Type-Safe Evolution of Web Services

João Campinhos, João Costa Seco, Jácome Cunha
NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa, Portugal
j.campinhos@campus.fct.unl.pt, joao.seco@fct.unl.pt, jacome@fct.unl.pt

***Abstract*—Applications based on micro or web services have had significant growth due to the exponential increase in the use of mobile devices. However, using such kind of loosely coupled interfaces provides almost no guarantees to the developer in terms of evolution. Changes to service interfaces can be introduced at any moment, which may cause the system to fail due to mismatches between communicating parts.**

In this paper, we present a programming model that allows the development of web service applications, server end-points and their clients, in such a way that the evolution of services' implementation does not cause the disruption of the client. Our approach is based on a type based code slicing technique that ensures that each version only refers to type compatible code, of the same version or of a compatible version, and that each client request is redirected to the most recent type compatible version implemented by the server.

We abstract the notion of version and parametrize type compatibility on the relation between versions. The relation between versions is tagged with compatibility levels, so to capture the common conventions used in software development. Our implementation allows multiple versions of a service to be deployed simultaneously, while reusing code between versions in a type safe way. We describe a prototype framework, based on code transformation, for server-side JavaScript code, and using Flow as verification tool.

***Keywords*-API evolution, web services, type safe, JavaScript**

I. Introduction

Micro-service and web service based architectures have had significant growth mostly due to the exponential increase in the use of mobile devices whose applications rely almost entirely on this kind of interfaces. Any modification to the behaviour or structure of a remote interface can cause unexpected execution errors on applications using it, as it is made clear in [6]. In fact, developers may not even know the exact version of all web services their applications use, making them error prone [5]. A recent study [17] shows that, from a sample of 317 libraries studied, 74% have at least one change in their interfaces, and 84% of those introduce disrupting changes. Thus, it is reasonable to conclude that software evolution often causes clients to break.

Several approaches aim at solving the issue of evolving web service interfaces, some do it at the level of description of services and request routing [13], [12], others propose good practices and design patterns [8], and others use a language-based approaches to reconfigure and synchronize client and server code [2], [3]. Although they allow for multiple versions to co-exist, they require manual and ad-hoc forwarding of control, and do not provide language based mechanisms to properly maintain the implementations. While inspired in the approaches above, we generalize the kind of safe variability presented in [4] in a language based approach.

To avoid the kind of disruption described above we introduce a novel programming model that supports the smooth evolution of service interfaces and corresponding implementations. We adopt a language based approach that makes it possible for several versions of a service to coexist in the same running system, to be defined by the same source code, to share and reuse functionality, and yet enjoy type soundness as separate slices of code. This approach extends the classic notions of software variability, where the same source can be used to produce different versions of the same program, but only one can be running at a given time [7], [9], [4]. Technically, version tag modifiers are attached to service interface declarations (in the routing component in an application server) and also to program declarations (e.g. variables, functions), which are used to support a version sensitive method dispatching and execution mechanism. The system's code is pre-compiled using the prescribed versioning policies, where binding of identifiers is statically resolved, and where a static type verification is applied to ensure the soundness of each version.

We characterize a generic notion of versioning system, and make the programming model parametrised on the definition of version and the definition of a partial order relation on versions. We thus allow for different styles of versioning systems to be used.

A client request is typically associated to a version of a given service. Our goal is to always route the request to a specific version of a service that is the *newest* version that is *compatible* with the request.

We introduce three kinds of compatibility modes that are used in the definition of the version relation and can also be used to shape the response to a client's request when searching for a compatible version. Mode *strict*, where type equality is required to both type a new version; mode *subtyping*, which allows for a new version to be defined provided that a subtyping relation is maintained; and mode *free* where no relation is required between versions. For instance, it is possible to define a versioning relation using major and minor versions, and patches, where *strict* compatibility is required between related patches, *subtyping* between minor versions, and no compatibility is required between major versions of the code (*free* mode).

Client applications may always be updated, but by using this version based mechanism there is the guarantee that there will be no type-compatibility issues until then. Moreover, the client

can request a particular version for which case the dispatching service will not search for the newest but return the exactly the requested one.

We developed a prototype implementation using type server-side JavaScript, taking advantage of the Flow [1] static type checker, based on data flow analysis, to verify that all versions of the code are sound, under the prescribed relation and compatibility modes between versions.

The remainder of this paper is organized as follows. Section II provides an illustrative example and introduces the principles for our programming model. In Section III we detail our approach including the relation and type system. Section IV describes the structure of a working prototype written in JavaScript termed NIVERSO. Section V discusses related work, and Section VI presents our concluding remarks and some directions for future work.

II. Design Principles By Example

Our development is guided by a set of principles, elicited by analysing a set of real web service interfaces, that summarize the good practices that we believe have a positive impact on the current state of the art. We analysed the APIs from Twitter, GitHub, stripe, Google Maps, and the ones studied by Sohan et al. [16]. We introduce and illustrate these principles by means of a small example.

Consider the implementation of a web service that provides information about the current user logged in to the system. The first implementation of the service, version 1.0, yields a JSON object as result, containing a string value.

server implementation – version 1.0

```
user() => { name: "John Doe" }
```

Calling the service implemented by method `user()` returns a JSON object with the key `name`. A possible (well typed) client for version 1.0, that uses the current user's name, is implemented by

client implementation – version 1.0

```
user().name.trim()
```

The next major version of the service, 2.0, is improved to return a structured JSON object where `name` is modified to denote an object with two fields: `first`, and `last`.

server implementation – version 2.0

```
user() => { name:{ first: "John",
                   last: "Doe" }}
```

This is suitable for a new client implementation, such as

client implementation – version 2.0

```
user().name.first+" "+user().name.last
```

but not for our first client implementation, of version 1.0 (`user().name`), which will stop working if put together with version 2.0 of the web service. This simple illustration pictures well what happens at the scale of the universe of cloud based applications, whose evolution and change demands are increasing by the day [17]. It is not feasible to update all client instances to the latest version, and it is not possible not to change the service interfaces.

Any solution that copes with the evolution of service interfaces should follow the *Coexistence principle*. Our solution is to make all the existing versions available as a complete application, and use annotations in the client requests and in the server code to identify which version(s) to use to attend a request. Thus, both versions 1.0 and 2.0 of the server should coexist at the same time and should be usable by clients.

Nevertheless, there are changes that can be made to the server implementations that should be transparent to the clients' implementation. For instance, a new minor version of the service, version 2.1, can be written as follows:

server implementation – version 2.1

```
user() => { name: { first: "John",
                    last: "Doe" },
            age: 42 }}
```

In this case, the type of the response is a subtype of the response in version 2.0. In many cases, it is still possible for clients to interact with the new version. This makes possible for version 2.1 to be used whenever version 2.0 is requested. Our solution is to use a pre-defined relation on versions that embeds the notion of compatibility between versions. This leads us to our second principle, the *Compatibility principle*, that states that a system should serve its clients with (type) compatible versions at all times.

The previous principle guide the design of the request dispatching mechanism. A third (and more general) principle becomes essential to reach a sound solution, which is the *Safety principle*. All possible entry points of the system should be type safe, in all versions, and considering all possible version redirections given by the version compatibility relation. In this particular example, we introduce no constraints (called the *free* compatibility mode) on changes between two different "major" versions (the return type of `user` is `{name:string}` in version 1.0 and `{name:{first:string, second:string}}` in version 2.0). Moreover, we use the *subtyping* mode to constraint the comparison of minor versions, which means that the relation on versions must follow the universal subtype relation on interface types (`{name:{...}}, age:int` in version 2.1). The parametric use of a relation on versions, and compatibility modes is an instance of our fourth principle, the *Modularity principle*, which decouples the static verification of code and the dynamic dispatching of requests and execution from the actual definition of what is a version and how they relate.

The coexistence of versions, only makes sense if it is possible to reuse code across versions, and to explicitly define the version on which an expression is evaluated. We introduce a language construct that allows the annotation of an expression with a version pattern. A version pattern defines the base for searching the correct implementation for all free names in a given scope. In our example, an annotation of the client's code with the version pattern `2.*` means that all free names are bound to the latest compatible version, higher than `2.0`.

second client implementation – version 2.0

```
(version = '2.*')=>{ user().name.first }
```

The client code is still in version `2.0`, but is requesting the use of the most updated version of the service that is compatible with the interface of version `2.0`. Both the static type system and the runtime dispatching system, take into consideration the relation between versions 2.0 and 2.1, using the subtyping mode between minor versions, as described above.

III. Programming Model - Introducing Versions

Our programming model is based on the addition of a simple version context expression to an imperative language, with the form `(version = 'v') => { e }`.
that determines where, on which version, the effects of expression `e` or the body of the context, are visible. In this case, all the declarations and computations contained in the elided body `e` of the expression take place in the context of version `v`. This also means that every subexpression is evaluated with reference to that particular version. One important factor, the reuse of code between versions, is supported by the nesting of version contexts. For instance, the type annotated implementation of service `user` in our example described in section II, where three versions of the service coexist (`1.0`, `2.0`, and `2.1`), is given by the code fragment of Figure 1. Our approach follows what is promoted by the chain of adapters pattern [8], where the implementation of the web service is based on the most recent version (`2.1`) that centralises the state and main operations, and where all "old" versions are obtained by a facade to the most recent version. However, this adaptation is automatic and systematically handled and verified by the programming language mechanisms. Notice that declarations must be placed in a concrete version, but expressions in general can be executed in a more flexible context, defined by a version pattern (`2.*`). Both the runtime and type system perform the binding of the occurrences of identifiers with the most appropriate compatible versions.

Also, in the design of the web service supporting architecture, binding is essential. A client's code can be written as follows:

```
(version = '2.*', mode = 'subtyping') =>
  { "Welcome " + user().name.first };
```

The major difference from the client to the server code is that the client may use different compatibility modes (the default mode being **strict**) according to its own capabilities (e.g., to cope with subtyping), while the server always knows the correct type for a given declaration. In this particular example the client is requesting the `user` function within the scope of the version pattern `2.*`. This means that the system should call the highest version of the service related to major version `2`, but only using version pairs in the relation that are tagged with the subtyping mode. In this case the server will find version `2.1` as the only one fulfilling the request. Version patterns are defined as regular expressions over the language of versions. In Section III-C we detail how the appropriate version is computed. In order to make this binding process deterministic, we need to discipline the definition of the versioning system.

```
(version = '2.1') => {
  currentuser =
    {name:{first:"John", last:"Doe"},
     age:42};
  function user():
    {name:{first: string, last: string},
     age:int}
  {
    return (version = '2.*')=>{currentuser};
  }
}

(version = '2.0') => {
  function user():
    {name:{first: string, last: string}}
  {
    return
      (version = '2.*')=>{name:user().name};
  }
}

(version = '1.0') => {
  function user():{name: string}
  {
    var u = (version='2.*') => {user()};
    return
      {name:u.name.first+" "+u.name.last};
  }
}
```

Fig. 1. Service implementation

A. The Relation

The relation between versions at the core of our programming model is a partial order extended with compatibility modes. Each pair of versions in the relation is tagged with a compatibility mode that denotes how code evolves between them. Note that there may be no relation between any pair of versions. For instance, Figure 2 pictures version `1.0` and `1.0-A` as related under the compatibility mode *strict*, meaning the implementation may vary but the type will not.

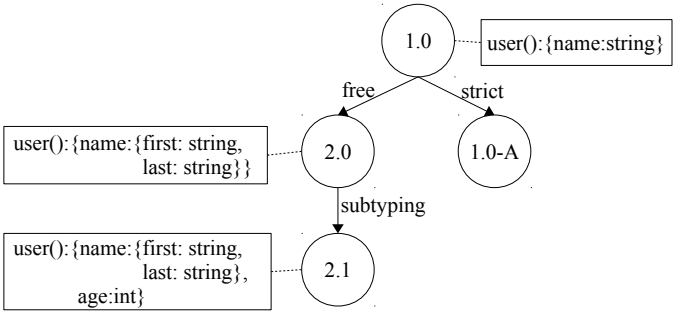

Fig. 2. Example of the type system structure

We introduce three compatibility modes. The **strict** mode, which imposes that all the changes performed from one version to another preserve the types of all declared identifiers (functions and variables). The **subtyping** mode that imposes that all changes performed from one version to another (new declarations) follow the subtype relation (e.g. in our running example, the type of function `user`

changes from `{name:{first:string,last:string}}` to `{name:{first:string,last:string},age:int}}`, which is a subtype). Finally, the **free** mode that does not constraint the changes on the type signatures, as to allow the developer to arbitrarily change the source code. Nevertheless, the code remains well-typed and safe, since the former signature is considered and no automatic redirection between these versions is taken.

B. Type System

The type system is designed to allow more than one declaration for the same identifier, and extra steps to actually type-check the code of a web service or client encoded using our programming model. For each declaration, independently of the version context where it occurs, our type system compares the type signatures in the immediate predecessor and successor versions and compares them according to the compatibility mode prescribed by the current relation on versions. Each occurrence of an identifier is bound to a given declaration depending on the current version and the relation of versions (as depicted by the tree in Figure 2). Note that, when typing any expression, it is always possible to determine the type of an identifier. But, when a real decoupling situation arises, like in the a web service's client code, we allow the explicit use of compatibility modes between the versions to shape the searching algorithm. The same applies to modules or libraries under a separate compilation strategy.

If the types are different, then the type checker will have in consideration the relation between the corresponding versions. For instance, if a relation between two immediately related versions is tagged as free, then no type comparison actually occurs between the declarations in one or the other. However, if the relation is tagged in the subtyping mode, then such relationship between the types is verified and a type error given if the relationship does not hold.

More formally, given a declaration d to type check, and given the versions' relation r the function `lookup()` will type check that declaration:

$$\begin{array}{l}\texttt{lookup}(r,d) = \\ \quad \forall (v_1,v_2)_t \in r \;.\; d \in v_1 \implies \texttt{lookup'}(\texttt{collect}(d,v_2),d,t)\end{array} \tag{1}$$

where t it the compatibility type between versions v_1 and v_2 and `collect()` collects all the declarations of identifier d contained in the code of version v_2.

The function `lookup'()` is defined as follows:

$$\texttt{lookup'}(\overline{d},d,t) = \begin{cases} true \;\; \text{if } \overline{d} = \emptyset \wedge \texttt{typecheck}(d) \\ true \;\; \text{if } \overline{d} \neq \emptyset \wedge t = free \\ true \;\; \text{if } \overline{d} \neq \emptyset \wedge t = subtyping \wedge \forall d' \in \overline{d} \;.\; d <: d' \\ false \text{ otherwise} \end{cases} \tag{2}$$

where the function `typecheck()` is the regular type checker function, and $<:$ represent the subtyping relation.

Note that when a declaration type checks it is added to the set of valid declarations, which is then used by the type checker (we omit these details for the sake of space).

C. Retrieving the Highest Version

Whenever the client makes a request to the web service the dispatching mechanism must decide which version to serve. Each request contains the function requested, a version pattern, and a compatibility mode. The dispatching mechanism then searches the relation for the version pattern starting from the lowest possible version. It then transitively continues to search the next version fitting the pattern and complying with the compatibility mode given. For instance, for our running example, the pattern `2.*` would start searching in the `2.0` node. Under mode strict, version `2.0` is served. But under mode subtyping the search continues to the next highest node, which is version `2.1`. If we look to Figure 2 we can see that the algorithm traverses the tree downwards. Under the free mode, the search function gets the highest version corresponding to the pattern without considering the compatibility mode between the versions. In strict/subtyping mode the function searches the most recent version available if there is a path in the relation to that version using only strict/subtyping relations.

More formally, we define a version lookup function (`version()`) that computes the highest version, given a set of available versions ($\overline{v}$), a starting version (v), and a search mode (t). the compatibility mode t can be instantiated by the relation $\leq$ for the strict mode, $\lesssim$ for the subtyping mode, and $\ll$ for the free mode.

$$\begin{array}{l}\texttt{version}(\overline{v},v,t) = v' \\ \qquad \text{where } v' \in \overline{v} \wedge v \; t \; v' \wedge \forall v'' \in \overline{v} \;.\; v \; t \; v'' \wedge v'' \; t \; v'\end{array} \tag{3}$$

Hence, a versioning system is explicitly defined, by defining how to form a version identifier, a version pattern, and a relation between version identifiers and using compatibility modes. This can be done in a variety of ways. For instance, by explicitly listing all the pairs of versions that are related together with the corresponding compatibility modes, or by providing a universal rule between versions, as it is the case of the major/minor version scheme. The constraint that must be observed is that a given version has a single direct lower value, and that there is always a single path from any version to the root of the versioning system.

IV. Prototype Implementation

Although the idea of a versioning system and versioning context is a general language construct, and can in principle be applied to any programming language, we illustrate it here in the context of a language extension to implement web services. We validate our approach by providing a prototype implementation, called NIVERSO, of a pre-processing system that statically resolves the binding of methods between versions, and dynamically dispatches service requests according to the version relation. In the following sections, we describe the implementation of NIVERSO in detail.

A. Architecture

The architecture of NIVERSO is depicted in Figure 3. The first step is the static analyser, Babel, that parses code with version and type annotations and produces an abstract syntax

tree (AST). Source code is written in Flow [1], which is basically JavaScript with type annotations. Version annotations follow standard JavaScript syntax for anonymous functions, and version literals are strings, just as described in the example in section II.

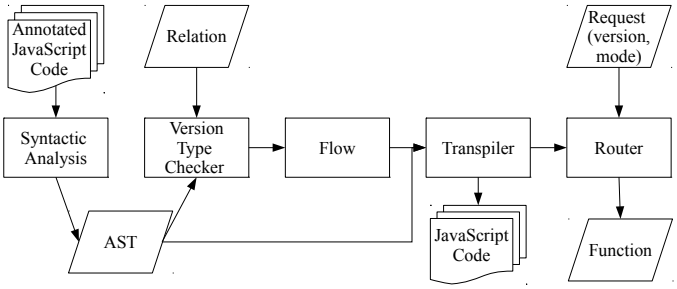

Fig. 3. Prototype architecture.

The second step consists of a type checker that takes the AST and the relation on versions as input, and ensures that the code is well-typed considering all the type annotations and the versioning relation. At this stage, the type checker only reads the type signatures for each declaration and compares its different versions under the corresponding compatibility modes. In the third step the NIVERSO executes Flow, which fully typechecks the annotated JavaScript code. Since Flow is not valid JavaScript we use in the fourth step a source-to-source compiler (transpiler) to remove all type annotations. Moreover, in this step we also perform the static binding of identifiers to the valid version in each context, according to the relation on versions.

Finally, the target code can then be executed by the runtime system router, that automatically dispatches each request, tagged with a version and a compatibility mode, to the appropriate implementation.

B. Implementing Versions

Versions are implemented by an abstract data type, with an opaque type `version` and the implementation of a predefined *relation interface*, that explicitly defines the relation on versions. The system expects a class with the following (here annotated) methods:

```
boolean isVersion(version v):
```

that verifies that a given version identifier `v` is a valid version of the relation (in our running example, type `version` is `string` following a certain syntactic pattern).

```
version[] pathToRoot(version v):
```

returns an array of versions representing a path starting in version `v` to the lowest "parent" version.

```
int[] typeToRoot(version v):
```

returns an array of integers representing the compatibility mode from the lowest version until version `v` (the default values are 0 to strict mode, 10 to subtyping and 20 to free).

Using these methods we can traverse the relation and execute the search algorithm and determine the appropriate version for each identifier.

C. Resolving Version Contexts

Context versions use standard JavaScript syntax for functions, but follow a standard pattern that is used to identify the current version and help establishing the correct binding for identifiers. Each version of an identifier is transformed into a different declaration. For instance,

```
(version = '1.0') => { var a = 0; }
(version = '2.0') => {
   var b = a + 1;
   var a = b * (version = '1.0') => a }
```

is transformed into

```
var a__1 = 0;
var b__2 = a__1 + 1;
var a__2 = b__2 + a__1;
```

D. Dispatching requests

The router component in our architecture is responsible for dispatching the service requests. Our prototype is implemented on top of the express framework, a minimal web application framework for JavaScript. Express is used to create JSON APIs that correspond to a method each time a specific route (version) is called. To cope with the dynamic binding of methods to versions, we extend the standard routing mechanism to allow multiple route definitions for a single entry point, and use a request parameter to decide (the version). Thus, in the server setup code, we declare that a given route is available in a given version as follows:

```
app.get('2.*','/user',(req,res) =>
  { res.send(user()); });
```

The router builds a dispatching table that is used as input of the search algorithm described in Section III.

Our prototype, in the form of source code, is available online at http://github.com/niverso. It features the full system (*niverso* repository), plus an example that uses a simple relation (*Relations* repository) to create a web service similar to the one presented in Section II.

V. Related Work

Chain of adapters [8] is an architectural pattern and 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. It also 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, redesign, and otherwise rethink both the service's interface and its implementation without being shackled by previous decisions. This technique provides web service evolution consisting of multiple versions concurrently deployed without code repetition and preserving backward compatibility. Since every version remains isolated on its own adapter, it should be easy to remove a version, as long as removed in a chronological order (oldest first). Our solution retains the reusability aspect without the overhead of multiple forwards. However, since we are creating new versions with code from older ones,

deleting a version is not trivial. Nevertheless, deleting versions is not something that should be done often, since the intended approach is to create a new one instead.

RIDDL [13] is an XML-based language used to incrementally compose REST APIs documentation by adding a change-log of the older version. Because the lack of a standard way to describe REST interfaces RIDDL 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. While RIDDL is an interesting approach for WSDL, with the added benefit of allowing service evolution, it only provides proper documentation and some skeleton code, at most. It does not tackle the issue of maintainability, where the client still needs to update the code whenever a breaking change occurs.

VRESCo [12] is a runtime environment that acts as a proxy between the client calls and the actual version of the web service. This work distinguishes different kinds of changes to the code and handles them in different ways. For instance, adding an operation is what can be called a transparent change which means that the runtime can handle it automatically. Other changes require intervention from the developer. VRESCo provides an interesting approach for using a proxy to route the client calls to the selected version. It also allows the developer to evolve its code base and the environment will then automatically update the tags based on the changes. This solution presents a similar approach to ours by re-routing the requests, though it will only work when a version is tagged.

WSDLDiff [15] is a tool to extract fine-grained changes from subsequent versions of a web service interface defined in WSDL. It extracts WSDL elements affected by changes in subsequent versions. The paper present a study on some well known WSDLs available in order to understand how they evolve over time. Thus, a subscriber to those WSDLs can predict which operations are more likely to change over time, providing a safeguard to developers. However, this still requires the developer intervention, and the goal of service evolution is leaning towards automation, as we proposes.

hRESTS [10] is a micro format for machine-readable descriptions of RESTful APIs, backed by a simple service model. It describes the main aspects of services: operations, inputs, and outputs. hRESTS translates HTML hierarchy into a hierarchy of objects and properties. By doing that, the developer can make the crucial parts of existing API documentation machine-readable, making it possible the auto-generation of client code, based on the gathered information. Although the use of hRESTS requires extra effort from the developer, hRESTS could be used to generate change-logs on the different versions and spot API breaking changes on the endpoints. Thus, providing machine-readable specification could mean automation on some tasks that currently require human interaction, such as migrating to new API versions.

More generally, variability is currently a hot topic of research. For instance, choice calculus (CC) is a framework which addresses variability for programming languages [4]. By using dimensions and choices, CC abstracts variability from the language itself. Thus, one can reason about the variability of the code, for instance, being able to search for inconsistencies without actually needing the code. As in our approach, CC has also its own type system. Because CC serves as a foundation for further development, we reuse some of its concepts and techniques in our solution, like the concept of dimensions, in our case called versions.

Another approach to handle variability is the use of software product lines (SPLs) [7]. One of the best approaches to implement SPLs is feature models. Feature models [9] are also a way to represent software functionality by listing its features. In SPLs, multiple variations of the same software are produced, and feature models are used to specifying all those possible combinations of the product line. Also related to SPLs is the line of work of feature-oriented programming [14] where features can be composed in a flexible way. As for our work in particular, although we are not aiming to build feature models, we found 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.

Our approach introduces a novel programming model and language based mechanisms, which are orthogonal to that of component registries, used in middleware systems, to keep a global repository for versions of component interfaces. We provide flexible and lightweight solution, that allows versions to coexist without a middleware.

VI. Concluding Remarks

Developing and maintaining multiples versions of the same software is a challenging task. It is even more critical when several other applications depend on it as it is the case of applications based on web services (as most mobile applications are). In this paper we introduced a new programming model to develop and evolve software with multiple versions at the same time, while coping with modular evolution of components by means of flexible version compatibility modes. Our approach is flexible enough to not depend, and be parametric, on any version control system developers may use. The type system we developed can typecheck programs with several versions at the same time and the dispatching function dynamically decides what is the best version of a particular component for each request. As a proof of concept we have implemented our programming model as a prototype using Flow and JavaScript. As future work, formal artefacts that certify the soundness of the approach should be used as the validation mechanism. Also, experimental work can be performed with larger case-studies.

We envision the integration of this programming model into existing development tools, such as integrated development environments and version control systems, where a programmer should be able to visualise and edit a particular version independently, and hence provide a better developer experience [11]. A sophisticated visualisation and editing integration could, in principle, abolish the need for manual version annotations.

Acknowledgements

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-016718, by FCT projects ref. UID/CEC/04516/2013 and PTDC/EEI-CTP/4293/2014, and by the FLAD project ref. 233/2014.

References

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

[2] João Costa Seco and Luís Caires. Types for dynamic reconfiguration. In Peter Sestoft, editor, *15th European Symposium on Programming Programming Languages and Systems (ESOP 2006)*, pages 214–229. Springer, 2006.

[3] Miguel Domingues and João Costa Seco. Type Safe Evolution of Live Systems. In *Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*, 2015.

[4] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, 2011.

[5] Michael W. Godfrey and Daniel M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance*, pages 129–138, 2008.

[6] Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, and Juan José Amor. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[7] Software Engineering Institute. Software product lines. www.sei.cmu.edu/productlines, 2016-8-16.

[8] Piotr Kaminski, Marin Litoiu, and Hausi Müller. A design technique for evolving web services. In *Proceedings of the 2006 Conf. of the Center for Advanced Studies on Collaborative Research*, Riverton, NJ, USA, 2006. IBM Corp.

[9] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, SE Institute, Carnegie Mellon University, 1990.

[10] Jacek Kopecký, Karthik Gomadam, and Tomas Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. In *IEEE/WIC/ACM Int. Conf. Web Intelligence and Intelligent Agent Technology*, pages 619–625, 2008.

[11] Duc Le, Eric Walkingshaw, and Martin Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150, Sept 2011.

[12] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. End-to-end versioning support for web services. In *IEEE International Conference on Services Computing*, volume 1, pages 59–66, July 2008.

[13] Juergen Mangler, Peter Paul Beran, and Erich Schikuta. On the origin of services using RIDDL for description, evolution and composition of restful services. In *IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, pages 505–508, 2010.

[14] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[15] Daniele Romano and Martin Pinzger. Analyzing the evolution of web services using fine-grained changes. In Carole A. Goble, Peter P. Chen, and Jia Zhang, editors, *IEEE 19th Int. Conf. Web Services*, pages 392–399. IEEE CS, 2012.

[16] S. M. Sohan, C. Anslow, and F. Maurer. A case study of web API evolution. In *2015 IEEE World Congress on Services*, pages 245–252, June 2015.

[17] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *IEEE Int. Conf. on Soft. Analysis, Evolution, and Reengineering*, pages 138–147, 2017.