A Model-Driven Method for Fast Building Consistent Web Services in Practice

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*

Keywords: Software Engineering, Web Applications, Web Services, Model-driven Engineering, Formal Verifications.

Abstract: Nowadays, lots of software companies rely on web technologies to test market hypothesis and develop viable businesses. They often need to quickly build web services that are at the core of their Minimum Viable Products (MVPs). MVPs must be reliable and are based on specifications and hypothesis that are likely to change. Model Driven Engineering approaches have been proposed and used to develop and evolve web services. However, these approaches lack the ability to be suitable for both *(i)* rapid prototyping, *(ii)* model verification and *(iii)* compatibility with common programming languages. Here we propose a meta-model to express web services and the related tool to verify models consistency. We adopt a shallow verification process to allow rapid prototyping by developers who are not formal methods experts, while still offering design-time guarantees that improve product quality and development efficiency. Web services are defined using parametric components which enable to express and formally verify web service patterns and to safely reuse them in other contexts. We built a tool to check consistency of models and associated components implementations in order to generate corresponding web services. This allows us to give flexibility to developers, as well as verification support and an easier onboarding for new developers.

1 INTRODUCTION

Context. Web agencies are software companies that often work with their customers to help them develop new projects involving the web. Most of these customers need web applications to bring value to their own customers. These web applications are built iteratively to limit their cost while allowing startups to converge toward a viable market. This approach begins by building a Minimum Viable Product (MVP). In this context, it is a web application with a high level of quality but a limited set of features. MVPs (among other kinds of web applications) need to be functional, reliable, usable and designed with users' emotions in mind, with only the features required to test market hypotheses. In this domain, it is very common for agencies and IT companies to be paid a lump sum which is negotiated with customers before projects begin. **Startup Palace**, a web agency that evolves in this context, is investigating new ways of building web applications for several years.

Motivation. Software companies, and more specifically web agencies, are always in need of new approaches to reduce time-to-market while ensuring quality of web applications and profits. When developing a MVP it is important to focus on features that really bring value to users, also called *game-changers*. Other features, called *show-stoppers*, do not bring value directly but are required to make the application functional, reliable or usable. With that in mind, lump sum payments imply that if a project takes too much time to reach the expected (and sold) level of quality and features, companies will have to reduce their profits. *Show-stopper* features are often tedious to implement and so error-prone. Furthermore, because of the iterative process and the purpose of MVPs, specifications are likely to evolve, which can introduce bugs. This means that developers need abstractions to be able to safely express, isolate, reuse and evolve behaviors. Some programming languages do provide such abstractions, along with modern type checkers that are able to statically verify consistency of programs. But this is not practicable in our context be-

Sferruzza, D., Rocheteau, J., Attiogbé, C. and Lanoix, A.
A Model-Driven Method for Fast Building Consistent Web Services in Practice.
DOI: 10.5220/0006531900150024
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 15-24
ISBN: 978-989-758-283-7

cause we would like to leverage existing expertises of developers instead of forcing them to learn a new programming language and its ecosystem from scratch. While these motivations relate to web applications, they also apply to web services development that is addressed in this article. Web services are applications provided by a server, which are communicating with other applications using the HTTP protocol through the network. Lots of web applications rely on web services as they allow to separate UI from core features and data management, which is useful, for example, when having several UIs (like in mobile applications).

Contribution. This work is an attempt to solve the problem of building web services using safe abstractions on top of an existing programming language in order to ease development and reuse of *show-stopper* features.

We introduce a meta-model to express web services and a corresponding semantics for verification. This leverages existing theory in Model Driven Engineering (MDE) (Bernardi, Cimitile, Di Lucca, et al. 2012; Rocheteau and Sferruzza 2016; Scheidgen, Efftinge, and Marticke 2016) and a component-based approach in order to provide an expressive and language-agnostic solution to build web services. This meta-model does not allow to completely specify web services but is rather limited to a high-level representation in order to provide support while keeping models simple.

We propose a tool (Sferruzza 2017) *(i)* to check models consistency, *(ii)* to generate working web services from a given valid model. For example, every component preconditions must be fulfilled in their instance contexts in order for a model to be consistent. This mechanism, coupled with the consistency checking, gives developers the ability to quickly and safely write and use components. But above all they can reuse existing components in a reliable way. This mechanism, coupled with the consistency checking, gives developers means to quickly and safely write and use components, and to reuse them in a reliable way.

The article is structured as follows. Section 2 describes the meta-model of web services. Section 3 defines consistency of compliant models and shows how they can be checked. Section 4 introduces a tool to generate actual implementations from models. Section 5 shows a case study that illustrates our approach. Section 6 presents related work. Finally, Section 7 concludes the article with some lessons and future work.

2 A META-MODEL TO EXPRESS WEB SERVICES

We introduce a meta-model of web services. This meta-model is voluntarily simple 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 (see Section 3).

2.1 Notations

The union or sum type of two types T_1 and T_2 is denoted $T_1 \uplus T_2$.

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 $t_1 \in T_1$, $\ldots$, $t_n \in T_n$. The notation $t(x)$ is also used to designate t_x, where $x \in \{1, \ldots, n\}$.

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 $label_1 : R \to T_1$, $\ldots$, $label_n : R \to T_n$. As for tuples, 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 $r.label_1$, $\ldots$, $r.label_n$. For example, for a record $person = (\text{"}Batman\text{"}, 35) \in \langle name : String, age : Int \rangle$ we have $name(person) = person.name = \text{"}Batman\text{"} \in String$.

A set whose elements are all of type T has the type $\mathscr{P}(T)$.

A sequence or list of type T is a set of elements of type T which are listed in a specific order. It is denoted $List(T)$. It is similar to a function from indexes I to values of T, that is $List(T) : I \to T$ where I is a $1..N$ and $N = card(List(T))$. It can also be written as $List(T) \equiv [t_1, \ldots, t_n]$ where $t_1, \ldots, t_n \in T$.

The projection of a set or list of tuples S on the i^{th} element of a tuple is written $Prj_i(S)$. Similarly, the projection of a set or list of records S on one of the elements of a record $label_i$ is written $Prj_{label_i}(S)$. For example, for a set of records $S \in \mathscr{P}(\langle name : String, age : Int \rangle)$ where $S = \{(\text{"}Batman\text{"}, 35), (\text{"}Robin\text{"}, 26)\}$, $Prj_{name}(S) = \{\text{"}Batman\text{"}, \text{"}Robin\text{"}\} \in \mathscr{P}(String)$.

2.2 The Meta-model

A meta-model of web services is defined by the UML class diagram in Figure 1. It does not aim to replace existing standard meta-models like for instance the OpenAPI Specification (Open API Initiative and The Linux Foundation 2017) or RAML (RAML Workgroup 2017), but rather to be compatible and com-

plementary with them. These models allow to define programming language-agnostic interface descriptions for HTTP APIs (i.e. informal contracts) in order to be able for both humans and computers to discover and understand their capabilities, while our meta-model allows to express actual implementations of such HTTP APIs. It is designed in order to match needs and requirements about verification (see Section 3), generation and ease of writing (see Section 4). For this reason, and to be more accessible to practitioners, our approach does not rely on existing standards such as BPEL (Fu, Bultan, and Su 2004) or WSDL (Gronmo et al. 2004).

A model of web services $m_i \in M$ is specified as a record of three elements: a set of entities of type E that stands for the data model, a set of components of type C that stands for the process model and an ordered list of services of type S that exposes component to the outer world (see Formula (1)).

$$M \equiv \langle entities : \mathscr{P}(E), components : \mathscr{P}(C), services : List(S) \rangle \quad (1)$$

Entities are non-primitive data types. An entity $e_i \in E$ is represented by a record of two elements: a name and a set of variables that represent attributes: $E \equiv \langle name : String, attributes : \mathscr{P}(V) \rangle$. A variable $v_i \in V$ is defined by a record composed of a name and a type: $V \equiv \langle name : String, type : T \rangle$. An attribute of an entity can be another entity, making it a recursive type.

Components are units of processes and computations that occur inside web services. Their execution happens in an isolated context, that can contain variables. They can mutate this context by adding and removing variables. A component $c_i \in C$ is defined as the union type of atomic components AC and composite components CC: $C \triangleq AC \uplus CC$. Both types of components are defined by a name and a set of variables that express components' parameters. Because they have parameters, we call them parametric components. An atomic component $ac_i \in AC$ is represented by a record of the following elements: name, parameters, preconditions (a set of variables that might be needed in the execution context), additions (a set of variables that will be added to the execution context) and removals (a set of variables that will be removed from the execution context): $AC \equiv \langle name : String, params : \mathscr{P}(V), pre : \mathscr{P}(V), add : \mathscr{P}(V), rem : \mathscr{P}(V) \rangle$. A composite component $cc_i \in CC$ is represented by a record of the following elements: name, parameters and an ordered list of component instances: $CC \equiv \langle name : String, params : \mathscr{P}(V), components : List(CI) \rangle$. A component instance $ci_i \in CI$ is represented by a record of two elements: a component and a set of bindings used to instantiate the component by associating arguments to its parameters: $CI \equiv \langle component : C, bindings : \mathscr{P}(\langle param : V, argument : Term \rangle) \rangle$. Terms can be variables or literal values: $Term \triangleq V \uplus Value$. Atomic components are meant to be along with an implementation written using a programming language whereas composite components are not. Components can be seen as an abstraction to encourage separation of concerns and reusability by leveraging two mechanisms: composition and parametrization.

Services are the entry points of web services. A service $s_i \in S$ is represented by a record of four elements: a HTTP method, a URL pattern, a set of variables that represent expected input parameters and a component instance: $S \equiv \langle method : String, url : String, params : \mathscr{P}(V), component : CI \rangle$. URL patterns are regular expressions with named capturing groups. It is easy to use one of these regular expressions to check if it matches the URL part of an incoming HTTP request, or to do a projection to get a set of expected parameters (corresponding to the named capturing groups). In a model of web services, services are gathered in an ordered list. This abstraction is very common in web frameworks and is often called *router*. Instead of considering a web application as a huge function of HTTP requests to HTTP responses, a router allows to dispatch HTTP requests to several such functions by filtering them declaratively by method and by URL pattern. That is to reduce complexity of the whole application by encouraging separation of concerns.

2.3 Concrete Syntax for Models of Web Services

We presented a mathematical definition of our meta-model in Section 2.2 and an equivalent class diagram in Figure 1. These notations are meant to introduce formal definitions that are used to define properties on the meta-model, which we do in Section 3. But they are cumbersome to read or write actual models.

We introduce a concrete syntax that is more compact and readable. Because it is equivalent to diagram in Figure 1, we won't define it formally here but only provide some intuition on it.

A model is represented, with respect to its formal definition, as an unordered list of definitions, each on its own lines. A definition starts with an element identifier: `e` for entity, `s` for service, `ac` for atomic component and `cc` for composite component. Element's properties are placed on their own indented

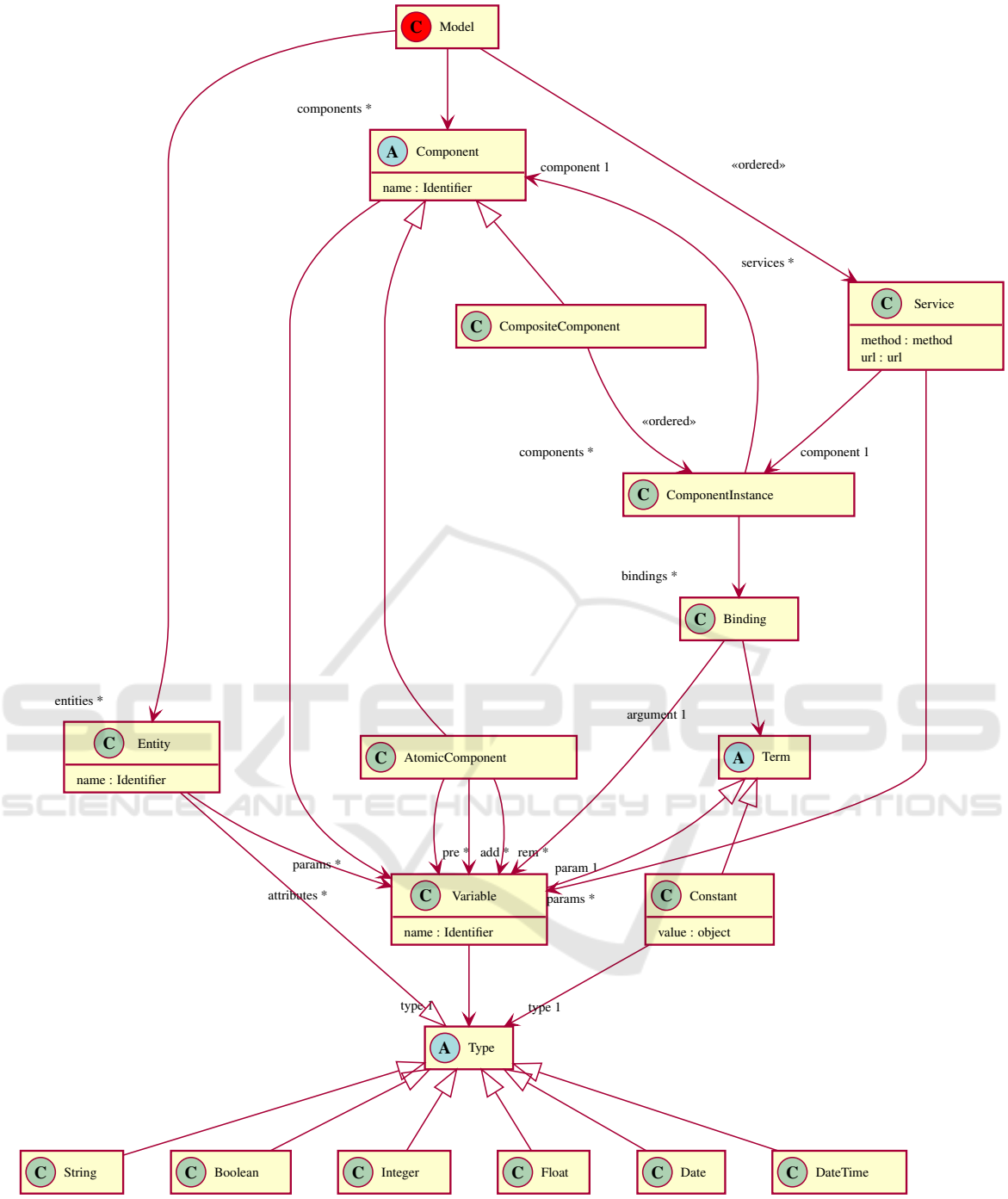

Figure 1: Meta-model diagram.

line and prefixed with the property name (or a short alias). Here is an example of a service declaration.

```
s
  method POST
  url \/getName\/(?<email>[^\/]+)
  params (email: String)
  ci GetName
```

For readability reasons, every item of a composite component list of component instances can be written on its own line (see examples in Section 5). The concrete syntax of the other structures of the meta-model is defined in a similar way.

2.4 Evaluating a Model of Web Services

Our meta-model gives helpful abstractions to develop web services but the built models need a rigorous evaluation semantics. Here is how web services based on an instance of this meta-model could handle incoming HTTP requests.

Routing. First, the application receives a HTTP request. Its list of services is sequentially scanned until a service matches the request; that is, the HTTP method is the same and the URL matches the pattern. If no service matches, then a static `404` HTTP response is sent back.

Flattening. Then, the component instance contained in the service is reduced to a flattened ordered list of instances of atomic components: instances of composite components are recursively replaced by their subcomponents. We define by cases a function $flatten(c)$ that flattens a given component:

$$\begin{aligned}
&\text{if } c \in AC, \text{ then } flatten(c) = [c] \\
&\text{if } c \in CC, \text{ then } flatten(c) = \bigcup_{ci \in c.components} flatten(ci) \\
&\text{if } c \in CI, \text{ then } flatten(c) = flatten(c.component)
\end{aligned} \tag{2}$$

Evaluating Components. An initial evaluation context is created by extracting parameters (if present) from the request URL and putting them into an empty context. This flattened list is then evaluated: every atomic component is executed given the previous context as an input and produces a new context as an output. This behavior is very similar to state monads (see § 2.5 in (Wadler 1992)).

Responding. Finally, a HTTP response is built. There are two cases to consider. If one of the evaluated components returned a HTTP response instead of a new context, the following components are not evaluated and this response is returned to the client. Otherwise, the context is serialized and encapsulated into a HTTP response of code `200`.

3 CONSISTENCY OF WEB SERVICES

Section 2 showed that our web services meta-model uses components as an abstraction to improve separation of concerns and reusability. In order to allow developers to safely use this abstraction we propose a way to do verification of models. This verification checks if a model is consistent. It can happen at *design-time* – outside any evaluation context – so that inconsistent models won't be run in production.

Definition. A model of web services $m \in M$ is consistent if it verifies the following properties identified by Formulas (3) to (13).

Component Name Uniqueness. Every component in a model has a unique name:

$$\begin{aligned}
&\forall c \in m.components, \forall c' \in m.components \cdot \\
&\qquad (c.name = c'.name \Rightarrow c = c')
\end{aligned} \tag{3}$$

Entity Name Uniqueness. Every entity in a model has a unique name:

$$\begin{aligned}
&\forall e \in m.entities, \forall e' \in m.entities \cdot \\
&\qquad (e.name = e'.name \Rightarrow e = e')
\end{aligned} \tag{4}$$

Same Entity's Attributes Name Uniqueness. Every attribute of an entity has a unique name:

$$\begin{aligned}
&\forall e \in m.entities, \forall a \in e.attributes, \forall a' \in e.attributes \cdot \\
&\qquad (a.name = a'.name \Rightarrow a = a')
\end{aligned} \tag{5}$$

Composite Components Flattenability. When recursively resolving component references from a composite component's subcomponents, there can't be a reference to this component:

$$\forall c \in m.components \cdot (c \notin flatten(c)) \tag{6}$$

Context Variable Name Uniqueness. Variables in atomic components cannot have the same name if they are not identical:

$$\begin{aligned}
&\forall c \in m.components, \forall v \in (c.pre \cup c.add \cup c.del), \\
&\qquad \forall v' \in (c.pre \cup c.add \cup c.del) \cdot \\
&\qquad\qquad (v.name = v'.name \Rightarrow v = v')
\end{aligned} \tag{7}$$

Composite Component Nonemptiness. Composite components must have subcomponents:

$$\begin{aligned}
&\forall c \in m.components, \cdot \\
&\qquad (c \in CC \Rightarrow c.subcomponents \neq \emptyset)
\end{aligned} \tag{8}$$

Context Immutability. Components don't override existing variables of the context. Every atomic component does not add a new variable to its output context if there is already a variable with the same name in its input context:

$$\forall c \in m.components \cdot (c \in AC \Rightarrow c.add \cap c.pre = \emptyset) \quad (9)$$

Component's Precondition Exhaustivity. Components depend on the variables they remove. Every atomic component has every variable it will remove from the context in its preconditions:

$$\forall c \in m.components \cdot (c \in AC \Rightarrow ac.rem \subseteq ac.pre) \quad (10)$$

Component Instances' Parameters Exhaustivity. Component instances provide values for every parameter of the instantiated component. Every component instance provides exactly as much arguments as the component it instantiates needs parameters. Names and types of the arguments match those of the parameters:

$$\forall c \in m.components \cdot (c \in CC \Rightarrow \forall ci \in c.components \cdot (Prj_{param}(ci.bindings) = ci.component.params)) \quad (11)$$

$$\forall s \in services \cdot (Prj_{param}(s.component.bindings) = s.component.params) \quad (12)$$

Context Validity. Components are instantiated in contexts that fulfill their preconditions. When building flat ordered lists of atomic components for each service (see Section 2.4), every atomic component of these lists has its preconditions fulfilled by its input context:

$$\forall s \in S \cdot (flatten(s.component) \blacktriangleleft s.params) \quad (13)$$

We introduce $\blacktriangleleft$ as a function of $List(C) \times \mathscr{P}(V) \rightarrow Boolean$ in infix notation[1]. This function is true when applied to a component and a context that satisfies the component's preconditions. It is defined by the following semantic rules:

$$\frac{ctx_0 \in \mathscr{P}(V)}{[\,] \blacktriangleleft ctx_0} \quad (14)$$

$$\frac{ctx_0 \in \mathscr{P}(V) \quad c_0.pre \subseteq ctx_0 \quad ctx_1 = ctx_0 \cup c_0.add \setminus c_0.rem \quad \forall i \in [0,n], c_i \in CI \quad [c_1,\ldots,c_n] \blacktriangleleft ctx_1}{[c_0,\ldots,c_n] \blacktriangleleft ctx_0} \quad (15)$$

The point of these properties is not to ensure that a model is consistent with functional specification of web services; it would be too intrusive in the development process and the cost would not be worth the gain in our context. It is rather to guarantee a *structural* consistency of a model at a very small cost, so that developers can use abstractions offered by our meta-model while having a certain level of confidence on the result. This cannot prevent all bugs from happening at runtime; this is the price of the flexibility of our approach. But this can catch bugs related to the misuse of abstractions offered by our meta-model at design-time, thus reducing the footprint in terms of added complexity and unreliability of our approach.

[1] $cs \blacktriangleleft ctx$ means that the context ctx satisfies the component list cs

4 A TOOL TO GENERATE CONSISTENT WEB SERVICES

The meta-model we introduced in Section 2 allows to write web services models whose consistency can be verified by the properties introduced in Section 3. We propose a tool (Sferruzza 2017) to build actual web services from models that conform to the meta-model, after performing consistency verifications on them (see Section 3). This tool takes two inputs: a model file containing a serialized representation of an instance of our meta-model, and a path to a directory which contains implementations of atomic components.

As shown in Figure 2, our tool follows three sequential steps:

Model Parsing. Input model is parsed as concrete syntax of the meta-model (see Section 2.3).

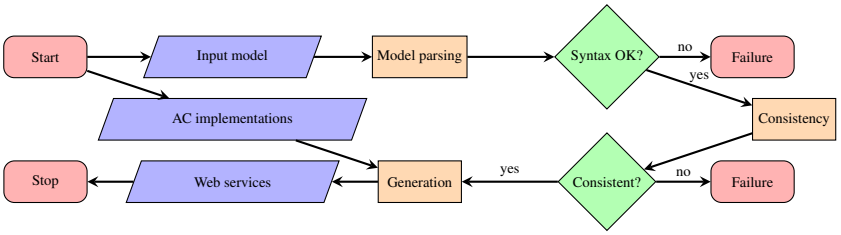

Figure 2: Tool process.

Model Consistency Verification. The model is checked in order to establish its consistency (see Section 3).

Web Services Generation. The model is used to generate an implementation of the web services that it represents.

This process is generic: it can be implemented using different technologies. For example, we choose to specialize our implementation by adding two hypotheses: atomic components' implementations uses the PHP programming language (The PHP Group 2016) and the generated web services relies on the *Laravel* web framework (Otwel 2016). This choice is made because PHP and *Laravel* are well known and used by our first end-users: **Startup Palace**'s developers.

To be able to conclude about consistency of web services generated from a consistent model (see Section 3), we need to assume or manually verify that atomic component implementations and their specifications in the model are compliant. This is currently a limitation that will be removed in a future work by automating the compliance verification.

The last step of our tool is the generation step. Many similar tools (see Section 6) take the approach to generate and output a standalone web application that includes everything necessary to operate it. We took another approach by generating code that should never be manually edited; the generated code does not override any existing files in a *Laravel*'s architecture and can be easily hooked to an existing web application through configuration.

Currently, developers still have to handle security aspects outside of the scope of our solution, for instance when writing atomic components, because they have control on implementation details.

5 EXPERIMENTATIONS

In order to validate our approach, that is to show that the meta-model defined in Section 2 is able to express real web services, we illustrate it here with a voluntarily minimalistic case study, for the sake of brevity. Further case studies focus on comparison with other approaches or technologies. This section specifies web services we want to build, shows how we implement them and discusses advantages of our approach such as flexibility, reusability and safety.

Problem Statement. In order to maintain a digital registration list for an event named *"Mastering cURL and cheese"*, we would like to create and expose web services that would, on one hand, allow people to register themselves, and allow organizers to check registered people, on the other hand.

Informal Specification. To register, a user would use cURL to send a `POST` HTTP request to the endpoint `/register/[name]/[email]` where `[name]` is her name and `[email]` her email address. If this user is already registered, the service would return an error message with a `403` HTTP code. If the given email address is invalid, the service would return an error message with a `422` HTTP code. If both conditions are fulfilled, the registration would be persisted in a relational database along with the registration datetime, and the service would return the input information with a `200` HTTP code. To see registered people, an organizer would use cURL to send a `GET` HTTP request to the endpoint `/attendees/[key]` where `[key]` is a string that needs to be the same as the one hardcoded in the service in order to authenticate the request. If the given key is invalid, the service would return an error message with a `401` HTTP code. If it is valid, it would fetch registrations from the database, serialize them and return them with a `200` HTTP code.

Implementation. In the following examples, we use the concrete syntax introduced in Section 2.3. We show a possible implementation of this informal specification into a model and explain the different steps to build it.

First, we define our data model, that is the `entities` element of our model. We have only one entity: `Registration`.

```
e
  name Registration
  attributes (name: String, email: String, date:
      DateTime)
```

Then, we define two services that are linked to two composite component instances that we will define afterwards. These services will be the interfaces between our application and the outer world, as defined in the informal specification.

```
s
  method POST
  url \/register\/(?<name>[^\/]+)\/(?<email
     >[^\/]+)
  params (name: String, email: String)
  ci Registration

s
  method GET
  url \/attendees\/(?<key>[^\/]+)
  params (key: String)
  ci GetAttendees(apiKey = "mykey")
```

Then, we define the two composite components linked to the services. As they are composite components, they are defined in terms of other components. This step is important because here we choose how to split the features into several components, to maximize separation of concerns and reusability.

```
cc
  name Registration
  ci ValidateEmail
  ci CheckDupRegistration
  ci CreateRegistration
  ci SaveRegistration
  ci RegistrationSerializer
cc
  name GetAttendees
  params (apiKey: String)
  ci CheckKey(correctKey = apiKey)
  ci FetchRegistrations
  ci RegistrationsSerializer
```

The last modeling step is to define the atomic components used in our composite components.

```
ac
  name ValidateEmail
  pre (email: String)
ac
  name CheckDupRegistration
  pre (name: String, email: String)
ac
  name CreateRegistration
  pre (name: String, email: String)
  add (registration: Registration)
ac
  name SaveRegistration
  pre (registration: Registration)
ac
  name RegistrationSerializer
  pre (registration: Registration)
ac
  name CheckKey
  params (correctKey: String)
  pre (key: String)
ac
  name FetchRegistrations
  add (registrations: Seq(Registration))
ac
  name RegistrationsSerializer
  pre (registrations: Seq(Registration))
```

Finally, we can implement these atomic components using PHP. As an example, here is the implementation of the `CheckKey` component.

```
<?php
class CheckKey implements Component
{
  public static function execute(Params $params,
                                 Ctx $ctx)
  {
    $correctKey = $params->get('correctKey');
    $key = $ctx->get('key');
    if ($correctKey === $key) return $ctx;
    else return response('Invalid_key', 401);
  }
}
```

Consistency. We use our tool to check whether the model verifies the properties defined in Section 3 in order to prove its consistency. As a partial example, we show that the `CheckKey` atomic component is well defined. It verifies Formula (9) because $\emptyset \cap \{(\text{"}key\text{"}, String)\} = \emptyset$. It verifies Formula (10) because $\emptyset \subseteq \{(\text{"}key\text{"}, String)\}$. We also show that the `CheckKey` atomic component is well used. It has one instance in the definition of the `GetAttendees` composite components. This instance verifies Formula (11) because it reduces to $\{(\text{"}correctKey\text{"}, String)\} = \{(\text{"}correctKey\text{"}, String)\}$. The `GetAttendees` component is instantiated by a service that provides $\{(\text{"}key\text{"}, String)\}$ as an initial context (according to its parameters). The `CheckKey` instance is well used in the context of this service because it reduces to $\{(\text{"}key\text{"}, String)\} \subseteq \{(\text{"}key\text{"}, String)\}$ (see Formulas (13) to (15)).

Generating Web Services. The next step is to generate a consistent and working implementation of the web services. First, PHP implementations for each atomic component of the model are written. We assume that these implementations are in adequation

with both the model and the informal specification. Next, our tool generates working web services that integrate easily in a *Laravel* application (see Section 4). Finally, we setup other parts of the *Laravel* application to benefit from helper features such as external dependencies management, database schema migrations or end-to-end tests.

Discussion. This case study is a first step to show that our meta-model is flexible enough to express minimalistic yet realistic web services. Components offer a first-class system to split computations into several smaller parts. Because assemblies of components are checked to ensure their consistency, developers can see them as black boxes which helps a lot when designing, implementing and testing these components. Implementing atomic components in the same language and environment as the generation target allows to have a fine control on what is generated without losing the advantages of building using high-level constructions. Because generated code integrates easily in a *Laravel* application, the result can be deployed like any *Laravel* application thus benefiting from good practices such as those described in (Wiggins 2012). Apart from generating code, our approach makes it easy to generate diagrams from models that allow easy visualization of how services are implemented or what are components dependencies, for example. This might be a game changer for new developers onboarding. Finally, MDE approach improves maintainability. For example, if we want to make an evolution and allow attendees to modify their registration information and status by adding a new service, it is easy to implement it while reusing existing components and still be confident about consistency, thanks to the verification step.

Yet, we performed preliminary experiments with the approach; developers of **Startup Palace** have been involved in these experiments. However we plan to implement systematic evaluation tools in order to support more realistic (by nature and by size) case studies.

6 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; Rocheteau and Sferruzza 2016; Scheidgen, Efftinge, and Marticke 2016).

This work is built on the approach of REIFIER, presented in (Rocheteau and Sferruzza 2016). As in REIFIER, we use a meta-model approach instead of a meta-programming one, as recommended for server-side code generation in (Scheidgen, Efftinge, and Marticke 2016). We also share the meta-modeling approach with the *M3D* tool introduced in (Bernardi, Cimitile, Di Lucca, et al. 2012) and extended in (Bernardi, Cimitile, and Maggi 2016), which is itself based on *Declare* (see (van der Aalst, Pesic, and Schonenberg 2009)). *M3D* uses as an input a 4-layers meta-model: information layer by means of UML class diagram, service layer by means of BPMN, presentation layer by means of (Spring/Android)-like meta-model and a process layer by means of *Declare*. *M3D* is evaluated on a case study to show that 90% of the time spent developing the application is spent on modeling. This is advantageous as modeling is often faster, produces more stable applications and models are easier to maintain than code.

Our approach was developed with this idea in mind, thus focusing on design-time support. But it differs from existing tools in several different ways, related to both flexibility and support. First, as REIFIER but not *M3D*, it is built to allow property verification at design-time. This verification takes the form of a set of rules that ensure structural consistency of models. This ensures confidence in the resulting systems without the drawbacks of heavy formal methods. Second, because our components are specified in terms of preconditions and effects, instead of inputs and outputs in REIFIER, we can ease their reusability by minimizing the coupling to their instantiation context while providing the same level of verification.

To avoid shortcomings such as described in (Gronmo et al. 2004), that is WSDL models contain too much technical details and are difficult to understand for humans, our method makes it possible to express technical details in components' implementation that are not captured in the model. This is an attempt to ease onboarding of new developers by both offering lots of MDE-related advantages and at the same time giving full flexibility on technology and implementation details.

Our approach is more related to Eiffel (Meyer 1992), than to BPEL (Fu, Bultan, and Su 2004) or to Join Calculus (Fournet and Gonthier 2002). As BPEL, we acknowledge that programming-in-the-large and programming-in-the-small require different types of languages (DeRemer and Kron 1975); but our meta-model is simpler than BPEL, thus relevant for developing web services. Join Calculus is devoted to the design or semantics of distributed languages, which is out of our scope. We share with Eiffel the principle of using contracts, of being extensible, reusable, and reliable. However, because our

contracts are simpler (a subset of first-order logic), our approach is suitable for being used by non formal methods experts while providing useful support for them to build safe and reusable components.

7 CONCLUSION

We proposed a method to build web services. It is fast, simple, robust and flexible. It is based on a metamodel that makes it possible to describe web services using high-level constructions (such as parametric components) which enables verifying their consistency using an axiomatic semantics as well as generating working web services. We also built a tool that leverages this process in the technological context of a web company, **Startup Palace**. The whole approach was illustrated on a case study to show its advantages. Even if one of the motivations was to develop MVPs applications, the approach is not limited to this scope and is suitable to most applications based on web services.

Several main prospects are here sketched. First, the type system used to describe component parameters, preconditions and model's entities (among others) is at the core of the consistency verification, yet it is not flexible enough. Making it more expressive, by allowing subtyping in component preconditions for example, while keeping at least the same level of verification might be necessary to reach a good reusability on bigger projects. Another perspective is to allow and ease safe model composition, to allow developers to reuse concepts between projects when it makes sense. Finally, the whole approach needs to be evaluated on more realistic (by nature and by size) case studies. This evaluation must rely on metrics that have a good correlation with the benefits of our approach, such as easing new developers to onboard on projects and reuse of exisiting code.

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*, Maui, HI, USA, November 02, 2012, 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.

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.

Fournet, Cédric and Georges Gonthier (2002). "The Join Calculus: A Language for Distributed Mobile Programming". In: *Applied Semantics*. Springer, pp. 268–332.

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, 2004. EEE'04. 2004 IEEE International Conference On*. IEEE, pp. 42–45.

Meyer, Bertrand (1992). *Eiffel: The Language*. Prentice-Hall, Inc.

Open API Initiative and The Linux Foundation (2017). *OpenAPI Specification*. URL: https://www.openapis.org (visited on 07/25/2017).

Otwel, Taylor (2016). *Laravel*. URL: https://laravel.com/ (visited on 07/25/2017).

RAML Workgroup (2017). RAML. URL: https://raml.org/ (visited on 07/25/2017).

Rocheteau, Jérôme and David Sferruzza (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.

Sferruzza, David (2017). *Safe Web Services Generator*. URL: https://gitlab.startuppalace.com/research/swsg.

The PHP Group (2016). PHP. URL: https://php.net/ (visited on 07/25/2017).

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.

Wadler, Philip (1992). "The Essence of Functional Programming". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, pp. 1–14.

Wiggins, Adam (2012). *The Twelve-Factor App*. URL: https://12factor.net/ (visited on 07/25/2017).