, discover endpoints that are of interest to potential clients and map them to back-end data or functionalities [96]. The *API developer* will then be responsible for the implementation of the API. However, API initiatives do not only involve the API designers and the developers in charge of the technical implementation of the API (hereinafter API developers), but also the developers who implement the client which uses the API (hereinafter *API client developers*) [69]. API client developers "end up choosing the API that is easy to use, easy to integrate, well-documented and easy to get started with" [20].

According to the ProgrammableWeb classification, APIs can be designed using different styles and protocols, such as Remote Procedure Calls (RPC), Simple Object Access Protocol (SOAP), REpresentation State Transfer (REST) etc. [19;

[2] `http://www.programmableweb.com`

112], and recently GraphQL[3]. ProgrammableWeb highlights the dominance of REST APIs, as 83% of the registered APIs on this directory use HTTP methods for client-server communication[4]. The 2020 State of the API Report based on a Postman survey with over 13'500 industry practitioners [166] confirms the trend as 93.4% of the respondents claim to use REST, with the other most popular styles, but with significantly less adoption, being webhooks (34.4%), SOAP (33.4%) and GraphQL (22.5%). REST dominance has also been confirmed by empirical studies [184; 146], which among other aspects, explored how service to service communication is implemented in industry. REST is an architectural style first presented by Roy Fielding's PhD thesis in 2000 [54]. REST APIs are structured around resources, which represent abstractions of information of potential interest for the API users. Unique Resource Identifiers (URIs) are used to access the information which is delivered as resource representation containing data, metadata and hypermedia links. An interaction between a client (API consumer) and the API may result in a creation or deletion of a resource, or the update or retrieval of the resource representation depending on the combination of the HTTP method and the URI. In practice, the REST style is preferred to SOAP as it supports different data formats, provides for loose-coupling and is less verbose which renders it more scalable [236]. Having in mind such dominance of REST APIs, this research focuses only on REST APIs, considering the other API types out of scope.

1.2 Problem Statement

APIs provide powerful abstraction mechanisms, however such abstraction does not come for free. The structure and the behaviour of the APIs need to be well understood both by the API developers and by the API client developers. Software engineering is a social and collaborative activity [99], with communication and knowledge sharing being among the most effort-requiring activities in software engineering projects [100]. Documentation aims at facilitating such knowledge sharing, and models have been frequently used for documenting software applications, as they abstract from implementation details [180]. Such models ought to be used for documentation and knowledge sharing regarding the structure and behaviour of APIs as well. Our literature review has shown that in the REST APIs domain existing documentation solutions cover well the structure of the API,

[3] https://graphql.org

[4] https://www.programmableweb.com/news/json-clearly-king-api-data-formats-2020/research/2020/04/03

however we have identified a gap in solutions targeting the documentation and knowledge sharing regarding the behaviour of REST APIs. In this dissertation we aim to address this gap.

1.3 Research Questions

Having identified the above mentioned research gap we have formulated a set of research questions which we have divided into logical sections. The ultimate objective of this research is to study the modeling and visualization of the behaviour of REST APIs with the aim of facilitating and improving their understanding, design, and usage.

1.3.1 RESTful Conversations

RQ1: *What are the entities and constraints that are needed to model the interactions with an API which is compliant with the REST architectural style?*

The characteristics of the interactions with a given API depend on the API style and the protocol used. As our research focuses on REST APIs, our first research question refers to defining the entities, and constraints which are characteristic for interactions with a REST API. The term RESTful conversations was first coined by Haupt, Leymann and Pautasso [81] to "indicate a set of basic HyperText Transfer Protocol (HTTP) request-response interactions that are driven by the same client interacting with one or more RESTful Web services". In this thesis we broaden the definition to also include multiple clients interacting with the same REST API. The characteristics of RESTful conversations are dictated by the REST architectural style constraints which we discuss in detail in Sec. 2.2. The communication primitives are dictated by the uniform interface constraint requiring the use of an HTTP method and URI in every request, and an appropriate status code, and when applicable hyperlinks to next possible actions, in the response. The stateless communication constraint requires that each interaction is initiated by the client and contains all relevant data to enable the server to process the request. Thus, a RESTful conversation is controlled by the server sending hyperlinks for resource discovery, but is driven by the client who can decide which hyperlink to follow next. In order to answer RQ1, after defining the RESTful conversation constraints in Sec. 2.2, in Sec. 4.2.1 we propose a concept dictionary of the relevant entities in a RESTful conversation and a metamodel to define the relationship between the entities.

1.3.2 Modelling REST APIs

RQ2: *What are the shortcomings of existing solutions for modelling REST APIs and how can those shortcomings be overcome?*

The existing industry tools for REST API documentation (e.g., RAML[5], Swagger[6], Blueprint[7], Mashape[8]) focus on **structural and data modeling aspects**, and as such do not capture the dynamics of the client-server interactions. At the end of 2015 the OpenAPI initiative[9] has been launched to promote the effort of standardizing the description of REST APIs. The Swagger 2.0 specification has become the basis for the proposed standard OpenAPI Specification (OAS), which is a vendor-neutral, portable and open specification for a YAML/JSON-based description format for APIs. OAS underwent a major revision towards the OAS 3.0[10]. This revision has introduced the notion of a Link Object which "provides a known relationship and traversal mechanism between responses and other operations", thus recognizing the importance of the knowledge of the sequences of interactions, in addition to the knowledge of the operations that can be performed on a single resource. Although the newly proposed link object has not been extensively used in industry yet, as service providers upgrade from OAS 2.0 to OAS 3.0, the use of this object can making it possible to use existing documentation to generate feasible RESTful conversations.

Visual notations have been used in software engineering since the 1940s [142]. Gurr [79] argues that diagrammatic representation systems are more effective than textual ones, as certain conclusions can be directly derived from the diagram with no need to make logical inference. For instance, in case of nested interactions, the nesting is more evident from a diagram than when explained textually. Visual information is also more likely to be remembered due to the picture superiority effect [142]. Jolak et al. [99] in their controlled experiment have also observed that graphical representation, in addition to providing better recall, also fosters active discussion and creative conflict discussions while decreasing the effort of conversation management compared to textual representation. As mentioned in [212], the main purposes of using conceptual visual modelling are communication, design, and understanding. Having in mind the goal of this the-

[5] http://raml.org
[6] http://swagger.io
[7] https://apiblueprint.org
[8] https://www.mashape.com
[9] https://www.openapis.org
[10] https://www.openapis.org/blog/2017/03/01/openapi-spec-3-implementers-draft-released

sis to facilitate exactly those aspects, we argue that visualizing RESTful conversations can be a potentially beneficial approach. In existing research, non-domain specific languages such as Petri Nets [117], UML state machines [187; 169; 170], or UML sequence diagrams [81] have been used to visually model the **behaviour of REST APIs**. There have also been attempts to visualize RESTful conversations with BPMN Choreographies [151] as Domain Specific Language (DSL) in the domain of visual modeling of interactions between business processes.

Using existing modelling languages has its advantage both in terms of the time consuming task of designing a DSL, as well as in terms of language dissemination and use of existing tool support for the DSL. However, the drawback of using general purpose languages, or borrowing notations from other domains, to visualise RESTful conversations, is that the REST domain specific facets discussed in RQ1 are not emphasised, are added as comments, or are simply omitted (such as for example the information regarding the flow of resource discovery). Lindland et al., in their framework for understanding the quality in conceptual modeling [119], claim that a very important aspect of a modeling language is precisely its domain appropriateness. Gurr [79] also argues that reasoning is made easier for the user if the structure of the representation matches the primary concepts over which one must reason. Representations which are too abstract, or unsupportive of basic reasoning tasks, create difficulties for the user. Cortes-Cornax et al. [29] emphasize the same when evaluating the quality of BPMN Choreographies. They state that "the language must be powerful enough to express anything in the domain but no more".

That said, in this thesis we propose RESTalk, a visual and textual DSL for modeling the behaviour of REST APIs. We have used BPMN Choreography as a starting point of the visual representation of the language in order not to built it from scratch. BPMN Choreograpies are part of the BPMN [101] ISO standard for process modeling and they focus on modeling the coordination in cross-organizational interactions. Process are similar to RESTful conversations in that they are a sequence of activities which lead to a desired outcome [210]. REST navigational style of following links naturally supports a workflow style of interaction between resources [191]. After all, service compositions, which are a sequence of calls to Web Services, are essentially fully automated processes [147]. Visual modelling of the behaviour of workflows [45] has been around for quite some years, and it has been proven to facilitate process understanding and optimization [109; 44]. Our aim is to transfer similar benefits in the REST domain by modeling and visualization of RESTful conversations. With RESTalk, we are modifying and extending the BPMN Choreography diagrams to fit the intrinsic properties of the REST architectural style. The extension is described in detail in

Sec. 4.2.2. We have been developing RESTalk's constructs through different iterations, as we have been using it to model different use cases [160; 87; 161; 91] described in Sec. 6.2. We compare RESTalk to standard visual notations for modeling interactions in Sec. 7.2.

1.3.3 Modelling Techniques Support

RQ3:*What type of a tool support can be built around a DSL for modeling REST APIs behaviour?*

The first set of tools that ought to be built around a DSL are DSL editors. As RESTalk is a visual language the intuitive solution could be to build a graphical editor. However, RESTalk targets developers who are more inclined toward textual editors due to the traditionally textual form of programming languages [22]. Having this in mind, we have studied over 30 existing editors for visual models, which can be textual editors or graphical editors, or both, and we have come up with a maturity model for going from text to visual models which we discuss in Sec. 3.3. The maturity model identified five maturity clusters, going from just binary or encoded textual model representation to hybrid tools with both textual and graphical editors. We went on with analysis of the design of the textual DSLs in the studied tools, identifying several approaches, such as the textual DSL being visually close to the graphical DSL (e.g., yUML), the textual DSL being close to programming languages syntax (e.g., ZenUML), or the textual DSL being close to natural language (e.g., PlantUML, NaturalMash [5]).

We have decided to take a different approach for the RESTalk textual representation and design a log-like DSL in order to leverage on the power of existing process mining algorithms to aid the modelling of RESTful conversations. Mining has been around for decades, and has gained on particular popularity in the visual modeling of business processes [208], where it has been proven to facilitate process understanding and optimization [109; 44]. Process mining basically takes as input process execution logs and through the use of algorithms reconstructs the process workflow. As discussed in RQ2, workflows are similar to RESTful conversations as they both focus on the sequence in which activities/interactions are performed. Thus, we had the novel idea of using a mining algorithm to reconstruct the sequences of interactions described by the API designer/developer in the form of simplified logs in order to generate the visual RESTalk diagram. This is a novel use of mining algorithms as a modelling aid on top of a textual DSL, thus extending their utility also to cases when execution logs are not available. The proposed textual DSL is discussed in Sec. 4.2.3 and

a proof of concept implementation of a textual editor of the core version of the same is presented in Sec. 5.2.2.

In addition to editors for designing models, it is also important to create an ecosystem of tools around the DSL which ideally integrate with existing related tooling in the domain, thus enabling the wider adoption of the DSL. In the case of RESTalk this would include integration with OAS as a widely accepted standard documentation for the structure of a REST API. Such integration would enable an integrated documentation describing both the structure and the behaviour of the API. In the spirit of Model Driven Engineering (MDE) [107], RESTalk models can also be used to translate a given goal of the client to a skeleton of client implementation code. RESTful conversation models can also be derived a posteriori from the actual usage of the REST API, by mining its logs. A prototype mining tool we have developed as part of a bachelor thesis is presented in Sec. 5.3. The mining result can bring to interesting insights regarding how different clients actually use REST APIs, and can thus help developers detect unexpected usage patterns of their APIs by comparing different clients' conversation logs. It can also pinpoint frequent interaction sequences which are worth optimizing as they are being used by most of the clients. Another idea is for a tool to be built on top of the mining algorithm to perform conformance checking to detect divergence of the actual conversation from the expected conversation [139]. We discuss our envisioned tooling ecosystem around RESTalk in detail in Sec. 5.1.

1.3.4 DSL Benefits

RQ4: *How can REST API developers and API client developers benefit from modeling the API's behaviour?*

The benefits that can be derived from modeling API's behaviour depend on the phases of the API lifecycle the models are used in. As evident in Fig. 1.1 both the API provider and the API consumer are involved in the API lifecycle.

In the **design phase**, when the "API First" approach is used fast feedback on the API's design is essential. Such feedback can be facilitated by modelling the designed API's behavior as it can reveal undesired traits, such as a chatty API or inappropriate selection of resources, or missing resources given the business goals. Fixing design mistakes in a published API is much harder than an error inside a Web application due to the impact on the API clients, which are not always known to the API designer [234]. When the API provider creates separate roles for the API designer and the API developer, modeling API's behavior can facilitate the communication and common understanding of the API design by

the API developer. Depending on the tooling around a DSL, behavioral models can also be used in the **creation phase** to transform models to code, a well known practice in the model-driven development community [190]. They can also be used in the **testing phase** to create test cases to verify that the API behaves as expected [115].

Before publishing the API so that it can be discovered by the API consumer, behavioural models can be used to complement the API documentation. A survey on what makes APIs hard to learn [176] has revealed that 78% of the respondents learn APIs by reading documentation. However, as indicated by a recent study of REST APIs [146], in more than half of the analyzed APIs, documentation is missing. And even when it is available, what is often missing, is the high-level design of API's behavior. In order to pass from the **discovery phase** to the **development phase** in the API consumer, the API client developers need to learn and understand how to use the API given their goal. Simple APIs, designed to fulfill the requirements of a single client, can minimize the number of exposed operations (method+URI pairs). Public and reusable APIs, however, are typically used by multiple clients, built at different times and operated by different organizations. In this case, API's size and complexity would grow out of control if an operation is added to satisfy the needs of each type of client using it. Thus, although reusable APIs publish the minimal set of operations to satisfy all clients, this set can sometimes reach over 50 operations [25; 146], requiring the API client developers to compose such operations through multiple interactions, thus having a conversation with the API to achieve their goals. This implies that API client developers need to have good understanding of the APIs they ought

Figure 1.1. API Lifecycle

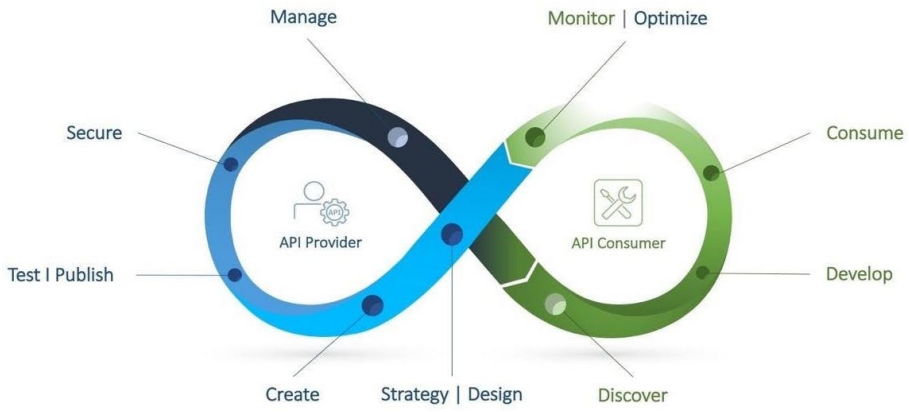
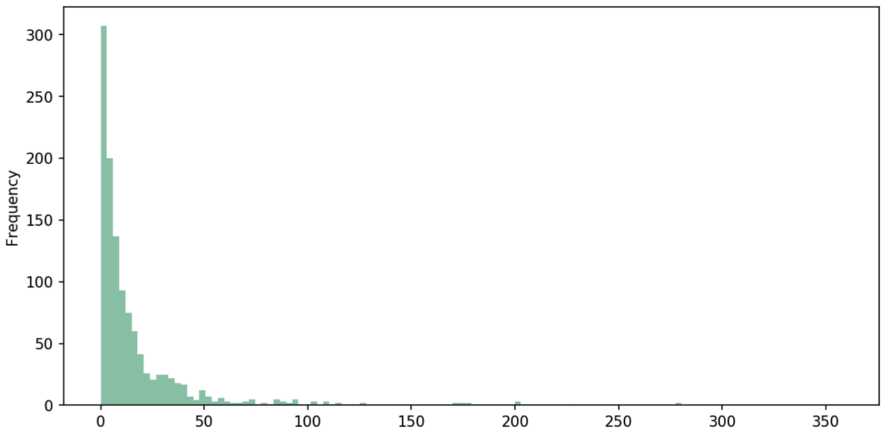


*Image taken from `https://dzone.com/`

Figure 1.2. Histogram of number of operations in a set of REST APIs



*Source: [106]

to use, as well as the flow of the conversation that will enable them to achieve their goal which is where conversation models can help.

A REST APIs visualization tool implemented as part of a master thesis we have supervised [106] has analyzed 1176 REST APIs showing that, although rare, there are APIs, such as Gitlab, with over 350 operations which can result in lengthy API conversations. However, as evident in Fig. 1.2 most of the analyzed APIs have few operations, which might create the need of composite conversations as the one described in Sec. 6.2.4 where multiple APIs are used to create a single service provider.

The API lifecycle does not end with the API consumption, as APIs evolve and change over time. One way to look at APIs is as a promise made by the API provider to the API consumer [130]. The API provider makes a promise that the API works now, that it will keep working in the future, and that it will be able to evolve and improve while still keeping the promise [122]. A structural description of the API allows for an exponential number of possible sequences of interactions, behavioral models are necessary to set up constraints on the supported behavior so that clear and manageable promises can be set by the API provider and maintained in future versions of the API.

In addition to the above stated reflection on the theoretical potential benefits of modelling the behaviour of REST APIs for the API developers and API client developers, we have also conducted an exploratory survey which we discuss in Sec. 6.1. The survey has showed that 38% of the respondents already use UML diagrams or in-house developed notations for modeling RESTful interactions. This

shows that the need for and benefits of modelling the REST API behaviour have already been identified in industry. One of the benefits stated by the survey respondents is improved productivity by improving the team's understanding of the interactions and by driving design discussions. As a product developer stated in the survey: "REST APIs usually include trivial conversation patterns. Regardless of their triviality, those should be explicitly noted in technical documentation. For high-level design that is intended to facilitate the design processes and possible conversations among different stakeholders, identifying conversation patterns can decrease unnecessary information and thus prove time-and energy-saving". On a different note, the survey results have also showed that the respondents find the core RESTalk easily understandable and less time consuming than the technique that they currently apply. However, as the survey was just used as an exploratory research, we make no inference of the results on the larger population of API developers.

RQ5: *How effective and efficient is the visual model created with RESTalk in facilitating the understanding and the use of a given API?*

The only way to answer this research question is by empirical research. To that end we have designed and conducted a controlled experiment with bachelor students (see Sec. 7.3). The experiment consisted of solving different tasks asking the participants to state the necessary sequence of interactions in order to achieve a given goal by using the Imgur API. As the goal was to study the effectiveness and efficiency of RESTalk diagrams, the independent variable was the used API documentation with one treatment being only OAS documentation and the other one being OAS documentation complemented with RESTalk diagrams. The experiment did not show any statistically significant difference in the results between the two treatments regarding the correctness/completeness of the solutions or the completion time. Further experiments are needed to verify whether such results hold true also for experienced API developers. The threats to the validity of the result are discussed in Sec. 7.3.4.

1.4 Outline

This dissertation is divided in three parts comprised of eight chapters in total and two appendixes. In this section we provide a brief description of the content of each of them.

The first part of the dissertation, ***Part I - Motivation and Context***, contains essential background knowledge needed to address the above state prob-

lem statement and is comprised of two chapters. As the thesis focuses on APIs in **Chapter 2 - REST APIs** we discuss the characteristics of service oriented architectures before diving into the principles and constraints of the REST architectural style. Such knowledge of REST is needed to be able to define one of the core concepts of this thesis, the concept of "RESTful conversations" and to look at and describe the existing REST APIs description languages, such as OAS, whose structure we discuss in detail in this chapter. In **Chapter 3 - State of the Art** we provide an overview of the state of the art research in different areas relevant for the topics addressed in this dissertation. Clearly we provide an overview of existing practices for modelling REST APIs from different perspectives, such as the structure, behavior or quality of service viewpoints, but also in emerging architectural styles using REST APIs such as the microservice architecture. Needless to say, once we have identified the need of and have taken the decision to design a DSL we had to look at best practices and research in the field of DSL design, but also design of modelling tools which we also describe in this chapter. Last but not least, designing a DSL requires an evaluation of the same thus we look at the state of art in the evaluation of DSLs and DSL tooling.

The second part of the dissertation, ***Part II - RESTalk***, focuses on the main contribution of this research thesis, i.e., the design of RESTalk and the tooling around it. In **Chapter 4 - RESTalk Language** we discuss the requirements we have identified for a DSL for modeling the REST API's behaviour, including the scope and purpose of the language, the targeted users, and their use cases. Then we continue with defining the abstract syntax and semantics of RESTalk in the form of a concept dictionary, a metamodel, and OCL constraints to the metamodel. As we have decided to provide both a graphical and a textual concrete representation of RESTalk we describe both of them in this chapter, discussing how we started from the BPMN Choreography diagrams and obtained RESTalk. We also present the assumptions we use to simplify RESTalk models, as well as the EBNF specification of the textual DSL. In **Chapter 5 - RESTalk Tooling** we discuss the ecosystem we have envisioned to be built around RESTalk to support its adoption and effective use in industry. We also present the proof of concept tooling we have managed to implement to support the design of RESTalk diagrams, but also the use of RESTalk for visualizing the results of mining the logs of interactions between clients and a giver RESTful service provider.

The third part of this dissertation, ***Part III - RESTalk Evaluation***, focuses on our evaluation efforts. In **Chapter 6 - RESTalk Formative Evaluation** we discuss the evaluation work which helped shape RESTalk to its current version through the gathered feedback. We present the design and the results of the exploratory survey which helped us evaluate the industry need for a DSL and provided us

with initial feedback on the design of the core visual RESTalk. Such design was evolved later on by using RESTalk to model different use cases, from short patterns of RESTful interactions to real APIs, which has improved its expressiveness as discussed in this chapter. In **Chapter 7 - RESTalk Summative Evaluation** we present the work we have done to evaluate the design of the graphical representation of the language using the Physics of Notation as a point of reference, but also by comparing RESTalk to non-domain specific languages such as UML Sequence diagrams and BPMN Choreography diagrams. This chapter also includes the description of the design and the results of the controlled experiments we have conducted with bachelor students.

In **Chapter 8 - Conclusions** we summarise our research thesis and answer the research questions we have set up in this introductory chapter. As no research work is ever perfect, and there are always ideas on how it can be improved, we conclude by discussing future work on the topic. To ensure transparency and replicability of our empirical evaluation work in **Appendix A - Exploratory Survey Questions** we provide an exported file of the actual survey discussed in Chapter 6, while in **Appendix B - Controlled Experiment Tasks and Survey** we provide the material which we have used for the controlled experiment discussed in Chapter 7.

1.5 Publications Overview

Parts of this dissertation are derived from publications in international peer-reviewed conferences and journals. In Tab. 1.1 we list the publications closely related to this dissertation, while in Tab. 1.2 and Tab. 1.3 we list work which has been published during the PhD as part of other projects.

Table 1.1. Summary of publications related to RESTalk

	Publications
2015	Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. ***"Modeling RESTful Conversations with Extended BPMN Choreography Diagrams."*** In Proceedings of the European Conference on Software Architecture **(ECSA)**, pp. 87-94. Springer, Cham, 2015.
2016	Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. ***"A Pattern Language for RESTful Conversations."*** In Proceedings of the European Conference on Pattern Languages of Programs **(EuroPLoP)**, pp. 1-22. ACM, 2016. Ana Ivanchikj, Cesare Pautasso, and Silvia Schreier. ***"Visual Modeling of RESTful Conversations with RESTalk"***. Software & Systems Modeling **(SOSYM Journal)** 17, no. 3 (2018): 1031-1051 and Invited SOSYM First Paper at the International Conference on Model Driven Engineering Languages and Systems **(MODELS)**. Invited SOSYM First Paper. ACM, 2016. Ana Ivanchikj. ***"RESTful Conversation with RESTalk -the Use Case of Doodle-."*** In Proceedings of the International Conference on Web Engineering **(ICWE)**, pp. 583-587. Springer, Cham, 2016.
2018	Ana Ivanchikj, and Cesare Pautasso. ***"Modeling REST API Behaviour with Text, Graphics or Both?***" Domain Specific Languages Design and Implementation **(DSLDI - Talk)**. 2018 Antonio Gamez-Diaz, Pablo Fernandez, Cesare Pautasso, Ana Ivanchikj, and Antonio Ruiz-Cortes. ***"ELeCTRA: Induced Usage Limitations Calculation in RESTful APIs."*** In Proceedings of the International Conference on Service-Oriented Computing **(ICSOC - Demo)**, pp. 435-438. Springer, Cham, 2018. Ana Ivanchikj, Ilija Gjorgjiev, and Cesare Pautasso. ***"RESTalk Miner: Mining RESTful Conversations, Pattern Discovery and Matching."*** In Proceedings of the International Conference on Service-Oriented Computing **(ICSOC - Demo)**, pp. 470-475. Springer, Cham, 2018.
2020	Ana Ivanchikj, and Cesare Pautasso. ***"Modeling Microservice Conversations with RESTalk."*** In Microservices, Science and Engineering, pp. 129-146. **Book Chapter**. Springer, Cham, 2020.

Table 1.2. Summary of publications related to the Bechflow project

	Publications
2015	Ana Ivanchikj, Vincenzo Ferme, and Cesare Pautasso. ***"BPMeter: Web Service and Application for Static Analysis of BPMN 2.0 Collections."*** In Proceedings of the International conference on Business Process Management **(BPM - Demos)**, pp. 30-34. Springer, Cham, 2015. Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. ***"A Framework for Benchmarking BPMN 2.0 Workflow Management Systems."*** In Proceedings of the International conference on Business Process Management **(BPM)**, pp. 251-259. Springer, Cham, 2015.
2016	Vincenzo Ferme, Ana Ivanchikj, Cesare Pautasso, Marigianna Skouradaki, and Frank Leymann. ***"A Container-centric Methodology for Benchmarking Workflow Management Systems."*** In Proceedings of the International Conference on CLOud Computing and SERvices Science **(CLOSER)**, pp. 74-84. 2016. Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. ***"Estimating the Cost for Executing Business Processes in the Cloud."*** In Proceedings of the International Conference on Business Process Management **(BPM)**, pp. 72-88. Springer, Cham, 2016.
2017	Ana Ivanchikj, Vincenzo Ferme, and Cesare Pautasso. ***"On the Performance Overhead of BPMN Modeling Practices."*** In Proceedings of the International Conference on Business Process Management **(BPM)**, pp. 216-232. Springer, Cham, 2017. Daniel Lübke, Ana Ivanchikj, and Cesare Pautasso. ***"A Template for Categorizing Business Processes in Empirical Research."*** In Proceedings of the International conference on Business Process Management **(BPM)**, pp. 36-52. Springer, Cham, 2017. Vincenzo Ferme, Marigianna Skouradaki, Ana Ivanchikj, Cesare Pautasso, and Frank Leymann. ***"Performance Comparison Between BPMN 2.0 Workflow Management Systems Versions."*** In Proceedings of the Workshop in Enterprise, Business-Process and Information Systems Modeling **(BPMDS)**, pp. 103-118. Springer, Cham, 2017.
2019	Vincenzo Ferme, Ana Ivanchikj, Cesare Pautasso, Marigianna Skouradaki, and Frank Leymann. ***"IT-Centric Process Automation: Study About the Performance of BPMN 2.0 Engines."*** In Empirical Studies on the Development of Executable Business Processes, pp. 167-197. **Book Chapter**. Springer, Cham, 2019.

Table 1.3. Summary of publications related to the BPMN Sketch Miner

	Publications
2019	Ana Ivanchikj, and Cesare Pautasso. ***"Sketching Process Models by Mining Participant Stories"*** In Proceedings of the International Conference on Business Process Management **(BPM)**, pp. 3-19. Springer, Cham, 2019.
2020	Ana Ivanchikj, Souhaila Serbout, and Cesare Pautasso. ***"From Text to Visual BPMN Process Models: Design and Evaluation."*** In Proceedings of the International Conference on Model Driven Engineering Languages and Systems **(MODELS)**, pp. 229-239. ACM, 2020. Invited paper for **SOSYM Journal** 2021.

During this PhD we have also supervised the bachelor/master thesis listed in Tab. 1.4 which directly or indirectly contributed to this work.

Table 1.4. Summary of supervised bachelor and master thesis

	Thesis
Bachelor	Ilija Gjorgjiev, *"RESTful Conversation Mining"*
Master	Ruben Folini, *"Automating Semantics-preserving BPMN Model Transformations"* Redona Kembora, *"APISYMPHONY: a Tool to Measure and Visualize Static Metrics of RESTful APIs"* Neha Tharani, *"Financial Literacy: Using Information System to Visualize Retirement Preparedness"* Gianmarco Palazzi, *"Towards Open Execution: Transparent and Tamper-proof Processes for Blockchain Applications"* Jonas Looser, *"Robotic Process Automation: a Survey"*

Chapter 2

REST APIs

In this chapter we provide relevant context for understanding and defining RESTful conversations, the principal entity in this research thesis. In Sec. 2.1 we provide a brief history of APIs, Service Oriented Architecture and communication protocols, while in Sec. 2.2 we focus on REST as architectural style and the constraints and architectural entities that characterise it. This enables us to define the concept of RESTful conversations and their properties in Sec. 2.3. In Sec. 2.4 we discussing the existing languages for describing REST APIs with special focus on the Open API Specification as the emerging standard in the field.

2.1 APIs, Web APIs and Service oriented architecture

An Application Programming Interface (API) is a set of functions or procedures used by applications to access services from the operating system, software libraries or other systems. APIs support some of the basic design principles of object oriented programming, such as abstraction, encapsulation and modularity. Namely, through information hiding APIs allow for encapsulation and modularity of software, thus promoting software reuse and reducing the software development effort. An API is essentially a contract that defines how two systems should communicate with each other. It defines both how calls to the API should be made, but also what is the expected behaviour of the API when the call is made. APIs have evolved significantly overtime. API as a term has been first mentioned in 1968 by Cotton [30] to refer to clearly defined interaction methods that insure the independence of the application being developed from the underlying hardware. Later on, in 1975 the term was also introduced in the field of databases [38], and soon after the term started being applied for all types of programming, not just application programming. The concept of an API gained

momentum and was extended again with the emergence of the World Wide Web when the first Web APIs appeared which allowed for exchange of data between different systems through a communication protocol over the network [218]. Although today the term API is most widely used to refer to a Web API, the term API has traditionally been used to specify the expected behaviour of a software library or to specify the interface between an application and the operating system. In both these cases the exchange of data is local, as opposed to Web APIs where the exchange of data is between two machines over the network. Ofoeda et al. [155] provide a recent literature review on API research in the last decade which includes work on different types of APIs and from different aspects, both in technical and social dimensions.

With the appearance of Web APIs, any discrete unit of functionality that can be accessed remotely became a service and a new architectural style called Service-Oriented Architecture (SOA) gained on popularity. “SOA is a software architecture that is based on the key concepts of an application frontend, service, service repository, and service bus. A service consists of a contract, one or more interfaces, and an implementation" [113].

The **application frontend** calls the service and receives the results of the service call. It is also known as service consumer or client. It can be a web application that interacts with the end users, or it can also be a batch program that invokes the service periodically. The Open Group consortium[1] defines the following characteristics of a **service**:

- a logical representation of a repeatable business activity with a specified outcome;
- self-contained;
- can be composed of multiple services;
- a black box to consumers of the service.

The service contract specifies the purpose and functionality of the service as well as the terms of use. It can be formally defined in a standard definition language such as IDL or WSDL. The functionality of the service is exposed through one or more interfaces-APIs. APIs can use different communication protocols which define how the services pass data among each other. The implementation of the service provides the business logic and the data, and its complexity is hidden by the API. Each service should be separately maintained and deployed, and loosely coupled with respect to other services. The **service repository** enables service discovery through the service meta data and serves as a database of different published services, while the **service bus** connects the application frontend with the

[1] `https://www.opengroup.org`

service. A service can provide a completely new functionality, or it can be used only as a wrapper around the functionalities of a legacy system to make them available remotely. SOA brings several promises such as reuse, agility and flexibility, application and data integration, as well as efficient time to market with reduced costs [181]. Whether these theoretical promises get delivered does not depend only on the technology, but also on the business vision of the company, the implementation methodology and the correct service abstraction.
As mentioned above, different protocols can be used to enable the communication between services. In 2000 Microsoft and IBM published the spec for the Simple Object Access Protocol (SOAP) as standardized communication protocol between servers which encodes messages in XML and transfers them over a common envelope [193]. In the same year Salesforce offered its first XML based Web API [182]. However, only few developer teams could take advantage of Salesforce XML API, which due to its complexity came with a more than 400 pages long user manual [2]. Meanwhile Fielding was working on a new architectural style for web applications which he called REpresentational State Transfer (REST), described in detail in his PhD dissertation [54] published also in 2000. By the end of the year EBay published its first API following the REST principles. Initially SOAP, supported by some of the largest organizations, gained a lot of popularity among developers, but REST caught up fast. While SOAP continues being used for enterprise-level web-services, REST is the dominant choice for building public APIs [141], as it requires lower network bandwidth and lower round-trip latency [143]. Tab. 2.1 shows the main differences between SOAP and REST, while in [162] Pautasso et al. compare the two integration styles in terms of architectural decisions. In this work we focus on the REST architectural style.

Table 2.1. SOAP vs REST Characteristics (adopted from [198; 218; 221])

SOAP	**REST**
Exposes operations/method calls	Returns data without exposing methods
Supports WS-Security	Security handled by underlying structure
Requires a SOAP library at the end of the client	No library support needed, typically used over HTTP
Not strongly supported by all the languages	Single resource for multiple actions
XML format required	Supports any content-type (XML and JSON used primarily)
Heavy message payload and high bandwidth	Lightweight messages and lower bandwidth
All calls send through POST	Typically uses explicit HTTP action verbs (CRUD)
Can be stateless or statefull	Stateless
Difficult caching due to complex XML	Caching can be used for performance improvement
WSDL - Web Service Definitions	Documentation can be supplemented with hypermedia
More difficult for developers to use	Less difficult for developers to use

2.2 REST Architectural Style

"REST is not an architecture, but rather an architectural style. It is a set of constraints that, when adhered to, will induce a set of properties" [53]. REST is a hybrid architectural style [204], which combines the layered, client-server, virtual machine, and replicated repository styles, with the additional constraints described bellow [54].

2.2.1 REST Constraints

The ***client-server constraint*** is based on the principle of separation of concerns which improves the portability of different components of the system (the client can run on different devices) and allows for independent evolution of such components. The client and the server can be implemented independently, using any programming language [159]. Any change in the server should not impact the client and vice versa [198].
The client-server interaction is affected by the ***statelessness constraint*** which requires a client's request to carry all the relevant information for understanding the request, such that the server does not need to remember the state of the conversation. As a consequence, every interaction within a RESTful architecture is always initiated by the client. Namely, this constraint explains the name REST (representation state transfer) as "interaction state is not stored on the server side; it is carried (transferred) by each request from the client to the server and encoded inside the representation of the resource the request refers to" [178]. The statelessness constraint improves scalability as the server does not need to dedicate any resources for keeping the session state. However, the trade-off of this constraint is that it can introduce per-interaction performance overhead by increasing the data that needs to be sent in sequential requests.
The ***cache constraint*** requires the cacheability of the data in a response to be implicitly or explicitly labeled. When data is labeled as cacheable, later requests for the same data can be served from the client cache, thus eliminating the need to interact with the server to retrieve the data. When the data is retrieved for the first time, the response indicates the expiry date of the cached data, after which the request can no longer be served from the client cache. Cache can also be used by the server to avoid the generation of a response each time the same resource representation is requested. The main reasons for the use of caching is to reduce latency and network traffic and improve the overall availability and reliability of a service.
The ***layered system constraint*** restricts the knowledge of the system to a single

layer, i.e., each component only knows about the layer which it interacts with. This allows the use of intermediaries, such as proxies and gateways, without changing the interfaces between components. Intermediaries are frequently used for security reasons, load balancing, caching or encapsulating legacy services. Layering also allows for changing and moving layers in and out of the system architecture as technology evolves, provided that they are loosely coupled [198].
The ***code-on-demand constraint*** requires the client to download and execute code sent by the server, such as scripts or plug-ins. Since it can lead to technology coupling between the client and the server it is the only constraint which is optional in REST.
The emphasis on the ***uniform interface constraint*** is one of the central features that distinguishes REST from other architectural styles. It allows for decoupling, information hiding and standardization. This is achieved by using standard methods to manipulate resource representations and using standard response status codes. Which methods are available for which resource is decided at design time, but can change at run time based on the state of the resource. When using the HTTP protocol the most frequently used methods are GET, POST, PUT, and DELETE. Which one will be used in the request, depends on the client's goal. GET is used to retrieve the resource representation, POST is used to create new resources, PUT is used to modify existing resources and DELETE is used to delete existing resources [51]. Some of these methods (e.g., GET) are safe in terms that they do not modify the resource. Others (e.g., PUT, DELETE), are not safe, but they are nonetheless idempotent, i.e, they can be called multiple times without changing the outcome of the call, which has important recovery implications in presence of temporary communication failures [52]. The standardization of the response status code is done in three-digit number classes. Thus, the responses with status code in the range 100 to 199 indicate that the information in the response is provisional, in the range 200 to 299 indicate a successful request, in the range 300 to 399 indicate the need of redirecting the request, in the range 400 to 499 indicate an error while in the range 500 to 599 indicate server failure and that the client should resend the request. The standardization of the interface allows for sustainability as the implementation can be replaced without impacting the users. The uniform interface constraint creates the necessity of further constraints to guide the behavior of different components, i.e., the identification of resources, the manipulation of resources through representations, the self-descriptive messages, and the Hypermedia As The Engine Of Application State (HATEOAS).
These constraints ensure a "design that creates an API that is not dictated by its architecture, but by the representations that it returns, and an API that – while

architecturally stateless – relies on the representation to dictate the application state" [198]. Clients do not know the internal format and state of resources, all they receive in the server response is a representation of the resource.

2.2.2 REST Architectural Entities

The main ***entity*** in REST is the *resource*. Resources are conceptual abstractions of any information or service that can be named, and thus can be identified as relevant to the client. A resource is the semantics of what needs to be identified, not the value of the semantics at the point when the reference is created. A resource can have subresources that represent its specific subordinate concepts [126]. The entry-point of the API is its root resource. REST APIs are collections of interlinked resources that adhere to REST architecture principles and constraints. Resources provide generalization which abstracts from the data type or implementation of the resource. Each resource is globally identified by a *Uniform Resource Identifier (URI)* used to address the request to that resource. Although Fielding does not talk about URI design in his thesis, there are certain best practices that should be followed when designing the API [126]. The role of a REST API is to provide a mapping of a URI to a representation of the resources. The current or intended state of a resource is captured in its *representation* which is a sequence of bytes used in the communication between REST components. "The representation is a way to interact with the resource but is not the resource itself" [126]. This allows to send different representations of the same resource to different clients depending on the request. The data format of the representation is called a media type. Different media types can be used such as plain text, JSON, XML, etc. Resources allow for late binding of the reference to a representation by content negotiation between the client and the server, depending on the desires of the client and the nature of the resource. The response can also include a *representation and resource metadata*. The last data element Fielding mentions in his thesis is the *control data* in the request or the response used to parameterize requests, override default caching behaviour etc.
Fielding identifies two primary ***connector*** types, *client* and *server*. Connectors are defined as abstract interface for component communication where ***components*** refer to the *origin server*, *gateway*, *proxy* or *user agent*. The client instantiates the communication with the server by making a request, while the server responds to the request. Essentially the client is interested in manipulating the resources managed by the server. Client's request consists of a method, URI, request-header fields and sometimes representation, while server's response consists of a status code, response-header fields and sometimes a representation.

The client discovers the URIs dynamically from the server's response, which can refer the client to related resources. The mechanism whereby hyperlinks (or resource references) are embedded into resource representations, or sent along in the corresponding meta data [154], is one of the core tenets of REST, known as Hypermedia [12]. The server's responses can contain from zero to many links, depending on the current state of the requested resource. The server might also send parametrized links based on which the client can dynamically construct the URI for the next request by providing the required parameter(s). The client should simply follow links, without making any assumptions about the URI's structure [196]. This constraint is known as HATEOAS (Hypermedia As The Engine of Application State) [54] and allows the server-side logic to be changed and to evolve independently of the client [120; 159].

2.3 RESTful Conversations

Web services borrowed the notion of conversation [18] to indicate richer forms of interactions going beyond simple message exchange patterns [220]. In traditional messaging systems, conversations involve a set of related messages exchanged by two or more parties [85; 17; 83]. As more and more Web services adopt the constraints of REST, conversations remain an important concept when reasoning about how clients make use of REST Web APIs [173] which by design are chatty [96]. Although the simplicity and standardization of the HTTP protocol allows for very trivial conversations which can be limited to single interactions, by studying and implementing RESTful Web services, we have noticed that for addressing some non-functional requirements (e.g., security, reliability, scalability), developers often combine several HTTP interactions, as part of conversations. Redirection is a very simple example of such a conversation, implemented in many development frameworks and libraries (e.g., Express.js, Play). Frequently, achieving a goal with a REST API requires multiple HTTP request-response interactions. Different clients might have different goals achievable with different HTTP request-response sequences. Thus, in the REST domain as well we can talk about conversations. We call a ***RESTful conversation*** *a model of a set of possible sequences of interactions that one or more types of clients can have with a given REST API*. Different run-time instances of a given RESTful conversation can take different paths in the conversation model as different clients might have different goals to achieve, or may take different paths to reach the same goal. RESTful conversations have been introduced in [81], where they are used as an abstraction mechanism to simplify the modeling of individual REST

APIs. The goal of these conversations is usually to retrieve or modify the state of one or more resources which are managed by the service provider.
To summarize, as a result of the characteristics and constraints of the REST architectural style mentioned in Sec. 2.2, RESTful conversations can be seen as a specific kind of message-based conversations defined by the following properties:

1. Interactions are always client-initiated, thus it is the client who drives the conversation forward and decides when to stop it;

2. Client requests are addressed to resources, identified through their URIs;

3. When the server is available to process client requests, every request message is always followed by a response. There may be different possible responses to the same request message, depending on the state of the requested resource;

4. Hypermedia: responses embed related URIs, which may be used to address subsequent requests;

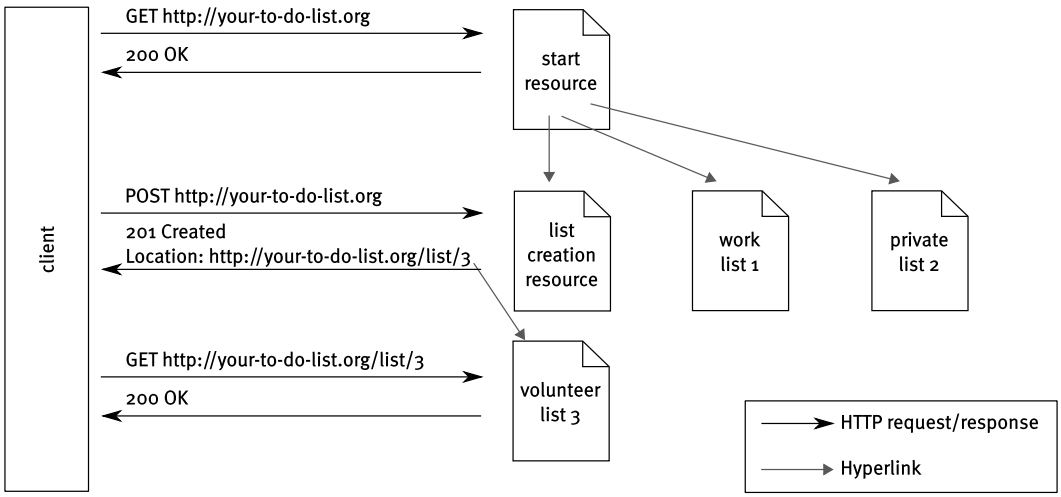
5. Statelessness: every request is self-contained and thus independent of the previous ones;

6. Uniform Interface: there is a fixed set of request methods a resource can support. Depending on the state of the resource, the server allows clients to use different methods when interacting with it.

These properties make it possible to share the responsibility for the conversation's direction between clients and servers. It is the client who initiates the conversation, but it is the server who guides the client towards the next possible steps by choosing to embed zero, one or more related URIs as hyperlinks in a response. The client may choose which hyperlink(s) to follow, if any (it may also decide to stop sending requests at any time). Thus, it is the client who decides the path to take to continue the conversation by selecting the next request from the options provided by the server in previous responses. As the client is following links, the REST API can be seen as a navigation graph-like structure of resources connected by hyperlinks [82]. In general, the client can accumulate URIs discovered during the entire conversation or may remember them from previous conversations. Zuzak et al. call this the *Link Storage* in their finite-state machine model for RESTful clients [238]. Additionally, responses may be marked as cacheable, and thus clients will not need to contact the server again when reissuing the same request multiple times.

The discussion so far assumes that servers are available and always reply to client's requests. However, servers may indicate their unavailability by sending responses with the 503 Service Unavailable status code. In case of failures, either due to loss of messages or due to the complete unavailability of the servers, an exception to the request-response rule must be made. Clients may thus decide to resend a request after a given timeout (for temporary failures) or eventually give up retrying (for permanent failures).
To give an example of a RESTful conversation, lets imagine a REST API for managing simple to-do lists allowing the client to create new lists and add or remove items to these lists. The RESTful conversation typically starts with the client accessing the root resource of the service, i.e., sending a GET request at `http://your-to-do-list.org`, for instance. In the response the client can find links to its existing lists, like `/list/1` which refers to its professional to-do list, or `/list/2` which refers to its personal to-do list, as well as a link to a resource, e.g., `/list`, for creating a new list. The client can now decide to access one of the existing resources using a GET request, or to create a new resource (list). In the latter case the client would send a POST request to `/list` with the name of the new list, e.g., "volunteer", as content in the request body. Since POST is not an idempotent verb, resending the POST request multiple times would result with the creation of multiple new lists with the same name if the design of the API is not robust enough to detect and handle such cases. Once the server creates the list, it would send a "201 Created" response with a Location link header referring to `/list/3`, the URI of the newly created list. If the client wants to access the new list it can easily follow this link using a GET request which would respond in a "200 OK" with the content of the new to-do list. However, this does not need to happen immediately. The client could also pause the conversation for a longer period of time and access the new volunteer list later, as the `/list/3` location link contains all the necessary information to identify this specific resource. Such property of the URI provides for the statelessness principle. Namely, there is no need for the server to store something like a session, i.e., the state of this client-server-communication, since all required information is contained in the request sent by the client. As all URIs, beside the first one, are provided in responses to previous requests, the client can discover the whole service by solely knowing the root URI of the service.
A visualization of an optional conversation of the client and the server is illustrated in Fig. 2.1 where the request and response bodies are left out for readability reasons. However, Fig. 2.1 only shows one possible direction of the conversation. The client can also decide to access its personal or professional list which would result with a different set of interactions. Visualizing all of the possible

Figure 2.1. An example of a RESTful conversation



conversation directions can be facilitated with the domain specific notation we present in Chapter 4.

2.4 REST APIs Description Languages

Good documentation of an API is not only crucial for attracting customers to build clients for the API, but is also very important for the maintenance and evolution of the same. In the API design first approach good documentation is also crucial for discussing the design of the API [71].
Initially the development of REST APIs was code-driven, and due to the simplicity of REST, documentation was considered redundant [121]. The first documentation attempt came to life in 2009 when the Web API Description Language (WADL) was designed by Sun Microsystems using XML. Soon after, the open source Swagger project, with no formal corporate backing, was started[2]. It used and still uses JSON format for the API documentation. In 2013 two more REST API description languages appeared, the API Blueprint[3] and the REST API Modelling Language (RAML)[4], both of them with corporate backing, API Blueprint by Apiary and RAML by MuleSoft. Swagger, API Blueprint and RAML started competing and co-existing, each with a different vision. Swagger started with a focus on the code-driven development needs and with vendor neutrality. API

[2] http://swagger.io
[3] https://apiblueprint.org/
[4] https://raml.org/

Blueprint focused on the API consumers and since the human readability of the documentation was important it opted for markdown formatting of the documentation. RAML on the other hand opted for the YAML format with a focus on the API design. It targets larger API providers.
At the end of 2015 the OpenAPI initiative[5] has been launched to promote the effort of standardizing the description of the interface of REST APIs. The Swagger 2.0 specification has become the basis for the proposed standard OpenAPI Specification (OAS). OAS is a language-agnostic, vendor-neutral, portable and open specification for a YAML/JSON-based description format for REST APIs, striving to become a widely accepted standard. The specification is both human and machine readable and allows its users to discover and understand the structure of REST APIs without accessing their source code. Recently, in addition to major companies such as Google, IBM and Microsoft, the creators of API Blueprint and RAML also joined the OpenAPI initiative, thus opening up the possibility for converging of the three description languages into one standard language.
The OpenAPI specification document is created with the support of dedicated DSLs or GUI editors[6], for the time being it can still not be created from an API implementation code. Although an OpenAPI document that conforms to the OpenAPI Specification can be represented in JSON or YAML format, it is actually a set of JSON objects, with a specific schema to define the naming, order, and contents of the objects. As evident in Tab. 2.2 an OpenAPI document can have eight root level objects.

Table 2.2. Root JSON objects in an OpenAPI documentation

Object	*Description*
openapi	semantic version number of the OpenAPI Specification
info	metadata about the API such as the title, a short description, the version, link to the license and the terms of service, and contact information
servers	array of server objects which provides the basepath to a target server, i.e., the part of the URL that appears before the endpoint
paths	endpoints provided by the API and operations on the same
components	an object that holds various schemas for the specification, i.e., re-usable definitions that appear in multiple places in the OpenAPI document
security	defines the security mechanisms that can be used in the API
tags	a list of tags to define additional metadata and arrange the endpoints into named groups
externalDocs	additional external documentation

[5] `https://www.openapis.org`
[6] `https://openapi.tools`

The authors in [48] derive a metamodel of the OpenAPI Specification, describing three main parts: 1) behavioural elements, 2) structural elements and 3) serialization/deserialization elements. The behavioural elements of the methamodel presented in Fig. 2.2 are of main interest for this thesis.

Figure 2.2. Behavioral elements of the OpenAPI metamodel

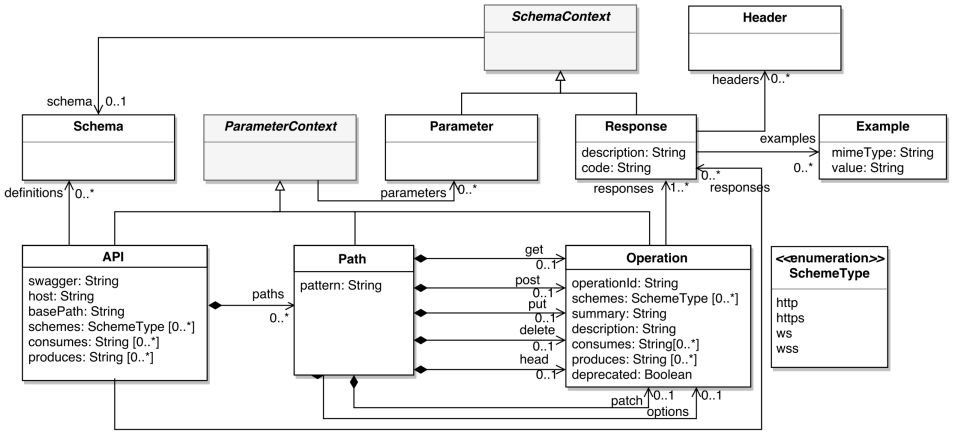


*Image taken from [48]

The `paths` object is comprised of path items sub-objects. In the example in Listing 2.1, taken from the OpenAPI Github repository examples[7], there are two paths `/2.0/users/username` and `/2.0/repositories/username`. The call to the resources on this path is made on the URL composed of the server URL defined in the server object plus the URL path stated in the path item object. For each endpoint, or path in OAS terminology, there is a list of `operations` objects defining the methods that can be called on the endpoint. The operations object has multiple properties that define both the request and the response to the resource in question. The first endpoint in the example in Listing 2.1 only supports the GET method. The path to the resource is parametric as evident from the `parameters` object where the `username` parameter is defined as a required parameter in the path. The `in` property of the parameter object defines where the parameter appears and accepts the following values: `header`, `path`, `query` and `cookie`. Request body parameters are described in the request body object. The current version of the OpenAPI specification cannot programmatically reflect the dependencies between parameters or the mutual exclusiveness of the parame-

[7] `https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/link-example.yaml`

ters. It can only be described in natural language in the `description` property of the parameters object.
Another important object inside the operations object is the `responses` object which defines the responses that can be obtained when making a call to the endpoint of interest. In addition to the status code, 200 in the example in Listing 2.1, the responses object can also include a reference object with a pointer to a description in the components object. The goal of using reference objects in the OpenAPI documentation is to simplify the code and to support the reuse. The components object can contain `schemas`, `responses`, `parameters`, `examples`, `requestBody`, `headers`, `securitySchemes`, `links` or `callbacks`. In addition to the official OAS documentation at github[8], there are also different tutorials available which explain the specification in detail [98].

Listing 2.1. YAML representation of an example of OpenAPI document

```yaml
openapi: 3.0.0
info:
  title: Link Example
  version: 1.0.0
paths:
  /2.0/users/{username}:
    get:
      operationId: getUserByName
      parameters:
      - name: username
        in: path
        required: true
        schema:
          type: string
      responses:
        '200':
          description: The User
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/user'
          links:
            userRepositories:
```

[8] https://github.com/OAI/OpenAPI-Specification

```
              $ref: '#/components/links/UserRepositories'
  /2.0/repositories/{username}:
    get:
      operationId: getRepositoriesByOwner
      parameters:
        - name: username
          in: path
          required: true
          schema:
            type: string
      responses:
        '200':
          description: repositories owned by the supplied user
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/repository'
          links:
            userRepository:
              $ref: '#/components/links/UserRepository'
components:
  links:
    UserRepositories:
      # returns array of '#/components/schemas/repository'
      operationId: getRepositoriesByOwner
      parameters:
        username: $response.body#/username
    UserRepository:
      # returns '#/components/schemas/repository'
      operationId: getRepository
      parameters:
        username: $response.body#/owner/username
        slug: $response.body#/slug
  schemas:
    user:
      type: object
      properties:
```

```
        username:
          type: string
        uuid:
          type: string
    repository:
      type: object
      properties:
        slug:
          type: string
        owner:
          $ref: '#/components/schemas/user'
```

The major revision of the standard in version 3.0 released in July 2017, and updated to the latest release 3.1.0 in June 2020, has made structural changes to the OpenAPI document to accommodate multiple servers, to emphasize reusability through the definition of the above mentioned `components` object and to encourage the use of examples.
In this major release also the above mentioned `links` component was added. This new component serves to describe how various parameter values returned by one operation can be used as input for other operations. Thus, it provides details about the relationship between the operations and a mechanism to traverse the operations. More detailed explanation on the use of the `links` object is available at `https://swagger.io/docs/specification/links/` and an example use is provided in Fig. 2.3. Links can also be used to refer to external documentation of different APIs.
However, the OpenAPI Specification does have some shortcomings. It does not provide any information about API keys, rate limits etc [64]. Furthermore, while the OpenAPI documentation enumerates all possible operations (combinations of resource path and method) provided to API clients, it lacks a description of which are the meaningful sequences of API interactions that can be followed by clients to achieve their goals. Such behavioral aspects, i.e., the dynamic interactions between the client and the API, are not always evident in large and complex systems [98] where frequently the interactions require to call a certain endpoint in order to get an object that is required in the parameters of another endpoint. OpenAPI documents using OAS 3.0 potentially can contain behavioral information whose format is machine readable, but not friendly for the human user who needs to traverse the links to get a general image of the API behaviour as evident in Fig.2.3, as no visualization of the relationships is available as part of the standard. A dataset of 1176 REST APIs that we analyzed as part of a master

Figure 2.3. The use of links in OAS 3.0 to describe the API behavior

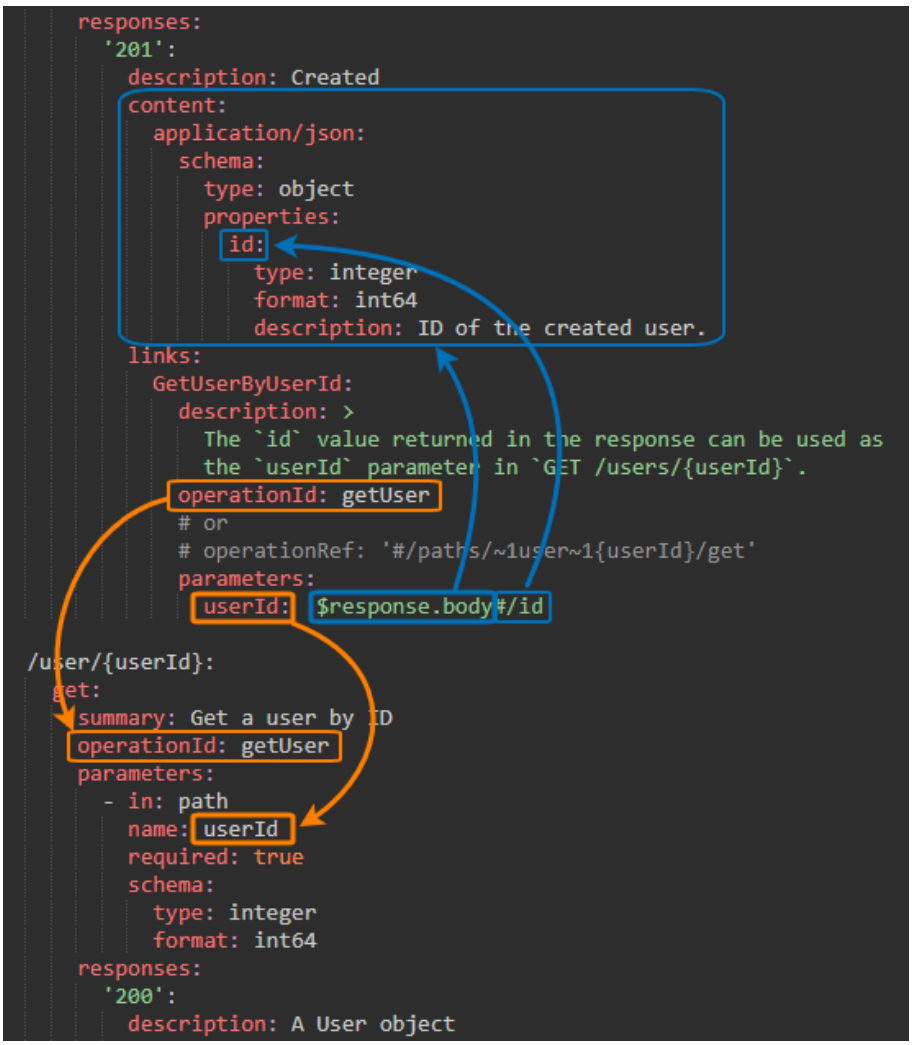


*Image taken from `https://swagger.io/docs/specification/links/`

thesis we supervised [106], has shown that none of the APIs in the API Harmony repository[9] actually uses OAS 3.0.

2.5 Chapter Summary

We started this chapter with a short discussion of the emergence of APIs and their popularisation with the SOA architecture and the World Wide Web, to then

[9] `https://apiharmony-open.mybluemix.net/public`

focus on REST as an architecture style. Studying the characteristics of REST was indispensable for enabling us to define the central concept of this thesis, the concept of RESTful Conversations whose properties are shaped by the the REST constraints. RESTful conversations are models of a set of possible sequences of interactions that one or more types of clients can have with a given REST API. As initially the documentation of REST APIs was considered redundant due to the simplicity of REST, existing languages for describing REST APIs, such as the emerging standard OpenAPI Specification (OAS), only focus on describing the structure of the API. It is only in the latest version of OAS that behavioral information about possible interactions between operations earned its place in the specification through the *link* object which, if used correctly, can provide machine readable traversal mechanism for the REST API.

Chapter 3

State of the Art

When faced with a problem, before putting efforts into inventing a new solution, one investigates existing solutions to see whether they can solve the problem effectively. Thus, in this Chapter we discuss the related work and position the contribution of this thesis with respect to the body of current research. In Sec. 3.1 we discuss existing solutions in different areas for modeling REST APIs, not only their behaviour, but also the structure and the quality of service of the same. After analysing existing modelling solutions, depending on how well they respond to the identified problem, one can decide to expand them, or to find a new modelling solution starting from scratch. In both cases, the solution needs to be designed. Thus, in Sec. 3.2 we discuss existing guidelines for DSL design, while in Sec. 3.3 we discuss existing approaches for designing editing tools for DSLs and based on our research we present a maturity model for editors supporting visual models. Once the solution has been designed in also needs to be evaluated which is why in Sec. 3.4 we reflect on existing work in software evaluation and DSL evaluation.

3.1 Modeling REST APIs

REST APIs can be modelled from different perspectives. In this section we will look at the existing work in some of the most relevant perspectives.

3.1.1 Modelling REST APIs Structure and Behaviour

The necessity of modeling Web service interactions is as old as Web services themselves. It has led to the creation of the "Web Services Choreography Working

Group"[1] in 2002 which aimed at defining a vendor-neutral choreography specification to facilitate service integration. Muehlen et al. [236] illustrates the history of the standardization process in this area, but the Web Services Choreography Description Language (WS-CDL) remained only a candidate standard language since the working group was closed in 2009. However, graphical representation of the choreographies was never in the scope of this group. Meanwhile, academia and industry were working on the visualization aspect of choreographies and came up with different proposals for modeling languages such as Let's Dance [229] or iBPMN [41]. They have eventually led to the introduction of the Choreography Diagram in version 2.0 of the BPMN standard [223, Chap. 5], which Nikaj et al. [151; 152] use to model RESTful conversations by adding REST-specific annotations. They focus on maintaining the business logic behind RESTful interactions, evident also in their further work in [153], where they derive RESTful choreographies from BPMN choreographies based on natural language processing of the activities' labels, and in [150] where they propose an orchestration tool called ChoreoGuide that can be deployed in a REST business process engine which serves the purpose of validating the REST request payload whenever it requires the change of the state of a resource. This makes their visual modeling approach verbose in cases where business process orchestration is not the main focus. While they target mainly RESTful interactions between different business processes, we abstract from the business process context and concentrate on the RESTful interactions in general, may they be in a business process context, microservice context etc. More details on their approach can be found in Nikaj's PhD thesis [149].

Another stream of studies focuses purely on REST APIs. Schreier [187], in her REST metamodel, identifies structural (static) and behavioral (dynamic) modeling of REST APIs. The static model defines the structural elements of an API, i.e., the resources, their URI, their methods, the supported representation media types etc. Significant theoretical work [54; 126; 80] and tool support (e.g., RAML, Swagger, Blueprint, Mashape, OAS) has already been provided for this structural aspect. The behavior model, on the other hand, refers to the request-response interactions and the behavior triggered by method calls. Existing work on the behavioral aspect usually addresses the question of validating compliance with REST. As such it tends to rely on Petri Nets [117] or UML state machines [187; 165; 169; 170] to visualize the dynamics.

Li and Chou [117] use Coloured Petri Nets [97], what they call REST Charts, to model REST APIs as a set of hypermedia representations and transitions between

[1] `http://www.w3.org/2005/12/wscwg-charter.html`

them. Later on in [118], they extend their modeling framework in order to decouple the resource representation from the resource connections and to provide for layered representations. The interaction itself is depicted in the transition element, however it lacks REST specific visual presentation, i.e., the request and response are not evident in the transition element. Alowisheq et al. [10] use UML collaboration diagrams to show the interactions between different REST resources, but they fail to depict alternative paths, nor do they visualize responses or hyperlink flows.
Rauf et al., tackling the research question of designing REST compliant and dependable Web services in a PhD thesis [170], identify the necessity of modeling the behavior of REST interfaces. They use UML class diagrams and UML protocol state machines to visualize the behavioral interfaces in order to take advantage of existing tools for model validation and consistency analysis. Their model is state centered and focuses on any data sent with POST, PUT or DELETE requests in order to trigger the transfer between states. The GET method is implicitly used to check for the state invariants. Based on Schreier's metamodel [187], van Porten in [215] develops and evaluates a visual notation for modeling Resource-oriented applications. Beside views for static aspects, he also offers two views for dealing with the behavioural aspects of the model. One focuses on the behavior of single method calls, thus not tackling multiple client-server interactions. The other view focuses on the state transitions of a single given resource.
Alarcon and Wilde [6] embrace the important REST principle of interlinked resources and propose ReLL (the Resource Linking Language). They agree that the most important aspects in modeling REST services are the links between resources and the necessary interactions to access the resources. The tool they have implemented to harvest REST resources outputs a typed graph of the discovered resources and the links between them, but it does not explicitly present the request method, response status codes and the control flow.
Mitra [137], proposes a sketching tool aimed at Web API designers, which offers two different canvas views depending on the API style, which he calls CRUD and Hypermedia style. The CRUD style reflects the resource, URI and HTTP method, while the Hypermedia style uses informal state diagram for visualization. This tool, by design, does not support logical behavior visualization. In Tab. 3.1 we provide a summary of the mentioned visual approaches for modelling the behavior of REST APIs.
Since the convergence towards Open API Specification as a standard documentation for REST APIs, work had been done in MDE to develop tools that make use of the specification. WAPIml [47] enables developers to create an OAS document, or import it, generate a UML class diagram based on the same in Eclipse which

Table 3.1. Existing approaches for modelling the behaviour of REST APIs

Language	Year	Reference	Identified shortfalls
ReLL	2010	[6]	The content of the request-response as well as the control flow are not explicit.
UML State Machines	2011	[165; 169; 170; 187]	Not domain specific, state based and thus use of the GET method is not explicit.
UML Collaboration Diagrams	2011	[10]	Not domain specific, alternative paths, responses and hyperlink flow is not visualized.
Visual REST	2012	[215]	Does not include multiple interactions or only focuses on a single resource.
Petri Nets	2013	[97; 117; 118]	Not domain specific, the request-response behaviour and hyperlink flow are not visualized.
Rapido	2015	[137]	Does not support logical behavior visualization by design.
BPMN Choreography Diagram	2015	[149]	Focus on maintaining the business logic behind RESTful interactions, no hyperlink flow visualization.

they can edit and then generate an updated OAS document based on the same. In [46] the authors present the OpenAPI Bot, a chatbot which in addition to providing users information about API's metadata, operations, and data structures, which are already available in the API specification itself, also provides insights which are not as easy to derive directly from the specification (e.g., which operations return instances of a given resource).

Recently, some commercial tools, such as Visual Paradigm[2], have recognized the importance of graphical design of REST APIs and provide support for using extended UML class diagrams to that end. While such representation helps identifying the shared data among resources, there is no possibility of modeling the interaction flow. Ballerina[3] on the other hand is an open-source project for building a programming language for integration. It uses both textual and graphical syntax to implement microservices with distributed transactions. It relies on UML Sequence Diagrams for the graphical visualization of independent parties' interactions via the ballerina composer, which supports both code editing and graph editing. Although Ballerina supports the development and integration of RESTful services, it is not REST specific.

Another modeling aspect which has been rarely addressed for RESTful services, and which is closely related to their behaviour, is the modeling of the Quality of Service (QoS). Work exists on annotating WSDL descriptions of services with QoS information [35], but RESTful services are not documented in WSDL. Thus, Yoo et al. [228] have proposed an integrated annotation for SOAP and REST Web services that also includes non-functional semantics. In the REST specific domain,

[2] `https://www.visual-paradigm.com/features/visual-api-designer/`

[3] `https://ballerina.io/learn/tools-ides-ballerina-composer/`

Sepulveda et al. [191] have extended the previously mentioned ReLL language to consider security constraints in RESTful services compositions. However, they do not take into consideration how the control flow of the RESTful conversation can impact the analyzed QoS.

3.1.2 Modelling RESTful Interactions in Microservice Architecture

Most of the works which mention the challenge of microservice communication and integration, focus on microservice architecture in general and only touch upon the communication challenge as evident from the literature survey conducted by Alshuqayran et al. in [11]. The authors in the same work also provide a survey of the different approaches used to model different aspects of the microservice architecture. They have discovered that the most frequently used diagrams are component diagrams to show the static inter-dependencies between microservicies. Some researchers have also used UML sequence diagrams, use case diagrams or class diagrams to depict different viewpoints of the microservice architecture. Srikanta et al. [158] use UML sequence diagrams to describe the communication flow in the microservice architecture that they propose for dynamic rating, charging and billing for cloud service providers. The microservices in their reference architecture are RESTful, however their use case is simplistic as it uses just three microservices which communicate among each-other in a sequential flow, with no control flow divergence. Toffetti et al. [206], in the context of cloud resources deployment, use a type graph to represent the needed cloud resources and the connections between them together with their cardinality, and an instance graph to show the actual deployment of the resources, visualized by square nodes and undirected edges. They propose using the same type of graphs for microservice based applications as well. De Lange et al. [40] in their Community Application Editor, built to support model driven web engineering, include the modeling of microservices as part of the server-side modeling view. They have RESTful resources as the central entity of their microservice view meta-model, together with their HTTP methods and responses. The communication dependencies between the microservices, or between a microservice and the front-end components, are drawn automatically by the tool in the communications view based on the data entered in the other views. In the communications view the microservice is visualized as a node, but the microservice call is also visualized as a node which violates the perceptual discriminability principle in the design of visual notations [142]. No control flow divergence/convergence constructs are

available, and the hyperlink flow is not visualized.
Granchelli et al. [72] use a model reverse engineering approach in their tool MicroART to recover the microservice architecture of a system which is available on Github. They use the communication logs to discover the inter-dependencies between the microservices. The automatically generated links between the microservices can be edited and refined by a knowledgeable human using the graphical editor of the tool. One refinement that they propose is to resolve the interfaces referring to what they call the Service Discovery service (e.g., an API gateway), which masks the real resource dependencies. Thus, the human should remove the API Gateway from the microservice architecture visual diagram, and reestablish the links (the calls) directly between the microservices. In their visual model, they also include information about the teams and the developers working on each of the microservices. As they group together all the resources belonging to a discovered microservice, their approach only reveals the resource URI and the microservice it belongs to, but not the method calls and the hyperlinks flow. Namely the diagram contains directed edges to show the static dependencies between the different resources, but they do not show the dynamic interaction behavior that can be followed at execution time.

3.2 Designing Domain Specific Languages

Domain Specific Languages (DSLs) are tailor-made for specific problem domain, which allows them to be very expressive in the domain that they target, but also makes them unsuitable for arbitrary domains. In addition to their expressiveness, DSLs also safeguard the integrity of models to a certain degree as their syntax and semantics are thought through to include some domain specific constraints [57]. The idea of using DSLs dates back to the 1960s [133] and has particularly gained on popularity with the raise of the Model Driven Development (MDD) paradigm [190] where models are frequently used for code generation. As per Fowler [56] defining a DSL requires three main steps: defining the abstract syntax, defining an editor to manipulate the abstract syntax, and defining a generator, i.e., translating the abstract representation into an executable representation.
A rather popular work in the field of designing visual notations in software engineering is Moody's Physics of Notation theory [142]. He defines a set of principles for designing cognitively effective visual notations: 1) semiotic clarity, i.e., one to one correspondence between semantic constructs and graphical symbols; 2) perceptual discriminability, i.e., clear difference between symbols; 3) semantic

transparency, i.e., the appearance of the symbols should suggest their meaning; 4) complexity management, i.e., explicit mechanisms in place for dealing with complexity; 5) cognitive integration, i.e., explicit mechanisms in place integrating information between diagrams; 6) visual expressiveness, i.e., usage of the full range of visual variables; 7) dual coding, i.e., usage of text to complete graphics; 8) graphical economy, i.e., keeping a cognitively manageable number of symbols; and 9) cognitive fit, i.e., usage of different dialects for different audiences. He also emphasises the importance of documenting design decisions. The findings of a recent survey with 104 practitioners shows that the Physics of Notation theory "is complete in the sense that it covers all the important requirements practitioners may have" and as such is a good candidate for a leading approach for the design of visual notations [212]. Ulrich [57] defines generic requirements related to the pragmatics of a DSL as well as requirements for the selection of a metamodeling language. He also specifies the macro-process of designing a DSL as being comprised of 7 steps: clarification of scope and purpose, analysis of generic requirements, analysis of specific requirements, language specification, design of graphical notation, development of a modelling tool, evaluation and refinement. When creating the DSL specification Ulrich advises to create a concept dictionary where each domain concept is defined and is checked against a set of acceptance criteria in order to decide whether the concept should be added in the DSL. These criteria include relevance of the concept, the invariant semantics (the concept is abstraction over types), the variance of type semantics (semantics of the respective instances should vary). He also provides the following guidelines for designing a graphical notation which do partially overlap with the guidelines in [142]: 1) build semantic categories of concepts, 2) create generic symbols for each category, 3) the bigger the semantic difference between two concepts, the bigger the graphical difference of the corresponding symbols should be, 4) prefer icons over shape, 5) combine shape (including icons), color and text effectively, 6) avoid symbol overload, 7) avoid redundant symbols, 8) represent monotonic semantic features of a concept through compositions of symbols, and 9) a graphical notation should include symbols that allow for reducing diagram complexity.

Recently Jannaber et al. [94] have conducted a literature survey and summarised existing guidelines for designing a DSL into a framework for the design of process specific DSLs. Although they target DSLs for modelling processes, much of their framework can also be used for other DSLs. The framework consists of three iterating layers: requirements layer, language layer and evaluation layer. The requirements layer is comprised of defining the scope and purpose of the language, as well as the generic and specific language requirements. The language

layer refers to the design and modification of the DSL on one hand, and definition of process elements and perspectives on the other. As advised also in [57], the design should include the concept directory as well as the meta-model which Jannaber et al. see as determining the abstract and concrete syntax of the language as well as its semantics. They also enlist applicable methods to use when the DSL is not designed from scratch, but is rather based on an existing language, i.e., unification, specialization, selection and extension. Last but not least, the evaluation lĐřyer provides means to detect faults and gaps in the previous layers so that the iterative approach can be followed. They argue that the evaluation can be performed using both qualitative and quantitative methods as well as pattern based analysis or ontological analysis.
Karsai et al. [103] propose guidelines for designing both textual and graphical DSLs, with main focus on textual DSLs. They categorise the guidelines in 5 categories: 1) language purpose, 2) language realization or implementation, 3) language content which refers to the elements of a language, 4) concrete syntax, and 5) abstract syntax. Kelly and Pohjonen [105] on the other hand identify pitfalls in different phases of the lifecycle of a Domain Specific Modelling (DSM) language based on the analysis of 76 DSM cases.
Zdun and Strembeck [230] discuss three reusable DSL design patterns that refer to the DSL development process, the concrete syntax style and the selection between an external vs. an embedded DSL. The DSL development process pattern offers three alternative solutions: language model driven DSL development, mockup-language driven DSL development and extracting the DSL from an existing system. They identify as alternatives for the concrete syntax style the textual concrete syntax, graphical concrete syntax, form/table based concrete syntax and textual concrete syntax with generation of visualizations.

3.3 Designing Modelling Tools

Significant effort in the MDE community has been placed on the model-to-text transformation where text stands for program code, documentation, test cases or model serialization [22]. However, when adopting MDE, the effort for creating the model should not be underestimated [131]. Besides the ease of use and the intuitiveness of the modeling language, an important role in this effort is played by the model editor. The abstract syntax of a model can have a concrete representation using a textual notation, a graphical notation or a combination thereof. Thus, a model can be created using an editor supporting textual or graphical input, or both.

Researchers have been long discussing whether textual or graphical model representations are better [75; 131; 99]. Melia et al. [131] highlight the main benefits of each notation type, and perform an empirical comparison of their usability. They analyze the impact the two input modalities have on the efficiency, effectiveness and satisfaction of novice programmers while performing maintenance tasks. Their experiment has shown that the novice programmers discovered more errors and were more efficient in fixing them when using the textual notation, but expressed preference for the graphical notation. Jolak et al. [99] conducted an experiment with 240 software students from four different universities to study the impact of the type of representation (textual vs. graphical) on user's ability to explain, understand, recall and actively communicate knowledge. They used UML class diagrams for the graphical representation. They have discovered that the graphical representation has a positive effect on the explaining and understanding ability, but with no statistical significance, while statistically significant advantage of graphical representation over textual representation is noticed for the recall ability. Also a statistically significant effect is noticed regarding the ability to actively communicate knowledge. Namely, it has been observed that the graphical representation fosters more active discussion than the textual representation and has a positive effect on the creative conflict discussions while requiring less conversation management effort.

Textual notation is frequently preferred to graphical notation by developers, due to their long tradition of using textual general purpose programming languages, which reduces their learning curve [22] for the DSL. Non-developers, on the other hand, might feel less comfortable with formal textual encoding of their knowledge and thus prefer graphical editors [110], which would explain the lack of textual editors for business-oriented DSLs, such as BPMN. Recently, DSLs are being adopted in new domains where the users are not trained developers or where developers need to work closely with non-developers when creating the models [219]. In such an environment a synchronized textual and graphical notation, where the user chooses which notation to use to create and edit the visual model, might be the win-win solution for the design of the editor. Experience with using UML at Ericsson [124] has shown that "when a DSL is large, covers a wide aspect and has different types of users, having one notation often does not suit the needs of all its users".

Only recently researchers have started discussing seamless hybrid graphical-textual modelling [4], motivated by the potential benefits of two-way synchronization, such as the definition of concern-specific views, the reduction in modeling time/effort and the possibility to edit textual models outside of the modeling environment. Furthermore, a hybrid editor can serve a broader range of mod-

eling purposes. Mature graphical editors with automatic layout make modeling more time efficient for an inexperienced modeler, and thus are more appealing for sketching models for discussion purposes. Textual editors, on the other hand, allow for precisely adding details for simulation or execution purposes which can remain hidden from the graphical visualization. Having both editors in the same modeling tool facilitates the transition between different modeling tasks.
In this section of the state of the art chapter we present a survey of existing solutions for textual, graphical and hybrid visual model editors inspired by our talk at DSLDI [89]. Based on that we derive a maturity model of editors supporting the use of text for visual modeling and we sketch a reference architecture of editors with each maturity cluster. To obtain the maturity model and the reference architectures, we have studied the features of over 30 mostly open source graphical, textual and hybrid editors. During the study, we also compared the DSLs and the user-facing features of textual editors. Starting from features identified in existing studies [62; 49; 189], we have added features observed in recent tools reflecting our focus on the textual input modality. The goal of our survey was not to identify an exhaustive set of editors, but to study a set of tools sufficiently diverse to identify different design alternatives and assess their maturity in terms of allowing the use of text to create visual models. Based on the results of the survey we have categorized the various design alternatives in textual editors with the aim of facilitating the sharing of design knowledge and fostering its reuse towards the next-generation of hybrid modeling tools.

3.3.1 Existing Tool Studies

Funes et al. [62] propose a requirements tree for evaluating UML tools from the user's perspective to facilitate tool selection. They look at tool's features, UML modelling support, customization, installation and performance, and support. Recently, Rajoo et al. [167] started working on a framework for empirical evaluation of UML modeling tools based on the productivity and the completeness of the tools in meeting the needs of healthcare informatics. Seifermann et al. [189] survey 36 textual notations, limiting them to UML as visual language. They propose a feature model, which includes the editor, but not the model editing or synchronization features of the same. Safdar et al. [183] have conducted an empirical study of the impact of three different UML modeling tools (IBM RSA, Magic Draw and Papyrus), on the productivity of the students involved in the study. While these works focus on facilitating the selection of a UML modelling tool by its users, in our survey we focus on the design decision to be taken by the developers of modeling tools for visual models.

Erdweg et al. [49] present a systematic overview of the design alternatives for language workbenches, i.e., tools for defining and composition of DSLs. They focus on functional properties of the language workbenches and categorize them in a feature model. In our survey we focus specifically on the “Editor" feature, we look at the different design alternatives proposed in [189] and [49], expand them, relate them to the textual input modality and derive a maturity model. Work has been done on maturity models in MDA [175] from the point of view of level of adoption of MDA practices in software development. However, we are not aware of any work on maturity models for modeling tools supporting text for visual modeling nor on reference architecture for the same.
In [129] Mazanec and Macek define important features of general purpose textual modeling languages, such as the ability to describe the entire software at various levels of abstraction, the readability and simplicity of the language, unambiguity, supportability and integrability. The UML textual editors that they used in their survey overlap with some of the ones we have surveyed, however the focus of their features' evaluation were not the editors, but the modeling languages themselves. Thus, while with their work they are targeting designers of textual modeling languages, we are targeting editor designers.

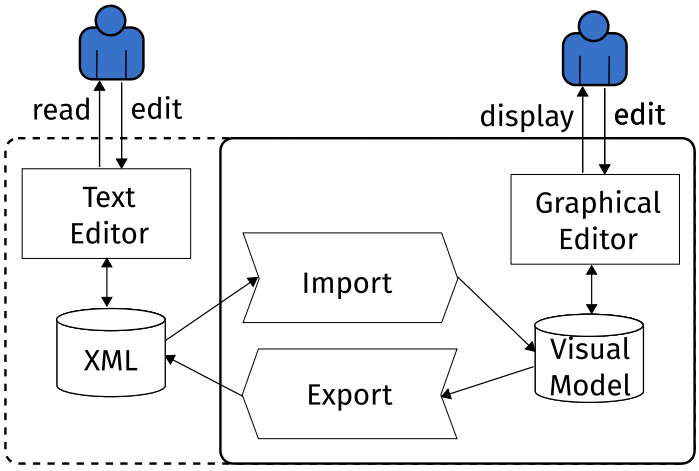
3.3.2 Maturity Model and Reference Architectures

Different editors, directly or indirectly, support visual modeling with text, allowing modelers to compile diagrams from textual descriptions, or to seamlessly switch between visual and text-based editing. A summary with the different maturity clusters we have identified in existing editors, their characteristics and representative tools is provided in Tab. 3.2.

Table 3.2. Text for visual modeling - Maturity model

	Characteristics	**Editors Embedded in the Tool**	**Representative Tools**
Cluster 0	Binary or encoded textual model representation	Graphical	Webdemo
Cluster 1a	Serialized textual representation (e.g., XML, SVG) editable outside the tool	Graphical	Signavio, Draw.io, Online Visual Paradigm
Cluster 1b	Serialized textual representation (e.g., XML, SVG) editable inside the tool	Graphical, Textual	Camunda, Eclipse plug-ins
Cluster 2	Textual DSL for visual element's properties, new elements added graphically	Graphical, Textual	Yakindu
Cluster 3	Textual DSL with text-to-visual one-way synchronization	Textual	PlantUML, ZenUML, Umple state machine diagram
Cluster 4a	Textual DSL with two-way synchronization, text in diagrams not editable in the graphical editor	Graphical, Textual	Ballerina, Sketch-n-sketch
Cluster 4b	Textual DSL with two-way synchronization, all visual elements can be edited in both editors	Graphical, Textual	Umple class diagram

Figure 3.1. Reference architecture for a model editor in Cluster 1

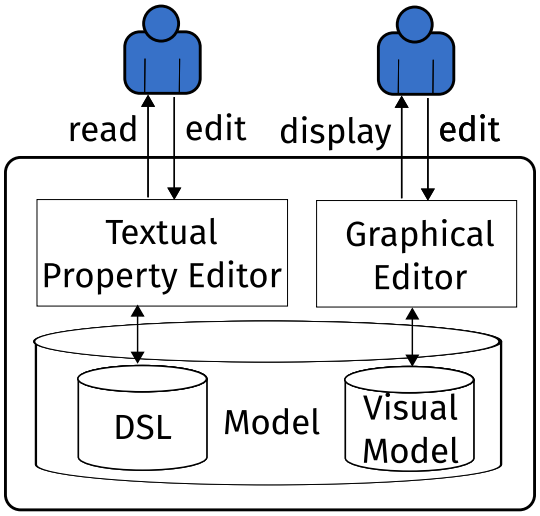


Every model is saved in some type of a computer readable format behind the scenes. Thus, for every diagram, regardless whether it is modeled using a textual or graphical editor, there is some persistent representation in textual or binary format. This may or may not be directly available to the users. In the graphical editing tools in ***Cluster 0***, the model can only be exported in a *predefined binary or encoded textual format* (e.g., ppt in Webdemo[4]) making it difficult to edit it using a textual editor.

In the editors in ***Cluster 1***, the *textual format is serialized* with direct mapping to the diagram elements and their position. The tools with this maturity allow exporting in SVG (e.g., Draw.io[5], Online Visual Paradigm[6]) or BPMN (e.g., Signavio[7]) as a standard XML-based format that can be edited in any text editor outside of the modeling tool. In this case, any changes to the textual representation will only be visualized after importing the model back to the modeling tool as evident in the reference architecture shown in Fig. 3.1. Camunda Modeler[8] and Eclipse[9] allow the XML representation to be edited inside the tool, in a separate tab that cannot be viewed at the same time with the visual graph tab (Camunda) or in a tab that is visible side by side with the visual graph (Eclipse). The changes in the textual representation are synchronized to the graphical model upon save (Eclipse) or upon window switch (Camunda). Although it is possible to manu-

[4] https://webdemo.myscript.com/views/diagram
[5] https://www.draw.io
[6] https://online.visual-paradigm.com
[7] https://www.signavio.com
[8] https://camunda.com/download/modeler/
[9] https://www.eclipse.org

Figure 3.2. Reference architecture for a model editor in Cluster 2



ally edit these standard textual representations of a visual model, this is rarely done in practice due to the verbosity of the textual representation.

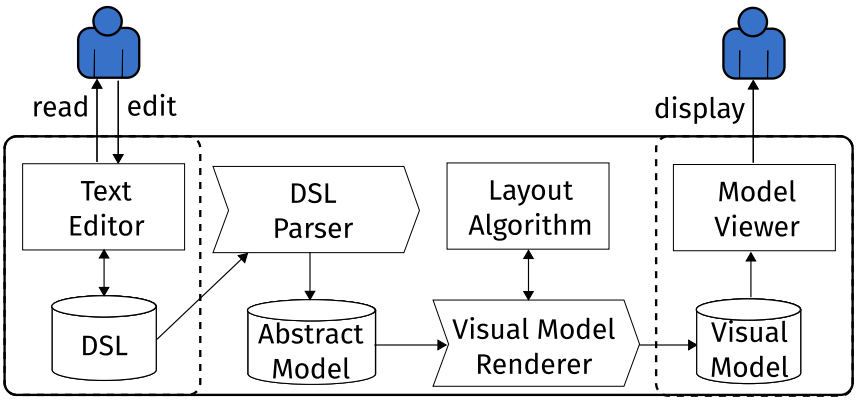
Tools in ***Cluster 2*** use *text for editing properties of the visual elements*. In Yakindu[10] UML elements in the state diagram have to be added using the graphical editor, but a textual editor can be used to encode the behaviour specification of each state. The textual editor is accessed by clicking on a specific state element in the graphical editor and uses a textual DSL which provides the simulation semantics for the model. As mentioned earlier, the textual DSL cannot be used to create the model as the tool requires the use of the graphical editor. The reference architecture of tools in Cluster 2 is provided in Fig. 3.2.

With the goal of increasing the acceptance level of MDE and improving the time efficiency of modelers who prefer using text, tools in ***Cluster 3*** use *textual DSLs* for visual modeling, which are at a higher abstraction level compared to the serialized text used in Cluster 1, and with a syntax that is meant to be more developer friendly and less verbose [189]. These tools (e.g., yUML[11], PlantUML[12], ZenUML[13], WebsequenceDiagrams[14]) provide only a textual editor linked to a graphical model viewer. Thus, only *one-way synchronization* from the text to the visual model is available as evident from the reference architecture in Fig. 3.3. A Layout algorithm is required in the architecture of such plain textual editors,

[10] `https://www.itemis.com/en/yakindu/state-machine/`
[11] `https://yuml.me`
[12] `https://www.planttext.com/`
[13] `https://app.zenuml.com/`
[14] `https://www.websequencediagrams.com`

to decide the position of newly created elements. One of the drawbacks of such architecture, is that due to the absence of a graphical editor, the layout cannot be manipulated manually, which results in a layout solution based on graph topology constraints which may be suboptimal for the needs of the users. Namely, the users might want to group the elements differently according to their intended semantics in order to improve the understandability of the model [197].

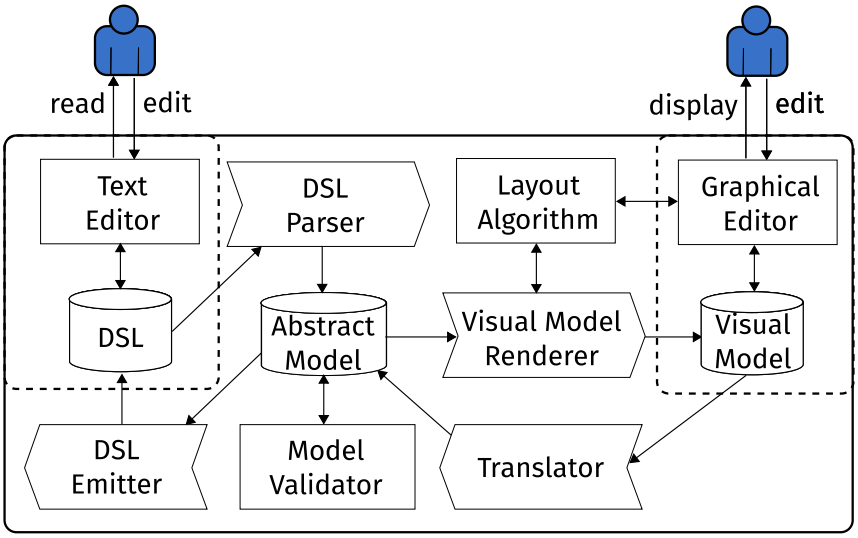
Figure 3.3. Reference architecture for a model editor in Cluster 3



Hybrid editors are in ***Cluster 4***. In Fig. 3.4 we present a reference architecture for these editors where the user has access to *two types of editors, a graphical editor and a textual editor*, each of them using their own DSL. In some of the tools we have surveyed, there is a direct translation from the textual to the graphical DSL [4]. This however hinders the flexibility of the architecture making it difficult to iteratively improve the DSLs while maintaining support for the previous DSL versions, like in the case of the new textual DSL for the UML Activity diagrams in PlantUML[15]. To that end we introduce a two-step translator which transforms the visual model into textual DSL (or viceversa) via an intermediate abstract form. As mentioned earlier for the tools in Cluster 3, translating from the textual to the graphical DSL requires also a Layout algorithm component to position newly created elements within the diagram. The intermediate abstract representation can be based on standards to make it possible to reuse and integrate multiple standalone textual or graphical editing tools. Likewise, model verification or validation features can be implemented based on this abstract representation so that error checking can be performed independently on whether the model was created visually or by using textual input.
In editors in Cluster 3 (Fig. 3.3, compared to hybrid editors in Cluster 4 (Fig. 3.4, there is no translation from the Visual Model to the Abstract Model and thus the

[15] `http://plantuml.com/activity-diagram-beta`

Figure 3.4. Reference architecture for a hybrid model editor in Cluster 4



DSL Emitter is not present as a component in the architecture of such tools. Furthermore, there is no Graphical Editor, but rather a read-only Model Viewer that displays the model to the user, but does not allow the user to edit it. As per the initial tool survey that we have conducted, hybrid editors as defined with the reference architecture in Fig. 3.4 are starting to appear (e.g., Sketch-n-sketch[16], Balerina[17], Umple[18]). However, in some tools not all model elements can always be edited from both the textual and the visual editor. For instance, although constructs in Ballerina can be added both in the textual and the graphical editor and the two views get automatically synchronized, some changes, such as renaming or setting conditions, have to be done in the textual editor. Thus, all edits to the labels of the visual elements have to be done in the textual editor. Similarly, the Sketch-n-sketch editor allows for adding a text element using the select and drop option, but editing the text inside can only be done in the textual editor. Umple, which supports textual DSL for visual modeling of UML class diagrams and state machines, provides a hybrid editor only for the UML class diagram, while for the UML state diagram only text-to-visual synchronization is provided.
In addition to the direction, another important design decision regarding the synchronization between the textual and visual model is the *timing* of the same. This is especially true for editors in Cluster 3 and 4, since it can impact the velocity of the visual feedback to the user. In most of the surveyed editors, the graphical and

[16] `https://ravichugh.github.io/sketch-n-sketch/releases/v0.5.2/`
[17] `https://ballerina.io`
[18] `https://cruise.eecs.uottawa.ca/umpleonline/`

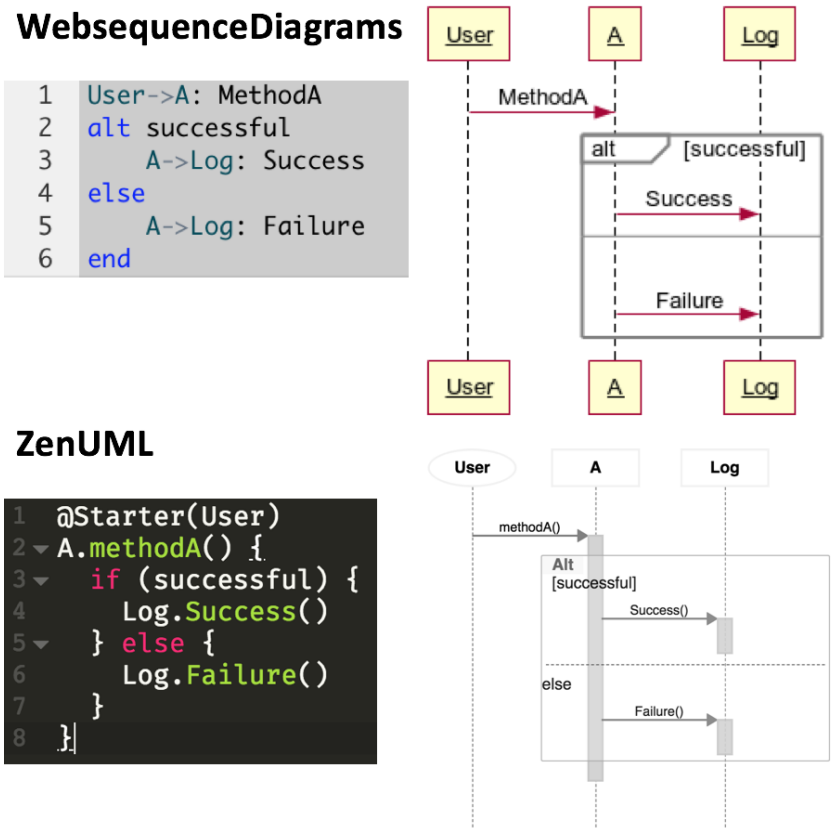
the textual representations are synchronized in real-time. Some editors require an explicit synchronization request. For instance, the synchronization happens after clicking the “Run Code" button in Sketch-n-sketch, or the “Refresh" button in the online PlantUML editor.
When using text for visual modeling, direct *links* between the textual and the graphical representation of a model element can facilitate model refactoring and correction. By direct link we intend a functionality where clicking on a given visual construct will shift the focus of the mouse to the corresponding point in the textual representation or vice-versa. While Ballerina and Umple only support the links from the visual element to the textual element, the WebsequenceDiagrams editor provides links in both directions. In the Sketch-n-sketch solution, when hovering over a visual element this will highlight in yellow the related content in the text editor, but will not bring the mouse focus to the same. Thus, if the textual representation is longer than one screen view, and the view has been scrolled down to the end of the text, if users clicks on the visual element whose corresponding text is in the beginning of the text, they will not be able to see the highlighting, and thus will not locate the searched text. As the graphical and textual editors are starting to blend, dynamic mapping of the two can facilitate model reviewing tasks, as needed modifications evident in the visual diagram can be easily tracked back to the text and viceversa. Using an internal abstract model representation can facilitate implementing such linking features as it provides a common reference for model elements that are displayed across multiple textual and graphical views.

3.3.3 Textual DSL Syntax

We can distinguish two types of textual DSLs used in visual modeling tools, based on the maturity model described in Sec. 3.3.2. Tools in Cluster 2, where the textual editor is complementary and not an alternative to the graphical editor, use textual DSLs only for defining the simulation or execution behaviour associated with the diagram elements. As such the DSLs used at this maturity are usually very close to programming language syntax. Given that the aim of these DSLs is not enabling the creation of visual elements in the diagram, in some tools changes in the textual DSL do not result in visual changes to the model. For instance, in Chalktalk[19] by clicking on a graphical shape a textual window pops out which allows the user to edit the behaviour defining variables via code. In Yakindu there are two types of textual editors with their dedicated DSLs. One is

[19] `https://github.com/kenperlin/chalktalk`

Figure 3.5. Sequence diagram definition in WebsequenceDiagrams vs ZenUML



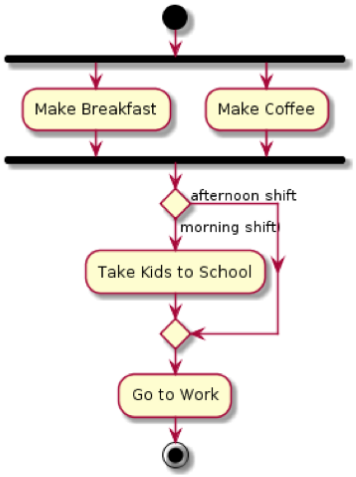
the statechart definition section used for defining interfaces and their variables, which as such does not result in any visual diagram changes. On the other hand, as mentioned earlier, in the properties section of the state elements in Yakindu there is a dedicated textual editor for defining the state behaviour which supports content assistance, error checking and syntax highlighting and reflects the changes in the visual diagram.

In editors in Cluster 3 or Cluster 4, the aim of the textual DSL is to represent the elements of the visual model in order to enable the creation of such models from scratch. Standard visual DSLs, such as UML, still do not have corresponding standard textual DSL [189], thus the designers of the textual editors have to design also the syntax of the textual DSL. In our tool survey we have identified several different approaches being taken. Many, like PlantUML, try to keep the textual DSL as close to *natural language* as possible, with the intention to make it easy to use for non developers as well. On the other hand, the DSL of some tools, such as ZenUML, is created aiming to get the textual DSL closer to developers,

Figure 3.6. Activity diagram definition in PlantUML vs yUML

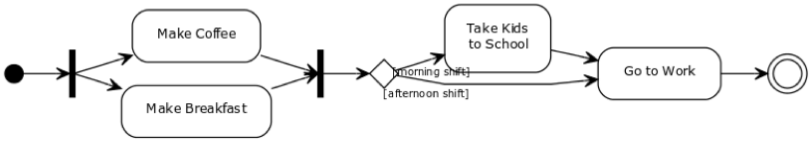
PlantUML

```
1  @startuml
2  start
3  fork
4  :Make Breakfast;
5  fork again
6  :Make Coffee;
7  end fork
8  if () then (morning shift))
9  :Take Kids to School;
10 else (afternoon shift)
11 endif
12 :Go to Work;
13 stop
14 @enduml
```



yUML

```
(start)->|a|,
|a|->(Make Coffee)->|b|,
|a|->(Make Breakfast)->|b|,
|b|-><c>[morning shift]->(Take Kids to School)->(Go to Work),
<c>[afternoon shift]->(Go to Work)->(end)
```



using *programming language* (Java/C#) syntax whenever possible.
For instance, as shown in Fig. 3.5 describing conditional sequence diagram behavior in ZenUML uses the typical `if(true){doSomething} else {doSomething}` programming language syntax, while in PlantUML the same is expressed as follows: `alt true doSomething else doSomething`.
In the Activity diagram in Plant UML, a semicolon and a new line is used to separate activities. However, the existence of gateways, where the control flow diverges or converges, has to be known by the modeler and explicitly stated in the textual DSL with `if else` or `fork` for parallelism, as evident in Fig. 3.6. In yUML, as evident in Fig. 3.6, the activity has to be stated between parenthesis and a textual arrow indicates the next activity, thus getting the *textual DSL visually closer to the graphical DSL*. Gateways have to be named and enclosed between the pipe character (i.e., `|a|`) to depict parallelism or between angle brackets (i.e., `<b>`) for alternative divergence. In some textual editors not everything that is

represented in the textual DSL is reflected in the diagram. For instance, in Umple the textual DSL contains information about the key attribute, while in PlantUML it is possible to specify element identifiers which, although not explicitly shown in the diagram, are used to disambiguate references to the same model element.

3.3.4 Textual Editor's Features

When designing a hybrid editor, three main user facing features should be considered, both in the textual and graphical editor: editing (addition, modification and deletion) of model elements, the layout of the model elements and the navigation within the diagram. Based on our survey, we discuss these features and their design alternatives in a textual editor for visual models (Clusters 3 and 4).

Model Editing The most common way to ***add elements*** in a textual editor is by *typing-in*, following the DSL syntax. However, the WebsequenceDiagrams editor applies a hybrid approach, where in addition to typing, to add elements the user can click on frequently used *patterns which are graphically visualized*. The text corresponding to the added pattern can later be modified to create the desired diagram. This editor also supports auto-completion from a *list of suggestions*, differentiating between keywords and participants. The look and feel of the diagram (sketch-like, colour, shapes etc.) can be selected from the visual catalog of predefined styles. Some editors allow *adding comments or notes* to the diagram. In PlantUML and WebsequenceDiagrams this is done by using the `note` keyword, while in ZenUML the programming language like `//comment` is used. PlantUML also allows to add titles to the diagram, and split the diagram into differently titled pages. We have observed only one option for ***deleting elements*** in textual editors, i.e., using the *keyboard*. When it comes to ***editing elements***, some of the textual editors, such as Ballerina and Yakindu Statecharts, support *refactoring*.

Layout A plain black and white, straight-line text, with no indentation, etc., can quickly become cumbersome to read and edit. Textual editors found within programming IDEs provide support for enhancing the readability of the code through features such as ***syntax highlighting***, ***semantic indentation***, better visualization of nested logic etc. However, not all textual editors found in the surveyed modeling tools take advantage of these improvements in the user experience. From the plain textual editors that we have surveyed, i.e., the editors in Cluster 3 that do not support graphical editing, only ZenUML provides pretty printing functionalities. On the other hand, most of the hybrid editors in Cluster 4, i.e., editors

which support both textual and graphical editing, such as Ballerina, Umple etc., do provide syntax highlighting and indentation.

Navigation Navigation assistance can be quite important in textual editors, especially when the diagrams become long and complex, requiring a lengthy text file to generate them. We have observed two types of navigation assistance in the studied tools, a ***navigation overview*** window which reflects the current position of the cursor in the entire text, implemented by only one of the surveyed editors, Ballerina. Another complementary option is to support ***text folding***, where related chunks of text can be folded, for instance conditional statements or nesting, to provide a more general view of the text. This feature is only used by Ballerina and ZenUML from the surveyed editors.

To summarize, based on our initial survey, the textual editors for visual modeling have still not reached the state of the art of modern programming language editors. They lack common features such as syntax highlighting or pretty printing. Of the surveyed editors, only Ballerina and Yakindu support refactoring and none perform quick fixes in response to detected errors. This is inline with the findings in [189]. Even features like text folding and navigation overview are rarely present. As these are features which modelers with preference towards textual editors are well acquainted to, the support of such features in a model editor, or the lack thereof, can impact users' acceptance of the same.

To conclude, as technology gets more and more integrated with business, hybrid textual and graphical editors would allow different user profiles to collaborate on modeling tasks by using the type of input modality that they prefer. To facilitate the transition towards hybrid editors, we have set up the foundations for a reference architecture for hybrid editors and proposed a maturity model based on their features. To that end we have surveyed different mostly open source editors, and encoded the gathered design knowledge. The list of design issues that we discuss for textual editors is not meant to be exhaustive, nor are the alternatives for each issue. However, we believe that they can be the first step towards providing guidance and support for the design process of the next generation of hybrid model editors.

3.4 Evaluating DSLs and DSL Tooling

As with any design, the design of a DSL and its tooling also need to be evaluated from different aspects. "An evaluation is either an experiment or an observational study, consisting of steps performed and data produced from those steps" [21]. Evaluation is used in many different areas such as marketing, medicine, software engineering etc., and can go from simple surveys to complex controlled experiments.
Molleri et al. [140] provide a recent literature review on available guidelines for empirical research in software engineering, while Gueheneuc and Khomh [78] discuss the rise of empirical software engineering and the seminal works that led to the same together with the challenges to be faced in the future. One of the seminal works that they mention as the "inflection point in software engineering" is the work of Wohlin et al. [227] who present the characteristics of different qualitative and quantitative research techniques such as literature reviews, surveys, case studies and experiments. A literature review is a systematic collection of publications to study the state of art of a phenomenon of interest. Surveys aim to study the public opinion on a topic and they usually collect data through questionnaires or interviews. Case studies usually observe a phenomenon in a real-life context, while experiments are conducted in a controlled laboratory environment. Wohlin et al. [227] then focus on experimentation, providing examples and explaining in detail the steps needed before, during and after conducting an experiment. Zeller et al. [231] in their parody of empirical research take a more sarcastic approach to identifying pitfalls to the analysis of empirical studies data and provide useful references to positive examples of empirical research.
A good evaluation starts with a careful **design** of the evaluation procedure. Wettel and Lanza [225; 224] provide guidelines for good experiment design based on a literature survey of experimental validation in software engineering, information visualization, and software visualization approaches. They call these guidelines an experimental design wish list and include the following in the same: choose a fair baseline for comparison, involve participants from industry, take into account the range of experience level of the participants, provide a tutorial of the experimental tool to the participants, find a set of relevant tasks, include tasks which may not advantage the tool being evaluated, limit the time allowed for solving each task, choose real-world systems, include more than one subject system in the experimental design, provide the same data to all participants, report results on individual tasks and provide all the details needed to make the experiment replicable.
When it comes to the **analysis of the data** collected during the evaluation dif-

ferent approaches can be taken depending on the research techniques used. The goal of exploratory surveys as a qualitative evaluation technique for example is not to make statistical inference regarding the population, but simply to gather deep understanding of the identified problem [227]. A controlled experiment on the other hand has the goal of making statistical inference about the population based on a hypothesis. However, the selection of the right statistical test is not straight forward and depends on many factors explained step by step by Marusteri and Bacarea in [125]. They describe in detail each factor (the type of data, the distribution of the population from which the sample is taken, the sample number and size, the dependency between the sample groups etc.) and provide a decision-tree to guide readers into selecting the appropriate statistical test for comparing means (medians) of one, two or more samples.
Blackburn et al. [21] provide useful instructions on how to make useful and sound claims based on the conducted evaluation where a claim is defined as "an assertion about the significance and meaning of an evaluation". They warn against two types of sins when making claims: the sin of reasoning and the sin of exposition. The sin of reasoning can happen when the scope of the claim and the scope of the evaluation do not perfectly overlap which might mean that some important evaluation data or data distribution is being ignored, or that the claim uses data that is missing from the evaluation or uses inappropriate metrics, or the evaluation compares incompatible entities. Avoiding the sin of reasoning is not always straightforward as evident from examples of unintentional misuse of independent variables or use of inappropriate metrics in the software development community that went unnoticed for years. The sin of exposition refers to inadequate description of the evaluation or the claim, or even a complete omission of a claim.
Last but not least, the identification and discussion of the **threats to the validity** of the conducted empirical research is of utmost importance for complete understanding of the research results and their impact. In the words of Zeller et al. [231], threats to validity "aid the reader by creating a lens through which the results can be viewed and interpreted". However, it was only in the 1990's that empirical research papers in software engineering started explicitly stating the threats to the validity of their work [78]. Wohlin et al. [227] discuss several types of threats to validity to be analysed. *Constructs validity* refers to the need to identify the correct measures and experimental setting to be used depending on the research goal and the hypothesis to be tested. For instance, a threat to this validity can be the fact that the selected participants might not be representative of the population, or the lack of mutual understanding of the meaning of tasks/questions between the participants and the survey/experiment designers.

Internal validity refers to whether the differences in the values of the dependent variable are caused by the treatments of the independent variables. The factors to be considered are the selection criteria of the study participants and how they are divided into groups. *External validity* refers to the validity of generalizing the results beyond the experiment setting. Some possible risks are wrong participants as subjects, the created environment being too artificial, or the timing of the experiment affecting the results. *Conclusion validity* refers to the ability to draw correct conclusions regarding the relation between the treatment of the independent variable and the outcome of the dependent variable. This validity can be affected by the choice of statistical tests, sample size, measurements etc. When it comes to the evaluation of DSLs themselves, Van Der Linden [214] propose a framework for applying the Physics of Notation theory to validate the design of visual notations [142]. They define a systematic approach for the reporting on the design of a visual notion based on the nine principles described in the Physics of Notation theory. Their framework uses a reporting template to be filled in for each principle. Kahraman and Bilgen [102] propose a framework for qualitative assessment of DSLs based on a combination of ISO software quality standards, capability maturity level evaluation and perspective based assessment. They adjust the ISO/IEC 25010:2011 standard to the needs of assessing DSLs and identify the following quality characteristics of a DSL: functional suitability, usability, reliability, maintainability, productivity, extensibility, compatibility, expressiveness, reusability and integrability. The authors believe that the success of a DSL depends on the goals of the evaluator who needs to map his/her goals to the above mentioned characteristics.

3.5 Chapter Summary

In this chapter we survey existing work in modelling RESTful APIs. Substantial work has been done and tool support has been provided for modelling the structure of RESTful APIs, while the work done in modelling the behaviour of the same is not domain specific. The languages used to visualize RESTful interactions rely mostly on UML State diagrams or UML Sequence diagrams, Petri Nets or Coloured Petri Nets, or BPMN Choreography diagrams. The main identified drawback of the currently used approaches is that, although they manage to model the behaviour from a certain viewpoint, they do not visually emphasise the main entities and properties of RESTful conversations. The situation does not change much also when modeling microservices which communicate over REST APIs. That said we have decided to dedicate our research to design a DSL for

modeling the behaviour of REST APIs. However, before starting with the design we looked at different existing work on guidelines for the process of designing a DSL in general, but also guidelines for design of visual or textual notations in particular which we discuss in this chapter. In order to gather ideas for building an editor for our DSL, we have studied the design of modelling tools, with special focus on tools which support textual modelling of visual DSLs. Based on the study we have built a maturity model to classify the different types of support for textual modelling that different tools offer and sketched a reference architecture for each maturity model. To conclude this chapter, we have surveyed existing work in the evaluation of languages and tools in software engineering to get guidelines on appropriate design of evaluation methods given a goal and appropriate analysis of the results.

Part II

RESTalk

Chapter 4

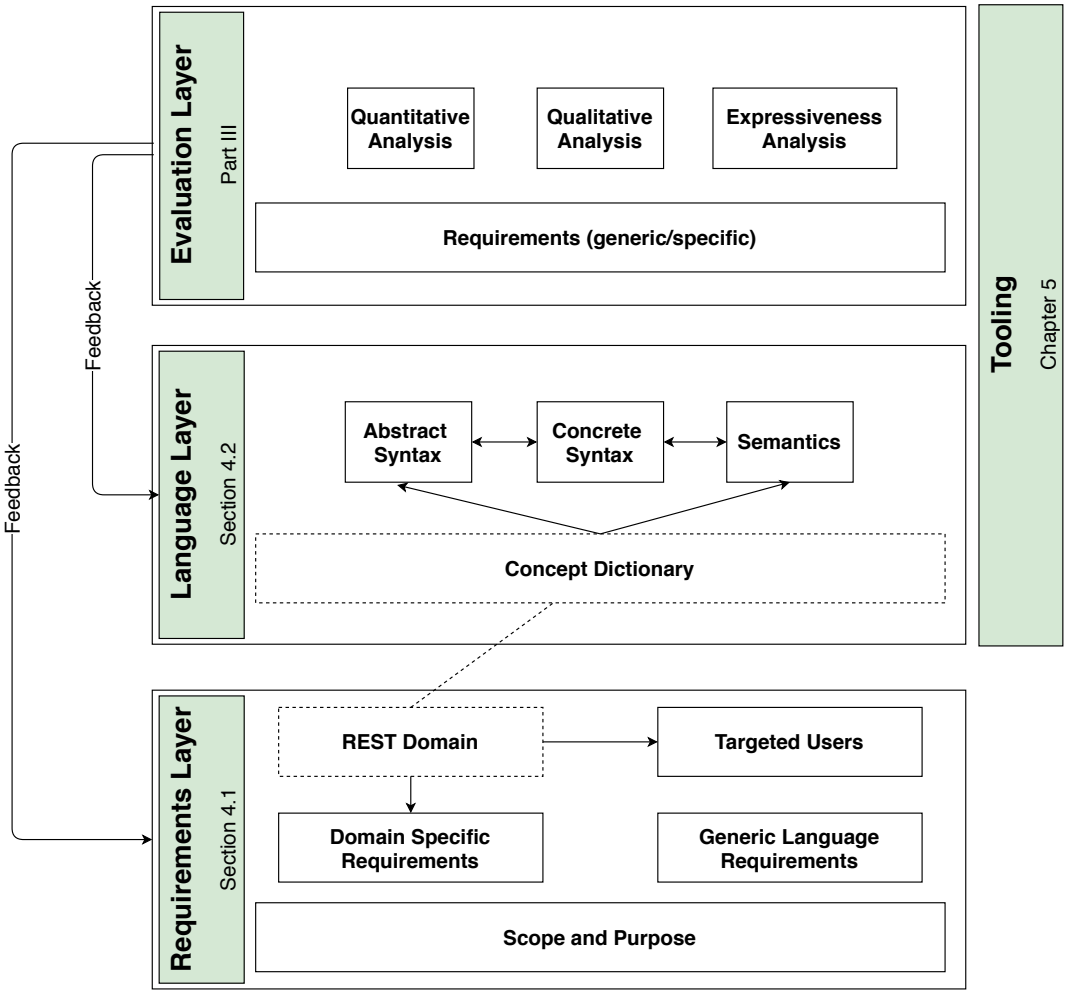
RESTalk Language

"A Domain-Specific Language (DSL) is typically a small, highly focused language used to model and solve some clearly identifiable problems in a domain; in contrast to a General-Purpose Language (GPL) that is supposed to be useful for multiple domains. [138]". While using standard UML sequence diagrams to visually represent a sample of RESTful conversations in [81], we have realized its limited capability to emphasise the most important facets of RESTful interactions, and thus the need for a domain specific notation, a need which has been confirmed during our review of the state of the art in Chapter 3. As the constructs of DSLs are tailored for a particular application domain they are considered to offer benefits in expressiveness and ease of use compared to a GPL [133]. Lindland et al., in their framework for understanding the quality in conceptual modeling [119], claim that a very important aspect of a modeling language is its domain appropriateness.
That said, in this chapter we will discuss the main contribution of this thesis, i.e., the design of RESTalk, a DSL for modeling the interactions with REST APIs. During the design process we have combined the practices discussed in Sec. 3.2, following the three layer framework described in [94] and adopted for RESTalk in Fig. 4.1. We will discuss the requirements layer in Sec. 4.1, passing then to the language layer discussed in Sec. 4.2.

4.1 RESTalk Requirements Layer

The requirements layer in a DSL design sets the ground for the next layer, i.e., the language layer [94]. This layer includes the definition of the scope and the purpose of the language, as well as the language requirements and the target users of the language.

Figure 4.1. RESTalk Development Framework (adapted from [94])



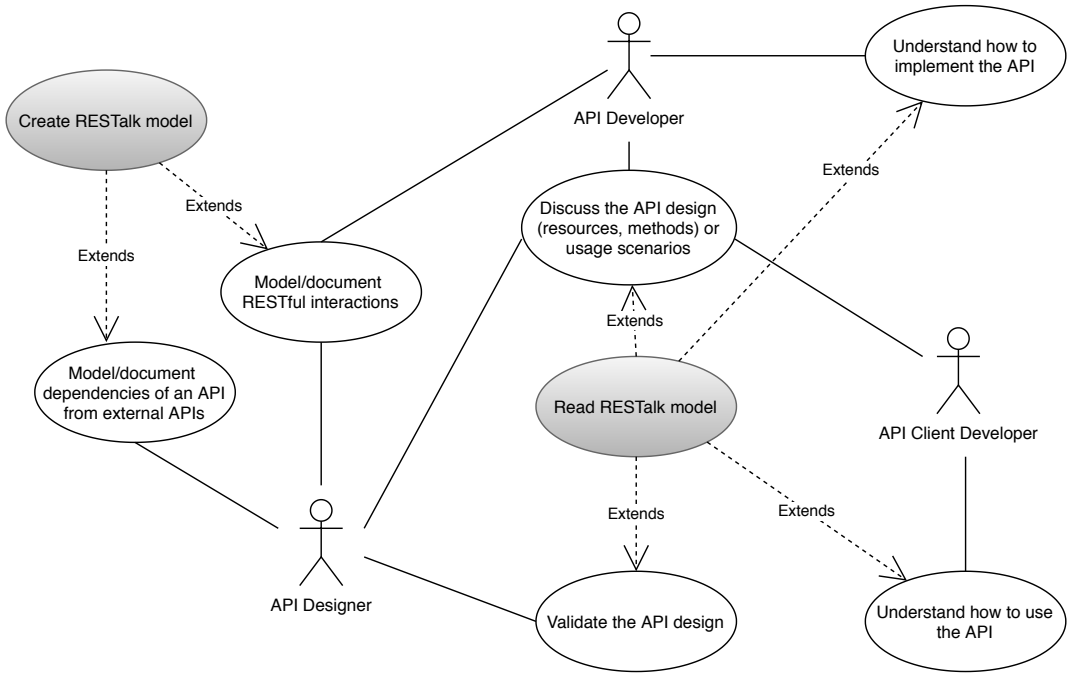
4.1.1 Language Scope and Purpose

REST APIs can be analysed from a static, structural aspect, and a dynamic behavioural aspect [187]. API description languages, such as OAS, already cover well the static aspect defining all of the resources and all of the methods that each resource supports. Thus, it is the behavioural aspect that is in the **scope** of RESTalk, reflected in what we define as a RESTful conversation in Sec. 2.3. Thus, RESTalk does not aim at making the resources the first-class citizen of the DSL, as is the case with OAS, but rather the interactions between resources. Although the initial goal of RESTalk was the visualization of interactions between a single client and a single server, its scope has been extended to also support RESTful conversations between clients with different roles and a single API, as

well as composite conversations with multiple layers to show the dependencies a given API has from external API(s). In the last mentioned case the internal API acts as a client to one or more external APIs, in the sense that when one of its endpoints is being called, in order to provide a response, the internal API needs to call the APIs of one or more external service providers, thus creating dependencies on those APIs. In this later use case, RESTalk diagrams are intended as a complementary documentation to the existing software architecture documentation. Modeling conversation of multiple clients talking to multiple servers is out of the scope of this work.
The ultimate goal of RESTalk is to support designing, documenting, analysing and learning RESTful services by visualization of the interactions needed to achieve predefined goals, thus facilitating the communication between stakeholders. Given the context, we have identified three main stakeholders as **target users** of RESTalk: API designers, API developers and API client developers.
RESTalk diagrams can be used as internal documentation in a given company, or made available to external audience when developing API clients. Depending on the applied methodology, RESTalk diagrams can be created before developing the API (Design First approach) or after developing it (Code First approach). Internal facing APIs, where the speed of delivery is important, usually use the Code First approach. For APIs targeting external customers or partners, the API's interface is a contract, and as such it cannot be changed frequently and radically as that would cause clients to break. Thus, lately with the evolution of API description languages, this type of APIs follow the Design First approach where the design stage is separated from the development stage. The design stage follows an agile approach by requesting frequent feedback on the API design from future API client developers [198]. In this context, RESTalk diagrams are envisioned to be created by the ***API designer*** as a complementary documentation when designing and modeling RESTful interactions, or by the API developer when documenting the API. RESTalk diagrams could also be generated automatically from OAS documentation if sufficient data is available. Once created, RESTalk diagrams can be used to discuss the API design and usage scenarios [20] among the different stakeholders, thus facilitating the feedback loop. The API designer could also use the diagram to validate the API design, in terms of naming consistency, the use of appropriate HTTP methods, the use of appropriate design patterns, the appropriateness of the relationship between resources etc. ***API developers*** themselves could use the diagram to understand how to implement the API, which hyperlinks to include in the responses, which response codes to support etc. And last but not least, ***API client developers*** could use RESTalk diagrams in their learning endeavours to understand how to use the API. In Fig. 4.2 we provide a use

case diagram showing how RESTalk can be used as an extension of the use cases different stakeholders have.

Figure 4.2. Use Case Diagram for RESTalk's Stakeholders



4.1.2 Language Requirements

The generic requirements of each DSL refer mainly to the need of specifying the language syntax and semantics, and thus are no different for RESTalk [94]. What differs for RESTalk are the specific language requirements which are derived from the properties and constraints of a RESTful conversation as well as the use cases presented in Fig. 4.2. We have identified the following specific requirements for a DSL for modelling and visualization of RESTful conversations, taking into consideration the core requirements of a DSL defined in [111]:

- *RQ1*: The DSL should provide constructs to support the modelling and visualization of relevant entities in a RESTful conversation (e.g., request, response etc.). The relevant entities are defined in details in Sec. 4.2.1. This is referred to as the conformity requirement. Care has to be taken of orthogonality which requires that "each construct in the language is used to represent exactly one distinct concept in the domain".

- *RQ2*: The DSL should enable the modelling and visualization of the order of interactions with a given API depending on users' goals. To that end, there should be a mechanism to show sequences of interactions, but also control flow divergence and convergence, due to decisions on client side or server side, or due to response time-out.
- *RQ3*: The DSL should support the modelling and visualization of the gradual discovery of available resources by the client with the help of hyperlinks or through the discovery of data to be used in subsequent requests.
- *RQ4*: The core of the DSL should focus on modelling one client - one server interactions. However, there should also be constructs supporting the interactions between multiple participants.
- *RQ5*: The DSL should be flexible enough to support adding optional concepts to support the visualization of details relevant to the intended use of the diagram (e.g., request header details when relevant, SLA restrictions etc.). This requirement is also known as extensibility.
- *RQ6*: The DSL should take into consideration the understandability vs. expressiveness trade-off. While supporting different use cases requires for grater expressiveness of the language, an important requirement of any DSL is the simplicity, i.e., "a language should be as simple as possible in order to express the concepts of interest and to support its users and stakeholders in their preferred ways of working" [111].
- *RQ7*: The DSL should set up constraints to ensure that the modelled RESTful conversations are valid.
- *RQ8*: The DSL should support both visual and textual concrete syntax to satisfy the needs of different stakeholders and different use cases.
- *RQ9*: There should be tooling around the DSL to support the creation and editing of the models. This is known as the supportability requirement.

The above stated language requirements have been used when defining the RESTalk constructs and constraints as described in the next section.

4.2 RESTalk Language Layer

OMG's Meta-Object Facility standard [3] defines four modelling layers: meta-meta model (MOF M3 layer), metamodel (MOF M2 layer), model (MOF M1

layer) and data layer (MOF M0 layer). The M3 layer is the language used to build metamodels (M2) which define the elements in a M1 model. The last M0 layer is used to describe real-world object. In Model Driven Architecture (MDA), "a model is a description of (part of) a system written in a well-defined language", while "a well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer" [107]. A language has an abstract syntax and a concrete syntax. The abstract syntax identifies the modeling elements, their semantics and the relationship between different modeling elements. As such it is frequently presented in a metamodel (MOF M2 layer). For RESTalk we show the metamodel in Sec. 4.2.1. The concrete syntax defines how the modelling elements should be represented by graphical or textual elements [13]. A model can have multiple representations of its abstract syntax into concrete syntax, as is the case with RESTalk, which has both graphical visual representation presented in Sec. 4.2.2 and textual representation presented in Sec. 4.2.3.

4.2.1 RESTalk Abstract Syntax and Semantics

The language realization guidelines in [103] advise to compose existing languages and reuse language definitions when possible as the development of a new language from scratch is labor-intensive. Since Business Process Model and Notation (BPMN) Choreography diagrams [223, Chap. 5] focus on the exchange of messages with the purpose of coordinating the interactions between two or more participants [101, pg. 315], while precisely describing the partial order in which the interactions may occur, we have decided to use its metamodel as basis for designing the interaction concepts of RESTalk. BPMN is an ISO standard since 2013 (ISO/IEC 19510). On the other hand, RESTalk's domain is the REST architectural style and as such the REST specific concepts have been borrowed from the OpenAPI metamodel [32; 48] and the REST metamodel [207; 152]. However, as stated in [103], "when designing a language not all domain concepts need to be reflected, but only those that contribute to the tasks the language shall be used for". Thus, RESTalk's metamodel does not include all of the concepts defined in the above mentioned three metamodels used as basis for its design, but only the ones relevant given its scope. As advised in [57; 94], before defining the metamodel, we present the **concept dictionary** for the RESTalk domain. The concept dictionary is a glossary whose primary goal is to ensure that the intended domain is correctly captured and complete. It contains the definition of domain relevant concepts and their attributes.

1. **Conversation:** A conversation (MOF M1 layer) is a set of possible sequences of interactions between conversation participants. A conversation instance (MOF M0 layer) is a subset of the conversation including the interactions executed at a single conversation instantiation. A conversation is comprised of one or more possible conversation instances. It can, but does not have to, include all the possible conversation instances with the given API.

2. **Composite Conversation:** A composite conversation is comprised of two or more conversations which are interconnected, as a request in conversation n triggers the conversation $n+1$ and the server in conversation n will only return a response when the conversation $n+1$ is finished.

3. **Participant:** A participant is a client or a server which takes part in the conversation. There is always at least one client and one server that participate in the conversation. This means that it is also possible for different types of clients to talk to the same RESTful server. A participant can act both as a client and as a server in composite conversations where once a request is received, the server can become a client in a subsequent request as it calls one or more different REST APIs in order to be able to respond to the initial request of its client.

4. **Sequence flow:** The sequence flow shows the order in which the client-server interactions can happen in a given conversation. Different flows are possible depending on client's ultimate goal.

5. **Gateway:** A gateway is used to control the divergence or convergence [101, pg.287] of the flow of the conversation. The divergence/convergence of the flow can be due to client's decisions, e.g., to navigate to a given resource or to end the conversation, or due to different alternative responses that can be sent by the server.

6. **Event:** As defined in BPMN [101, pg.29], an event is something that happens and affects the flow of the conversation, i.e., it impacts the normal request-response sequence of the conversation. It can be, for example, a server time-out or a decision by the client to stop the conversation after reaching the goal.

7. **Resource:** A resource is any concept that can be named and can carry information that is of interest to potential users of the REST API. Each re-

source has a path (URI) which makes it the conceptual target of a hypertext reference/link.

8. **Request:** A request is a call sent to an individual endpoint (path to a given resource) with a standard HTTP method (e.g., GET, POST, PUT, DELETE) used to define the type of operation to be performed on the resource.

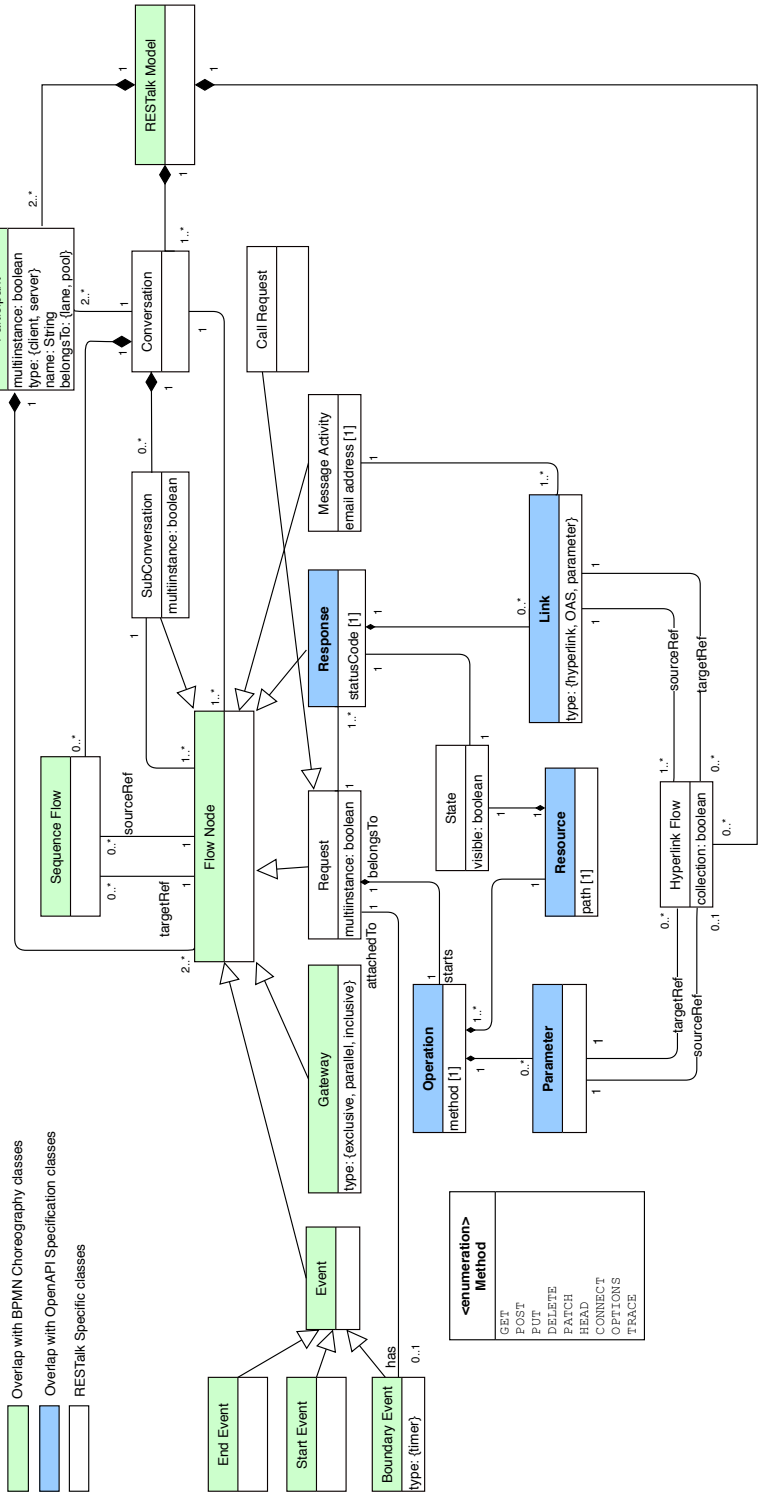
9. **Response:** A response is the data sent by the server as a consequences of a client's request. The data in the response will depend on the request and on the state of the requested resource. The response always contains a standard response code and optionally a representation of the requested resource, when available. In some human-user facing APIs in addition to the response the server can also send an e-mail message to the human-user containing hyperlinks for future requests.

10. **Link:** A link can be a hyperlink, a parameter that needs to be used to construct the URI in a subsequent request from a given URI template [74], or a reference to a link parameter defined in the OAS documentation. Links are necessary to enable future interactions with the API.

11. **Parameter:** A parameter is a variable that needs to be added in the path of the request or in the request header. A path can contain one or more parameters, depending on the URI template used. The value of the parameter changes at run time during the execution of different conversation instances.

Metamodeling has became "the prevailing technique for describing the abstract syntax of modeling languages: meta-classes represent all modeling concepts, meta-attributes their variations, and meta-associations their relationships" [13]. The reason for defining metamodels is "to precisely describe a language so that modelling tools, such as editors, can be created to support the use of the language" [157]. In Fig. 4.3 we present RESTalk's metamodel which depicts the relationships between the concepts defined in the concept dictionary. The concepts borrowed from the BPMN Choreograpy metamodel [101, pg.322] are colored in green. The classes coloured in blue refer to the concepts which overlap with the Open API Specification metamodel, and thus provide an opportunity for automating the derivation of RESTalk diagrams from the specification. The remaining white classes are specific to RESTalk.

Figure 4.3. RESTalk meta-model



The semantical meaning of most of the classes in the domain specific language metamodel is defined in the concept dictionary above. To explain better the behaviour of a REST API we have added some abstract classes which we explain below, together with the attributes and relationships between the main classes.

1. A *Conversation* can contain multiple flow nodes, but each flow node belongs to just one conversation;

2. A *Sub-conversation* has the same characteristics as the conversation, i.e., it can also contain multiple flow nodes. The difference is that a sub-conversation is a flow node itself and as such is contained inside a conversation and thus it has to have a source and a target reference sequence flow. A conversation can, but does not have to, contain sub-conversations. A sub-conversation can be used to delimit logical groups of interactions, for instance a set of interactions that accomplish a certain goal or subgoal, or it can also be used for groups of interactions executed in a *Multi-instance loop* iterating over a given array parameter retrieved in a previous response. For each element of the array the same sub-conversation is executed;

3. A *Start event* marks the beginning of the conversation. An *End event* marks the end of the conversation, when the API client stops sending further requests as it has achieved its goal. A boundary event of type *Timer* shows alternative paths to be taken in case the server takes too long to respond to the request. The alternative path can lead to re-sending the request, if the response is crucial for the conversation, or it can simply continue to the next request. The timer event element is used attached to the request element to show its interrupting nature [101, pg.342] that breaks the normal request-response sequence, and introduces a request-timeout-request sequence;

4. In a normal conversation execution, without timeouts, each *Request* is followed by at least one *Response*. The request corresponds to exactly one *Operation* in OAS terminology which contains the HTTP method (GET, POST, PUT, DELETE) that can be called on the *Resource* that the request refers to. The resource is identified by its URI, i.e., its *Path*, and has a *State* which is reflected in its representation sent with the response. As resources can be called with different methods, each resource can have one or more operations;

5. Three types of *Gateways* are supported in RESTalk: XOR - exclusive gateway that allows only one of the outgoing flows to be taken. This is the gate-

way type that must be used when modeling alternative server responses; OR - inclusive gateway that allows one, some or all of the outgoing flows to be taken; AND - parallel gateway that requires all outgoing flows to be taken. Similar logic is used when the gateways are used to converge the flow. Namely, in order to continue the conversation after an XOR - exclusive join gateway the response from only one of the incoming flows has to be received; for OR - inclusive join gateway the responses from all flows that have been activated with a split need to be received; for AND - parallel join gateway the responses from all concurrent flows need to be received;

6. The *Message Activity* element is to be used to model messages send by the RESTful server to the email account of a human-user. Thus, a message activity element has to contain the email address attribute. The email can contain one or more links which the user can follow in order to continue the conversation with the RESTful server;

7. *Flow node* is a sub-conversation, request/response, gateway, event, or message activity node in the RESTalk graph and as such can be a source or a target of a *Sequence flow* which represents an edge in the RESTalk graph. When a request can only have one response the request is directly followed by the response and there is no need to use an edge between them, which is the case when a response flow node is not a target reference of a sequence flow. Some types of flow nodes, such as the start event or boundary event, can only be a source reference of a sequence flow, while the end event type of flow node on the other hand can only be the target reference of a sequence flow. Requests, responses, message activities and sub-conversations are the source reference of exactly one sequence flow and the target reference of a different sequence flow. Gateways are the only type of flow nodes that can be the source or the target of multiple sequence flows depending on whether they act as a split or a join;

8. The *Participant* element refers to the client(s) and the server(s) who participate in the conversation. When there is just one client and one server there is no need to explicitly state the name of the roles as the client is always the one that makes the requests and the server is always the one that provides the responses. Explicitly differentiating between the participants becomes necessary when there are multiple types of clients using the same API in which case we use *Lanes* to depict them stating the name of the client explicitly on top of the lane. Thus, a lane is a container of a conversation or a part of a conversation, between a single client and a single

server. When applicable, sequence flow can pass between lanes as it means that the participation of multiple types of clients is needed to achieve the given conversation goal. Explicitly differentiating between the participants is also needed in composite conversations when a server becomes a client calling an external REST API to obtain data necessary for sending the response to the initial client, in which case we use *Pools* to depict the different servers being involved in the RESTalk model. Each pool contains a conversation between the specified participants in that pool. Thus, a pool is a container of a conversation with a single server and can contain none or multiple lanes. When using pools a *Call Request* has to be used to specify that the response to that request will be provided only after the called external API provides its response. The name of the client needs to be stated on top of the pool while the name of the server needs to be stated at the bottom of the pool. Sequence flow cannot cross between pools as each pool defines a separate conversation and dependencies between conversations are depicted by using multiple pools. Thus, in composite conversations a RESTalk model contains multiple pools. Lanes and pools assign different flow-elements to different participants;

9. The *Hyperlink flow* basically shows the flow of data during the interactions. In a single conversation it highlights the usage of resource identifiers discovered from previous responses. It acts as an edge whose source reference is a *Link* that is found in a response while the target reference is a *Parameter* inside a request operation. The parameter can have one value or a *Collection* of values. If in the previous response a full hyperlink was provided, then the target reference is the full resource path. When the source reference of the hyperlink flow provides a collection of parameter values, the request whose URI path contains the target reference parameter has to be inside a loop or has to have the multi-instance marker, as it will need to be executed for each of the parameter values provided from the source reference. In a composite conversation hyperlink flow is used to show the data flow from the call request to the request(s) sent by the server to an external server, or to show the data flow from an external server response to the response of the initial server. Thus, while in a single conversation hyperlink flow always connects a link from the response to a parameter in the request, in composite conversations a hyperlink flow can connect a parameter from a call request to a request in a different pool, as well as a link from a response to a link in a response in a different pool. As opposed to BPMN message flow between pools, which shows the time dimension in

terms of when the data is sent or received and in which order, in RESTalk the hyperlink flow between pools shows just how the data flows between pools (i.e., the provenance of the data) without any reference to when the data is actually transferred. A response to a call request is only sent once the called conversation has finished although the data might have been already available before. An example of a composite conversation is provided in Sec. 6.2.4;

10. A *RESTalk Model* is a model of the behaviour of one or more APIs which can contain a conversation between one client and one server, or multiple types of clients and one server, or multiple conversations in the case of composite conversations. As hyperlink flow can connect elements between different conversations a RESTalk Model can have zero or many hyperlink flows (the hyperlink flows inside a conversation belong to the conversation) and it can have two or more participants which can belong to lanes or pools.

Some of these elements are mandatory when creating a RESTalk diagram, while others are optional. The minimal set of elements to use for a valid diagram are: 1) start and end events, 2) at least one request with specification of the operation method and the resource path, 3) at least one response with specification of the response code. Additional information can be added in the request/response elements depending on the goal of the diagram (e.g., microservice name, header/body information, SLA information etc.).
Meta-models can be enhanced with "well-formedness rules written as OCL invariants that impose restrictions, which have to be satisfied in each well-formed instance of the abstract syntax metamodel" [13]. The Object Constraint Language (OCL) has been adopted as the formal specification language for the definition of constraints and is part of the UML standard. It adds precise semantics to visual models. There are five types of OCL expressions: 1) invariants (inv) to state the conditions that have to be satisfied when a model is instantiated; 2) initialization expressions ($init$) to state the initial values that the properties of an object must take upon creation; 3) derivation rules ($deriv$) to define how the value of derived model elements should be computed; 4) query operations (def); and 5) operation contracts (pre, $post$), i.e., set of operation preconditions and postconditions [26]. Objects in OCL are accessed using the "." symbol, while collections are accessed using the "$->$" symbol. All constraints in OCL are associated with a context, which defines the entity for which the constraint has to hold. The following OCL rules apply for creating valid RESTalk diagrams:

- //The conversation is always initiated by the client making a request

context Start Event **inv**
ConversationStart: self.targetRef.oclIsKindOf(Request)

- //A request with multiple possible responses is followed by an exclusive gateway
  context Request **inv** DifferentResponses:
  if (self.response -> size()>1) then (self.targetRef.oclIsKindOf(Gateway) and self.targetRef.type=exclusive)

- // When a response provides multiple values of a given variable the request based on that response should have a multi-instance marker or the subconversation based on that response should have a multi-instance marker
  context HyperlinkFlow **inv** Multiinstance:
  if (self.collection = true) then
  (self.targetRef.operation.belongsTo.multiinstance = true or
  self.restalkmodel.conversation -> includes(s:SubConversation|
  (s.multiinstance = true and s.flownode ->
  includes(self.targetRef.operation.belongsTo))))

- // A boundary event is attached to a request and thus it has no incoming flows and only one outgoing flow
  context BoundaryEvent **inv** AttachedEvent:
  self.sourceRef -> isEmpty() and self.targetRef -> size() = 1

- // A start event has no incoming flows and only one outgoing flow
  context StartEvent **inv** StartEventFlow:
  self.sourceRef -> isEmpty() and self.targetRef -> size() = 1

- // An end event has no outgoing flows and only one incoming flow
  context EndEvent **inv** EndEventFlow:
  self.targetRef -> isEmpty() and self.sourceRef -> size() = 1

- // A gateway has multiple incoming or outgoing flows
  context Gateway **inv** GatewayFlow:
  self.sourceRef -> size() > 1 or self.targetRef -> size() > 1

- //Each conversation should have at least one client and one server
  context Conversation **inv** Participation:
  self.participant -> exists(p1: Participant | p1.type = client) and
  exists(p2: Participant | p2.type = server)

- // A conversation where n clients talk to n servers is out of scope for RESTalk, only conversations between multiple clients and one server or multiple servers and one client are supported
  context Conversation **inv** NoNtoNConversations:
  if (self.participant->select(p|p.type=client) -> size() > 1) then
  (self.participant->select(p|p.type=server) -> size() = 1) or if
  (self.participant->select(p|p.type=server) -> size() > 1) then
  (self.participant->select(p|p.type=client) -> size() = 1)

- // A hyperlink flow can have as a target a response only if that response is related to a call request
  context Response **inv** ResponseToResponseHyperlinkFlow:
  if (self.link.targetRef -> size()>0) then self.request.oclIsTypeOf(Call Request)

- // A hyperlink flow can have as a source a request only if that request is a call request and the target is a request in another conversation
  context Hyperlink Flow **inv** RequestToRequestHyperlinkFlow:
  if (self.sourceRef.oclIsTypeOf(Parameter) -> size()=1) then
  (self.sourceRef.operation.oclIsTypeOf(Call Request) and
  self.targetRef.operation.oclIsTypeOf(Request) and
  self.sourceRef.operation.request.conversation <>
  self.targetRef.operation.request.conversation)

- // If no lanes/pools are used in the model then the participants are not named
  context Participant **inv** NamingParticipants:
  if self.belongsTo = null then self.name = null

- // When a conversation has multiple clients they should all be named
  context Participant **inv** LabelingParticipants:
  if (select(p: Participant|p.type=client) -> size() > 2) then forAll(p.name <> null)

- // When a model has participants belonging to different pools each participant should have a name
  context Participant **inv** LabelingParticipantsInPools:
  select(p: Participant|p.belongsTo=pool) -> forAll(p.name <> null)

The following OCL rules on the other hand are recommended optional best practices:

- // The related request / response elements can be united, i.e., it is recommended not to use a sequence flow between them
  context Request **inv** OneResponse:
  self.response -> size()=1 and self.targetRef -> isEmpty()

- // It is best practice to only have unique requests (method + URI pairs) in the model
  context Request **inv** UniqueRequest:
  self.allInstances() -> forAll(r1, r2|r1<>r2 implies (r1.operation.method <> r2.operation.method or r1.operation.resource.path <>r2.operation.resource.path))

- // Unless required to show more detailed responses, best practice is not to have multiple responses with the same response code related to a single request
  context Request **inv** UniqueResponses:
  self.response -> isUnique(statusCode)

4.2.2 RESTalk Graphical Representation

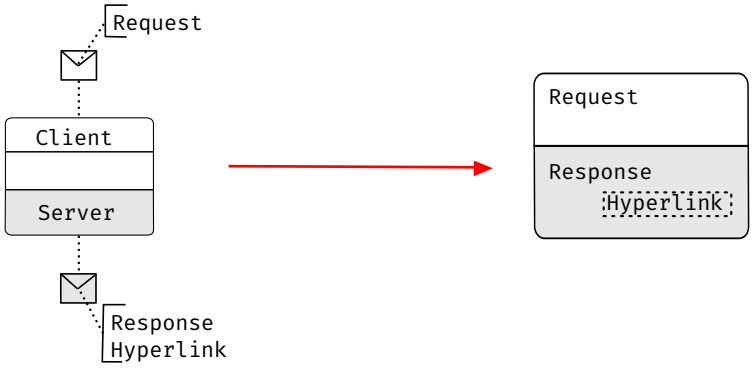
The graphical notation of a Domain Specific Language is very relevant for the comprehensibility, usability and productivity of the DSL [171]. When designing the visual notation of RESTalk we have kept in mind the design guidelines presented in [57]. Since BPMN Choreographies do have a visual concrete syntax, we have used them as a starting point for the RESTalk visual notation, which already gave us good basis for visualizing the main concepts mentioned in Sec. 4.2.1. We used a subset of the BPMN Choreographies syntax as-is (the control flow constructs such as sequence flows, gateways, events), but we also modified some elements (such as the Choreography Task) to emphasise REST specific facets. When modifying and expanding BPMN Choreographies, as advised in [57], we have grouped the main concepts into categories and created generic symbols for each category, trying to keep the visual distance in line with the semantic distance of the different concepts [142]. We have created such visual distance through shape, colour or text. In certain cases in addition to shapes we also used icons to create a reference to the represented concept (e.g., timer event, sending message activity). Most symbols are assigned to one particular concept (avoiding “symbol overload”) and most concepts can only be represented by one symbol (avoiding “redundant symbols”) [142].

From BPMN Choreography to RESTalk

Cortes-Cornax et al. [29] when evaluating the quality of BPMN Choreographies state that "the language must be powerful enough to express anything in the domain but no more". Thus, to render the subset of BPMN Choreography graphical symbols more concise when targeting the modeling of RESTful conversations, as opposed to generic message-based conversations for which the BPMN Choreographies were originally designed, in [160] we have proposed minor, but significant changes to the standard notation. We will discuss such changes, before defining the symbols for the remaining concepts from the metamodel.

Modification 1: Contrary to business processes, where it is important to highlight which participant is responsible for initiating the interaction, in a RESTful conversation the initiator is always the client, and there is no one-way interaction, as every successful request is followed by a response. The content of the messages is of a particular interest, because it defines, as a minimum, the resource and the action to be taken by the server, by defining the HTTP method. To comply with these differences and bring the visual construct closer to its meaning [142], we replace the BPMN Choreography task with an interaction (request/response) element. The BPMN Choreography task can have an optional incoming/outgoing message with a text annotation to depict the message content. The three band choreography task contains participants' names and a task name. We replace it with a two band request-response element with embedded message content (Fig. 4.4). The required content of the request-response messages is the request method, the URI, as well as the response status code, and where applicable links.

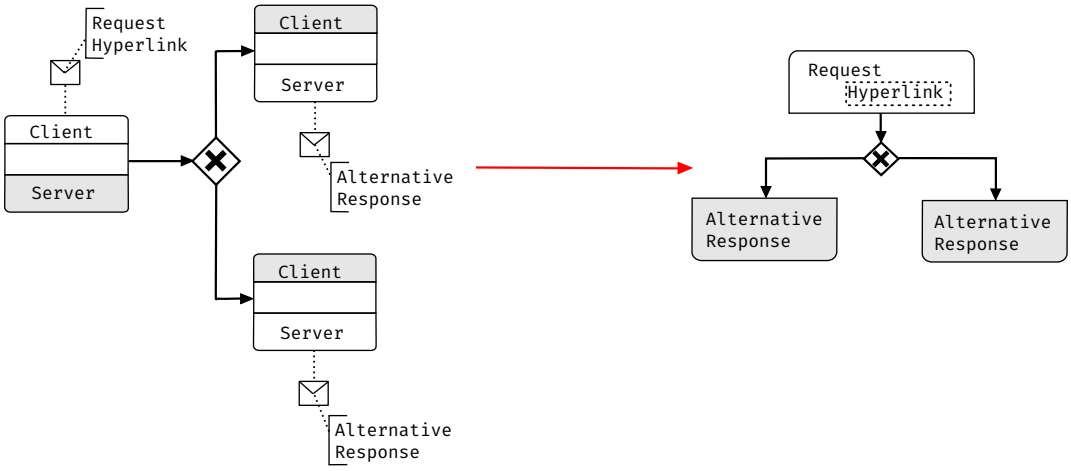
Figure 4.4. Modification 1: replacing the BPMN Choreography Task



Modification 2: Since in a RESTful interaction a request is always followed by a response, the request-response bands always go together, except when there

is path divergence due to different possible responses from the server to a given client's request. Only in this case the request is separated from the responses by a smaller size exclusive gateway to show the two or more alternative responses that can be sent by the server. As evident in Fig. 4.5 this makes RESTalk less verbose and less bulky in the visual representation.

Figure 4.5. Modification 2: allowing for different server responses

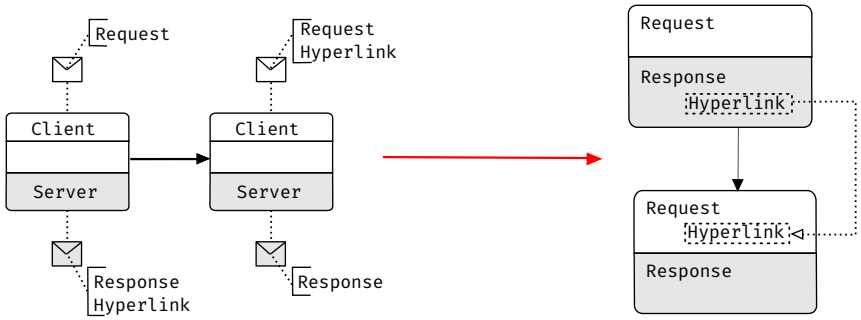


Modification 3: The *hyperlink flow* indicates how URIs are discovered from hyperlinks embedded in a preceding response to clarify how clients discover and navigate among related resources (Fig. 4.6). Adding this element is important in RESTful conversations where, due to the HATEOAS constraint in a REST compliant API, clients should not be forced to guess URIs, neither to retrieve the URIs from out-of-band knowledge [217]. Thus, the hyperlink flow makes it possible to distinguish which requests are sent by clients using hard-coded knowledge about URIs, and which are sent by dynamically discovering the URI from previous responses. Namely, if there is a client request in the conversation model (with the exception of the first request), where the URI is not extracted from a hyperlink flow, the API designer is aware that a client would need out-of-band knowledge to complete the conversation.
An exemplary RESTful conversation would look as in Fig. 4.7a, when modeled with the standard BPMN Choreography notation, and as in Fig. 4.7b when modeled with the above mentioned modifications made when designing RESTalk.

Core and Extended Constructs

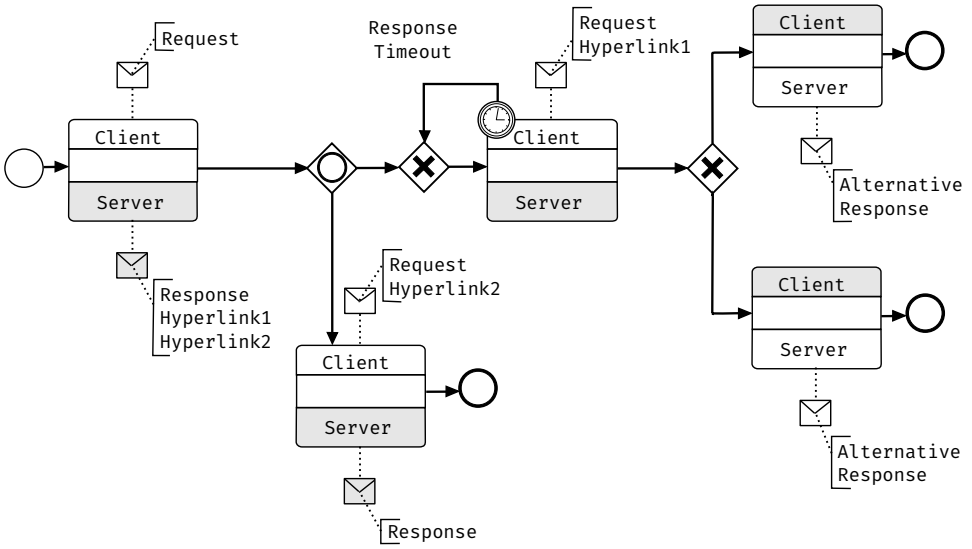
The core constructs in RESTalk are used to express simple one to one (client-server) conversations. Although the concepts they represent have been defined

Figure 4.6. Modification 3: hyperlink flow

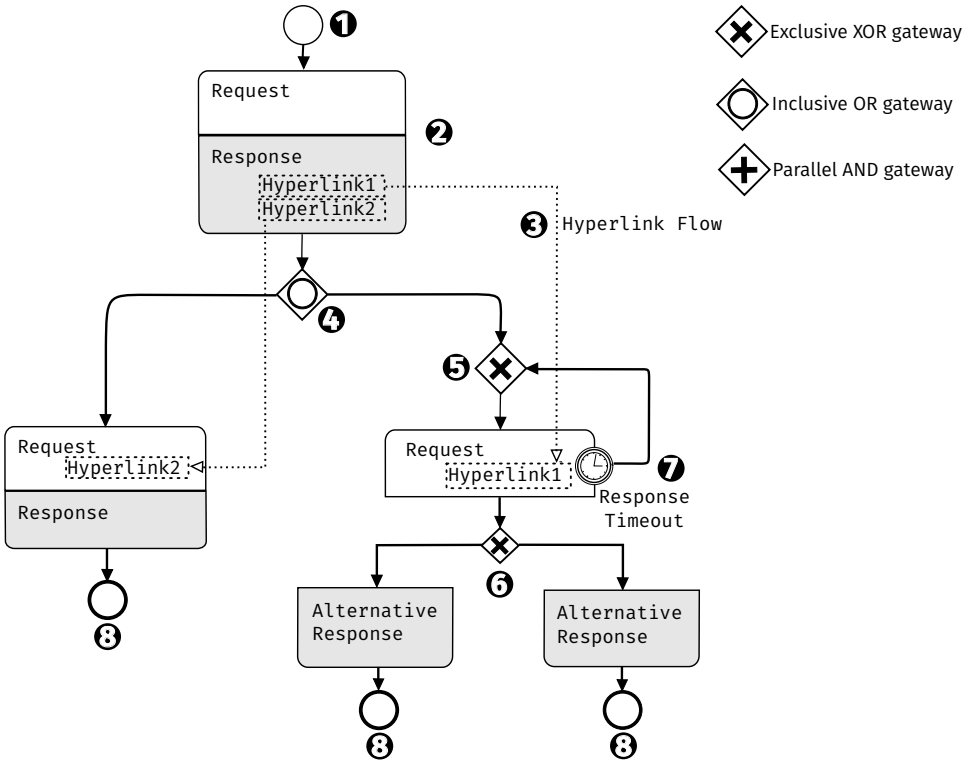


in Sec. 4.2.1, we will briefly summarise them here while making reference to the visual representation of the same (please refer to the enumerated elements in Fig. 4.7b):

1. *Start event* to mark the beginning of the conversation;
2. An *interaction element* containing the content of the request (white) and response (gray) messages;
3. *Hyperlink flow* to highlight the usage of resource identifiers discovered from previous responses;
4. *Control flow split gateways* to show path divergence due to client's decisions. As mentioned RESTalk supports three types of gateways: XOR - exclusive gateway, OR - inclusive gateway, AND - parallel gateway;
5. *Control flow merge gateways* to show path convergence after a split. The same constructs are used as for the split gateways, but the semantical meaning is to allow the conversation to continue after a certain condition is met;
6. *Exclusive split* due to different possible responses from the server;
7. *Response timeout* to model situations where it is relevant for the conversation to show that, if the server takes too long to respond, the client will resend the request or quit. Such timeouts can happen after every request, but we recommend to explicitly use the response timeout event only for non-idempotent requests;
8. *End event* to mark the end of the conversation, when the client stops sending requests after achieving its goal. Different end events (reflecting different outcomes) are possible for a given conversation.



(a) Exemplary RESTful conversation modeled using standard BPMN Choreography



(b) Exemplary RESTful conversation modeled using RESTalk with numbered notation constructs

As mentioned earlier we applied an iterative approach in the design of RESTalk. Thus, based on the results of an initial exploratory survey regarding the cognitive characteristics and the perceived usefulness of the core RESTalk (described in Sec. 6.1), by applying RESTalk on different use cases (specified in Sec. 6.2), and based on the results from an in-class experiment (described in Sec. 7.3) we have gradually extended the core RESTalk with the following constructs (see Fig. 4.8) necessary to model more complex conversations and multi-party conversations.

1. *Lanes* to show the roles of different participants in the same conversation, when viable interactions depend on such roles;

2. *Resource state change* to show client-server interactions which lead to changes in the related resource's state;

3. *Sending message activity* to show asynchronous interaction, e.g., through email notifications sent to participants;

4. *Pools* to show the roles of participants in different conversations, connected to each other through call requests;

5. *Sub-conversation* to show conceptual nesting in the main conversation;

6. *Multi-instance marker* to show multiplicity. When used in a lane it shows multiparty conversations where multiple clients of the same type interact with the server. When the marker is used in a sub-conversation it shows that the sub-conversation will be executed multiple times as part of a single instance of the parent conversation. When used in a request it shows a loop where multiple requests of the same type will be sent before continuing with the conversation (for instance, using different values of the same variable);

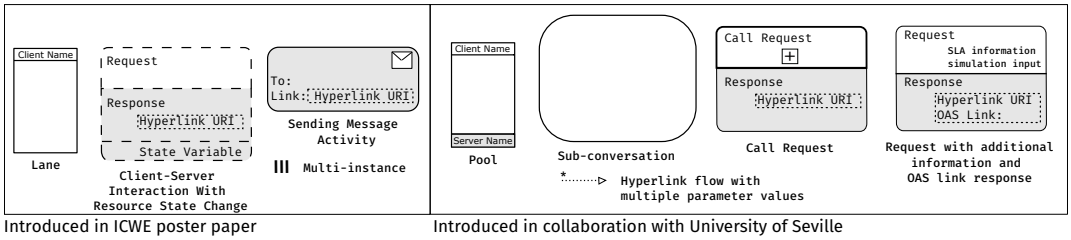
7. *Multiple parameter values* in the hyperlink flow to show a single request/response which includes a collection of values for a given parameter;

8. *Call request* to show layering in service composition where an API request calls another API and only receives a response when the other call has returned;

9. *OAS link* in the response as an addition, or an alternative, to the hyperlink in the response to facilitate pointing to parts of the Open API Specification;

10. *Additional information* referring for example to the limitations as per the SLA agreement, to requests where user input is required for simulation, as for example in case of loops or to the name of the microservice a resource belongs to.

Figure 4.8. Extended RESTalk constructs



Simplifications

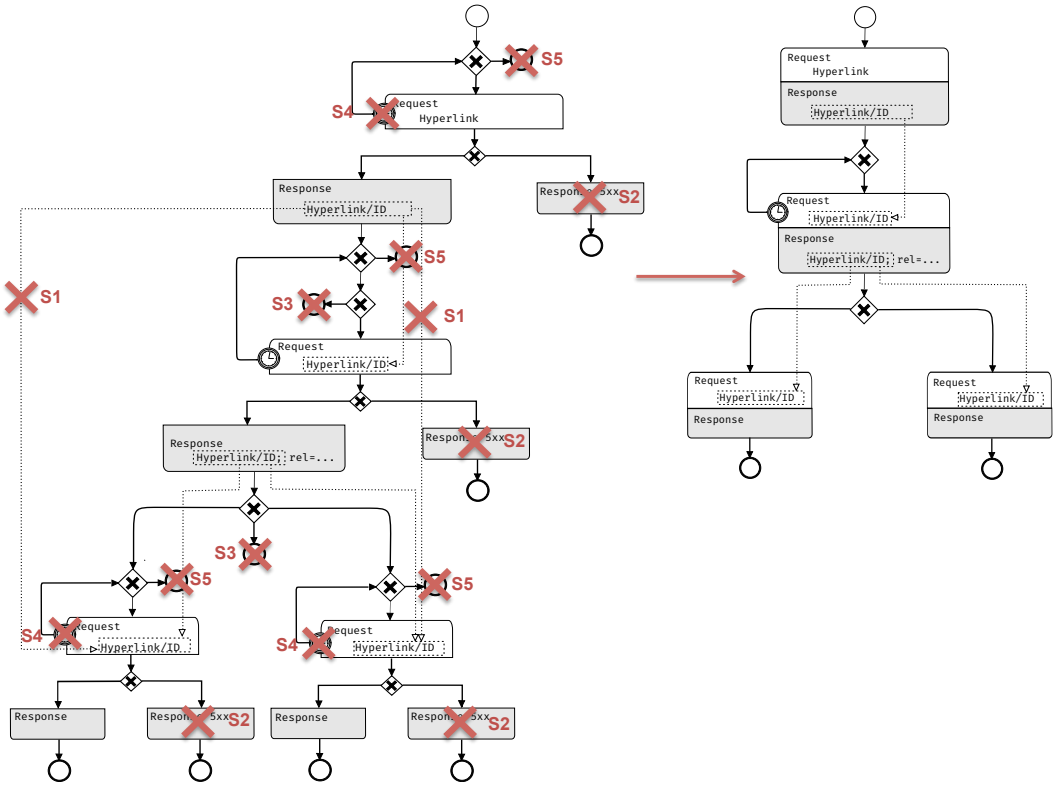
Often in high-level conceptual modeling [177, pg. 93], various assumptions and simplifications are necessary to avoid overwhelming the reader with too many visual elements. As a result, certain details are excluded from the model representation. Having in mind the characteristics of RESTful conversations, mentioned in Sec. 2.3, we introduce the following assumptions that help simplify RESTalk diagrams (the visual effect of the same is evident in Fig. 4.9):

1. While a hyperlink that has been discovered by the client can be used at any time in the future, to avoid decreased readability due to too many hyperlink flow edges, we only take into consideration the hyperlink obtained from the nearest previous response (Fig. 4.9 (S1));

2. While servers may send responses that include many different HTTP status codes, we only include the status codes which are relevant for the specific conversation. For example, 5xx status codes can occur at any time. The conversation model should only explicitly indicate how a client will need to react to such errors depending on the specific conversation domain and error semantics (Fig. 4.9 (S2));

3. While clients may decide to stop sending requests at any time, we model a path as finished (by using an end event), only if an initially intended goal has been achieved (Fig. 4.9 (S3));

Figure 4.9. RESTtalk diagram simplified following the simplification guidelines



4. While clients may choose to resend idempotent requests (GET, PUT, DELETE) an arbitrary number of times, we only model situations where the client retries sending non-idempotent request (POST, PATCH) after a *response timeout* event occurs. This is because resending idempotent requests does not affect the outcome of the request (Fig. 4.9 (S4));

5. Likewise, clients may eventually give up resending requests after a timeout. This additional branch and end event is not typically shown in optimistic models representing how clients deal with temporary failures (Fig. 4.9 (S5));

6. We assume that the cache is empty by default and thus responses are always served from the server, not the cache. Nonetheless, if cacheability of the responses is important for understanding the conversation, additional information relative to the cache can be added in the request;

7. Fundamental general resource states are implicitly stated in the response status code. Thus, we recommend states to be visually modelled only if

they are important for the application and impact the interaction between the participants;

8. Proxy layers in the communication act as intermediary resending the request/response. Thus, we only recommend modelling them if they are in a form of a gateway implementing certain business logic.

Style Guidelines

Style is frequently influenced by the personal preferences of the modeler or by the complexity of the reality that the model needs to describe. Thus, the implementation of the following recommendations is subject to modeler's choice in addition to the best practice constraints mentioned in Sec. 4.2.1.
To accommodate the request-response content presented within the interaction element, we recommend a vertical flow for the diagram layout, as opposed to the horizontal direction usually used in standard BPMN Choreographies. This should enhance the readability of the diagram [163] as the control flow can be followed from top to bottom: with a starting event leading directly to client's request, followed by the corresponding server's response leading directly to the next request or an end event. The events that the modeler can use are not limited to the default none or to the timer event type. BPMN Choreographies offer a plethora of event types, and depending on the needs of the conversation, any of them can be used within RESTalk. For instance, events may be used to represent the out-of-band discovery of links, e.g., when they are extracted from an e-mail message. As opposed to BPMN, where best practices advise documentation of the decisions in exclusive and inclusive gateways, in RESTalk the modeler can abstract from describing how clients make decisions on which path to follow, if such details are not entirely relevant to the conversation. Given the scope of RESTalk described in Sec. 4.1 and based on the use-cases on which we have applied RESTalk (see Sec. 6.2), we have found the three types of gateways we used (exclusive, inclusive, and parallel) sufficient to express the behaviour of a REST API. Other types of gateways, such as the event-based gateway or complex gateways, could be used in RESTalk models if the semantics of the API behaviour requires it. Note that there is a slight difference in the semantics of event-based gateways in BPMN Choreographies compared to BPMN orchestration diagrams. In BPMN orchestration diagrams event-based gateways represent a race between possible events (external data) which determines the path to be taken. In BPMN Choreographies event-based gateways are used to represent situations in which the participant initiating the interaction takes an internal decision and the other

participants have to react to that decision (external to them), so the decision data is not known to the recipient of the message beforehand. In RESTful conversations the decisions are taken by the client, but the server reacts to such decisions based on the state of the requested resource, as it is the server who provides and stores the data about the resource. As such the use of event-based gateways is more limited.
The request method, the URI, the response status code and links are mandatory elements of the request-response interactions. However, as mentioned in the extended RESTalk constructs, RESTalk does not limit the content of a message to only the above mentioned elements. Depending on the notation usage, whiteboard or drawing tool, any headers or content details considered necessary can be added by the modeler.
In RESTalk, the sample values of URIs found in the hyperlink flow are meant as placeholders. They do not necessarily have to reflect the actual structure of the URI, since as per the HATEOS principle the server is free to create any URI to be sent to the client embedded in a response. Sometimes a concrete URI is not expressive enough, for example, for providing the client with a link for searching, where the search term typically is not known in advance by the server. In such cases, an HTML form [84] or a URI template [74] is provided by the server, so that the client can mint the concrete URI by replacing the parameter(s) in the given template. This requires the client to know the semantics of all parameters. Such URI templates can also be contained in the hyperlink flow.
"Color can be used as an additional discriminator. This makes sense, if a concept is very similar to others so that it would be difficult and/or misleading to define a separate symbol for representing it." [171]. Using different symbols for different types of HTTP requests (GET, POST, PUT, DELETE), can be misleading and add additional cognitive workload for RESTalk users. Thus, in order to help the RESTalk diagram readers, the modeler can use colours to mark the type of HTTP request. We suggest to use the colours already used in Postman, a widely used platform for API development and testing[1], i.e., green for GET, yellow for POST, blue for PUT, and red for DELETE. Color as a secondary notion can also be used to mark resources or nesting of resources, depending on the needs of the user. More details will be discussed in Sec. 8.3.2.

[1] `https://www.postman.com`

4.2.3 RESTalk Textual Representation

Researchers have been working on textual support for general modeling languages, in order to facilitate the adoption of those languages, as developers seem to be more inclined to use textual editors as opposed to graphical editors. The reason behind that is the long tradition of using textual general purpose programming languages, which reduces the learning curve [22] for the textual DSLs. Textual DSLs are also considered more scalable, more searchable and easier to collaborate on, as text is supported by version control systems [75] and real-time collaborative editors. In [192] the authors perform an empirical research based on 28 developers in order to study the impact of structured textual vs. graphical representation on users' efficiency in comprehending requirements. They have discovered that, although accuracy does not seem to be affected, the developers who used the textual representation were faster than the ones that used the graphical representation, but nonetheless majority of the participants preferred using the graphical representation. In areas such as business process modelling, studies have shown that starting with textual input (e.g., activity tables, written use-case scenarios) and then abstracting the information using visual models improves process understanding for people who are not process modelling experts [36; 50; 156]. Furthermore, one of the respondents in our exploratory survey discussed in Sec. 6.1 stated that a "prerequisite for easy usage is that the graphical notation can be derived from a simple textual notation". Given such results, in addition to the graphical representation of RESTalk, we have decided to also propose a textual representation so that targeted users can select which representation to use. We have first proposed a textual RESTalk syntax in [91] and expanded it later.

As we have seen in the survey of editors that use textual DSL for visual modeling described in Sec. 3.3.3, most existing textual DSLs use programming language syntax whenever possible, requiring explicit modeling of all visual elements in the model. This approach requires the modeler to create a mental abstraction of the reality (s)he is describing, very similar to the abstraction required from a modeler who needs to use a graphical editor to model a RESTful conversation with RESTalk. In the textual representation for RESTalk we have decided to take a different approach which would require less effort to do the abstraction in order to address the needs of people who struggle with, or are not fond of, abstraction and thus prefer to think in execution traces which are more similar to examples of possible interactions or user-stories. User-stories are a common practice in software engineering [28]. The goal of classical user-stories is for the software user to describe the desired functionalities of the software to be developed. Clas-

sical user-stories have a predefined content structure to facilitate understanding, but are written in natural language. In the case of RESTalk, the purpose of the user-stories is to describe how the functionality of the system (i.e., the goal of the conversation) is mapped to the sequences of required interactions to achieve that goal (i.e., the conversation instance). To that end we have decided to design the textual DSL as constrained natural language based on lists of required interactions. We chose such approach as it requires little or no memory of RESTalk specific visual constructs and allows for more intuitive story telling technique for creating the model.

Our goal is to keep the textual DSL for the **core RESTalk** elements rather *intuitive* so that RESTalk models can be sketched quickly without the need of complex algorithms to generate the visual model, while requiring some *textual modelling* effort for the use of the **extended RESTalk** elements. The textual DSL for RESTalk is designed with a tool support in mind that leverages process mining algorithms [208] to reconstruct the conversation flow. This is a novel modelling approach, which we have presented in [88; 92] for business processes modelled with the BPMN standard modelling language [101] and have decided to also apply it for RESTalk. A typical HTTP log would contain substantial amount of traces with each traces carrying information about the IP address of the client, the URI the call is addressed to, the HTTP head, a timestamp of when the call was made [178]. The HTTP head does contain the HTTP Method as well as the response status code. The textual DSL for RESTalk can benefit from log-like structures as API developers and API designers are likely to be familiar with them, but it can abstract from the timestamp and the IP address information required for the mining algorithm by pre-processing the logs to add such information algorithmically.

The design of such log-like textual DSL requires an appropriate trade-off between the use of textual modelling and mining to aid the textual modelling. That said, an important decision for the support of the **RESTalk core** is the definition of the minimal necessary user input in order to maximize what can be expressed in the output. The desired output in our case is a draft RESTful conversation model, for which the required minimal input are the request-response interactions and the sequence in which they have to be completed. Namely, to get the most simple core RESTalk diagram which only shows a sequence of interactions, with no control flow divergence, and no hyperlink flow it is sufficient for the user to write each client-server interaction (request - response pair) as a new log entry using a new line, in the order in which the interactions are to be completed. Each line has to contain the following minimal required elements separated by an empty space: 1) the HTTP method, 2) the resource URI starting with a `/` and the 3) numerical

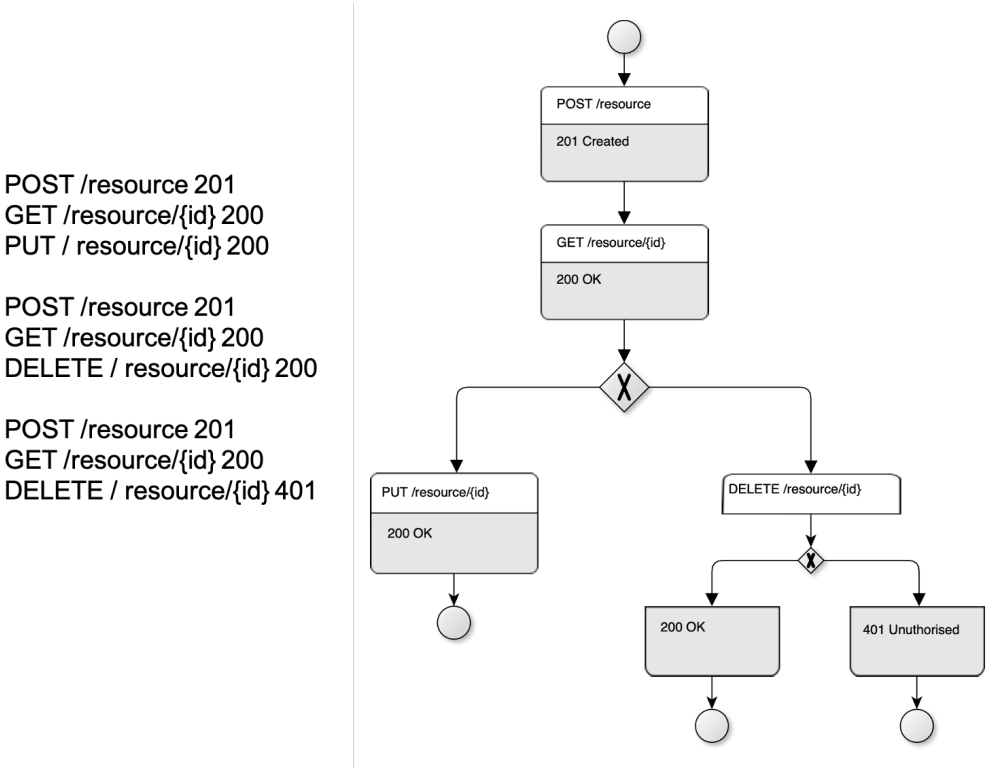
response status code (per design the standard textual description of the numerical status code should be added by the tool). No time-stamp is required as the sequential order is deduced from the order in which the interactions are written. As evident modelling a sequential flow is simple both in textual and graphical syntax. However, unlike the graphical syntax, the textual one is characterized by its mono-dimensional structure [70]. This constraint makes it an adequate choice for representing sequential conversations, but makes it challenging to use plain text to represent control flow graphs of arbitrary structures like the ones which can be visualized in RESTalk. Inspired by our work in the BPMN field [90], we address this challenge by using a process mining algorithm [208] to reconstruct a model of the conversation control flow graph from a set of written sequential interaction traces, which can be easily stated in plain text separated by an empty line.

Fig. 4.10 shows a simple example of the use of the core textual RESTalk DSL to model a simple conversation by using three interaction traces. As can be noticed in the figure the DSL is designed such that it does not require the exclusive gateway to be explicitly stated by the user, instead it is intended to be deduced by a mining algorithm. For instance, if two different requests never appear together in the same conversation trace, the miner will deduce that an exclusive gateway needs to be visualized before reaching these requests. The same would be true also for exclusive gateway due to different responses by the server. In this case if the same request is followed by different response codes in different instances, the mining algorithm would need to add the exclusive gateway between the request and the different responses that can be provided by the server to that request.

To get the full conversation model, a complete interactions instance trace is required for each possible path that can be taken in the conversation. The drawback of this requirement is that stating all the possible conversation paths becomes repetitive when there are many alternative paths which have many shared interactions. To address this issue, we introduce the "..." symbol which is essentially a placeholder for missing parts of the trace which are defined elsewhere. The precise semantics of the "..." symbol depends on its position relative to the start or end of the conversation instance trace, i.e., relative to the empty line. Following are the possible **types of conversation instance traces** and their semantics:

- Conversation instance trace that ***does not contain*** the "..." symbol: a sequence of interactions which shows a complete possible execution path of the conversation from the start to an end;
- Conversation instance trace that ***starts with*** the "..." symbol: the start of the

Figure 4.10. Execution logs like textual DSL for core RESTalk elements



conversation instance has been defined in the trace of other conversation instances. This fragment of a conversation instance trace states a sequence of interactions that leads to an end of the conversation;

- Conversation instance trace that ***ends with*** the "..." symbol: the end of the conversation instance has been defined in the trace of other conversation instances. This fragment of a conversation instance trace states a sequence of interactions that follows from the start of the conversation;

- Conversation instance trace that ***contains*** the "..." symbol: there is a part of the conversation instance that has been defined in the trace of other conversation instances. This fragment of a conversation instance trace states a sequence of interactions that starts and ends the conversation, but skips the middle of the conversation instance trace;

- Conversation instance trace that ***starts and ends with*** the "..." symbol: this type of trace is used as a fragment to fill in the placeholders in other traces in order to complete them. It does not correspond to a separate

conversation instance.

The goal of the RESTalk textual DSL is to support an iterative model refinement process, where the initial model is obtained quickly from its constrained natural language description and further fine-grained in a visual editor or by using textual modelling. Thus, basic type annotation keywords would need to be learned only if it becomes necessary to classify model elements during a second refinement step. The most important design decisions we took to support such goal involve:

- annotating the name of the client followed by "`:`", to speed up annotating interactions or events with their client **roles**. The role is applied to all interactions following the annotation until another role is declared. The roles are automatically mapped to lanes depending on the presence or absence of handovers or pools if client roles are matched to server roles. Multi-instance marker on a lane/pool is added when "`(s)`" is added at the end of the participant's name.

- **events** are distinguished from interactions because they are entered in round brackets referring to their round visual shape "`( )`". The mining algorithm should determine automatically whether they are *start*, *intermediate*, or *end* events, depending on whether there are stated interactions preceding/following the event in question. Boundary events are stated in double round brackets "`(( ))`" in the same line with the request they should be attached to.

- interactions and events can be annotated by prefixing them with a **type**. This addresses the requirement of modelers who need to refine their model after the initial sketch. Stating event types (e.g., timer, error) and interaction types (e.g., message activity, call request, subconversation) requires the use of keywords, which attempt to match how those constructs are named or used.

- **exclusive split gateways** can be annotated with a label, specified as a question (i.e., a line ending with "`?`)" in the text. As exclusive gateways denote decisions, the line after the question represents the chosen alternative and becomes the expression associated with the outgoing conditional flow of the gateway.

- we intentionally decided not to mine **parallel and inclusive** gateways as it would require manually entering multiple traces including different permutations of the same set of interactions, which early adopters considered

too much effort. Instead, parallel interactions are simply declared on the same line by separating them with "|", while an inclusive split in the control flow is marked by separating the interactions with "||" as after all an inclusive gateway split can act as a parallel split if all of its conditions are evaluated to true.

- the **hyperlink flow** is deduced from the matching hyperlink names in the response and the subsequent request, or from the matching link parameters.
- the **state** of the resource should be included in "< >" after the response code or any potential response links.
- **additional information** can be included in the request annotation. Such information has to be in a string format or included in quotation marks and has to be placed after the URI in the request.

The main design challenge for the textual RESTalk DSL consists of the trade-off between the usability, the learnability, and the expressiveness of the language. Thus, the textual DSL attempts to draw the line between the usefulness of mining algorithms to infer RESTalk constructs implicitly embedded into the textual description vs. explicitly stating a construct, as evident in Tab. 4.1. Our goal is, whenever possible, to reduce the cognitive effort of the users [235] by not requiring them to explicitly state RESTalk constructs. For example, as hyperlink flows can be inferred by detecting a matching hyperlink name between a response and a subsequent request, we do not require the user to explicitly include such flows in the textual description.
In Fig. 4.11 we show a more complex example using textual modelling on top of the mining algorithm to also show lanes and the use of inclusive gateway. We also use the above mentioned placeholder symbol "..." to avoid repeating already stated sequences of interactions. More examples using different constructs of the textual DSL for RESTalk can be found in Sec. 6.2. Different mining algorithms can be used on top of the above described textual DSL to build the tooling support for the DSL. They can be built from scratch for the RESTalk purposes or they can be adapted from other fields, such as business process mining, where substantial work has been done on the matter [208].

RESTalk Textual DSL EBNF Specification

The textual DSL concrete syntax is meant to be closer to natural language than to common textual programming languages, which are based on control

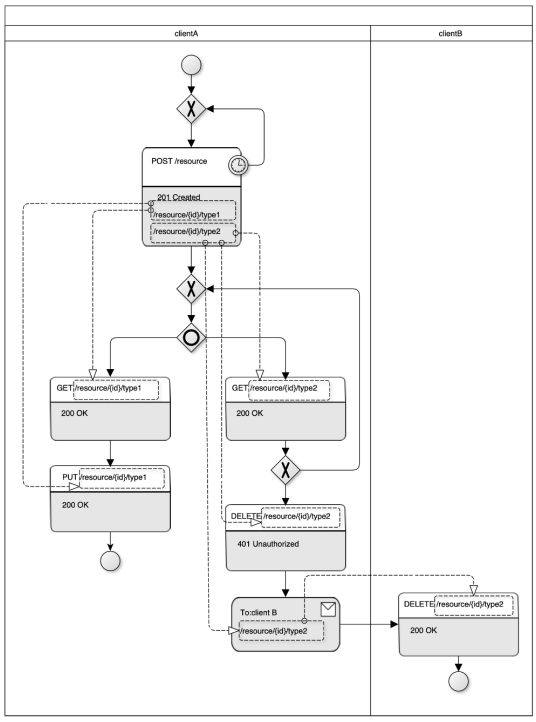
Table 4.1. RESTalk Constructs supported by the textual RESTalk DSL

RESTalk Construct	**Derived from mining**	**Explicitly modelled**	**DSL construct**
request		✓	state each request in a new line separating the HTTP method from the URI with an empty space
response		✓	state the response code numerically after the request separated with an empty space
links		✓	use "`/`" at the beginning of a URI to indicate a hyperlink, use "`{}`" to indicate a parameter inside the URI or in the response, use the "`OAS`" keyword followed by the OAS reference id to model references to OAS
events	✓		use "`( )`" to indicate an event. Whether it is a start, intermediate or end event is automatically inferred.
event type		✓	use the "`timer`", "`error`", "`notify`", "`publish`", "`terminate`" keywords
boundary event	✓		include the event name in "`(( ))`" stating the event type keyword followed by the request the event should be attached to in the same line and the exceptional behaviour in a new line
different responses control flow divergence	✓		use the same method+URI pair followed by a different response code
exclusive gateway split/merge	✓		repeat the last common request-response interaction before the split
labeled exclusive gateway split		✓	use "`?`" after the name of the label
loops	✓		repeat the interactions in the loop
sequence flow	✓		state interactions in the order in which they happen
conditional flow	✓		state the condition following the gateway label
hyperlink flow	✓		use matching names for the response hyperlink and the request URI or use matching parameter names
hyperlink flow with multiple parameters		✓	use "`[ ]`" to indicate multiple parameters in the request/response
parallel gateway		✓	separate the parallel interactions with "`|`"
inclusive gateway		✓	separate the inclusive interactions with "`||`"
lanes		✓	state the name of the participant followed by ":" and an interaction
pools	✓		lanes become pools in case server roles are stated at the end of the textual input after the "`SERVERS`" keyword followed by matching client name separated from the server name by a "-"
call request		✓	use the "`call`" keyword before the interaction and optionally indent the called conversation to show the nesting
message activity		✓	use the "`send`" keyword before the name of the email recipient
state		✓	use the "`< >`" around the name of the resource state after the response hyperlink
additional information		✓	state any additional information (e.g., microservice name, SLA constraint) following an empty space after the request URI
multi-instance marker		✓	use "`(s)`" after the name of the participant to add a multi-instance marker in a lane/pool. Use the "`forEach`" keyword as a placeholder for a multi-instance subconversation.
subconversation		✓	only multi-instance subconversations are supported
text annotation		✓	use "`//`" before the interaction to which the text annotation is to be attached
comment		✓	use "`///`" to add a comment in the text

structures, blocks, and the use of keywords. A formal syntax of a textual language is useful for naming the various syntactic parts of the language, showing the valid sequences of symbols and showing the structure of any sentence in the language. To provide a formal description of the RESTalk textual DSL we have decided to use EBNF, an ISO standard language [195], as a syntactic

Figure 4.11. Execution logs like textual DSL for extended RESTalk elements

```
clientA: POST /resource 201 /resource/{id}/type1, /resource/{id}/type2
GET /resource/{id}/type1 200 || GET /resource/{id}/type2 200
...

...
GET /resource/{id}/type1 200
PUT / resource/{id}/type1 200

...
GET /resource/{id}/type2 200
GET /resource/{id}/type1 200
...

...
GET /resource/{id}/type2 200
DELETE / resource/{id}/type2 401
send clientB /resource/{id}/type2
clientB: DELETE / resource/{id}/type2 200

((timer)) POST /resource
POST /resource
```



metalanguage. The main building blocks in EBNF are terminals (atomic parts of the language which cannot be split into smaller components) and non-terminals (a combination of terminals and non-terminals which create syntactic parts in the language). A terminal can be a quoted literal, a regular expression, or a name referring to a terminal definition. Examples of terminals are variable identifiers, keywords, literals, separators, whitespaces, comments. Terminals are defined using string constants or regular expressions. The grammar is defined by essentially defining production rules, i.e., rules that define how a valid non-terminal can be composed. Each rule is composed of a left-hand side which is the name of the rule, and a right-hand side describing that name, separated by =. Alternative items are separated by a stroke (|), an optional item is enclosed between square-brackets ([ ]) and repeatable items are enclosed between curly-brackets ({}) meaning that the item can be repeated zero or more times. In order to use these symbols in the DSL we are defining, we include them in single quotation marks (e.g., '['). A comma (,) is used to concatenate items, and semi-colon (;) is used to show the end of a production rule. The RESTalk textual DSL's context-free grammar is expressed in the following EBNF specification:

(* Assumptions: A character can be a letter, digit, or a symbol, while a string is comprised of multiple characters of type letter. EOL stands for End Of Line.*)

label = character, { character } ;

role = string, ':' | string, '(s)', ':';

method = 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'CONNECT' | 'OPTIONS' | 'TRACE' ;

status-code = [1-5], [0-9], [0-9] ;

(* showing how single parameters vs list of parameters should be annotated*)

parameter = '{', label, '}' | '[', label, ']' ;

uri = '/', label, {[parameter], [label]};

oas = 'OAS', [digit, {digit}], ' ', label ;

state = '<', label, '>' ;

additional-info = string, { string } | '"', label, '"'

request = [role], [' '], ['call'], ' ', method, ' ', uri, ' ', [additional-info] ;

response = status-code, ' ', ([uri, {',', ' ', uri}] | [parameter, {',', ' ', parameter}] | [oas]), ' ', [state], [' '], [additional-info] ;

interaction = request, ' ', response ;

dots = '...' ;

parallel = interaction, '|', interaction, {'|' interaction} ;

inclusive = interaction, '||', interaction, {'||' interaction} ;

event-type = 'start' | 'finish' | 'timer' | 'publish' | 'notify' | 'error' | 'escalate' | 'terminate' ;

event = '(', [event-type], [' '], label, ')';

(* showing the XOR label and the condition after *)

XOR-label = label, '?', EOL, label ;

message-activity = 'send', ' ', label, ' ', uri ;

boundary-event = '(', event, ')', ' ', request, EOL, request ;

subconversation = 'forEach', ' ', parameter, EOL, conversation ;

annotation = '//', label ;

comment = '///', label ;

empty-line = ' ', EOL ;

line = dots | interaction | parallel | inclusive | event | XOR-label
| message-activity | boundary-event | subconversation
| annotation | comment, EOL ;

instance = line, { line }, empty-line ;

conversation = instance, { instance } ;

servers = 'SERVERS', EOL, string, ' ', '-', ' ', string ;

compositeConversation
= conversation,{ conversation }, empty-line, servers ;

Usability vs. Expressiveness Trade-off

The trade-off between usability and expressiveness is present in different areas [59; 237]. The fundamental reason for this trade-off is the fact that greater expressiveness requires more language constructs, which hinders the usability as the users need more time and effort to learn the language. In existing textual modeling languages the user typically needs to write text which mimics the graphical constructs of the modelling language (e.g., "->" to denote an edge in plantBPMN) or use the terminology of the modelling language (e.g., pool, task etc. in plantBPMN, fork in PlantUML) in order to obtain the visual model. This requires the user to have prior knowledge of the visual modelling language. Usability, and fast learnability as part of it, can be important when targeting users who might not be frequently exposed to the visual modelling language in their daily job. Thus, although we did try to provide almost full support of the RESTalk constructs (subconversations which do not show a multi-instance execution are not supported), using the textual DSL for some very complex diagrams may become verbose and loose its benefits as it would require memorising the less frequently used textual DSL constructs. Thus, we envision the textual DSL being

used mostly for quick sketching of simple conversations as it takes the form of notes taking during a workshop discussion or brain-storming sessions. Using and implementing its full expressiveness would be more time-intensive and would require a higher learning curve.

Potential Improvement of Modeling Efficiency

Wasana et al. [15] have identified the *modelling methodology* and the *modelling tool* as two factors impacting the success of a process modelling project, but this can also hold for modelling projects in general. While the methodology refers to the modelling approach being followed (e.g., how is the requirements and information gathering phase performed), the modelling tool refers to the software being used for the design of the models. Each of these factors has its related costs in terms of efficiency and cognitive load.
When modeling conversations, we can differentiate between the cognitive load for 1) identifying and naming operations and mapping them to possible responses, and 2) identifying the correct topology of the control flow graph. The intrinsic cognitive load of people depends on their prior knowledge and the complexity of the task that they need to perform [202]. It has been shown that, when it comes to understanding visual models, readers first identify smaller submodels and later connect everything together [77]. Our textual DSL design takes advantage of this by allowing the user to specify sub-models in form of single conversation instances. These are then assembled by the mining algorithm to build the end-to-end conversation model. Similar approach, called Programming by Example, has also been discussed for decades in the programming domain. In Programming by Example the user should "work through an algorithm on a number of examples and then the system tries to infer the general program structure" [144]. So as in our case, it is the system who deals with the abstraction while the user just describes examples, or what we called earlier, user stories. We expect that the proposed approach of textual modelling aided by a mining algorithm should decrease the cognitive load of the designers thanks to the use of a mining algorithm for reconstructing the control flow graph, which is an especially complex task in larger conversation models. There are various techniques for measuring the cognitive load, such as self-reported scales about the mental effort, or the difficulty of the tasks, as well as through response time [42], or eye movement tracking and pupillary response [24]. Due to time constraints such experiments for the effectiveness of the RESTalk textual DSL in decreasing the cognitive load have remained as future work.
The type of modelling tool on the other hand, in addition to the cognitive load,

can also impact the time efficiency of the modelling task per se. In a graphical editor a modeler would need to select the correct constructs from the available palette, place them in the modeling space, frequently using the drag&drop functionality, connect them in the correct order, and type the content of the request/response interactions. When using an editor for a textual DSL the user still needs to type the content of the request/response interactions as when working with the graphical editor. However, there is no drag&drop involved, thus there is no repositioning of the cursor within the interaction shapes as all the request/response pairs are written on different lines of the same text editor. In other words, for equally experienced users we expect our textual entry to be more efficient than drawing graphical models, as argued in [76; 60]. Of course with no empirical evidence this remains just our expectation, thus as future work controlled experiments can be conducted where one group is asked to use RESTalk's textual DSL and another group is asked to use a RESTalk's graphical editor to construct the same model.

4.3 Chapter Summary

In this chapter we defined the scope of RESTalk, the three types of RESTful conversations it is meant to support, as well as the possible use-cases of its targeted users: API designers, API developers and API client developers. Then we set up the requirements of the language and discussed how they have been implemented in the design of the same. We discussed the important entities in a RESTful conversation and their interconnections through the provided RESTalk meta-model.

Both the graphical and the textual representation of RESTalk are discussed in this chapter. The graphical representation draws its basis from the BPMN Choreography diagram elements, adopting some elements as-is, modifying the choreography task construct, and adding additional constructs to expand its expressiveness. Iterative approach with frequent feedback cycles has been used in the design of the language. We distinguish between core RESTalk and extended RESTalk to facilitate its learnability depending on the modeler's needs and domain expertise.

The textual representation takes an innovative approach and uses a log-like form in order to leverage on mining algorithms to generate the control flow and graph layout. The greatest challenge in its design was to find the sweet spot between the use of textual modelling and mining as a modelling aid in order to maximise the effectiveness and efficiency of the DSL user. Our hypothesis is that this novel

approach of textual modelling facilitated by the use of a mining algorithm can decrease the cognitive load of the modeller and speed up the creation of the visual models, but additional evaluation is needed to test such hypothesis.

Chapter 5

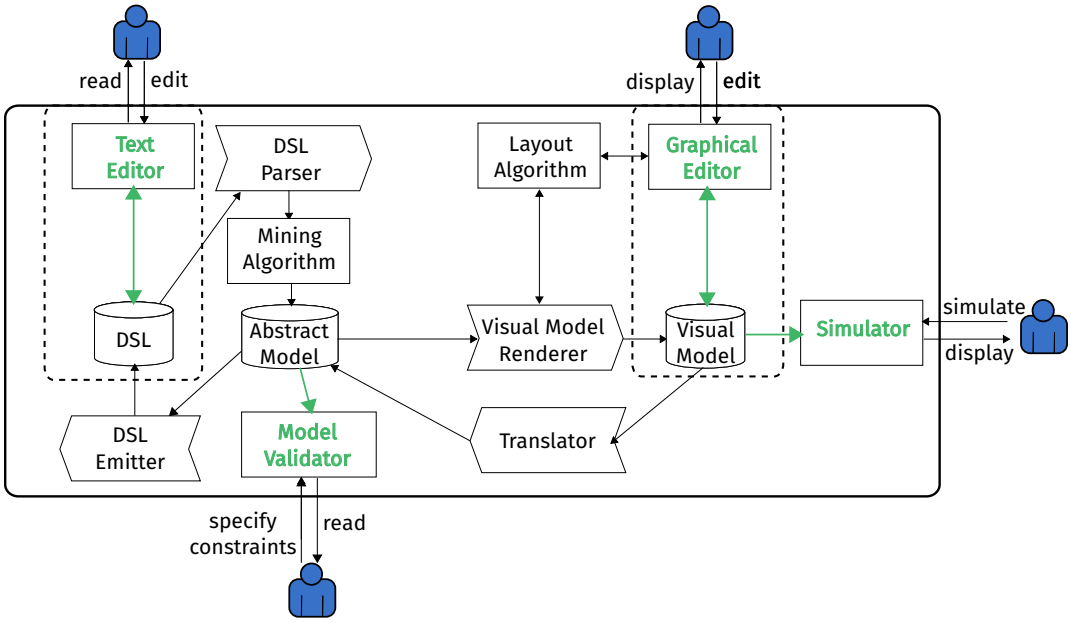
RESTalk Tooling

Although a visual DSL can be used for whiteboard sketching during discussions (as per a decade old survey [188] agile developers most frequently use whiteboard sketching for modelling - 98% of the respondents), providing a tool support for a language increases its potential for a wider community adoption. In our exploratory survey, described in Sec. 6.1, although some respondents stated that simple diagrams can be drawn manually or with existing drawing tools, 69% of the respondents have expressed preference for a modeling tool support for RESTalk. Furthermore, one respondent proposed to develop a tool for automatic extraction of the diagrams from code, while another one conditioned the RESTalk's acceptance to the availability of a specific tool: "If I could simply generate it from a textual description of the graphs, I would use it". Such feedback was taken into consideration when deliberating on the possible tool support for RESTalk. In this chapter we describe an ecosystem we have envisioned to be built around RESTalk in Sec. 5.1. Given the time restrictions not all of the envisioned tools have been implemented, but we did manage to implement some proof of concept tools. For instance, in Sec. 5.2 we discuss the existing support in terms of modelling editors for RESTalk, while in Sec. 5.3 we discuss the existing support when the Code First approach is used.

5.1 RESTalk Envisioned Ecosystem

As mentioned in Sec. 4.1.1, depending on the applied methodology, RESTalk diagrams can be created before developing the API (Design First approach) or after developing it (Code First approach). Different approaches would require different tool support.
In the **Design First approach** the diagram is created by the API designer/de-

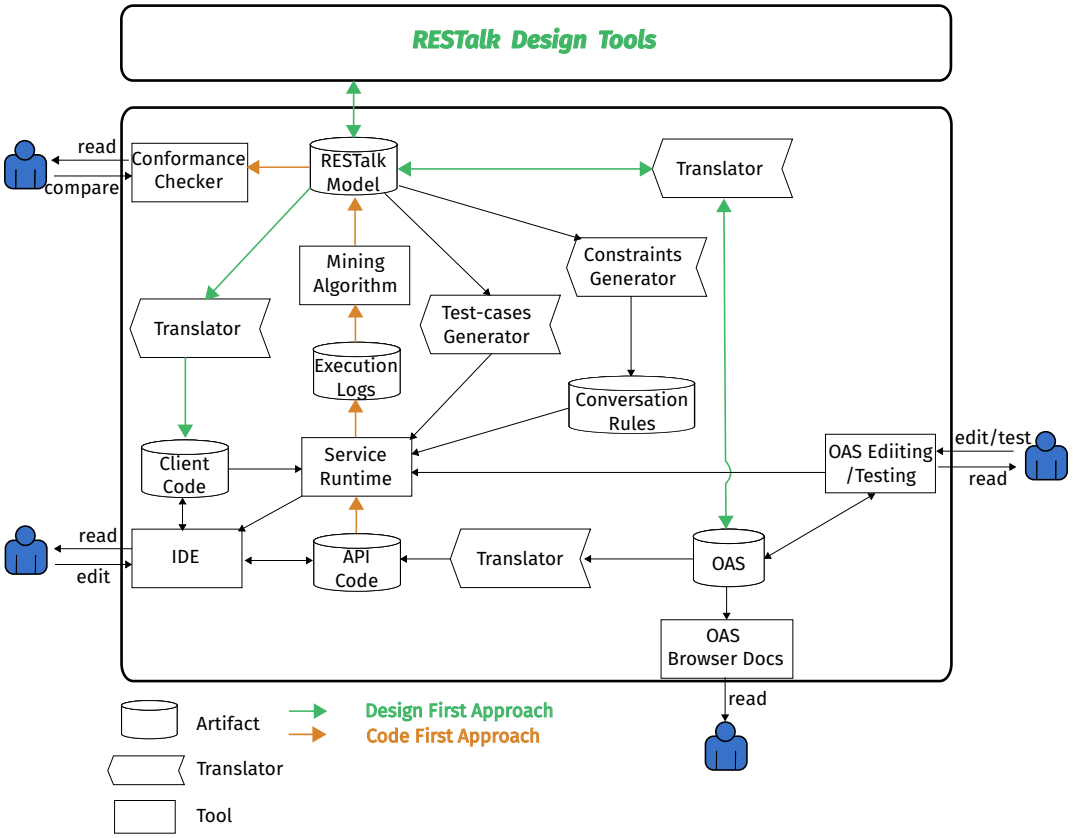
Figure 5.1. Architecture of envisioned RESTalk modelling and simulation tools



veloper using model editing tools, which depending on the preferred language representation can be either a **graphical editor** or a **textual editor**, or as mentioned in Sec. 3.3.2 ideally it would be a tool of Cluster 4b maturity which supports two-way synchronization between the textual and the visual model where all visual elements can be edited in both editors. Given the intentional design of the RESTalk textual representation in a log-like interaction statements, a mining algorithm ought to be used on the parsed textual DSL to create the abstract model as shown in Fig. 5.1. The abstract model is then used by the visual model renderer to create the visual model. A layout algorithm would be used to generate the graph layout, but the user would be able to edit the layout to his/her pleasing. Ideally collaborative editing environment would be used [58]. **Model validator** would be in place to control the validity of the created models against RESTalk's metamodel and constraints, or even against logical correctness properties specified by the designer of the concrete RESTalk diagram [194]. It would also ensure that the model does not contain any deadlocks or any unreachable interactions. Another tool that can be of value when creating RESTalk models is the conversation **simulator** which can be used to validate whether the designed model reflects the desired behaviour. The simulation can also support time and cost estimates given the constraints per user/per request stated in the service-level agreement [64]. The RESTalk design support tool could provide multi-level

view of the RESTful conversation, expanding on sub-conversations when desired, or expanding to view additional details regarding the request header or the response content, or depending on the use case, also details about Service Level Agreement (SLA) limitations, or simulation input/results can be expanded. This would ensure that the readability of the diagram is not hindered by the amount of information it needs to include. Another alternative to expanding such details in the diagram, would be to provide direct links from the interaction constructs in the diagram to the relative structural documentation of the API or to the SLA documentation.
In addition to the DSL specific tools to be built around RESTalk to support the model design shown in Fig. 5.1, we also envision integration with other existing tools to build an ecosystem around RESTalk as shown in Fig. 5.2.

Figure 5.2. Architecture of envisioned ecosystem built around RESTalk



We envision RESTalk as complementary to, and not a substitute of, the Open API Specification described in Sec. 2.4. That said, an integration between the two could bring about great benefits given the wide acceptance and the stan-

dardization of OAS, as well as the tooling ecosystem already built around it[1] which allows for API structural documentation generation, specification editing and API endpoints testing through tools such as Postman. The YAML/JSON file of the specification could be parsed in order to generate at least a skeleton of the interactions which can be included in a RESTalk model, leaving it up to the user to order such interactions in the correct sequence. Possible full RESTalk conversation models could be generated when sufficient information is available through appropriate use of the *links* component in OAS which shows the relationship between interactions and can serve as a mechanism to traverse the interactions. On the other hand, RESTalk models themselves can also be used to create the skeleton of the OAS file.

Focusing still on the Design First approach, one of the cornerstone ideas behind Model-driven Engineering (MDE) is the use of DSLs to "express application domain concepts and design intent" [185] and create prescriptive models, which through model-to-text transformations are then converted to code, documentation, test cases, or model serialization [33; 22]. Thus, a tool support can be built so that, given as input the goal of the client using the API (for instance, the input can be in form of clicks on the interactions in the RESTalk diagram in the correct sequence), the tool can automatically generate a skeleton for the client implementation code based on the RESTalk model. Similarly, test-cases can be generated to be used for test-driven development [95], or to be used for testing purposes during the development and evolution of the API in the **Code First approach**.

In the Code First approach, once the API has been in production for a sufficiently long time to generate sufficient amount of execution logs, such logs can be fed into a mining algorithm which can reconstruct the RESTalk conversation model based on the logs. Mining techniques have been successfully applied in the area of business processes for almost two decades, resulting in process discovery, conformance checking, prediction of delays, supporting decision making, recommending process redesign etc. [208]. The IEEE Task Force on Process Mining has even issued a process mining manifesto to promote the topic [209]. In the domain of REST APIs, the reconstructed model of the actual use of the API can serve various purposes. The most obvious purpose is that it can become part of the API documentation to help API client developers to learn the API. Depending on the size of the API, before making it available as documentation, it might make sense to split the reconstructed conversation model into several separate diagrams organized in some logical order based on possible client goals. Another purpose

[1] `https://openapi.tools`

of the reconstructed model, which as mentioned above has been widely used in business process modelling [139], is conformance checking. Conformance checking compares the model of the expected conversation to the reconstructed model of the actual conversation, thus helping to discover divergence between the two models. The service provider can use such information to discover endpoints which are never used and as such can be depreciated, or to discover frequently used paths in the conversation whose performance can be optimised, or also to adjust the limits in the SLA based on the actual usage data.
One of the goals of RESTalk is to set up constraints on the otherwise exponential number of possible sequences of interactions with the service provider, in order to enable the service provider to safeguard the quality and maintainability of its service. Thus, RESTalk models can be used to generate constraints which translate into conversation rules to be used by the runtime service to check the validity of the ongoing conversation with the client [150].
Building the above described envisioned ecosystem around RESTalk requires time and resources, the investment of which can be justified by sufficient interest/adoption of the language.

5.2 Design First Approach - RESTalk Modeler

As mentioned, in the Design First approach the model is generated by the API designer/developer by hand or by using a dedicated editor. In this section we propose two RESTalk specific proof of concept tools, one graphical editor and one textual editor.

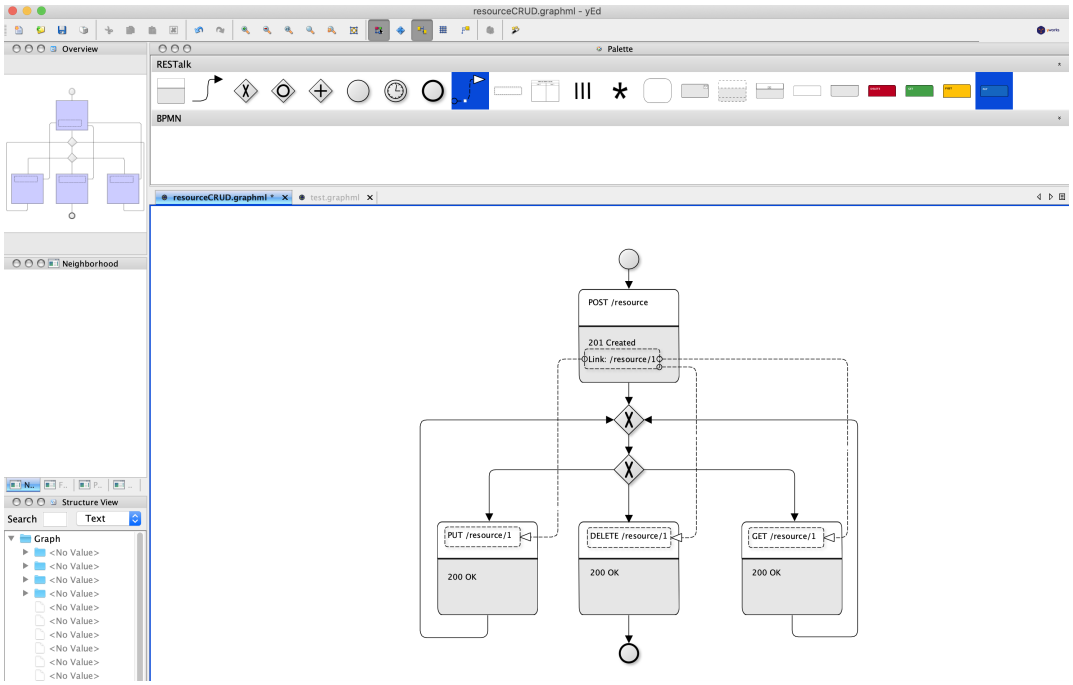
5.2.1 RESTalk Graphical Editor

During the design of RESTalk we have used Inkscape[2], a graphical vector editor [14], to design the language constructs. The features of the tool allow for great flexibility in the design of the constructs and allow export in many different formats such as svg, eps, pdf, etc. Initially we also provided an Inkscape template with all the available constructs which users could copy paste into their diagrams.
To provide more traditional and user friendly graphical drag and drop editor experience we have decided to create a RESTalk palette in yEd[3]. yEd is a freely available tool which runs on all major operating systems. It has been extensively

[2] `https://inkscape.org`
[3] `https://www.yworks.com/products/yed`

used in research (a google scholar key term “yEd graph” search yields over 1’000 results). It supports many of the standard diagrams (BPMN, UML, ER, flowcharts, social networks etc.), but it also provides the functionality of creating custom palettes using constructs from other palettes or importing custom constructs. To use constructs from other palettes it is sufficient to drag them to the canvas and then add them to a palette of choice. Thus, for the RESTalk constructs which are shared with BPMN (e.g., gateways, events, sequence flow, etc.), we have added the constructs from the BPMN palette. To use custom made construct, the user needs to import them in one of the supported formats (jpg, png, svg). Thus, we have created the RESTalk specific constructs (e.g., client-server interaction, state change, message activity, etc.) in Inkscape, exported them as svg and then imported them to yEd.

Figure 5.3. yEd RESTalk palette and example diagram



In Fig. 5.3 we show a screenshot of the tool and an example simple model created with the same. As yEd is a graphical editor dedicated to drawing graphs, drawing edges is more user friendly then in Inkscape as the edges get attached to the constructs thus rendering the layout change easier and faster. It also has some other convenient common features, like graph overview, which is useful to help user’s orientation in large diagrams. Links can be added to each individual element’s properties which enables linking diagrams together, or linking interac-

tion elements to external documentation, for example OAS documentation. The created diagrams can be exported in the usual visual formats (svg, eps, gif, pdf, etc).
yEd uses graphml as a native file format, which is an XML-based file format for graphs. Thus, diagrams are saved in graphml. The advantage is that the xml structure of the file allows for easy algorithmic search for the elements. Description can be added to custom template elements which can be used to algorithmically search for all elements of a given type. These features can facilitate future implementation of a model validator. The RESTalk palette itself can be exported in graphml format, so that it can be shared with yEd users interested in using RESTalk.

5.2.2 RESTalk Textual Editor

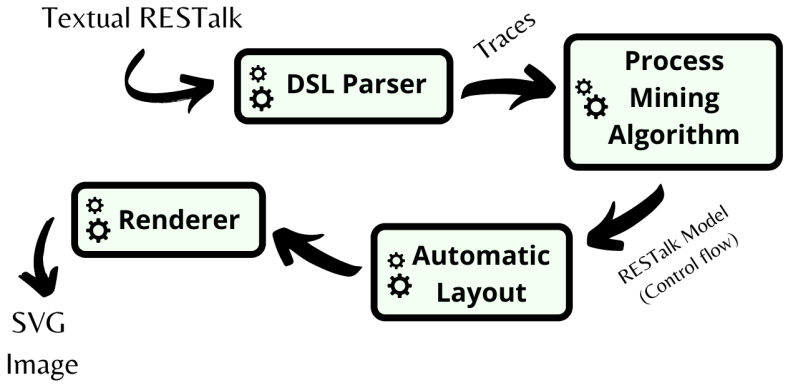
The challenge of graphical editing tools, such as the one described above, is that they are based on explicit modeling, requiring good knowledge of the notation and its semantics, as well as the ability to analyze and abstract the possible interactions and capture them by correctly using the notation. As a consequence, their use can be cumbersome for live modeling during design workshops, where participants should not only provide input but also give feedback on how it has been represented in a model. To overcome this, we have designed a tool which combines notes taking in constrained natural language with the use of a mining algorithm to automatically produce RESTalk diagrams in real-time as API designers/developers describe them with stories. The tool facilitates textual modelling through the use of a mining algorithm, thus reducing the complexity and the number of keywords of the textual DSL as described in Sec. 4.2.3. Its modelling environment replaces the complex stencil palette usually found in graphical editors with a simple text editor. Textual descriptions are transformed into diagrams, which correctly use the RESTalk visual syntax, simultaneously while the modeler is typing them.

Current Version of the RESTalk Textual Editor

As a proof of concept for the applicability of the proposed approach to use mining algorithms for sketching RESTful conversation models, we have designed a tool which takes as input a conversation described with the DSL presented in Sec. 4.2.3 and provides as an output a draft RESTalk model of the conversation. It supports live modeling environment, where the RESTalk model is produced in real-time as the textual description is typed. As it is only a proof of concept tool

it has only implemented part of the core RESTalk constructs.

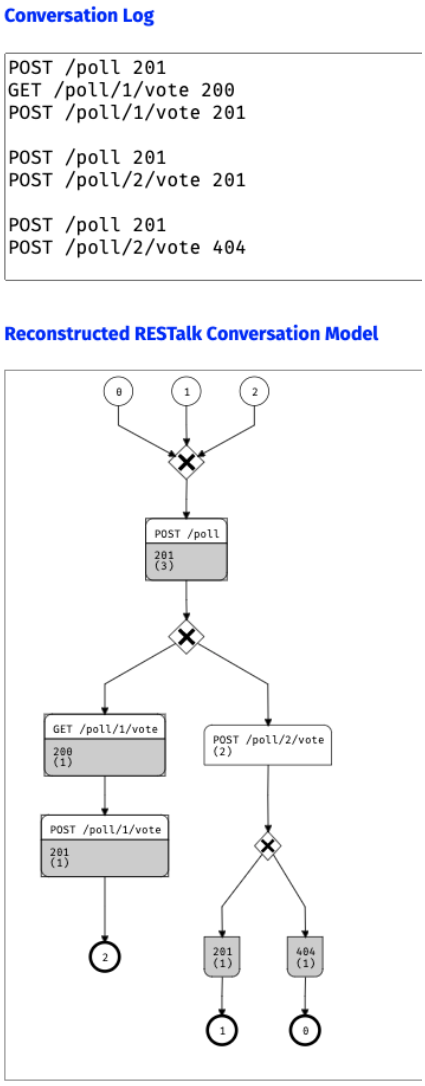
Figure 5.4. Pipeline of the current version of the RESTalk textual editor



The architecture of the current version of the RESTalk textual editor is shown in Fig. 5.4. The textual input is parsed and translated into traces that are then fed into the mining algorithm that generates the control flow model. Then the hierarchical layout algorithm [65; 61], which has been tailored to consider idioms of the notation, is used to produce the graph layout. The result is rendered as a vector image, using the dagre-d3 library[4] to display it. The mining and visualization algorithms are based on the bachelor project work described in Sec. 5.3
The DSL parser translates sequences of interactions into a log form, but it only parses for three information per line, i.e., the method and URI for the request and the status code for the response. No additional information, such as hyperlinks, can be added for the time being. Different alternative conversation instances are separated by an empty line, thus the exclusive gateway to show divergence in the interactions flow due to client decision or due to different server responses is discovered by the mining algorithm and does not need to be explicitly stated. Fig. 5.5 shows a screenshot of the current version of the tool implementation. Other types of gateways, or events different than the start and end event, are still not supported. Also the "..." symbol as placeholder for missing parts of the traces described in Sec. 4.2.3 is not yet supported.
The current version of the tool uses the Alpha algorithm for mining the expended traces [222]. Other mining algorithms could be used in the approach, the only requirement being the existence of an API to automatically feed the mining algorithm with the expanded traces. Additionally, users typing the traces should not have to wait to see the resulting model, but the model should be updated *live* as

[4] `https://github.com/dagrejs/dagre-d3`

Figure 5.5. Screenshot of the current version of the RESTalk textual editor



new entries of the traces are added. This could be achieved with an incremental mining approach [127]. Aiming at a proof of concept of this novel modeling approach, the validation of different mining algorithms for its implementation was not in the scope of this dissertation work.

Envisioned Version of the RESTalk Textual Editor

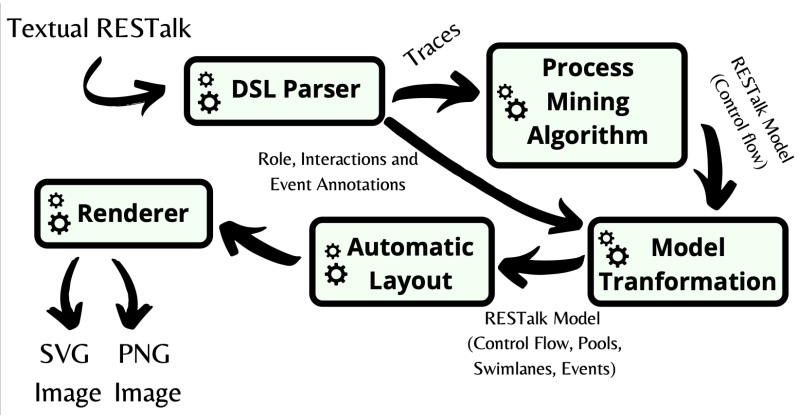
Inspired by the initial proof of concept tool for RESTalk textual editor described above, we decided to use the innovative approach of leveraging on mining al-

gorithms to decrease the cognitive effort of the user when modelling business processes with BPMN [88; 92]. The reasons for changing focus from RESTalk to BPMN was higher interest expected from the BPMN community as BPMN is a well established and frequently used standard in industry [116], thought in universities, with existing tool support [66], while RESTalk still needs to obtain its acceptance as a DSL. Furthermore, requirements elicitation interviews for business processes typically involve people with different skills and background [9], such as process participants with business domain knowledge, the business analysts with process modelling knowledge, and the software engineers with IT background. Business process participants, but also BPMN novice students at universities, might be less accustomed to use abstraction compared to computer scientists usually involved in the REST API design. Enabling these novice BPMN users to tell their stories as an ordered list of tasks seemed like a stronger use-case than the one of RESTalk, thus motivating us to focus on the development of the BPMN Sketch Miner[5]. Nonetheless, the design decisions we took for the BPMN Sketch Miner and many of the functionalities which Prof. Pautasso implemented in the tool are valid for and can be replicated in future versions of the RESTalk textual editor. In this part of the thesis we explain them in detail.
Some of the design decisions used for the BPMN Sketch Miner but applicable for a RESTalk textual editor as well are: 1) to deliver it as a Web-based tool, thus avoiding the need for software installation to get started with a modeling session; 2) to embed the textual description of the model in the browser link so that it can be easily shared in an email message or chat room; 3) to support life update of the visual RESTalk diagram as the user is editing the textual description with no need to click a submit or refresh button; 4) to support export of the generated RESTalk model in SVG, PNG, or XML formats so that it can be displayed or further refined in any compatible tool. The design decision to support life update of the visual RESTalk diagram has the aim of facilitating the learning of the basic RESTalk constructs, in addition to providing fast feedback to the users of the tool. Life update is not frequently used with existing textual modeling languages. For instance, in PlantUML the textual model gets synchronized with the visual model only upon request, while plantBPMN generates a file in a BPMN XML format which then needs to be imported into a dedicated tool, such as Signavio[6] or an Eclipse plug-in, for visualization and further editing.
Using textual input to generate more complex RESTalk visual models requires multiple model generation and transformation steps as shown in Fig. 5.6. In

[5] `https://www.bpmn-sketch-miner.ai`
[6] `https://www.signavio.com`

addition to the components mentioned in Fig. 5.4, a model transformation component needs to be added because the textual annotation of roles, call requests, event types etc. is part of the textual modelling and thus is not fed into the mining algorithm, but used during a second stage as follows: first, the nodes of the control flow graph are transformed into interactions or events (according to the type annotations found in the original description). Then, when applicable the role annotations are used to place the interactions and events in the corresponding swimlanes. Additionally we would like to enable an export option in different formats such as svg or png, as in the current version only display with no export is available.

Figure 5.6. Pipeline of the envisioned version of the RESTalk textual editor (Adapted from [92])



The envisioned future version of the tool would implement the placeholder "..." symbol, following the same logic as in the implementation of the BPMN Sketch Miner. Namely, the tool would first expand the traces written in the DSL to obtain the complete traces of all the possible paths that can be taken in the conversation. Namely, the DSL user input would be parsed and each time the "..." symbol would be encountered, depending on its position, the algorithm would perform one of the following actions:

- If the "..." symbol is at the start of the conversation instance trace the algorithm would search for the first interaction which is after the "..." symbol in all the other expanded conversation instance traces. When it finds it, it would take all the interactions which precede it, thus creating one or more missing conversation fragments, which would then be used to expand the initial conversation instance trace;
- If the "..." symbol is at the end of the conversation instance trace the algorithm would search for the last interaction which is before the "..." symbol in all the

Figure 5.7. From "..." placeholder to expanded traces

Compressed DSL conversation instance log	Expanded conversation instance log	Conversation instance number	Assigned interactions to participant
participant X A B C	 A B C	 1	 A B C
participant Y ... C D ... C E ... I J	 A B C D A B C E F G H I J	 2 3	 D E I J
participant Z ... E F G H I ...	Not expanded, only used to expand compressed logs		F G H

other expanded conversation instance traces. When it finds it, it would take all the interactions which succeed it, thus creating one or more missing conversation fragments, which would then be used to expand the initial conversation instance trace;

• If the "..." symbol is in the middle of the conversation instance trace the algorithm would search for the first interaction which is before the "..." symbol and the first interaction which is after the "..." symbol in all the other expanded conversation instance traces. When it finds both these interactions in the correct order, it would take all the interactions which are between them, thus creating

one or more missing interaction fragments, which would then be used to expand the initial conversation instance trace.
As the trace expansion algorithm would act recursively and search all expanded traces, if different sequences are to be identified as a match in different conversation instance traces, then they would all be used to expand the analyzed trace resulting with multiple expanded traces per one compressed trace. A simple example of how compressed traces would be expanded by the tool before passing them to the mining algorithm is provided in Fig. 5.7. As evident from the example, the assumption used by the expansion algorithm is the existence of at least one common interaction between the compressed trace and the other traces, as the algorithm uses the common interaction to identify the missing part of the path. The use of the "..." symbol is meant to make the writing down of traces more time efficient, as it allows the users to avoid repeating sequences of interactions which they have already written down. This is especially handy when there are alternative flows in large diagrams. However, a limitation of such approach is that it can lead to over-fitting, i.e., the automatically derived model might allow for conversation instances which are not described in the textual DSL. To deal with this limitation traces of such over-fitting instances could be generated and presented to the user so that the user can decide if some of those traces should be excluded from the model.
This envisioned textual editor for RESTalk could be built from scratch, as is the case with its current version, or in the future the possibility of using a DSL workbench to build such an editor can be evaluated, such as the Ensō DSL workbench[7].

5.3 Code First Approach - RESTalk Miner

With the commercialization of the world wide web, web mining has gained on importance, with three main streams: web content mining, web structure mining and web usage mining [16]. Web usage mining refers to automatic discovery of user access patterns with the goal of optimizing or customizing websites. While web usage mining refers to purely sequential logs from human users browsing websites, in Web services the interaction is machine to machine, and concurrent interactions may occur. Process mining does not necessarily involve human interactions and allows for non-sequentially. Therefore, the applicability of process mining in the Web service context has been pinpointed [211], as it can be used to discover dependencies in service-oriented systems.
Most of the web-services today use the REST architectural style and mining their

[7] `http://enso-lang.org`

logs can bring to interesting insights regarding how different clients actually use REST APIs. This can help developers detect unexpected usage patterns of their APIs by comparing different clients' conversations, or to pinpoint interactions which are worth optimizing as they are being used by most of the clients. For instance, if there is a sequence of several requests which is frequently followed, the API designer might decide to provide in the first request a direct link to the last request, thus avoiding the clients having to make the intermediary requests. Bugs might also become evident, such as unauthorized access to some resources or frequent error messages after a certain sequence of requests. Mining of RESTful services requires a model-driven approach to RESTful conversations and a visualization language such as RESTalk. Although different mining tools with graph visualization already exist [1; 216; 200], their visualization is not REST domain specific. On the other hand, commercial REST specific analytic tools[8] provide incoming API requests capturing and database querying analytics regarding the usage of the API with different filtering options and timeseries visualization. However, they do not support visualization of clients' conversations with the server.
That said we have decided to dedicate the bachelor thesis of Ilija Gjorgjiev to developing a REST specific mining tool which we called the RESTalk Miner[9]. RESTalk Miner takes as an input a log file, containing the log entries of conversation instances with different clients. Several assumptions regarding the logs need to be valid in order for them to be correctly parsed by the tool. Each log entry should comply to the following format:

$$\overbrace{DD/MM/YYYY}^{\text{Date}}\ \overbrace{HH:MM:SS}^{\text{Time}}\ \overbrace{3.171.112.202}^{\text{Client IP Address}}\ \overbrace{POST}^{\text{Method}}\ \overbrace{/job}^{\text{URI}}\ \overbrace{202}^{\text{Status Code}}$$

Also each log entry should be written in a new line. The logs should be just from one API provider, thus conversations with multiple servers are not supported. Given the statelessness principle, RESTful interactions do not include a correlation identifier to distinguish between different conversation instances. In fact, Stroinski et al. [201; 199] propose enriching logs with context information to enable mapping of logs to execution instances. When such information is missing the only viable assumption is that all interactions with a given IP address (client) are part of the same conversation instance. However, in order to relax such improbable assumption, we used an algorithm to split the interactions of a given IP address in different time periods based on a difference threshold. Then

[8] `https://www.moesif.com`
[9] https://github.com/USI-INF-Software/RESTfulConversationMining

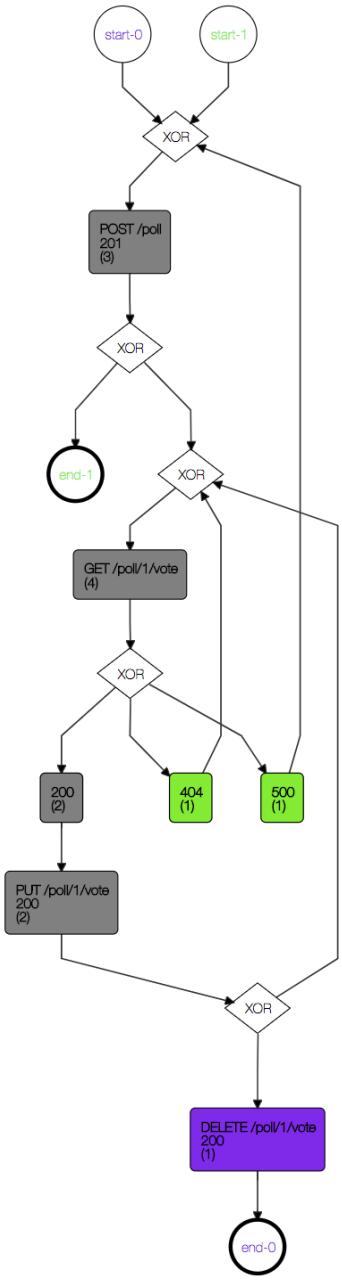
the user of the tool can decide which time periods to visualize.
In addition to the log file, the RESTalk Miner takes an optional file input which contains the URI templates derived from the API's OAS. For instance, to abstract the following URI /content/serial/ title /issn/03029743 the URI template would look like this /content/serial/ title /issn/:id. Such abstraction ensures that identical method calls to the same type of resource are visualized as one request. The main default output of the RESTalk Miner is a simplified RESTalk graph showing all the conversations different clients have initiated with one particular server. Alternatively, the user can select to visualize only the conversations of client(s) of interest. As in RESTalk, the nodes in the graph take the form of a juxtaposed request-response, but in addition to the information about the HTTP method, URI and response status code, they also contain information about the number of log entries in which this node has appeared. From the remaining RESTalk constructs, the tool currently only supports the exclusive gateway, both in case of diverging paths taken by the client, and in case of alternative responses provided by the server, as well as the start and end event. Depending on user's preference, the graph can be flattened by abstracting from the URI information and showing only the methods that have been called.

5.3.1 RESTalk Graph and Comparative Statistics Visualization

Once the above mentioned graph has been generated, the user can activate or deactivate one or more different interactive visualizations:

- The ***node frequency colouring*** colours nodes from red to yellow depending on the number of log entries that contain the particular request/response pair with red being the most frequent node and yellow being the least frequent node. It allows the user to spot the frequency of the nodes visually in addition to the absolute number of the logs containing the node already present as an information in the node itself;
- The ***edge frequency thickness*** adjusts the thickness of the edges based on how many clients follow the same path. It shows to the user the most frequently used paths;
- The ***edge delay coloring*** colours the edges from red to yellow depending on the average difference in the timestamps of the nodes that the edge connects. Gradient coloring is used where red is the maximum delay and yellow is the minimum delay;
- The ***edge probability*** shows a probability of an alternative path being taken after an exclusive gateway. The probability is computed based on the log entries;

Figure 5.8. RESTalk Miner overlapping vs. unique parts of conversations visualization



- The **status colouring** colours responses based on their status codes. This facilitates the identification of certain status codes by the user (e.g., erroneous status codes);

- The ***conversation path colouring*** colours in a unique colour all the requests made by the same client and in a mix of colours the nodes which are shared between clients in case multiple clients are selected. It is activated by clicking on the start event of a given conversation instance. This feature enables the user to visualize the interactions of a given client with respect to other clients. We show an example of this visualization in Fig. 5.8 where the nodes unique for client 1 are purple, the nodes unique for client 2 are green and the shared nodes between client 1 and client 2 are gray.

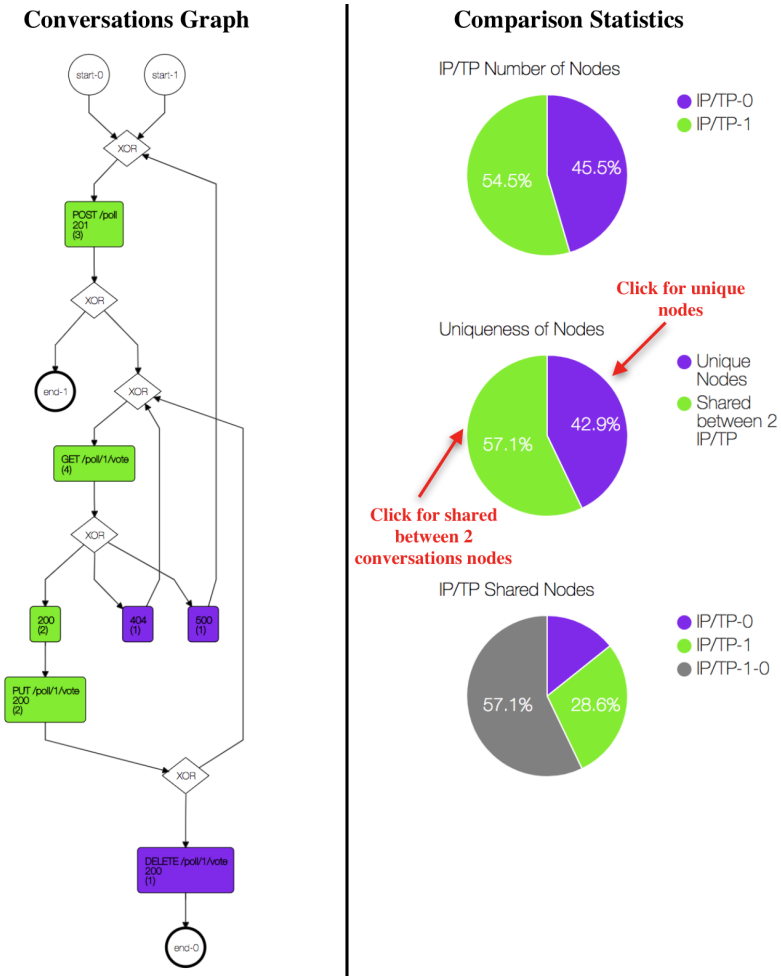
The tool also provides the user with pie chart visualization of statistical data regarding the analyzed clients of the RESTful service (see Fig. 5.9).

- The ***number of nodes*** pie chart shows how many of the total request/response nodes belong to each individual conversation (given client in a given time period), i.e., how lengthy each conversation is.
- The ***uniqueness of nodes*** pie chart shows how many nodes are unique to just one client, how many are shared between two, three clients etc. Clicking on a certain slice of the pie colours in the same colour the nodes it refers to as evident in Fig. 5.9.
- The **shared nodes** pie chart shows the number of requests shared between combinations of different specific clients. For example IP/TP-0 in the pie chart will show the number of nodes that are only present in conversation 0, IP/TP-1-0 will show the number of nodes that are shared only between conversations 1 and 0.
- The ***dynamic sharing*** pie chart uses the same computation as the shared nodes pie chart, but only for the clients selected by the user instead of all clients. It is shown only if the conversation path coloring feature is active.

5.3.2 Pattern Matching, Discovery and Visualization

Patterns [134] represent a systematic form of knowledge sharing as they establish a common vocabulary to describe recurring RESTful conversations [161]. Patterns can be used to pinpoint and discuss API design best practices or the absence of the same. RESTalk Miner supports two types of pattern searches. Searching for unknown patterns, i.e., pattern discovery, and searching for known patterns, i.e., pattern matching. The pattern discovery can help identify new API design approaches and best practices, while the pattern matching can allow to search for patterns of interest. When ***searching for unknown patterns*** the user needs to specify the number of nodes the pattern should contain and the minimal number of clients that must have used that pattern. If patterns that match these criteria are identified, the user can select them one by one from a dropdown list, visual-

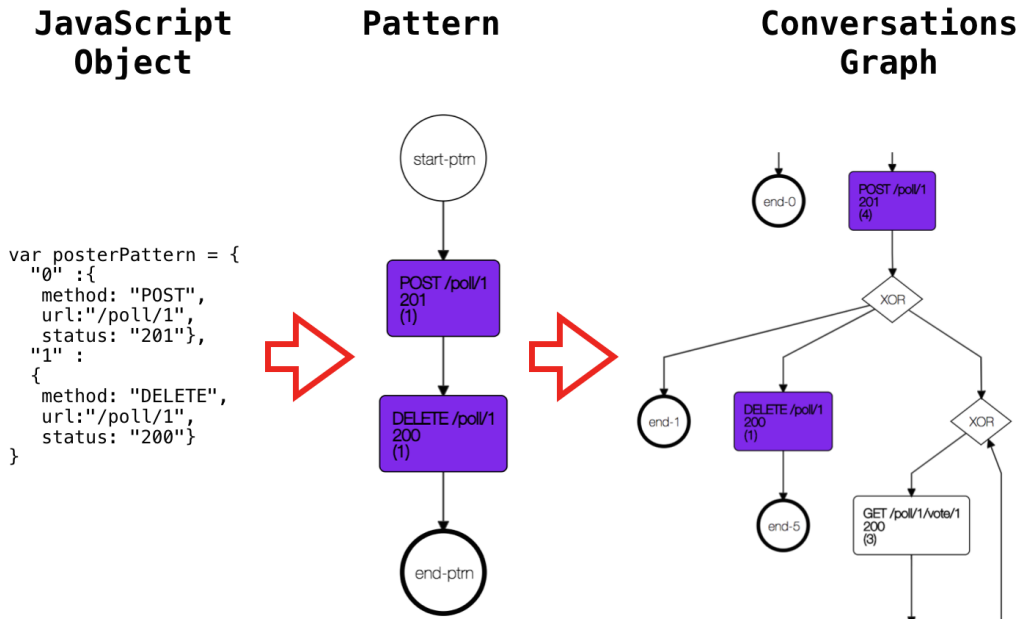
Figure 5.9. RESTalk Miner pie chart visualizations



ize them and/or save them. Saved patterns can be used later as ***known patterns*** to be searched for in other conversations. The user can also upload known patterns based on his/her experience or bast practices and search for them in the given conversation. Such patterns need to be described in JSON with log object describing the conversation pattern to be matched (see Fig. 5.10). Every pattern log entry has a similar structure as the execution logs (without the timestamp and IP address), with the difference that any of the elements (Method, Status Code, URI, etc.) can be substituted by a $*$ symbol, meaning that any value of that element will be considered a match when searching for the pattern. URI's values can also be used as placeholders, i.e., ensuring that the same URI is used without precisely specifying the URI value. An optional separator element in the pattern description allows for the specified log entries not to be direct succes-

sors. For instance, if we are searching for a pattern with two log entries (POST /example/1 and DELETE /example/1), if we use this separator element in the pattern description, an occurrence of POST /example/1 followed by PUT /example/1 followed by DELETE /example/1 will also be considered a match. Such description of the pattern we are searching for allows for greater expressiveness to match targeted patterns.

Figure 5.10. RESTalk Miner pattern matching



As evident in Fig. 5.10 matched patterns are coloured in the graph. If multiple occurrences of the matched pattern exist, each occurrence will have a different color.

A screencast of the main functionalities of RESTalk Miner is available on YouTube[10] and a demo paper has been published in [88]. Details regarding the implementation of the tool are available at Ilija Gjorgjiev's thesis. At its current state the RESTalk Miner does not support the visualization of the hyperlink flow nor does it detect concurrency in the mined logs or other constructs such as timer events or participant lanes. In the future it can be extended to fully support the RESTalk language. Its user interface can also be improved and it can be transformed from a command line tool to an online web service which would allow for user registration, and thus storage of user's logs and patterns. Studies which use the RESTalk Miner given a real log of API interactions can be performed to

[10] https://youtu.be/N94clNa5Mlg

search for interesting findings with respect to that log. The challenge here would be to avoid spaghetti like graphs which are hard to analyze and draw conclusions from. An option would be to add features in the tool letting the user to balance between the precision of the generated graph (including all log entries) and the generalizability of the generated graph (including only the most frequent log entries) [208].

5.4 Chapter Summary

Depending on the intended use of a DSL, designing and implementing the tooling ecosystem around it can be an important and resource intensive endeavour. In this chapter we have discussed our vision of the ideal RESTalk editor which would support the synchronous editing of both the graphical and the textual DSL representation. We also presented other possible tools to be integrated into the editor, such as a model validator and a simulator. The current state of the implemented standalone editors for the graphical and textual representation are far from our vision for an integrated editor which remains as future work.
To ensure wide adoption of a DSL, it is important to built an ecosystem around it, integrating it with existing tools in the domain. In the case of RESTalk we consider the integration with OAS important to work towards completeness of REST APIs documentation from different viewpoints. We also envision support for the MDE stream of thought, where RESTalk models would be used to generate skeleton for the client implementation code. Once an API has been used for a sufficiently long time and has sufficient logs of calls from clients to the API, a mining algorithm can be used to look into the actual conversations with different clients. As part of a bachelor project we have implemented such a miner which we described in this chapter and whose functionalities can be enhanced in the future, among other things, also with conformance checking option given a RESTalk model of the expected API behaviour.

Part III

RESTalk Evaluation

This third part of the thesis refers to the third layer of the DSL design framework discussed in [94] and presented in Fig. 4.1, the Evaluation layer. The aim of the Evaluation layer is to assess the language against predefined criteria in order to identify potential need for refinement [94]. It is rarely possible to follow a waterfall approach when designing a DSL, as evidenced by the continuous release of new versions of many standard languages. Each DSL keeps on evolving as its domain is evolving, and as users provide feedback on identified gaps in the language, or situations which can not be reliably modeled with it. RESTalk is not an exception to this practice, and we support the agile approach in language design with frequent iterations.
To be useful a model must: 1) provide for **abstraction** and hiding of irrelevant details; 2) be **understandable** which depends directly on the modelling notation used and its expressiveness; 3) be an **accurate** representation of reality; 4) allow for **predicting** the behaviour of the modelled system; and 5) be **inexpensive** to create [190]. In this part of the dissertation we are going to discuss whether some of these characteristics are satisfied by models created with RESTalk. We will start in Chapter 6 by evaluating the need of RESTalk and the capability of RESTalk to express real-world examples of RESTful API interactions. Then in Chapter 7 we will use different research techniques to evaluate the design of RESTalk, as well as its usefulness for understanding and using a given RESTful API.

Chapter 6

RESTalk Formative Evaluation

In this chapter we will discuss the formative evaluation of RESTalk used throughout the PhD, which contributed to the iterative design of the language to its current version. We start with an exploratory survey in Sec. 6.1 conducted with industry practitioners with the aim of evaluating the need of a DSL and obtaining feedback on the initial design of the core DSL. Then, in Sec. 6.2 we discuss different use-cases which allowed us to expand the expressiveness of RESTalk by identifying different realistic or real API behaviours scenarios which we considered important to be supported by RESTalk.

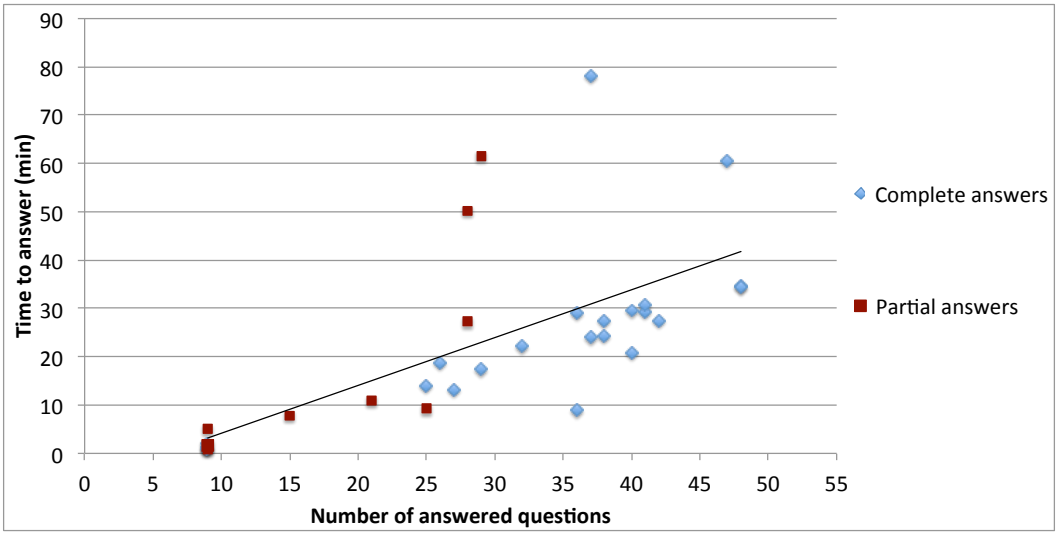
6.1 Exploratory Survey

One general problem with modeling languages is their dissemination and acceptance by the targeted modeler community. To address this issue, after drafting the first version of RESTalk and testing its expressiveness with several examples of RESTful conversations, we have evaluated it with an exploratory survey. An exploratory survey is a qualitative research technique for understanding the viewpoint of the surveyed subjects about the addressed problem [227, Chap. 2]. The goal of the survey was to obtain initial feedback, primarily from industry, on some of RESTalk's cognitive dimensions [73]. Namely, we wanted to obtain insights to help us manage the unavoidable trade-off between the expressiveness and completeness of RESTalk on one hand, and its simplicity and understandability on the other [67]. We also wanted to elicit whether there is a need in industry for explicit modeling of RESTful conversations with a domain specific language.

6.1.1 Survey design

Given the exploratory nature of the survey, we have mostly used open-ended questions. Their purpose was to gain understanding on industry's existing practices in representing the client-server interactions with a REST API, and to obtain respondents' opinion on RESTalk, its cognitive characteristics, and usefulness. To reach a greater audience we have translated the survey both in English and German and made it available on-line. When answering all the questions in detail, the expected duration of the survey as designed was 30-40 min, which fits with the actual time spent by the participants (Fig. 6.1).

Figure 6.1. Time dedicated to filling out the survey with distinction between complete and partial answers



We have divided the questions in the following seven groups: demographic data, background on used notations in practice, RESTalk's intuitiveness, RESTalk vs. standard BPMN Choreography, reading task, modeling task, and RESTalk's evaluation.

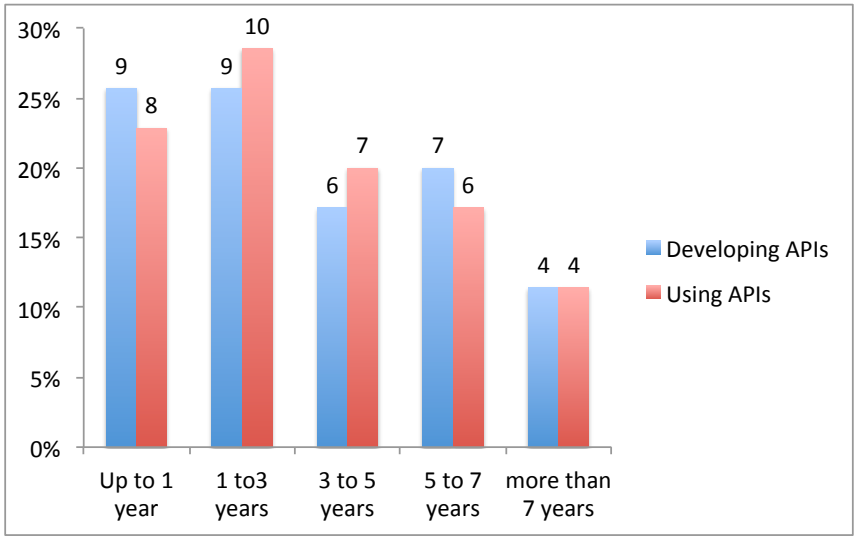
We have started with questions about participants' background and experience in designing and using REST APIs. To understand the existing modeling practices in the REST API design, we have further inquired the visual notation(s) respondents have used in such circumstances. The question concerning the used notations was a multiple-choice question in order to get a full picture of all the notations used in practice. However, the more detailed questions referred only to one of the selected notations in order to avoid constructing a too lengthy survey. The priority list for the detailed questions has been as follows: BPMN Choreography,

In-house developed notation, UML Sequence diagrams, UML Activity diagrams, Other standard notations. The detailed questions were open-ended and explored respondents' perception about the pros and cons of the notation, as well as the effort of learning it and the effect its usage has had on the team's productivity.
To assess the intuitiveness of RESTalk, before explaining it in detail, we have asked the respondents to describe, in their own words, a simple conversation we had provided the diagram of, and then to respond to several specific multiple-choice questions assessing their understanding of the conversation.
As per Gemino and Wand's [67] framework for empirical evaluation of conceptual modeling techniques, such techniques ought to be compared based on their grammar, i.e., the modeling constructs and the rules for combining them. Thus, to those respondents who had basic knowledge of BPMN we have also asked some inter-grammar comparison questions. We have first explained them the extensions and the modifications we have made to the standard BPMN Choreography. Then, based on a generic example modeled in the standard BPMN Choreography (Fig. 4.7a) and in RESTalk (Fig. 4.7b), we have asked them to compare the two in terms of conciseness, expressiveness, and ease of understanding. We have closed this block of questions by asking which notation they would prefer using.
For the respondents who did not have prior BPMN knowledge, we have provided a short RESTalk tutorial (Fig. 4.7b), describing all of the constructs used in subsequent tasks as well as their semantics, without diving in depth into the full complexity of BPMN. This tutorial was made available for further reference during the survey. We have not presented the assumptions and simplifications of the modeled reality mentioned in Sec. 4.2.2 to any of the respondents, in order to explore their validity.
To further evaluate RESTalk, we have included both a reading and a modeling task to be answered by all respondents. After these tasks, we have asked them to evaluate RESTalk in terms of conciseness, understandability and efficiency, as well as to identify any ambiguous semantics or missing elements. We have used open-ended questions in order to obtain more detailed feedback.
Not being interested in statistical inference, we have made most of the questions optional in order to encourage greater survey participation. Only the questions used in determining the survey logic, i.e., which questions to be disclosed to which audience, were mandatory. Essentially, the question regarding experience with using visual notations for modeling RESTful conversation, and the question regarding prior BPMN knowledge. A print out of the online survey is available in Appendix A.

6.1.2 Survey sample

We have kept the on-line survey open for almost 3 months, i.e., until we have gained sufficient valuable input from the respondents. Thus, we were not aiming at reaching a minimal sample size of the targeted population. While we were targeting primarily industry practitioners, we have also used some conferences and social media to reach a broader audience. The reason for focusing mainly on industry, was to get feedback on their willingness to use visual notations. Namely, while in academia modeling notations such as UML or BPMN are frequently taught, their acceptance in industry seems to be limited [164]. However, we believe that if a notation is developed in an agile manner, with a continuous direct contribution from industry practitioners, it is more likely to fit their actual needs and thus to achieve greater adoption.

Figure 6.2. Respondents' experience with REST APIs

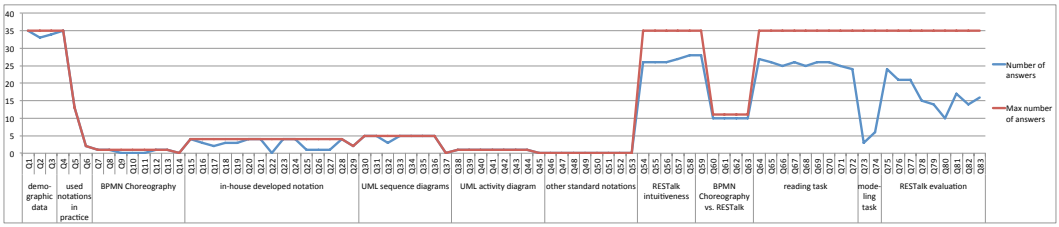


From the total of 35 respondents, 74% (26 individual responses) are from industry and 26% (9 individual responses) are from academia, based on the job title they have provided. Their experience with using, and/or designing REST APIs, ranges from couple of months to more than 10 years, with almost half of them having more than 3 years of experience. More details can be found in Fig. 6.2. The data labels in the graph show the absolute number of respondents in each experience group. No other demographic data has been considered relevant for the current study.

The sample includes a broad range of practitioner roles from researchers, through IT consultants, software quality engineers, developers, architects and up to a

CTO. The average time dedicated to filling out the survey has been calculated to 23 minutes, with the time increasing as the number of answered questions increased. Given the fact that the survey could be paused and resumed later, the average time has been calculated after removing the data for 3 persons who took over 3 hours to fill out the survey. Nonetheless, their answers have been taken into consideration for the rest of the survey analysis.
Since conditional logic was used in the survey, the number of questions to be answered differed among different respondents, with 8-14 additional questions for respondents with modeling experience. However, double checking the responses with time to answer values above the trend line, has indicated that they have dedicated more time to provide more exhaustive answers. Further details are available in the scatter plot in Fig. 6.1 where also complete vs. partial answers are evident. The scatter plot shows that all of the partial answers contain the first 10 questions, with 28 questions emerging as a limit. As can be derived from Fig. 6.3, this limit is due to the modeling task which the respondents probably found more challenging or time consuming.

Figure 6.3. Maximum vs. actual number of answers per question



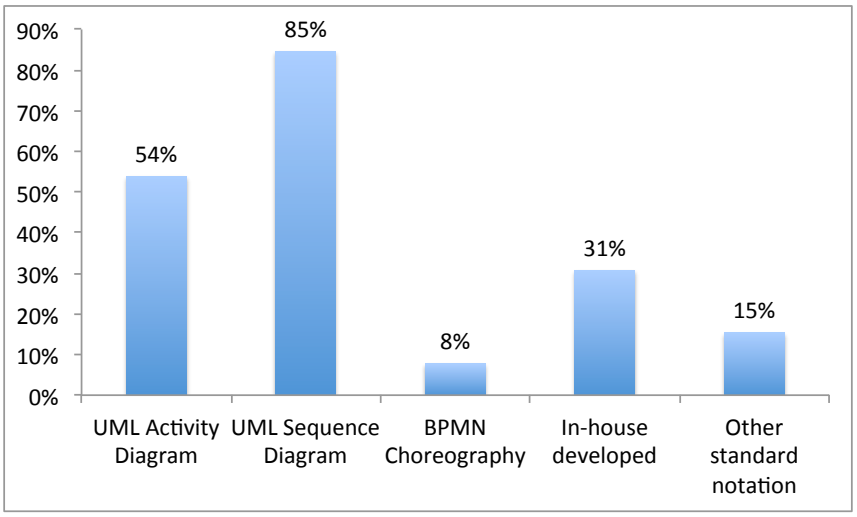
6.1.3 Survey results

As mentioned in Sec. 6.1.1, most of the survey questions were not mandatory and survey conditional logic was incorporated in two question groups: **Used notations in practice**, which was only presented to participants who have used the concrete notation before, and **RESTalk vs. Standard BPMN Choreography**, which was only presented to participants with prior BPMN knowledge. Fig. 6.3 shows the theoretical maximum number of respondents per question, given the survey sample of 35, and the actual number of respondents. You can use it as a reference for the absolute number of respondents per question, given the percentages mentioned in the remaining part of this section.

Used notations in practice

Out of all the respondents, 38% have used some visual notations to discuss the life-cycle of resources and allowed HTTP interactions within a REST API. The percentage is equal in industry and academia. These respondents have been asked to choose one or more of the following notations they have used: BPMN Choreography, In-house developed notation, UML Sequence diagrams, UML Activity diagrams, or Other standard notations. UML Sequence diagrams have emerged as the most widely used, with as many as 85% of the respondents using it, while BPMN Choreography has been used by just one person from academia who stated that he has used it for "research incentives". Details are available in Fig. 6.4. Such results are not surprising given the longevity of UML. The distribution is similar even if we analyze the answers disaggregated between industry and academia.

Figure 6.4. Used visual notations for RESTful conversations in practice



In-house developed notations have been used by four persons, mainly due to lack of knowledge of existing standards, their complexity or lack of flexibility. These are similar reasons to what Petre has identified regarding UML use in practice in [164]. The respondents have developed simple, ad-hoc notations, or simplified the UML notation, so that the diagrams can easily be drawn by hand or with a ready-to-use tool. The one person who is no longer using the in-house developed notation, states the reason being the fact that "it was not searchable, shareable or usable after forgetting the context".

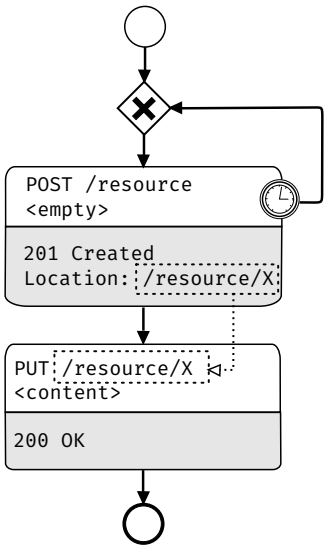
All of the persons who have declared having used UML sequence diagrams, are still using them, mainly due to their effectiveness in depicting the order of interactions between the client and the server. The UML sequence diagram language

features they appreciate the most are the possibility to show the creation and destruction of resources, concurrency and the direction of the exchanged messages. Nonetheless, they consider it challenging to express “dynamic calls or a lot of third party services and failover tools”, as well as conditions, loops or resource’s state. Regardless of such limitations, they believe that using UML sequence diagrams has improved their productivity by improving the team’s understanding of the interactions and driving design discussions. Furthermore, they believe that the UML notation has a fast learning curve, which is expected given that many study it during their formal education.

RESTalk’s intuitiveness

A notation is intuitive if it is easy to understand without explicit instruction. Therefore, before providing any tutorial for RESTalk or explaining how it is different from the BPMN Choreography, we have shown respondents the diagram in Fig. 6.5, and posed an open-ended question to describe its meaning using natural language.

Figure 6.5. Diagram used for assessing RESTalk’s intuitiveness



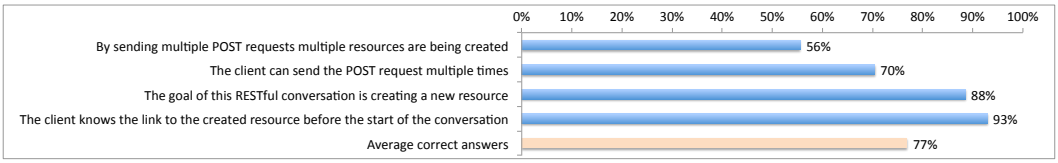
As per the answers, it is intuitive from the diagram that an empty resource is created and the content is added later on. The concept of the hyperlink flow seems to be clear as well. The elements that have emerged as ambiguous were the exclusive merge gateway and the timer event. They depict a situation in which the server does not respond to the POST request, and thus the client sends

the request again. For instance, one person expressed doubts on whether the "process makes a POST or is waiting for a POST", while another one was not sure whether the timeout occurs "if the response takes too long (or if the request is empty?)". One person from academia interpreted the gateway as an unspecified data-based decision being made.
After the open-ended question we have also asked 4 multiple choice questions with only one correct answer (marked in bold):

1. The goal of the conversation is:
    - editing an existing resource
    - **creating a new resource**
    - creating multiple new resources
    - none of the answers
2. The client can send the POST request multiple times:
    - **true**
    - false
    - I don't know
3. By sending multiple POST requests multiple resources are being created:
    - **true**
    - false
    - I don't know
4. The client knows the link to the created resource before the start of the conversation:
    - true
    - **false**
    - I don't know

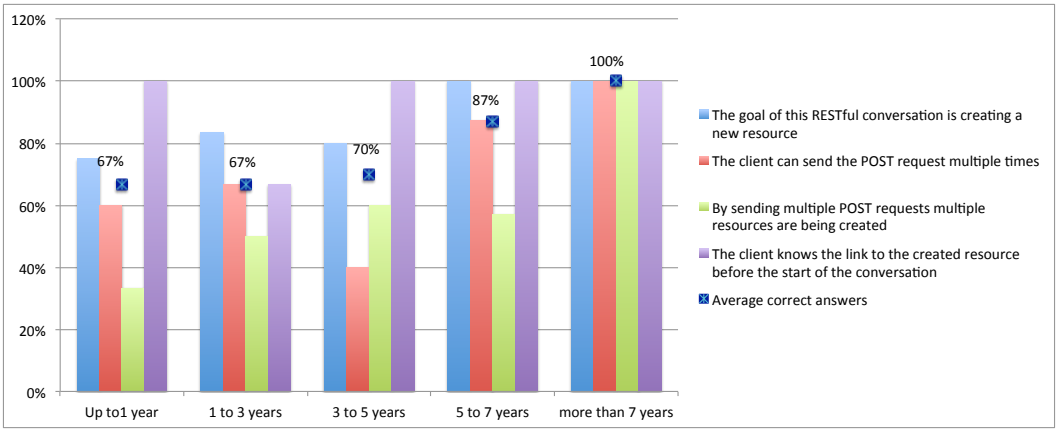
As can be seen in Fig. 6.6, there is a fairly high average number of correct answers (77%), which we consider an indication of a rather intuitive notation. As was evident from the open-ended question answers, the timer event has caused confusion on how many resources are created if sending the POST request multiple times. While 56% have understood it well, other 22% have stated that they do not know the answer. If we disaggregate the answers by sector, the academia has higher percentage of correct answers to this question (75%). This is not surprising given that 57% of them have prior BPMN knowledge, and the timer event is one of BPMN's core constructs. It could also explain the slightly higher average number of correct answers in academia (81%) compared to industry (75%).
If we look at the intuitiveness from the respondents' experience perspective, we notice an increase in the average number of correct answers as the respondents' experience increases (Fig. 6.7). This is to be expected in such a simple diagram, where knowledge of the REST context can be sufficient for making educated

Figure 6.6. Multiple choice questions for assessing RESTalk's intuitiveness (ordered by percentage of correct answers)



guess on constructs which might be less intuitive. On the other hand, drilling down on the results applying as a criteria respondents' experience with using a visual notation to depict RESTful conversations, has not revealed meaningful differences in the answers between respondents who have used visual notations and those who have not.

Figure 6.7. Assessing RESTalk's intuitiveness from respondents' experience perspective

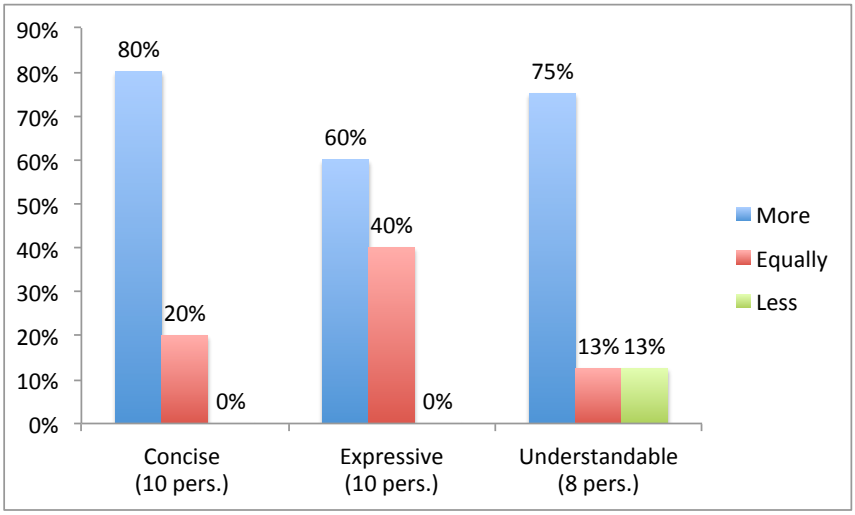


RESTalk vs. Standard BPMN Choreography

Given that RESTalk is based on BPMN Choreographies, we used different approaches in explaining it to people with and without BPMN experience. As per the survey results, 41% of the total respondents have some basic BPMN knowledge, with that percentage being higher in academia (57% or 4 persons) than in industry (35% or 7 persons). After having explained the main modifications made to the standard BPMN Choreography, we have asked these respondents to compare an exemplary model designed with standard BPMN Choreography (Fig. 4.7a) to the same model designed with RESTalk (Fig. 4.7b). As can be seen from Fig. 6.8, respondents' perception is positive since nobody finds RESTalk

less concise or less expressive than the standard BPMN Choreography. Only one person from industry found the standard BPMN Choreography more understandable, but without providing further details on the reasons. The number of persons who have answered the questions is provided in parenthesis in the x-axis labels in Fig. 6.8.
Since as we have seen in Fig. 6.4 UML Sequence diagrams are the most widely used notation for depicting RESTful interactions, we have decided to delve into the answers of respondents who have used UML Sequence diagrams for this purpose before (7 persons). All of them but one find RESTalk less time consuming and more than or equally concise to UML Sequence diagrams.

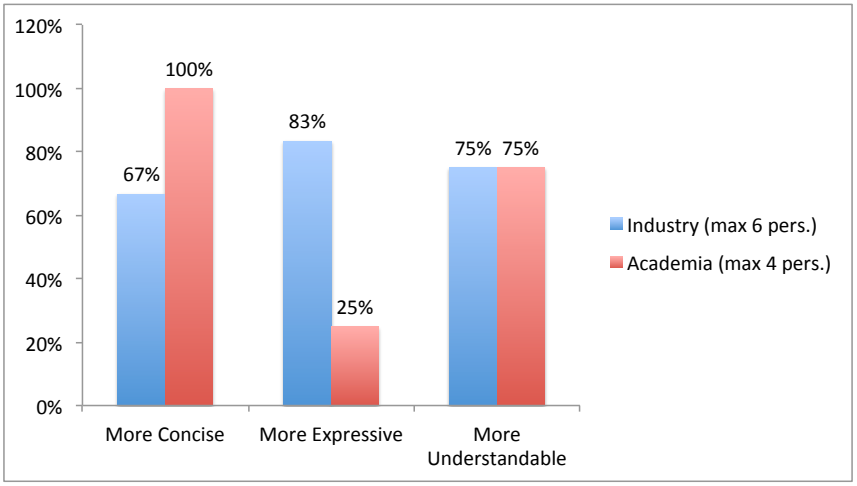
Figure 6.8. RESTalk vs. Standard BPMN Choreography



If we analyze the responses comparing industry vs. academia (Fig. 6.9), we notice that industry participants find RESTalk significantly more expressive than the ones from academia, while they both agree on it being more understandable than the standard BPMN Choreography. These questions were answered by 4 to 6 participants from industry[1] and 4 persons from academia.
However, all respondents prefer using RESTalk to the standard BPMN Choreography due to lower overhead, greater simplicity and "better notion of what response belongs to what request".

[1]Since the questions were not mandatory a few industry participants did not answer all of them.

Figure 6.9. RESTalk vs. Standard BPMN Choreography per sector



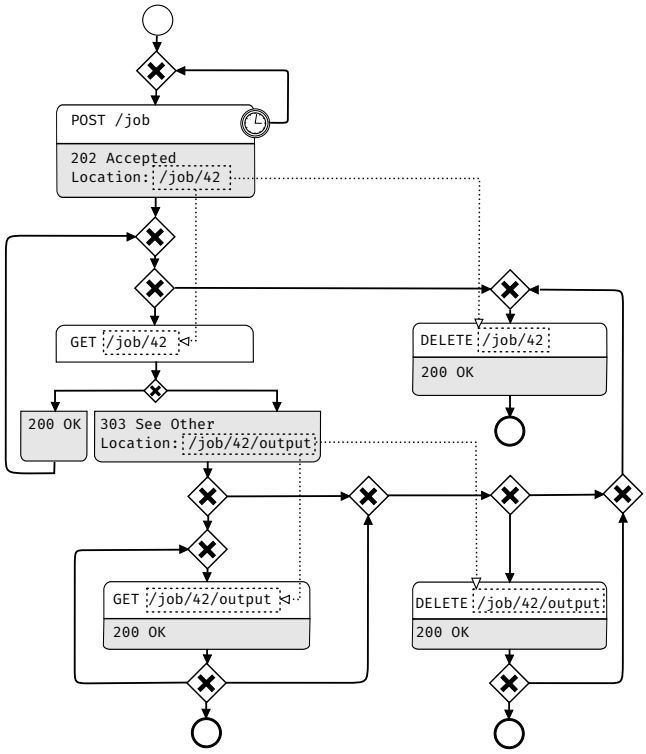
Reading task

For the reading task we have applied RESTalk to a common conversation that can be found in different REST APIs, e.g., Amazon Glacier's API for long term storage of infrequently used data[2]. Fig. 6.10 shows the diagram included in the survey. It depicts a RESTful conversation where the retrieval of data is turned into a job resource in order to avoid the client having to keep the connection open for too long, while waiting for the data retrieval. The client can keep on polling for the job output, and will only get redirected to it when the job has finished. Then it can read the job output multiple times, or it can decide to delete it. The job itself can be deleted at any point, thus either implicitly stopping the job, if it has not finished yet, or deleting the finished job since it is no longer necessary.
This group of questions followed after all the respondents got introduced to the basics of RESTalk. To test their ability to read diagrams modeled in RESTalk, we have posed the following multiple choice questions with one correct answer (marked in bold):

1. How many resources are created during this conversation:
   - none
   - one
   - two
   - **one or two**
   - more than two
2. What happens when you try to access the job resource while the job has

[2] `http://docs.aws.amazon.com/amazonglacier/latest/dev/job-operations.html`

Figure 6.10. Long running request conversation modeled with RESTalk



not completed yet:

- you get a 200 OK status code and a placeholder link to where the output will be saved once the job has completed
- **you get a 200 OK status code and can try to access the job again later**
- you cannot send a GET job request before the job has completed

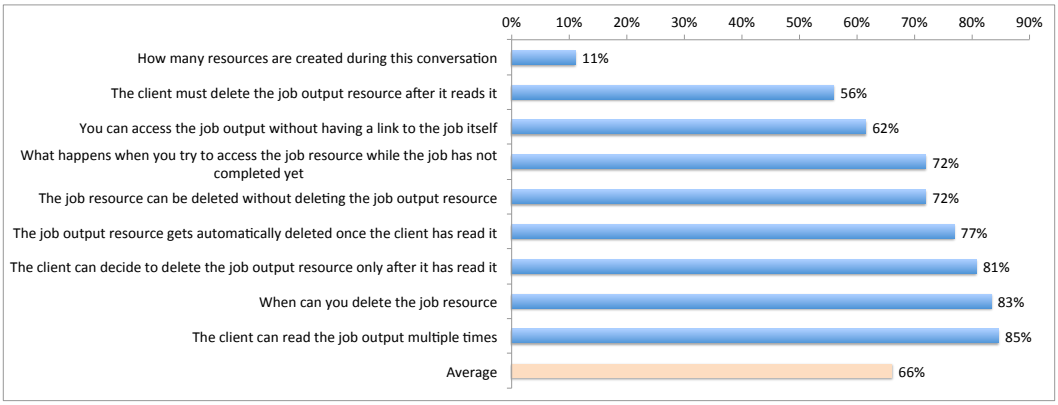
3. When can you delete the job resource?
   - only after the job has completed
   - only before the job has completed
   - only after having read the output
   - only before having read the output
   - only after the job has completed and you have read the output
   - only after the job has completed, but before you have read the output
   - **at any time after the creation of the job resource**

We have also asked the following true/false/I don't know questions (the correct answer can be found in brackets):

1. You can access the job output without having a link to the job itself (**false**);

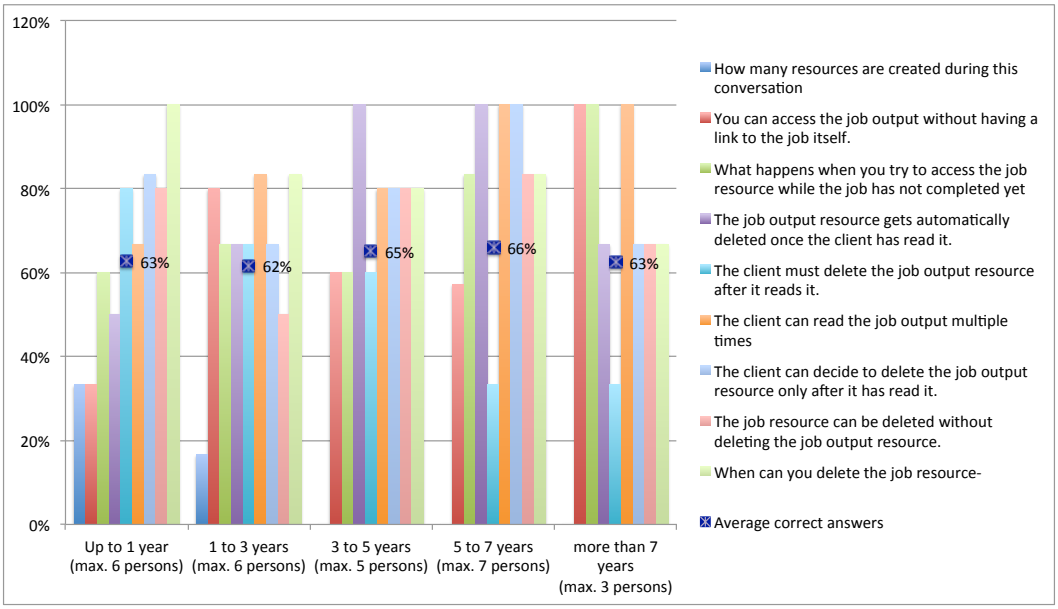
2. The job output resource gets automatically deleted once the client has read it (**false**);
3. The client must delete the job output resource after it reads it (**false**);
4. The client can read the job output multiple times (**true**);
5. The client can decide to delete the job output resource only after it has read it (**false**);
6. The job resource can be deleted without deleting the job output resource (**true**).

Figure 6.11. Assessing the reading of RESTalk diagrams (questions are ordered by percentage of correct answers)



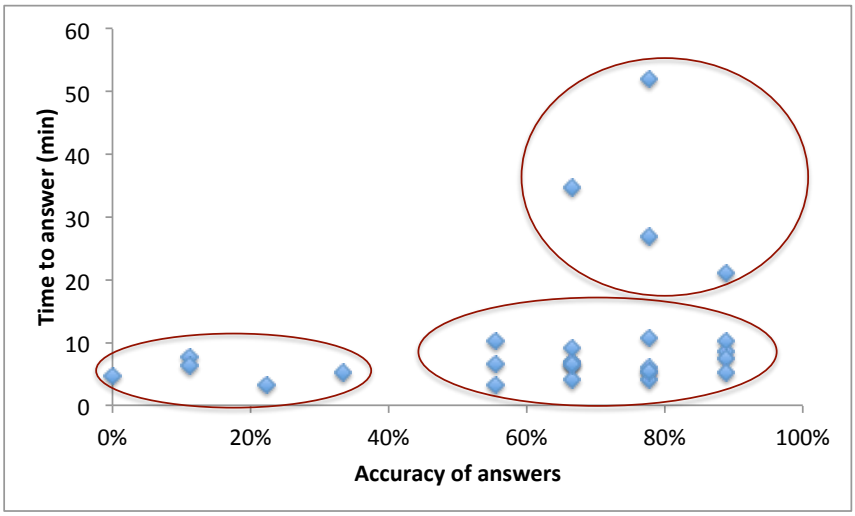
As evident from Fig. 6.11, all of the questions were answered with more than 50% of accuracy, except the question regarding the number of resources created during the conversation. Since the job can be deleted even before it is finished, we considered "one or two" as a correct answer, i.e., either only the job resource or both the job resource and the job/output resource are created. However, as per the discussion with some of the respondents, probably many of them (63%) consider the job/output a sub-resource of the job resource, and thus have identified "one" as the correct answer. Hence, if we regard this question as a conceptual rather than modeling question, and therefore discard it in the calculation, the average number of correct answers rises from 66% to 73%. On sector level, this indicator is much higher in academia (86%) than in industry (68%), while on BPMN knowledge level, it is higher for respondents with no previous BPMN knowledge (76%) than for respondents with basic BPMN knowledge (70%).
While in the intuitiveness group of questions, the impact of experience on the correctness of the answers was evident, this is not the case in the reading task group of questions, possibly due to the greater complexity of the modeled reality. No meaningful differences among experience groups have been noticed in this

Figure 6.12. Assessing the reading of RESTalk diagrams from respondents' experience perspective



case. Fig. 6.12 shows the results as well as the maximum number of respondents per group.

Figure 6.13. Correlation between time to answer and accuracy of answers



Although the correlation coefficient between time to answer the reading task questions and accuracy of the answers is not high (0.25), a scattered plot of

such correlation (Fig. 6.13) reveals several clusters of respondents. Those who rushed through the questions without paying too much attention (accuracy less than 40% in less than 10 minutes), those who understood well RESTalk and/or the REST concept and answered the questions quickly and accurately, and those who took their time to reason to get to the correct answers (accuracy higher than 70% in up to 50 minutes).

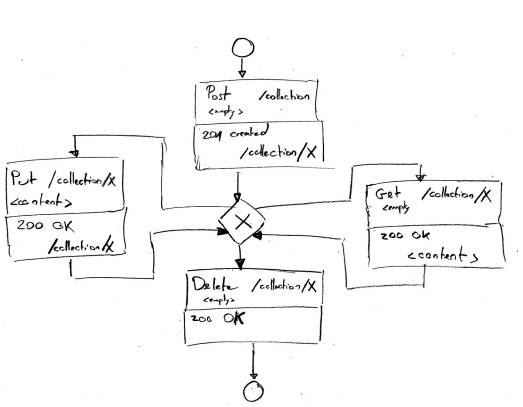
Modeling task

We have asked the respondents to use RESTalk to model the life-cycle of a collection item, i.e., all the possible CRUD (Create, Read, Update, Delete) operations that can be performed on the same resource.
Only three of the respondents, all from academia, have performed the task. They have all modeled the reality from the server's perspective, i.e., using just one end event after the resource is deleted. Thus, they made the simplification of only modeling with an end event a conversation which could never be resumed in the future, and abstracting from situations where the client ends the conversation without deleting the resource. They have all correctly used the interaction constructs with the request-response content, and two of them have also used the hyperlink flow element. The exclusive split and join gateways have been merged in one combined gateway by two of the respondents (Fig. 6.14a and Fig. 6.14b), while the third respondent made a mistake in merging the control flows, which resulted with an infinite loop between reading and updating the resource (Fig. 6.14c). The fact that 2 out of 3 respondents have decided not to use a tool to perform the modeling task, indicates the appropriateness of RESTalk for both whiteboard discussions and API documentation. Respondents' personal opinion on the matter is presented in Sec. 6.1.3.

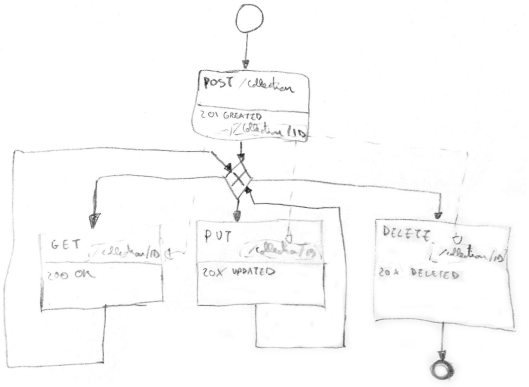
RESTalk's evaluation

After having used RESTalk, to understand or model a RESTful conversation, we have posed open-ended questions to assess whether it is considered easily understandable, and how it stands in terms of conciseness and efficiency compared to the textual or visual notation that the respondents had used before. To give some structure to the obtained answers, we have classified them in the groups shown in the graphs (Fig. 6.15 to Fig. 6.18).
83% of the respondents have a positive view on the understandability of RESTalk, finding it easy or somewhat easy to understand. One IT consultant stated that it is "surprisingly easy". While almost all the respondents from academia find it

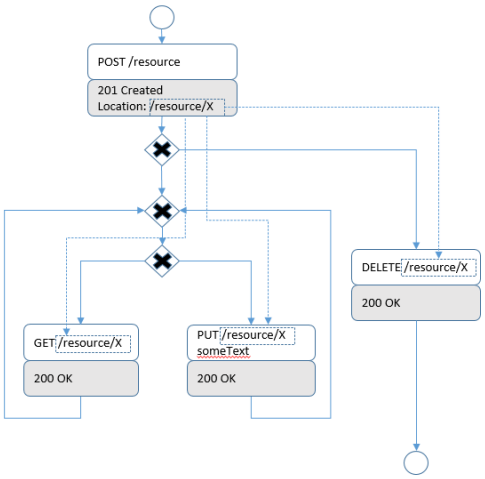
Figure 6.14. Respondents' models of CRUD operations on a collection item



(a) Respondent A



(b) Respondent B



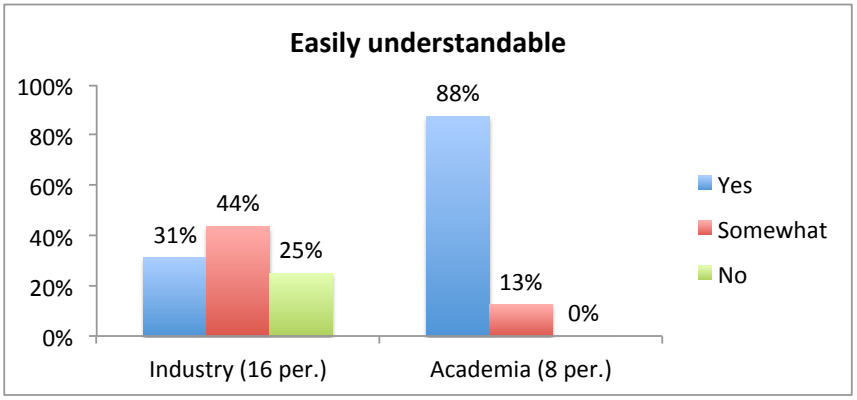
(c) Respondent C

decisively easy to understand, industry respondents seem to be more reluctant (Fig. 6.15). The ones that did not find RESTalk easy to understand, did not provide further details for such evaluation, while others expressed doubts of its understandability with more "complex conversations".

57% of the respondents find RESTalk less time consuming than the notation (textual or visual) they had used before (Fig. 6.16)[3]. All the ones who found RESTalk more time consuming are respondents who have not used visual notations be-

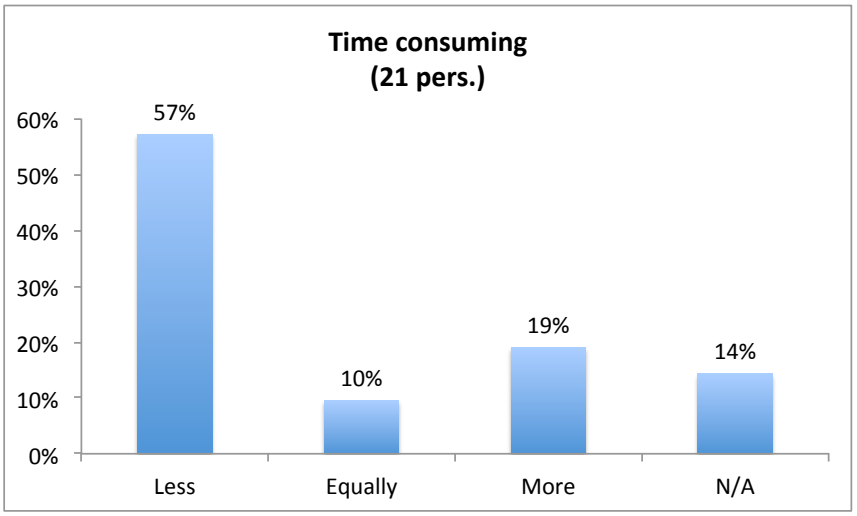
[3]N/A stands for persons who did not use any notation before and thus could not make the comparison

Figure 6.15. Assessing RESTalk's understandability per sector



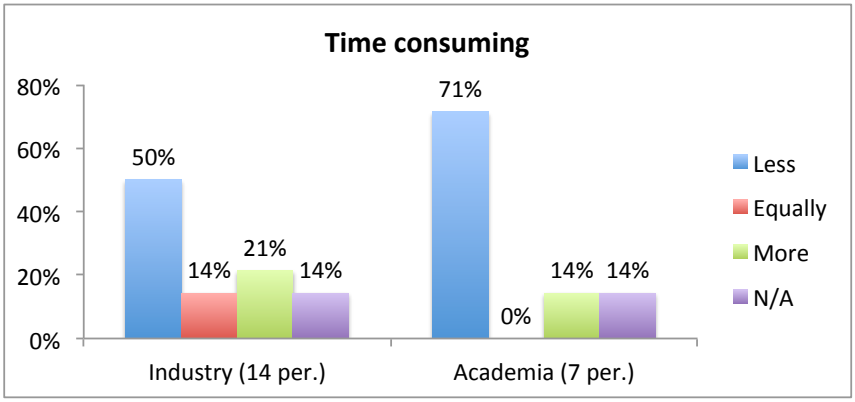
fore to depict RESTful conversations. Thus, it is reasonable that they consider drawing a diagram more time consuming than textual description. An IT consultant states that a "prerequisite for easy usage is that the graphical notation can be derived from a simple textual notation". As expected, given the results from the understandability assessment (Fig. 6.15), respondents from industry are more skeptical than in academia (Fig. 6.17), however the differences are not as evident as in the understandability question.

Figure 6.16. Assessing RESTalk's efficiency



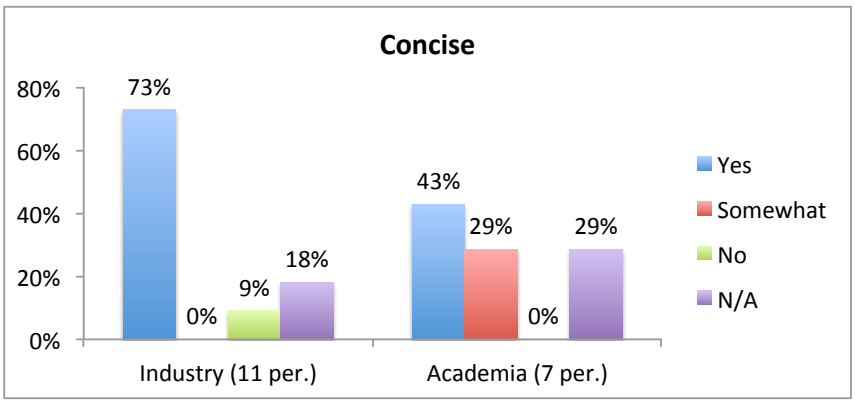
Of those 18 who responded to the conciseness question, 61% consider RESTalk more concise. One respondent states that it "probably depends on the graph and the number of responses at a given request". When analyzing this indicator from

Figure 6.17. Assessing RESTalk's efficiency per sector



the sector viewpoint, contrary to the previous results, in this case it is the industry respondents who are more united in their positive judgment about RESTalk (Fig. 6.18).

Figure 6.18. Assessing RESTalk's conciseness per sector



If we define positive sentiment about RESTalk as an opinion that RESTalk is superior or equal to existing notations, then a rather high percentage of support is present in all three indicators (understandability - 83%, efficiency - 67%, and conciseness - 72%).

Regardless of the above disclosed assessment, all respondents but two are willing to try using RESTalk in their own projects. One freelance software engineer states: "it will take some time to get familiar with but I think we could boost the development process even more". An IT consultant, on the other hand, conditions its acceptance to the availability of a specific tool: "If I could simply generate it from a textual description of the graphs, I would use it". 69% of the

respondents seem to have preference for a modeling tool support for RESTalk to facilitate the drawing, although some do agree that simple diagrams can also be drawn manually on whiteboards or using existing drawing tools. One also proposes to develop a tool for automatic extraction of the diagrams from code.
When asked whether they find the HTTP details (methods, URIs, status codes, links) sufficient for understanding what the RESTful conversation is aiming at, 71% have responded affirmatively. Some of the proposed HTTP elements to be added include authentication details as well as "some additional headers and parameters".
Interesting remarks have emerged from open-ended questions on RESTalk missing elements or ambiguities, as well as RESTful conversations used in practice which the respondents find challenging or impossible to model with RESTalk. Although the BPMN modeling best practice guidelines [8, Chap. 2] advise not to use combined gateways (one gateway as both a split and a join), several respondents have stated that multiple gateways create unnecessary "visual and mental distractions". In BPMN, exclusive gateways are data based decisions, so some considered it confusing not having the precise data conditions defined in the gateway. Some have also emphasized that RESTalk lacks details regarding the length of the timer or the maximum number of iterations allowed in a loop. Also people who deal with applications where the state changes are crucial, have noticed that the state transition is not explicitly marked in the current version of RESTalk. Another constructive remark is that currently it is not clear when there is a dependence between resources (e.g., a resource and a sub-resource). This is an important structural aspect, since updating or deleting the resource might result with the automatic update or deletion of the sub-resource.

6.1.4 Discussion

The purpose of this empirical study was to obtain the first cycle of feedback on RESTalk, the modeling notation we are proposing, in order to guide further improvements. Although 35 respondents might not seem high as an absolute number, we consider the constructive remarks we have obtained as truly valuable for our future work.

Need for a Domain Specific Language

Despite of the fact that currently there is no domain specific notation for modeling dynamic aspects of REST APIs, such as RESTful conversations, 38% of the respondents have already used some existing UML standard notation, or an in-

house developed notation, to depict sequences of client-server interactions. As a CTO indicates, a notation is "helpful for explaining concrete scenarios. It permits a 'behind the scenes look' and thereby understanding of consequences". This indicates that there is an identified need for visualizing RESTful conversations in practice. However, even the currently most widely used notation, the UML Sequence diagram, has its limitation when it comes to depicting RESTful interactions. One programmer names the following challenges when using UML Sequence diagrams: "If you want to know more about an object than just if it's engaged (active) in an interaction or not, this can be difficult to represent on a sequence diagram and it can make the diagram too complex and unreadable. Another case when a sequence diagram can become too complex very quickly, is when we have multiple objects and object lifelines that we wish to represent, and the interactions between the objects are too convoluted to read". Our goal with proposing RESTalk as a domain specific language is exactly to overcome such limitations of the existing standards.

Evaluation of the Cognitive Characteristics of RESTalk

Since this was an exploratory survey, with no intention of making statistical inference based on the same, we have only tackled some of the cognitive dimensions defined in [73], which we considered the most important in this initial stage of RESTalk's design. Namely, we have addressed the consistency, the closeness of mapping and the abstraction gradient dimensions.
We have taken intuitiveness as a broader approach to what is considered consistency in [73], i.e., "When some of the language has been learnt, how much of the rest can be inferred?". Without providing any prior information about RESTalk, it was encouraging to see that most of the respondents understood great part of the presented diagram and answered correctly to 77% of the multiple choice questions. In some cases, we have realized that prior knowledge of BPMN causes certain degree of confusion, particularly in circumstances where, due to the REST architecture domain characteristics, there is a slight difference in the semantics between the standard BPMN Choreography and RESTalk. For instance, while in BPMN Choreography exclusive gateways are data based and explicitly showing the decision point options is necessary, in RESTful interactions client's conditional decisions can be abstracted from. However, the respondents' feedback has made us realize that including the rationale behind server's decisions could indeed facilitate understanding under which condition each alternative response can be expected. It is also encouraging to see that even respondents with no prior BPMN knowledge have high average score of 76% of correct answers on

the reading task.
The results of what is defined in [73] as a "closeness of mapping" between the notation and the problem world, are both positive and beneficial for future improvements. Namely, when asked to compare RESTalk to the BPMN Choreography, all of the respondents have found it at least equally, if not more concise and expressive. On the other hand, when asked to compare it to the notation they are currently using only one person found it less concise, while 61% found it more concise than what they are currently using. This is an encouraging result, since it involves the first iteration of the design of RESTalk. However, what has been particularly beneficial for us is that the respondents have provided several examples of conversations which cannot be modeled with the existing elements in RESTalk. For instance, conversations where the state transitions are important for the conversation logic, and should thus become explicitly visualized. Thus, we have implemented this in the extended version of RESTalk. Likewise, in some cases the number of maximum retries for accessing a temporary unavailable resource are limited and need to be explicitly specified.
Another important cognitive characteristic of a notation, which we were interested in, was the "abstraction gradient", i.e., the levels of abstraction. While, as stated in Sec. 4.2.2, we have abstracted from some details when designing RESTalk, we have intentionally omitted to state the assumptions and simplifications behind such abstractions in the tutorial provided in the survey. We wanted to evaluate whether some abstractions are excessive, and thus respondents would identify them as missing elements or vice-versa. The modeling task, for instance, has revealed that the respondents have taken the end event simplification even further. While we abstract from modeling an end event after every server response, which is the real world description, they have opted for only modeling one end event to show that the conversation can never be continued in the future. On the contrary, we have used end events to show all the potentially successful conversation logs that can occur in a given conversation. The reading task, on the other hand, has disclosed that RESTalk lacks the expressiveness in showing the dependencies between resources, or that the length of the timer is not specified. Regardless of such findings, the understandability of RESTalk reaches a fairly high level of 66% based on the correct answers of the reading task multiple-choice questions, and an even higher level based on the respondents' subjective opinion, where 83% have evaluated RESTalk as easy or somewhat easy to understand.
To conclude, the predominantly positive sentiment regarding general understandability, conciseness and efficiency of RESTalk per se, or in comparison to standard BPMN Choreography or other visual notations, and above all the will-

ingness of 88% of the respondents to put RESTalk into practice, gave us the assurance we have sought to continue improving RESTalk and dealing with the modeling challenges it poses.

6.1.5 Threats to Validity

Regardless of the seemingly encouraging results, and the exploratory nature of the survey itself, we are aware of the threats to the validity of the results and the conclusions caused both by the small sample size of 35 respondents and the fact that they might have been aware that the survey is conducted by us, the designers of RESTalk, who some of them did know personally which might have biased their opinion, both when grading the language and when providing the feedback. To mitigate this risk we shared the survey on different targeted relevant social media groups and conferences where people are less likely to know us personally, however obtaining respondents through non-personal contacts is never easy. Another thereat to the construct validity is that the survey was designed to study both the need of a visual DSL for modeling RESTful interactions, and the characteristics of the proposed DSL, which might have biased the respondents towards accepting a solution to a problem they were not even aware of prior to the study. Furthermore, the design decision to leave most of the answers optional in order to incentivise participation, also poses a threat to the validity of the results as respondents could simply skip questions if they did not want to express their opinion on the matter or if they were in a hurry, leaving us with less answered questions towards the end of the survey as evident in Fig. 6.3.

6.2 RESTalk Expressiveness

In this section we will focus on the expressiveness of RESTalk which we have evaluated and iteratively augmented through different use cases. Expressiveness is an important quality of a language, and it depends on the suitability of the language to express a set of facts. “Expressiveness is defined as the degree to which a given modelling language is capable of denoting the models of any number and kind in a certain domain” [179]. That said, assessing RESTalk’s expressiveness requires to use the DSL for a variety of non-trivial RESTful conversations. To that end in this section we start by discussing different RESTful conversation patterns which we visualize using RESTalk in Sec. 6.2.1. As in the Language Requirements layer in Sec. 4.1.2 we have defined three types of targeted RESTful conversations whose modeling should be supported by RESTalk, we continue the

expressiveness evaluation by providing use-case examples for each targeted type of conversation: one client - one server conversation (Sec. 6.2.2), multiple clients - one server conversation (Sec. 6.2.3), and composite conversation (Sec. 6.2.4). We have used the patterns and use-cases to iteratively expand and improve the DSL and its expressiveness.

6.2.1 Modelling RESTful Conversation Patterns

A widely accepted technique for evaluating the expressiveness of a given modelling language is using the language for modelling design patterns [226; 66; 55]. "Design patterns, in the context of software development, provide generalized approaches and guidance to solving commonly occurring problems, or addressing common situations typically informed by intuition, heuristics and experience." [168]. The first patterns in software engineering referred to human-computer interaction and user interface design [31]. With the rise of Web services, the need for capturing system to system interaction has emerged. Hohpe and Woolf present patterns for enterprise integration through asynchronous messaging in [86], while Barros et al. [17] present patters for web-services integration mainly using BPEL and WSDL as implementation languages. They do not deal with using REST for enterprise integration. However, as the number of REST APIs is growing and software engineers are gaining experience in designing them, it is important to capture and share that experience to foster APIs' quality and usability. Patterns have emerged as an efficient method for attaining that goal [186]. Daigneau's patterns for Web service API design and implementation deal with REST-related aspects, such as the importance of the HATEOAS constraint in the "Linked service" pattern [34, p. 77]. However, he does not describe different interaction patterns that stem from this constraint. Such patterns can be uncovered from the request-response messages used to solve common RESTful design problems in [7; 174; 205].

Silvia Schreier, an experienced REST APIs designer at innoQ, expressed interest in a joint work on identification of RESTful conversation patters, which was a good opportunity for us to put RESTalk on test to see whether it can express visually the identified patters. Although visualization is considered an optional element in pattern description, it can significantly help people in grasping complex problems. Thus, using RESTalk for the visualization was appropriate given the domain. This joint work resulted in the publication of a pattern language for RESTful conversation patterns: abstract templates for simple, often recurring, conversations between one client and one REST API [161]. While Silvia Schreier was mostly involved in the identification and definition of the patterns, the RE-

STalk visualization and polishing of the patterns' description was provided by us. Namely, as advised by Meszaros and Doble [134] for every pattern we presented: its name, a simple summary, the context to be considered when selecting it, a brief discussion of the problem it addresses including the forces that make it difficult, the corresponding solution modeled with RESTalk representing the conversation template, and its consequences, as well as some examples of pattern's known uses in practice. Where applicable we also stated possible variants of the pattern with high-level details of the possible extensions. While all such details are available in [161], in this section we only include a short description of each pattern and its RESTalk visualization.
The pattern language is organized following the life cycle of a resource, i.e., its create, read, update and delete (CRUD) operations, which for protected resources, may require proper client authentication and authorization. The basic patterns we include in the pattern language are divided in four groups: resource creation patterns (POST Once Exactly, POST-PUT Creation, and Long Running Operation with Polling), resource discovery patterns (Server-side Redirection with Status Codes, Client-side Navigation following Hyperlinks, Incremental Collection Traversal), resource editing patterns ((Partial) Resource Editing, Conditional Update for Large Resources), and resource protection patterns (Basic Resource Authentication, Cookies-based Authentication). These basic conversation patterns can also be composed into longer conversations, and thus provide useful abstractions to manage larger conversation's complexity.

Resource Creation Patterns

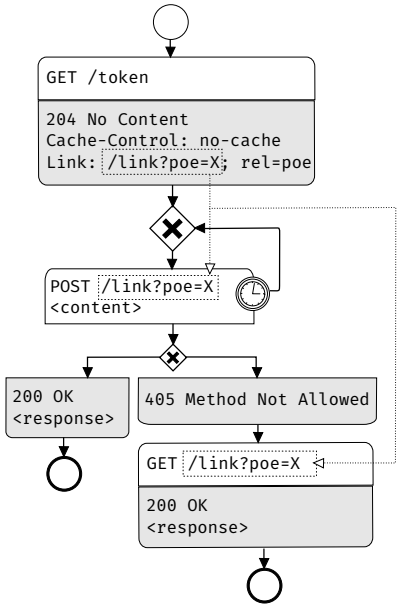
While HTTP offers the POST and PUT methods for creating resources in a single request-response round, the conversation patterns we present deal with resource creation under certain constraints or failure scenarios.

POST Once Exactly The goal of this pattern is to prevent creation of duplicate resources in case of errors. The proposed solution with the pattern is: the server should offer a resource where the client can retrieve a token, i.e., a unique target URI, for its request. As this unique URI is used when making the POST request, the server can check whether the corresponding resource already exists. The resource will only get created if this URI has not been used for a POST request before, otherwise, the server will respond with a message that the requested action is not allowed for this resource. If the server takes too long to reply, the client can decide to repeat the request without being exposed to the risk of creating the resource twice. The response to the POST request can either be received

directly (200) or in case of duplicate requests (405), and the client can fetch it with a GET on the same URI. A RESTalk textual and visual model of the solution is provided in Fig. 6.19.

Figure 6.19. RESTalk textual and visual model of the POST Once Exactly pattern

```
/// definition of the left-hand side instance
GET /token 204 /link?poe=X "Catche-Control: no-catche"
POST /link?poe=X "<content>" 200 "<response>"

/// definition of the right-hand side instance
GET /token 204 /link?poe=X "Catche-Control: no-catche"
POST /link?poe=X "<content>" 405
GET /link?poe=X 200 "<response>"

/// definition of the boundary event
((timer)) POST /link?poe=X "<content>"
POST /link?poe=X "<content>"
```

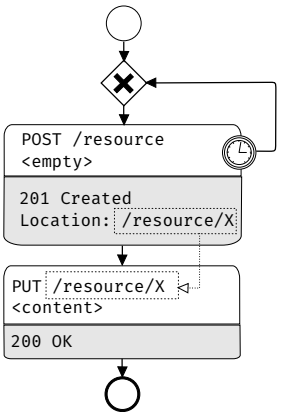


POST-PUT Creation This pattern shares the goal of the previous Post Once Exactly pattern, i.e., to prevent creation of duplicate resources in case of errors. However, it proposes a different solution. In this case it should be possible to

distinguish between the technical creation, i.e., the creation of a new URI, and the execution of the application domain specific creation behavior. The resource creation is split into two steps, the technical creation of its identifier and the actions that are required by the application domain. So the client sends first an empty POST request, which results in the creation of an empty resource resulting in no side-effect relevant for the application domain. Server's response contains a link to the URI of the created empty resource to which the client can add domain-specific content using a PUT request. The first PUT request will then trigger the consequences of the creation in the domain. Since the PUT is idempotent, resending it multiple times will not have side effects. A RESTalk textual and visual model of the solution is provided in Fig. 6.20.

Figure 6.20. RESTalk textual and visual model of the POST-PUT Creation pattern

```
/// definition of the main flow
POST /resource "<empty>" 201 /resource/X
PUT /resource/X "<content>" 200

/// definition of the exceptional boundary event flow
((timer)) POST /resource "<empty>"
POST /resource "<empty>"
```

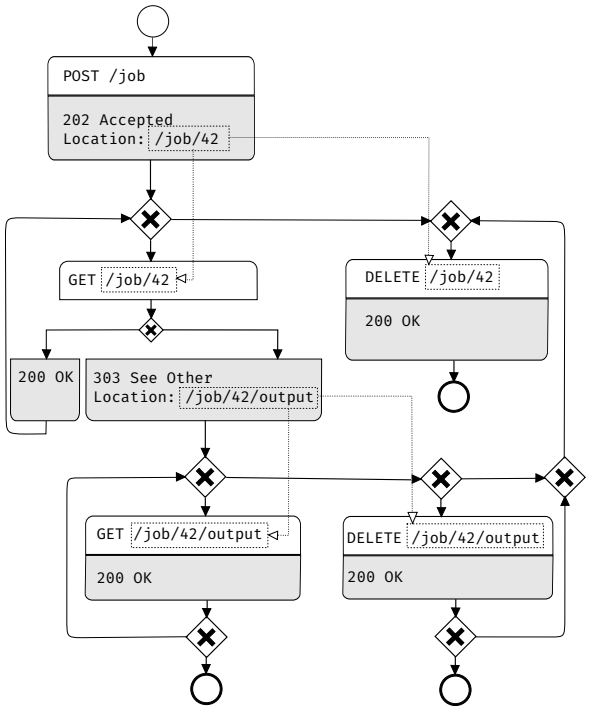


Long Running Operation with Polling The goal of this pattern is to avoid client timeouts when waiting for long running operation results. The proposed solution with the pattern is: the long running operation itself should be turned into a resource, created using the original request with a response telling the client

where to find the results. The results will be available once the operation running in the background completes. The client may poll the resource to GET its current progress, and will eventually be redirected to another resource representing the result, once the long running operation has completed. Since the output has its own URI, it becomes possible to GET it multiple times, as long as it has not been deleted. Additionally, the long running operation can be cancelled with a DELETE request, thus implicitly stopping the operation on the server, or deleting its output if it had already completed in the meanwhile. A RESTalk textual and visual model of the solution is provided in Fig. 6.21.

Figure 6.21. RESTalk textual and visual model of the Long Running Operation with Polling pattern [160]

```
POST /job 202 /job/42
GET /job/42 200
GET /job/42 303 /job/42/output
GET /job/42/output 200

POST /job 202 /job/42
DELETE /job/42 200

...
GET /job/42 303 /job/42/output
DELETE /job/42/output 200

...
GET /job/42/output 200
GET /job/42/output 200
DELETE /job/42/output 200
DELETE /job/42 200

...
GET /job/42 303 /job/42/output
DELETE /job/42 200
```



Resource Discovery Patterns

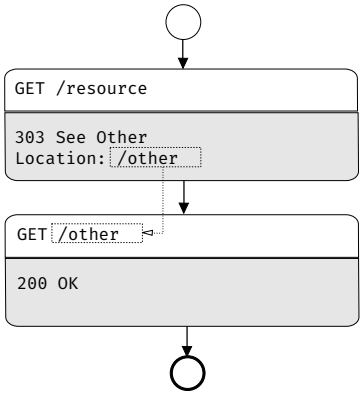
The HATEOAS constraint, mentioned in Sec. 2.2.1, promotes the design of APIs featuring a single entry point URI, and the dynamic resource discovery based on hypermedia. However, the entry point URI might not lead directly to the resource needed by the client, due to access rights, or the resource being moved

to a different location, or the resource being part of a collection of resources. The following patterns help to discover resources in such situations.

Server-side Redirection with Status Codes The goal of this pattern is to decouple clients from evolving resource locations. The proposed solution with the pattern is: if a client accesses a resource with an outdated URI the server should answers with a 3xx redirection status code usually in combination with a "Location" header to guide the client to the new URI. The client is then responsible for using this URI depending on the status code specification. A RESTalk textual and visual model of the solution is provided in Fig. 6.22.

Figure 6.22. RESTalk textual and visual model of the Server-side Redirection with Status Codes pattern

```
GET /resource 303 /other
GET /other 200
```

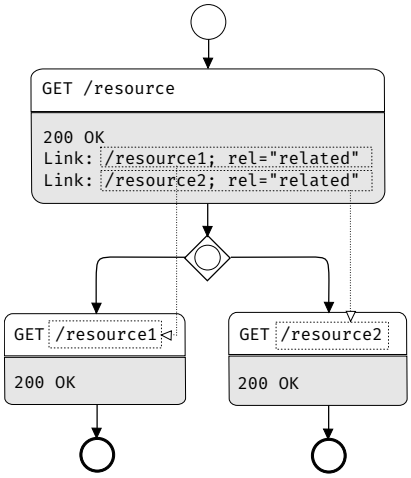


Client-side Navigation following Hyperlinks The goal of this pattern is to allow clients to choose from different navigation alternatives. The proposed solution with the pattern is: the server should provide all hyperlinks related to the requested resource such that the client can decide to follow one or more of the provided links, as depicted by the inclusive gateway in Fig. 6.23. It is important to note that, in addition to continuing with a GET request to a linked resource, other HTTP verbs can be used as well, depending on the semantics of the link relation and the intention of the client, e.g., like used in the (Partial) Resource Editing pattern. A RESTalk textual and visual model of the solution is provided in Fig. 6.23.

Incremental Collection Traversal The goal of this pattern is to use hypermedia to incrementally discover large collections. The proposed solution with the

Figure 6.23. RESTalk textual and visual model of the Client-side Navigation following Hyperlinks pattern

```
GET /resource 200 /resource1, /resource2
GET /resource1 200 || GET /resource2 200
```



pattern is: when a client requests the first item in a collection, the server should provide links to the next and the last item as well. Each following response to a GET request on a specific item in the collection, makes it possible for the client to select whether it wants to follow the link to the first, the previous, the next, or the last item, thus enabling it to gradually discover the collection by always following the link to the next item, or by moving back and forth using the provided links. To trade-off the size of each response against the number of interactions needed to traverse the collection, the right level of granularity needs to be determined, which can range from single items to pages (or groups) of multiple ones. A RESTalk textual and visual model of the solution is provided in Fig. 6.24. In order not to hinder the readability of the diagram, we have omitted the hyperlink flows which can easily be inferred from the sequence flows. This solution is based on the Client-side Navigation following Hyperlinks pattern.

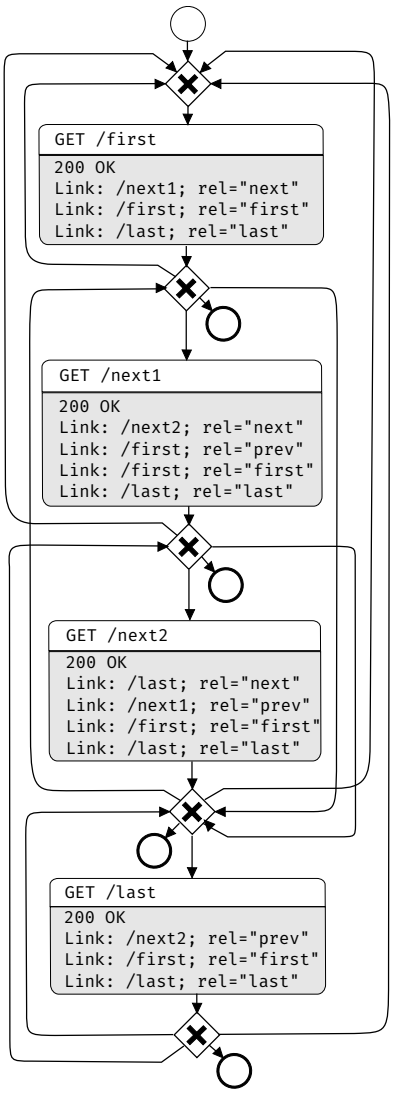
Resource Editing Patterns

The read-only Web is long gone and editing resources has become a common operation which can be performed using the patterns discussed below.

(Partial) Resource Editing The goal of this pattern is to use hypermedia to let the client discover how to update existing resources. The proposed solution with the pattern is: when responding to a GET request on an existing resource, the server should provide a link to a page with a form representing all the editable

Figure 6.24. RESTalk textual and visual model of traversals of a collection resource with four items (Example of the Incremental Collection Traversal conversation pattern)

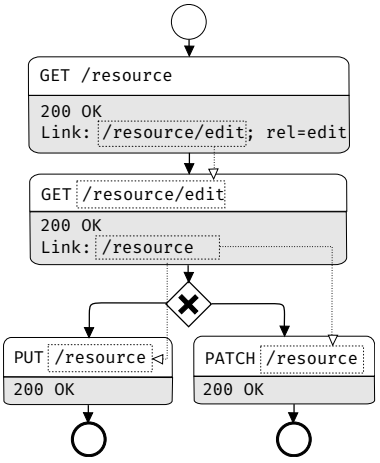
```
GET /first 200 /next1, /first, /last

GET /first 200 /next1, /first, /last
GET /first 200 /next1, /first, /last
GET /next1 200 /next2, /first, /first, /last

...
GET /next1 200 /next2, /first, /first, /last
GET /next2 200 /last, /next1, /first, /last

...
GET /next1 200 /next2, /first, /first, /last
GET /first 200 /next1, /first, /last

...
GET /next1 200 /next2, /first, /first, /last
GET /last 200 /next2, /first, /last

...
GET /next2 200 /last,/next1, /first, /last
GET /first 200 /next1, /first, /last

...
GET /next2 200 /last, /next1, /first, /last
GET /next1 200 /next2, /first, /first, /last

...
GET /next2 200 /last, /next1, /first, /last
GET /last 200 /next2, /first, /last

...
GET /last 200 /next2, /first, /last
GET /last 200 /next2, /first, /last
GET /first 200 /next1, /first, /last

...
GET /last 200 /next2, /first, /last
GET /next2 200 /last,/next1, /first, /last
```



content of the requested resource. The client can decide to update such content using a PUT request, thus overwriting the entire content of the resource, or using a PATCH request, thus sending an incremental update. The RESTalk textual and visual model of the solution is provided in Fig. 6.25.

Figure 6.25. RESTalk textual and visual model of the (Partial) Resource Editing pattern

```
GET /resource 200 /resource/edit
GET /resource/edit 200 /resource
PUT /resource 200

...
GET /resource/edit 200 /resource
PATCH /resource 200
```



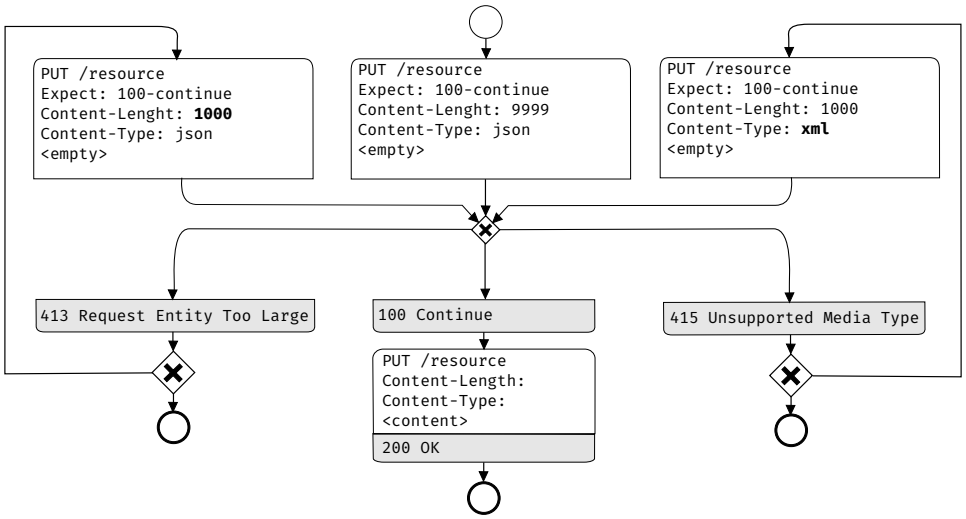
Conditional Update for Large Resources The goal of this pattern is to enable the client to declare its expectations to validate if the intended resource update is possible. The proposed solution with the pattern is: before sending the actual data, the client should send an empty body with an "Expect" and "Content-Length", or "Content-Type" or "Accept" header, which the server should use to control the appropriateness of the request to be sent. If the request is appropriate, evidenced by a 100 Continue server response, the client makes a PUT request with the same headers, except the "Expect" header, and the actual content. If retrieving another 4xx status code, the client can try with another content length or media type, suitable authorization, or end the conversation. The textual and visual RESTalk model for the solution is available in Fig. 6.26.

Resource Protection Patterns

Depending on the resource's content, some or all of the CRUD (Create, Read, Update, Delete) operations might be available only to a restricted group of clients. Client's access rights can be controlled using different patterns, including the ones presented below.

Figure 6.26. RESTalk textual and visual model of the Conditional Update for Large Resources pattern

```
PUT /resource "Content-Length:9999, Content-type: json" 413

PUT /resource "Content-Length:9999, Content-type: json" 415

PUT /resource "Content-Length:9999, Content-type: json" 413
PUT /resource "Content-Length:1000, Content-type: json" 415

...
PUT /resource "Content-Length:1000, Content-type: json" 415
PUT /resource "Content-Length:1000, Content-type: xml" 100
PUT /resource "<content>" 200
```

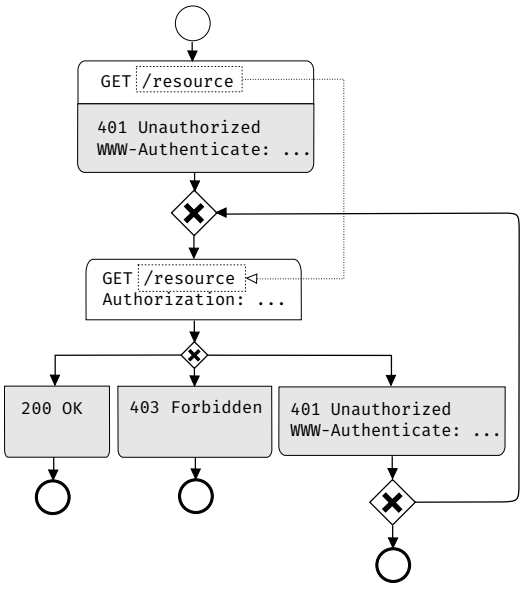


Basic Resource Authentication The goal of this pattern is to limit the access to authenticated and authorized users with basic HTTP authentication. The proposed solution with the pattern is: when a client requests to access a protected resource the server should reply with a 401 Unauthorized response thus challenging the client to provide authorization credentials. If the credentials are valid, the client is granted access, otherwise, it is challenged to provide credentials again or to end the conversation. Depending on the targeted clients this pattern can also be used in parallel with the Cookies-based Authentication pattern. This pattern is not only applicable to GET requests, but to every other HTTP verb as well. In case of large representations the Conditional Update for Large Resources pattern can be considered. A textual and visual RESTalk model of the solution is

provided in Fig. 6.27.

Figure 6.27. RESTalk textual and visual model of the Basic Resource Authentication pattern

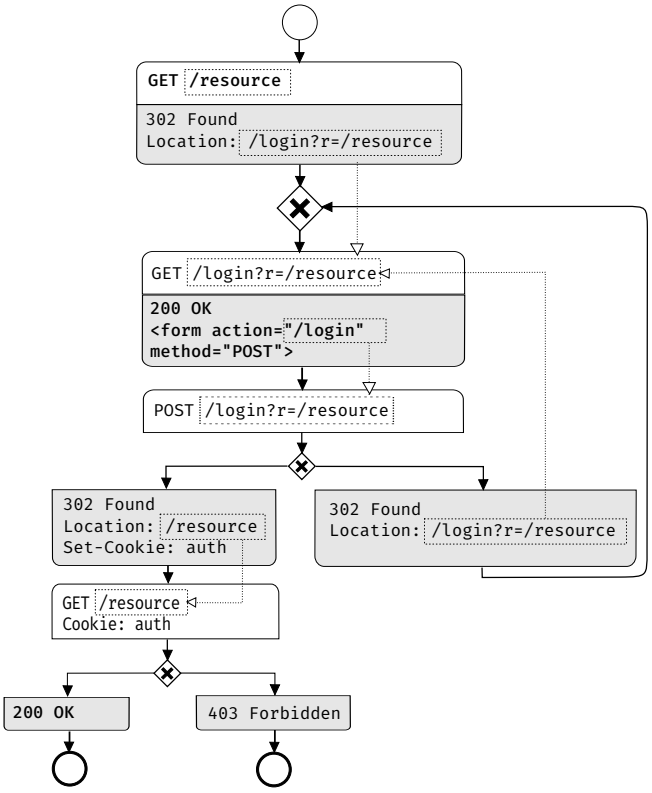
```
GET /resource 401
GET /resource Authorization 200

GET /resource 401
GET /resource Authorization 403

GET /resource 401
GET /resource Authorization 401

GET /resource 401
GET /resource Authorization 401
GET /resource Authorization 401
...
```



Cookies-based Authentication The goal of this pattern is to limit the access to authenticated and authorized users using cookies. The proposed solution with the pattern is: after the initial request for accessing a protected resource, the server should redirect the client (see the Server-side Redirection with Status Codes pattern) to a login page containing an authentication form that the client needs to fill in. If the login data is valid, the client is redirected to the initially requested resource and a cookie is set to be used in future requests, otherwise, it is redirected back to the login page. Depending on the targeted clients this pattern can also be used in parallel with the Basic Resource Authentication pattern. A textual and visual RESTalk model of the solution is provided in Fig. 6.28.
The visualization of the above described RESTful conversation patterns with RESTalk has provided us with an initial feedback on the expressiveness of the first version of RESTalk.

Figure 6.28. RESTalk textual and visual model of the Cookies-based Authentication pattern

```
GET /resource 302 /login?r=/resource
GET /login?r=/resource 200
POST /login?r=/resource 302 /resource "Set-Cookie: auth"
GET /resource "Cookie:auth" 200

...
POST /login?r=/resource 302 /resource "Set-Cookie: auth"
GET /resource "Cookie:auth" 403

...
GET /login?r=/resource 200
POST /login?r=/resource 302 /login?r=/resource
GET /login?r=/resource 200
...
```



6.2.2 One Client - One Server Conversation

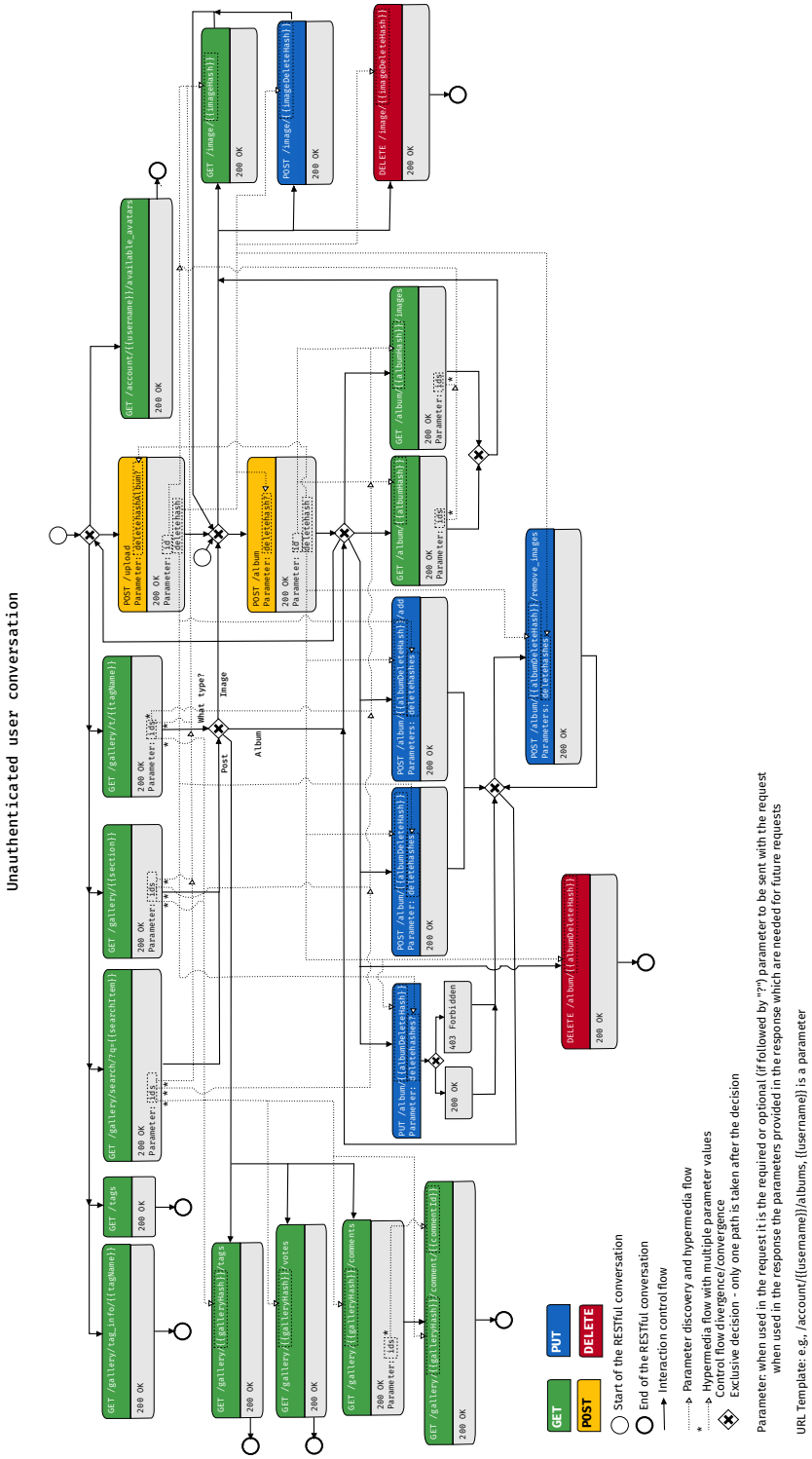
The initial target interactions of RESTalk as a DSL were conversations between a single client and a single server. In this section we present two such use cases: modelling the behaviour of a real REST API and modelling the behavior of REST APIs in a microservice architecture.

Modelling Imgur - a Real Life API

After applying RESTalk to visualise short design patterns as shown in Sec. 6.2.1, to test its ability to model the behaviour of larger APIs we have used it on Imgur[4], an image host and image sharing community which offers a REST API[5]. The API exposes most of the website functionality via a programmatic interface, thus API developers can use it to access content shared by other users, or publish personal content and manage it inside albums. The **gallery** contains the **images** and **albums** which are shared with the community. A user can also decide to upload content without sharing it with the community. Shared content can be tagged for easier searching, it can also be added to favourites, voted on or commented on by users. Thus, Imgur's API is comprised of five main resources: *image* and *album* to manage images and albums respectively, *gallery* to retrieve info and manage gallery posts, *comment* to manage the comments on the gallery posts and *account* to retrieve information relevant to a given user's account. Given the size of the API, which has over 80 endpoints, we have realized that visualizing its behaviour into one graph would be overwhelming. Thus, we have started thinking in terms of API use cases and came up with a logical separation of the possible API interactions depending of the feasible goal of the conversation. We believe that such logical separation in large APIs can help the API client developers to only focus on the endpoints of interest, instead of studying all the API endpoints, thus reducing the cognitive load of the API client developers. We have decided upon the following three visual RESTalk models: in Fig. 6.29 we show the interactions which can be performed by an unauthenticated user, in Fig. 6.30 we show the available interactions for an authenticated user for publishing and managing own content, while in Fig. 6.31 we show the available interactions for an authenticated user for discovering and interacting with other user's content.

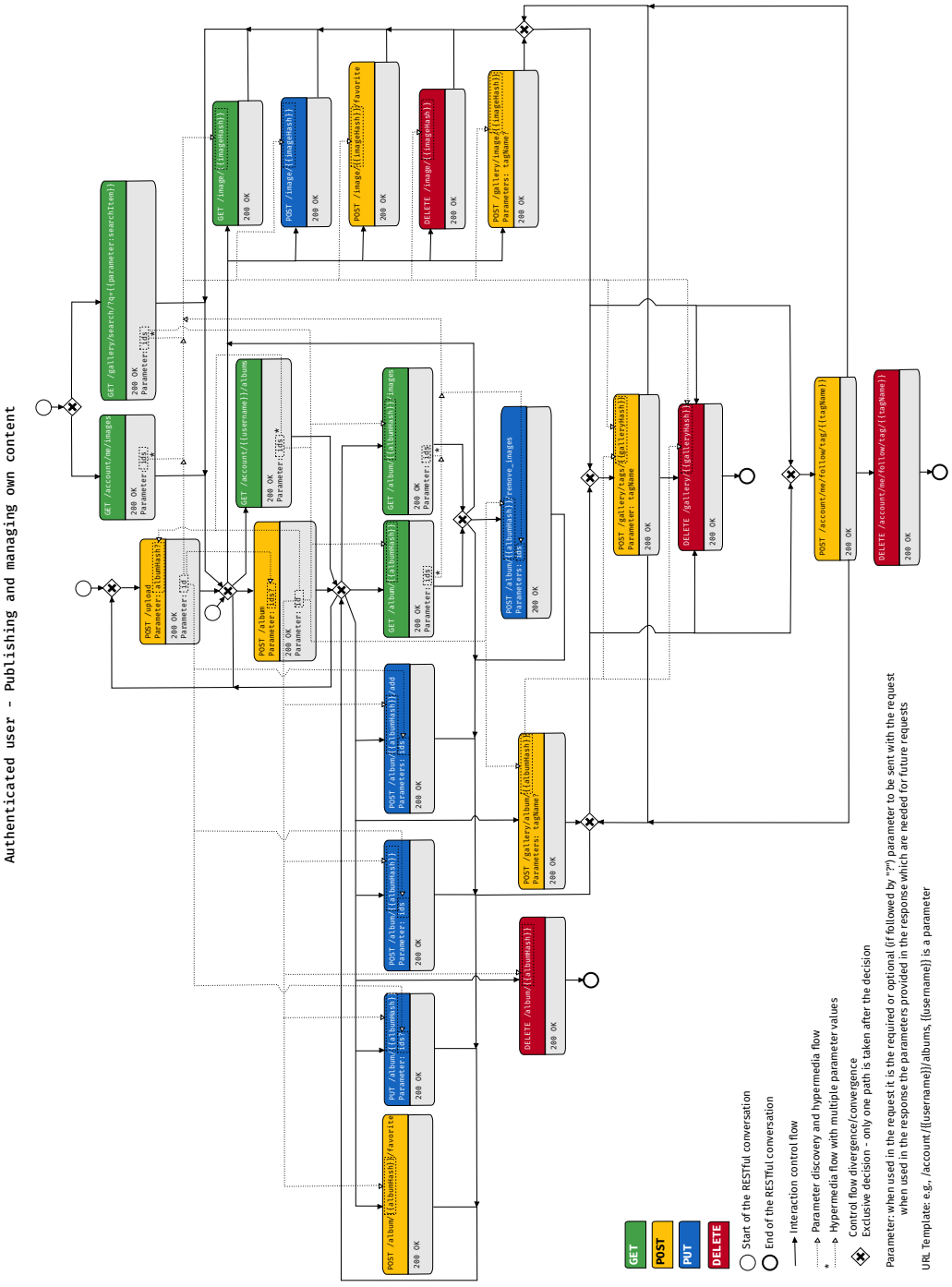
[4] `https://imgur.com`
[5] `https://apidocs.imgur.com`

Figure 6.29. RESTalk visual model for Imgur's API interactions with an unauthenticated user



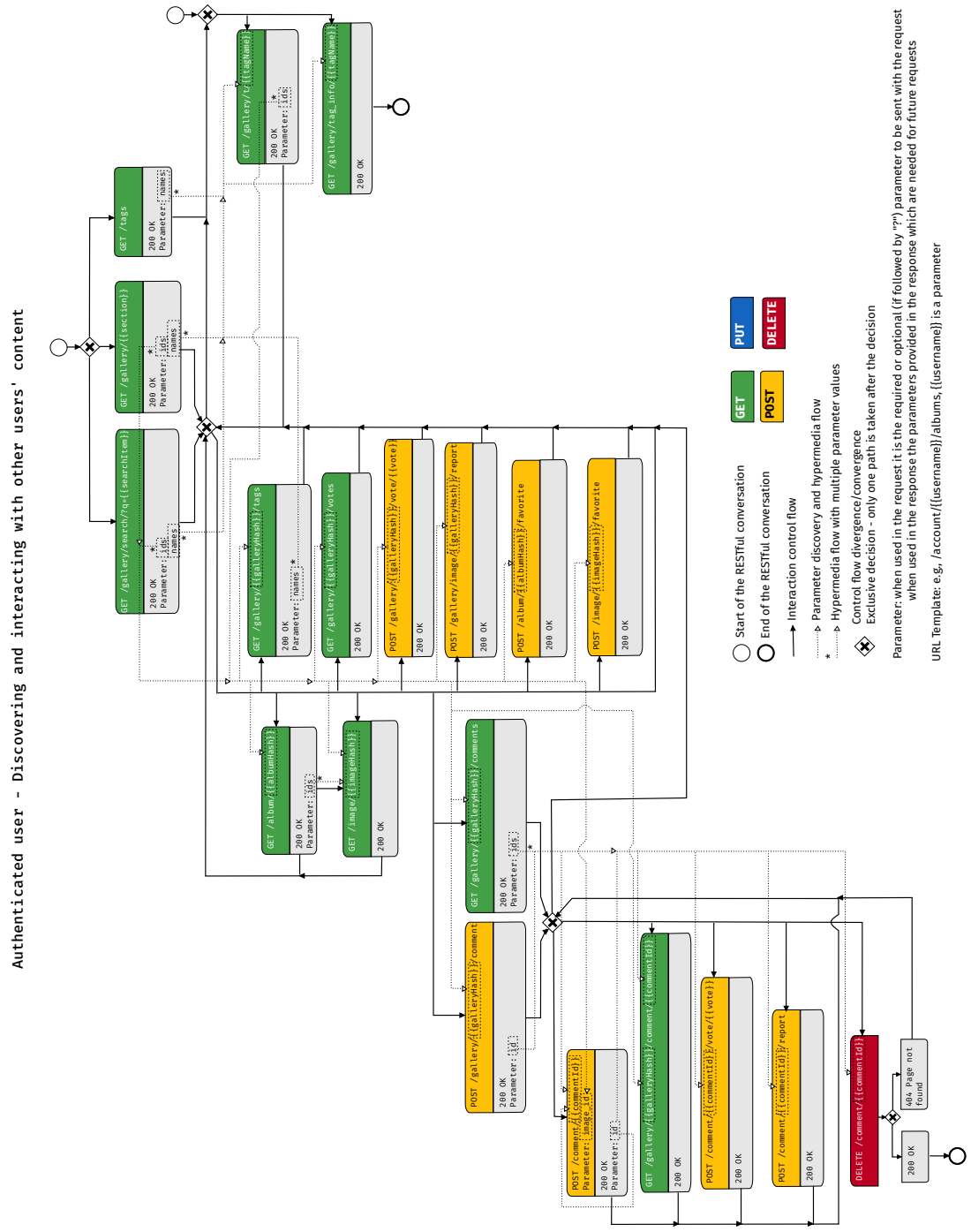
*Available at: `http://design.inf.usi.ch/restalk-experiment/ImgurUnauth.html`

Figure 6.30. RESTalk visual model for Imgur's API interactions with an authenticated user regarding publishing and managing own content



*Available at: `http://design.inf.usi.ch/restalk-experiment/ImgurAuth.html`

Figure 6.31. RESTalk visual model for Imgur's API interactions with an authenticated user regarding interacting with other user's content



*Available at: `http://design.inf.usi.ch/restalk-experiment/ImgurAuthExternal.html`

To test whether such division of the API into single models helps the understanding of the API behaviour we have conducted a controlled experiment described in detail in Sec. 7.3. After the feedback from the first experiment, to facilitate the readability of the behaviour we colored the requests based on the CRUD semantics instead of the method as we have noticed that the methods are not always appropriately used (for instance POST album/albumDeleteHash/add allows an unauthenticated user to edit an album resource by adding images to the same and as such should have used the PUT method). We have also added interactivity to the diagram providing short natural language description of the request when hovering over it in addition to links to the OAS documentation and we also added parameter coloring on hover over a parameter to show where the values of the parameter can be used for next requests, or how can we get the value of a given parameter we need to use in a request.
To show that reading such an interactive diagram can be beneficial with respect to reading the OAS documentation we use a simple example. For instance, let's imagine that the API client developer would like to use the service to share an image in the gallery. In the OAS documentation (s)he can easily find in the $gallery$ resource the correct endpoint to send a POST request to (/gallery/image/{{imageHash}}). But what might not be so evident is where to find information about the imageHash parameter. If (s)he knows the domain it can be logical to imagine that the imageHash parameter value would be included in the response when uploading a new image, but what might not be as evident is that there are also other endpoints which can provide the value of the parameter which are easy to detect in Fig. 6.30 by following the hyperlink flow or in a case of interactive diagram by hovering over the imageHashs parameter in the request for publishing the image to the gallery.
Of course creating such large diagrams as the ones in Figs. 6.29, 6.30, 6.31 is very time consuming and requires substantial cognitive effort both for the creation of the flow and of the layout. However, using the textual DSL in form of user stories fed into a mining algorithm as mentioned in Sec. 5.2 can facilitate such effort as it breaks the conversation into smaller chunks. Some of the user stories which can be told for the diagram in Fig. 6.30 are shown in Fig. 6.32. We do not state all of the user stories needed to create the diagram as we would not like to bore the reader, but we believe Fig. 6.32 renders the idea. Imgur uses the "hash" terminology only in the URIs ($imageHash$, $albumHash$, $galleryHash$), while in the response uses the term "id". This can be easily overcome when drawing the diagram manually as the user can connect with a hyperlink flow different terms. When using the textual DSL this has to be done algorithmically, which means that unique terms need to be used to express matching concepts both in

the request and in the response. Thus, in the example user stories in Fig. 6.32 we do not use the generic "id" term, but only the specific "hash" term.

Figure 6.32. Sample of textual DSL user stories for the Imgur API

```
/// upload an image
POST /upload 200 imageHash
///get an album to add it to
GET /account/username/albums 200 [albumHash]
///add the image to the album
POST /album/albumHash/add "imageHash" 200
...

/// upload an image
POST /upload 200 imageHash
///create an album to add it to
POST /album 200 albumHash
///add the image to the album
POST /album/albumHash/add "imageHash" 200
...

/// get uploaded images on my account
GET /account/me/images 200 [imageHash]
///add a selected image to the gallery
POST /gallery/image/imageHash 200
...
```

Modelling Interactions between Microservices

In recent years, there is a clear trend towards the micro-service architectural style where each component (i.e., micro-service) can evolve, scale and get deployed independently. This style increases the flexibility of the system and has been applied in demanding web applications such as eBay, Amazon or Netflix [128]. Splitting up a monolith into smaller scalable loosely-coupled microservices [233], requires defining an efficient way of communication between the newly created services. This is because, microservices are required to be technically self-contained, but not functionally self-contained, as they may interact with other microservices to provide their business functions [108]. Different integration technologies between microservices can be used [43], some supporting

synchronous request-response interactions, some asynchronous event-based interactions, and some using a mix of both [39]. The best approach depends on the use case [148], but in this use-case we will focus on the lightweight synchronous interactions built in accordance with the REST architectural style as it blends well with the microservices doctrine. Such synchronous communication frequently requires an orchestrated approach for driving the communication between the client and the microservices, seen as a conversation composed of multiple basic HTTP request-response interactions. One such orchestration approach is implemented by using the API Gateway pattern [172]. An API Gateway is a single entry point that provides access to different microservices [93]. It simplifies the client by moving the logic for calling multiple microservices from the client to the API gateway [123]. Namely, the API Gateway can be in charge, among other things, for service discovery, and thus act as a router and decide which resources of the microservices to access based on the nature of the client (e.g., mobile or desktop) and the received call (e.g., retrieving data, creating data, etc.) [39]. This decision can be rather simple when the microservices are independent from each other and thus they can all be called at the same time upon the request of the external client. So in this case the API Gateway simply fans out the requests to the related microservices. However, in real-world microservice architectures this is rarely the case, as frequently data provided from one microservice is needed to make the request to another microservice, thus requiring a predefined sequence of interactions. This emphasizes the importance of the knowledge gathering and the documentation of such sequences of calls to the microservices which are necessary to achieve a given client's goal.

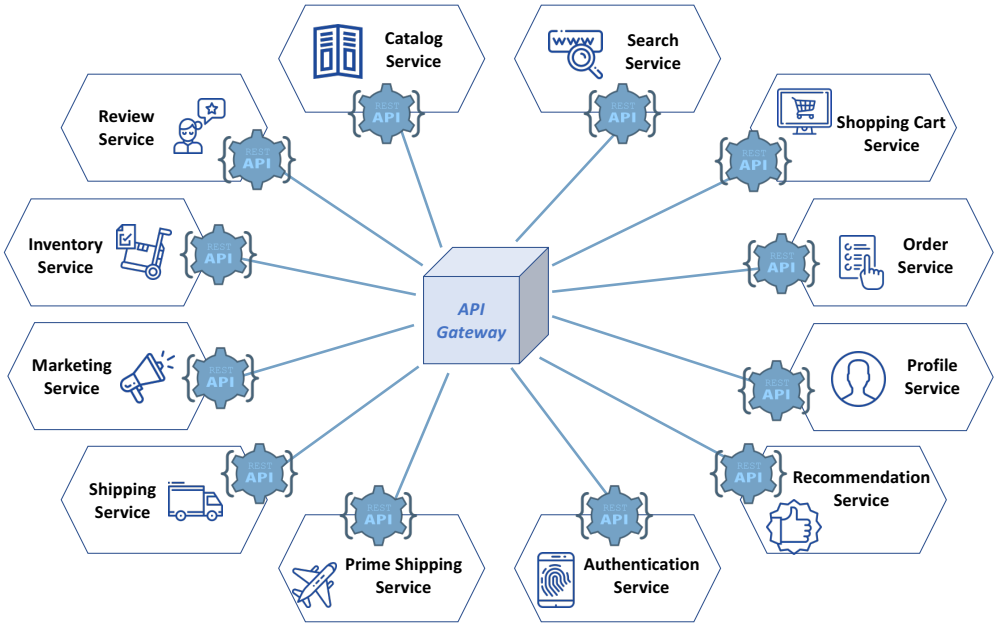
Big e-commerce companies are known for their microservice architecture [104; 172], and they do use tools to facilitate the design and documentation of that architecture. For instance, as eBay was restructuring its existing APIs into microservices, it adopted the OpenAPI Specification for documenting the structure of the REST APIs[6].

In this section we show how RESTalk can be useful for understanding and documenting the interactions between different microservices in the context of a microservice architecture. We argue that visualizing the inter-dependencies between the microservices requires the system designers to actively think about them, both when designing and when extending the system, and it empowers a discussion regarding the same with the interested parties, for instance the developers of the individual microservices. Having a DSL helps to capture all relevant,

[6] `https://www.openapis.org/blog/2018/08/14/ebay-provides-openapi-specification-oas-for-all-its-restful-public-apis`

REST specific details about the interactions.
In the microservices context, each microservice can be considered a resource, as resources in general terms are conceptual abstractions of any information or service that can be named. It is also possible that a microservice is comprised of multiple resources. What is important is that the server can create different *representations* of the same resource depending on the received request [148] and thus serve different clients differently.
Due to the frequent use of microservices in e-commerce companies [104], we have opted for the e-commerce domain to provide an example of the use of RESTalk inspired by Amazon. In our example we assume that the microservice architecture includes an API gateway, which means that the client makes a call to the API Gateway which in turn makes different calls to all of the relevant microservices. In Fig. 6.33 we show a possible microservice architecture of an e-commerce solution.

Figure 6.33. Microservice architecture of the example e-commerce company



As users can create their profiles for faster and more personalized shopping experience the *Profile service* stores all the relevant user data, such as address, contact details, the type of user, etc. User's authentication, log-in credentials and validity of the access token, is controlled by the *Authentication service*. User's orders are managed by the *Order service*, while draft orders, which have not been submitted yet, are managed by the *Shopping cart service* to which items can be added

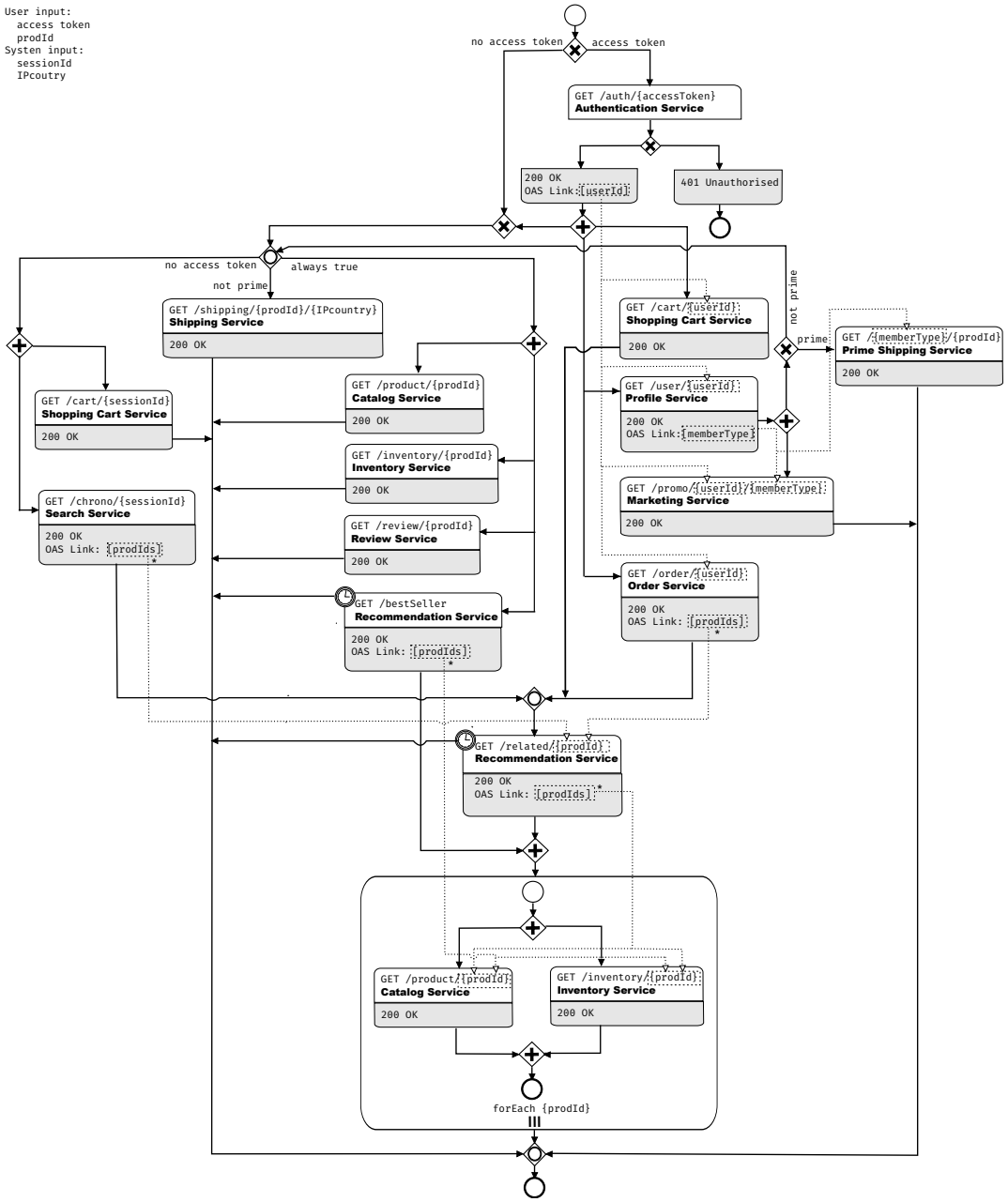
both by logged-in users and not logged-in users. The shipping of ordered items is managed by a call to an external provider noted as the *Shipping service* for non-prime users and *Prime shipping service* for prime users. Frequent users may also receive special discounts and promotions which are managed by the *Marketing service*. The *Search service* provides the searching functionality and stores session ids and product ids related to the session to later be used by the *Recommendation service* which provides the business logic over recommending certain products over others. All the details about a product, including its characteristics and price, are stored in the *Catalog service*, while the *Inventory service* handles the up-to-date information about the available quantity of a given product. Last but not least, the *Review service* stores and aggregates customer's reviews about a given product.

In a microservice architecture using an API Gateway there are two layers of communication. In the first layer there is the communication between the client and the API Gateway, abstracting from the existence of microservices, as the client would make the same calls also in the case of a monolith application. In the e-commerce example this would refer to a conversation between the client and the server which includes searching the web-site, looking at the products, adding them to the shopping cart and up until the placing and modification of an order. The second layer of communication refers to the interactions between the API Gateway and the REST APIs of the microservices, triggered by a specific client call in the first layer of communication. RESTalk can be used to represent any of the layers, however, in this use case we present a visual diagram of the conversation occurring within the second layer of communication.

In Fig. 6.34 we present the conversation that is triggered by the API Gateway as soon as a call for rendering a specific product item's web page is made, which in the e-commerce context, happens as soon as the user clicks on one of the items in the search results. We assume that when entering the home page of the e-commerce website the system stores the session ID and performs a geolocation query to determine the country based on the IP address. Thus, these two parameters, session ID and country, are already known when making the call for rendering the product item's web page. The input provided by the user when making this call is the product ID and optionally the access token. When there is no access token it means that the user is not logged-in, thus only the left part of the conversation diagram will be executed.

Most of the microservices can be called in parallel, as they only require the parameters that are already available at the start of the conversation. This is the case with the Catalog service, Inventory service, and Review service which only require the product ID. Note that these services will be executed even if the user

Figure 6.34. RESTful conversation for rendering a product item page



is logged-in as they are on the outgoing path of the inclusive gateway split which has a condition that is always true. The IDs of the best seller products provided by the Recommendation service will also be retrieved in parallel with the above mentioned microservices as no parameter is required for the call. For each of

the best selling product IDs the Inventory service will need to be called to check whether the product is available, and the Catalog service to check its price, before they can be rendered on the web page. The same sequence needs to be followed when generating the recommendations based on the search chronology for the user who is not logged-in, or based on the order history, for the user who is logged-in. Modelling such use case required us to add a new construct and a new marker to the core RESTalk, the sub-conversation with a multi-instance marker, which have been borrowed both as a syntax and as semantics from BPMN's sub-process with multi-instance marker. This allowed us to group the parallel calls to the Catalog service and the Inventory service and visually represented them as a sub-conversation which is executed for each product ID generated by any of the resources of the Recommendation service, as evident from the hyperlink flow visual construct.
The timer event on all the calls to the different resources of the Recommendation service will ensure that at least all the page data, except for the recommended products, is rendered in case the Recommendation service is slow (or down), as the Recommendation service just provides added value for the users, but is not crucial for the users to continue with their order.
The Shopping cart service is called both for logged-in users and not, using different parameters: the session ID for a non logged-in user, and the user ID for a logged-in user. While the session ID is available from the start of the conversation, the hyperlink flow visual construct shows that the user ID is obtained from the response of the Authentication service. This service, based on the validity of the provided access token, can send the user ID or a 401 status code if the token is no longer valid. As the Profile service stores the durable user data, when provided with the user ID it reveals whether the user is a prime member, and thus whether the shipping microservice or the Prime shipping microservice should be invoked to render the estimated shipping time and price on the web page. The Marketing service also uses the user ID and the membership type data to calculate different promotions available to the user. As this microservice requires data which is obtained from the Authentication service and the Profile service it cannot be called before receiving the response from both of these microservices.
As evident from the diagram in Fig. 6.34 each time a client makes a call to the API Gateway for rendering the page of a product item at least 5 calls to different microservices are made to render all the data for a non logged-in user, plus all the optional calls needed for making the recommendations.
For our e-commerce example some possible user-stories would look like in Fig. 6.35. The goal is to show the use of textual DSL constructs we have not seen used before, such as stating interactions in parallel and continuing the flow

after as well as stating the sub-conversation construct. These are examples where textual modeling is used as it is more efficient then deducing the constructs with the mining algorithm as discussed in Sec. 4.2.3. It refers to a part of the conversation instance when the user has valid log-in credentials and just states part of the parallel interactions that happen after it is authenticated, the ones leading to the Recommendation service. The other possible conversation instances that need to be stated in order to generate the diagram in Fig. 6.34, are not shown due to their length. However, they would follow the same syntax as the provided examples in Fig. 6.35.

To conclude, in microservice architectures it might be easy to reason about the behaviour of each individual component, but understanding the behaviour of the entire system can become rather complex [39]. That said, visualizing the communication flow between the microservices makes it possible to explain their mutual dependencies and interactions to newbie developers, and helps developers to document the interactions in the existing architecture from a behavioral viewpoint. While a RESTful conversation model complements existing structural models, together they can be used to discuss any possible extensions in terms of additional API usage scenarios. Furthermore, structured knowledge about the inter-dependencies between microservices can help to identify patterns and anti-patterns in this relatively new architectural style which still faces the issue of communication optimization [11]. On another note, having a precise communication model is a needed step for building automatic testing frameworks that test the communication behaviour of microservices [43].

The goal of the example is to show the expressiveness of RESTalk and its semantics, but also to facilitate the discussion of the potential benefits of visualizing the dynamic microservice communication. Namely, in the name of achieving better scalability, performance and maintainability, the microservice architecture introduces complexity in terms of microservices communication compared to a monolith architecture. Encoding the knowledge about such – unavoidable by design – interactions between the microservices helps in sharing that knowledge and leveraging it to induce the discussion and application of best-practices. Although this knowledge could be visualized and encoded also in existing general purpose languages, such as UML, using a domain specific language, such as RESTalk, helps to emphasize important facets of REST and the underlying HTTP protocol in the visualization. This work has been published as a book chapter in the book “Microservices, Science and Engineering” [91].

Figure 6.35. Sample of textual DSL user stories for the e-commerce microservice architecture

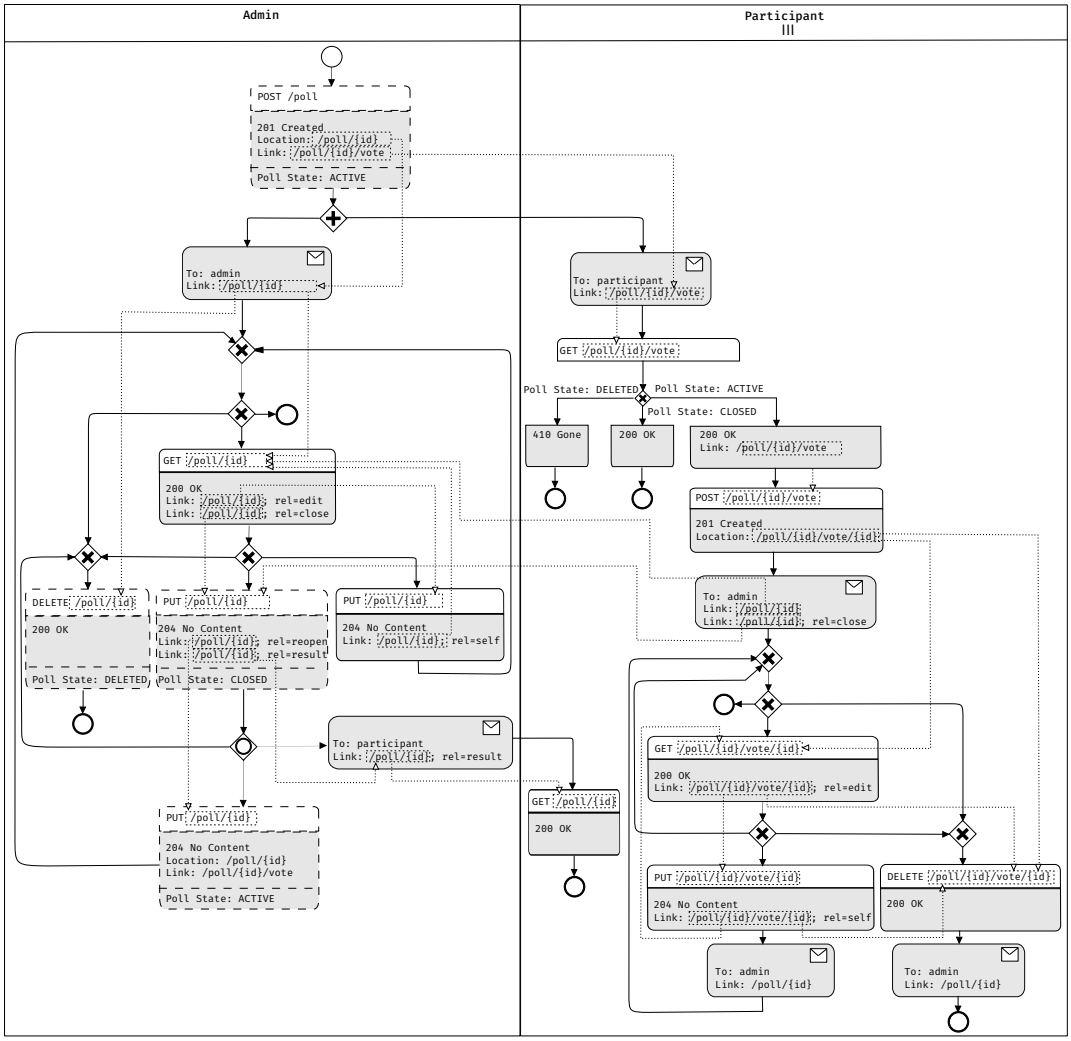
```
///condition based on data input
Access token?
Yes
GET /auth/accessToken Authentication Service 200 OAS userId
///different parallel interactions
GET /product/prodId Catalog Service 200 |
GET /user/userId Profile Service 200 OAS memberType |
GET /order/userId Order Service 200 OAS [prodId] |
GET /cart/userId Shopping Cart Service 200
...

///the flow from one of the above stated parallel interactions
...
GET /order/userId Order Service 200 OAS [prodId]
GET /related/prodId Recommendation Service 200 OAS [prodId]
forEach [prodId]
...

///the flow from another one of the above stated parallel interactions
...
GET /cart/userId Shopping Cart Service 200
GET /related/prodId Recommendation Service 200 OAS [prodId]
...

///defining what happens in the multi-instance sub-conversation
...
forEach [prodId]
GET /product/prodId Catalog Service 200 |
GET inventory/prodId Inventory Service 200
```

6.2.3 Multiple Clients - One Server Conversation

Reaching a certain resource state, frequently requires undertaking a predefined sequence of interactions or choosing among different alternative paths, thus shifting from the concept of a single RESTful interaction to the concept of a RESTful conversation [81]. To show-case the appropriateness of RESTalk for documenting real REST APIs, where the state of the resource matters, we model

the behaviour of Doodle, a well known scheduling Web service[7] which depicts RESTful conversations between multiple clients and one server [87]. Doodle's basic free offering is a Web service which facilitates event scheduling between multiple participants. We have opted for modeling the basic service, where no log-in is required. Due to the lack of detailed API documentation[8], we have used a prescriptive instead of descriptive approach in modeling its dynamics.

Figure 6.36. Doodle RESTful conversation with RESTalk



[7] `http://doodle.com`

[8] http://support.doodle.com/customer/en/portal/articles/664212

Figure 6.37. Textual DSL admin user story for the Doodle API

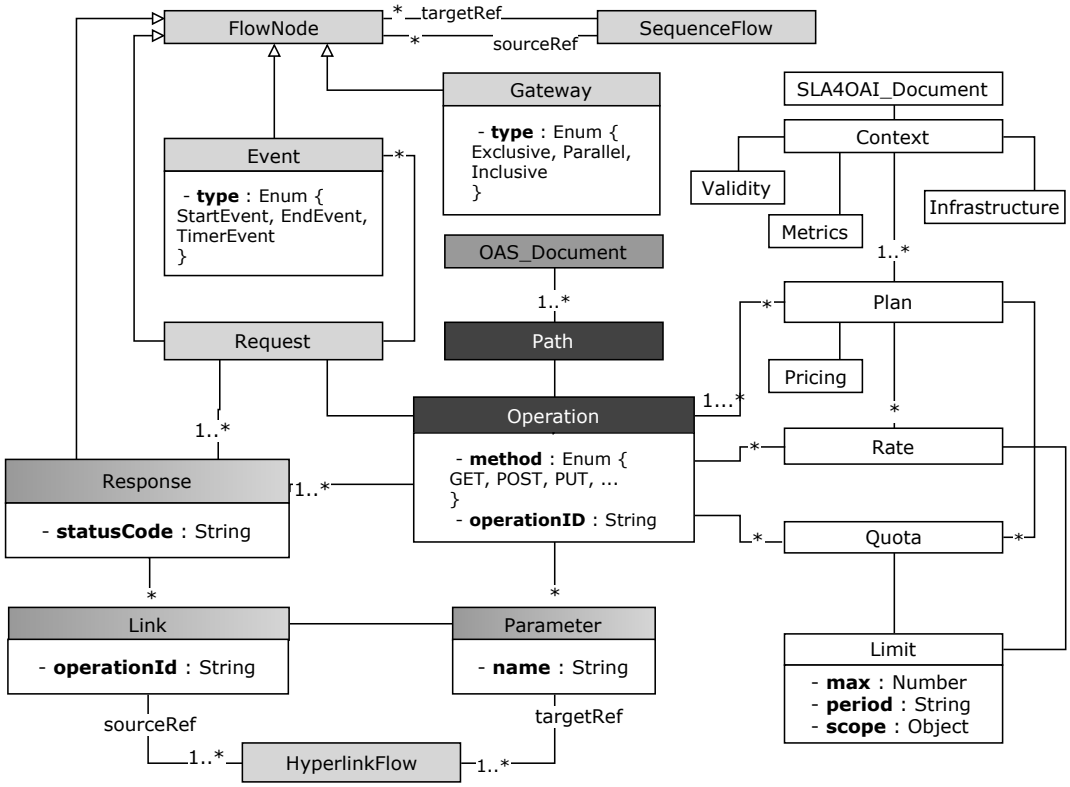
```
///specifying the role and state
Admin:POST /poll 201 /poll/id, /poll/id/vote <ACTIVE>
///specifying parallel email messages to admin and participant
send admin /poll/id | Participant:send participant /poll/id/vote
...

///the admin deleting the poll
...
admin:send admin /poll/id
DELETE /poll/id 200 <DELETED>

///the admin editing the poll
...
send admin /poll/id
GET /poll/id 200 /poll/id/edit, /poll/id/close
PUT /poll/id/edit 204

///the admin closing the poll
...
GET /poll/id 200 /poll/id/edit, /poll/id/close
PUT /poll/id/close 204 /poll/id/reopen, /poll/id/result <CLOSED>
send participant /poll/id/result ||
PUT /poll/id/reopen 204 /poll/id, /poll/id/vote <ACTIVE> ||
DELETE /poll/id 200 <DELETED>
...

///the admin sending results to participants
...
send participant /poll/id/result
Participant: GET /poll/id 200

///the admin reactivating the poll
...
Admin: PUT /poll/id/reopen 204 /poll/id, /poll/id/vote <ACTIVE>
GET /poll/id 200 /poll/id/edit, /poll/id/close
...
```

Following is the natural language description of the conversation modelled in Fig. 6.36. The client can take two distinct roles leading to different privileges. The admin role, assigned to the client who has created the poll, and the participant role, assigned to the clients who are invited to participate in the poll. After the admin creates the poll, emails are sent to the admin, with a link to the poll resource, and to all participants, with a link to the poll voting resource. Visualizing the email message required us to add a new construct to RESTalk which we decided to show in gray, to depict it sends links which can be used in future requests, and to add an envelope to it in order to achieve a certain level of semantic immediacy [142].
The poll can have three states: active, closed, or deleted. To model the state of the resource we had to add another new construct to the core RESTalk. Namely, the interaction which results with entering in a certain state is marked with a dashed line and the state variable is noted under the response box. We did not want to change significantly the interaction shape used in the core RESTalk because it is essentially the interaction that causes the state change.
The admin can edit, close or delete the poll at any time, as well as reactivate it if closed. When the poll is closed, an email notification may be sent to the participants. The participants can only vote, change or delete their votes when the poll is active, resulting with an email being sent to the admin. If the poll is closed, the participants can only see the results, without editing them. If the poll is deleted, they get an error message.
As the new added constructs in the language are in the admin role in Fig. 6.37 we show the textual DSL description of the admin user story.

6.2.4 Composite Conversation

Composite service providers, the ones which expose REST APIs implemented by combining services from other providers, can make use of modeling their conversations with the external service providers in order to understand the dependencies of their API endpoints from the external service providers. Composite service providers need such information for setting usage limitations of their APIs based on the external service providers' limitations. Determining the usage limitations requires explicitly modeling developers' knowledge regarding the Service-Level Agreement (SLA) limitations based on the API structure and the implemented RESTful conversation, which requires a holistic model uniting the three different modeling perspectives (API's structure, behaviour and SLA limitations). To create such a holistic model, we have been approached by a research group at the University of Seville who worked on SLA4OAI [64], an extension of the OAS

Figure 6.38. Simplified unified OAS (medium gray), RESTalk (light gray) and SLA4OAI (white) metamodel



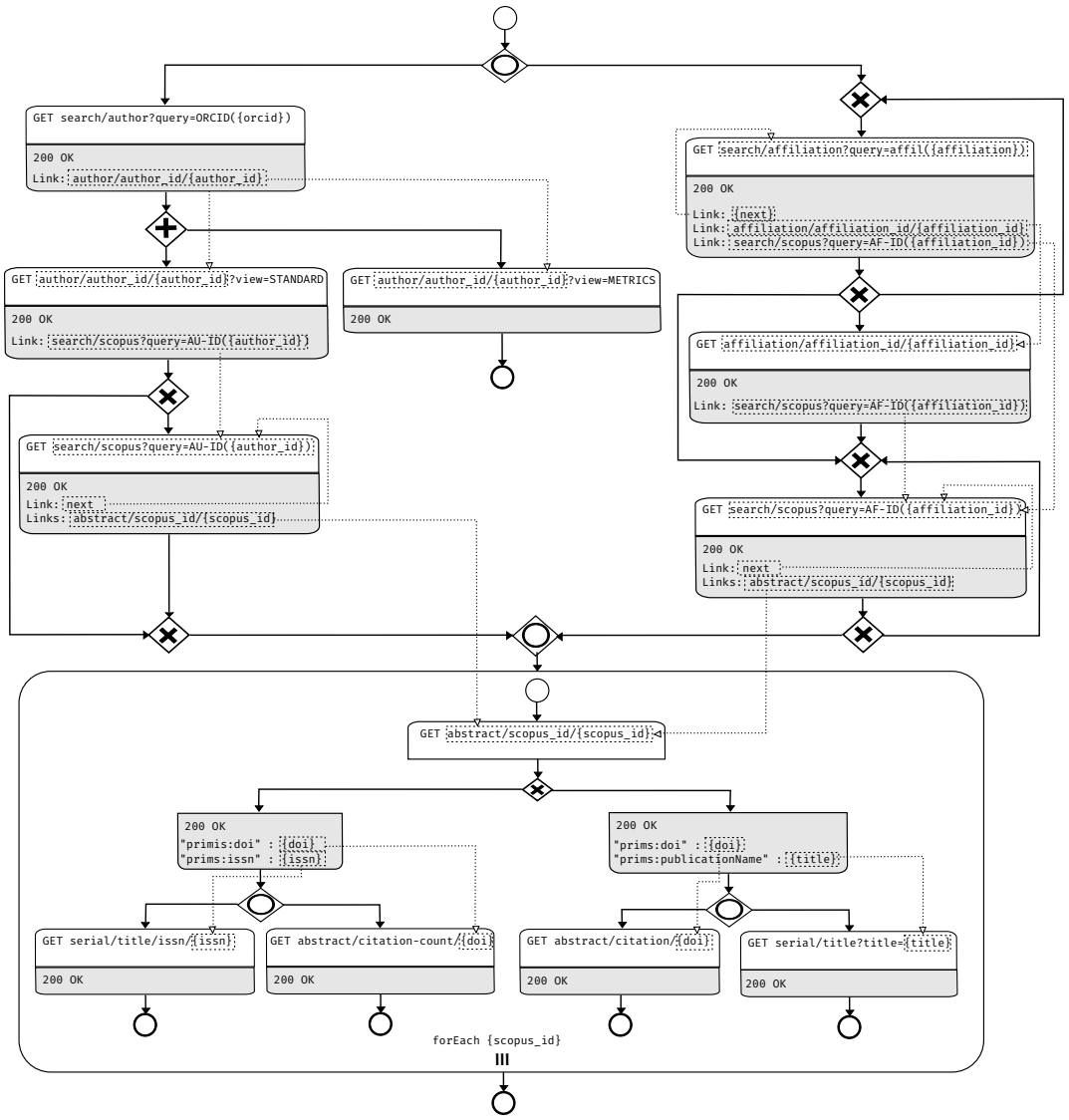
documentation with SLA details [9]. To begin with, we have outlined a comprehensive meta-model of the three REST API modeling aspects emphasizing their commonalities, i.e., the resource URI and the method (see Fig. 6.38). The elements pertaining exclusively to the structural viewpoint (OAS) are colored in medium grey, the ones pertaining exclusively to the conversational viewpoint (RESTalk) in light grey and the ones pertaining exclusively to the the SLA viewpoint (SLA4OAI) in white. Furthermore, for the common elements of the structural and conversational viewpoint a gradient is used, while the elements shared by the three viewpoints are colored in dark grey. Only by combining the information found in the different models it becomes possible to answer questions concerning the impact of complex interaction sequences on the SLA limitations of different usage plans.

The motivation behind our work in this field is a real problem faced by the University of Seville, who have a project called SABIUS that aims at generating reports

[9] `https://github.com/isa-group/SLA4OAI-Specification`

needed for the internal evaluation of their researchers. Such reports rely on the Scopus external API provided by Elsevier[10] for gathering the required data about authors and their publications.

Figure 6.39. Conversational model of a subset of the Scopus API for retrieving an Author (top-left branch), an Affiliation (top-right branch) and a Publication (bottom)



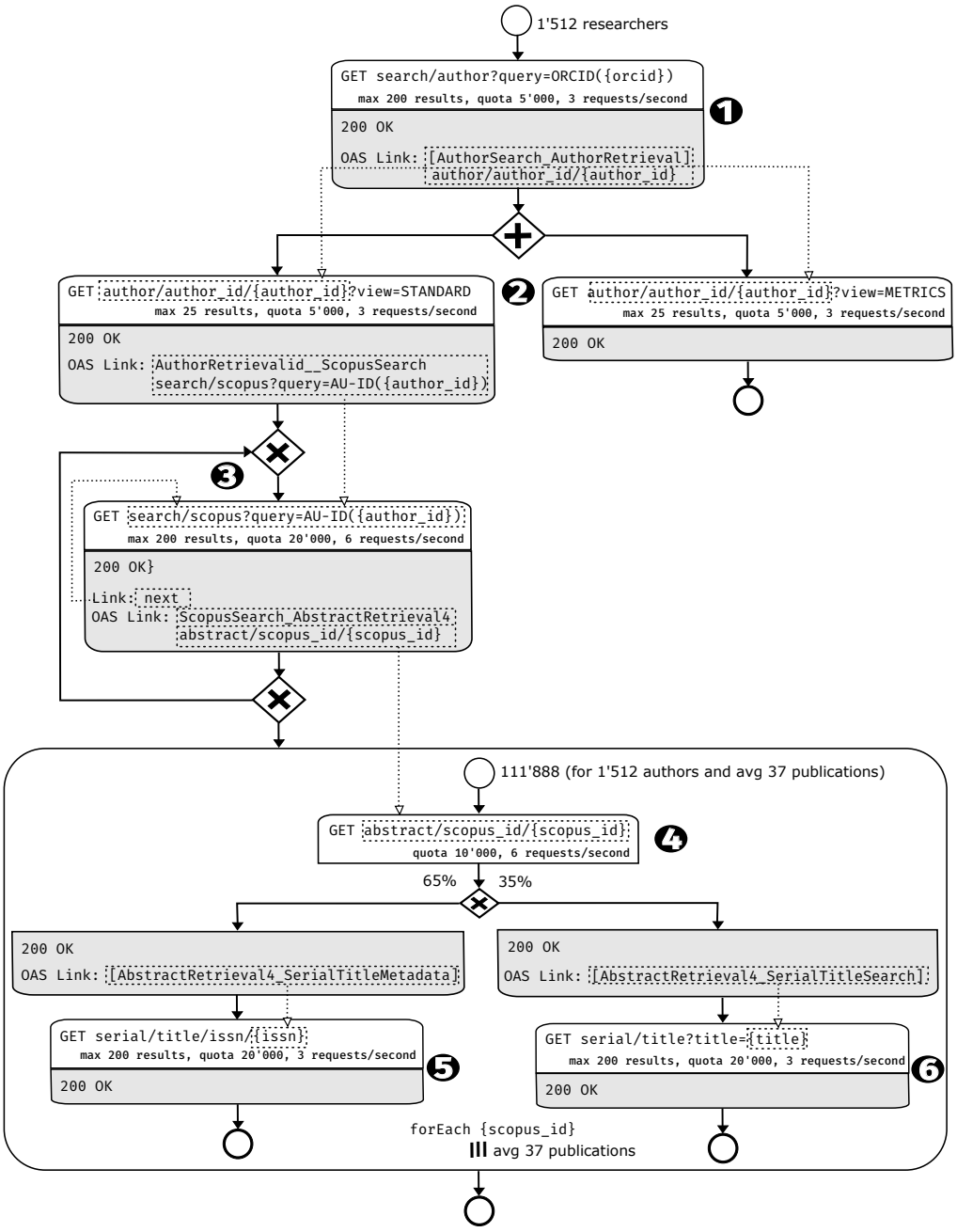
For the time being, Scopus APIs are only documented from a structural viewpoint

[10] https://dev.elsevier.com/api_docs.html

in Swagger[11] and from an SLA limitation viewpoint in natural language[12], while the conversational viewpoint is not documented at all. Each API is documented independently and there is no linking between related concepts across the APIs (i.e., there is no explicit correspondence between different identifiers used in the different APIs). In Fig.6.39 we use RESTalk to show a conversation of a subset of the Scopus APIs which depicts different conversations a client might decide to have regarding its final goal. For instance, a client can search only for an author and based on the author id can search metrics or standard information about the author and its publications. Or the client can search only for an affiliation and based on its id retrieve the related publications. Based on the information about the scopus id of the publication(s) of interest the client can search the database for the publication's abstract and identification metadata (such as doi or issn) which can then be used to get information regarding the citations, or the publication metadata (e.g., publishing venue).
Given the different available endpoints by Scopus and due to the lack of documentation regarding the conversational aspect of the Scopus APIs, the developers of SABIUS had to invest time in discovering the appropriate sequence of requests to the Scopus external API in order to get all the required data to generate the R00 report that they need. This conversation with Scopus is shown in Fig. 6.40. Given a set of ORCID with a certain size, each one is converted to an internal Scopus identifier by calling `/search/author` (1). Next, two actions are carried out in parallel, getting personal information about the author (affiliations, areas of interest etc.) together with a link for searching author's publications, and retrieving some metrics about the author itself (2). The obtained link in the previous step is used to retrieve the set of identifiers for the author's publications by calling `/search/scopus`, with a maximum of 200 results per page, which requires the client to loop until it retrieves the ids of all the publications in case they are more than 200 (3). Then, for each obtained identifier in the previous step the client collects information about the publication itself, by calling `/abstract/scopus_id` (4), and retrieving information whether it was published in a journal or in a conference, depending on the available parameters in the response. If an issn number is available, it means it was published in a journal so the client calls `/serial/title/issn` to retrieve information about the journal (5), otherwise it calls `/serial/title` to retrieve information for the conference (6). When available, the client can directly follow the links included in the response, otherwise, such as for retrieving the publication venue, it can refer to the specified OAS link

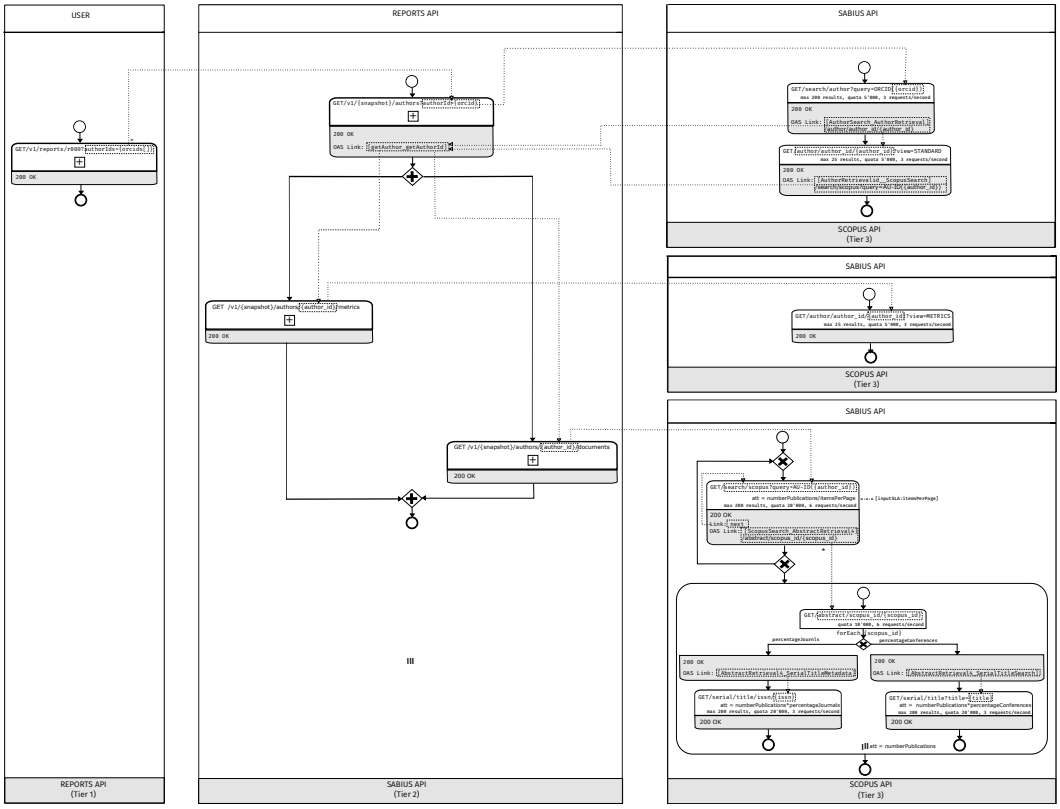
[11] `https://dev.elsevier.com/scopus.html`
[12] `https://dev.elsevier.com/api_key_settings.html`

Figure 6.40. RESTalk Conversation required to obtain the R00 report (enhanced with SLA metadata on quotas and rate limits for the Subscriber Plan, OAS links for the hypermedia flow and branch probabilities extracted from the actual data).



in the OAS documentation to retrieve the necessary parameters.
The RESTalk model in Fig. 6.40 is annotated with execution data and SLA data. As the Scopus API keeps on evolving, it changes the SLA limitations (quotas and rates), which impacts the number of authors SABIUS can generate the report for, and the time it takes to generate the report for one author without violating the invocation quotas and rate limitations set by Scopus. Thus, the SLA that SABIUS imposes for its API needs to change as well. While Fig. 6.40 shows only one possible use-case (client implementation) of the API provided by Scopus, it does not show how the API endpoints provided by SABIUS are related to Scopus, nor does it show the SABIUS API. Namely, SABIUS itself exposes a REST API with four endpoints that can be used to build different reporting clients depending on the type of reports that need to be generated. Those clients themselves could expose their services as APIs, and so on. By having an integrated modeling approach, the impact of the changes in the external provider is more evident and can be automatically calculated by using a constraint satisfaction optimization problem as shown in the demo tool by Gamez-Diaz [63] which was a result of our joint work in the field. Namely, the structural and SLA aspects of the SABIUS API are not sufficient to identify performance bottlenecks, to calculate the usage plans that SABIUS should offer to its clients, or to determine which of the usage plans offered by Scopus (subscriber or non-subscriber) it should commit to. Knowledge about the behavior of both the SABIUS and the Scopus APIs is necessary, as some calls are inside a loop (e.g., in case of pagination limits), and the conversation sometimes contains alternative paths which affects the use of the quotas. Thus, visualizing the bigger picture of how the SABIUS endpoints depend on the Scopus endpoints and how the SABIUS API can be used by a client depending on its goals can promote discussion in the SABIUS development team on which resources to expose depending on different factors (e.g., susceptibility to change due to changing SLA by Scopus, the different reports that can be generated etc.). To enable RESTalk to support such architectural modelling of REST APIs' behaviour we had to expand it to visualize calls to external services. The visualization for the above described example is provided in Fig. 6.41).
The external services do not necessarily need to come from the same provider, different providers can be used for different SABIUS endpoints. The name of the service provider is stated at the bottom of each pool in a gray box to be consistent with the fact that we use gray to show responses, while the name of the client is stated on the top of the pool in a white box to be consistent with the fact that we use white to show requests. In case SABIUS decides to change the provider for one of its endpoints only that pool will need to be changed in the diagram. The hyperlink flow between pools shows the data that is needed in order to start

Figure 6.41. Interaction dependencies between the end-user, SABIUS and Scopus APIs



the conversation in the other pool (e.g., the ORCIDs from the end user), or the data received in the external service response (e.g., the author_id received from Scopus) that is needed for further calls in SABIUS. A sample of the textual DSL to show the newly added elements of a call request and poll is provided in Fig. 6.42. As before, not all the user stories are presented, but just the ones sufficient to show the new constructs.

Figure 6.42. Sample textual DSL for the SABIUS-Scopus composite conversation

```
///specifying the client in pool1 and its call request
USER: CALL GET /v1/reports/r000??authorIds=orcids[] 200
///specifying the client in pool2 with multi-instance marker
///the new line before the OAS keyword is just for space reasons
REPORTS API(s): CALL GET /v1/snapshot/authors?authorId=orcid 200
OAS getAuthor_getAuthorId author_id
SABIUS API: GET /search/author?quiery=ORCID(ordid) 200
OAS AuthorSearch_AuthorRetrieval, /author/author_id/author_id
GET  /author/author_id/author_id?view=STANDARD 200 OAS
AuthorRetrivalid_ScopusSearch, /search/scopus?query=AU-ID(author_id)
REPORTS API(s): CALL GET /v1/snapshot/authors/author_id/metrics 200 |
CALL GET /v1/snapshot/authors/author_id/documents 200
...

///mapping the clients to their servers
SERVERS
USER - REPORTS API(s)
REPORTS API(s) - SABIUS API
SABIUS API - SCOPUS API
```

6.3 Chapter Summary

As firm believers in the iterative design of RESTalk, in this chapter we have presented the formative research techniques we have used that enabled us to improve and expand RESTalk to its current version. The exploratory survey results showed that some standard or internally developed visual notations are already used in industry which means that the need of modelling and discussing RESTful interactions has already been identified. The comparison of the core version of RESTalk to the visual notations in use has shown that RESTalk does not lag behind in terms of conciseness, understandability or efficiency and that it is rather intuitive, at least in its core version. Its ability to express domain specific concepts is appreciated. The exploratory survey together with the different use cases we have modelled with RESTalk allowed us to identify new constructs that we added to the extended version of RESTalk to be able to express resource state, sending email, logical grouping of interactions, multi-instance loops, multi-party and composite conversations, etc. Being able to model the use cases mentioned in this chapter verifies that RESTalk's expensiveness is sufficient for its currently intended use, but can be extended in case new use cases are identified.

Chapter 7

RESTalk Summative Evaluation

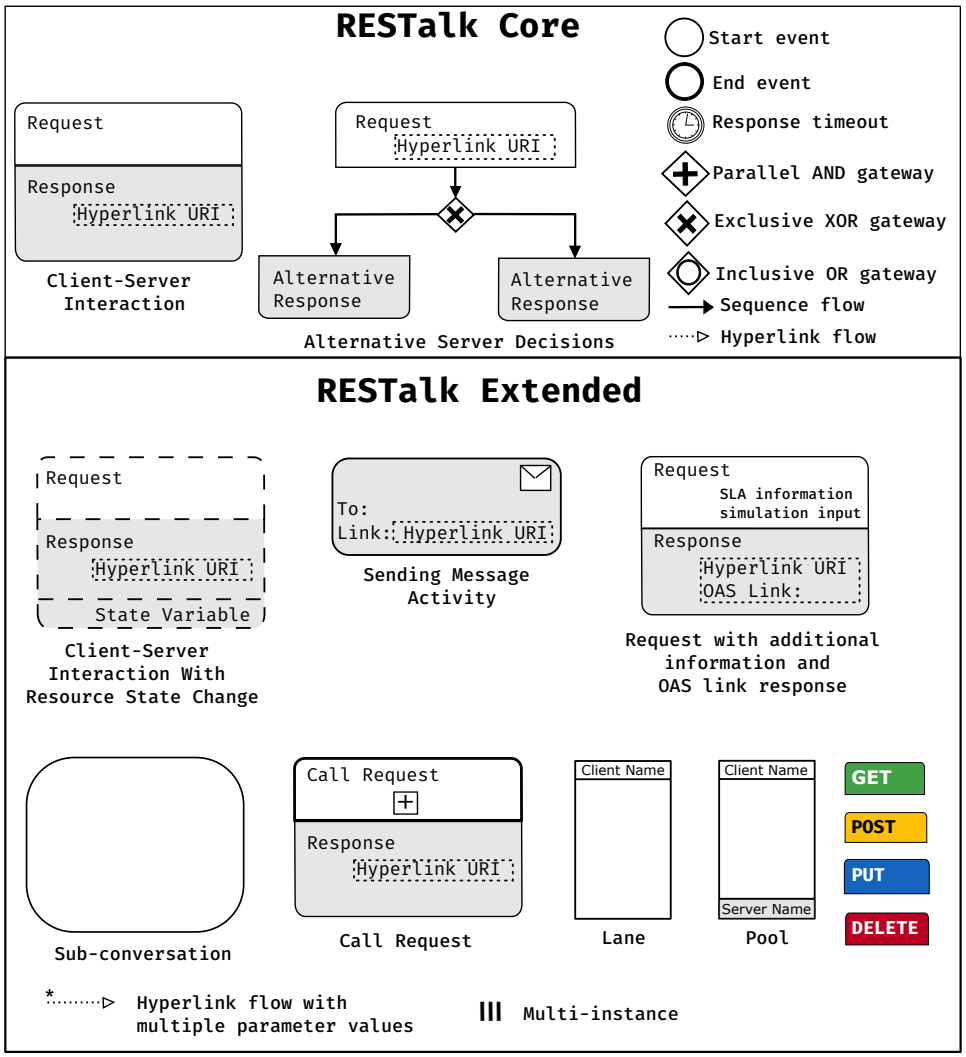
In this Chapter we present the design and the results of different research techniques that we have applied in order to evaluate RESTalk from different aspects. In Sec. 7.1 we use the Physics of Notation theory to evaluate the design of RESTalk as a visual notation. In Sec. 7.2 we compare RESTalk to other visual notations which can be used to model REST API interactions. In Sec. 7.3 we show the design and results of a mainly quantitative research technique, a controlled experiment, which we conducted with bachelor students.

7.1 Design Validation of the Graphical RESTalk Representation

Over the last years the Physics of Notation theory for the design of visual notations gained on popularity [142]. It focuses on improving the **cognitive effectiveness** of a visual notation. Linden et al. [214] propose a framework for improving the verifiability of notations based on this theory. They emphasise the importance of explicitly documenting the design rationale to understand and assess the choices made during the design phase and also provide guidance on what needs to be reported in order to be able to verify the application of the different principles of the Physics of Notation theory. For instance, they state that there are only three principles for which the design rational does not need to be reported on: semiotic clarity, complexity management and graphical economy. The critical part that they identify in the reporting requirements is a "complete representation of the semantic and visual constructs". We have a complete list of the semantic constructs defined in a meta-model and discussed textually in Sec. 4.2.1, while the visual constructs are enlisted in Sec. 4.2.2. To facilitate

the discussion in this section in Fig. 7.1 we report a summary of all the visual constructs, differentiating between core constructs and extended RESTalk constructs. We will discuss bellow each principle and to what extent it has been

Figure 7.1. RESTalk full notation



applied in RESTalk's design.

Semiotic clarity refers to having one to one correspondence between semantic constructs and visual constructs. Thus, the following situations should be avoided: 1) one semantic construct represented by multiple visual constructs (symbol redundancy), 2) multiple semantic constructs represented by the same visual construct (symbol overload), 3) visual constructs that do not correspond

to any semantic construct (symbol excess), and 4) semantic constructs that do not have any visual constructs (symbol deficit). In order for this principle to be validated all semantic constructs and all visual constructs need to be stated. We state the semantic constructs of RESTalk in Sec. 4.2.1 and the visual constructs in Sec. 4.2.2. Generally we have applied the one to one correspondence for most of the semantic constructs. Following are the exceptions: 1) the `participant` semantic entity can refer to a client or a server and can be implicit or explicit in the visual notation. The goal of allowing the participant to be implicitly stated by the colour of the interaction box is to decrease the verbosity of the DSL when possible. When the participant is to be explicitly stated, in addition to the colour, also the position in the lane/pool rectangle visual construct distinguishes between a client and a server. We decided for a top position for the client as it is the client who initiates the interaction throughout the conversation, and a bottom position for the server. The selection of colours was inherited from the BPMN Choreography initiator vs recipient color scheme; 2) the `operation` and the `resource` entities are not visually represented, but rather defined via text in the `request` visual construct; 3) The `hyperlink-flow` visual construct is used inside a conversation, to show the discovery of hypermedia in previous requests, but the same construct is also used between conversations when there is a call request to show the flow of data between different APIs. While this positively affects the graphic economy principle (i.e., keeping the number of visual constructs minimal), and allows for cognitive integration, i.e., linking different diagrams, it does cause symbol overload. However, Moody [142] himself points out the possible trade off between cognitive integration, and the one to one relationship between semantic and visual constructs.

Perceptual discriminability refers to having clear difference between symbols to allow for quick recognition. The primary discriminant proposed in the theory is the shape of the symbols, but also redundant coding (e.g., shape and color as discriminant) can be used. This principle is the most important requirement for practitioners using visual notations according to a recent survey [212]. In RESTalk, most of the visual elements (`events`, `gateways`, `sequence flow`, `lane`, markers (collapsed `call marker` and `multi-instance marker`), `sub-conversation`) have been borrowed from the BPMN standard notation [101]. The adherence of BPMN to the principles of the Physics of Notation theory has been analysed in [68]. BPMN uses different shapes for different types of meta-classes, e.g., events are represented by a circle shape, gateways are represented by a diamond shape. The `client-server interaction` element has been inspired by the choreography task element in the BPMN Choreography diagrams as explained in Sec. 4.2.2. The shape of the `pool` has been adjusted

from BPMN to be able to state both the client and the server, distinguishing them based on the colour and position. We include the `link/parameter` entities in a dashed rectangle as we also use a dashed line for the `hyperlink flow` to distinguish it from the `sequence flow`. We were inspired by relationships in a UML class diagram for using the star symbol on the `hyperlink flow to visualise multiple parameters`. We use dashed line of the client-server interaction element to visually distinguish interactions which lead to a change in the `state`. We decided not to use a completely different shape for this purpose as resources always have states and state change is always caused by a request (except for a GET request), so it is up to the modeler to decide when such change is important to be emphasized visually. We have designed the `sending message activity` to have a similar shape to the response construct as it sends the links needed to make the next request. As opposed to a response construct which uses sharp edges in the upper angles, the sending message activity uses soft (round) edges in all angles.

The **semantic transparency** principle requires that the appearance of the visual constructs should suggest their meaning. The theory distinguishes between semantic opacity (arbitrary relationship between meaning and appearance), semantic perversity (appearance suggests different meaning) and semantic immediacy (meaning can be immediately inferred from appearance without explanation). Most of the elements in the RESTalk graphical representation use semantic opacity (they are not directly related to the REST domain or the semantic meaning of the element). We borrow some BPMN elements which have semantic immediacy such as the timer event showing response timeout which uses the clock symbol, or the call request marker which uses a plus sign to show a concept that has additional content and needs to be unrolled (opened). The sending message activity also has some semantic immediacy due to the use of an envelope which is intuitive for showing the sending of a message.

The **complexity management** principle requires that explicit mechanisms are in place for dealing with complexity. Namely, the theory argues that the number of elements in a diagram that a person can comprehend at a time is limited by the working-memory capacity. Thus, to be cognitively effective a notation should provide mechanisms for decreasing the size of diagrams by modularization (e.g., subsystems) and hierarchical structuring, i.e., levels of abstraction (e.g., decomposable constructs). In RESTalk two levels of abstraction are allowed. While `subcoversations` are used as a logical grouping mechanism but at the same level of abstraction as the rest of the conversation, `call requests` are used to

show the recursive decomposition of the diagram where the response to a call request will only be received when the conversation initiated with that call request has finished.

The **cognitive integration** principle requires explicit mechanisms to be in place to integrate information between diagrams in order to help the reader assemble the information into a "coherent mental representation of the system" (conceptual integration) and to simplify navigation between diagrams (perceptual integration) [142]. RESTalk supports summary diagrams, i.e., multiple `conversations` in the same diagram only in the case of composite conversations where the information flow between the different conversations is shown via the `hyperlink flow` construct. However, currently the perceptual integration is missing as there is no way of linking different diagrams to represent a system (like in the Imgur API use case in Sec. 6.2.2).

The **visual expressiveness** principle recommends the usage of the full range of visual variables, such as shape, colour, size, brightness, orientation, texture etc. Most software engineering notations use only the shape as a visual variable [142]. In RESTalk, in addition to shape we decided to also use size, colour and texture to support visual expressiveness. `Exclusive gateways` which show different possible responses from the server are designed to be smaller in size with respect to exclusive gateways which show a path divergence due to client's decision. To visually distinguish between `requests` and `responses`, as well as between `clients` and `servers` we use colours, white for requests/clients and gray for responses/servers. The decision to use these two colours was inspired by the way these colours are use in BPMN Choreography diagram, where white is used to show the party initiating the interaction and gray is used to show the recipient party. Additionally, to distinguish between different `methods` in a request we allow for using colours for the request different from white, requiring colours which are already used to distinguish against such methods in Postman. We use different texture for the edges, dashed for `hyperlink flow` and solid line for `sequence flow`. All of the nodes use solid line, except for the interactions that lead to a `change in state` which use a dashed line. We also recommend a vertical orientation of the diagram, when possible, so that in case of juxtapose interaction the incoming sequence flow is connected to a request and the response is followed by an outgoing sequence flow.

The **dual coding** principle refers to the usage of text to complete graphics. An example would be using commonly understood words to complement the visual constructs [213]. Both annotations (comments) and hybrid constructs (using text inside a visual construct to both expand and reinforce the meaning) are suggested as dual coding approaches. In RESTalk many of the constructs are hybrid

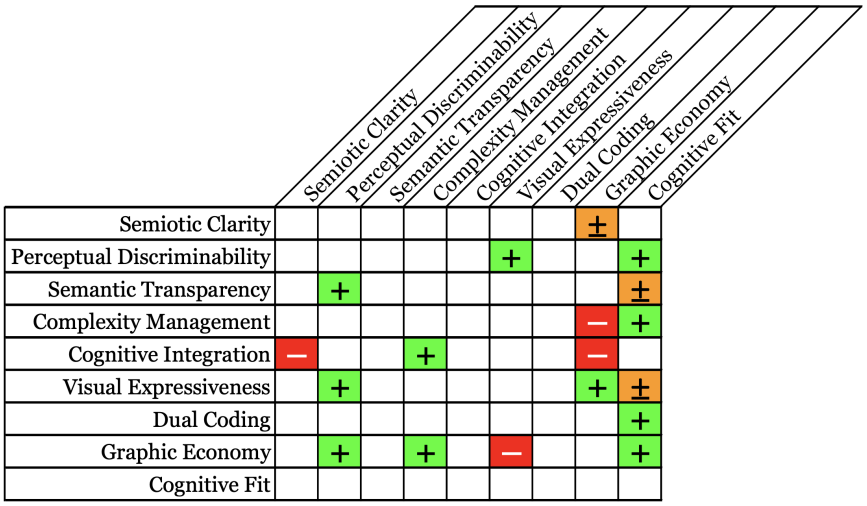
constructs. We use text to depict some semantical constructs such as `operation` or `resource`, and we require text also to show the status code/links in the response. We advise to use upper case letters for stating the operation (method) and to start each URI with the "/" symbol as this is how method and URI are commonly used in frequently used API tools, such as Postman. Furthermore, adding a state name in case of a `state change` is encouraged. A `lane` has to contain a textual annotation of the name of the client, while a `pool` should also have a textual annotation of the name of the server. A `sending message activity` should have a textual annotation stating who the message is sent to and which links are included. Furthermore, we allow text to be used to add additional relevant information to the first-class citizens, i.e., the request/response constructs depending on the goal of the diagram. By doing so we intend to make the DSL more versatile to respond to different needs. `Gateways` do not have to labeled with text, but users can label them if it helps the understanding of the diagram.

The **cognitive fit** principle refers to the usage of different dialects for different audiences, thus ensuring that the notation fits with the cognitive background of the targeted users. In RESTalk we do differentiate between the core constructs which are sufficient to express simple one to one (client-server) conversations and are thus suitable also for novice modelers, and extended constructs needed to support multi-party conversations or more complex client-server conversations more suited for expert modelers.

The **graphical economy** principle refers to keeping a cognitively manageable number of visual constructs in a given notation. Software engineering notations tend to increase the number of visual constructs over time in the effort to increase their semantic expressiveness [142]. Miller [136] has established the upper limit of 7 ± 2 as the maximum number of visual constructs which a person can effectively memorise at a given time. RESTalk has 10 symbols in its core dialect, and other 8 symbols in the extended RESTalk dialect as evident in Fig. 7.1. Our design decision aimed at keeping this number as low as possible to facilitate the learnability of the language.

Following all of the above mentioned principles in the design of RESTalk, or any other visual notations, is impossible due to the innate dependency between the requirements of the principles themselves. A visual matrix of the trade-offs between the principles presented by Van der Linden et al. [212] is shown in Fig. 7.2.

Figure 7.2. Trade-off between the Physics of Notation principles (presented in [212])

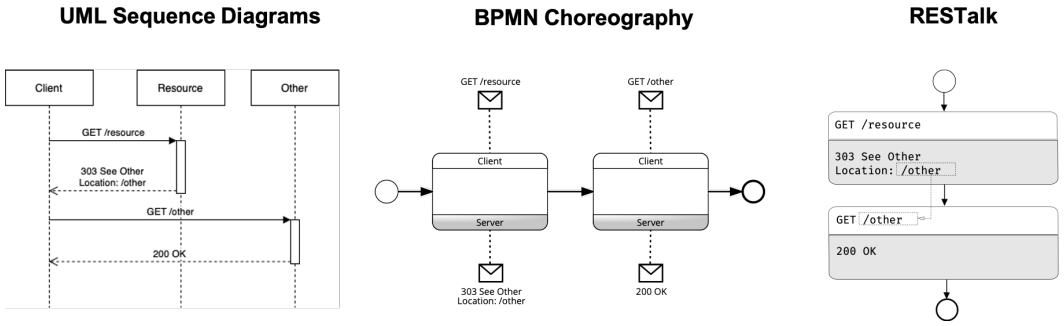


7.2 RESTalk vs Non-domain Specific Languages

In addition to cognitive effectiveness of the visual notation discussed in Sec. 7.1, the conciseness of the language, given its domain, is also an important aspect. Our exploratory survey, discussed in Sec. 6.1, has pointed to the UML sequence diagrams as the most frequently used in practice for modelling the interactions with REST APIs. As a matter of fact, it was when using UML sequence diagrams [81], and later BPMN Choreography diagrams [151], to visualise RESTful interactions that the idea of designing a DSL was born. When discussing the results of our exploratory survey in Sec. 6.1.3 we also discussed the perceived characteristics of RESTalk in terms of conciseness, expressiveness etc. compared to UML Sequence diagrams and BPMN Choreography diagrams, with the results being encouraging for the use of RESTalk (see Fig. 6.8). In this section we are going to go into more technical analysis of these notations and discuss one simple and one slightly more complex RESTful interaction pattern modeled with all three notations.

In Fig. 7.3 we show the very simple redirect pattern discussed in Sec. 6.2.1 as well as in [81] using the UML and BPMN standard visual languages in addition to RESTalk. This being an overly simplified example, any of the mentioned diagrams renders the idea of resource redirection as all three languages focus on modeling interactions. However, in our opinion what changes between the different diagrams is the first class citizen, the visual elements that stand out from

Figure 7.3. Redirect pattern modelled with UML Sequence diagram, BPMN Choreography diagram and RESTalk

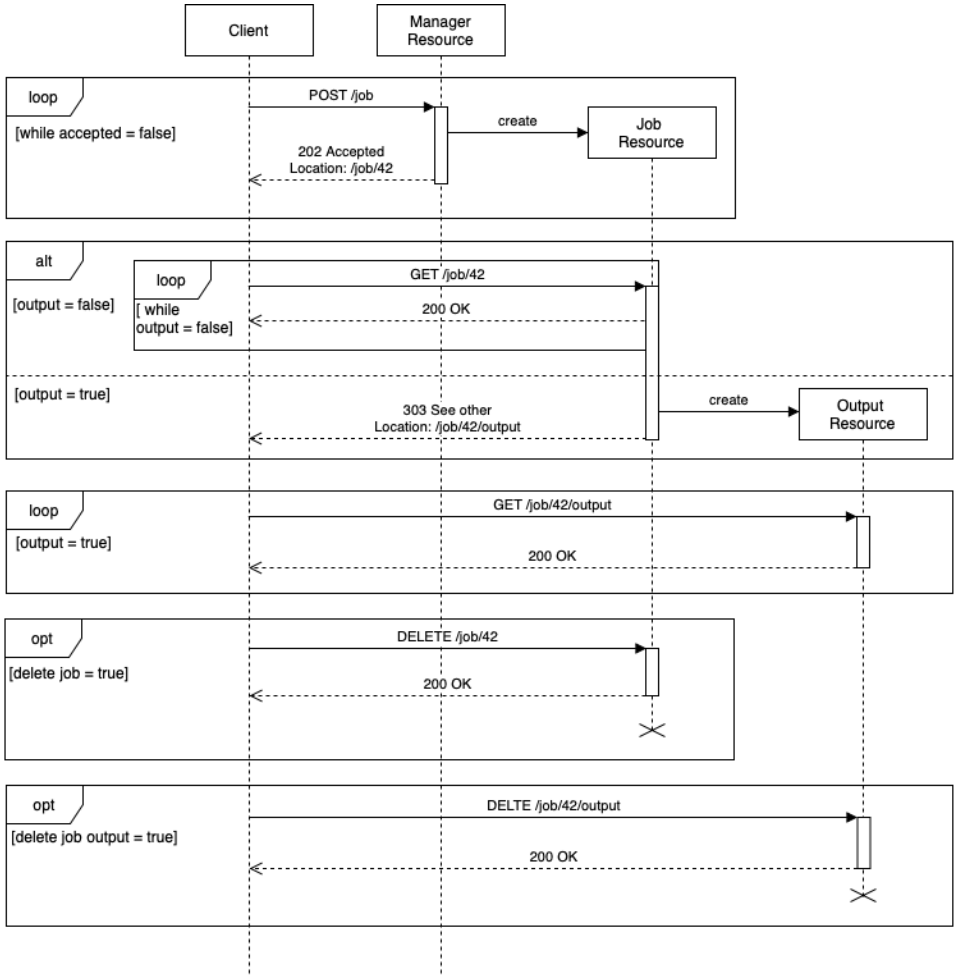


the rest of the diagram.

UML Sequence diagrams are one of the most popular UML notations for visualizing interactions [135]. The main focus is on showing who is talking to whom and with what frequency. The participants and their lifelines are well visible as is the nesting of the interactions, distinguishing clearly between synchronous and asynchronous interactions. The execution specification can provide some notion of time. However, all types of interaction convergence/divergence (e.g., choice, iteration, parallelism) are visually presented with a fragment box and the type is textually specified by a fragment operator, e.g., $loop$, alt, par etc. (see Fig. 7.4). The respondents of our survey in Sec. 6.1.3 have pointed to conditions and loops being challenging to use in UML Sequence diagrams. When used in a RESTful interactions domain, the lifelines become representations of resources leaving less space for visualizing or distinguishing participants in multiparty conversations from resources.

While UML is intentionally not domain specific, BPMN targets business processes with BPMN Choreographies focusing on the interaction coordination between participants in cross-organizational settings. As such different types of control flow convergence/divergence have dedicated visually distinctive constructs in the language. The first hand citizen in BPMN Choreographies are the choreography tasks which always have an initiator of the message and a recipient of the message and they are designed to support multiple participants in the conversations. The goal of the choreography task is summarised/contained in label making the message content in the message construct optional. As both UML Sequence diagrams and BPMN Choreographies are not specific to the REST domain they visually do not emphasize some important RESTful interaction facets, for instance the details of the request/response pairs such as the method and status codes combinations. In standard languages, such as UML or BPMN, these

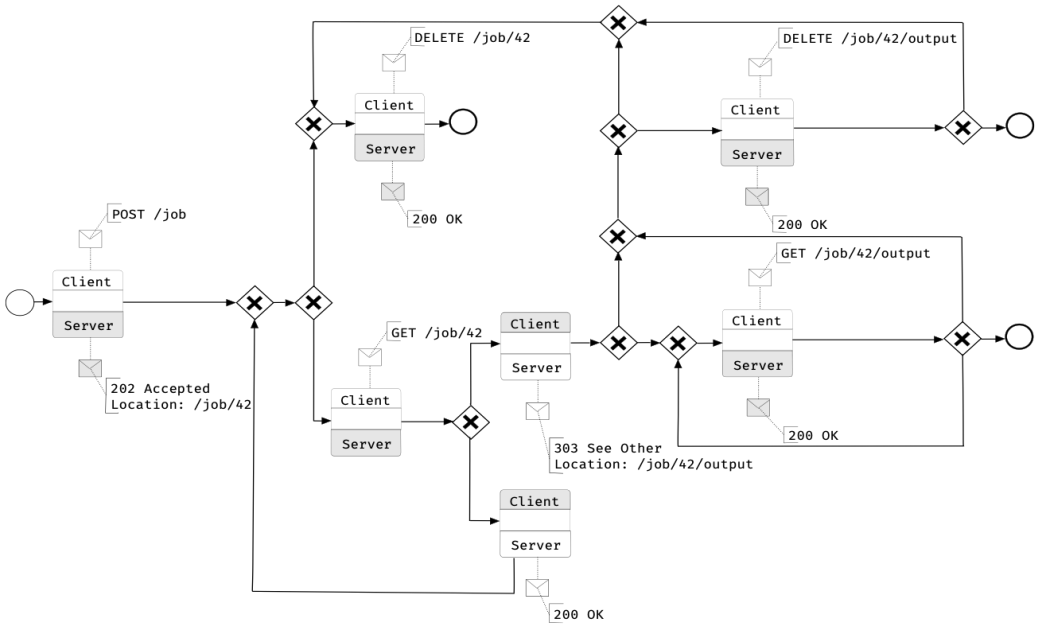
Figure 7.4. Long Running Operation with Polling pattern modelled with UML Sequence Diagram



would need to be captured by adding domain-specific semantics to model annotations and comments [81; 151], thus cluttering the readability of the diagram. Of course such cluttering can be disregarded in simple models such as the one in Fig. 7.3, but becomes more evident in even slightly more complex patterns such as the Long Running Operation with Polling pattern also described in Sec. 6.2.1 which for comparison purposes we have designed using UML Sequence diagram in Fig. 7.4, using BPMN Choreography diagram in Fig. 7.5 and using RESTalk in Fig. 7.6. An important domain addition in RESTalk compared to the non-domain specific languages is the visualization of resource discovery paths through the use of hyperlinks.

REST can be defined as a set of constraints which we described in detail in

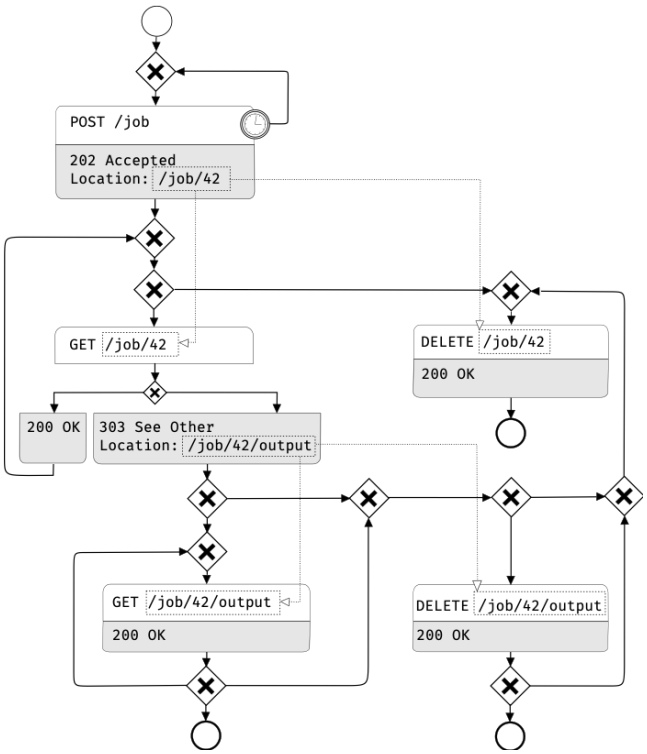
Figure 7.5. Long Running Operation with Polling pattern modelled with BPMN Choreographies



Sec. 2.2.1. Thus, in this section we will also discuss how RESTalk supports those constraints in addition to comparing it to non-domain specific languages. The client-server constraint requires that changes in the server do not break the client. On the other hand the satelessness constraint requires the client to send all the necessary data when making a request. A RESTalk model specifies where the client can get the data in order to make the request, i.e., which prior requests need to be made in order to obtain that data and thus respects the satelessness constraint. On a model verification level the data flow and the satelessness constraint can also be verified in general purpose languages through the control flow, however there is no visual construct to help the diagram reader to detect such flows. On the other hand, the RESTalk model provides the API developer with knowledge of the required data the client expects when making a set of requests. Thus, the API developer can ensure backward compatibility in terms of API behaviour when changing the API in order not to break existing clients. RESTalk requires the request to contain one of the accepted HTTP methods and a URI or URI template, thus enforcing the uniform interface constraint. When designing RESTalk we did not consider necessary to model the cacheability of a response as this property does not change the sequence of needed requests, but just the

place from where the response is being served. We also did not consider necessary to model the different layers between the request from the client and the response from the server to maintain diagram simplicity, as the goal of RESTalk is not to show the system architecture for which other diagrams complementary to RESTalk can be used. When it comes to the HATEOS constraint requiring clients to follow links instead of hard-coding the business logic, RESTalk can help API designers in the API Design First approach to reason on which links should be included in which response depending on the state of the resource. HATEOS appears to be the most difficult REST constraint to be implemented, as most APIs which claim to be RESTful do not actually enforce this constraint [178; 82]. In such current state of the art reality, RESTalk can also help the API client developers to identify the right path to take to achieve the goal based on following the hyperlink flow in the visual diagram when no hypermedia is available in the responses.

Figure 7.6. Long Running Operation with Polling pattern modelled with RESTalk



7.3 Controlled Experiment

In order to investigate the possible advantages of using RESTalk diagrams as a complementary documentation to the OAS documentation of a given REST API we have designed and conducted an in-class controlled experiment.

7.3.1 Experiment Design and Setup

Experiment Design

A good design of the controlled experiment is essential for the quality of the experiment and the conclusions to be drawn from the same. Thus, when designing our experiment we tried following the guidelines in [225; 224] when applicable and when feasible. We also aimed at avoiding the sin of irreproducibility [21] by documenting the design in detail.
As a baseline for our comparison we chose to compare the approach of using RESTalk diagrams, in addition to the API provider documentation, to the approach of using only the API provider documentation. The **research questions** we aimed at responding with the experiment are the following:

RQ1: *Does RESTalk help to discover complete and correct sequences of requests given a goal?*

RQ2: *Does RESTalk help to discover the required sequences of requests faster?*

RQ3: *What is the perceived usefulness of the RESTalk diagram?*

Each scientific experiment involves two types of variables, independent variables, which are manipulated by the researcher, and dependent variables, which are measured to check the variability of their values as a result of the manipulation of the independent variables [227]. In our experiment we use one **independent variable**, i.e., the REST API documentation used to solve the tasks. The API documentation variable has two levels, RESTalk and OAS documentation as one level, and only OAS documentation as a second level or baseline. The **dependent variables** used to answer *RQ1* are: 1) completeness of the sequences of calls to the REST APIs, i.e., given the task are all the required calls stated in the participant's answer; 2) correctness of the sequences of calls to the REST APIs, i.e., given the task are the calls stated in the participant's answer necessary and, if they are, are they stated in the correct order. The completeness and correctness are the measures of effectiveness of the proposed approach. For *RQ2* on the other hand we measure the completion time, i.e., the time it takes the participants to

state their answer, which is a measure of efficiency of the proposed approach. *RQ3* is investigated by using a questionnaire where we ask participants to rate their perceived helpfulness of the RESTalk diagrams.
The **participants** in the experiment were bachelor students taking the Web Atelier course where, among other things, they also learn how to develop clients for REST APIs. During the course the students were already working on an application offering a canvas for drawing images. Therefore, for the experiment we have decided to use an image sharing service provider, so that at a later point the students could integrate the external service presented during the experiment to their application to upload their canvas drawings as well as to find and save images from the service provider to their canvas application. After looking at the APIs of different image sharing service providers we have decided to use the REST API of Imgur as mentioned in Sec. 6.2.2 which provides OAS documentation and is supported by Postman to facilitate testing.

Experiment Execution

The experiment designed as described above has been executed during lecture hours in two consecutive years of the Web Atelier course (hereinafter called Experiment 1 and Experiment 2). While the research questions and the content of the tasks to be performed by the students remained unvaried, there were several **main differences between the two experiments**.
The first difference refers to the *experimental treatments*, i.e., how the participants were exposed to the independent variable. In Experiment 1 we used between-subject design to minimise the notation learning effect. The students were divided into two groups and Group A solved all tasks using only the OAS documentation, while Group B solved the first (warm up) task using only the OAS documentation and the rest of the tasks using the RESTalk diagrams as complementary documentation to the OAS documentation. In Experiment 2, on the other hand, in order to minimise the random noise created by different existing knowledge or interests of the participants, we have decided to use a within-subject design, where the same participant tests both levels of the independent variable. Namely, we have again divided the participants into two groups. Then, to Group A only the OAS documentation was made available in order to complete the first and fourth task, while to Group B also RESTalk diagrams were made available for those tasks. For the second and third task, Group B could no longer use the RESTalk diagrams, but they were made available to Group A. In both experiments we did not use a random assignment of the participants to the groups, but rather tried to balance the groups in terms of prior knowledge, using

their grades in previous course assignments as a factor.
Another difference refers to the *number of RESTalk diagrams* available to the participants in the relevant group. While in Experiment 1 the group which had the RESTalk diagrams available always had all three diagrams available (as mentioned in Sec. 6.2.2 we have split the API conversations into three diagrams, one showing the interactions which can be performed by an unauthenticated user, one showing the available interactions for an authenticated user for publishing and managing own content, and one showing the available interactions for an authenticated user for discovering and interacting with other user's content). It was up to the participant to decide which diagram to use. In Experiment 2 we have decided to only make available to the users the diagram that can help them solve the tasks, instead of all three diagrams as in Experiment 1.
The third important difference between the experiments refers to the *RESTalk diagrams* themselves. While in Experiment 1 they were static black and white diagrams, in Experiment 2 we have introduced colour for the methods as a secondary notation and we have made the diagrams interactive and better integrated with the OAS documentation. Namely, by clicking on any request element in the diagram the user was redirected to the exact point of the OAS documentation dealing with that request. A short one sentence description of the semantics of the request was provided when hoovering over it. When hovering over a parameter it was marked in red how it can be discovered (if it is in the request) or what are the possible requests where the parameter can be used (if it is in the response). The diagram was rendered searchable by a simple "command + F" functionality of the browser. And last but not least, by clicking on the coloured boxes in the order in which the user wants to state the requests in his task solution they get copied in the text box thus facilitating the compilation of the solution. In both experiments, given that the participants are novices in the use of REST APIs, we have used only the core RESTalk constructs to create the diagrams.
The fourth difference between the experiments refers to the *number of interaction sequences* that were required to be stated by the participants. While in Experiment 1 the wording of the tasks left the possibility to state different possible solutions open, in Experiment 2 the wording encouraged students to continue to the next task once they have identified one possible solution.
The last difference refers to the *exposure of the participants to the Imgur application and RESTalk* before the experiment. Namely, in Experiment 1 we did not provide any introduction to Imgur as an image sharing community, thus the participants had to discover Imgur during the experiment. We were simulating a situation where a developer runs into a new service provider and has to discover both the functionality it provides and its API. We also provided no introduction to

RESTalk as we waned to test the intuitiveness of the semantics of the notation. In Experiment 2, on the other hand, before the experiment we have provided an introduction to the basic functionalities and the terminology used in Imgur, as well as a short example of how the Imgur API can be used to add an image to an album modelled in RESTalk. We also presented the interactive content of the RESTalk diagrams discussed above. Thus, in Experiment 2 participants who attended the lecture before the experiment were not completely new to Imgur and RESTalk.

Before Experiment 1 we have conducted a test-run with Prof. Pautasso and the teaching assistant of the Web Atelier course, Andrea Gallidabino, to verify the estimated time needed to complete the experiment as well as the correct wording of the tasks. Before Experiment 2 we have decided to conduct a larger pilot study with the members of the Software Institute during one of their regular seminar sessions. The Software Institute includes researchers, PhDs, Postdocs as well as professors in different areas in computer science[1]. One day before the pilot study, the members of the Software Institute were required to fill in a background survey regarding their experience with writing RESTful clients so that they can be split into two groups with similar level of experience. The experimental treatment in the pilot study was equal to the one in Experiment 2, i.e., a within-subject treatment. On the day of the pilot study, the participants were provided a quick 5 min tutorial on Imgur, REST APIs and RESTalk. Then, they had 30 min to solve the tasks to be followed by a discussion regarding the design of the controlled experiment in terms of research questions, independent and dependent variables, as well as the tasks content and understandability. Given the limited time for conducting the pilot study itself, the goal was not to go through all of the tasks in the experiment, but to give the participants a feeling of what the tasks are like and obtain feedback on the general design, content and set up of the experiment.

For the experiments no time limit was posed for completing the Experiment, but the students were incentivised with a fail/pass/exceed grading for providing a correct and fast solution.

The experiments have been conducted using USI's e-learning platform, iCorsi, which tracks participants IDs and records a timestamp of task submission. The pilot study with the members of the Software Institute has been conducted using Google Forms.

[1] `https://www.si.usi.ch`

Experiment Structure

The experiment consisted of five parts. **Part** I gathers background information on the experiment participants regarding their experience with using REST APIs and Imgur. **Part II** was only present in Experiment 2 and aimed at evaluating the understanding of Imgur and REST based on the warm-up introductory session in the lecture prior to the experiment. It evaluates both perceived understanding using a 10 point Likert scale and conceptual understanding using two multiple-choice questions and one open-ended questions. **Part III** refers to functionalities that can only be used by an authenticated user and consists of 3 tasks asking the participants to state the correct sequence of interactions with the Imgur API in order to achieve given goals using the available documentation. The goal in Task 1 is to search the Imgur gallery, vote an item and add it to favorites providing a link to the same to the user, the goal in Task 2 is to upload an image, add it to a new album, share it with the community and start following the tag used for the image and add it to favorites, and the goal in Task 3 is to replace an existing image in an album. The tasks had an increasing complexity when moving to the next task and also contained some multiple-choice or open-ended subquestions. **Part IV** refers to functionalities that can be used by unauthenticated users and consists of true/false and open-ended questions. **Part V** is a survey aimed at evaluating the perceived complexity of the tasks as well as the perception about the usefulness of RESTalk for performing the tasks. All of the questions and tasks as well as their solutions when applicable are available in Appendix B. The Appendix also contains the treatment regarding the independent variable (API documentation) for each part/task in each of the experiments. The treatment is also summarized in Tab. 7.1.

Table 7.1. Experimental treatment in different experiment runs

	Experiment 1		**Pilot Study**		**Experiment 2**	
	Group A	**Group B**	**Group A**	**Group B**	**Group A**	**Group B**
Part I - Background Information	✓	✓	✓	✓	✓	✓
Part II - General Understanding of Imgur and REST					✓	✓
Part III - Tasks Authenticated User	✓	✓	✓	✓	✓	✓
-Task 1: Vote, Favorite and link to the first search result	OAS	OAS	OAS	OAS+RESTalk	OAS	OAS+RESTalk
-Task 2: Publish in album and gallery, favorite and follow tags	OAS	OAS+RESTalk	OAS+RESTalk	OAS	OAS+RESTalk	OAS
-Task 3: Replace image by title	OAS	OAS+RESTalk	OAS+RESTalk	OAS	OAS+RESTalk	OAS
Part IV - Tasks Unauthenticated User	OAS	OAS+RESTalk	OAS	OAS+RESTalk	OAS	OAS+RESTalk
Part V - Survey	✓	✓	✓	✓	✓	✓

Participants

The **pilot study** with the Software Institute members consisted of 20 participants out of which 65% stated that they have written a client that calls REST APIs

before. However, most of them (38.5%) had up to two months of experience in using REST APIs while 30.8% had more than 2 years of experience. 90% of them have never used Imgur as an application.
In Experiment 1 there were total of 29 students, 14 in Group A and 15 in Group B. There were just three students who had experience with writing clients for REST APIs, two in Group A (one had between 2 and 6 months of experience and the other one between 1 and 2 years) and one in Group B (less then one month of experience). None of them had experience with the Imgur API. 6 participants had used Imgur as a web application (3 in Group A and 3 in Group B).

Figure 7.7. Grading structure and REST API experience per groups per experiment

In Experiment 2 there were total of 24 students, 12 in each group. The proportion of students who had experience with writing clients for REST APIs was much more important in this run with 9 students in total, 6 in Group A (4 had less then 6 months of experience, one had between 7 and 12 months of experience and one between 3 and 4 years) and 3 in Group B (two with less then one month of experience and one between 7 and 12 months of experience). None of them had experience with the Imgur API. Only 4 participants had used Imgur as a web application (3 in Group A and 1 in Group B).
As mentioned earlier, before the start of the experiment the participants in Experiment 1 and Experiment 2 were divided into groups based on their grading in the course with the scope of keeping the knowledge/interest balanced. In Fig. 7.7 we present the grading structure and the REST API experience in each of the groups in each experiment.

Metrics and Analysis

The **completeness and the correctness** of the sequences of calls to the Imgur API stated in participants' answers was manually graded by myself, thus assigning a single measure to the effectiveness of the proposed treatment regarding the available API documentation. We have expressed this measure as a percentage, with 100% referring to an answer which is both complete and correct and points being deducted if some requests are incorrect or missing.
The finishing time for each of the tasks has been logged by iCorsi as the participants needed to submit a separate form for each task. We have then calculated the **completion time** of each task as the difference between the finishing time between two sequential tasks. We express the measured time in minutes.
We have measured the **perceived helpfulness** of the RESTalk diagrams based on a 10 point Likert scale answers from the participants as well as based on open-ended questions regarding the pros and cons of RESTalk.
In order to answer *RQ*1 and *RQ*2 of the controlled experiment, for the first two metrics, completeness/correctness grade and completion time, we performed **statistical significance analysis** to determine whether the identified differences are statistically significant and thus whether they can be inferred for the population as a whole. Determining statistical significance requires statistical tests, the nature of which depends on different factors [125]. The first factor to consider is the number, the size and the nature of the samples. In our case, one sample is comprised of the results from a given treatment. As our independent variable, i.e., the used API documentation, has two levels (level 1: OAS and level 2: combination of OAS and RESTalk) we have two samples in each experiment. Next we need to consider the nature and the distribution of the collected data, i.e., completeness/correctness grade and completion time. We are working with quantitative continuous data, which is not normally distributed. Furthermore, the sample size in our experiments is less then 30. These two factors require the use of non-parametric tests. This type of tests are less powerful than parametric tests. The power of a statistical test refers to the probability that the test will reject the null hypothesis when the alternative hypothesis is true. However, parametric tests require the use of large samples and assume normal distribution [125], two assumptions which were not true in our experiment.
Although in Experiment 2 we exposed each participant to the two different levels of the independent variable, the participant was not required to do the same task once only with the OAS documentation and once with the combination of OAS and RESTalk documentation which makes the samples in both experiments unpaired as each participant resolves each task only once. The difference in the

metrics between samples could go in any direction (increase or decrease), thus a two-tailed test was required. Considering all of the above factors, the appropriate test to run was the Mann-Whitney test which is the non-parametric alternative for the independent t-test.
The Mann-Whitney test replaces all dependent variable measurements with their rank numbers (1 to n for n sample size). Higher scores measured get higher rank numbers. The H_0 of this test is that the mean rank of the dependent variable between the two samples is equal. If the grouping variable (in our case the use of OAS or OAS + RESTalk) does not affect the ratings, then the mean ranks should be roughly equal for both groups. The alternative hypothesis (H_1) is that the mean rank of the dependent variable is different across the two samples. Thus, rejecting the H_0 and accepting H_1, when $P-value < 0.05 = \alpha$, means that there is **statistically significant difference** in the dependent variable (grade or time) depending on the used API documentation. The level of significance ($P-value$) refers to the accepted level of probability that the observed result is a false positive, i.e., it is due to chance. It is usually conventionally set to 5% or 1% [27]. The power refers to the probability of false negative, i.e., accepting the H_0 when there is actually a difference between the results. The most commonly accepted level of power is 80% or above [27].

7.3.2 Experiment Results

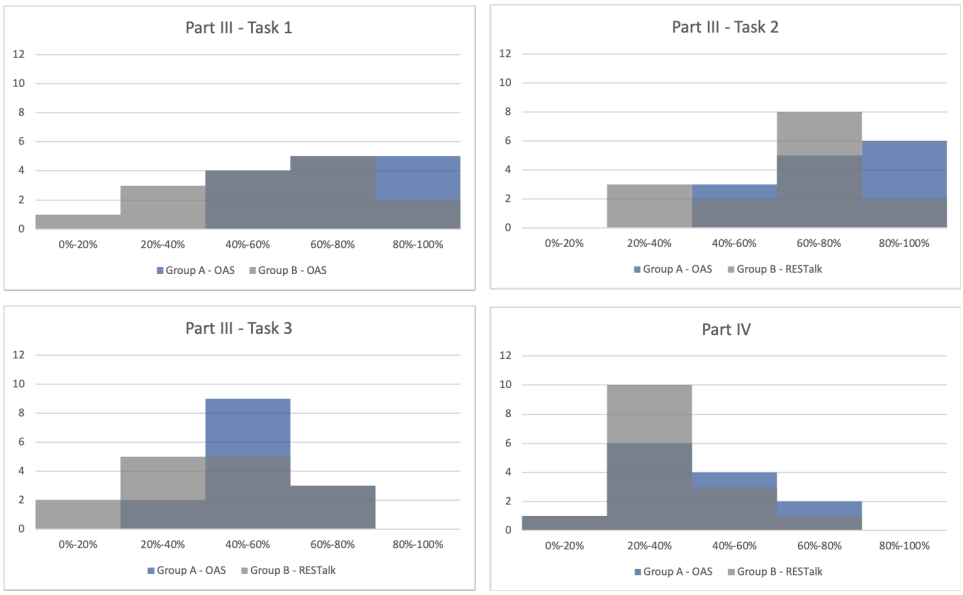
Before analysing the data we have preprocessed it to ensure its completeness and correctness. We have removed the incomplete data from students submitting just Part I and Part II of the experiment (we have removed 2 participants in Experiment 1 and 5 participants in Experiment 2 on these grounds). The number of participants stated above is after such data cleansing. We have also noticed missing data points due to system error resulting in some of the answers to tasks not being recorded or being only partially recorded by iCorsi, as well as cases where participants finished some tasks the day after the experiment thus invalidating the completion time measure. In these cases, we have kept the valid participant data points and only removed the missing data points from the analysis (pairwise deletion). The missing data points can be classified as missing data at random (MAR), i.e., the missing data is not connected with its hypothetical value or the values of other variables. Using listwise deletion (deleting the complete case) was not feasible due to the small sample size, neither was replacing the missing values with mean values due to the large number of missing data (6 data points in Experiment 1 and 21 data points in Experiment 2 mainly referring to the answers for determining the correctness/completeness grade) [232].

In the next subsections, for each of the experiments and the pilot study we are going to provide histograms and descriptive statistics measures to describe the main features of the collected data in quantitative terms.

Experiment 1 Results

We use histograms to show the distribution of the data regarding the grading of the correctness and completeness of the answers (see Fig. 7.8), as well as the completion time per task (see Fig. 7.9). To show the differences between Group A and Group B we use overlapping histograms where Group A, which always used the OAS documentation to perform the tasks, is shown in blue and Group B, which in addition to OAS documentation also had black&white non-interactive RESTalk diagrams available from Task 2 on, is shown in light gray with the overlap being shown in dark gray.

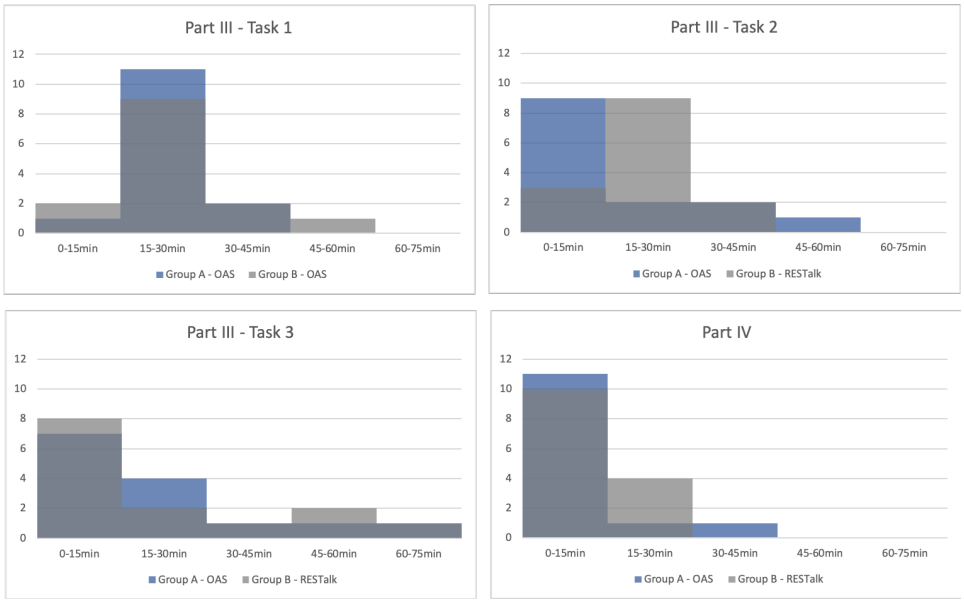
Figure 7.8. Distribution of Correctness and Completeness Grades per group in different tasks in Experiment 1



To visualise the distribution of the correctness and completeness grade (see Fig. 7.8), we use a histogram with each bin containing a range of 20% in the grade. While in Tasks 1 and 2 the grades are distributed between all the bins with some shift towards the right, i.e., higher correctness and completeness of the answers, for Task 3 and Part IV the shift is more towards the middle/left which shows lower correctness and completeness of the answers, with no answer being more then 80% correct/complete. For Part IV particular aggregation

of the responses in the 20%-40% bin is evident for Group B. If we analyse the difference in distribution between the two groups, higher grades are noticeable in Group A.

Figure 7.9. Distribution of Completion Time per group in different tasks in Experiment 1



In the histograms regarding the task completion time (see Fig. 7.9), we have used a range of 15 min per bin. Contrary to the histograms in Fig. 7.8, here the desired result is greater shift towards the left, which is more noticeable after the warm-up with Task 1 where the distribution for both groups is rather similar. For Task 2, Group A has shorter completion time, while for the following two tasks the distribution between the groups is rather similar.
In Tab. 7.2 we provide descriptive statistics metrics per group for each of the tasks. We provide the mean with a 95% confidence interval, as well as the median, together with the minimum and maximum value. Although the RESTalk diagrams were available from Task 2 on for Group B, 47% of the participants stated that they have not used them to answer the tasks. Therefore, in Tab. 7.2 we also provide the descriptive statistics regarding participants who have used the RESTalk diagrams and participants who did not use them (all participants from group A as well as the ones from Group B who stated that they have not made use of the diagrams).
From Tab. 7.2 we can notice that it is Task 2 the one with the highest mean regarding the correctness/completion grade (82% ± 7% for Group A and 70% ±

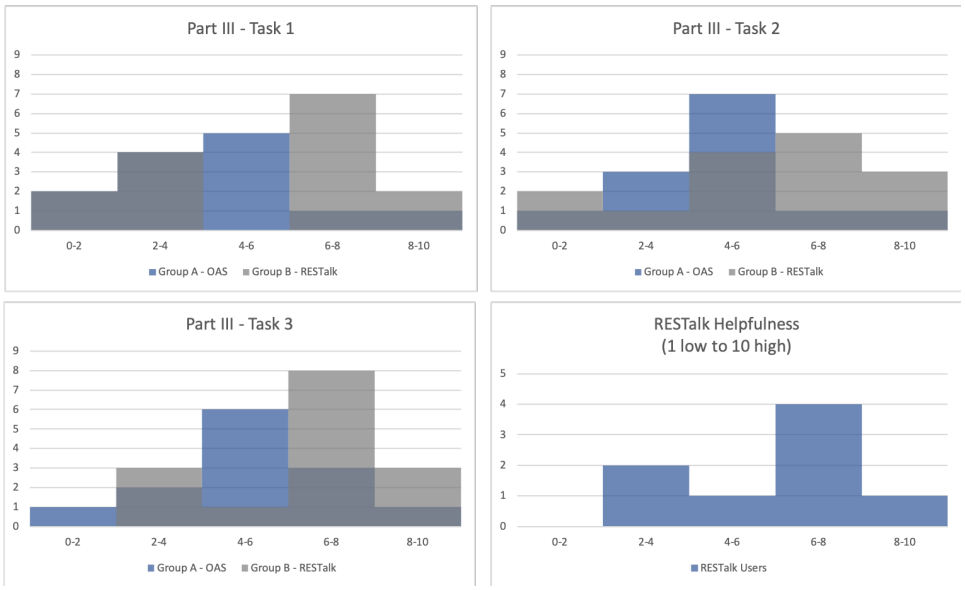
Table 7.2. Descriptive statistics about the correctness/completeness grade metric and completion time metric in Experiment 1

		Part III - Task 1		*Part III - Task 2*		*Part III - Task 3*		*Part IV*		*Total*	
	No. Participants	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**
Group A - OAS	***14***										
Mean with 95% CI		77% ± 11%	24 min ± 3 min	82% ± 7%	20 min ± 7 min	50% ± 7%	22 min ± 9 min	41% ± 9%	13 min ± 3 min	63% ± 5%	81 min ± 22 min
Median		75%	24 min	80%	12 min	50%	16 min	39%	13 min	61%	65 min
Min		50%	13 min	60%	10 min	33%	7 min	18%	7 min	52%	37 min
Max		100%	39 min	100%	57 min	75%	70 min	71%	30 min	87%	196 min
Group B - RESTalk + OAS	***15***										
Mean with 95% CI		57% ± 13%	25 min ± 6 min	70% ± 10%	20 min ± 4 min	43% ± 10%	27 min ± 11 min	34% ± 6%	12 min ± 3 min	51% ± 8%	85 min ± 13 min
Median		50%	24 min	80%	19 min	50%	15 min	32%	12 min	54%	74 min
Min		13%	9 min	40%	11 min	17%	12 min	14%	4 min	27%	53 min
Max		100%	57 min	100%	40 min	75%	82 min	61%	30 min	71%	141 min
RESTalk not used	***21***										
Mean with 95% CI		N/A	N/A	79% ± 7%	20 min ± 5 min	50% ± 7%	23 min ± 7 min	38% ± 6%	13 min ± 2 min	56% ± 5%	55 min ± 9 min
Median		N/A	N/A	75%	21 min	33%	15 min	32%	10 min	58%	48 min
Min		N/A	N/A	40%	13 min	25%	13 min	21%	4 min	24%	31 min
Max		N/A	N/A	96%	31 min	50%	82 min	61%	30 min	82%	105 min
RESTalk used	***8***										
Mean with 95% CI		N/A	N/A	67% ± 15%	20 min ± 4 min	36% ± 8%	30 min ± 18 min	35% ± 10%	12 min ± 8 min	46% ± 9%	63 min ± 17 min
Median		N/A	N/A	80%	15 min	50%	15 min	36%	13 min	45%	50 min
Min		N/A	N/A	40%	10 min	17%	7 min	14%	5 min	32%	40 min
Max		N/A	N/A	100%	57 min	75%	70 min	71%	30 min	64%	114 min

*Only OAS was used in both groups for Task 1

10% for Group B), while Part IV has the lowest mean regarding the completion time (13 min ± 3 min for Group A and 12 min ± 3 min for Group B). The average correctness/completeness grade between all four tasks is 63% ± 5% for Group A and 51% ± 8% for Group B. The average of the total completion time of the experiment in Group A is 81 min ± 22 min, while in Group B it is 85 min ± 13 min. The relative situation does not change much if we split the participants based on their actual declared use of the RESTalk diagrams instead of based on their availability taking into consideration only the last three tasks. Also in this case the group which did not use RESTalk has higher average grade of 56% ± 5% and lower average completion time 55 min ± 11 min with respect to the group of participants who claim to have used the diagrams (46% ± 9% and 63 min ± 17 min respectively).

Figure 7.10. Distribution of the Perceived Task Difficulty per group in different tasks in Experiment 1 as well as rating of RESTalk's helpfulness for solving the tasks



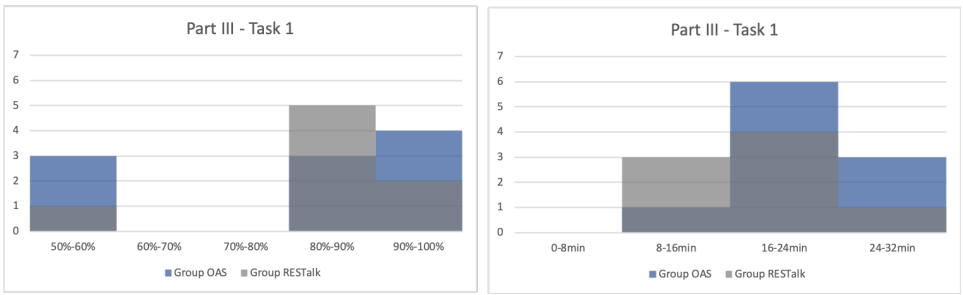
After completing all the tasks, the participants were asked to evaluate the perceived difficulty of discovering the appropriate sequence of interactions with the Imgur API in the available documentation in each task. In Fig. 7.10 we show the distribution of their answers in each group, with 1 meaning "very easy" and 10 meaning "very difficult". We can see that in general the perceived difficulty increases going from Task 1 to Task 3, especially for Group B which in general evaluated the tasks as more difficult (average rate of 7) then Group A (average

rate of 5). This is inline with the results shown in Fig. 7.8 and Fig. 7.9. In the lower right corner in Fig. 7.10 we also show a histogram of the evaluation of the helpfulness of RESTalk diagrams in solving the tasks by the 8 participants who decided to use the available diagrams, with 1 meaning "not helpful at all" and 10 meaning "very helpful". The average score of RESTalk based on this question is 6 and participants stated that the diagrams mostly helped them in solving Task 3 and Part IV. When one participant who decided not to use the RESTalk diagrams was asked how the API documentation can be improved he stated "Probably, recommend some methods that frequently follow other methods (i.e. linking methods together) for instance when you create a gallery, you probably want to add an image", which at the end is the goal of RESTalk. But when asked why he did not use the diagrams he answered that "the visual models were kinda complicated at first, so I emphasized on the documentation".
As in this experiment the participants were not provided any introduction to RESTalk before the experiment, we asked them which visual constructs used in the models they were unsure how to interpret. Their responses pointed to the "X" in the exclusive gateways and the dotted lines of the hyperlink flow. When asked for suggestions on how to improve the models one participant proposed using colours and adding the state of the client after each response.

Pilot Study Results

As mentioned earlier, given the limited time for conducting the pilot study, the goal was not running through all the tasks and obtaining data on the same, but rather obtaining feedback. Nonetheless, we do present the data we have obtained. All participants in the study, except for two participants from the RESTalk group, managed to complete the first task. The distribution of the completeness and correctness of the Task 1 and the time completion is available in Fig. 7.11.

Figure 7.11. Distribution of Correctness and Completeness Grades and Completion Time per group for Task 1 during the pilot study



Regarding the results from Task 1, as noticeable in Fig. 7.11, most of the participants managed to identify the correct or almost correct sequence of interactions, taking between 8 and 28 minutes to do so. The group who could use RESTalk for the task (Group B) was faster in completing the task with an average completion time of 16 min ± 4 min with 95% confidence interval, compared to the group (Group A) who only had OAS documentation available which had an average completion time of 21 min ± 2 min. The average correctness and completion grade in the RESTalk group was 89% ± 9% while in the OAS group it was 85% ± 11%. Task 2 was completed by only one member of Group A and 7 members of Group B.
The experiment was followed by a discussion as well as a formal survey which has been filled in by 13 of the participants, out of which 61% stated to have used the RESTalk diagrams when available. Out of the 8 participants who claimed to have used RESTalk diagrams, 75% rated it with a grade over 7 on a 1 to 10 scale where 1 stands for "not helpful at all" and 10 for "very helpful". What they found most useful regarding the models were the connections between the endpoints and the one page overview of a vast API endpoints. The main reason for not using the diagrams stated by the remaining 39% of the participants was the size of the diagrams being characterised as "too big to navigate fast enough".
During the discussion valid points have been brought out, which we implemented when feasible before running Experiment 2. One of the constructive advises we did implement was splitting the task description into a list of steps to facilitate task understanding. Another one was adding a short title for the functionality of each request. Another valuable feedback was that the amount of new information to be processed in terms of REST APIs knowledge, Imgur domain knowledge, RESTalk knowledge etc. can be too much for the students. To mitigate this risk, contrary to Experiment 1 set up, we have decided to provide an introduction to all of these fields during a lecture that preceded Experiment 2.

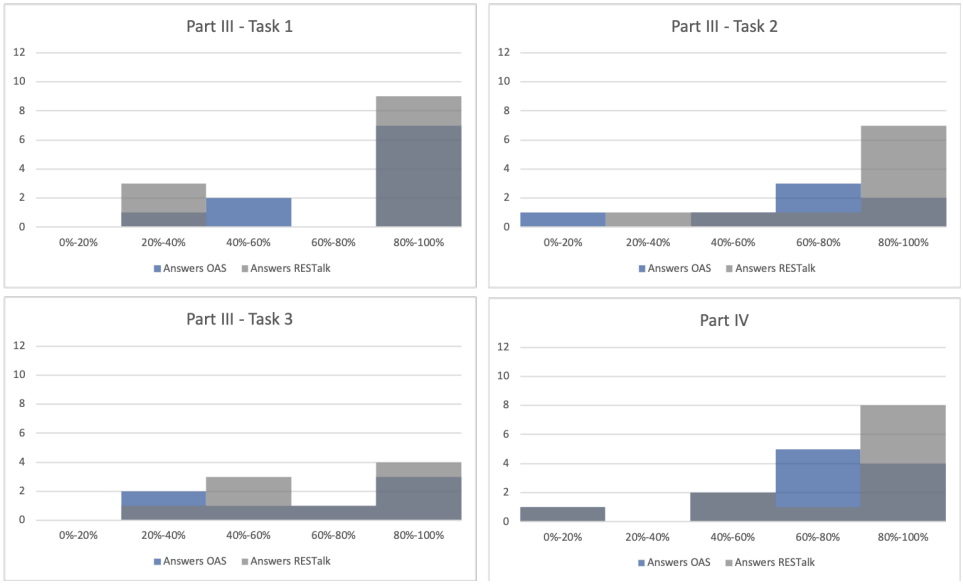
Experiment 2 Results

In Experiment 1 we used Task 1 as a baseline between the groups, however the low actual use of the diagrams made us believe that as all the students got familiar to the OAS documentation during the first tasks they just continued using it for the next tasks as well to avoid the effort of learning a new visual documentation. Therefore, in Experiment 2 we used as a baseline Part II in Appendix B, where we asked one open-ended question and one multiple-choice question to evaluate participants' Imgur domain knowledge, as well as one multiple-choice question to evaluate their REST APIs domain knowledge. The obtained knowl-

edge is balanced enough between the groups with average grade of the three questions of 6.4 ± 1.1 in Group A and 6.1 ± 1 in Group B on a scale of 10 (where 10 is the highest grade). The range of grades in Group A was between 3.5 and 9.5 while for Group B it was between 3 and 8.5. These numbers were similar to the auto-evaluation of the participants' understanding of the warm-up session preceding the experiment, which was rated on average 6.2 ± 1.1 in Group A and 5.8 ± 1.5 in Group B.

As mentioned earlier, in Experiment 2 we used a within-subject design where Group A used RESTalk diagrams for Tasks 2 and 3, while Group B used them for Task 1 and Part IV. Thus, we do not show the results per group, but rather per availability of RESTalk diagrams, as only three participants claimed that they did not use the diagrams at all in resolving the tasks. In Fig. 7.12 we show the distribution of the correctness and completeness grade when RESTalk is not available (blue colour) vs. when it is available (gray colour) per task. Most of the participants in both cases scored high in Task 1, while in Task 2 and Part IV there is greater shift to the right for the participants who could use the RESTalk diagrams. The distribution is rather balanced between the two cases regarding Task 3.

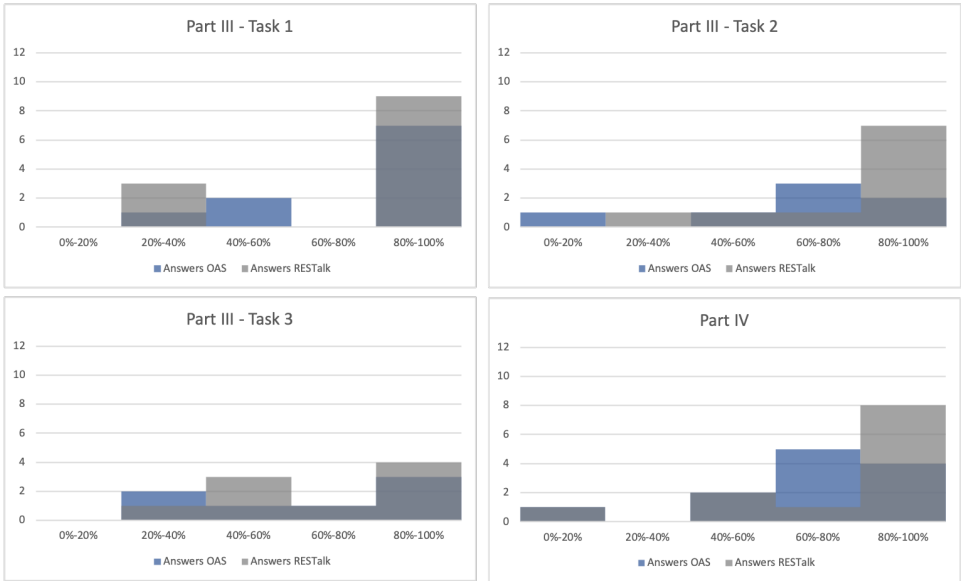
Figure 7.12. Distribution of Correctness and Completeness Grades per availability of RESTalk diagrams in different tasks in Experiment 2



In Fig. 7.13 we show the distribution of the completion time per task for each of the two cases. As evident from the histograms none of the participants took

more then one hour for completing a task. In Task 3, as for the correctness and completeness grade discussed before, also the distribution of the completion time between the two cases is rather similar. In Task 2 all of the participants who had the RESTalk diagram available finished the task in 15 to 30 min, while the ones who did not have it available took up to 45 min. The fastest and rather balanced completion time between the two cases is evident in Part IV.

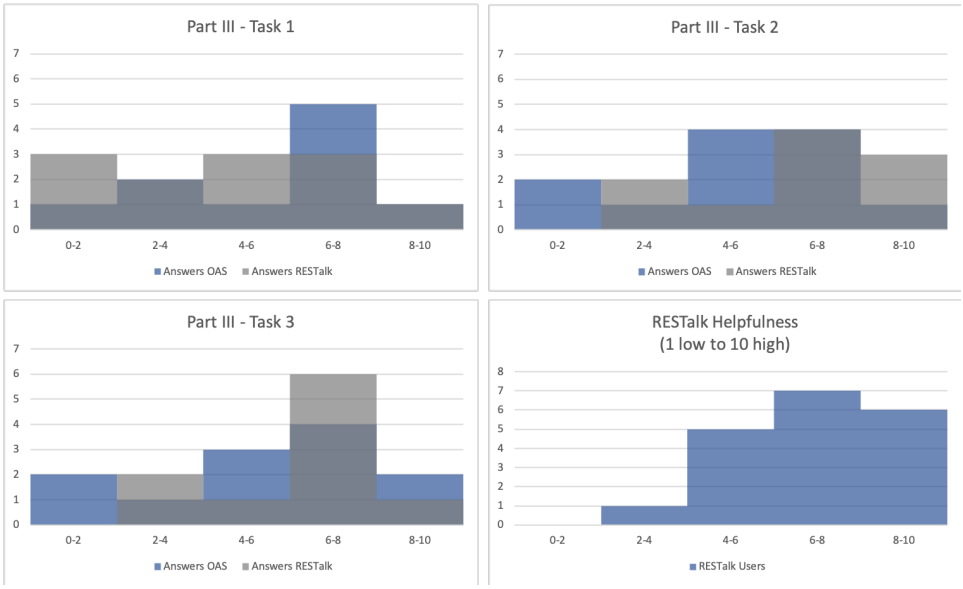
Figure 7.13. Distribution of Completion Time per availability of RESTalk diagrams in different tasks in Experiment 2



The descriptive statistics metrics for Experiment 2 are available in Tab. 7.3, again divided in two cases based on the availability of the RESTalk diagrams. Due to the mentioned missing data points we also report on the number of actual data points for each task. The average correctness and completeness grade between all four tasks is 77% ± 11% with average total completion time of 84 min ± 9 min for the participants who did not have RESTalk diagrams available, and 79% ± 7% and 81 min ± 9 min respectively for the participants who had RESTalk diagrams available. The greatest difference in the correctness and completeness grade between the two cases is noticeable for Task 2 where the average is 70% ± 20% for the participants who did not have RESTalk diagrams available, and 83% ± 14% for the participants who had RESTalk diagrams available, however this is also the task with the greatest difference in the number of data points between the two cases. Time-wise the greatest difference is evident in Task 1 with average completion time of 28 min ± 6 min when working without RESTalk diagrams and

22 min ± 7 min when working with RESTalk diagrams.
The average perceived task difficulty when RESTalk diagrams are not available is equivalent between different tasks and amounts to 6 points on a 10 point scale where 10 stands for “very hard”. When RESTalk is available Tasks 2 and 3 are perceived as more difficult (average grade of 7) compared to Task 1 (average grade of 5). In Fig. 7.14 we show the distribution of the perceived difficulty.

Figure 7.14. Distribution of the Perceived Task Difficulty in different tasks in Experiment 2 as well as rating of RESTalk’s helpfulness for solving the tasks



In the lower right of the same figure we also show the distribution of the perceived helpfulness of RESTalk for solving the tasks. Nobody considered RESTalk not helpful at all and just one participant give it a grade lower then 6 while 26% of the participants considered it very helpful with a grade of 10. The average grade of RESTalk’s helpfulness between all participants was 8. Based on their open ended answers what they found most helpful regarding the RESTalk diagrams was that they show sequences of requests, and flow of data. Many appreciated the link to the documentation feature and the additional information displayed when hoovering over a request. Two participants also mentioned that they appreciated the colour coding in the diagrams. One participant commented that “they were really easy to use and understand. They also gave a global vision about the API in order to retrieve the data”. One interesting proposal for an improvement was: “I would group all the similar tasks under a labelled folder that the user can expand since seeing everything together is a bit confusing”. From

Table 7.3. Descriptive statistics about the correctness/completeness grade metric and completion time metric in Experiment 2

		Part III - Task 1		*Part III - Task 2*		*Part III - Task 3*		*Part IV*		*Total*	
	No. Participants	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**	**Correctness / Completeness**	**Completion Time**
Answers OAS	***12***										
Mean with 95% CI		81% ± 13%	28 min ± 6 min	70% ± 20%	24 min ± 6 min	74% ± 21%	23 min ± 7 min	75% ± 14%	10 min ± 3 min	77% ± 11%	84 min ± 9 min
Median		90%	24 min	67%	21 min	80%	19 min	80%	10 min	79%	86 min
Min		40%	10 min	20%	9 min	37%	10 min	10%	1 min	30%	41 min
Max		100%	46 min	100%	45 min	100%	58 min	100%	23 min	100%	106 min
No. Datapoints		10	12	7	12	7	12	12	11	12	12
Answers RESTalk + OAS	***12***										
Mean with 95% CI		77% ± 16%	22 min ± 7 min	83% ± 14%	23 min ± 2 min	73% ± 18%	24 min ± 7 min	81% ± 15%	14 min ± 5 min	79% ± 7%	81 min ± 9 min
Median		90%	19 min	92%	24 min	67%	20 min	95%	13 min	76%	81 min
Min		30%	10 min	33%	17 min	33%	9 min	20%	6 min	63%	54 min
Max		100%	49 min	100%	29 min	100%	48 min	100%	31 min	100%	106 min
No. Datapoints		12	12	10	12	9	11	12	10	12	12

the three participants who decided not to use the visual diagrams, two stated that the reason was that they found enough information in the OAS documentation and one stated that "It was easier to search the API docs than leverage the information in the visual models. The visual model could have been more interactive - auto zoom, collapse boxes, highlight specific sections (based on reachability of flow)".

7.3.3 Statistical Significance Analysis

While in the previous section we provided a quantitative summary of the experiment results, in this section we show the results of the inferential statistics test we have described in the Metrics and Analysis subsection in Sec. 7.3.1. On the raw data for the correctness/completeness grade and completion time per task and in total for each experiment, we have run the Mann-Whitney Test using IBM SPSS Statistics Version 24. The goal was to test the significance of the differences between the different treatments. In Experiment 1 we excluded Task 1 from the analysis as it was used as baseline where both groups had only OAS documentation available for solving the task.
The first research question that we wanted to answer is ***RQ1:*** *Does RESTalk help to discover complete and correct sequences of requests given a goal?* Thus, our null hypothesis H_0 for RQ1 is that there is no difference in the mean completeness/-correctness grade when OAS documentation is used and when OAS documentation is complemented with RESTalk diagrams.
The second research question that we wanted to answer is ***RQ2:*** *Does RESTalk help to discover the required sequences of requests faster?* Thus, our null hypothesis H_0 for RQ2 is that there is no difference in the mean completion time when OAS documentation is used and when OAS documentation is complemented with RESTalk diagrams.
In Tab. 7.4 we show the mean rank per task and per group for both experiments. The largest difference between the mean rank of the OAS and RESTalk groups in Experiment 1 is noticeable in Task 3 Grade (difference of 6.73) and in the Total Grade (6.21). However, as evident in Tab. 7.5, this difference is still not statistically significant as the two-tailed test statistics is 0.053 for Task 3 Grade and 0.081 for Total Grade. For the difference to be statistically significant, and thus to reject the H_0, the test statistics should be smaller then or equal to the significance level which we have set to 0.05 to have a confidence interval of 95%. The difference in the mean ranks is even less evident in Experiment 2 where the largest difference is noticeable in Task 1 Time (3.66) and Task 2 Grade (2.91). In this case as well the two-tailed test statistics of 0.213 for Task 1 Time and

Table 7.4. Mann-Whitney mean ranks per group and per task for Experiments 1 and 2

Experiment 1	Group	N	Mean Rank	Sum of Ranks	*Experiment 2*	Group	N	Mean Rank	Sum of Ranks
					Task 1 Grade	OAS	10	11.85	118.5
						RESTalk	12	11.21	134.5
						Total	22		
					Task 1 Time	OAS	12	14.33	172
						RESTalk	12	10.67	128
						Total	24		
Task 2 Grade	OAS	21	16.43	345	Task 2 Grade	OAS	7	7.29	51
	RESTalk	8	11.25	90		RESTalk	10	10.2	102
	Total	29				Total	17		
Task 2 Time	OAS	20	13.4	268	Task 2 Time	OAS	12	12.13	145.5
	RESTalk	8	17.25	138		RESTalk	12	12.88	154.5
	Total	28				Total	24		
Task 3 Grade	OAS	21	16.86	354	Task 3 Grade	OAS	7	8.5	59.5
	RESTalk	8	10.13	81		RESTalk	9	8.5	76.5
	Total	29				Total	16		
Task 3 Time	OAS	20	14.23	284.5	Task 3 Time	OAS	12	11.42	137
	RESTalk	8	15.19	121.5		RESTalk	11	12.64	139
	Total	28				Total	23		
Part IV Grade	OAS	20	15	300	Part IV Grade	OAS	12	11.25	135
	RESTalk	8	13.25	106		RESTalk	12	13.75	165
	Total	28				Total	24		
Part IV Time	OAS	19	14.66	278.5	Part IVTime	OAS	11	9.68	106.5
	RESTalk	8	12.44	99.5		RESTalk	10	12.45	124.5
	Total	27				Total	21		
Total Grade	OAS	21	16.71	360	Total Grade	OAS	12	12.67	152
	RESTalk	8	10.5	75		RESTalk	12	12.33	148
	Total	29				Total	24		
Total Time	OAS	20	13.6	284.5	Total Time	OAS	12	13.33	160
	RESTalk	8	16.75	121.5		RESTalk	12	11.67	140
	Total	28				Total	24		

Table 7.5. Mann-Whitney Test statistics for Experiments 1 and 2

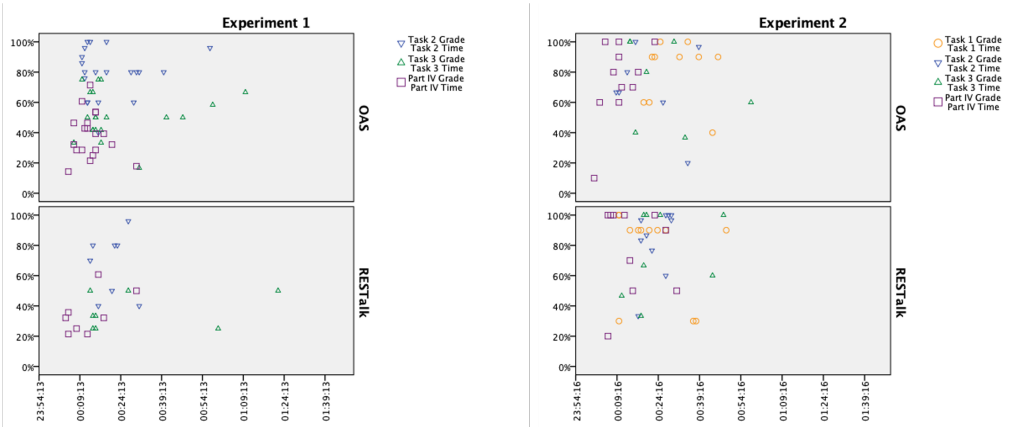
	Task 1 Grade	Task 1 Time	Task 2 Grade	Task 2 Time	Task 3 Grade	Task 3 Time	Part IV Grade	Part IV Time	Total Grade	Total Time
Experiment 1										
Mann-Whitney U			54	58	45	74.5	70	63.5	48	62
Z			-1.51	-1.121	-1.94	-0.281	-0.511	-0.666	-1.757	-0.917
Exact Sig. (2-tailed)			0.136	0.253	0.053	0.852	0.625	0.522	0.081	0.374
Experiment 2										
Mann-Whitney U	56.5	50	23	67.5	31.5	59	57	40.5	70	62
Z	-0.253	-1.273	-1.183	-0.26	0	-0.432	-0.892	-1.024	-0.116	-0.578
Exact Sig. (2-tailed)	0.8	0.213	0.255	0.788	1	0.662	0.383	0.322	0.921	0.58

0.255 for Task 2 Grade is not smaller then the significance level. As evident from Tab. 7.5 in both experiments, in none of the analysed tasks and metrics, the identified difference is statistically significant which means that H_0 is accepted for both RQ1 and RQ2, i.e., based on the samples the use of the RESTalk diagrams as complementary documentation to the OAS documentation does not significantly affect the correctness and completeness of the identified sequence of interactions, nor does it significantly affect the task completion time.

7.3.4 Discussion

As emphasised in [21], without a clear discussion of the evaluation results an evaluation work is meaningless as the reader is left to decide for himself what is the claim of the work. Avoiding the sin of reasoning and the sin of exposition is not an easy task, but we try our best to make a sound claim regarding the evaluation results.
In Fig. 7.15 we have plotted the metrics regarding RQ1 and RQ2 for each task depending on the effectively used API documentation (note that this is not the same as available API documentation in Experiment 1) for each experiment so that it is easier to discuss the data both on experiment level and between experiments. In Experiment 1 greater clustering based on the tasks both in terms of correctness/completion grade and in terms of time is noticeable, with higher grades for Task 2 and longer completion time for Task 3. The clustering tendencies are similar between the OAS group and the RESTalk group. Although Fig. 7.15 as well as Tab. 7.2 show what can seem as an important increase in correctness and completeness of the solutions when RESTalk is not used, especially for Task 2 and Task 3 (mean of 79% ± 7% for OAS group Task 2 vs 67% ± 15% for RESTalk group Task 2 and mean of 50% ± 7% for OAS group Task 3 vs 36% ± 8% for RESTalk group Task 3), the statistical analysis procedure has discarded these differences as not significant. The observed actual difference can be due to different understanding of the REST domain and the Imgur domain between the two groups, evident in Task 1 when everyone used only the OAS documentation for the solution and Group A had a mean grade of 77% ± 11% compared to 57% ± 13% for Group B.
The mentioned clustering of the results per task is not evident in Experiment 2 as per Fig. 7.15. Completion time wise Task 1 seems a bit shifted to the right compared to other tasks, but that can also be understood as a warm up task where the participants were trying to understand how to use the available API documentation. When it comes to comparing the results between the OAS and the RESTalk group greater concentration of the data points in the top left angle is

Figure 7.15. Scatterplot of the correctness/completeness grade and completion time per task per group in Experiment 1 and Experiment 2



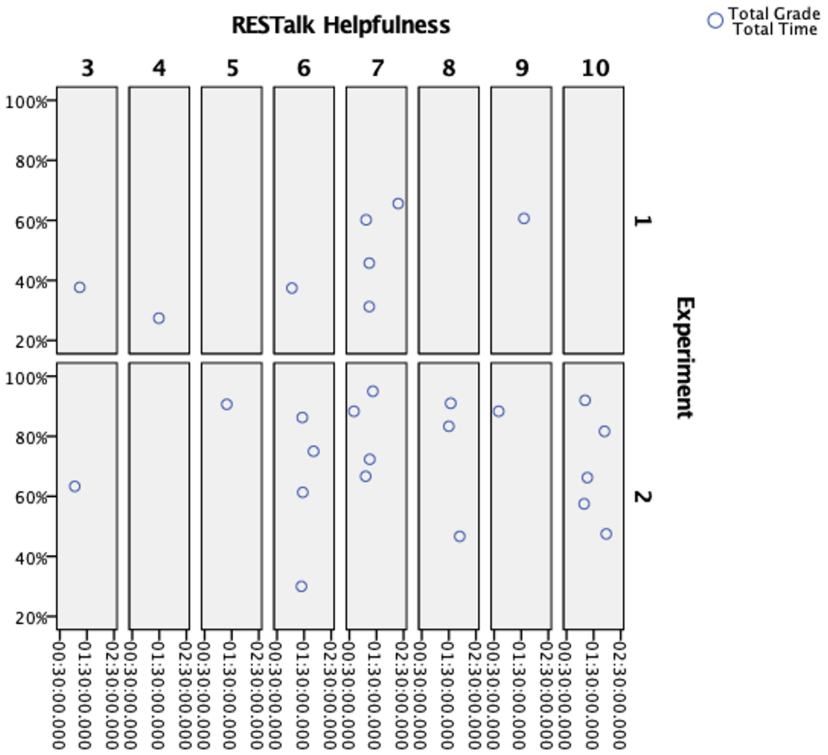
noticeable. From Tab. 7.3 this is especially evident for Task 2 regarding the grade (79% ± 20% for the OAS group vs 83% ± 14% for the RESTalk group) and for Task 1 regarding the completion time (28 min ± 6 min for the OAS group vs 22 min ± 7 min for the RESTalk group). However, as was the case with Experiment 1, also in Experiment 2 the statistical analysis procedure has shown that such differences are circumstantial and are not statistically significant.

If we are to discuss the difference in results between the two experiments, better results are evident in Experiment 2, both regarding correctness and completeness of the solutions and regarding completion time (see Fig. 7.15), regardless of the type of API documentation used. This is most probably due to higher experience with the use of REST APIs among the participants in Experiment 2 (9 out of 24 students) compared to Experiment 1 (3 out of 29 students) as well as due to the introductory lesson to REST APIs and RESTalk before the controlled experiment. Both Experiment 1 and Experiment 2 show that the use of RESTalk as a complementary documentation to OAS documentation does not help in a statistically significant way to discover complete and correct sequences of requests given a goal, nor does it help to discover them in a time efficient way (RQ1 and RQ2). However, these conclusions are relevant for novice users of the REST API paradigm, not of experienced REST API developers who might leverage more from the visual documentation. Such hypothesis needs to be tested with further empirical evidence.

Regarding RQ3 which refers to the perceived usefulness of the RESTalk diagrams, we have noticed that the interest of the participants to use the RESTalk diagrams has significantly increased between Experiment 1 (only 8 out of 29 participants

used them) and Experiment 2 (21 out of 24 participants used them). This can be due to the introductory lesson on RESTalk provided to students before the controlled experiment in Experiment 2, but it can also be due to the increased interactivity of the diagrams themselves in Experiment 2 where they also contained links to the OAS documentation and highlighted the parameter discovery options when hoovered over, functionalities which were appreciated by the participants based on their answers to the open ended questions. Another disincentivizing factor in Experiment 1 could also have been the fact that three different diagrams were provided to the participants and they were expected to identify the correct one to follow, while in Experiment 2 only one diagram was provided.

Figure 7.16. Scatterplot of the relation between RESTalk's helpfulness rating and correctness/completeness grade and completion time in Experiment 1 and Experiment 2



With the increased use also the RESTalk perceived helpfulness rates raised from an average of 6 in Experiment 1 to an average of 8 in Experiment 2 on a scale from 1 to 10 with 10 being very helpful. In Fig. 7.16 we have plotted the different RESTalk helpfulness rating by different participants in Experiment 1 and Experiment 2 with relation to their average correctness/completeness grade across the

tasks and the total completion time. While in Experiment 1 it might seem like lower RESTalk grades were coming form participants whose solutions received lower grading (thus whose understanding of the correct solution was lower), this does not seem to be the case in Experiment 2. However, due attention also needs to be paid to the fact that the sample size of the participants who effectively used RESTalk in Experiment 1 is rather small (8 participants). Nonetheless, almost half of the participants in Experiment 2 rated RESTalk with 8 or higher.

Table 7.6. Mann-Whitney Test statistics for RESTalk's Helpfulness rating between Experiments 1 and 2

	Experiment	**N**	**Mean Rank**	**Sum of Ranks**
RESTalk Helpfulness	1	8	10.88	87
	2	19	15.32	291
	Total	27		
Mann-Whitney U	51			
Z	-1.355			
Exact Sig. (2-tailed)	**0.187**			

Although comparing the difference in RESTalk ratings between the two experiments was not our initial interest, we did run the Mann-Whitney test on the rating. The results are shown in Tab. 7.6. As evident the two-tailed test statistics of 0.187 is not smaller then the significance level of 0.05 which means that the identified difference is not statistically significant. Further experiment, with additional targeted questions would be needed to establish the impact of the diagram interactiveness on the participants perceived helpfulness of the same.

Threats to Validity

"Threats to the validity of empirical studies and of their results are unavoidable" [78]. Thus, identification and clear documentation of the threats to validity of a controlled experiment is important for avoiding the sin of exposition and ensuring good understanding of the claims of the experiment results. We have identified the following threats to the controlled experiments we have conducted: **Threats to Construct Validity** - For the controlled experiment we have used second year bachelor students which might not be representative of professionals due to their lack of REST API experience. Another threat comes from the selected API used in the tasks whose OAS documentation might not be representative of the REST API documentation population. The objectiveness of the participants evaluating the helpfulness of the RESTalk diagrams might have been impacted

by the fact that part of the research group conducting the experiment is directly involved in the evaluation of their course work which in combination with the non-anonymity of their answers might have made them reluctant to provide objective and sincere evaluation. Another threat comes from the inherent subjectiveness of the grading used to determine the correctness and completeness of the solutions of the participants. To mitigate this risk we created a systematic grading scale and systematically applied it on all solutions.

Threats to Internal Validity - When planning for the experiment, we have tried to mitigate the risk of the difference in the results being due to the heterogeneity of the subjects by balancing the different groups based on their grading of previous course assignments. However, the results of the baseline evaluation (Task 1) during Experiment 1 show that background knowledge or maturation (learning speed) differences did exist. To mitigate this risk in Experiment 2 we have used within-subject design and an introductory lesson. The baseline evaluation results from Part II show rather balanced groups in this experiment. However, an internal threat in Experiment 2, which does not hold in Experiment 1, is that the participants might have had access to the task solutions from students taking the course in the previous year.

Threats to Conclusion Validity - To mitigate this risk we have followed the instructions regarding the selection of an appropriate statistical test provided in [227] and checked that the assumptions for the use of the Mann-Whitney test were met in our experiments. Of course using a small sample size decreases the statistical power of the test. Furthermore, the missing data points due to system data recording error may also impact the power of the test.

Threats to External Validity - The results we obtained present limited generalizability since: they depend on the previous REST APIs knowledge and experience of the participants; the sample size of the population was rather small; all the tasks ware based on only one API (the Imgur API) and can be seen as simplistic for a real environment. Further experiments are needed to improve and delimit the generalizability of our results.

Although when designing the experiment we did try to take into consideration the different types of validity and mitigate the risks when possible, different threats to validity can conflict each-other. For instance using students enabled us to have a more controlled environment setting as opposed to using an online call for participants, but it has reduced the heterogeneity of the sample and possibly its representativeness of the real world targeted users of RESTalk.

7.4 Chapter Summary

In this chapter we have introspectively looked into RESTalk's design both in terms of its concrete graphical representation characteristics and in terms of its support for the REST constraints, and then had it evaluated externally through a controlled experiment. Although during the design of the visual notation we did our best to follow state of the art best practices to improve the cognitive effectiveness of the language, the summative analysis in terms of the 9 principles of the Physics of Notation theory showed how hard it is to balance the trade-off between the principles and that there is always room for improvement. In the current version of RESTalk we do not support the perceptual integration principle and we do introduce some level of symbol overload. However, our design decisions were supported by design rational, which although might have been suboptimal in some cases still lead to a language which some of the students defined as easy to understand. It does reflect most of the REST constraints emphasizing visually facets which are important in RESTful interactions, i.e., the discovery of resources, the respect of the uniform interface constrains with a predefined pairs of http methods and status codes. The controlled experiment did not identify statistically significant improvements in the quick identification of the correct sequence of interactions caused by the use of RESTalk in the API documentation. Nonetheless, it did show that users are more propense to give RESTalk diagrams a try when the diagrams dissect large APIs into smaller chunks thus pointing the users to a subset of the API and when the diagrams embed interactivity and links to the OAS documentation.

Chapter 8

Conclusions

8.1 Summary

We are living in a digital era, an era of interconnected systems where we are used to having quick access to data. And this is both in our private lives and at work. Interconnected systems can consist of multiple different technologies and APIs provide the necessary abstraction to connect such different technologies and make them exchange information. They allow developers not to reinvent the wheel and efficiently focus on innovation and development of new functionalities. The REpresentational State Transfer (REST) has gained its dominance as an API architectural style due to the uniform interface it promotes, its support of different data formats, among which the easily parsed and less verbose JSON, as well as it scalability and lower use of bandwidth. It is rare that a single interaction between the client and the REST API is sufficient to achieve a given goal, thus we witness real client to server conversations. Although such conversations happen in reality, our state of the art survey has showed us that there is no existing domain specific solution to document RESTful conversations. Thus, we have built this thesis around identifying and offering a possible solution for this gap.

8.1.1 Contributions

The nature of the contributions of this thesis goes from theoretical analysis of the REST domain and the benefits of modeling the behaviour of REST APIs, to more practical contributions through the design of a visual and textual DSL and the possible ecosystem around it. We will look into them in detail through the discussion of the individual research questions we have set in Chapter 1.

RQ1: *What are the entities and constraints that are needed to model the interactions with an API which is compliant with the REST architectural style?*

In Chapter 2 we studied in detail REST as an architectural style to determine which of its **entities** are strictly related to the behavioural aspect of REST APIs. While one of the main entities in REST is the resource and its representation, its importance in RESTful interactions is related to what is defined in OAS as an operation (the pair of HTTP method and resource URI) and the importance of the resource representation is related to the hyperlinks or parameters provided in the representation which are needed for future requests. Thus, the first class citizen entities in a RESTful interactions are the *request* and *response*. However, interactions with a REST API are not entirely independent from each-other as they need to happen in a given predefined order depending on the state of the resource and the goal of the client. To depict these richer forms of interaction Haupt, Leymann and Pautasso coined the term *"RESTful conversation"* which we extended to include not only single client to single server conversations, but also interactions where there can be multiple types of clients having an interrelated conversation with a given server. We also introduce the notion of composite conversations where there are several layers of API communication each with its own internal conversation, and the response of a request in the n^{th} layer depends on a sequence of responses from a server in the $(n+1)^{th}$ layer. That said, other important entities when modeling the behaviour of REST APIs are borrowed from the modelling of interactions, in our case from BPMN Choreographies, and refer to the control flow which determines the possible sequences of interactions and their divergence/convergence referred to as *sequence flow* and *gateways*. We provided a domain concept dictionary and a metamodel in Sec. 4.2.1.
Several of the REST architectural style **constraints** affect the properties of a RESTful conversation. The *stateless communication* constraint results in the conversation always being initiated by the client, but also being driven by client's decisions on which path to follow and when to stop the conversation. The server can influence the conversation through the resource representation it sends, but the server cannot drive the conversation forward. The *uniform interface* constraint determines the standardization of the form of the interactions taking place during a RESTful conversation. It dictates the mandatory elements of an interaction, such as the method, URI and response status code. The *client-server* constraint requires independent evolution of the client and the server. This means that ideally the evolution of the server should not change the RESTful conversation model of the API, or if it does it should expand the possible paths without changing existing paths which existing clients depend on.

RQ2: *What are the shortcomings of existing solutions for modelling REST APIs and how can those shortcomings be overcome?*

As discussed in RQ1, modelling the behaviour of REST APIs is bound to certain domain specific entities and constraints. As discussed in Sec. 3.1, domain specific solutions do exist for modelling the structure of REST APIs, the one striving to become a standard being the solution offered by the OpenAPI initiative, based on the Swagger 2.0 specification, and revised to what is today known as the OpenAPI Specification (OAS). The major revision of this specification to its version 3.0 in mid 2017 has acknowledged the importance of behaviour information by adding the new *links* object which can be specified for each operation. However this new object, does not provide a human-readable ready to use behavioural model, just possibly enough information to algorithmically build one. Thus, to the best of our knowledge, apart from the DSL proposed in this thesis, there is not yet a domain specific solution for modelling the behaviour of REST APIs. This is not to say that the behaviour of REST APIs has never been modelled before, but that non-domain specific standard notations, such as UML or BPMN Choreography diagrams, have been used as evident both from our state of the art survey in Sec. 3.1 and from our exploratory survey with industry practitioners described in Sec. 6.1. The main drawback of such solutions as discussed in Sec. 7.2, is the lack of REST specific constructs (such as the hyperlink flow) and the lack of emphasis in the representation of the important REST facets of the first-class citizens of a RESTful conversation, the request and response (such as the method, URI, status codes).
That said, the most important contribution of this thesis is RESTalk, the REST specific DSL which we have designed in an attempt to overcome the above stated drawbacks, and with the aim to propose a more concise and less verbose alternative to the mentioned non-domain specific solutions. We designed RESTalk following an iterative approach, using the feedback from the mentioned exploratory survey in Sec. 6.1 to improve and extend the language, but also by setting up challenges to model different use cases as discussed in Sec. 6.2 which revealed the need of new constructs. However, there is a well-known trade off between the expressiveness of the language and its understandability. We tried to address it by using the cognitive fit principle of the Physics of Notation theory and splitting the language into core and extended RESTalk, where the core targets novice users, while more experienced modelers who need to show more complex conversations can use the extended RESTalk. The creation of RESTalk and its constructs is explained in Chapter 4.

RQ3:*What type of a tool support can be built around a DSL for modeling REST APIs behaviour?*

Our exploratory survey in Sec. 6.1 has shown us that different targeted users have different preferences and requirements for adopting a DSL. While pen and paper is good enough for some, others condition the use of a DSL to tool support. Building an extensive tool support for a new DSL, such as RESTalk, is expensive both in terms of time and in terms of resources. Thus, we limited ourselves to contributing to the community an envisioned tooling ecosystem designed around RESTalk which integrates related existing solutions in the domain as discussed in Sec. 5.1. Naturally people are convinced more quickly to adopt the use of a model which they get for free from an existing documentation, such as OAS, or in the MDE spirit, a model which automatically generates code based on an input of the conversation path to follow. Such tools can be developed in the future.
For the time being, we only invested in the implementation of proof of concept tools that support the creation of RESTalk diagrams, i.e., RESTalk editors (Sec. 5.2), and a tool that uses RESTalk diagrams to visualize actual use of a REST API by mining its logs (Sec. 5.3).
In addition to the tools themselves, a valid contribution of this thesis is the innovative design of the textual editor for RESTalk which leverages on the use of a mining algorithm to facilitate the cognitive effort of the model designer who, in addition to the identification of the interactions, also has to think of the control flow and the graph layout of the RESTful conversation. The mining algorithm deduces the control flow from the textual DSL, which was intentionally designed to follow a log-like structure as described in Sec. 4.2.3. When used for sketching, such textual editor encourages the creation of user stories, which can come in handy during brain-storming sessions with other developers or with potential clients. The textual support of most of the core RESTalk is rather intuitive and does not require any keyword memorisation. Each possible path is stated as a list of interactions (method + URI + status code) with each interaction being stated in a new line. Alternative paths are lists of interactions separated with an empty line. The more visual constructs can be expressed with the textual DSL, the more steep the learning curve becomes, but that is an unavoidable trade-off as mentioned in RQ2.

RQ4: *How can REST API developers and API client developers benefit from modeling the API's behaviour?*

Designing and conducting controlled experiments with API developers to effectively evaluate the actual benefits from modelling the REST API behaviour and draw statistically valid conclusions is rather difficult in terms of reaching to the right subjects for the experiment and convincing enough of them to participate. Our attempt to reach to the REST API developers community at the beginning of our research through the exploratory survey described in Sec. 6.1 resulted with a total of 35 responses, out of which 26 were from industry, with a survey which was opened for 3 months. Although the number of responses was low and by design we did not aim at making statistical inference of the results, we did get the sense that the need of and benefits from modeling the API's behaviour have already been identified in industry. In fact 38% of them are already using UML sequence diagrams or in-house notations to visualize such models. Some of the benefits they see is increased productivity and increased design discussions and knowledge sharing.
As stated in the introduction in Chapter 1, the ultimate aim of the proposed DSL is facilitating and improving the understanding, design, and usage of REST APIs. The benefits stated by the survey respondents go in that direction. Additionally, to answer to this question in Sec. 1.3.4 we reflect upon a number of potential benefits which we have identified in different phases of the API life-cycle both when developing the server and when developing the client. In addition to fostering design discussions and common understanding during the API design phase, for the API provider it is also important to limit the supported behavior of the API in order to be able to ensure maintainability and evolution of the same. The number of permutations in the possible order of interactions grows exponentially even for a rather small API when there are no constraints on the allowed behaviour. Modelling the accepted behaviour of an API allows the service provider to set up such constraints. On the part of the client developers, behavioral models can facilitate the learnability of the API as they can be used as a complementary documentation to the existing OAS documentation. In large APIs the benefits would be even larger if instead of one behavioral models, there are several of them split in the right logical granularity.

RQ5: *How effective and efficient is the visual model created with RESTalk in facilitating the understanding and the use of a given API?*

While in RQ4 we discussed the abstract notion of modelling the behavior of a REST API, with this research question we wanted to focus on RESTalk and whether it can facilitate the understanding and the use of a given API. To that end, we have designed and conducted a controlled experiment with second year bachelor students as described in Sec. 7.3. As we were interested in the effect of the use of RESTalk, as a complementary documentation to OAS documentation, on the understanding and use of a given API we have set up tasks which required to use the Imgur API to reach a certain goal (e.g., upload an image and place it in a public gallery). We then measured the correctness and completeness of the responses as well as the completion time. We repeated the experiment twice, with two different groups of students and with somewhat different experiment design. Although the quantitative results did not show any statistically significant difference in the above stated metrics between the control group and the experimental treatment group in any of the experiments, we did get positive feedback from the students regarding the usefulness of RESTalk, with an average vote of 6 out of 10 in Experiment 1 and 8 out of 10 in Experiment 2. We are aware that such results can be biased by the fact that the results of the experiment were considered as an in-class assignment grade. Nonetheless, the increased perceived usefulness of the diagrams and the increased actual use of the same in Experiment 2 compared to Experiment 1, has provided us some useful insights. Namely, what changed in Experiment 2 was that students were provided a quick RESTalk tutorial before the experiment and the experimental treatment group was made available only one diagram per task (as opposed to 3 diagrams to choose from in Experiment 1) which we believe has convinced more students to actually use the diagrams. This has confirmed our theory that splitting up large diagrams into smaller logical chunks has positive effect on the usability of the same. Another difference between the two experiments was that in Experiment 2 the diagrams used colour as a secondary notation and they also were more interactive with short natural language descriptions when hovering over the request and a clickable link to the relative OAS documentation. Students' qualitative feedback showed us that they have particularly appreciated the newly added diagram features.

The contributions discussed above can be used by the REST research community as building blocks for future research work on the topic of modelling the behaviour of REST APIs, as well as by REST practitioners in their daily work if they realize and decide to embrace the discussed benefits of visualizing RESTful conversations with a dedicated DSL.

8.2 Limitations

Designing a DSL is a challenging task as there are many inherent trade-offs to be considered. In Sec. 7.1 we discussed the trade-offs between the principles to be followed when designing the constructs of a visual notation. The first version of RESTalk, what we now call the core RESTalk, where the scope is limited to only modelling conversations between a single client and a single server, could achieve graphical economy with only 10 constructs. However, as we started increasing the scope of the language to support modeling of multi-party conversations and more complex conversations, the expressiveness of the language increased, but this came with a cost. Having more language constructs means higher learning curve and thus higher cognitive burden for the model designer. Finding the sweet-spot is not easy, but as Kelly and Pohjonen state in [105], a domain specific modeling "isn't about achieving perfection, just something that works in practice. It will always be possible to imagine a case that the language can't handle. The important questions are how often such cases occur in practice, and how well the language deals with common cases". After all, we do not propagate using RESTalk as the sole technique for documenting REST APIs. It merely focuses on the API's dynamic behaviour thus leaving out static details. They can be added to RESTalk as textual annotations, notes disclosed on click when using a modeling tool, or depending on modeler's goal, RESTalk can be used as a complementary technique to already existing techniques, which are more inclined to static and structural viewpoints and simply provide link to them for more details.
Apart from the limitations related to the design of RESTalk, other limitations come from the lack of evaluation of certain aspects of the language which we will discuss in Sec. 8.3, as well as from the threats to validity to the existing evaluations as discussed in Sec. 6.1.5 and Sec. 7.3.4.

8.3 Future Work

The iterative approach that we followed in the development of RESTalk depicted in Fig. 4.1 makes RESTalk a living organism that continues evolving. As new requirements and use cases get identified, the language gets expanded, which should be reflected in the tooling support and evaluated accordingly. Thus, much work is left to be done around RESTalk, and in this section we provide a non-exhaustive list of ideas for future work.

8.3.1 Requirements and Language Layers

Although as mentioned in Sec. 8.2 the scope of RESTalk was increasing as we got further into our research, it still does not reflect all of the possible use-cases. A multi-party conversations where multiple clients are interacting with multiple servers in order to reach a common goal is not yet supported by the language.
As pointed out by one of the survey respondents, the language does not allow to visually connect resources with their sub-resources, which can be important in a conversation in cases where updating or deleting the resource results with the automatic update or deletion of the sub-resource. However, as discussed in Sec. 8.2 adding new constructs to the language has its costs, thus this pin-pointed unsupported use-case in the language can be solved by interactive visualizations in the editor, as discussed in Sec. 8.3.2.
Some formalism of RESTalk as a language has been provided by the meta-model and its OCL constraints, as well as thorough the EBNF syntax of the textual representation. However, further formalization of the semantics of the language could provide a more rigorous method of reasoning about the language and more structured comparison of the graphical and textual representation of the same.

8.3.2 Tooling

In Chapter 5, in addition to the existing tools, we also discussed some envisioned tooling and how the existing proof of concept tools can be improved. Regarding the integration of RESTalk with the OAS documentation, the time is not mature yet for obtaining a full RESTalk diagrams from OAS documentation regardless of the support of the *links* object in OAS v3. Namely, many of the service providers used automated tooling to transition from OAS v2 to OAS v3, without actually augmenting the documentation with the new *links* object, so we leave it for future work to see whether full RESTalk diagrams can actually be generated from OAS documentation.
All the theoretical work behind RESTalk model verification with respect to the metamodel and the OCL constraints as well as the tooling support for the same has been out of scope for this research thesis and left as future work.
When it comes to the RESTalk editors, in addition to getting full textual DSL support in the textual editor, an ideally envisioned future step is to merge the graphical editor and the textual editor in a single web based tool with two-way synchronization between the graphical and the textual model. To address some of the feedback obtained during the controlled experiment different interactive visualizations can be supported in the editor. Colors can be used not only for

distinguishing between different methods, as we did in the Imgur API diagrams, but also for marking all the available interactions with the same resource, or for connecting resources with their sub-resources. As visualizing all of this different aspects at the same time can be overwhelming, there should be an option in the tool to decide which aspect to visualize, or another option is to use darker coloring of all the interactions belonging to the same resource when hovering over a given request.
Instead of links to OAS documentation, which force the user to navigate to a new website, a tool can support a pop-up window on a request click which would show all the detailed information about the request (e.g., request header data) and the response (e.g., example resource representation).
On the other hand, to deal with large graphs, a tool can support dynamic graph creation one interaction at a time based on user's decision on which path to follow. This would allow for gradual discovery of the entire graph, thus avoiding the overwhelming effect of a big diagram. Another approach would be splitting the diagrams into smaller chunks depending on the possible goals that can be achieved with the same and providing appropriate means for linking diagrams between them. The use of artificial intelligence can be explored for identifying the right diagram granularity.

8.3.3 Evaluation Layer

As the textual DSL for RESTalk was added in the last stages of this PhD research, the evaluations of the same remains as future work. Evaluations are needed for the design and intuitiveness/understandability of the textual DSL, but also for the benefits that the approach of textual modelling aided by mining can provide. A controlled experiment can be conducted to evaluate whether the use of the textual editor provides for greater accuracy and greater time efficiency in the generation of the diagrams. The independent variable would be the creation of a RESTalk diagram with two treatments: use of a textual editor and use of a graphical editor with the same power of language expressiveness. Experiments can also be done to test the impact of the approach of textual modelling aided by mining approach on the cognitive effort of the modeller. As we mentioned in Sec. 5.2.2, the RESTalk textual editor shares the same idea with the BPMN Sketch Miner editor that we have developed, and the initial results of the evaluation of the BPMN Sketch Miner DSL and tool have been encouraging [92]. The controlled experiment conducted in [92] was not comparative and only included the use of the textual editor, but it showed that novice modelers could identify the needed traces to model a non-trivial business process with an average accu-

racy of over 75% in less then an hour and a half. We also conducted a survey using the questions of the standard System Usability Scale (SUS [23]) which included both the novice participants (18 students) and industry practitioners (12 respondents) which resulted with overall score of 69 ± 5 and industry score of 76 ± 7 (acceptability threshold are usually between 64 and 69, depending on the source). A similar study should also be conducted for the RESTalk editor.
The evaluation work which we have done targeted the visual RESTalk and its effectiveness and efficiency about facilitating the understanding and the use of a given API. Thus, it targeted API client developers and the use of RESTalk diagrams as API documentation. However, the drawback is that the controlled experiment has been done with students, who are not the primary target users of RESTalk as they lack the domain specific experience. In the future, efforts can be done to find industry practitioners and repeat the controlled experiment. More complex experiment design can be used where multiple independent variables are added, such as the experience of the participants in developing REST APIs, the size of the API etc. On the other hand, in the future, evaluation needs to be done which targets API server developers and the use of RESTalk diagrams in the design of the API to facilitate discussion and knowledge sharing.
Further evaluation and testing of the RESTalk Miner tool described in Sec. 5.3 is also needed, both in terms of the perceived usefulness of the tool in general, as well as in terms of the scalability of the tool for analysing larger API usage logs. The challenge for such evaluation is to find such logs in the first place. Also the logs should have a structure that allows to differentiate between requests which belong to different conversations.

Appendices

Appendix A

Exploratory Survey Questions

Modeling RESTful Conversations (English version)

Welcome to the RESTful conversations survey. RESTful conversations are compex interactions between client(s) and server(s). For more details on the conversation based approach for modeling RESTful APIs please refer to the following paper: http://design.inf.usi.ch/sites/default/files/biblio/wicsa2015.pdf.

**Please note that having in mind the exploratory goal of this survey, going back to the previous question is not an option in the same.*

Thank you for agreeing to take part in this exploratory research about the existing practices in visualization and modeling of HTTP interactions between clients and RESTful APIs. Regardless of whether you are a RESTful APIs designer or user, your answers will significantly contribute in directing our research. Our ultimate goal is to facilitate the high-level communication within and between RESTful APIs development teams. We believe that using efficient visualization tools and frequently used conversation patterns can improve teams' productivity, promote reuse and leverage knowledge sharing. Clear specification of the possible HTTP request-response sequences at the design stage can help avoid errors in the development.

Completing the whole survey will take about 30 to 40 minutes. You can skip some of the longer questions and complete the survey in about 20 minutes as well, which we completely understand. Every answer is helpful
for us.

We assure you that the provided answers will be kept confidential.

There are 90 questions in this survey

Current experience

What is your current job title?

Please write your answer here:

Do you have experience designing RESTful APIs? If yes, how many months/years of experience do you have?

Please write your answer here:

Do you have experience using RESTful APIs designed by others to integrate services into your projects? If yes, how many months/years of experience do you have?

Please write your answer here:

Usage of existing visual notations

Do you use /have you ever tried using some visual notations to discuss the lifecycle of used resources and allowed HTTP interactions with REST APIs? *

Please choose **only one** of the following:

- ○ Yes, I use/have used visual notations
- ○ No, I only use textual notations

Which visual notation(s) do you / have you used? *

Only answer this question if the following conditions are met:
Answer was 'Yes, I use/have used visual notations ' at question '4 [B1]' (Do you use /have you ever tried using some visual notations to discuss the lifecycle of used resources and allowed HTTP interactions with REST APIs?)

Please choose **all** that apply:

- ☐ UML activity diagrams
- ☐ UML sequence diagrams
- ☐ BPMN choreography
- ☐ In-house developed notations
- ☐ Other standard notations

Which other standard notations do you / have you used? *

Only answer this question if the following conditions are met:
Answer was at question '5 [B11]' (Which visual notation(s) do you / have you used?)

Please write your answer here:

Usage of BPMN Choreography

Why did you decide to use BPMN Choreography?

Please write your answer here:

Which constructs of BPMN Choreography do you appreciate the most and you find core for depicting RESTful conversations?

Please write your answer here:

Do you feel like certain flows or situations cannot be expressed with BPMN Choreography? If yes, please elaborate on examples.

Please write your answer here:

What was the impact on your team's productivity since you have started using BPMN Choreography?

Please write your answer here:

What was the effort of learning BPMN Choreography?

Please write your answer here:

Which modeling tool do you use / have you used to create BPMN Choreography diagrams?

Please write your answer here:

Are you currently still using BPMN Choreography? *

Please choose **only one** of the following:

- ○ Yes
- ○ No

Why did you stop using BPMN Choreography?

Only answer this question if the following conditions are met:
Answer was 'No' at question '13 [B181]' (Are you currently still using BPMN Choreography?)

Please write your answer here:

Usage of in-house developed notation

Why did you decide to use an in-house developed notation?

Please write your answer here:

Which constructs of your in-house developed notation do you appreciate the most and you find core for depicting RESTful conversations?

Please write your answer here:

Do you feel like certain flows or situations cannot be expressed with your in-house developed notation? If yes, please elaborate on examples.

Please write your answer here:

What was the impact on your team's productivity since you have started using your in-house developed notation?

Please write your answer here:

What was the effort of learning your in-house developed notation?

Please write your answer here:

Which modeling tool do you use / have you used to create your in-house developed diagrams?

Please write your answer here:

Please provide a short description of the in-house developed notation you use / have used.

Please write your answer here:

We would appreciate an example of the in-house developed notation you use, if you have any available. Please upload a file.

Please upload at most one file

Kindly attach the aforementioned documents along with the survey

When designing your in-house developed notation which of the following two approaches did you follow? *

Please choose **only one** of the following:

- ○ Extend/adapt a standard notation
- ○ Build a completely new notation

Which standard notation did you extend/adapt?

Only answer this question if the following conditions are met:
Answer was 'Extend/adapt a standard notation ' at question '23 [B172]' (When designing your in-house developed notation which of the following two approaches did you follow?)

Please write your answer here:

Why did you decide to extend/adapt the notation?

Only answer this question if the following conditions are met:
Answer was 'Extend/adapt a standard notation ' at question '23 [B172]' (When designing your in-house developed notation which of the following two approaches did you follow?)

Please write your answer here:

What type of extensions/adaptations did you do?

Only answer this question if the following conditions are met:
Answer was 'Extend/adapt a standard notation ' at question '23 [B172]' (When designing your in-house developed notation which of the following two approaches did you follow?)

Please write your answer here:

Are you currently still using your in-house developed notation? *

Please choose **only one** of the following:

- ○ Yes
- ○ No

Why did you stop using your in-house developed notation?

Only answer this question if the following conditions are met:
Answer was 'No' at question '27 [B182]' (Are you currently still using your in-house developed notation?)

Please write your answer here:

Usage of UML sequence diagrams

Why did you decide to use UML sequence diagrams?

Please write your answer here:

Which constructs of UML sequence diagrams do you appreciate the most and you find core for depicting RESTful conversations?

Please write your answer here:

Do you feel like certain flows or situations cannot be expressed with UML sequence diagrams? If yes, please elaborate on examples.

Please write your answer here:

What was the impact on your team's productivity since you have started using UML sequence diagrams?

Please write your answer here:

What was the effort of learning UML sequence diagrams?

Please write your answer here:

Which modeling tool do you use / have you used to create UML sequence diagrams?

Please write your answer here:

Are you currently still using UML sequence diagrams? *

Please choose **only one** of the following:

- ○ Yes
- ○ No

Why did you stop using UML sequence diagrams?

Only answer this question if the following conditions are met:
Answer was 'No' at question '35 [B183]' (Are you currently still using UML sequence diagrams?)

Please write your answer here:

Usage of UML activity diagrams

Why did you decide to use UML activity diagram?

Please write your answer here:

Which constructs of UML activity diagrams do you appreciate the most and you find core for depicting RESTful conversations?

Please write your answer here:

Do you feel like certain flows or situations cannot be expressed with UML activity diagrams? If yes, please elaborate on examples.

Please write your answer here:

What was the impact on your team's productivity since you have started using UML activity diagrams?

Please write your answer here:

What was the effort of learning UML activity diagrams?

Please write your answer here:

Which modeling tool do you use / have you used to create UML activity diagrams?

Please write your answer here:

Are you currently still using UML activity diagrams? *

Please choose **only one** of the following:

- ○ Yes
- ○ No

Why did you stop using UML activity diagrams?

Only answer this question if the following conditions are met:
Answer was 'No' at question '43 [B184]' (Are you currently still using UML activity diagrams?)

Please write your answer here:

Usage of other standard notations

Why did you decide to use this/these standard notations?

Please write your answer here:

Which constructs of this/these standard notation(s) do you appreciate the most and you find core for depicting RESTful conversations?

Please write your answer here:

Do you feel like certain flows or situations cannot be expressed with this/these notation(s)? If yes, please elaborate on examples.

Please write your answer here:

What was the impact on your team's productivity since you have started using this/these notation(s)?

Please write your answer here:

What was the effort of learning this/these notation(s)?

Please write your answer here:

Which modeling tool do you use / have you used to create diagrams with this/these standard notation(s)?

Please write your answer here:

Are you currently still using this/these standard notation(s)? *

Please choose **only one** of the following:

- ○ Yes
- ○ No

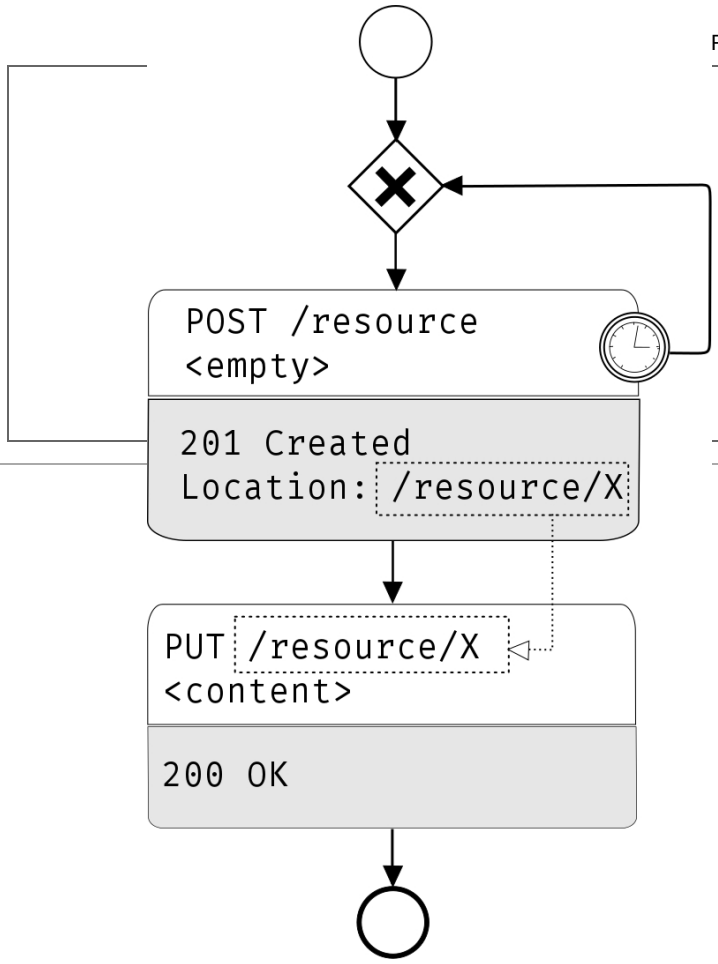
Why did you stop using this/these standard notation(s)?

Only answer this question if the following conditions are met:
Answer was 'No' at question '51 [B185]' (Are you currently still using this/these standard notation(s)?)

Please write your answer here:

RESTful conversation example

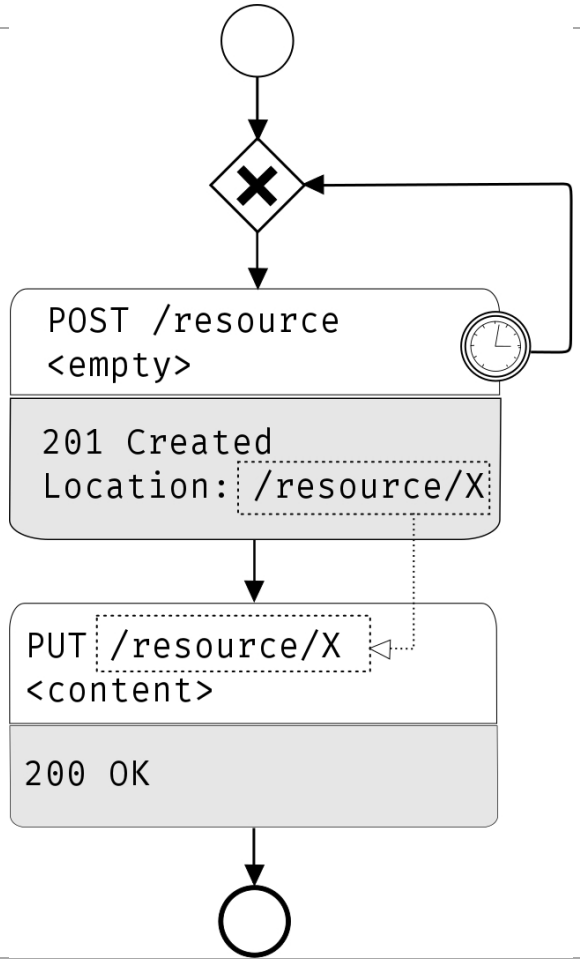
Could you briefly explain your understanding of the following diagram?



Please write your answer here:

Understanding RESTful conversations

Given the following RESTful conversation, please answer the following questions:



The goal of this RESTful conversation is:

Please choose **only one** of the following:

- ○ Editing an existing resource
- ○ Creating a new resource
- ○ Creating multiple new resources
- ○ None of the above

In this RESTful conversation the client can send the POST request multiple times.

Please choose **only one** of the following:

- ○ True
- ○ False
- ○ I don’t know

In this RESTful conversation, by sending multiple POST requests multiple resources are being created.

Please choose **only one** of the following:

- ○ True
- ○ False
- ○ I don’t know

In this RESTful conversation, the client knows the link to the created resource before the start of the conversation.

Please choose **only one** of the following:

- ○ True
- ○ False
- ○ I don’t know

BPMN knowledge

Do you have basic knowledge of BPMN? *

Please choose **only one** of the following:

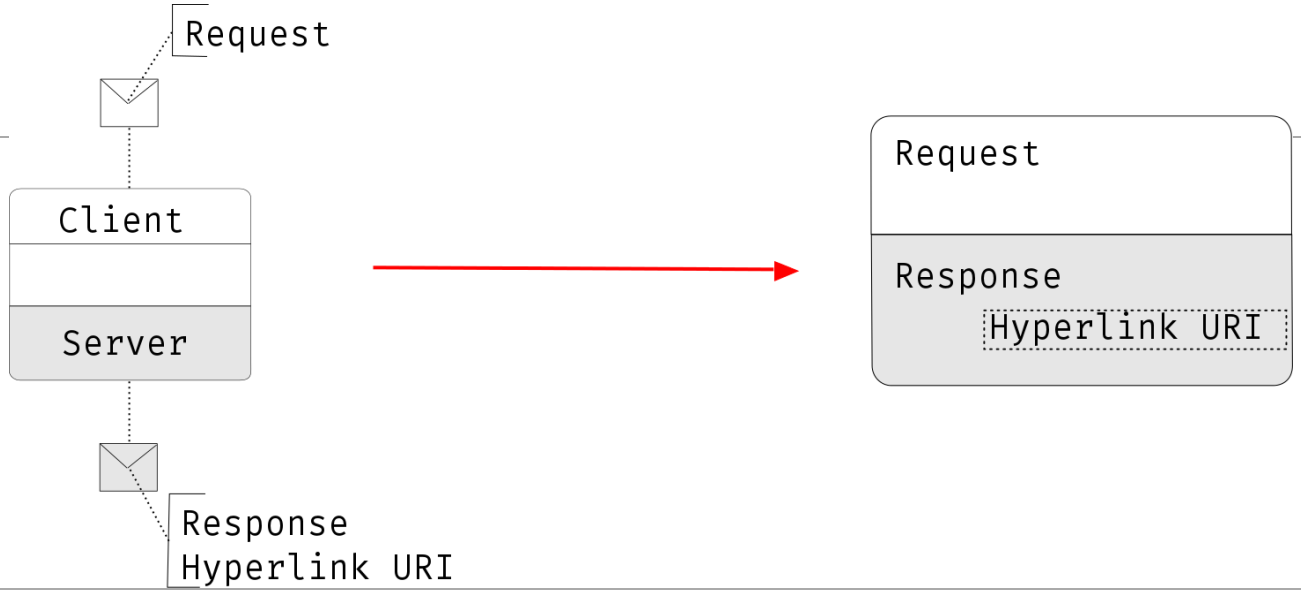
- ○ Yes
- ○ No

BPMN Choreography vs RESTful Choreography

To render the BPMN Choreography domain specific and more concise we propose the following extensions:

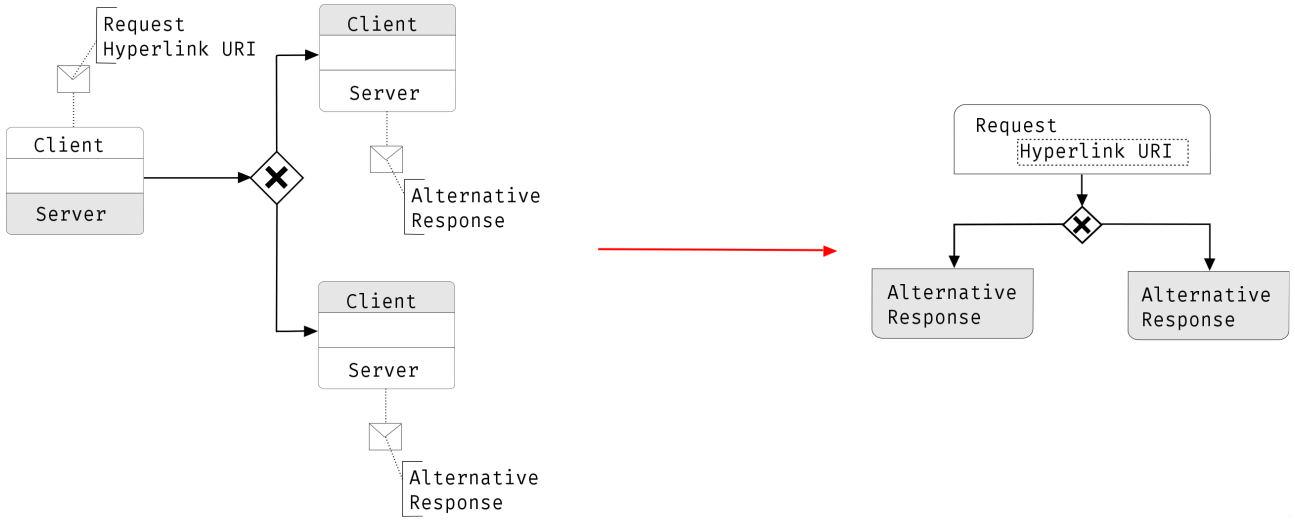
Extension 1:

Replacing the BPMN activity with a more compact two band request/response element with embedded message content. Since the participants in all RESTful interactions are always the client and the server we do not need to explicitly name them. This way the focus is on the content of the exchanged messages.



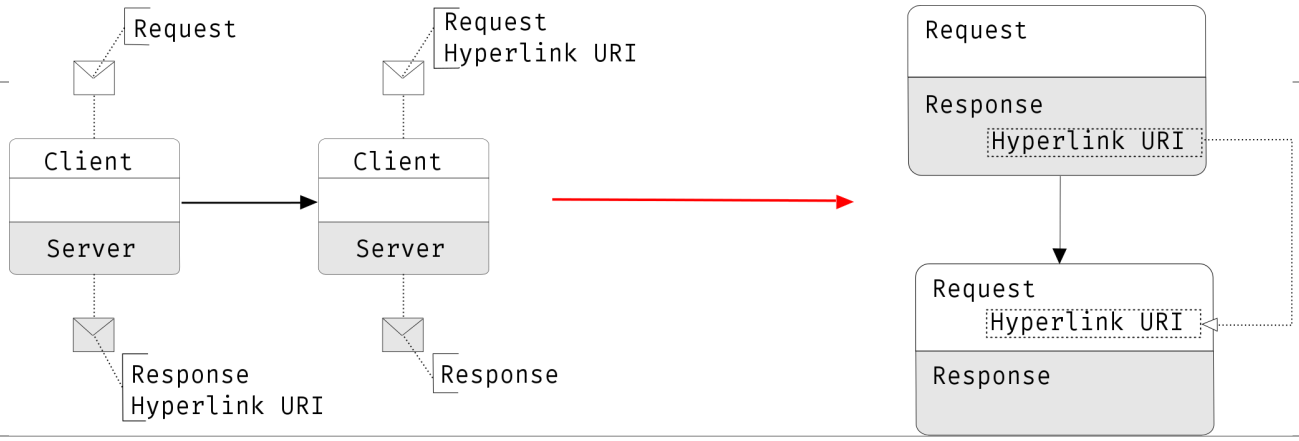
Extension 2:

Since in a RESTful interaction a request is always followed by a response, the request / response bends always go together, except when there is path divergence due to different possible responses from the server to a given client's request. In this case the request response bends are separated by an exclusive gateway to show the alternative responses that can be sent by the server.



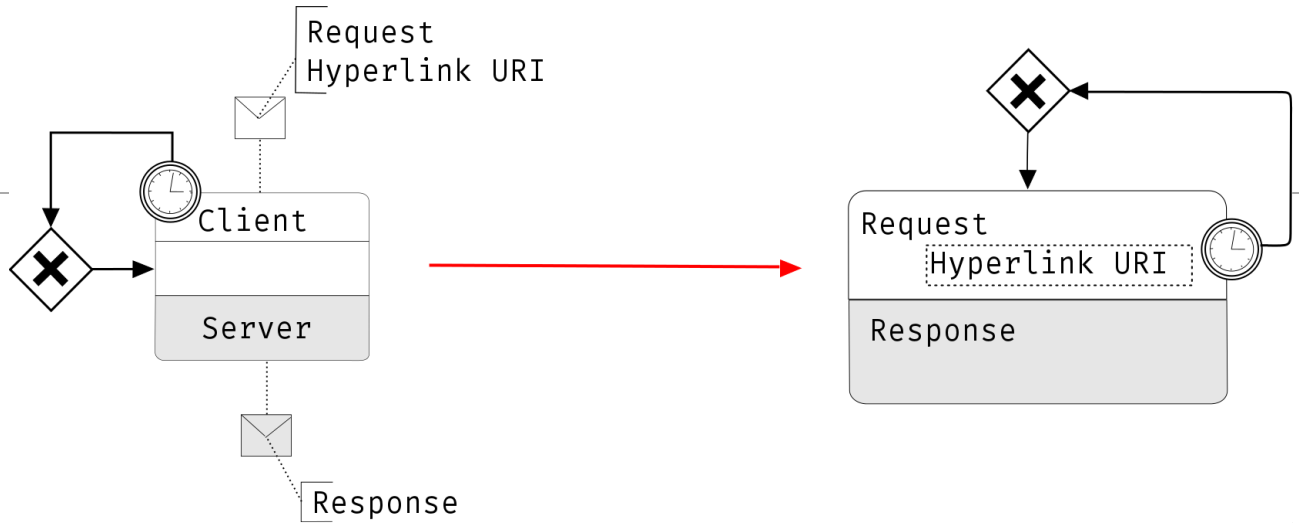
Extension 3:

Replacing the standard horizontal flow with a vertical flow and adding a hyperlink flow element. This element indicates how URIs are discovered from hyperlinks embedded in the preceding response. The purpose is to clarify how clients discover and navigate among related resources.



Extension 4:

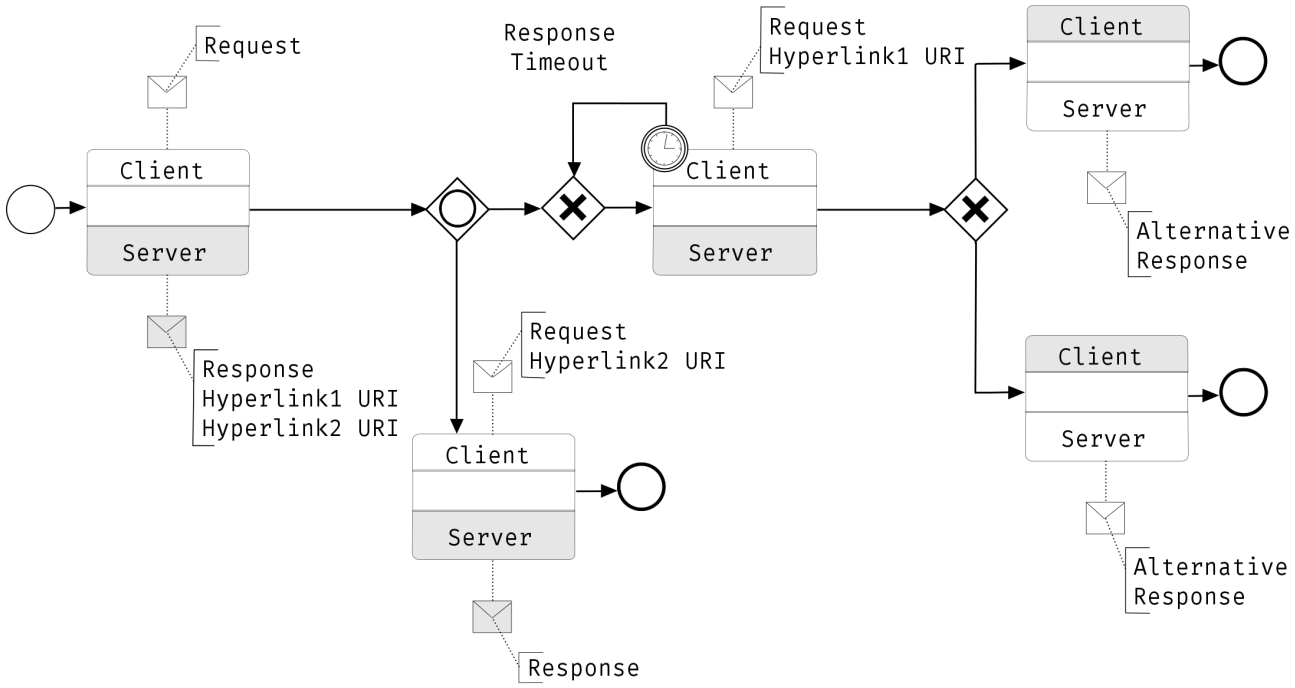
Using the timer event to model situations where the server takes too long to respond and thus the client decides to resend the request.



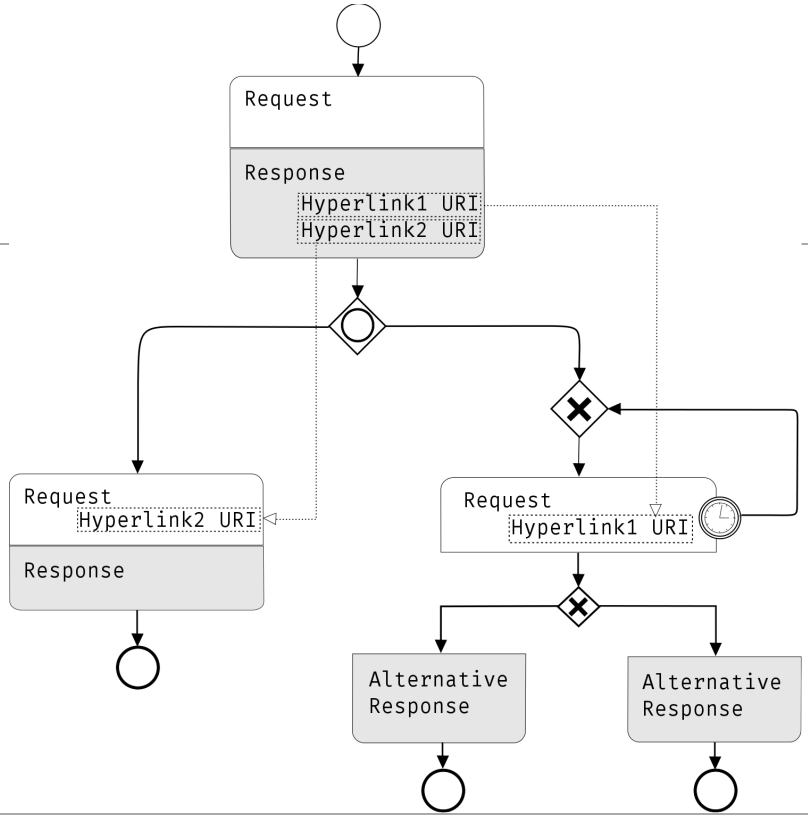
BPMN vs. RESTful Choreography evaluation

Following is a generic RESTful conversation modeled:

a) using BPMN Choreography



b) using our extended version of the BPMN Choreography



Which of the above mentioned notations do you find more concise?

Please write your answer here:

Which of the above mentioned notations do you find more expressive?

Please write your answer here:

Which of the above mentioned notations do you find easier to understand?

Please write your answer here:

Which of the two notations would you prefer using and why?

Please write your answer here:

RESTful Choreography

For visualizing and modeling RESTful conversations we propose an extended version of the BPMN Choreography comprised of the following elements:

1) Start event (to mark the beginning of the conversation)

2) An activity containing the content of the request and response messages

3) Hyperlink flow to show the flow of discovery and usage of hyperlinks

4) Control flow split gateways to show path divergence due to client's decisions:

XOR – exclusive gateway that allows only one of the outgoing paths to be taken

OR – inclusive gateway that allows one, some or all of the outgoing paths to be taken

AND – parallel gateway that requires all outgoing paths to be taken

5) Control flow merge gateways to show path convergence:

XOR – exclusive gateway that allows the conversation to continue once a request from one of the incoming flows has been received

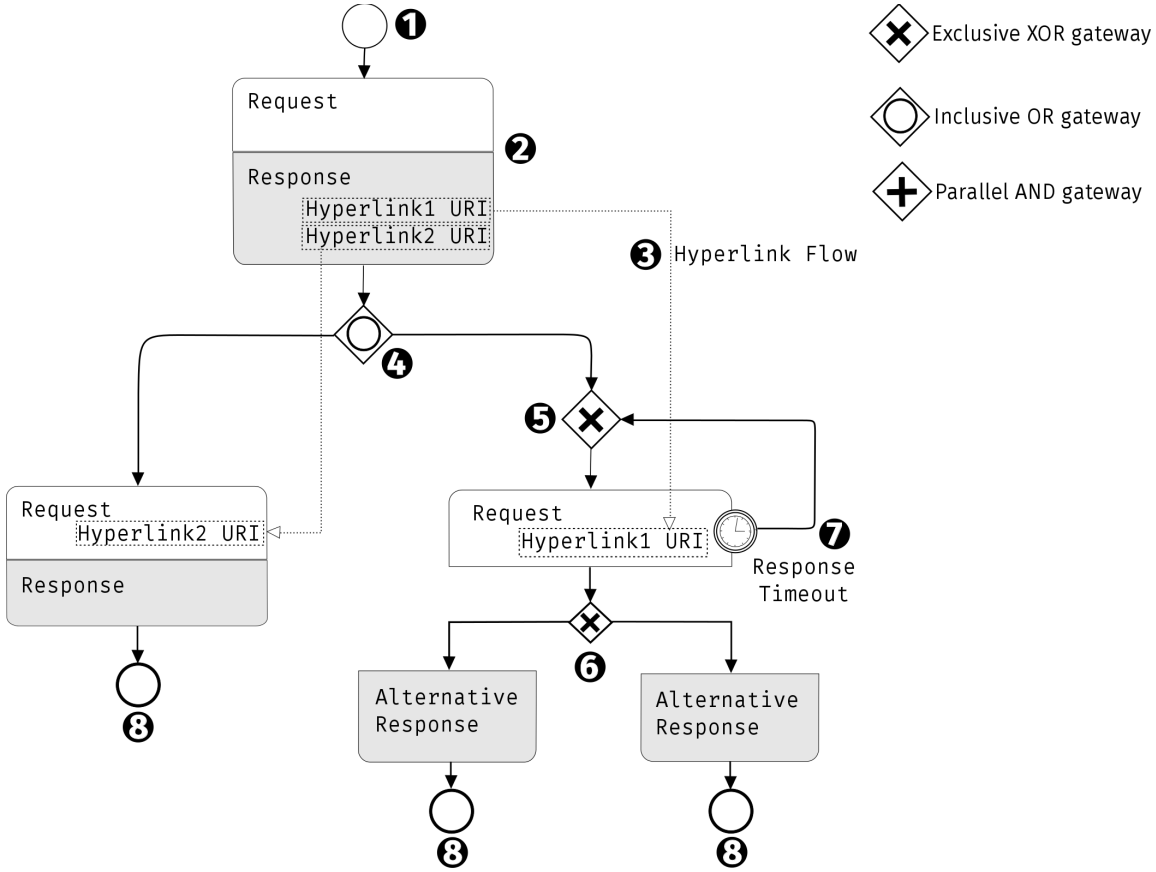
OR – inclusive gateway that allows the conversation to continue once the requests from one or more of the incoming flows have been received

AND – parallel gateway that requires all irequests from incoming flows to be received

6) Exclusive split due to different possible responses from the server

7) Response timeout to model situations where the server takes too long to respond and thus the client decides to resend the request.

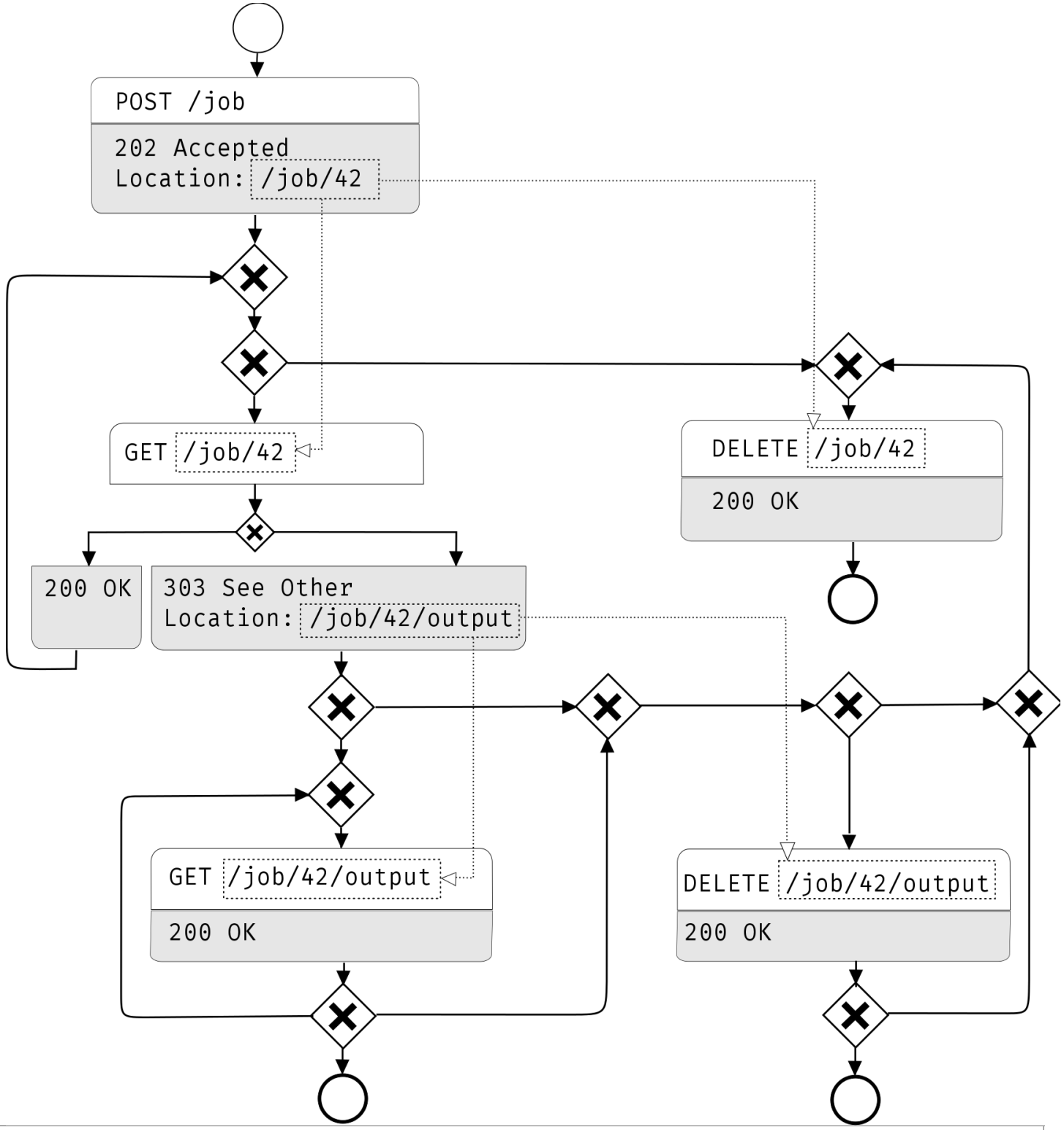
8) End event (to mark the end of the conversation)



Reading task

Following is a diagram of a RESTful conversation modeled with the proposed notation:

Reminder about the RESTful Choreography syntax can be found at the provided link.



How many resources are created during this conversation?

Please choose **only one** of the following:

- ○ None
- ○ One
- ○ Two
- ○ One or two
- ○ More than two

You can access the job output without having a link to the job itself.

Please choose **only one** of the following:

- ○ True
- ○ False
- ○ I don't know

What happens when you try to access the job resource while the job has not completed yet?

Please choose **only one** of the following:

- ○ You get a 200 OK status code and a placeholder link to where the output will be saved once the job has completed.
- ○ You get a 200 OK status code and can try to access the job again later.
- ○ You cannot send a GET job request before the job has completed.

For each one of the following statements determine whether it is true or false:

Please choose the appropriate response for each item:

	True	False	I don't know
The job output resource gets automatically deleted once the client has read it.	○	○	○
The client must delete the job output resource after it reads it.	○	○	○
The client can read the job output multiple times.	○	○	○
The client can decide to delete the job output resource only after it has read it.	○	○	○
The job resource can be deleted without deleting the job output resource.	○	○	○

When can you delete the job resource?

Please choose **only one** of the following:

- ○ Only after the job has completed.
- ○ Only before the job has completed.
- ○ Only after having read the output.
- ○ Only before having read the output.
- ○ Only after the job has completed and you have read the output.
- ○ Only after the job has completed, but before you have read the output.
- ○ At any time after the creation of the job resource.

Do you find the notation easy to understand?

Please write your answer here:

Do you find it more or less time consuming compared to the notation (textual or visual) you have used before?

Please write your answer here:

Do you find it more concise in terms of detecting all the possible paths interactions can take, compared to the notation (textual or visual) you have used before?

Please write your answer here:

Modeling task

Using the proposed notation model on a piece of paper a RESTful conversation that describes the lifecycle of a collection item, i.e., the possible CRUD (Create, Read, Update, Delete) operations on it. We would appreciate you uploading an image of your diagram if possible.

Please upload at most one file

Kindly attach the aforementioned documents along with the survey

Reminder about the RESTful Choreography syntax can be found at the provided link.

Please tell us what problems you have discovered while drawing the diagram or why you have decided not to do the task.

Please write your answer here:

RESTful Choreography evaluation

Do you find the proposed notation intuitive to use? If not, what seems complex, ambiguous or unclear?

Please write your answer here:

Do you think some elements are missing from the proposed notation?

Please write your answer here:

Please describe a RESTful conversation you have used which you believe cannot be described with the proposed notation.

Please write your answer here:

Would you use this modeling notation in your projects and why (not)?

Please write your answer here:

Do you find the used HTTP details (methods, URIs, status codes, links) sufficient for understanding what the RESTful conversations is aiming at? Would you add or delete some details?

Please write your answer here:

Would you prefer a tool for developing such diagrams? Do you think it is suitable for white board usage or manual sketches?

Please write your answer here:

RESTful conversation patterns

Please provide a short description of some reoccurring RESTful conversations that you have encountered during your professional experience.

Please write your answer here:

Do you think that identifying and naming RESTful conversation patterns can facilitate the high-level design of RESTful APIs? In which way?

Please write your answer here:

General comments

We would be happy to hear about any comments, suggestions or questions you might have.

Please write your answer here:

Thank you!

Thank you for you time. Your contribution to our research is highly appreciated. If you would like to be contacted in the future with updates or surveys on this research topics please provide us with your contact details:

Please write your answer(s) here:

Name:	
Surname:	
E-mail address:	

You can find a recent paper we have published regarding the proposed RESTful Choreography notation on the following link:

http://design.inf.usi.ch/publications/2015/ecsa

We remain available for any clarifications or discussions. Please do not hesitate to contact us at:

Ana Ivanchikj: ana.ivanchikj@usi.ch

Cesare Pautasso: c.pautasso@ieee.org

Silvia Schreier: silvia.schreier@innoq.com

Submit your survey.
Thank you for completing this survey.

Appendix B

Controlled Experiment Tasks and Survey

Part I – Background Information

Both Experiment 1 and 2 both groups

(1) Have you ever written a program to call a RESTful API?
Yes/No

(1a) If yes: For how long have you been using RESTful APIs?
0-1month, 2-6 months, 7-12 months, 1-2 years, 3-4 years, >4 years

(1b) If yes: Have you ever used Imgur's API?
Yes/No

(2) Have you ever used Imgur as a Web application?
Yes/No

Part II – General Understanding of Imgur and REST

Only Experiment 2 both groups

(1) How would you rate your understanding on yesterday's warm up session on Imgur's API on a scale from 1 (I did not understand it) to 10 (I understood it very well)?

(2) What is a gallery in Imgur and what is a galleryHash?
Solution: A gallery is a set of public images and albums, the ones that have been shared with the community. It is searchable by keyword or title or tag. The galleryHash is the unique identifier for each album or image in the gallery and is equivalent to the imageHash/albumHash. Remember that a galleryHash is not an id for a specific gallery, there is only one gallery in Imgur.

(3) What can you vote on in Imgur? *(Solutions in italics)*
Only on images; Only on albums; Any uploaded image; Any uploaded album;
Only images posted in the gallery; Only albums posted in the gallery;
Comments; Replies to comments

(4) Where can we use parameters in a RESTful interaction? *(Solutions in italics)*
URL; Request body; Request headers; Response body; Response headers

Part III – Tasks Authenticated User

For all of the tasks assume that:
- your user is authenticated with the Imgur API
- you have the user credentials (you can use your own Imgur profile for testing purposes).
At this phase you do not need to implement anything, but just to reason about the design of an application which uses Imgur as a service provider. If necessary, you could use Postman or the Advanced REST Client to test some of the endpoints and check your assumptions.

To answer the questions you can use:

1) the Imgur's user interface (https://imgur.com) to get familiar with the terminology and the basic actions available,
2) the following documentation on Imgur's API https://apidocs.imgur.com/#intro to discover the available resources, the methods they support and their purpose,
3) any other source that you find helpful

Task 1: Vote, Favorite and link to the first search result – Equal for both groups

Experiment 1:
Group A: only OAS documentation
Group B: only OAS documentation

Experiment 2:
Group A: only OAS documentation
Group B: OAS documentation and one RESTalk diagram available at:
http://design.inf.usi.ch/restalk-experiment/ImgurAuthExternal.html

Just for group B experiment 2:

Remember that:

- by clicking on the URL of the request in the diagram you get redirected to the API documentation
- by hovering over the coloured request box you get a short description of the semantics of the request
- by hovering over a parameter you get marked in red how it can be discovered (if it is in the request) or what are the possible requests where the parameter can be used (if it is in the response)
- you can use the "command + F" functionality of the browser to search the diagram
- by clicking on the coloured boxes in the order in which you want to state the requests in your answer they get copied in the text box (lower left angle) and you can copy-paste them to your answer (remember to add the parameters manually to your answer)

Imagine that you need to implement the following functionality in your application. Based on user's string input, your application should:

- **search** the Imgur **gallery** for related albums or images,

- **vote** with "**up**" the first item found in the gallery,

- put it in user's **favorites** and,

- **provide** the user a **URL link** to the gallery item that has been favored.

(1) Given the API's documentation and the description above what is a possible sequence of requests that your application should make to Imgur's API in order to achieve the above-stated goal?

For each request that you need to make, please provide the **Method** (GET, POST, PUT, DELETE..), the **URL** of the endpoint and the **parameters** to be sent in the body of the request, if any.

Please list the requests in the **right ordered sequence** to achieve your goal.

(If different/alternative sequences are plausible, please enumerate them separately – this was only requested in Experiment 1).

For instance, if your goal is to add a comment to an existing gallery post your answer could have the following format:

1. -> GET /gallery/search/q={{searchItem}}

   <- (response body parameter: galleryHash)

2. -> POST /gallery/{{galleryHash}}/comment

   <- (response body parameter: commentId)

SOLUTION:

1. To find the gallery post related to user's input:

-> GET /gallery/search/{{sort}}/{{window}}/{{page}}?q={{userInput}}
<- (response parameter: galleryHash which is equal to imageHash / albumHash)

2. To vote up the gallery post:
-> POST /gallery/{{galleryHash}}/vote/up

3. The gallery post that you find can be an image or an album. To add it to favorites, if it is an image you should use:

3a. POST /image/{{imageHash}}/favorite

If it is an album you should use:
3b. POST /album/{{albumHash}}/favorite

(2) Which resource(s) did you use to achieve your goal? *(Solution in italics)*
Account, Comment, *Album, Gallery, Image*

(3) On a scale from 1 (low) to 10 (high), how confident are you that your solution will work?

(4) If you have any doubts or uncertainties regarding your proposed solutions please state them here.

Task 2: Publish in album and gallery, favorite and follow tags

Experiment 1:
Group A: only OAS documentation
Group B: OAS documentation + RESTalk (all Imgur models without colors and interactivity)

Experiment 2:
Group A: OAS documentation and one RESTalk diagram available at:
http://design.inf.usi.ch/restalk-experiment/ImgurAuth.html
Group B: only OAS documentation

Just for group A experiment 2:

Remember that:

- by clicking on the URL of the request in the diagram you get redirected to the API documentation
- by hovering over the coloured request box you get a short description of the semantics of the request
- by hovering over a parameter you get marked in red how it can be discovered (if it is in the request) or what are the possible requests where the parameter can be used (if it is in the response)
- you can use the "command + F" functionality of the browser to search the diagram
- by clicking on the coloured boxes in the order in which you want to state the requests in your answer they get copied in the text box (lower left angle) and you can copy-paste them to your answer (remember to add the parameters manually to your answer)

Just for group B experiment 2:

***For this task, please DO NOT use the link to the RESTalk visual model that you received in the previous task as it DOES NOT contain the solution!!!** Just use the following documentation resources: https://apidocs.imgur.com/#intro.*

Imagine that you need to implement the following functionality in your application. Your application should allow the user to:

- **upload** a new image,

- place the new image in a **new album**,

- **share** the image **to the community** adding a **tag** to the same,

- start **following the tag** the user has added in the gallery post,

- add the image to favorites.

(1) As in Task 1, please describe a possible sequence of requests your application should send to the Imgur API in order to achieve the above-stated goal.

For each request that you need to make, please provide the **Method** (GET, POST, PUT, DELETE..), the **URL** of the endpoint and the **parameters** to be sent in the body of the request, or received in the response, if any.

Please list the calls in the **right ordered sequence** to achieve your goal.

For instance, if your goal is to add a comment to an existing gallery post your answer could have the following format:

1. -> GET /gallery/search/q={{searchItem}}

   <- (response body parameter: galleryHash)

2. -> POST /gallery/{{galleryHash}}/comment

   <- (response body parameter: commentId)

SOLUTION:

The assumption is that the user is authenticated and thus with each request in the header we send the accessToken.

You can choose to first upload the image or first create the album, both solutions are correct. In the following we explain the one where we first create the album.

1. To create the new album. If you had crated the image before, here you would use its id (imageHash) as a parameter in the body of the request in order to add it to the album.

-> POST /album (request body parameter: title)
<- (response body parameter: albumHash)

2. To upload the new image, if you had crated the album before, here you would use its id (albumHash) as a parameter in the body of the request in order to add it to the album.

-> POST /upload (request body parameters: albumHahs, image)
<- (response body parameter: imageHash)

3. To add the image to the public gallery, this is also the only way to add tags to the image as you cannot add tags when uploading the image.

-> POST /gallery/image/{{imageHash}} (request body parameters: title, tags)

4. To follow the tag provided by the user ({{tagName}})

-> POST /account/me/follow/tag/{{tagName}}

5. To add the image to the user's favorites

-> POST /image/{{imageHash}}/favorite

(2) What input would you need from the user to perform the whole sequence(s)?

Solution: The necessary input from the user is the title of the album, the image to be upload, as well as the title and the tag(s) for the gallery post.

(3) On a scale from 1 (low) to 10 (high), how confident are you that your solution will work?

(4) If you have any doubts or uncertainties regarding your proposed solutions please state them here.

Task 3: Replace image by title

Experiment 1:
Group A: only OAS documentation
Group B: OAS documentation + RESTalk (all Imgur models without colors and interactivity)

Experiment 2:
Group A: OAS documentation and one RESTalk diagram available at:
http://design.inf.usi.ch/restalk-experiment/ImgurAuth.html
Group B: only OAS documentation

Just for group A experiment 2:

Remember that:

- by clicking on the URL of the request in the diagram you get redirected to the API documentation
- by hovering over the coloured request box you get a short description of the semantics of the request
- by hovering over a parameter you get marked in red how it can be discovered (if it is in the request) or what are the possible requests where the parameter can be used (if it is in the response)
- you can use the "command + F" functionality of the browser to search the diagram
- by clicking on the coloured boxes in the order in which you want to state the requests in your answer they get copied in the text box (lower left angle) and you can copy-paste them to your answer (remember to add the parameters manually to your answer)

Just for group B experiment 2:

For this task, please DO NOT use the link to the RESTalk visual model that you received in the previous task as it DOES NOT contain the solution!!! *Just use the following documentation resources: https://apidocs.imgur.com/#intro.*

Imagine that you need to implement the following functionality in your application.

Your application should allow the user to publish a new version of an existing image he has uploaded previously. To do so, the user should provide the title of the existing image in order to **search the existing image and the album** it is placed in.

Then the **previous** version of the **image** should be **deleted**. The **new version** should be **uploaded in the same album as the old image**.

Note: The existing image can but does not have to be present in the gallery.

The user should get a report on the number of views and votes that the old version of the image had (this was only a requirement in Experiment 1).

(1) As in previous Tasks, please describe a possible sequence of requests your application should make to Imgur's API in order to achieve the above-stated goal.

For each request that you need to make, please provide the **Method** (GET, POST, PUT, DELETE..), the **URL** of the endpoint and the **parameters** to be sent in the body of the request, or received in the response, if any.

Please list the calls in the **right ordered sequence** to achieve your goal.

For instance, if your goal is to add a comment to an existing gallery post your answer could have the following format:

1. -> GET /gallery/search/q={{searchItem}}

   <- (response body parameter: galleryHash)

2. -> POST /gallery/{{galleryHash}}/comment

   <- (response body parameter: commentId)

   SOLUTION:

   The assumption is that the user is authenticated and thus with each request in the header we send the accessToken.

   1. To get the ids of all albums that the user has created so that you can iterate over them to see where the image is. The response to this request will not reveal to you the title and the id of the images that are in each album.

   -> GET /account/{{username}}/albums/
   <- (response body parameter: albumHash(s))

   2. To get the id of the image to be replaced. This request should be in a loop based on the list of ids from the previous request. The goal is to check whether the image we are searching for is in this particular album based on the title provided by the user.

   -> GET /album/{{albumHash}}
   <- (response body parameter: imageHash)

   3. To delete the existing image based on the imageHash that you have found with the previous request.

   -> DELETE /image/{{imageHash}}

   4. To upload the new image. The body of the request should contain as a parameter the albumHash of the album where the existing image used to be, which you have discovered when executing the loop in step 2.

   -> POST /upload (request body parameter: albumHash, title, image)

   Note that if you use GET /gallery/search/{{sort}}/{{window}}/{{page}}?q=title:{{title}} to find the image with the title provided by the user, you will only be searching the gallery, and the specification states that the image does not have to be in the gallery. Furthermore, the gallery will contain images from other users as well, so you would also need to check the accountId or the username in order to make sure that it is the image that the user is searching for. On another note, the response would provide the id of the album where the image is located. So a possible approach would be to use this request, checking against the username, and if you do not find the image

here to use the solution above. This would save you from retrieving and iterating over all of the albums.

(2) What input would you need from the user to perform the whole sequence(s)?

Solution: The necessary input from the user is the image to be upload, as well as the title of the existing image which is supposed to be deleted. Note that as per the specification the user does not provide the id of the existing image, but its title, and he also does not provide the id of the album where the existing image is placed. You should discover these two ids yourself.

(3) Is there a difference between the methods DELETE /image/{{imageHash}} and DELETE gallery/{{galleryHash}}? If yes, what is the difference?

Solution: DELETE /image/{{imageHash}} will delete the image both from the gallery and the user account, while DELETE /gallery/{{galleryHash}} will delete the image only from the gallery. Note that the second DELETE will not delete the entire gallery and all the images/albums in it.

(4) On a scale from 1 (low) to 10 (high), how confident are you that your solution will work?

(5) If you have any doubts or uncertainties regarding your proposed solutions please state them here.

Part IV – Tasks Unauthenticated User

Experiment 1:
Group A: only OAS documentation
Group B: OAS documentation + RESTalk (all Imgur models without colors and interactivity)

Experiment 2:
Group A: only OAS documentation
Group B: OAS documentation and one RESTalk diagram available at:
http://design.inf.usi.ch/restalk-experiment/ImgurUnauth.html

Just for group A experiment 2:

***For this task, please DO NOT use the link to the RESTalk visual model that you received in the previous task as it DOES NOT contain the solution!!!** Just use the following documentation resources: https://apidocs.imgur.com/#intro.*

Just for group B experiment 2:

Remember that:

- by clicking on the URL of the request in the diagram you get redirected to the API documentation

- by hovering over the coloured request box you get a short description of the semantics of the request
- by hovering over a parameter you get marked in red how it can be discovered (if it is in the request) or what are the possible requests where the parameter can be used (if it is in the response)
- you can use the "command + F" functionality of the browser to search the diagram
- by clicking on the coloured boxes in the order in which you want to state the requests in your answer they get copied in the text box (lower left angle) and you can copy-paste them to your answer (remember to add the parameters manually to your answer)

(1) Can an unauthenticated (unregistered) user (Yes / No):
(a) create an album? - *Yes*
(b) get a list of all the images that he has added to the album? - *Yes*
(c) upload an image? - *Yes*
(d) add a comment or vote to a post? - *No*

(2) How can an unauthenticated user add images to an album? Only in experiment 1: If there are multiple options, please mention them all.

Solution: Following are some of the ways in which an unauthenticated user can add images to an album:
- POST /album,
- POST /album/{{albumDeleteHash}},
- POST /album/{{ albumDeleteHash}/add, - POST /image,
- PUT /album/{{albumHash}}.
Note that he will always need to use the deleteHash of the image in order to add it.

(3) What is the difference in managing images on Imgur between an authenticated and unauthenticated user?

Solution: In addition to some functionalities being available only to authenticated users (such as sharing with the community, adding to favorites, commenting, voting etc.), an important distinction is that unauthenticated users need to use the imageDeleteHash for deleting or editing images they have uploaded, while registered users can use the imageHash to do so.

(4) As an unauthenticated user, how can you discover the albumHash you need in order to GET an album? Please name all the endpoints (method + URI) which can give you back information about the albumHash.

Solution: Following are some of the ways in which an unauthenticated user can discover the albumHash to be used to GET an album (requests whose response body can contain an albumHash):
- GET /gallery/{{section}},
- GET /gallery/t/{{tagName}},
- GET /gallery/search/q={searchedItem},
- POST /album,
- GET /account/{{username}}/albums/ids/{{page}}

(5) On a scale from 1 (low) to 10 (high), how confident are you that your solution will work?

(6) If you have any doubts or uncertainties regarding your proposed solutions please state them here.

Part V – Survey

Experiment 1:
Group A: only questions 1 to 5
Group B: all questions

Experiment 2:
All questions for both Group A and Group B

(1) Given your goal in task 1 (searching for a related item), on a scale from 1 (very easy) to 10 (very hard), how hard was it to understand the sequence of calls you need to make based on the available documentation?

(2) Given your goal in task 2 (posting a new image), on a scale from 1 (very easy) to 10 (very hard), how hard was it to understand the sequence of calls you need to make based on the available documentation?

(3) Given your goal in task 3 (posting a new version of existing image), on a scale from 1 (very easy) to 10 (very hard), how hard was it to understand the sequence of calls you need to make based on the available documentation?

(4) What would you change in the documentation to make it more understandable and easier to use? (only in Experiment 1)

(5) Did you use PostMan/Advanced REST Client to test some of the API endpoints?
Yes/No

(6) (When available) Did you use the visual models to complete any of the tasks?
Yes/No

(6a1) If yes: On a scale from 1 (not helpful at all) to 10 (very helpful), how helpful did you find the visual models in answering the questions in this survey?

(6a2) If yes (only in Experiment 2): What did you like /found helpful regarding the models/visual language?

(6a3) If yes: What would you change in the visual models/visual language to make them more understandable and easier to use?

(6B) If no (only in Experiment 2): Why did you not use the visual models to complete the tasks?

(7) (Only in Experiment 1) For which task did you find the visual models the most helpful and why?

(8) (Only in Experiment 1) Which visual construct of the language used in the visual models you were unsure how to interpret?

Bibliography

[1] Disco. https://fluxicon.com/disco/. Last accessed: 2018-08-20.

[2] The history of REST APIs. https://blog.readme.io/the-history-of-rest-apis/, 2016.

[3] Adaptive et al. Meta Object Facility. OMG, October 2016. https://www.omg.org/spec/MOF/2.5.1/.

[4] Lorenzo Addazi, Federico Ciccozzi, Philip Langer, and Ernesto Posse. Towards seamless hybrid graphical–textual modelling for UML and profiles. In *European Conference on Modelling Foundations and Applications*, pages 20–33, 2017.

[5] Saeed Aghaee. *End-user development of mashups using live natural language programming*. PhD thesis, Università della Svizzera italiana, 2014.

[6] Rosa Alarcon and Erik Wilde. Linking Data from RESTful Services. In *Third Workshop on Linked Data on the Web*, Raleigh, North Carolina, 2010.

[7] Subbu Allamaraju. *RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity*. " O'Reilly Media, Inc.", 2010.

[8] Thomas Allweyer. *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. BoD–Books on Demand, 2010.

[9] Youseef Alotaibi and Fei Liu. Survey of business process management: challenges and solutions. *Enterprise Information Systems*, 11(8):1119–1153, 2017.

[10] Areeb Alowisheq, David E Millard, and Thanassis Tiropanis. Resource oriented modelling: Describing RESTful web services using collaboration diagrams. In *e-Business (ICE-B), 2011 Proceedings of the International Conference on*, pages 1–6. IEEE, 2011.

[11] Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 44–51. IEEE, 2016.

[12] Mike Amundsen. *Building Hypermedia APIs with HTML5 and Node*. O'Reilly, 2011.

[13] Thomas Baar. Correctly defined concrete syntax. *Software & Systems Modeling*, 7(4):383–398, 2008.

[14] Tavmjong Bah. *Inkscape: guide to a vector drawing program (Digital Short Cut)*. Pearson Education, 2010.

[15] Wasana Bandara, Guy G Gable, and Michael Rosemann. Factors and measures of business process modelling: model building through a multiple case study. *European Journal of Information Systems*, 14(4):347–360, 2005.

[16] Pranit Bari and PM Chawan. Web usage mining. *Journal of Engineering, Computers & Applied Sciences (JEC&AS)*, 2(6):34–38, 2013.

[17] Alistair Barros, Marlon Dumas, and Arthur H.M. ter Hofstede. Service Interaction Patterns. In WilM.P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649 of *LNCS*, pages 302–318. Springer, 2005.

[18] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Web service conversation modeling: A cornerstone for e-business automation. *Internet Computing, IEEE*, 8(1):46–54, 2004.

[19] Matthias Biehl. *API Architecture-The Big Picture for Building APIs*. CreateSpace, 2015.

[20] Matthias Biehl. *RESTful API Design*, volume 3. API-University Press, 2016.

[21] Stephen M Blackburn, Amer Diwan, Matthias Hauswirth, Peter F Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, et al. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(4):1–20, 2016.

[22] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice: Second Edition*. Morgan & Claypool Publishers, 2nd edition, 2017.

[23] John Brooke. SUS-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[24] Ricardo Buettner. Analyzing mental workload states on the basis of the pupillary hippus. *NeuroIS*, 14:52, 2014.

[25] Frederik Bülthoff and Maria Maleshkova. RESTful or RESTless – current state of today's top web APIs. In Valentina Presutti, Eva Blomqvist, Raphael Troncy, Harald Sack, Ioannis Papadakis, and Anna Tordai, editors, *The Semantic Web: ESWC 2014 Satellite Events*, pages 64–74, Cham, 2014. Springer International Publishing.

[26] Jordi Cabot and Martin Gogolla. Object constraint language (OCL): a definitive guide. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 58–90. Springer, 2012.

[27] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992.

[28] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[29] Mario Cortes-Cornax, Sophie Dupuy-Chessa, Dominique Rieu, and Marlon Dumas. Evaluating choreographies in BPMN 2.0 using an extended quality framework. In *Business Process Model and Notation*, volume 95 of *LNBIP*, pages 103–117. Springer, 2011.

[30] Ira W Cotton and Frank S Greatorex Jr. Data structures and techniques for remote computer graphics. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 533–544. ACM, 1968.

[31] Ward Cunningham and Kent Beck. Using pattern languages for object-oriented programs. In *Proceedings of OOPSLA*, volume 87, 1987.

[32] Rafael Corveira da Cruz Gonçalves and Isabel Azevedo. RESTful web services development with a model-driven engineering approach. In *Code Generation, Analysis Tools, and Testing for Quality*, pages 191–228. IGI Global, 2019.

[33] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems and Structures*, 43:139 – 155, 2015.

[34] Robert Daigneau. *Service Design Patterns:Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, 2011.

[35] Andrea D'Ambrogio. A model-driven WSDL extension for describing the QoS of web services. In *International Conference on Web Services, 2006. ICWS'06*, pages 789–796. IEEE, 2006.

[36] Nadja Damij. Business process modelling using diagrammatic and tabular techniques. *Business process management journal*, 13(1):70–90, 2007.

[37] Florian Daniel and Maristella Matera. *Mashups: Concepts, Models and Architectures*. Springer, 2014.

[38] Christopher J Date and Edgar F Codd. The relational and network approaches: Comparison of the application programming interfaces. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control: Data models: Data-structure-set versus relational*, pages 83–113, 1975.

[39] Shahir Daya, Nguyen Van Duy, Kameswara Eati, Carlos M Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, Valerie Lampkin, Marcelo Martins, et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016.

[40] Peter de Lange, Petru Nicolaescu, Ralf Klamma, and Matthias Jarke. Engineering web applications using real-time collaborative modeling. In *CYTED-RITOS International Workshop on Groupware*, pages 213–228. Springer, 2017.

[41] Gero Decker and Alistair Barros. Interaction Modeling Using BPMN. In *Business Process Management Workshops*, volume 4928 of *LNCS*, pages 208–219. Springer, 2008.

[42] Krista E DeLeeuw and Richard E Mayer. A comparison of three measures of cognitive load: Evidence for separable measures of intrinsic, extraneous, and germane load. *Journal of educational psychology*, 100(1):223, 2008.

[43] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[44] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Essential Process Modeling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

[45] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

[46] Hamza Ed-Douibi, Gwendal Daniel, and Jordi Cabot. OpenAPI Bot: A chatbot to help you understand REST APIs. In *International Conference on Web Engineering*, pages 538–542. Springer, 2020.

[47] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Francis Bordeleau, and Jordi Cabot. Wapiml: towards a modeling infrastructure for web apis. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 748–752. IEEE, 2019.

[48] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Example-driven web api specification discovery. In *European Conference on Modelling Foundations and Applications*, pages 267–284. Springer, 2017.

[49] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.

[50] Dirk Fahland and Matthias Weidlich. Scenario-based process modeling with greta. In Marcello La Rosa, editor, *Proceedings of the Business Process Management 2010 Demonstration Track*, volume 615 of *CEUR Workshop Proceedings*, pages 52–57. CEUR-WS.org, 2010.

[51] Roy Fielding et al. Hypertext transfer protocol – http/1.1 method definitions. `https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`, 1999.

[52] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol–HTTP/1.1. Request for Comments: 7230, 2014.

[53] Roy T Fielding, Richard N Taylor, Justin R Erenkrantz, Michael M Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. Reflections on the REST architectural style and principled design of the modern web architecture (impact paper award). In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 4–14. ACM, 2017.

[54] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[55] Marcus Fontoura and CJP De Lucena. Extending UML to improve the representation of design patterns. *Journal of Object Oriented Programming*, 13(11):12–19, 2001.

[56] Martin Fowler. Language workbenches: The killer-app for domain specific languages. `https://martinfowler.com/articles/languageWorkbench.html`, 2005.

[57] Ulrich Frank. Domain-specific modeling languages: requirements analysis and design guidelines. In *Domain Engineering*, pages 133–157. Springer, 2013.

[58] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta, and Henry Muccini. Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, 44(12):1146–1175, 2017.

[59] Andre Freitas, Edward Curry, Joao Gabriel Oliveira, and Sean O'Riain. Querying heterogeneous datasets on the linked data web: challenges, approaches, and trends. *IEEE Internet Computing*, 16(1):24–33, 2012.

[60] Nicole Freund. Development of a Text-Based Representation of BPMN Models. Master's thesis, Leibniz Universität Hannover, Hannover, Germany, 2018.

[61] Carsten Friedrich and Falk Schreiber. Flexible layering in hierarchical drawings with nodes of arbitrary size. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*, pages 369–376, P.O. Box 319 Darlinghurst, NSW 2010Australia, 2004. Australian Computer Society, Inc.

[62] Ana Funes, Aristides Dasso, Carlos Salgado, and Mario Peralta. UML tool evaluation requirements. In *Argentine Symposium on Information Systems ASIS*, pages 29–30, 2005.

[63] Antonio Gamez-Diaz, Pablo Fernandez, Cesare Pautasso, Ana Ivanchikj, and Antonio Ruiz-Cortes. ELeCTRA: Induced usage limitations calculation in RESTful APIs. In *International Conference on Service-Oriented Computing*, pages 435–438. Springer, 2018.

[64] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. Automating SLA-Driven API Development with SLA4OAI. In *International Conference on Service-Oriented Computing*, pages 20–35. Springer, 2019.

[65] Emden R Gansner and Stephen C North. An open graph visualization system and its applications to software engineering. *Software: practice and experience*, 30(11):1203–1233, 2000.

[66] Matthias Geiger, Simon Harrer, Jörg Lenhard, and Guido Wirtz. BPMN 2.0: The state of support and implementation. *Future Generation Computer Systems*, 80:250–262, 2018.

[67] Andrew Gemino and Yair Wand. A framework for empirical evaluation of conceptual modeling techniques. *Requirements Engineering*, 9(4):248–260, 2004.

[68] Nicolas Genon, Patrick Heymans, and Daniel Amyot. Analysing the cognitive effectiveness of the BPMN 2.0 visual notation. In *International conference on software language engineering*, pages 377–396. Springer, 2010.

[69] Alan Glickenhouse. What are the recommended roles for an API initiative? `https://developer.ibm.com/apiconnect/2017/05/19/recommended-roles-api-initiative/`, 2018.

[70] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of concrete textual syntax mapping approaches. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, pages 169–184, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[71] Harish Goteti. API Driven Development, Bridging the gap between Providers and Consumers. Technical report, CA Technologies, 2015.

[72] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. Towards recovering the software architecture of microservice-based systems. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 46–53. IEEE, 2017.

[73] Thomas R. G. Green and Marian Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[74] Joe Gregorio, R Fielding, Marc Hadley, Mark Nottingham, and David Orchard. URI Template. Request for Comments: 6570, 2012.

[75] Hans Grönninger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. In *Proc. of the 4th International Workshop on Software Language Engineering*, 2007.

[76] Hans Grönninger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased modeling. In *Proc. of the 4th International Workshop on Software Language Engineering*. Springer-Verlag, 2007.

[77] Volker Gruhn and Ralf Laue. Reducing the cognitive complexity of business process models. In *2009 8th IEEE International Conference on Cognitive Informatics*, pages 339–345. IEEE, 2009.

[78] Yann-Gaël Guéhéneuc and Foutse Khomh. Empirical software engineering. In *Handbook of Software Engineering*, pages 285–320. Springer, 2019.

[79] C.A. Gurr. Effective diagrammatic communication: Syntactic, semantic and pragmatic issues. *Journal of Visual Languages and Computing*, 10:317–342, 1999.

[80] Florian Haupt, Dimka Karastoyanova, Frank Leymann, and Benjamin Schroth. A model-driven approach for REST compliant services. In *International Conference on Web Services (ICWS 2014)*, pages 129–136. IEEE, 2014.

[81] Florian Haupt, Frank Leymann, and Cesare Pautasso. A conversation based approach for modeling REST APIs. In *Proc. of the 12th Working IEEE / IFIP Conference on Software Architecture (WICSA 2015)*, Montreal, Canada, May 2015.

[82] Florian Haupt, Frank Leymann, and Karolina Vukojevic-Haupt. Api governance support through the structural analysis of REST APIs. *Computer Science - Research and Development*, 33(3):291–303, Aug 2018.

[83] Pat Helland. Data on the Outside Versus Data on the Inside. *Conference on Innovative Data Systems Research (CIDR)*, pages 144–153, 2005.

[84] Ian Hickson, R Berjon, S Faulkner, T Leithead, ED Navara, E O'Connor, and Silvia Pfeiffer. HTML5. A vocabulary and associated APIs for HTML and XHTML. W3C Recommendation, 2014.

[85] Gregor Hohpe. Let's have a conversation. *Internet Computing, IEEE*, 11(3):78–81, 2007.

[86] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley, 2004.

[87] Ana Ivanchikj. RESTful conversation with RESTalk -the use case of Doodle-. In *Proceedings of the International Conference on Web Engineering (ICWE'16)*, pages 583–587. Springer, 2016.

[88] Ana Ivanchikj, Ilija Gjorgjiev, and Cesare Pautasso. RESTalk miner: Mining restful conversations, pattern discovery and matching. In *International Conference on Service-Oriented Computing*, pages 470–475. Springer, 2018.

[89] Ana Ivanchikj and Cesare Pautasso. Modeling REST API behaviour with text, graphics or both? Boston, USA, November 2018.

[90] Ana Ivanchikj and Cesare Pautasso. Sketching process models by mining participant stories. In *International Conference on Business Process Management*, pages 3–19. Springer, 2019.

[91] Ana Ivanchikj and Cesare Pautasso. Modeling microservice conversations with RESTalk. In *Microservices, Science and Engineering*, pages 129–146. Springer, 2020.

[92] Ana Ivanchikj, Souhaila Serbout, and Cesare Pautasso. From text to visual BPMN process models: Design and evaluation. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20)*, pages 229–239. ACM, 2020.

[93] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, May 2018.

[94] Sven Jannaber, Dennis M Riehle, Patrick Delfmann, Oliver Thomas, and Jörg Becker. Designing a framework for the development of domain-specific process modelling languages. In *International Conference on Design Science Research in Information System and Technology*, pages 39–54. Springer, 2017.

[95] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[96] Claus T Jensen. *APIs for dummies*. John Wiley & Sons, Inc, 2015. IBM Limited Edition.

[97] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer Science & Business Media, 2013.

[98] Tom Johnson. OpenAPI specification and Swagger. `https://idratherbewriting.com/learnapidoc/restapispecifications.html`, 2018.

[99] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard, and Michel RV Chaudron. Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication. *Empirical Software Engineering*, pages 1–45, 2020.

[100] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does distance still matter? revisiting collaborative distributed software design. *IEEE Software*, 35(6):40–47, 2018.

[101] Diane Jordan and John Evdemon. Business Process Model And Notation Version 2.0. OMG, January 2011. `http://www.omg.org/spec/BPMN/2.0/`.

[102] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14(4):1505–1526, 2015.

[103] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM' 09)*, New York, NY, USA, 2009. Association for Computing Machinery.

[104] Piotr Karwatka, Mariusz Gil, Mike Grabowski, Aleksander Graf, Pawel Jedrzejewski, Michal Kurzeja, Antoni Orfin, and Bartosz Picho. Microsevice Architecture for eCommerce. `https://divante.co/books/PDFy/microservices-architecture-for-ecommerce.pdf`, 2017.

[105] Steven Kelly and Risto Pohjonen. Worst practices for domain-specific modeling. *IEEE software*, 26(4):22–29, 2009.

[106] Redona Kembora. APISymphony: a tool to measure and visualize static metrics of RESTful APIs. Master's thesis, Software Institute, Univresita della Svizzera italiana, Lugano, Switzerland, 2019.

[107] Anneke Kleppe, Jos Warmer, and Wim Bast. MDA explained. the practice and promise of the model driven architecture. *Boston Pearson Education, Inc*, pages 1–31, 2003.

[108] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption–a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 14:1–1, 2019.

[109] Ryan KL Ko. A computer scientist's introductory guide to business process management (BPM). *Crossroads*, 15(4):4, 2009.

[110] Jana Koehler, Rainer Hauser, Jochen Küster, Ksenia Ryndina, Jussi Vanhatalo, and Michael Wahler. The role of visual modeling and model transformations in business-driven development. *Electronic Notes in Theoretical Computer Science*, 211:5–15, 2008.

[111] Dimitrios S Kolovos, Richard F Paige, Tim Kelly, and Fiona AC Polack. Requirements for domain-specific languages. In *Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD)*, volume 2006, 2006.

[112] Jacek Kopeckỳ, Paul Fremantle, and Rich Boakes. A history and future of Web APIs. *it-Information Technology*, 56(3):90–97, 2014.

[113] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005.

[114] Daniel Le, Nathaniel Neudecker, Diana Winckler, Heiko Dosch, Anastasija Nikitin, Stefanie Auge-Dickhut, Christian Betz, and Katharina Schache. Understanding the business relevance of Open APIs and open banking for banks, informaton paper, 2020. https://developer.commerzbank.com/shared/documents/commerzbank-open-banking-whitepaper-2020.pdf.

[115] Lucas Leal, Leonardo Montecchi, Andrea Ceccarelli, and Eliane Martins. Exploiting mde for platform-independent testing of service orchestrations. In *2019 15th European Dependable Computing Conference (EDCC)*, pages 149–152. IEEE, 2019.

[116] Henrik Leopold, Jan Mendling, and Oliver Günther. Learning from quality issues of BPMN models from industry. *IEEE Software*, 33(4):26–33, 2015.

[117] Li Li and Wu Chou. Design and Describe REST API without Violating REST: A Petri Net Based Approach. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 508–515, July 2011.

[118] Li Li and Wu Chou. Designing Large Scale REST APIs Based on REST Chart. In *Web Services (ICWS), 2015 IEEE International Conference on*, pages 631–638, June 2015.

[119] O.I. Lindland, G. Sindre, and A. Solvberg. Understanding quality in conceptual modeling. *Software, IEEE*, 11(2):42–49, March 1994.

[120] Olga Liskin, Leif Singer, and Kurt Schneider. Teaching Old Services New Tricks: Adding HATEOAS Support as an Afterthought. In *Proceedings of the Second International Workshop on RESTful Design*, pages 3–10. ACM, 2011.

[121] Jerome Louvel. A Short History of OAI and API Specifications. http://restlet.com/company/blog/2017/04/26/a-short-history-of-oai-and-api-specifications/, 2017.

[122] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–24, 2019.

[123] Divyanand Malavalli and Sivakumar Sathappan. Scalable microservice based architecture for enabling dmtf profiles. In *2015 11th International*

Conference on Network and Service Management (CNSM), pages 428–432. IEEE, 2015.

[124] Salome Maro, Jan-Philipp Steghöfer, Anthony Anjorin, Matthias Tichy, and Lars Gelin. On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience. In *Proc. of the International Conference on Software Language Engineering*, pages 1–12. ACM, 2015.

[125] Marius Marusteri and Vladimir Bacarea. Comparing groups for statistical differences: how to choose the right statistical test? *Biochemia medica*, 20(1):15–32, 2010.

[126] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[127] Florent Masseglia, Pascal Poncelet, and Maguelonne Teisseire. Incremental mining of sequential patterns in large databases. *Data & Knowledge Engineering*, 46(1):97–121, 2003.

[128] Tony Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design, 2015.

[129] Martin Mazanec and Ondrej Macek. On general-purpose textual modeling languages. In *Proc. of the International Workshop on DAtabases, TExts, Specifications and Objects*, volume 837, pages 1–12, 2012.

[130] Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen. *Continuous API Management: Making the Right Decisions in an Evolving Landscape*. O'Reilly Media, 2018.

[131] Santiago Meliá, Cristina Cachero, Jesús M. Hermida, and Enrique Aparicio. Comparison of a textual versus a graphical notation for the maintainability of MDE domain models: an empirical pilot study. *Software Quality Journal*, 24(3):709–735, 2016.

[132] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2011.

[133] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[134] Gerard Meszaros and Jim Doble. A pattern language for pattern writing. *Pattern languages of program design*, 3:529–574, 1998.

[135] Zoltán Micskei and Hélène Waeselynck. The many meanings of UML 2 Sequence Diagrams: a survey. *Software & Systems Modeling*, 10(4):489–514, 2011.

[136] George A Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[137] Ronnie Mitra. Rapido: A Sketching Tool for Web API Designers. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 1509–1514, Florence, Italy, 2015.

[138] Parastoo Mohagheghi and Øystein Haugen. Evaluating domain-specific modelling solutions. In *International Conference on Conceptual Modeling*, pages 212–221. Springer, 2010.

[139] Thomas Molka, David Redlich, Marc Drobek, Artur Caetano, Xiao-Jun Zeng, and Wasif Gilani. Conformance checking for BPMN-based process models. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1406–1413, 2014.

[140] Jefferson Seide Molléri, Kai Petersen, and Emilia Mendes. CERSE-catalog for empirical research in software engineering: A systematic mapping study. *Information and Software Technology*, 105:117–149, 2019.

[141] Anna Monus. SOAP vs REST vs JSON comparison. `https://raygun.com/blog/soap-vs-rest-vs-json/`, 2019.

[142] Daniel Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Trans. Softw. Eng.*, 35(6):756–779, November 2009.

[143] Gavin Mulligan and Denis Gračanin. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 1423–1432. IEEE, 2009.

[144] Brad A Myers. Visual programming, programming by example, and program visualization: a taxonomy. *ACM sigchi bulletin*, 17(4):59–66, 1986.

[145] Rahul Narain, Alex Merrill, and Eric Lesser. Evolution of the API economy-adopting new business models to drive future innovation. `https://www.ibm.com/downloads/cas/XG8RY063`, 2016. Last accessed: 2019-07-01.

[146] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An analysis of public REST Web service APIs. *IEEE Transactions on Services Computing*, 2018.

[147] Eric Newcomer and Greg Lomow. *Understanding SOA with Web services*. Addison-Wesley, 2005.

[148] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[149] Adriatik Nikaj. *RESTful Choreographies*. PhD thesis, Business Process Technology Group, Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam, Germany, 2019.

[150] Adriatik Nikaj, Marcin Hewelt, and Mathias Weske. Towards implementing REST-enabled business process choreographies. In *International Conference on Business Information Systems*, pages 223–235. Springer, 2018.

[151] Adriatik Nikaj, Sankalita Mandal, Cesare Pautasso, and Mathias Weske. From Choreography Diagrams to RESTful Interactions. In *The Eleventh International Workshop on Engineering Service-Oriented Applications*, pages 3–14, 2015.

[152] Adriatik Nikaj and Mathias Weske. Formal Specification of RESTful Choreography Properties. In *Proceedings of the International Conference on Web Engineering (ICWE'16)*, pages 365–372. Springer, 2016.

[153] Adriatik Nikaj, Mathias Weske, and Jan Mendling. Semi-automatic derivation of RESTful choreographies from business process choreographies. *Software & Systems Modeling*, pages 1–14, 2017.

[154] Mark Nottingham. Web Linking. Request for Comments: 5988, October 2010.

[155] Joshua Ofoeda, Richard Boateng, and John Effah. Application programming interface (api) research: A review of the past to inform the future. *International Journal of Enterprise Information Systems (IJEIS)*, 15(3):76–95, 2019.

[156] Avner Ottensooser, Alan Fekete, Hajo A Reijers, Jan Mendling, and Con Menictas. Making sense of business process descriptions: An experimental comparison of graphical and textual notations. *Journal of Systems and Software*, 85(3):596–606, 2012.

[157] Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. A tutorial on metamodelling for grammar researchers. *Science of Computer Programming*, 96:396–416, 2014.

[158] Srikanta Patanjali, Benjamin Truninger, Piyush Harsh, and Thomas Michael Bohnert. Cyclops: a micro service based approach for dynamic rating, charging & billing for cloud. In *Telecommunications (ConTEL), 2015 13th International Conference on*, pages 1–8. IEEE, 2015.

[159] Sanjay Patni. *Pro RESTful APIs*. Springer, 2017.

[160] Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. Modeling RESTful conversations with extended BPMN choreography diagrams. In Danny Weyns, Raffaela Mirandola, and Ivica Crnkovic, editors, *Proc. of the 9th European Conference on Software Architecture*, volume 9278 of *Lecture Notes in Computer Science*, pages 87–94. Springer, 2015.

[161] Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. A pattern language for RESTful conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs*, page 4. ACM, 2016.

[162] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful web services vs." big'"web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814, 2008.

[163] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[164] Marian Petre. UML in practice. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 722–731. IEEE Press, 2013.

[165] Ivan Porres and Irum Rauf. Modeling behavioral RESTful web service interfaces in UML. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1598–1605, 2011.

[166] Postman. 2020 state of the API report. `https://www.postman.com/state-of-api-report-2020.pdf`, 2020.

[167] Mageswari Rajoo and Noor Maizura Mohamad Noor. Important evaluation factors of UML tools for health informatics. *Journal of Telecommunication, Electronic and Computer Engineering*, 9(3-3):191–195, 2017.

[168] Luke V Rasmussen, Will K Thompson, Jennifer A Pacheco, Abel N Kho, David S Carrell, Jyotishman Pathak, Peggy L Peissig, Gerard Tromp, Joshua C Denny, and Justin B Starren. Design patterns for the development of electronic health record-driven phenotype extraction algorithms. *Journal of biomedical informatics*, 51:280–286, 2014.

[169] D. M. Rathod, S. M. Parikh, and B. V. Buddhadev. Structural and behavioral modeling of RESTful web service interface using UML. *2013 International Conference on Intelligent Systems and Signal Processing (ISSP)*, pages 28–33, 2013.

[170] Irum Rauf. *Design and validation of stateful composite RESTful web services*. PhD thesis, Turku Centre for Computer Science, 2014.

[171] Iris Reinhartz-Berger, S Cohen, J Bettin, T Clark, and A Sturm. *Domain engineering: Product Lines, Languages, and Conceptual Models*. Springer, 2013.

[172] Chris Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.

[173] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O'Reilly, 2013.

[174] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[175] Erkuden Rios, Teodora Bozheva, Aitor Bediaga, and Nathalie Guilloreau. MDD maturity model: A roadmap for introducing model-driven development. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 78–89. Springer, 2006.

[176] Martin P Robillard. What makes APIs hard to learn? Answers from developers. *Software, IEEE*, 26(6):27–34, 2009.

[177] Stewart Robinson, Roger Brooks, Kathy Kotiadis, and Durk-Jouke Van Der Zee. *Conceptual Modeling for Discrete-Event Simulation*. CRC Press, Inc., 2010.

[178] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. REST APIs: a large-scale analysis of compliance with principles and best practices. In *International conference on web engineering*, pages 21–39. Springer, 2016.

[179] Manuel Caeiro Rodríguez, Martín Llamas Nistal, and Luis Anido Rifón. Towards a benchmark for the evaluation of ld expressiveness and suitability. *Journal of Interactive Media in Education*, 2005(1), 2005.

[180] Rolando Rodríguez, Roberto Espinosa, Devis Bianchini, Irene Garrigós, Jose-Norberto Mazón, and Jose Jacobo Zubcoff. Extracting models from web API documentation. In *International Conference on Web Engineering*, pages 134–145. Springer, 2012.

[181] Michael Rosen, Boris Lublinsky, Kevin T Smith, and Marc J Balcer. *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons, 2012.

[182] Margaret Rouse. RESTful API? `https://searchmicroservices.techtarget.com/definition/RESTful-API?src=5919398&asrc=EM_ERU_115532474&utm_content=eru-rd2-rcpC&utm_medium=EM&utm_source=ERU&utm_campaign=20190703_ERU%20Transmission%20for%2007/03/2019%20(UserUniverse:%20559842)`, 2019.

[183] Safdar Aqeel Safdar, Muhammad Zohaib Iqbal, and Muhammad Uzair Khan. Empirical evaluation of UML modeling tools–a controlled experiment. In *European Conference on Modelling Foundations and Applications*, pages 33–44. Springer, 2015.

[184] Gerald Schermann, Jürgen Cito, and Philipp Leitner. All the services large and micro: Revisiting industrial practice in services computing. In Alex Norta, Walid Gaaloul, G. R. Gangadharan, and Hoa Khanh Dam, editors, *Service-Oriented Computing – ICSOC 2015 Workshops*, pages 36–47, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[185] Douglas C Schmidt. Model-driven engineering. *Computer*, 39(2):25, 2006.

[186] Douglas C. Schmidt, Mohamed Fayad, and Ralph E. Johnson. Software patterns. *Commun. ACM*, 39(10):37–39, October 1996.

[187] Silvia Schreier. Modeling RESTful applications. In *Proceedings of the Second International Workshop on RESTful Design*, pages 15–21. ACM, 2011.

[188] Ambler Scott. Simple tools for software modeling -or- it's "use the simplest tool" not "use simple tools". `http://www.agilemodeling.com/essays/simpleTools.htm`, 2008. Last accessed: 2018-08-20.

[189] Stephan Seifermann and Henning Groenda. Survey on the applicability of textual notations for the unified modeling language. In *International Conference on Model-Driven Engineering and Software Development*, pages 3–24. Springer, 2016.

[190] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19–25, 2003.

[191] Cristian Sepulveda, Rosa Alarcon, and Jesus Bellido. QoS aware descriptions for RESTful service composition: security domain. *World Wide Web*, 18(4):767–794, 2015.

[192] Zohreh Sharafi, Alessandro Marchetto, Angelo Susi, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 33–42. IEEE, 2013.

[193] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming web services with SOAP: building distributed applications*. " O'Reilly Media, Inc.", 2001.

[194] Ellis Solaiman, Wenzhong Sun, and Carlos Molina-Jimenez. A tool for the automatic verification of BPMN choreographies. In *2015 IEEE international conference on services computing*, pages 728–735. IEEE, 2015.

[195] EBNF Syntaxt Specification Standard. EBNF: ISO/IEC 14977: 1996 (e). `https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`, 1996.

[196] Thomas Steiner and Jan Algermissen. Fulfilling the Hypermedia Constraint Via HTTP OPTIONS, the HTTP Vocabulary In RDF, And Link Headers. In *Proceedings of the Second International Workshop on RESTful Design*, pages 11–14. ACM, 2011.

[197] Harald Störrle. On the impact of size to the understanding of UML diagrams. *Software & Systems Modeling*, 17(1):115–134, 2018.

[198] Michael Stowe. *Undisturbed REST: A guide to designing the perfect API*. mulesoft.com, 2015.

[199] Andrzej Stroinski, Dariusz Dwornikowski, and Jerzy Brzezinski. Resource mining: Applying process mining to resource-oriented systems. In *Business Information Systems: 17th International Conference, BIS 2014, Larnaca, Cyprus, May 22-23, 2014, Proceedings*, volume 176, page 217. Springer, 2014.

[200] Andrzej Stroinski, Dariusz Dwornikowski, and Jerzy Brzezinski. RESTful web service mining: Simple algorithm supporting resource-oriented systems. In *Proc. of ICWE*, pages 694–695. IEEE, 2014.

[201] Andrzel Stroinski, Dariusz Dwornikowski, and Jerzy Brzezinski. RESTful web service mining: Simple algorithm supporting resource-oriented systems. In *2014 IEEE International Conference on Web Services*, pages 694–695, June 2014.

[202] John Sweller. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review*, 22(2):123–138, 2010.

[203] Daniel Szmukler. Understanding the business relevance of Open APIs and open banking for banks, informaton paper, 2016. `https://www.abe-eba.eu/media/azure/production/1522/business-relevance-of-open-apis-and-open-banking-for-banks.pdf`.

[204] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[205] Stefan Tilkov, Martin Eigenbrodt, Silvia Schreier, and Oliver Wolf. *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web (in German)*. dpunkt.verlag, 3rd edition, 2015.

[206] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pages 19–24. ACM, 2015.

[207] Francisco Valverde and Oscar Pastor. Dealing with REST services in model-driven web engineering methods. *V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB*, pages 243–250, 2009.

[208] Wil van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[209] Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves De Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter Van Den Brand, Ronald Brandtjen, Joos Buijs, et al. Process mining manifesto. In *International Conference on Business Process Management*, pages 169–194. Springer, 2011.

[210] Wil MP Van der Aalst and Minseok Song. Mining social networks: Uncovering interaction patterns in business processes. In *International conference on business process management*, pages 244–260. Springer, 2004.

[211] Wil MP van der Aalst and HMW (Eric) Verbeek. Process mining in web services: The WebSphere case. *IEEE Data Eng. Bull.*, 31(3):45–48, 2008.

[212] Dirk van der Linden, Irit Hadar, and Anna Zamansky. What practitioners really want: requirements for visual notations in conceptual modeling. *Software & Systems Modeling*, 18(3):1813–1831, 2019.

[213] Dirk van der Linden, Anna Zamansky, and Irit Hadar. How cognitively effective is a visual notation? on the inherent difficulty of operationalizing the physics of notations. In Rainer Schmidt, Wided Guédria, Ilia Bider, and Sérgio Guerreiro, editors, *Enterprise, Business-Process and Information Systems Modeling - 17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13-14, 2016, Proceedings*, volume 248 of *Lecture Notes in Business Information Processing*, pages 448–462. Springer, 2016.

[214] Dirk Van Der Linden, Anna Zamansky, and Irit Hadar. A framework for improving the verifiability of visual notation design grounded in the physics of notations. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 41–50. IEEE, 2017.

[215] Oliver van Porten. Development and Evaluation of a Graphical Notation for Modelling Resource-Oriented Applications. Master's thesis, FernUniversität, Hagen, Germany, 2012.

[216] HMW Verbeek, JCAM Buijs, BF Van Dongen, and Wil MP van der Aalst. ProM 6: The process mining toolkit. *Proc. of BPM Demonstration Track*, 615:34–39, 2010.

[217] Ruben Verborgh, Michael Hausenblas, Thomas Steiner, Erik Mannens, and Rik Van de Walle. Distributed Affordance: An Open-World Assumption for Hypermedia. In *Proceedings of the 4th International Workshop on RESTful Design*, pages 1399–1406. ACM, 2013.

[218] John Vester. RESTful API lifecycle management. `https://unityconstruct.org/media/doc/coding/dzone-ref-cardz/4960646-dzone-rc238-restfulapilifecyclemanagement.pdf`, 2017.

[219] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Software & Systems Modeling*, 18(1):585–630, 2019.

[220] Markus Völter, Michael Kircher, and Uwe Zdun. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Wiley, 2013.

[221] Kishor Wagh and Ravindra Thool. A comparative study of SOAP vs REST web services provisioning techniques for mobile host. *Journal of Information Engineering and Applications*, 2(5):12–16, 2012.

[222] Sawitree Weerapong, Parham Porouhan, and Wichian Premchaiswadi. Process mining using α-algorithm as a tool (a case study of student registration). In *2012 Tenth International Conference on ICT and Knowledge Engineering*, pages 213–220. IEEE, 2012.

[223] Mathias Weske. *Business Process Management: Concepts, Languages, and Architectures*. Springer, 2nd edition, 2012.

[224] RICHARD Wettel, M Lanza, and R Robbes. Empirical validation of codecity: A controlled experiment. *Tech Report 2010/05*, 2010.

[225] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 551–560. IEEE, 2011.

[226] Petia Wohed, Marlon Dumas, Arthur HM Ter Hofstede, and Nick Russell. Pattern-based analysis of BPMN-an extensive evaluation of the control-flow, the data and the resource perspectives. 2006.

[227] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[228] HyunKyung Yoo, YooMi Park, and Hyunjoo Bae. Semi-automatic semantic service annotation for SOAP and REST web services. In *Digital Enterprise and Information Systems*, pages 70–77. Springer, 2011.

[229] Johannes Maria Zaha, Alistair Barros, Marlon Dumas, and Arthur ter Hofstede. Let's dance: A language for service behavior modeling. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 145–162. Springer, 2006.

[230] Uwe Zdun and Mark Strembeck. Reusable architectural decisions for DSL design: Foundational decisions in DSL projects. In Michael Weiss, editor, *EuroPLoP 2009: 14th Annual European Conference on Pattern Languages of Programming, Irsee, Germany, July 8-12, 2009*, volume 566 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[231] Andreas Zeller, Thomas Zimmermann, and Christian Bird. Failure is a four-letter word: a parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, pages 1–7, 2011.

[232] Zhongheng Zhang. Missing data imputation: focusing on single imputation. *Annals of translational medicine*, 4(1), 2016.

[233] Olaf Zimmermann. Microservices tenets. *Computer Science - Research and Development*, 32(3):301–310, Jul 2017.

[234] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface representation patterns: Crafting and consuming message-based remote apis. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, page 27. ACM, 2017.

[235] Michael Zimoch, Rüdiger Pryss, Thomas Probst, Winfried Schlee, and Manfred Reichert. The repercussions of business process modeling notations on mental load and mental effort. In *Proc. BPM*, pages 133–145. Springer, 2018.

[236] Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standardsâĂŤthe case of REST vs. SOAP. *Decision Support Systems*, 40(1):9 – 29, 2005.

[237] Michael Zur Muehlen, Jan Recker, and Marta Indulska. Sometimes less is more: Are process modeling languages overly complex? In *2007 Eleventh International IEEE EDOC Conference Workshop*, pages 197–204. IEEE, 2007.

[238] Ivan Zuzak, Ivan Budiselic, and Goran Delac. A Finite-State Machine Approach for Modeling and Analyzing RESTful Systems. *Journal of Web Engineering*, 10(4):353–390, December 2011.