João Campinhos

A Versioned Approach to Web Service Evolution

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientadores:	Jácome Cunha, Assistant Professor, NOVA University of Lisbon
	João Costa Seco, Assistant Professor, NOVA University of Lisbon

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2016

A Versioned Approach to Web Service Evolution

Copyright © João Campinhos, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa.
A Faculdade de Ciências e Tecnologia e a Universidade NOVA de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objetivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

To my familly and friends.

Acknowledgements

I would like to thank my advisors, Prof. Jácome Cunha and Prof. João Costa Seco for the opportunity given and all the support throughout the duration of this project.

My colleagues and friends in the department that day after day were there to push me into a job well done.

At last, I would like to thank my family and friends, for their patience and support, without which this work would not have been possible.

Abstract

Applications based on micro-services or web services have had a significant growth due to the exponential increase in the use of mobile devices whose applications rely almost entirely on this type of interfaces. However, using an external interface comes with no guarantees to the developer. Changes may be introduced at any moment, which can break the software that uses that API. It is necessary to give the consumers guarantees that their software will not break, but not at the expense of stagnating the development of said web service.

In this document we present a programming model to evolve web services in a sustainable way and to automate most of the maintainability that might be required by the client. This model works by allowing multiple versions to be deployed, and then using a relation containing metadata to type check versions. By doing this, it is possible to guarantee type safety between all the versions to provide a sustainable way to evolve the service. A prototype framework was implemented in JavaScript, where it is possible to visualize the model working in an environment similar to what it is used in the industry nowadays.

Finally we present a comparison of our prototype with the state of the art, thus demonstrating that our solution presents a viable method of evolution of web services.

Keywords: Software Variability, Web Services, Software Evolution, Versioning, Programming Languages

Resumo

As aplicações baseadas em micro-serviços ou serviços web tiveram um crescimento significativo devido ao aumento exponencial da utilização de dispositivos móveis, cujas aplicações dependem quase exclusivamente deste tipo de interfaces. No entanto, utilizar uma interface externa não dá nenhuma garantia ao programador. Podem ser introduzidas alterações a qualquer momento, o que pode inutilizar o software que consome essa API. É necessário dar garantias ao consumidor de que o seu software não vai deixar de funcionar, mas não à custa da estagnação do desenvolvimento do referido serviço web.

Neste documento apresentamos um modelo de programação para desenvolver serviços web de uma forma sustentável, automatizando parte da manutenção que possa ser exigida ao cliente do serviço. Este modelo funciona permitindo que múltiplas versões possam ser disponibilizadas, utilizando uma relação contendo metadados para fazer a verificação de tipos entre as versões. Ao fazer isso, é possível garantir a segurança de tipos entre todas as versões criando assim um método sustentável de evolução do serviço. Também implementamos um protótipo em JavaScript onde é possível visualizar o modelo a funcionar num ambiente semelhante ao que é utilizado na indústria.

Por fim apresentamos uma comparação do nosso protótipo com o estado da arte, demonstrando assim que a nossa solução apresenta um método viável de evolução de serviços web.

Palavras-chave: Variabilidade no Software, Serviços Web, Evolução do Software, Versionamento, Linguagens de Programação

Contents

List of Figures **xv**

Listings **xvii**

1 Introduction **1**
- 1.1 Problem Statement 2
- 1.2 Our Solution 2
  - 1.2.1 Contributions 3
- 1.3 Document Organization 3

2 State of the Art **5**
- 2.1 Web Api Evolution 5
  - 2.1.1 Chain of Adapters 6
  - 2.1.2 RIDDL 7
  - 2.1.3 VRESCo 9
  - 2.1.4 WSDLDiff 9
  - 2.1.5 hRESTS 10
  - 2.1.6 RESTdesc 11
  - 2.1.7 restify 13
- 2.2 Choice Calculus 14
  - 2.2.1 Syntax 14
  - 2.2.2 Choice Elimination 15
  - 2.2.3 Type System 16
- 2.3 Software Product Lines 17

3 Typesafe Evolution of Web Services **21**
- 3.1 Programming Model Principles 21
- 3.2 Version Relations 22
- 3.3 Versioned Featherweight Java 23
  - 3.3.1 Syntax 24
  - 3.3.2 Type System 25
  - 3.3.3 Operational semantics 25

4 Prototype **31**
4.1 Technologies 31
4.2 Implementation 32
4.2.1 Version Context 33
4.2.2 Type System 33
4.2.3 Framework Router 34
4.2.4 Relation 36
4.3 Architecture 37

5 Evaluation **39**
5.1 ASP.NET API Versioning 40
5.2 restify 41
5.3 niVerso 41
5.4 Final remarks 42

6 Conclusion **45**
6.1 Future Work 46

Bibliography **47**

A ASP.NET implementation **51**

B Restify implementation **53**

C niVerso implementation **57**

List of Figures

2.1 Chain of Adapters 7
2.2 Choice calculus syntax 14
2.3 Feature Diagram 18

3.1 Example of a simple relation 23
3.2 Example of a more intricate relation 23
3.3 Syntax of versioned Featherweight Java 24
3.4 Lookup functions 27
3.5 Typing rules 28
3.6 Operational Semantics 29

4.1 Example of the type system structure 34
4.2 Prototype architecture 37

Listings

2.1 RIDDL example . . . 8
2.2 hRESTS annotations . . . 11
2.3 RESTdesc example . . . 12
2.4 restify example . . . 13
2.5 Choice calculus example . . . 15
3.1 Example of two classes defined using Featherweight Java . . . 24
4.1 Example of a naive code transformer . . . 32
4.2 Example with valid JavaScript . . . 33
4.3 Express route . . . 35
4.4 niVerso route . . . 35
4.5 niVerso example server . . . 35
4.6 Sample relation implemented in JavaScript . . . 36
4.7 Relation from Figure 4.1 implemented in JavaScript . . . 36
5.1 A versioned route using ASP.NET . . . 40
5.2 A versioned route using restify . . . 41
5.3 Comparison between niVerson and restify . . . 42

Chapter 1

Introduction

Software is being developed more and more to adapt itself to the client's necessities on different environments. Whether it is by allowing it to be compiled to multiple architectures or by working with various database management systems, software adapts through configurable behavior. This configurable behavior that modifies how the system handles itself is what can be defined as variability. Although this makes software more reusable, it is hard to reason on a code base when we add variability throughout it. Even this being a common practice, it will add complexity to the code. By inserting small variability snippets on different parts of the code, developers are diminishing its readability, which can result in bugs that are hard to track. Spencer et al. explores this issue, explaining why adding variability through preprocessors like CPP [8] with `#ifdef` is considered harmful and what can developers do to reduce its impact [24].

However, variability can be useful when it is integrated into the system instead of being an add-on. A proper use of variability will improve software's readability and maintainability. We will see this in more detail in **Section** 2.2

There are a lot of areas that could be improved with a responsible use of variability. Web service evolution is a good example, and it is the primary focus of this work. Web services tend to work on a single version that gets updated or with multiple versions, isolated from each other. The introduction of variability on web service evolution unifies the multiple versions integrating them into a single one, giving a clearer view of the software's evolution with easy to reason about code.

There is a need to develop better web service evolution systems, but the lack of a standard way to do it often results in *ad hoc* solutions that will not work on every use case. That is the main argument for the use of variability in this setting, as it paves the way for our solution.

1.1 Problem Statement

Variability can be harmful, and it should be used carefully so it does not grow out of proportion [1]. Looking at web development, in specific to the state of web services, developers spend a considerable amount of time designing the interface because, once it is published, any significant changes may cause disruption on applications that rely on the said service. This methodology is the opposite of what is happening in the web development world. Teams are deploying faster and more often, using agile methodologies like scrum instead of waterfall software development, so it is only logical that the same happens with web services. Allowing this evolution will often result in more breaking changes. Those changes, if not dealt with accordingly by the developers of client applications, could render that software useless, as they will not comply with recent API changes. Thus, the goal of this work it to find a solution that keeps the client applications running for as long as possible whithout stoping the service evolution.

1.2 Our Solution

Although there are services that already let us track API documentation changes [2], for interfaces that change often, maintaining client code that uses those APIs can easily become a daunting task. Thus, we present a programming model that allows developers to focus on evolving their web services, while also easing the burden of maintaining a fast changing interface. Our approach to support the evolution of web APIs is to use variability mechanisms in the server definition. We allow methods to be tagged in a particular version of the system, and the tool ensures the soundness of the entire server system. This approach resembles a code slicing approach, where all related code is guaranteed to consistently contribute to a version (or sets of compatible versions) of the system. This model works with any versioning system if it is possible to implement our relation interface, that works with a tree of versions and relations between those versions, as well as their level of compatibility between them. As for the clients using the web service, we can guarantee that they will not have any execution error due to an unexpected server side upgrade, which is tipically what occurs on APIs without a versioning system. Our solution, which we modeled as an extension of an object calculus, creates a version context to allow a propper execution of a given version on a safe way. Due to the type system we developed to type check the versions. we can provide those compatibility guarantees on a sound system. It is then possible to ensure that only compatible versions are returned to the client, without compromising the clients software. We will explain our solution further and all the technicalities behind it in **Chapter** 3.

Our implementation was carefully designed to allow a seamless integration with the workflow currently used when creating web services. We have implemented a prototype in JavaScript, that takes advantage of the type annotations of Flow [4], a static type checker, to type check the versions ensuring the desired compatibility between versions.

This way we have a solution that does not require any boilerplate code to work, since those solutions will often not gain traction with existing web services due to the time it takes to prepare them to work with the new system. It is also useful when comparing how this solution performs when comparing it to the state of the art.

1.2.1 Contributions

The main contributions of this thesis are the following:

- A programming model that eases the maintenance of web services by dealing with evolution as a component of the whole system, reducing the damage that breaking changes might have on the clients.
- A type checker to ensure the soundness of the system, by type checking each version individually and compatibility between versions on a given relation.
- An implementation of our programming model in a framework to support multiple API versions and to redirect requests accordingly.

Publications

Part of the results in this thesis were published in the form of a communication, accompanied with a poster and a presentation.

Evolução Controlada de Arquitecturas de Serviços Web João Campinhos, João Costa Seco and Jácome Cunha. Comunicações do oitavo Simpósio de Informatica, Lisboa, September, 2016

1.3 Document Organization

The remainder of this document is organized as follows:

Chapter 2 describes the state of the art. Web service evolution state and its techniques and tools are described in this chapter. It provides an introduction to Choice Calculus and Software Product Lines, since these are the main theoretic basis behind our work.

Chapter 3 describes the thought process behind our solution as well as an example language with a built in version control system. We present its syntax, semantics and the type system that supports it.

Chapter 4 presents a prototype that implements our programming model. We justify our technology decisions, and all the implementation details.

Chapter 5 presents an evaluation of our approach, which is based on a comparison with existing solutions, thus being able to evaluate the various aspects of our solution.

Chapter 6 concludes this work with a summary, and gives some possible directions for future work.

Chapter 2

State of the Art

This chapter covers a wide array of different fields, since we are addressing variability in software and the problem of web service evolution. We relate previous work with ours, and how we can build upon that. In **Section** 2.1, we present a case study on how some popular public APIs are evolving, followed by tools and techniques that can be used to achieve this evolution. In **Section** 2.2, we describe the choice calculus language and how its foundation serves as a base for our work, being a generic language aiming to solve the problem of variability. In **Section** 2.3, we introduce software product lines, an important development paradigm that helps us abstract parts of our problem using feature models, a concept from software engineering that helps reason about multiple variations of the software system.

2.1 Web Api Evolution

Researchers have put a lot of thought on the topic of web API evolution. Like we said earlier, since the first minute an API is deployed, there could be people using it, and deprecate APIs endpoints is not a viable solution. Likewise, we need to make sure that we evolve the API without compromising any of the features already implemented. Sohan et al studied some of the most popular APIs on how they approached evolution [23]. Due to the lack of a standard way to evolve APIs, they found out and categorized various solutions on how to deal with different versions.

Numbered identifiers

This solution uses an integer to identify the API's version. It gives the client a sense of evolution but it is useless as is, since it does not provide a way for the API client to know what changed and what is deprecated.

Timestamped identifiers

This solution uses a timestamp to identify versions. Like the solution presented above, this one creates a separation of code and makes the use of different versions of the API hard, if possible at all.

Major and Minor identifiers

Although this solution provides more information to the client, it is still hard to understand if there are breaking changes just by looking at the version number. Not only that but due to the fast evolutive nature of the web, publishers often add breaking changes without publishing them into a new version.

However, not every API supports multiple versions simultaneously.

Single version

This solution uses a single version and deprecates older versions. It gives the users a specific timeframe to migrate. This solution is the most disruptive, as any application that uses this APIs needs maintenance at unexpected times.

Multiple versions

A lot of the popular web APIs uses this method of supporting several versions. This is good because it gives users plenty of time to migrate to the newest version, as well as makes possible the use of different versions at the same time. However, if a developer allows the API to evolve without migrating, the task of migration will get harder on each new version since multiple breaking changes might be introduced in every version.

This case study [23] concludes giving a list of recommendations on how new research can approach the problems of Web API versioning. These recommendations focus on how versioning needs to provide information on what changed. Semantic Versioning [21] is a good example of a naming technique that could provide that. The other question that this analysis addresses is the fact that there needs to be a way to easily deliver multiple versions of the API at the same time, to provide a better environment for the developer to migrate to the new versions. We will take both of these recommendations in consideration and will explore them further in **Chapter** 3. On the following subsections, we present three existing techniques and three tools, focused on web service evolution, with both REST and SOAP being addressed.

2.1.1 Chain of Adapters

Chain of adapters [14] is a design technique used for evolving Web Services. The goal of chain of adapters is to permit the evolution of a service's interface and implementation while remaining backwards-compatible with clients written to comply with previous versions. Not only that but it focuses on retaining a common data store to achieve a consistent state throughout all the versions and it tries to avoid code duplication by incrementally extend the web services interface. The developer should also be allowed to refactor, re-design, and otherwise rethink both the service's interface and its implementation without being shackled by previous decisions.

The chain of adapters works by duplicating the interface into a separate namespace. This interface needs to be implemented in the form of a pass-through adapter for the current version. Then we rename it with the desired version (v1 for the first version) and publish the v1 interface endpoint. As for the following versions, you proceed in the same way. If you are changing existing operations, the new adapter needs to provide a default value when forwarding. Changes in the data structures need to be translated from the old one to the new one or vice versa. When removing an operation, it needs to be re-implemented in the adapter using other operations available in the current interface. By using the "freeze, adapt, and delegate" technique, we can create a chain of adapters as shown in **Figure** 2.1.

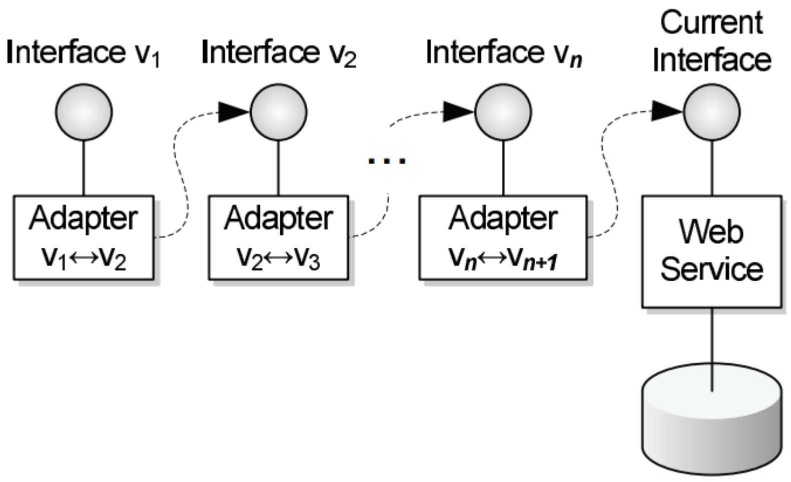
Interface v_1
Interface v_2
Interface v_n
Current Interface
Adapter $v_1 \leftrightarrow v_2$
Adapter $v_2 \leftrightarrow v_3$
...
Adapter $v_n \leftrightarrow v_{n+1}$
Web Service

Figure 2.1: Chain of Adapters

This technique provides a Web Service evolution that consists of multiple versions concurrently deployed without code repetition but preserving backward compatibility. Since every version remains isolated on its own adapter, it should be easy to remove a version, as long as we are removing in a chronological order (oldest-to-newest). We can also evolve new versions unconstrained, as long as we can reimplement the previous contract. As the author states, one thing that we must consider when using the chain of adapters is the overhead that multiple forwards will generate when multiple versions are implemented.

2.1.2 RIDDL

RIDDL is an XML-based language used to incrementally compose REST APIs documentation by adding a changelog of the older version [18].

Because the lack of a standard way to describe REST interfaces, they cannot benefit from the same advantages as a Web Service using a description language (WSDL). There are several benefits of having an interface definition language, such as supporting the generation of skeleton code, development support through visualization tools, and even sharing the same configuration through client and server. But RIDDL also covers the requirements of service composition and evolution, to allow changes in the interface and implementation while remaining backwards-compatible. Although WSDL 2.0 can be used to describe RESTful services, because it is still missing some features and the results are verbose, it has not gotten much popularity.

The way we describe a RESTful service using RIDDL can be observed in **Listing** 2.1.

Listing 2.1: RIDDL example

```
1  <description xmlns="http://riddl.org/ns/description/1.0">
2    <message name="update">
3      <header name="authorization" type="string"/>
4      <parameter name="status" type="string"/>
5    </message>
6    <message name="xml">
7      <parameter name="return" mimetype="text/xml"/>
8    </message>
9    <resource>
10     <resource relative="statuses">
11       <resource relative="show">
12         <resource relative ="\d+\.xml">
13           <get in="*" out="xml"/>
14         </resource>
15       </resource>
16       <resource relative="update.xml">
17         <post in="update" out="xml"/>
18       </resource>
19     </resource>
20   </resource>
21 </description>
```

REST resources are described as a tree structure. In the example above, we are looking at the resource `/statuses` or `/statuses/update.xml` and a `GET` and `POST`, respectively. Request headers can also be specified as we can see on line 2. As for service evolution, RIDDL has a similar approach as Chain of Adapters 2.1.1. Each adapter (that works as a different version of the API) instead of being an interface is a service. The multiple services descriptions are then mashed together to add backwards-compatibility. The paper is not clear on how RIDDL proceeds in the case of a modification of one operation, that would inevitably cause a conflict, but we are assuming the basic solution which is to retain the new version and ignore the older one.

While RIDDL is a good alternative to what is already available with WSDL, with the added benefit of allowing service evolution, our solution will try to be the least intrusive possible, to keep the simplicity of REST intact, since it is what makes developers pick it instead of established service oriented architectures like SOAP.

2.1.3 VRESCo

VRESCo is a runtime environment that acts as a proxy between the client calls and the actual version of the web service. This is done by using version tags such as INITIAL, STABLE or LATEST [17]. This paper starts by differentiating multiples types of changes that could be made to evolve the code base. Let us focus on Interface changes that can be to add operations, remove or change the operations' signature. Adding an operation is what can be called a **transparent change**. It means that the runtime can handle this change automatically. This makes sense because when we add a new operation, the client code that was working on a previous version will work on the new one without modifications. We cannot say the same for removing operations because the client code now needs to be adapted to eliminate the need for that operation. Changing the operation signature can also be transparent if we are not changing the mandatory parameter list (assuming the runtime accepts a variable number of parameters.

These are the changes the runtime can take advantage of, as for the versions, VRESCo also as a way to handle them. The versions themselves are handled via a service graph. Each node of the graph represents an actual version of the Web Service. The root represents the first version of the Service and each branch represents two or more variants that are evolving in parallel. Some of those nodes have tags that the client can pick when connecting to VRESCo proxy. These tags can be defined by the developer or can be selected automatically by VRESCo (if we are evolving with transparent changes). VRESCo will then proxy the client calls to the right version.

Proxy rebinding can be **fixed**, **Periodic**, **OnDemand**, and **OnInvocation** each with it is advantages and disadvantages.

For instance, on a periodical rebinding proxy, the proxy is initialized with a domain, a selection strategy (what version do we want) and a rebinding interval. The proxy then periodically queries the VRESCo registry according to the rebinding interval, and checks if the current binding is still valid. If there is a new service which better matches according to the selection strategy, the proxy discards the current binding and constructs a new one.

Although — as the authors state — this tool is not fully implemented, VRESCo provides a really interesting approach of using a proxy to route the client calls trough the selected version. Not only that but it allows the developer to evolve its code base and the environment will then automatic update the tags based on the changes.

2.1.4 WSDLDiff

WSDLDiff is a tool to extract fine-grained changes from subsequent versions of a web service interface defined in WSDL [22]. It extracts WSDL elements affected by changes in subsequent versions. This tool is based on Eclipse Modeling Framework (EMF) [9], which is the framework that will do the heavy lifting of parsing the WSDL files and matching them together.

The process for extracting the fine-grained changes between two versions of a WSDL interface consists of four stages:

Stage A
: Parses the WSDL into EMF Models. Two EMF Models are created each corresponding to the WSDL definitions from the two versions.

Stage B
: Transforms each of the EMF models into the XSD contained by the WSDL. This step will help to improve the extraction process of the changes.

Stage C
: By using EMF, this stage will match both versions to detect which nodes exist on both versions.

Stage D
: The match model generated on the above stage is then analyzed to detect the differences among the two WSDL models. It outputs a tree of structural changes in terms of additions, removals, moves, and modifications.

The paper then presented a study on some well known WSDLs available (like Amazon EC2) in order to understand how they evolve over time. That way, a subscriber to those WSDLs could then predict which operations are more likely to change over time, providing a safeguard to the developer. WSDLdiff is still in early development, and although the tool looks solid and useful, there is still a lot to be done in order to become something to consider for future developers. This still requires a lot of work by the developer, and the goal of service evolution is leaning towards automating that process.

2.1.5 hRESTS

hRESTS is a microformat for machine-readable descriptions of RESTful Web APIs, backed by a simple service model [16]. It describes the main aspects of services—operations, inputs, and outputsḣRESTS translates HTML hierarchy into a hierarchy of objects and properties. By doing that, we can make the crucial parts of existing Web API documentation machine-readable, making it possible the auto-generation of client code, based on the gathered information.

Since usually Web API documentation is available in the form of HTML pages, it makes sense that a machine should be able to parse that information. hRESTS is then used to provide that information machine-readable by parsing specific classes on the documentation HTML.

Listing 2.2: hRESTS annotations

```
<div class="service" id="svc">
  <p>Description of the
    <span class="label">ACME Hotels</span>
    service:
  </p>
  <div class="operation" id="op1">
  <p>
    The operation <code class="label">getHotelDetails</code> is
invoked using the method <span class="method">GET</span>
at <code class="address">http://example.com/h/{id}</code>,
with <span class="input">the ID of the particular hotel replacing
the parameter <code>id</code></span>.
It returns <span class="output">the hotel details in an
<code>ex:hotelInformation</code> document.</span>
</p></div></div>
```

The example above is a Web API description annotated with hRESTS. To give further explanation, we will break down each of the definitions.

service class Indicates that the following HTML block is an hRESTS element, containing a Web service or API description.

operation class is used to indicate that the following block is a description of a Web service or API operation, meaning that it will have an **address** and **method** classes.

address class specifies the URI of the operation and can be used on a textual element (the URI is the content) or on an anchor (the URI is the target).

method class if used on a textual element (<span> for example), and represents the HTTP verb used on the operation (`GET POST PUT` ...)

input and output classes are used to identify the operation's input and output. While this provides useful information, it is not translated as a machine-readable information.

label class is used to specify a human-readable label for a service, operation or message.

Although this requires an extra effort from the developer, hRESTS could be used to generate changelogs on the different versions and spot breaking API changes on the endpoints. Moving forward, providing machine-readable specification could mean automation on some tasks that right now requires human interaction, such as migrating to new API versions.

2.1.6 RESTdesc

RESTdesc is another specification for describing RESTful Web APIs where they are described using pre and post conditions [27].

RESTdesc uses the semantic web [28] as the base of its solution. Using RESTdesc we can create a descriptor using Notation3 [26], which is a language based on the core language for Semantic Web. The main elements of this descriptor are **preconditions**, which indicate the state a certain resource should have before the interaction, **postconditions**, which describe the new state of the resource, and **request details**, which explain which HTTP request should be made.

The syntax itself it is self-explanatory but can be hard to understand at first so let us take a look at a simple example. Imagine an API where, by going to `/photos/:id` we get the respective photo. In other, words, lets say that *I can retrieve a photo by going to /photos/ and appending its identifier*. Rephrasing it again we might say that for all the photos with a given id if we do a `GET` request to `/photos/:id/` we receive the respective photo. And that is how we represent this operation using RESTdesc, which can be seen in the example below.

Listing 2.3: RESTdesc example

```
@forAll :photo, :id.
@forSome :request, :response.
{
  :photo :photoId :id.
}
log:implies
{
  :request http:methodName "GET";
           tmpl:requestURI ("/photos/" ?photoId);
           http:resp :response
  :response tmpl:represents :photo.
}.
```

By using this composition, we can create a powerful description not possible with the current languages. We start by stating all the quantifiers, and then with the implication, we are saying that it exists a request and response for the photoId if the there is a photoId that relates the photo that we want with its id (because not all photographs in the world have an id, only the ones in the database). There is a lot of power in the language that can be used to create much more powerful descriptions, as well as web service discovery although that goes beyond the scope of our work.

When it comes to change and evolution, this focus on the runtime aspect makes RESTdesc adapted to changes. If the client makes a request and that request does not meet the preconditions, the server will return an error, and it might even give hints to the client on how to solve it. the central idea is "Given a certain input, how can the service descriptions reach my predefined goal?". And it is a very interesting view that contrasts with the traditional static approach.

While there might be useful applications, RESTdesc is the description language that requires the most work from the developer. It also uses a language based on the core concepts of web semantics that it is still different from what we would normally use. It is still unsure whether it is possible to automate the generation of client code, although, since the principles are different from the multiple descriptions languages available, there might be some interesting for it, especially with client code trying to cope with changes.

2.1.7 restify

Restify is a node.js framework used to create rest APIs. It is not the most popular node.js framework, but it is perhaps the best to introduce in this section, as it supports versioned routes out-of-the-box. As we will see throughout this document, the restify implementation has the same basis as ours. The server object has a method for every HTTP verb, assigning a specific function when a request is made to a specific route of a specific version.

As for the routing into the correct version, versioned HTTP requests are made by specifying the desired version on the header of the request with the flag accept-version. This flag also accepts wildcards, meaning that for instance 1, will give the newest version of the relation that has the type 1.x.x. This will ensure if semantic versioning is respected, that the client will receive the newest version from major 1 and that that version will not have any breaking change.

Listing 2.4: restify example

```
var restify = require('restify');

var server = restify.createServer();

function sendV1(req, res, next) {
  res.send('hello: ' + req.params.name);
  return next();
}

function sendV2(req, res, next) {
  res.send({hello: req.params.name});
  return next();
}

var PATH = '/hello/:name';
server.get({path: PATH, version: '1.1.3'}, sendV1);
server.get({path: PATH, version: '2.0.0'}, sendV2);

server.listen(8080);
```

As for its limitations when compared to our solution, we can mention a few. Although their decision of utilizing semantic versioning can be justified because that is the versioning system adopted by the node.js ecosystem, it limits the solution, as others versioning systems cannot be used.

Programmers are not enforced on respecting the versioning system's rules, and as such, compatibility between versions is compromised. It is, however, a step in the right direction when it comes to API versioning.

2.2 Choice Calculus

As we said earlier in **Chapter** 1, variability is currently an issue in programming. Choice calculus [6] is a language that aims to solve that issue. By using dimensions and choices, choice calculus abstracts all the variability. With that, we can reason on variability, which would be harder to do using a preprocessor like CPP [8].

2.2.1 Syntax

Choice calculus syntax presented in **Figure** 2.2 is very simple and minimal, providing only the fundamental constructions to add variability to a program.

Figure 2.2: Choice calculus syntax

$e ::= a \prec e,\ldots,e \succ$	Object Structure
$\mid \textbf{Dim}\ D\langle t,\ldots,t\rangle\ \textbf{in}\ e$	Dimension
$\mid D\langle e,\ldots,e\rangle$	Choice
$\mid \textbf{share}\ v = e\ \textbf{in}\ e$	Sharing
$\mid v$	Reference

Starting with **object structure**, it is what makes choice calculus a generic language, in the sense that it can use any programming language whose programs can be represented using a tree structure. The object structure is not more than a tree representing the program in which we want to insert variability. For instance, imagine that we have a function that sums two values. We will use JavaScript syntax to simplifying.

```
function add(a, b) {
    return a + b;
}
```

If we were to use choice calculus, we would need to convert this function abstract syntax tree (AST) to an object structure, which would be straightforward as it would only be necessary to grab the previous AST and wrap it around Y-brackets. Doing that, we could have the following structure:

```
function≺block≺return≺+≺a,b≻≻≻≻
```

The variability is added with **dimension** and **choice** syntax. Choice calculus uses the notation of dimensions and we can imagine the multiple dimensions being the different versions of the program. For instance, if we have a program that compiles to 32 and 64 bits architecture, we might have one architecture dimension with two possibilities–32 and 64, respectively. Using choice calculus, such dimension could be created using the following syntax:

```
Dim Arch<32,64> in ...
```

With that said, the other nuclear component of choice calculus is to decide what implications those dimensions have. Using the same example, we use the **choice** syntax to specify what happens if we are on the alternative 32 or 64 of the Operating System (OS) dimension. We know that an integer has a length of 4 bytes on 32 bits Operating Systems and 8 bytes on a 64 bits OS. If we were to allocate the corresponding memory we could use the following:

```
Dim Arch<32,64> in
  malloc(Arch<4,8>)
```

This is the core of the choice calculus syntax, but by being such a minimal language, choice calculus allows extensions to its syntax. One of those extensions is **sharing** and **reference**. Sharing and reference work similar to a variable in a programming language, but it gets removed once we resolve the variability and remove all the constructions. We use share to create a temporary value that can be used to avoid repeating choices on our dimensions. One way to use sharing and reference is to prevent code repetition, and consequently, to make our program less prone to error.

If we want to support on our program multiple operating systems—Windows, Linux and OS X, we might create one dimension with those three alternatives. Although they are quite different, OS X and Linux share similar features, for example, line endings. Windows uses `\r\n` while Unix-based operating systems like Linux and OS X uses `\n`. We might want to create a variable line ending that will have the current line ending depending on the Operating system, and it might even throw an error if we are trying to compile this program on an unknown OSUsing choice calculus, we would do that as follows:

Listing 2.5: Choice calculus example

```
dim OS<Linux,Mac,Windows,Unknown> in
  share v=newline = "\n"; in
    OS<v,v,newline = "\r\n";,new Error("Unknown␣OS!");>
```

This way we avoid repeating `newline = "\n";` multiple times and when we look at the code generated by a specific choice, it remains simple and easy to read, as it should.

2.2.2 Choice Elimination

In order to remove all the variability from our code for it to compile, we must eliminate all the constructs of choice calculus. This is achieved through **choice elimination**. We will just cover some concepts of choice elimination since our solution is not supposed to behave like a preprocessor, but it is important to understand some inner workings of choice elimination. The first thing we should do is to delete all the **share** and replace the references to the corresponding values. Sharing is just a way to avoid code repetition, so it is logical that we deal with that first. Since sharing is scoped and we are working on an object structure that behaves like a tree, this is just finding the share, and then replace the value on all the references until the end of the tree or until we find another share with the same name, because if we proceed we will violate its scope.

Although we have not mentioned the select syntax, there is a paper focused on that [7]. We will not explore it because although one might want to insert it into the code base, one might also want to do it when compiling, as a command line argument, or even just compile all the possible combinations. But let us imagine that for the example mentioned in **Listing** 2.5, we want to compile the Linux version. We already got rid of the share, so we proceed by going to line 1 and remove that line from the object structure, and then go down the object; if we find the OS dimension, we remove it and add the code for that choice. However, if we find another choice from a different dimension, we need to keep the search for **all** the alternatives. Once we removed all the dimensions and choices, the result is a program without variability and ready to be compiled.

2.2.3 Type System

Choice calculus has also its own type system. For once, we must ensure that the configuration of our program is well made, by checking if we are not inserting choices outside its dimension scope. It is also necessary to check if every choice has the right amount of possibilities. The goal is to check which decisions must still be made in order to resolve the variational expression into a plain variant in the object language—the **configuration status**.

The configuration status of a choice calculus expression is captured by a judgment of the form $\Gamma \vdash e : \Delta$ which means that expression e has a configuration type Δ on environment Γ. The configuration type only has two distinct types. It has the type Φ if the expression is fully configured—all decisions were made, and has the type $D\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n\rangle;\Delta$, which represents two different things:

$D\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n\rangle$ represents the dimension D with alternatives $t_1, \ldots, t_n$ and dimensions $\Delta_1, \ldots, \Delta_n$ dependent on the selection on dimension D.

Δ represents the subsequent dimensions independent of dimension D.

As an example, the expression

$\prec$**dim** $A\langle a_1, a_2\rangle$ **in** $A\langle 1,2\rangle\succ$

would have the configuration type

$A\langle a_1 \Rightarrow \Phi, a_2 \Rightarrow \Phi\rangle;\Phi$

However, the expression

$\prec$`dim` $A\langle a_1,a_2\rangle$ `in` $A\langle 1,$ `dim` $B\langle b_1,b_2\rangle$ `in` $B\langle 2,3\rangle\rangle,\ C\langle c_1,c_2,\rangle\ \langle 4,5\rangle\succ$

would have the configuration type

$A\langle a_1,a_2 \Rightarrow B\langle b_1,b_2\rangle\rangle;C\langle c_1,c_2\rangle$

which reveals the relationships between A, B, and C.

To check if the dimensions are well formed, we resort to the environment. The environment Γ has two kinds of bindings. The first is the standard mapping from variables to configuration types, the second maps dimensions names to a pair of integers used to support the typing of choices. This is done in the form of $D : (n,i)$, which means that a choice in dimension D must have exactly n alternatives, and the ith alternative is considered to have been selected.

Finally, we need to address the type system for the actual program with variability. What happens when we are using a preprocessor is that type checking is only executed after the preprocessor generated the code. On software product lines, however, since we can have a large number of different versions, type checking each one individually is not a viable solution. Research have been made to find alternatives and a type system for variational lambda calculus [5] might be something to consider. Variational lambda calculus is choice calculus instantiated by the object language of the lambda calculus. For our solution, we believe that a brute force approach that will check every possibility will suffice. However, we might need to take into consideration variatonal lambda calculus if we are building something to be used in a context where this approach is not practicable.

Although our solution does not go specific into choice calculus, it is still a good starting point and a simple way to deal with variability. Because choice calculus serves as a foundation for further development, we can reuse many of their concepts and techniques into our solution, like the concept of dimensions and their philosophy of keeping the variability simple and isolated from the rest of the code. One thing that still needs to be addressed is how to reason about such programs. Choice calculus can become complex, and a visualization tool, that for instance could create separate views for the different dimension's alternatives, might become useful to help the development of variational software.

2.3 Software Product Lines

As defined by Software Engineering Institute [13], "a software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." Although software product lines focus on some areas that are beyond the scope of this work, it provides some techniques that can help us deal with variability in a responsible way, abstracting most of the unnecessary details. The best example is feature models.

Feature Models

Feature Models are a way to represent software functionality by listing its features [15]. In software product lines, multiple variations of the same software are produced, and feature models are used to specifying all those possible combinations of the product line.

Feature models are a good way to reason on variability on a higher level. Feature diagrams (a graphical representation of feature models) can be used to quickly overview all the different features and versions of the software on the product line. We must not forget that eventually, all the variability represented on the feature models needs to be in the system but, as we mentioned earlier, it can be hard to move into a new code base and start evolving it, especially if said code base is full of variability.

Feature diagrams are the graphical representation of feature models and widely used in software engineering. It represents features on a tree where leaves are primitive features and interior nodes that represent compound features. Feature diagrams notations revolve around stating optional and mandatory features, as well as alternative features. **Figure** 2.3 represents a simple feature diagram that yields some of the following configurations.

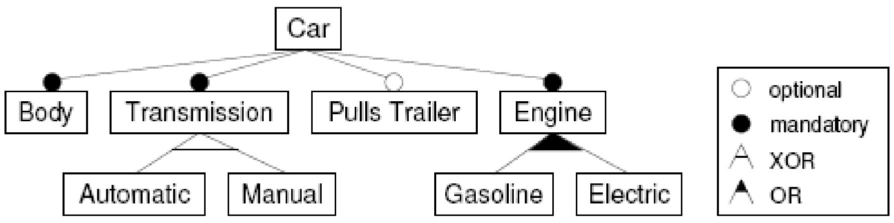

Figure 2.3: Feature Diagram

1. `Body, Transmission Automatic, Trailer, Engine Gasoline`

2. `Body, Transmission Automatic, Engine Gasoline Electric`

3. `Body, Transmission Automatic, Engine Electric`

4. ...

Feature models are an established technique in the world of software engineering and variability that we need to take into account when building something that relates to the topic. Although feature diagrams might not relate directly to our solution, it is still important to understand how other researchers tackled the variability problem, in particular with such a notable technique widely used by companies such as Boeing or Siemens.[12]

As for our work in particular, although we are not aiming to build feature models, we found out that we can adapt the theory behind it to our specific use case, as different configurations can be seen as the multiple software versions. Having the possibility to abstract variability (or in this specific case, versions) helped us building our system that deals with multiple versions.

Chapter 3

Typesafe Evolution of Web Services

In this Chapter, we present our solution to address the problems discussed before. This solution consists of a programming model that we present below, starting with our decisions on the design phase. We first present the principles (**Section** 3.1), that is, every requirement that our solution needs to satisfy. We then introduce relations, which are an important abstraction to versioning systems (**Section** 3.2). We then present a programming language with a built-in version control, to serve as a base of work and help us reason about all the different concepts, (**Section** 3.3).

3.1 Programming Model Principles

Since the focus of this thesis is web service evolution, it is really important to analyze multiple public APIs and decide on what improvements could be made. Most of the available APIs only provide the most updated version, and that can cause multiple issues on the clients. They often do this to avoid maintaining multiple versions at the same time, which takes more time. A solution that could deploy multiple versions at the same time would only be an improvement if it would not increase the maintenance required to keep the service available.

When evolving an API, developers often treat them like software libraries with multiple versions, being two completely separated pieces of software. In reality, we can visualize a web service as building blocks, where to evolve to the next version, we reutilize the older versions already created. This approach allows to treat versions as an integrated part of the development process, providing a much clearer way to reason on web service evolution. Next we introduce a set of principles we want our solution to comply.

1. It must allow the execution of multiple versions of the same service. In order to

maintain older versions working, having multiple versions deployed is a must-have.

2. It must ensure the type safety between versions. In order to prevent execution errors that might occur, we need to ensure that all versions are well typed. In that way we can prevent type errors.

3. It must ensure safe and automatic updates. Clients requesting older versions might receive an updated and compatible version if it exists. Since we are already allowing multiple versions to be deployed, it is only logical to update clients whenever it is possible.

4. It must be parametric in the versioning system. Multiple versioning systems have been adopted to solve different problems. To be stuck with one would jeopardize our solution since it would not work on a broader spectrum.

3.2 Version Relations

We created relations to work as an abstraction of versioning systems. In order for our system to support as many versioning systems as possible, relations abstract and normalize them.

Relations represent the versioning system as a tree. We made this decision because there are already many popular versioning systems based on trees and we think it represents evolution better than a graph. A graph, being a superset of a tree, would also give us the benefits of a tree, but with more unnecessary complexity that would cause much more ambiguity when resolving the automatic updates. This relation must be specified by the programmer. The option of selecting a default versioning system, like semantic versioning [21] was discussed, although eventually that idea was droped as it would constrain our solution.

Each node of the tree represents a single version and must contain a unique identifier. Branches represent a direct relation between two versions with a specific degree of compatibility. The degree of compatibility will be used by the system to type check versions and to give the client the best version. We defined three compatibility modes: **Strict**, **Subtyping** and **Free**. Free mode means that between versions, identifiers that exist in both of them may have different types. On the other two modes, the rules are as follows. Let R be a relation, V_1, V_2 be versions, id_1, id_2 be identifiers, where $a > b$ means that version a is bigger/newer than version b and τ_x being the type of identifier x:

$$\forall(V_1, V_2) \in R_{strict} \wedge V_1 > V_2, \forall id_1 \in V_1 \wedge id_2 \in V_2, id_1 = id_2 \implies \tau_{id_1} = \tau_{id_2}.$$

$$\forall(V_1, V_2) \in R_{subtype} \wedge V_1 > V_2, \forall id_1 \in V_1 \wedge id_2 \in V_2, id_1 = id_2 \implies \tau_{id1} \leq \tau_{id2}.$$

Figure 3.1 is an example of a relation. It uses the format x.y to represent a version, with x and y being positive integers. For the compatibility mode, it uses subtyping between y versions and free between x. **Figure** 3.2 uses the same relation, but creates different variations on the same version, representing the software evolving in two different directions.

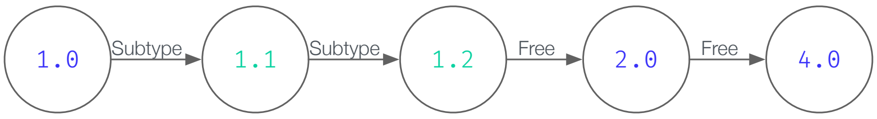

Figure 3.1: Example of a simple relation

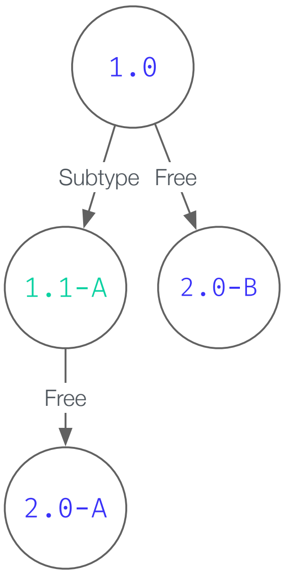

Figure 3.2: Example of a more intricate relation

Further explanation on how the system uses these relations will be given in the next section, where we introduce a language that has a built-in version control system.

3.3 Versioned Featherweight Java

Featherweight Java [11] is a minimal core calculus created to model Java's type system. It focus on minimalism and as such, it is similar to an object calculus. It serves as a starting point to our solution, in which we extend Featherweight Java to support versions. In that way we can illustrate how a programming language with a built-in version control system would work, and demonstrate how to build a typed and versioned dispatcher of methods. Featherwight Java does not maintain state, making classes immutable. Assuming the classes A and B on **Listing** 3.1, the expression $@1(\texttt{new}\ A(\texttt{new}\ B()).elem)$ will be reduced into $\texttt{new}\ B()$, returning the instance of the class B created in version 1. Further explanation ofthis language syntax, semantics and its type system will be given on the following sections.

Listing 3.1: Example of two classes defined using Featherweight Java

```
class A {
  B elem;
  A@1(B elem) { this.elem = elem}
}
class B extends Object {
  B@1() { }
}
```

It is important to note that, altough our solution mentions subtyping, we discarded it from Featherweight Java. That formalization will make the rules overly complex and as such it is subject for future work.

3.3.1 Syntax

$$
\begin{aligned}
\mathsf{L} &::= \texttt{class}\ C\ \{\overline{C\ f\ v};\overline{K}\ \overline{M}\} \\
\mathsf{K} &::= C@v(\overline{C\ f})\{\overline{\texttt{this}.f = f;}\} \\
\mathsf{M} &::= C\ m@v(\overline{C\ x})\{\texttt{return}\ e;\} \\
e &::= x \mid e.f \mid e.m(\overline{e}) \mid \mathsf{new}\ C(\overline{e}) \mid \mathsf{V}(e) \\
\mathsf{V} &::= @v \mid @@v \mid @!v
\end{aligned}
$$

Figure 3.3: Syntax of versioned Featherweight Java

The syntax of the language is presented in **Figure** 3.3. We have class declarations ($\texttt{class}\ C\ \{\overline{C\ f\ v};\overline{K}\ \overline{M}\}$), that have a set of fields each with a type and a version ($\overline{C\ f\ v}$). Classes also have a set of constructors for different versions ($\overline{K}$) and a set of methods ($\overline{M}$). Constructors ($C@v(\overline{C\ f})\{\overline{\texttt{this}.f = f;}\}$) are declared as a normal java constructor, but with the version v appended to the name. It receives a set of variables that are initialized as instance variables. Methods ($C\ m@v(\overline{C\ x})\{\texttt{return}\ e;\}$) are also defined on a specific version and its body only contains a return keyword followed by an expression. The mandatory `return` is due to the fact that classes are immutable, and a classe state is changed by returning a new instance of it, with the new state. Expressions can be identifiers, acessing fields ($e.f$), calling methods ($e.m(\overline{e})$) or constructors ($\mathsf{new}\ C(\overline{e})$). The syntax also includes an expression `V(e)` that for a given expression e evaluates in version v with a given compatibility mode `V`. The version modifier `V` includes three alternatives: the $@v$ mode (also called *strict* mode), that allows the execution to choose the most recent available definition of constructors, methods and fields with the same types, $@@v$ (also called *free* mode) to indicate that the evaluation fetches the most recent available version independently of the types, and @!v (also called *exact* mode) to indicate that the expression will be evaluated exactly in version v.

In our solution, a (partial) order relation between versions must be defined (e.g. version 2.0 is greater than 1.0). This can be achieved with version relations, as we explained of section 3.2.

3.3.2 Type System

The typing of a set of class declarations is defined by the typing rules presented in **Figure** 3.5, based on several lookup functions defined in **Figure** 3.4. We define a version lookup function ($\mathtt{version}(\overline{v}, v, t)$) that computes the most recent version, given a set of available versions ($\overline{v}$), a starting version (V), and a search mode (t). When using the exact version mode (@!) the function definition is straightforward because it is either defined with the desired version v or it is undefined, and every rule using it, fails. In the free mode (@@) the function denotes the most recent version available, with no other restrictions. In strict mode (@) the function searches the most recent version available if there is a path in the tree formed by the order relation to that version using only strict relations. Notice that the versioning relation must be transitive in exact and strict modes.

Getting the fields of a class on a specific version is given by function $\mathtt{fields}(C, v, t)$, with C the name of the class, v the desired version and t the search mode. The function definition works by selecting, for each identifier $g_i \in \overline{g}$, a set of available versions $\overline{u_i}$, where $\mathtt{version}(\overline{u_i}, v, t)$ denotes the adequate version among the available ones for identifier g_i, according to the version search mode t starting in version v. Using the same approach, we define function $\mathtt{mtype}(m, C, v, t)$ to yield the type of a method on a specific version, and $\mathtt{mbody}(m, C, v, t)$ to yield the body of a method as its result.

The typing rule for classes (T-Class) follows a partitioning method for the declarations by dividing them in strictly similar sets of versions of method declarations, and for all versions v_i. The typing judgement for expressions is tagged with a version modifier, the actual base version, and the search mode. All typing rules are quite similar to the original rules in [11], with the extension of recording a current search mode and starting version, that is used to parametrise the auxiliary functions. The base case for setting the context version is rule (T-Context).

3.3.3 Operational semantics

The operational semantics for the language is defined in **Figure** 3.6 by means of a reduction relation, defined with a relation to a current version and version search mode like $e \xrightarrow{v,t} e'$. Rule (R-Constr) ensures that the correct constructor is returned on the lookup method. The same happens with (R-Field) and (R-Invk), with the former being the getting of a specific field from a class on a specific context and the latter the selection of the appropriate method to execute, again using a specific current version and search mode and a lookup function. These two rules are different than the ones on the original calculus, since there are multiple fields with the same name (one for each version) and there is a possibility for multiple methods with the same name as well (but with different versions), hence why the “current” version is so important in these situations. The remainder of the rules are quite similar to the original semantics, with the exception of (RC-Ctxt) which is a new rule for this extension. It works as a simple reduction where it is just necessary to change the search mode and the currect version, executing the expression on that specific context. The operational semantics is done in sync with the typing relation.

$$\texttt{version}(\overline{v}, v, t) = \begin{cases} v' \text{ if } t = @,\ v \simeq v' \text{ and } \forall_{v''\in\overline{v}}.\ v'' \simeq v' \\ v' \text{ if } t = @@,\ v \lesssim v' \text{ and } \forall_{v''\in\overline{v}}.\ v'' \lesssim v' \\ v' \text{ if } t = @!,\ v' \in \overline{v} \text{ and } v = v' \end{cases}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \{\overline{D\ g\ u};\overline{K}\ \overline{M}\} \quad \overline{F_i} = \overline{D_i}\ g_i\ \overline{u_i} \quad i = 1..n \\ F'_i = D_{ij}\ g_i\ u_{ij} \quad \texttt{version}(\overline{u_i}, v, t) = u_{ij}\end{array}}{\texttt{fields}(C, v, t) = \overline{F'}}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \{\overline{F};\overline{K}\ \overline{M}\} \quad M'_i = B\ m@u_i(\overline{B\ x})\{\ldots\} \in \overline{M} \quad i = 1..n \\ B'\ m@u_{ij}(\overline{B'\ x})\{\ldots\} \in \overline{M'} \quad \texttt{version}(\overline{u_i}, v, t) = u_{ij}\end{array}}{\texttt{mtype}(m, C, v, t) = \overline{B'} \to B'}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \{\overline{F};\overline{K}\ \overline{M}\} \\ M_i = B\ m@u_i(\overline{B\ x})\{\texttt{return e;}\} \in \overline{M} \quad i = 1..n \\ B\ m@u_{ij}(\overline{B\ x'})\{\texttt{return e;}\} \in \overline{M} \quad \texttt{version}(\overline{u_i}, v, t) = u_{ij}\end{array}}{\texttt{mbody}(m, C, v, t) = \overline{x'}.e}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \{\overline{F};\overline{K}\ \overline{M}\} \\ K_i = C@u_i(\overline{D\ f})\{\overline{\texttt{this}\ .f = f;}\} \in \overline{K} \quad i = 1..n \\ C@u_{ij}(\overline{D\ f}')\{\overline{\texttt{this}\ .f = f;}\} \in \overline{K} \quad \texttt{version}(\overline{u_i}, v, t) = u_{ij}\end{array}}{\texttt{ctype}(C, v, t) = \overline{D\ f}' \to C}$$

$$\frac{\begin{array}{c}\texttt{class}\ C\ \{\overline{F};\overline{K}\ \overline{M}\} \\ K'_i = C@u_i(\overline{C f v})\{\overline{\texttt{this}.f = f;}\} \in \overline{K} \quad i = 1..n \\ K'_{ij} = C@u_{ij}(\overline{C f v})\{\overline{\texttt{this}.f = f;}\} \in \overline{K'} \quad \texttt{version}(\overline{u_i}, v, t) = u_{ij}\end{array}}{\texttt{constr}(C, v, t) = K'_{ij}}$$

$$\texttt{partition}(\overline{M}) = \{(\overline{M'_i}, v_i) \mid v_i \in \overline{v}, \overline{M'_i} = \{(m\ v') \in \overline{M} \ \wedge\ v' \simeq v_i\}\}$$

Figure 3.4: Lookup functions

$$\frac{\begin{array}{c}\mathtt{mtype}(M'_{i1}) = \overline{B_i} \rightarrow B_j \quad i = 1,..n \quad j = 1,..m_i \\ \mathtt{partition}(\overline{M}) = \overline{(\overline{M'},v)} \quad \Gamma \vdash_C M'_{ij} : \overline{B_i} \rightarrow B_j\end{array}}{\Gamma \vdash \mathtt{class}\ C\ \{\overline{M}\} : C} \text{(T-Class)}$$

$$\frac{\begin{array}{c}\mathtt{ctype}(K'_{i1}) = \overline{C\ f}_i \rightarrow C \quad i = 1,..n \quad j = 1,..m_i \\ \mathtt{partition}(\overline{K}) = \overline{(\overline{K'},v)} \quad \Gamma \vdash_C K'_{ij} : \overline{C\ f}_i \rightarrow C\end{array}}{\Gamma \vdash \mathtt{class}\ C\ \{\overline{K}\} : C} \text{(T-Constr)}$$

$$\frac{\mathtt{fields}(D,v,t) = \overline{D\ f\ v'} \quad \Gamma, \overline{f : D}, \overline{x : C} \vdash_{v,@} e : C}{\Gamma \vdash_D C\ m@v(\overline{C\ x})\{\mathtt{return}\ e;\} : \overline{C} \rightarrow C} \quad \text{(T-Method)}$$

$$\Gamma \vdash_{\_}\ x : \Gamma(x) \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash_{v,t} e_0 : C_0 \quad \mathtt{fields}(C_0,v,t) = \overline{C\ f\ v'}}{\Gamma \vdash_{v,t} e_0.f_i : c_i} \quad \text{(T-Field)}$$

$$\frac{\Gamma \vdash_{v,t} e_0 : C_0 \quad \mathtt{mtype}(m, C_0, v, t) = \overline{D} \rightarrow C \quad \Gamma \vdash_{v,t} \overline{e} : \overline{D}}{\Gamma \vdash_{v,t} e_0.m(\overline{e}) : C} \quad \text{(T-Invk)}$$

$$\frac{\mathtt{fields}(C,v,t) = \overline{D\ f\ v} \quad \Gamma \vdash_{v,t} \overline{e} : \overline{D}}{\Gamma \vdash_{v,t} \mathtt{new}\ C(\overline{e}) : C} \quad \text{(T-New)}$$

$$\frac{\Gamma \vdash_{v,t} e : C}{\Gamma \vdash_{\_}\ tv(e) : C} \quad \text{(T-Context)}$$

Figure 3.5: Typing rules

$$\frac{\texttt{constr}(C,v,t) = \texttt{new}\ K(\overline{C f v})\{\ldots\}}{\texttt{new}\ C(\overline{e}) \xrightarrow{v,t} \texttt{new}\ K(\overline{e})}\text{(R-Constr)}$$

$$\frac{\texttt{fields}(C,v,t) = \overline{C\ e\ v}}{(\texttt{new}\ C(\overline{e})).f_i \xrightarrow{v,t} e_i}\text{(R-Field)}$$

$$\frac{\texttt{mbody}(m,C,v,t) = \overline{x}.e_0}{(\texttt{new}\ C(\overline{e})).m(\overline{d}) \xrightarrow{v,t} [\overline{d}/\overline{x},\texttt{new}\ C(\overline{e})/\texttt{this}]e_0}\text{(R-Invk)}$$

$$\frac{e_0 \xrightarrow{v,t} e_0'}{e_0.f \xrightarrow{v,t} e_0'.f}\text{(RC-Field)}$$

$$\frac{e_0 \xrightarrow{v,t} e_0'}{e_0.m(\overline{e}) \xrightarrow{v,t} e_0'.m(\overline{e})}\text{(RC-Invk-Rec)}$$

$$\frac{e_i \xrightarrow{v,t} e_i'}{e_0.m(\ldots,e_i,\ldots) \xrightarrow{v,t} e_0'.m(\ldots,e_i',\ldots)}\text{(RC-Invk-Arg)}$$

$$\frac{e_i \xrightarrow{v,t} e_i'}{\texttt{new}\ e_0(\ldots,e_i,\ldots) \xrightarrow{v,t} \texttt{new}\ e_0(\ldots,e_i',\ldots)}\text{(RC-New-Arg)}$$

$$\frac{e_0 \xrightarrow{v,t} e_0'}{tv(e_0) \xrightarrow{-} tv(e_0')}\text{(RC-Ctxt)}$$

Figure 3.6: Operational Semantics

Chapter 4

Prototype

We created a prototype of our system named niVerso – an anagram for Version – to help the understanding of the system as a whole with all of its components. Its main purpose is to allow developers to create an API that can be updated on a safe and sustained way, without introducing breaking changes to API clients. In this chapter, we start by introducing our technology choices and the reason behind those choices. In **Section** 4.2 we go into detail on the implementation, describing the components needed, as well as the thought process behind their implementation.

4.1 Technologies

As for the technologies, we decided to use JavaScript, as it is the *de facto* language of the web. Since JavaScript uses dynamic type checking and our version type checker is designed to be static, we need to make some modifications in order to implement our type system.

Seeing that there are already multiple static type checkers available, it is better to use one and extend it to support our needs. The two most popular choices are Flow [4] and TypeScript [20]. While both work as a static type checker, TypeScript is also a superset of JavaScript, with multiple features that we do not need for our prototype. In the end, the better solution for our prototype is Flow. Flow uses method signatures and type annotations which is exactly what we need to verify the versions. Although the ideal solution is to extend Flow to support our use case, given the time constraints and the complex nature of Flow, the decision was to create a separated version checker that would run before Flow, and then let Flow check the rest of the code.

Inserting versions in JavaScript can also be a challenge. There is not a clean syntax to do that and JavaScript does not allow multiple methods with the same name. The solution is to modify the AST and admit different names for the same method. Listing 4.1 shows a naive solution to this problem. Multiple issues arise with this solution, since it will only provide basic mapping and does not fully implement the context version. In order to implement context versions in JavaScript, we needed to have access to the AST. We use a popular AST parser, Babel [3] to create the AST and manipulate it. Babel is the only viable choice since it can parse Flow annotations. However, we plan to have a custom syntax to create versions just like on our model language. That is not possible because then Babel is not able to parse and create the AST. The only way to do it is to have valid JavaScript when creating version contexts. Our initial proposal was `Version 1 { }`, but since this is not possible, we decided on this syntax: `(version="1") => { }`. There is not much difference between both syntaxes, but the latter is valid JavaScript that can be parsed by Babel. A more elegant solution would be to create hygienic macros that would transform the first syntax into the second, but we believe that for a prototype, our approach will suffice.

As for the requests, we also need a web framework where we can implement our rerouting system on top of it. The most popular frameworks are Hapi [10] and Express [25]. Since Express works through middlewares, our implementation could be very simple if we make use of that, and that is why we decided to use Express.

Listing 4.1: Example of a naive code transformer

```
//Original
version 1 {
  var a = 1;
}

version 2 {
  var b = a + 1;
}

//Transformed
var a__1 = 1;
var b__2 = a__2 + 1; // Reference Error
```

4.2 Implementation

With all the technologies choosed, we built the following components:

1. Version Contexts, to provide the same features observed on our Featherweight Java extension.

2. Type System, to type check the versions

3. Framework router, to reply the client requests with the best possible version

While not being a component by itself, the **relation** must also be mentioned. It must be provided by the programmer of the web service, and must comply with the specifications listed on **Section** 4.2.4.

4.2.1 Version Context

Supporting different contexts on JavaScript will work like a compiler since it is not possible to just replace the identifiers with the context they are in (as we can see in **Listing** 4.1). The versions, however, need a more refined treatment. **Listing** 4.2 shows a basic example of an identifier with three different versions. However, there are two distinct version contexts that will work independently. The first inside the server, where versions are created, and the second making the bridge between the server and the client. The version context on the server side will be sorted out on compile time, switching all the version references into valid JavaScript but the second one needs to remain active to allow a proper response from the server when requested a specific version. To simplify the syntax inside the server the only search mode available is strict. This still allows some reusability but without overpopulating the syntax with constructions to change the search mode. In the communication between the client and the server, all the search modes are active since the mode is changed on the request header and does not require any additional syntax at all.

Listing 4.2: Example with valid JavaScript

```
(version = '1.0')   => { var a = 1  }
(version = '1.1-A') => { var a = 20 }
(version = '2.0-B') => { var a = "text" }
```

4.2.2 Type System

The Type System is responsible for ensuring the type safety of all versions. For our type system to work, we must have access to type signatures of identifiers defined inside contextes. The type system then tries to store every identifier and its type, and for every version that it encounters, it compares with the other existing versions, checking if the types for different version are well typed, according to the relation, and for every version that it encounters, it compares with the other existing versions, checking if the types for different versions are well typed, according to the relation.

If we were to add type annotations to the **Listing** 4.2, the type system would generate a structure similar to the one on **Figure** 4.1. The type system with that information can verify if every id present in multiple versions has a correct type according to the relation.

As for the subtypes in specific, our type system can only detect subtyping inside JavaScript objects. That is fundamental as we are building JSON APIs on our prototype. More intricate subtyping examples will not work in our system, although it is something that we consider for future work, as a complete integration with the Flow ecosystem would provide those possibilities. The type system iterates through the object and collects all the keys and its types, verifying if the greater version on the pair has the same or more fields on the object. If that is the case then the system validates it.

A clarification as for why strict and subtyping modes are included. Generally speaking on JSON APIs, the introduction of strict mode might seem overkill with subtyping mode being more appropriate since the same fields that can be accessed on an earlier version are still available on the new one. However, if the API client is iterating through the JSON, there might be some unexpected fields which might result in an error when using the updated version. For those cases, we introduced strict mode, as the developers might also need to verify that they do not give to the API clients more fields than the ones that are strictly necessary.

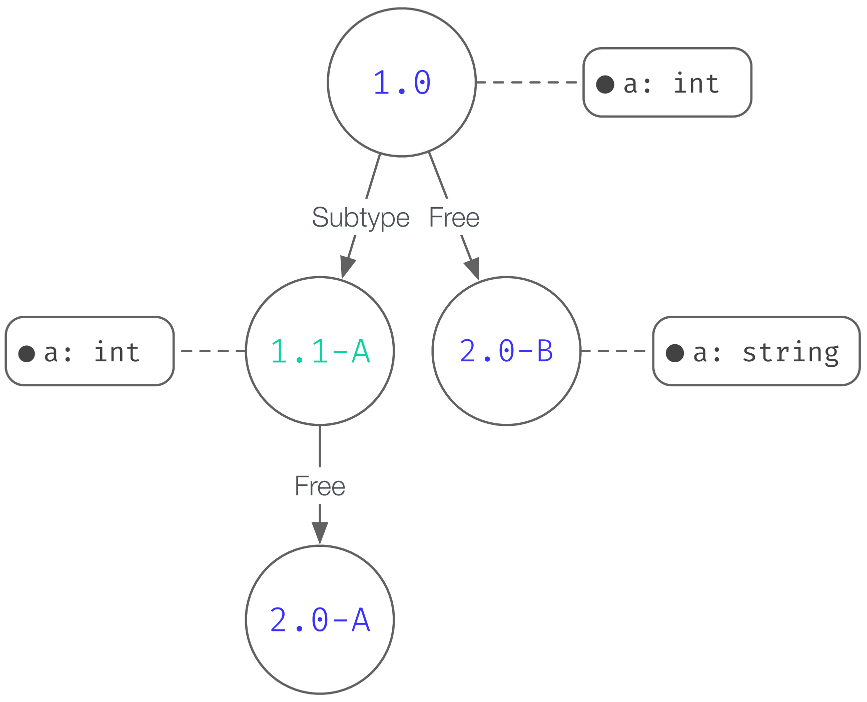

Figure 4.1: Example of the type system structure

4.2.3 Framework Router

The framework is responsible for rerouting the requests and make the client updates whenever needed. It was made on top of express [25], a minimal web application framework.

Express is used to create JSON APIs and works by running a method each time a specific route is called. This is a very direct approach that does not work on our system, since we might have different methods for different versions. The way our framework works is by creating a custom method for each route that will run the appropriate method, given the version. A simple route in express can be seen in **Listing** 4.3, and it means that for every GET request on endpoint `/users/`, the server will run that function that returns a string `Hello World!`. To work on multiple versions we just need to pass on which version does that route applies to, so we added the support for a version parameter to work just like on **Listing** 4.4. What the router does is storing every route, then for every request it runs the search algorithm, in order to reply with the best version possible. The way the communication works is through the HTTP protocol, and how the API specifies the desired version and compatibility mode is done through the header presented in the request. Each request made by the client should have two header fields: `X-Version` (mandatory) and `X-Mode` (optional). The first sets the base version from where the framework should search, and the second the level of compatibility that we are using.

X-Version expects a version that belongs to the relation. If that version is preceded by an !, the system will deliver that version or none at all.

X-Mode expects one of three existing modes representing three levels of compatibility explained in section 3.2

Listing 4.3: Express route

```
app.get('/users', (req, res) => { res.send('Hello World!'); });
```

Listing 4.4: niVerso route

```
app.get('1.0', '/users', (req, res) => { res.send('Hello World!'); });
```

By itself the router can reroute requests and update clients based on the relation given, but its limited because it cannot verify if versions are indeed well typed, and we would still have the usual maintainability problems that multiple versions entail. This is why the prototype has three components. If we combined them togheter we get complete functionality. **Listing** 4.5 shows an example of a simple server with two versions.

Listing 4.5: niVerso example server

```
(version = '1.0') => {
  function users(req, res): {firstName: string, lastName: string}> {
    return {
      firstName: 'John',
      lastName: 'Doe'
    };
  };
};


(version = '2.0') => {
  function users(req, res): {name: string}> {
    return {
      name: (version='1.0') =>
        users(req, res).firstName + ' ' + users(req, res).lastName)
    };
  };
};

niverso.get('1.0', '/api/users', (version='1.0') => users);
niverso.get('2.0', '/api/users', (version='2.0') => users);
```

It might seem that there might be some duplication when creating a new route, since the version appears in two distinct places. In fact, we are inputing the version to two different components, and since one makes all the verificiations in compile time and the other in runtime, we request the version on both sides for simplicity.

4.2.4 Relation

We introduced relations in **Section** 3.2, and despite not being a component, the relation must be provided by the programmer. Since this prototype was written in JavaScript, the relation must also be implemented in JavaScript. Each version should be represented with an object with key, mode and children, representing its identifier, the degree of compatibility that connects to its parent, and newer versions originated from that.

For example, a relation between version A and B with the degree of compatibility set for subtyping should be defined as displayed in **Listing** 4.6.

Listing 4.6: Sample relation implemented in JavaScript

```
 {
   key: 'A',
   mode: null,
   children: [
     { key: 'B',
       mode: 'subtyping',
       children: []
     }
   ]
 }
```

With this tree structure, we can traverse the tree and apply the rules to switch version and execute code on the right version. **Listing** 4.7 represents the relation in **Figure** 4.1, where each version is represented by an identifier x.y-z, with x and y being positive integers. z represents a different path to the evolution of the software. Between y versions (e.g. 1.0, 1.1).The compatibility mode is subtyping, and between x versions (e.g. 1.0, 2.0), the compatibility mode is set to free.

Listing 4.7: Relation from Figure 4.1 implemented in JavaScript

```
 { key: '1.0', mode: null, children: [
   { key: '1.1-A', mode: 'subtyping', children: [
     { key: '2.0-A', mode: 'free', children: [ ]}
   ] }
```

```
    { key: '2.0-B', mode: 'free', children: [ ] }
  ] }
```

4.3 Architecture

In order to combine all the components together, we created a build process that starts with the code with version and type annotations and ends up runnig it on a web server. We can observe the overall architecture in **Figure** 4.2, which we described below.

The server code with version and type annotations is parsed into an AST. Our version type system receives that AST and the relation, and verifies if the relation is not violated. Then we execute Flow, that will typecheck the rest of the program. Since the code is still anotated and is not valid JavaScript code yet, we then remove all type annotations and execute our search algorithm to remove versions and select the accurate version for each identifier. That code is then executed over our router, that will automatic update every request accordingly.

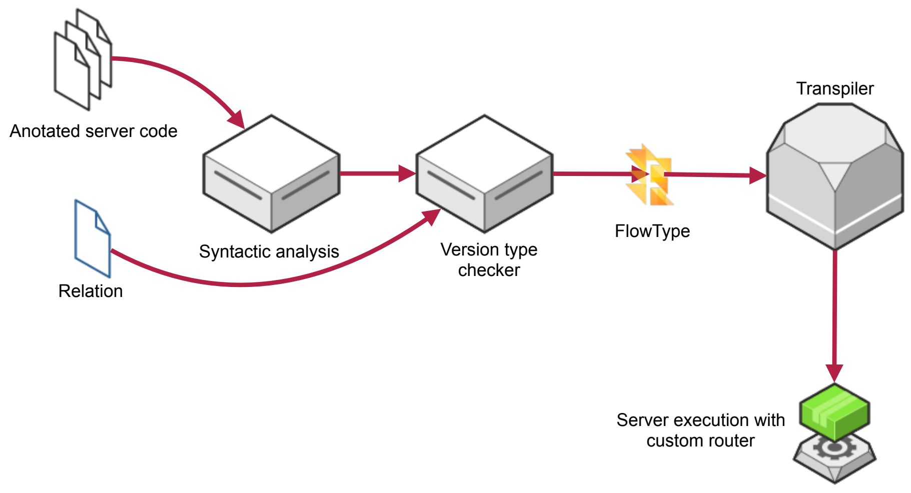

Figure 4.2: Prototype architecture

Chapter 5

Evaluation

In this chapter we present an evaluation of our work demonstrating that our apporach is a viable alternative to the state of the art. To validate our solution usefullness it is important to compare our solution to what is currently used in the industry.

We dill use two different systems from **Chapter** 2.1 as a base of comparison. We selected ASP.NET API Versioning since it provides official API versioning for a major framework, and restify since its approach is somewhat similar to ours. To keep it a fair comparison, we decided to create an example API to be implemented on the three platforms.

The sample API will allow a remote control of various aspects of a smart light bulb. We chosed this example because it provides a way of modeling a simple API while being a real world example. The API will control brightness, color and obviously allowing to turn the bulb on and off. For this example, we will follow the relation on **Figure** 3.2, and as such, four different versions of the API were created. The three implemented APIs are available for download at `https://github.com/joaocampinhos/lightbulb`. Next we present a list of endpoints to be implemented on those versions assuming an HTTP API.

1.0

```
GET                 /isOn
POST              /turnOn
POST             /turnOff
GET, POST          /color
GET, POST     /brightness
```

1.1-A

GET	/isOn
POST	/turnOn
POST	/turnOff
POST	/toggle
GET, POST	/color
GET, POST	/brightness

2.0-A

GET	/isOn
POST	/toggle
GET, POST	/color
GET, POST	/brightness

2.0-B

GET, POST	/state

5.1 ASP.NET API Versioning

ASP.NET provides a set of rules and methods to create a versioned API. As for the versions themselves, it uses a system of MAJOR.MINOR, similar to what we presented before but without branches, meaning that versions are related by a total order relation. Because of that, it is virtually impossible to implement our sample API as is. The solution is to drop version 2.0-B altogether and implement the rest of the versions, with 1.1-A renamed as 1.1 and 2.0-A renamed as 2.0. This versioning system only accepts breaking changes between majors, with the definition of breaking change meaning any change to the contract of an API [19]. While a reasonable definition, respecting these rules are up to the developer and are in no way mandatory when using the framework. Assuming that developers will never introduce breaking changes between minors (willingly or not) is a dangerous claim that might cause some disruption to the clients of that API.

The development of the API is really straightforward with almost no configuration needed. **Listing** 5.1 shows what the creation of a route looks like. In this case, we need to specify every version where this controller is versioned. If there are a reasonable amount of controllers that do not change often, this code might get a bit cumbersome, and might originate some duplicated code if not dealt with accordingly. Another important point is the request itself. Requesting a specific version is done through the url of the request (`/api/v1.0/brightness`), which does not facilitate automatic version update.

Listing 5.1: A versioned route using ASP.NET

```
[ApiVersion("1.0")]
[RoutePrefix("api/v{version:apiVersion}/brightness")]
public class BrightnessController : ApiController {
    [Route]
    public IHttpActionResult Get() => Ok(new { brightness = bulb.brightness });
    ...
}
```

The full implementation can be seen on **Appendix** A.

5.2 restify

Since the restify approach is similar to ours, the implementation was straightforward. It is only necessary to create functions and assign them to the desired endpoints. The first limitation arises when trying to implement the relation. Restify only supports semantic versioning and as such it is necessary to adapt the versions. Luckily, our relation uses a versioning system similar to semantic versioning, thus adapting is as easy as adding .0 to all versions. This way 1.0 becomes 1.0.0, 1.1-A becomes 1.1.0-A and so on.

Listing 5.2 shows how the creation of a route looks like. In this case, we need to specify every version where this route is valid. While a reasonable solution, it will add unnecessary complexity in the future if there are more versions, since old code needs to be modified in order to create a new version. An adequate solution will not increase in complexity just for adding more versions, as that might discourage the use of versions altogether. By way of example we can imagine the unnecessary complexity that an API with more than 1000 versions would have.

Listing 5.2: A versioned route using restify

```
server.get({
  path: '/brightness',
  version: ['1.0.0','1.1.0-A','2.0.0-A']
}, getBrightness);
```

Restify also does only solve part of the problem. We stated before that developers not respecting the versioning systems are a big problem that can make unexpected errors occur. Restify, while trying to enforce the semver and its set of rules, in no way makes those rules mandatory. This might result in a breaking change to occur on a version where that is not allowed.

There are however several benefits when using restify. The overhead for the developer is minimal, since it is only required to add on what versions that route is supported. In this sense, restify does a great job easing the introduction to versioned APIs which are a must in current web development.

The full implementation can be seen on **Appendix** B.

5.3 niVerso

Like we mentioned earlier our approach follows a somewhat similar approach to the restify. In our case, however, we only list where the resource gets available (the first version) and where it was deprecated. This allows a much more clean approach since there is no need to list that in every version. **Listing** 5.3 is an example of our approach and restify.

Listing 5.3: Comparison between niVerson and restify

```
// niverso
niverso.get(
  '1.0',
  '/brightness',
  (version='1.0') => getBrightness
);

// restify
server.get({
  path: '/brightness',
  version: ['1.0.0','1.1.0-A','2.0.0-A']
}, getBrightness);
```

What separates our solution from the rest is a version type checker that only allows an API to be published if it complies with the relation. This ensures greater assurance to API clients, since the introduction of breaking changes will not happen outside the versions allowed by the interface. This is what we feel is the great advantage when comparing to other solutions. However, this feature comes with more complexity for the developers, since they now have to manage the context versions and have to inevitably deal with that.

There is also a build system that needs to be configured in order to use our solution. That is, however, something that could be integrated into the platform on a later version. Static type checkers like FlowType for JavaScript still act like add-ons and as such it will never feel like developing "pure" JavaScript. It is undoubtedly simpler to use a system without it but in this case, the benefits outweigh the extra work.

The full implementation can be seen on **Appendix** C.

5.4 Final remarks

Table 5.1 shows the supported features between all three of the systems. In short, it is clear that niVerso is the system that provides the most features and guarantees, in expense for some extra configuration and adaptation by the programmer. In terms of performance however, none of the systems is affected, as they are just as fast as their non versioned counterparts. To conclude we present each of the features in a more detailed way and how each of the systems does support it, if at all.

Custom relation

Only niVerso supports custom relations, as that was one of the principles behind our solution. The other two systems rely on conventions used on the community, that albeit useful does limit their solution.

Version typechecker
Again only niVerso supports it, as we belive it is really important to prevent future errors to enforce the developers to respect the relation, that will end up providing an extra level of safety on the API.

Automatic upgrade
Resity provides partial support to version upgrade, since it is based of semantic versioning which was wildcards specific for that purpose. niVerso fully supports it, thanks to the abstraction of a relation that can really simplify its process. ASP.NET does not, having the requests always returning the requested version, or none at all.

Extra configuration
niVerso requires some extra configuration to be up and running, with also the implementation of the relation itself, something not required at all by the other two systems, since they only utilyze one versioning system. ASP.NET also requires the instalation of a package specific for versioned APIs, as well as some boilerplate code.

Extra build process
The problem of having a version typecheker means that it will also need an extra build process. Since the other two systems do not provide that feature, they also do not possess this drawback.

Performance overhead
Since all systems only do a basic redirect, performance overhead is not a concern. Some approaches, like Chain of Adapters which we presented on **Chapter** 2.1.1 could have a significant impact on the performance since it makes multiple redirects, which is not the case.

Table 5.1: Comparisson between solutions

Features	niVerso	restify	ASP.NET
Custom relation	yes	no	no
Version typechecker	yes	no	no
Automatic upgrade	yes	partial	no
Extra configuration	some	none	some
Extra build process	yes	no	no
Performance overhead	no	no	no

Chapter 6

Conclusion

Existing web and mobile applications are centered in data and often resort to web services to retrieve this data and use it. Cloud storage systems, social networks and a wide array of companies often provide public web services for third party users. Until now they require carefully planning and it's evolution is minimal, to avoid compromising software using those interfaces. Without guarantees, there is a possibility of introducing errors that might break software reliant on these services.

In this thesis, we propose a programming model to evolve and consume web services. This approach has the benefits of ensuring that all versions are well typed, as well as automatic upgrade client requests with a newer but compatible version. The compatibility is checked through the relation provided by the developer, which informs on which versions are compatible.

We designed the model and implemented it, first as a programming language, using as featherweight java as the base. We used this implementation to test different approaches and study all the possible cases. With a baseline established, we then created a prototype framework written in JavaScript. To complement the framework, a type system was designed to attest the soundness of versions and a transpiler responsible for desugaring the code with versions into code ready to run.

We evaluated our work by comparing our prototype with current solutions to build versioned web services. By implementing the same interface across all servers, we were able to verify that while our solution offers more assurances to developers when safely developing an API, there is the need to deal with some additional configuration when compared to existing solutions. However, if developers are already using a type checker like FlowType, the adaptation to our system is much smoother. In the end, we could attest to the usefulness and relevance of our solution.

6.1 Future Work

There is room for improvement in our solution, especially when it comes to to increasing readability and performance.

A better syntax for versions could improve readability and consequently make it easy to reason about the code. As an example, a construction `Version v { e }` to create versions and `e@v` to call a particular version resembles the syntax used in our model language and could help the readability of the overall code. This syntax could be achieved with the aid of a library to create hygienic macros or with an extension to the AST parser.

The possibility to scaffold versions. Our current implementation requires some work to convert a web service to be compatible with our system. This process could be easily automated with the creation of an IDE plugin. Our goal of code readability could also benefit from an IDE, as older versions could be hidden as a way for the programmer to focus on the current implementation.

Instead of running our type system and flow on top of it, it could be interesting to integrate our implementation entirely inside of flow. Flow is written in OCaml and has it's own AST parser. An extension of our system would provide improvements in performance since it would minimize the steps needed to build a program written in it. It would also be a much more complete experience since the software would run as a flow project as a whole rather than as separated files.

Bibliography

[1] I. Abal, C. Brabrand, and A. Wasowski. "42 Variability Bugs in the Linux Kernel: A Qualitative Analysis". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 421–432. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642990. URL: http://doi.acm.org/10.1145/2642937.2642990.

[2] *API Changelog*. URL: https://www.apichangelog.com/ (visited on 08/11/2016).

[3] *Babel*. URL: https://babeljs.io/ (visited on 01/09/2016).

[4] A. Chaudhuri. *Flow: a static type checker for JavaScript*. SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity. 2015.

[5] S. Chen, M. Erwig, and E. Walkingshaw. "An Error-tolerant Type System for Variational Lambda Calculus". In: *SIGPLAN Not.* 47.9 (Sept. 2012), pp. 29–40. ISSN: 0362-1340. DOI: 10.1145/2398856.2364535. URL: http://doi.acm.org/10.1145/2398856.2364535.

[6] M. Erwig and E. Walkingshaw. "The Choice Calculus: A Representation for Software Variation". In: *ACM Trans. Softw. Eng. Methodol.* 21.1 (Dec. 2011), 6:1–6:27. ISSN: 1049-331X. DOI: 10.1145/2063239.2063245. URL: http://doi.acm.org/10.1145/2063239.2063245.

[7] M. Erwig, K. Ostermann, T. Rendel, and E. Walkingshaw. "Adding Configuration to the Choice Calculus". In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*. VaMoS '13. Pisa, Italy: ACM, 2013, 13:1–13:8. ISBN: 978-1-4503-1541-8. DOI: 10.1145/2430502.2430520. URL: http://doi.acm.org/10.1145/2430502.2430520.

[8] i. free software foundation. *c preprocessor*. URL: https://gcc.gnu.org/onlinedocs/cpp/ (visited on 08/10/2016).

[9] T. E. Foundation. *Eclipse Modeling Framework*. URL: https://eclipse.org/modeling/emf/ (visited on 01/15/2016).

[10] E. Hammer. *hapi.js*. URL: http://hapijs.com/ (visited on 01/10/2016).

[11] A. Igarashi, B. C. Pierce, and P. Wadler. "Featherweight Java: A Minimal Core Calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505. URL: http://doi.acm.org/10.1145/503502.503505.

[12] S. E. Institute. *Product Line Hall of Fame*. URL: http://www.splc.net/fame.html (visited on 01/30/2016).

[13] S. E. Institute. *Software Product Lines*. 2016. URL: http://www.sei.cmu.edu/productlines/ (visited on 08/16/2016).

[14] P. Kaminski, M. Litoiu, and H. Müller. "A Design Technique for Evolving Web Services". In: *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '06. Toronto, Ontario, Canada: IBM Corp., 2006. DOI: 10.1145/1188966.1188997. URL: http://dx.doi.org/10.1145/1188966.1188997.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[16] J. Kopecky, K. Gomadam, and T. Vitvar. "hRESTS: An HTML Microformat for Describing RESTful Web Services". In: *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT '08. IEEE/WIC/ACM International Conference on*. Vol. 1. 2008, pp. 619–625. DOI: 10.1109/WIIAT.2008.379.

[17] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. "End-to-End Versioning Support for Web Services". In: *Services Computing, 2008. SCC '08. IEEE International Conference on*. Vol. 1. 2008, pp. 59–66. DOI: 10.1109/SCC.2008.21.

[18] J. Mangler, P. Beran, and E. Schikuta. "On the Origin of Services Using RIDDL for Description, Evolution and Composition of RESTful Services". In: *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. 2010, pp. 505–508. DOI: 10.1109/CCGRID.2010.126.

[19] Microsoft. *ASP.NET versioning guidelines*. URL: https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#12-versioning (visited on 10/10/2016).

[20] Microsoft. *TypeScript*. URL: http://typescriptlang.org (visited on 01/05/2016).

[21] T. Preston-Werner. *Semantic Versioning*. URL: http://semver.org/ (visited on 08/10/2016).

[22] D. Romano and M. Pinzger. "Analyzing the Evolution of Web Services Using Fine-Grained Changes." In: *ICWS*. Ed. by C. A. Goble, P. P. Chen, and J. Zhang. IEEE Computer Society, 2012, pp. 392–399. ISBN: 978-1-4673-2131-0. URL: http://dblp.uni-trier.de/db/conf/icws/icws2012.html#RomanoP12.

[23] S. Sohan, C. Anslow, and F. Maurer. “A Case Study of Web API Evolution”. In: *Services (SERVICES), 2015 IEEE World Congress on*. 2015, pp. 245–252. DOI: 10.1109/SERVICES.2015.43.

[24] H. Spencer and G. Collyer. “#ifdef considered harmful, or portability experience with C News”. In: *USENIX Summer Technical Conference*. San Antonio (Texas), USA, June 1992, pp. 185–197.

[25] StrongLoop. *Express*. URL: http://expressjs.com/ (visited on 01/10/2016).

[26] D. C. Tim Berners-Lee. *Notation3*. URL: https://www.w3.org/TeamSubmission/n3/ (visited on 01/15/2016).

[27] R. Verborgh, T. Steiner, D. Deursen, J. Roo, R. V. D. Walle, and J. Gabarró Vallés. “Capturing the Functionality of Web Services with Functional Descriptions”. In: *Multimedia Tools Appl.* 64.2 (May 2013), pp. 365–387. ISSN: 1380-7501. DOI: 10.1007/s11042-012-1004-5. URL: http://dx.doi.org/10.1007/s11042-012-1004-5.

[28] W. W. W. C. (W3C). *Semantic Web*. URL: https://www.w3.org/standards/semanticweb/ (visited on 01/15/2016).

Appendix A

ASP.NET implementation

```
class bulb {
  public boolean on { get; set; }
  public int brightness { get; set; }
  public int cr { get; set; }
  public int cg { get; set; }
  public int cb { get; set; }
}

[ApiVersion("1.0")]
[ApiVersion("1.1")]
[ApiVersion("2.0")]
[RoutePrefix("api/v{version:apiVersion}/brightness")]
public class BrightnessController : ApiController {
  [Route]
  public IHttpActionResult Get() => Ok(new { brightness = bulb.brightness });

  [Route]
  public IHttpActionResult Post( [FromODataUri] b boolean ) {
    if ( !ModelState.IsValid ) { return BadRequest( ModelState ); }
    bulb.brightness = b;
    return Ok(new { brightness = bulb.brightness });
}

[ApiVersion("1.0")]
[ApiVersion("1.1")]
[ApiVersion("2.0")]
[RoutePrefix("api/v{version:apiVersion}/color")]
public class ColorController : ApiController {
  [Route]
  public IHttpActionResult Get() =>
    Ok(new { color = new {r = bulb.cr, g = bulb.cg, b = bulb.cb}});

  [Route]
  public IHttpActionResult Post( [FromODataUri] r int, g int, b int ) {
    if ( !ModelState.IsValid ) { return BadRequest( ModelState ); }
    bulb.cr = r;
    bulb.cg = g;
    bulb.cb = b;
    return Ok(new { color = new {r = bulb.cr, g = bulb.cg, b = bulb.cb}});
}

[ApiVersion("1.0")]
[ApiVersion("1.1")]
[ApiVersion("2.0")]
[RoutePrefix("api/v{version:apiVersion}/isOn")]
public class isOnController : ApiController {
  [Route]
  public IHttpActionResult Get() => Ok({ isOn = bulb.on});
}

[ApiVersion("1.0")]
[ApiVersion("1.1")]
[RoutePrefix("api/v{version:apiVersion}/turnOn")]
public class TurnOnController : ApiController {
  [Route]
  public IHttpActionResult Post() {
    bulb.on = true;
    return Ok(new { isOn = bulb.on });
}

[ApiVersion("1.0")]
[ApiVersion("1.1")]
[RoutePrefix("api/v{version:apiVersion}/turnOff")]
public class TurnOffController : ApiController {
  [Route]
  public IHttpActionResult Post() {
    bulb.on = false;
    return Ok(new { isOn = bulb.on });
}

[ApiVersion("1.1")]
[ApiVersion("2.0")]
[RoutePrefix("api/v{version:apiVersion}/toggle")]
public class ToggleController : ApiController {
  [Route]
  public IHttpActionResult Post() {
    bulb.on = !bulb.on;
    return Ok(new { isOn = bulb.on });
}
```

Appendix B

Restify implementation

```
var restify = require('restify');

var bulb = {
  on: false,
  color: {r: 255, g: 255, b: 255},
  brightness: 1
}

function toggle(req, res, next) {
  bulb.on = !bulb.on;
  res.send(bulb.on);
  return next();
};

function turnOn(req, res, next) {
  bulb.on = true;
  res.send(bulb.on);
  return next();
}

function turnOff(req, res, next) {
  bulb.on = false;
  res.send(bulb.on);
  return next();
}

function isOn(req, res, next) {
  res.send(bulb.on);
  return next();
}

```

```
function getColor(req, res, next) {
  res.send(bulb.color);
  return next();
}

function setColor(req, res, next) {
  bulb.color = req.body;
  res.send(bulb.color);
  return next();
}


function getBrightness(req, res, next) {
  res.send(bulb.brightness);
  return next();
}

function setBrightness(req, res, next) {
  bulb.brightness = req.body.brightness;
  res.send(bulb.brightness);
  return next();
}

function getState(req, res, next) {
  res.send(bulb);
  return next();
}

function setState(req, res, next) {
  bulb = req.body;
  res.send(bulb);
  return next();
}

var server = restify.createServer();
server.use(restify.bodyParser());

server.use(function (req, res, next) {
  res.setHeader('matchedVersion', req.matchedVersion());
  return next();
});

server.get({path: '/isOn', version: ['1.0.0','1.1.0-A','2.0.0-A']}, isOn);
server.post({path: '/turnOn', version: ['1.0.0','1.1.0-A']}, turnOn);
server.post({path: '/turnOff', version: ['1.0.0','1.1.0-A']}, turnOff);

server.get({path: '/color',
  version: ['1.0.0','1.1.0-A','2.0.0-A']}, getColor);
server.post({path: '/color',
  version: ['1.0.0','1.1.0-A','2.0.0-A']}, setColor);
```

```

server.get({path: '/brightness',
  version: ['1.0.0','1.1.0-A','2.0.0-A']}, getBrightness);
server.post({path: '/brightness',
  version: ['1.0.0','1.1.0-A','2.0.0-A']}, setBrightness);

server.post({path: '/toggle', version: ['1.1.0-A','2.0.0-A']}, toggle);

server.get({path: '/state', version: '2.0.0-B'}, getState);
server.post({path: '/state', version: '2.0.0-B'}, setState);

server.listen(8080, function() {
  console.log('%s listening at %s', server.name, server.url);
});
```

Appendix C

niVerso implementation

```
const niverso = require('niverso');
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

var bulb = {
  on: false,
  color: {r: 255, g: 255, b: 255},
  brightness: 1
}

(version = '1.0') => {
  function isOn(req, res): boolean {
    return bulb.on;
  };

  function turnOn(req, res): boolean  {
    bulb.on = true;
    return isOn(req, res);
  };

  function turnOff(req, res): boolean  {
    bulb.on = true;
    return isOn(req, res);
  };

  function getColor(req, res): {r: number, g: number, b: number} {
    return bulb.color;
  };

  function setColor(req, res): {r: number, g: number, b: number} {
    bulb.color = req.body;
    return getColor(req, res);
  };

  function getBrightness(req, res): number {
    return bulb.brightness;
  };

  function setBrightness(req, res): number  {
    bulb.brightness = req.body.brightness;
    return getBrightness(req, res);
  };
};

(version = '1.1-A') => {
  function toggle(req, res): {
    if (bulb.on) return turnOff(req, res);
    else return turnOn(req, res);
  };
};

(version = '2.0-B') => {
  function getState(req, res): {
      on: boolean,
      color: {
        r: number,
        g: number,
        b: number
      },
      brightness: number
    } {
    return bulb;
  };

  function setState(req, res): {
      on: boolean,
      color: {
        r: number,
        g: number,
        b: number
      },
      brightness: number
    } {
    bub = req.body;
    return getState(req, res);
  };
};


niverso.use(require('relation'));

niverso.get('1.0', '/isOn', (version='1.0') => isOn);
niverso.post('1.0', '/turnOn', (version='1.0') => turnOn);
niverso.post('1.0', '/turnOff', (version='1.0') => turnOff);

niverso.get('1.0', '/color', (version='1.0') => getColor);
niverso.post('1.0', '/color', (version='1.0') => setColor);

niverso.get('1.0', '/brightness', (version='1.0') => getBrightness);
niverso.post('1.0', '/brightness', (version='1.0') => setBrightness);

niverso.post('1.1-A', '/toggle', (version='1.1-A') => toggle);

niverso.post('2.0-A', '/turnOn', niverso.deprecate);
niverso.post('2.0-A', '/turnOff', niverso.deprecate);

//deprecate all
niverso.get('2.0-B', '/isOn', niverso.deprecate);
niverso.post('2.0-B', '/turnOn', niverso.deprecate);
niverso.post('2.0-B', '/turnOff', niverso.deprecate);
niverso.get('2.0-B', '/color', niverso.deprecate);
niverso.post('2.0-B', '/color', niverso.deprecate);

niverso.get('2.0-B', '/brightness', niverso.deprecate);
niverso.post('2.0-B', '/brightness', niverso.deprecate);

niverso.post('2.0-B', '/toggle', niverso.deprecate);

niverso.get('2.0-B', '/state', (version='2.0-B') => getState);
niverso.post('2.0-B', '/state', (version='2.0-B') => setState);

niverso.start(app);

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(3000, () => console.log('Server listening on port 3000'));
```