Extending OpenAPI 3.0 to Build Web Services from their Specification

David Sferruzza, Jérôme Rocheteau, Christian Attiogbé, Arnaud Lanoix

▶ **To cite this version:**

David Sferruzza, Jérôme Rocheteau, Christian Attiogbé, Arnaud Lanoix. Extending OpenAPI 3.0 to Build Web Services from their Specification. International Conference on Web Information Systems and Technologies, INSTICC, Sep 2018, Seville, Spain. hal-01868498

HAL Id: hal-01868498
https://hal.science/hal-01868498

Submitted on 5 Sep 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Extending OpenAPI 3.0 to Build Web Services from their Specification

David Sferruzza[1,3], Jérôme Rocheteau[1,2], Christian Attiogbé[1] and Arnaud Lanoix[1]
[1]*LS2N, UMR CNRS 6004, F-44322 Nantes Cedex 3, France*
[2]*ICAM, 35, avenue du Champ de Manœuvres, 44470 Carquefou, France*
[3]*Startup Palace, 18, rue Scribe, 44000 Nantes, France*
david.sferruzza@ls2n.fr, christian.attiogbe@ls2n.fr, arnaud.lanoix@ls2n.fr, jerome.rocheteau@icam.fr, david.sferruzza@startup-palace.com

Keywords: Software Engineering, Web Applications, Web Services, Model-Driven Engineering, OpenAPI 3.0.

Abstract: Web services are meant to be used by other programs. Developers (or other programs) need to understand how to interact with them, which means documentation is crucial. Some standards like OpenAPI define ways to document web services and target both humans and programs. Many tools can be used to help developers to work in a forward engineering process: they use hand-written OpenAPI models as input and automatically generate a skeleton of a working application, for example. However, this approach is not suitable to generate working applications if several evolutions occur over time, which often results in a misalignment between the OpenAPI model and the web services implementation. Here we show how we extend the OpenAPI 3.0 specification to allow building actual web services using a Model-Driven Engineering (MDE) approach. We extend the SWSG tool to make it possible to generate code from an extended OpenAPI model. This leverages a MDE approach to build web services from a model while benefiting from OpenAPI 3.0 tooling and ecosystem.

1 INTRODUCTION

Context. Web services often aim to back various web or mobile applications which deliver a service to end-users. Lots of different processes and tools can be used to build web services. Some of them fit well for rapid-prototyping whereas others were made with production-readiness in mind. The first must allow fast iteration loops while progressively building from a high-level to a low-level, and the latter must promote good stability, maintainability and performance. Both approaches can be interesting depending on the context; and some of them can fit both needs. For example, SWSG (Sferruzza et al. 2018), a framework to generate consistent web services for both prototyping and production, was developed in the context of *Startup Palace* (a web company). At the same time, it is important to design web services in a way that makes them actually usable. This implies providing a good documentation that can be used by developers writing consumer applications or by these consumer applications if they can adapt their behavior dynamically. While this kind of documentation can take many forms, some standards such as OpenAPI [1] and RAML [2] are widely used by the industry. These standards define formats of specifications that describe HTTP APIs of web services and are the base of ecosystems of tools. This paper focuses on OpenAPI 3.0 because it is the standard used at *Startup Palace*. Tools around OpenAPI can be used in two main ways. First, with a forward engineering process, developers create manually an OpenAPI model, and refine it using various tools. For example: one-time generation of an implementation skeleton in a given technology. Second, with a reverse engineering process, a tool is used to extract an OpenAPI model from a working implementation (which can be enhanced by annotations).

Motivation. Web companies like *Startup Palace* would benefit from using the first process because its top-down approach fits well in the activity of helping startups to launch their products step-by-step. While the second process is useful to document existing web services, it cannot leverage the benefits of building web service from a high-level (model) to a low-level

[1]https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md
[2]https://raml.org/

(implementation). Yet (DeRemer and Kron 1975) shows the advantages of using different languages for programming-in-the-large and for programming-in-the-small. The first process makes it possible to partially leverage these advantages, but lacks the ability to keep the OpenAPI model and the implementation aligned throughout the life of the project. Indeed, implementation is often obtained by a projection of the model that is then manually modified to implement business logic. Any following model evolution needs to be projected again which would require manual modifications to be re-applied from the beginning. This work is intended to go beyond this limitation.

Contribution. We extend the approach presented in (Sferruzza et al. 2018). It consisted of a meta-model designed to express a high-level representation of web services in order to provide support such as visualization, verification and code generation. We add the possibility to better specify service parameters, improve the type system and make re-usability easier by adding a mechanism to bind variables in component definitions and in their equivalents in instances' contexts.

We introduce extensions to OpenAPI 3.0 (Sferruzza 2018); we take advantage of them to make it possible to add the various information of our models of web services into any OpenAPI model. This allows us to merge our meta-model with one supported by the industry and to polish it by removing parts that already exist in OpenAPI while benefiting mutually of our SWSG tool or of any tool designed for OpenAPI 3.0.

We propose a prototype [3] to build consistent web services from an extended OpenAPI 3.0 model. This tool is an enhanced version of the tool we proposed in (Sferruzza et al. 2018), that could *(i)* check model consistency and *(ii)* generate source code of working web services from a given valid model. We improve model consistency verification and code generation, according to the recent evolutions of the meta-model. Then we add support for OpenAPI 3.0 models that use our extensions. The prototype converts these models to models that are compliant with our meta-model so they can benefit of our tooling.

The article is structured as follows. Section 2 presents related work. Section 3 shows, through an example project, how OpenAPI is usually used in the industry. Section 4 defines extensions to the OpenAPI 3.0 specification. Section 5 improves the meta-model presented in (Sferruzza et al. 2018). Section 6 presents a new process to use these extensions in SWSG and automatically generate web services from an OpenAPI model. Section 7 re-runs the example in Section 3 using the newly introduced tools. Finally, Section 8 concludes the article with some lessons and future work.

[3] https://gitlab.startup-palace.com/research/swsg

2 RELATED WORK

The use of MDE for development and automatic generation of web services or web applications is not a new topic (Bernardi, Cimitile, Di Lucca, et al. 2012; Bernardi, Cimitile, and Maggi 2016; Scheidgen, Efftinge, and Marticke 2016). Indeed, this work is itself built on top of the approach of SWSG (Sferruzza et al. 2018) and REIFIER (Rocheteau and Sferruzza 2016).

SWSG shares the meta-modeling approach with tools such as *M3D* (introduced in (Bernardi, Cimitile, Di Lucca, et al. 2012) and extended in (Bernardi, Cimitile, and Maggi 2016)) that also focus on building web services using MDE. One of the main differences between SWSG and *M3D* is that SWSG was developed with a focus on design-time support. For example, it allows to automatically verify properties about the structural consistency of models. Even if SWSG is definitely related to existing standards such as BPEL (Fu, Bultan, and Su 2004) or WSDL, our approach differs on several aspects. First, we want to avoid the shortcomings described in (Gronmo et al. 2004), that is WSDL models contain too much technical details and are difficult to understand for humans. Indeed, our meta-model is simpler and less expressive than WSDL or BPEL. Second, this allows SWSG to provide more support to users (the balance between flexibility and support is discussed in (van der Aalst, Pesic, and Schonenberg 2009)). Finally, SWSG now relies on OpenAPI.

OpenAPI is an active and growing industry standard that is also involved in various research areas. In (Cao, Falleri, and Blanc 2017), it was chosen for its popularity over WADL and other industry standards in order to automatically transform plain HTML documentations of web services to a machine-readable format. Moreover, (Cremaschi and De Paoli 2017) provides a great state of the art of service description formats (which seems to be an updated version of the work proposed in (Tsouroplis et al. 2015, §3.2)) that brings out OpenAPI as the most promising choice at the moment and enriches it with semantic annotations. (Schwichtenberg, Gerth, and Engels 2017) shows an approach that is agnostic to service description formats but uses OpenAPI in the article. The popularity of OpenAPI is also highlighted by its use in other do-

mains. For example (Willighagen and Mélius 2017) describes a case where OpenAPI 2.0 is used in combination of other tools from the life sciences community and points out that the specification extension mechanism of OpenAPI 3.0 (that we use in this article) might be an interesting opportunity of improvement. Another example in the telecommunication domain is presented in (Pugaczewski et al. 2017) which provides a section to emphasizes the trade-offs of Model-Driven Engineering and argues that they can be overcomed "with increased investment in the tools that support the development process"; we share the same vision.

3 A COMMON USAGE OF OpenAPI

The OpenAPI Specification defines a standard to express interfaces to HTTP APIs in a language-agnostic way. It aims at allowing "both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection": that is a meta-model. As in MDE, the point of having meta-models is to have tools that can rely on them in order to safely manipulate models and offer support to developers. Indeed, an ecosystem of tools was developed around OpenAPI by various actors. Such tools have several purposes, including but not limited to: providing an interactive graphical user interface from a model [4][5], generating functional tests from a model [6] or generating a model from an annotated implementation [7].

To ease the development of these tools, official examples of OpenAPI models are shipped with the specification. In this article, we focus on the *Petstore* example [8]. It describes a simple application that exposes 4 services to list, show, add and remove data records representing animals. We assume that we are in the context of a company such as *Startup Palace*; that means the goal is to develop these web services in order for them to be consumed by user interface applications; for example, desktop and mobile applications for the owner of the store.

The common top-down development process follows. First, developers make several iterations on writing an OpenAPI model. This model must match the functional specifications and describe web services that are fully exploitable by consumer applications. For example, the description of one of the Petstore services is shown in Listing 1.

```
/pets/{id}:
  get:
    description: Returns a user based on a single ID,
    ↪  if the user does not have access to the pet
    operationId: find pet by id
    parameters:
      - name: id
        in: path
        description: ID of pet to fetch
        required: true
        schema:
          type: integer
          format: int64
    responses:
      '200':
        description: pet response
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
```

Listing 1: A Service in the Petstore Example.

When this model is stable enough, developers can use it as a specification to start building web services and consumer applications. There are some kinds of tools that can take the OpenAPI model as input and help to build compliant web services. For example, by generating a skeleton of an application using a given technological stack [9], or by generating automated tests that can be used to check if the web services are compliant [10]. But these tools have a major lack: they require humans to update the OpenAPI model whenever the web services evolve. This is a very common situation: specifications must evolve either because business requirements have changed or new constraints have been discovered while developing. Even if some tools can mitigate this issue, it is likely that developers will eventually stop maintaining the OpenAPI model after the web services reach production, making the two diverge over time. In long-term projects, this means giving up on every advantage provided by OpenAPI and its MDE approach.

To fix this shortcoming, we present in Section 6 an improved version of this process that leverages a meta-model of web services we introduced in (Sferruzza et al. 2018). In order for this new approach to be feasible, we first extend the OpenAPI 3.0 Specification in Section 4 and improve our meta-model in Section 5.

[4]https://github.com/swagger-api/swagger-editor
[5]https://github.com/swagger-api/swagger-UI
[6]https://github.com/apiaryio/dredd
[7]https://github.com/vanderlee/PHPSwaggerGen
[8]https://github.com/OAI/OpenAPI-Specification/blob/3.0.1/examples/v3.0/petstore-expanded.yaml
[9]https://github.com/pmlopes/slush-vertx
[10]https://github.com/apiaryio/dredd

4 EXTENDING OpenAPI 3.0

The OpenAPI Specification describes a meta-model to express an interface to web services. Therefore, it does not describe how web services are implemented. To make our approach possible, we propose a way to merge OpenAPI with our meta-model of web services introduced in (Sferruzza et al. 2018).

To preserve tools compatibility we make use of *Specification Extensions*, as defined in OpenAPI 3.0. This mechanism allows to add data to models without breaking their compliance to the specification or their ability to be used by tools designed to be compatible with it. In (Sferruzza 2018) we present our extensions to OpenAPI 3.0. The following paragraphs discuss the nature of these extensions for each aspect of our meta-model.

Because an OpenAPI model can be seen as a tree (before references are resolved) that has the `OpenAPI` object as root, we make use of the notion of *paths*. The `OpenAPI > components` path designate the child of the root named `components`. Sub-children are separated using a > symbol; the `OpenAPI > components > requestBodies` designates the `requestBodies` child of the `components` child of the root.

Data Model. An OpenAPI 3.0 model can contain a set of `Schema` objects[11]. A `Schema` object defines a data type for input or output data, which can then be referenced from elsewhere in the model. This mechanism provides more expressiveness than ours and was made for the same purpose. Thus it is a good fit for the *entities* of our meta-model.

Processes. On purpose, there are no equivalent of our component system in OpenAPI 3.0, because it describes processes that are internal to the web services which is out of OpenAPI's scope. Accordingly we add two properties in the `Components` object of OpenAPI. They contain sets of atomic and composite component definitions[12]. Exact schemas of these components follow on from their definition in our meta-model.

Services. Describing services is the main feature of OpenAPI. As such they can be specified in a quite expressive way. Yet the service meta-model in OpenAPI is not a superset of ours because of two lacks that require it to be extended. First, each service must be associated to a component instance that describes its behavior[13]. Second, if a service has a `requestBody` property (that describes the type of the data required in the request body), the `RequestBody` object must contain a variable name[14]. When generating the web services code, this is used to include the request body contents in a variable of the execution context.

[11] In the `OpenAPI > components > schemas` path.

[12] Their paths are `OpenAPI > components > x-swsg-ac` and `OpenAPI > components > x-swsg-cc`.

5 EXTENDING A META-MODEL OF WEB SERVICES

Notations. A tuple T is a product type between n types T_1 to T_n, with $n \geq 2$. It is denoted $T \equiv T_1 \times \ldots \times T_n$. A value t of type T is written as $t = (t_1, \ldots, t_n)$ where $t_1 \in T_1, \ldots, t_n \in T_n$.

A record R is a tuple with labeled elements. It is denoted $R \equiv \langle label_1 : T_1, \ldots, label_n : T_n \rangle$. It is syntactic sugar over a tuple $T_1 \times \ldots \times T_n$ and n functions $label_1 : R \to T_1, \ldots, label_n : R \to T_n$. A value r of type R is written as $r = (t_1, \ldots, t_n)$ where $t_1 \in T_1, \ldots, t_n \in T_n$. Associated functions can also be written $r.label_1, \ldots, r.label_n$.

$\mathscr{P}(T)$ is the type of sets of elements of type T.

In (Sferruzza et al. 2018) we presented a meta-model of web services. This meta-model has a voluntarily simple design in order to provide two advantages: *(i)* to give developers good abstractions to write reusable code while giving them a good flexibility and *(ii)* to allow tools to provide support to developers, such as design-time consistency verification. It is a combination of three elements: a set of entities that stands for the data model, a set of components that stands for the process model and an ordered list of services that exposes components to the outer world. It is defined by $M \equiv \langle entities : \mathscr{P}(E), components : \mathscr{P}(C), services : List(S) \rangle$[15]. We identified some design issues and we propose evolutions to fix them.

Weakness of Type System. Some parts of our meta-model describe a type system. This type system is used to define entities' attributes and components' contracts. Valid types are defined by the union of a set of predefined scalar types (such a *String* or *Boolean*) and the parametrized type $EntityRef(E)$ where E is the name of an entity declared in the current model. In

[13] In the `x-swsg-ci` property of the service.

[14] In a `x-swsg-name` property.

[15] $List(T)$ designates a list of elements of type T; it is better introduced in (Sferruzza et al. 2018, §2.1).

order to deal with realistic use cases, we add two new parametrized types:

SeqOf(T). Represents an ordered list of elements of any valid type T.

OptionOf(T'). Represents an optional value of type T, that is every value of type T plus a special `null` value that represent the absence of value.

From now on, E' denotes the entities that make use of this enhanced type system.

Restricted Component Reusability. When defining an atomic component in a model, one must provide its contract: three sets that contain variables required on the execution context, variables that will be added and variables that will be removed. The names of these variables are then used in the component implementation. A problem appears when instantiating such a component (for example in a composite component definition): variables in the component's contract must have the same name in every execution context it is instantiated in. For example, it is possible to instantiate an atomic component $c_1 = (\text{"}atomic\text{"}, \emptyset, \emptyset, \{(\text{"}v1\text{"}, String)\}, \emptyset)$ twice in a row in a composite component $c_2 = (\text{"}composite\text{"}, \emptyset, [(\text{"}atomic\text{"}, \emptyset), (\text{"}atomic\text{"}, \emptyset)])$. But, because this component adds a variable to the execution context, this will violate a verification rule. Indeed the second instance cannot add an already defined variable to the execution context. To fix this, we introduce an *alias* property of type $\mathscr{P}(\langle source : String, target : String\rangle)$ to the component instance record (previously denoted by $CI \equiv \langle component : C, bindings : \mathscr{P}(\langle param : V, argument : Term\rangle)\rangle$). The *source* name designates a variable in the component definition, and *target* is the name it should take in this instance's context only. This makes it possible to define and implement components independently from their instantiation contexts. This results in $CI' \equiv \langle component : C', bindings : \mathscr{P}(\langle param : V, argument : Term\rangle), aliases : \mathscr{P}(\langle source : String, target : String\rangle)\rangle$ where CI' is the updated component instances and C' the set of components that uses them. Both of them use the enhanced type system. Using this mechanism, c_2 becomes $c_2' = (\text{"}composite\text{"}, \emptyset, [(\text{"}atomic\text{"}, \emptyset, \emptyset), (\text{"}atomic\text{"}, \emptyset, \{(\text{"}v1\text{"}, \text{"}v2\text{"})\})])$; there won't be a verification error anymore, the resulting execution context will contain `v1` and `v2` and the definition of c_1 remains the same.

Unexpressive Service Parameters. In a service definition, it is possible to specify parameters that will be extracted from the HTTP request's URL at runtime. The resulting variables are then injected in the execution context so that components can access them. We change the type of the *params* attribute of the service record from $\mathscr{P}(V)$ to $\mathscr{P}(\langle location : L, variable : V\rangle)$ where $L \equiv \{Query, Header, Path, Cookie, Body\}$. This makes it possible to extract parameters' values from various parts of the HTTP request which is often necessary in real world services. Now on, S' denotes the services that integrate these modifications and make use of CI' component instances and the enhanced type system.

The resulting meta-model $M' \equiv \langle entities : \mathscr{P}(E'), components : \mathscr{P}(C'), services : List(S')\rangle$ is defined by the BNF grammar in Figure 1.

6 AN EXTENSION OF SWSG TO BUILD WEB SERVICES

To support the approach of automatically building web services from an OpenAPI model, we extend the SWSG tool introduced in (Sferruzza et al. 2018). First, its internal meta-model and the corresponding concrete syntax are updated to match the changes presented in Section 5. Second, an OpenAPI 3.0 parser is implemented; it is designed to work with the extensions presented in Section 4. Finally, a custom model-to-model transformation is implemented in order to convert extended OpenAPI models to SWSG models. The process that follows on from these improvements is illustrated by Figure 2; it is an extended version of (Sferruzza et al. 2018, Figure 2).

Transformation of OpenAPI Models. Transforming extended OpenAPI models to SWSG models is quite straightforward, except for schemas/types. OpenAPI defines some primitive types and relies on a modified version of the *JSON Schema Specification* [16] for complex types. There are two issues. First, the *JSON Schema Specification* is more expressive than SWSG's type system. For example, it allows to define refined types, e.g. to add a minimum length to a string. Second, it supports both literal and referenced definition of attributes' types. Moreover, references are quite expressive and can target many places in the OpenAPI main document or even in another one. In comparison, SWSG only supports literal primitive types or references to other *entities*.

While the second issue is more an engineering problem, the first would require to improve SWSG's

[16] https://tools.ietf.org/html/draft-wright-json-schema-00

model	::=	⟨*entities*: entity*, *components*: component*, *services*: service*⟩
identifier	::=	`[A-Za-z][A-Za-z0-9_]*`
entity	::=	⟨*name*: identifier, *attributes*: variable*⟩
term	::=	variable | constant
variable	::=	⟨*name*: string, *type*: type⟩
constant	::=	⟨*type*: type, *value*: object⟩
type	::=	string | boolean | integer | float | date | datetime | entity-ref | seq-of | option-of
entity-ref	::=	⟨*entity*: identifier⟩
seq-of	::=	⟨*seqOf*: type⟩
option-of	::=	⟨*optionOf*: type⟩
component	::=	atomic-component | composite-component
atomic-component	::=	⟨*name*: identifier, *params*: variable*, *pre*: variable*, *add*: variable*, *rem*: variable*⟩
composite-component	::=	⟨*name*: identifier, *params*: variable*, *components*: component-instance*⟩
component-instance	::=	⟨*component*: identifier, *bindings*: binding*, *aliases*: alias*⟩
binding	::=	⟨*param*: variable, *argument*: term⟩
alias	::=	⟨*source*: variable, *target*: variable⟩
service	::=	⟨*method*: method, *path*: path, *params*: service-parameter*, *component*: component-instance⟩
method	::=	`[A-Z]+`
path	::=	`.+`
service-parameter	::=	⟨*location*: parameter-location, *variable*: variable⟩
parameter-location	::=	query | header | path | cookie | body

Figure 1: BNF Grammar of the extended Meta-Model of Web Services.

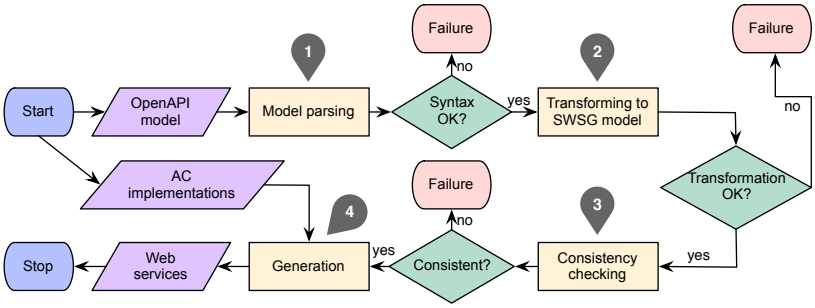

Figure 2: Process of the Updated Prototype.

type system in order for it to support expressing every possible OpenAPI type. Yet, we choose not to address them for now because they are not essential to test and validate our approach. Therefore our prototype might return errors when working with some OpenAPI models that contain unsupported types or references in schemas.

Code Generation. The process defined by Figure 2 is generic: it does not rely on a specific language or technology. Yet the language and technologies used to implement atomic components must be identical or compatible with those of the code generation target. Because we experiment in *Startup Palace*'s context, our prototype targets the PHP programming language [17] with the *Laravel* web framework [18], which is a common tool stack.

[17]https://php.net/

[18]https://laravel.com/

Code generation is very similar to the one presented in (Sferruzza et al. 2018). The generated code is not meant to be manually edited, but can be easily integrated in a manually-developed *Laravel* application. Because this MDE approach was designed to allow shallow consistency verification, most inconsistencies in the model are caught at compile-time. This does not prevent developers to create flawed applications, as they have full control on the atomic components implementations. Indeed this flexibility comes at the cost of a bit of support.

7 CASE STUDY AND ASSESSMENT

We derive a new process from the common process explained in Section 3. Step 1: design a stable OpenAPI model. Step 2: developers need to design SWSG com-

ponents that will implement their services and use the extensions we introduced in Section 4 to write them inside the OpenAPI model. Every atomic component defined in the model must be provided with an implementation. Step 3: SWSG can check the model and generate working web services if the verification is successful.

We re-run the example presented in Section 3 with this new process. The example service defined by Listing 1 is a part of a standard OpenAPI model. We need a component that will handle the response generation when this service will receive requests. We create a composite component called `FindPet` and reference it from the service, as shown in Listing 2. This composite component has two children that are atomic components. The first takes an ID as input, uses it to query the database and adds the `Pet` result to the context. The second takes a `Pet`, serializes it in JSON and put it in an HTTP response. These three components are defined in Listing 3. Implementations are written for every atomic components. Listing 4 shows the implementation of the `GetPetById` component as an example[19].

```
/pets/{id}:
  get:
    x-swsg-ci:
      component: FindPet
```

Listing 2: Instantiating a Component from a Service.

We get a *PreconditionError* when we run SWSG on these inputs. This verification error indicates that a component's precondition is not fulfilled in a given instantiation context. In the current case, we learn that the `GetPetById` misses a string named `id` when instantiated by the `FindPet` component in the `GET /pet/{id}` service. Indeed, we voluntarily introduced an error in Listing 3: the `GetPetById` component is given an integer (by the service) whereas it requires a string. In this particular example, it should require an integer `id` variable in order to be consistent with the service parameter. Nevertheless, in more complex projects, this component might have been used inside several other composite components and services. Thus, it might not be a good solution to just change the component's definition because it might break other workflows. This is the kind of mistakes

[19]The PHP class in Listing 4 depends on the `Component` interface and on the `Ctx` and `Params` classes. They are defined in code output by the code generator and are just implementation details of the SWSG specification in this specific code generator. Different code generators could require different constraints on implementations of atomic components.

```
components:
  x-swsg-cc:
    - name: FindPet
      components:
        - component: GetPetById
        - component: RenderPet
  x-swsg-ac:
    - name: RenderPet
      pre:
        - name: pet
          type:
            entity: Pet
    - name: GetPetById
      pre:
        - name: id
          type: String
      add:
        - name: pet
          type:
            entity: Pet
```

Listing 3: Components in the SWSG Petstore Example.

```
<?php
namespace App\Components;
use App\SWSG\Component, App\SWSG\Ctx, App\SWSG\Params,
↪  DB;

class GetPetById implements Component {
    public static function execute(Params $params, Ctx
    ↪  $ctx) {
        $pet = DB::table('pet')->where('id',
        ↪  $ctx->get('id'))->first();
        $ctx->add('pet', $pet);
        return $ctx;
    }
}
```

Listing 4: Implementation of the `GetPetById` Atomic Component.

SWSG can prevent us to make: because they are reported very early at compile-time, instead of runtime which is too late. Developers can study the problem and decide if they have to build a better implementation or if the process model was badly designed. With the contributions of this article, the decision can even escalate to whether the OpenAPI design has flaws or not, because SWSG guarantes that the process model and the implementations are aligned with the OpenAPI model altogether.

The model and code of this case study are available in the repository of SWSG[20].

[20]https://gitlab.startup-palace.com/research/swsg/tree/master/examples/petstore

8 CONCLUSION

We refined the meta-model presented in (Sferruzza et al. 2018) in order to improve its expressiveness and fix shortcomings with the reusability of some elements it allows to define. We also introduced extensions to the OpenAPI 3.0 Specification and merged the previous meta-model into OpenAPI. This allowed web services developers to define implementations of web services using MDE, starting from the corresponding high-level contract as expressed by a standard OpenAPI model. Evolutions were made to the SWSG tool so that it could support and automate this process. In addition to the previous advantages of SWSG, this made easier to write a model of web services and to keep it aligned with the implementation.

Even if the type system was improved, it could be improved further and allow subtyping in component preconditions, for example. Model composition is theoretically handled by OpenAPI but not currently supported by SWSG. The whole approach still needs better evaluation on more realistic (by nature and by size) case studies. Finally, continuous evaluation remains beneficial, and SWSG could still be improved to be able to automatically check the compliance of atomic components to their contract in the model. This could allow developers to focus on less tedious work that adds direct value to their products.

REFERENCES

Bernardi, Mario Luca, Marta Cimitile, Giuseppe Di Lucca, et al. (2012). "M3D: A Tool for the Model Driven Development of Web Applications". In: *Proceedings of the Twelfth International Workshop on Web Information and Data Management*. WIDM 2012. USA, pp. 73–80.

Bernardi, Mario Luca, Marta Cimitile, and Fabrizio Maria Maggi (2016). "Automated Development of Constraint-Driven Web Applications". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, pp. 1196–1203.

Cao, Hanyang, Jean-Rémy Falleri, and Xavier Blanc (2017). "Automated Generation of REST API Specification from Plain HTML Documentation". In: *International Conference on Service-Oriented Computing*. Springer, pp. 453–461.

Cremaschi, Marco and Flavio De Paoli (2017). "Toward Automatic Semantic API Descriptions to Support Services Composition". In: *European Conference on Service-Oriented and Cloud Computing*. Springer, pp. 159–167.

DeRemer, Frank and Hans Kron (1975). "Programming-in-the Large versus Programming-in-the-Small". In: *ACM Sigplan Notices*. Vol. 10. ACM, pp. 114–121.

Fu, Xiang, Tevfik Bultan, and Jianwen Su (2004). "Analysis of Interacting BPEL Web Services". In: *In Proc. 13th Int. World Wide Web Conf.* Citeseer.

Gronmo, Roy et al. (2004). "Model-Driven Web Services Development". In: *E-Technology, e-Commerce and e-Service*. EEE'04. IEEE.

Pugaczewski, Jack et al. (2017). "Software Engineering Methodology for Development of APIs for Network Management Using the MEF LSO Framework". In: *IEEE Communications Standards* 1.1, pp. 92–96.

Rocheteau, Jérôme and David Sferruzza (Oct. 5, 2016). "Reifier: Model-Driven Engineering of Component-Based and Service-Oriented JEE Applications". In: ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. Saint-Malo, France.

Scheidgen, Markus, Sven Efftinge, and Frederik Marticke (2016). "Metamodeling vs Metaprogramming: A Case Study on Developing Client Libraries for REST APIs". In: *European Conference on Modelling Foundations and Applications*. Springer, pp. 205–216.

Schwichtenberg, Simon, Christian Gerth, and Gregor Engels (2017). "From Open API to Semantic Specifications and Code Adapters". In: *Web Services (ICWS), 2017 IEEE International Conference On*. IEEE, pp. 484–491.

Sferruzza, David (2018). *Specification of SWSG Extensions for OpenAPI*. URL: https://gitlab.startup-palace.com/research/swsg/tree/master/openapi-extensions-specification/1.0.0.md.

Sferruzza, David et al. (Jan. 23, 2018). "A Model-Driven Method for Fast Building Consistent Web Services in Practice". In: 6th International Conference on Model-Driven Engineering and Software Development. Funchal, Madeira, Portugal.

Tsouroplis, Romanos et al. (2015). "Community-Based API Builder to Manage APIs and Their Connections with Cloud-Based Services." In: *CAiSE Forum*, pp. 17–23.

Van der Aalst, Wil M.P., Maja Pesic, and Helen Schonenberg (2009). "Declarative Workflows: Balancing between Flexibility and Support". In: *Computer Science-Research and Development* 23.2, pp. 99–113.

Willighagen, Egon and Jonathan Mélius (2017). "Automatic OpenAPI to Bio.Tools Conversion". In: *bioRxiv*. DOI: 10.1101/170274.