MicroBuilder: A Model-Driven Tool for the Specification of REST Microservice Architectures

Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, Ivan Luković,
University of Novi Sad, Faculty of Technical Sciences, 21000 Novi Sad, Serbia
{branko.terzic, dimitrieski, slavica, grist, ivan}@uns.ac.rs

***Abstract*—In this paper we present MicroBuilder, the tool used for the specification of software architecture that follows REST microservice design principles. MicroBuilder comprises MicroDSL and MicroGenerator modules. The MicroDSL module provides the MicroDSL domain-specific language used for specification of microservice architecture. In this paper we present the MicroDSL meta-model, specified in Ecore, the concrete syntax of the MicroDSL and examples of its usage. The MicroGenerator module is used to generate executable program code based on MicroDSL specifications**.

I. Introduction

In the last few decades, due to the development of Internet-related technologies, Representational State Transfer (REST) web service software architectures become more popular, due to their simplicity, interoperability and scalability [1]. Software applications which follow this software architecture style comprise a set of REST web services that are using Hypertext Transfer Protocol (HTTP) [2]. Each REST web service provides a certain set of functionalities exposed to different web clients through an Application Programming Interface (API) specification [3]. REST web services can be organized in a form of: (i) software units that can exist and run only within the core application to which they belong, or (ii) loosely coupled software units that can exist and run independently from the core application [4]. Therefore, there are two main approaches in development of REST web service software architectures: (i) through monolithic software applications and (ii) through microservice software applications [5, 6].

Monolithic software applications are composed of software units that are not independent from the core application which they belong to [7]. Development of monolithic applications was dominant programing style used by a majority of engineers over the past several decades. This type of development approach forced engineers to work in large teams which resulted in software solutions that were hard to maintain and understand because of their size and architecture complexity [7]. Microservice software applications were introduced as suits of small, independent and one-purpose software units, called microservices [8]. Microservices have a small set of responsibilities and have separate database storage. Thus, microservices are autonomous software units which can be written in different programming languages and still be a part of the same software solution. Their functionalities are usually exposed in the form of REST API specification, while inter-service communication is based on lightweight protocols such as the HTTP [9]. In this way, engineers can independently build parts of application as microservices and deploy them separately to web servers [7].

A large number of microservices per application leads to new challenges: (i) *user request acceptance and routing*, i.e. providing a unified access interface and a single entry point to the whole microservice ecosystem, (ii) *microservice auto-discovery and registering*, i.e. providing a single point for microservice instances monitoring and microservice name, host and port registry, and (iii) *load balancing*, providing an improvement of workload distribution across the microservices [10]. In order to provide a solution to these challenges, a redundant program code that covers the same functionality needs to be written at different layers of the software architecture. This often leads to mistakes as a developer introduces unintentional errors to the repetitive code constructs. Furthermore, the configuration of microservice architecture is not a trivial task and it requires a complex and redundant work to be performed. Therefore, in order to alleviate and speed up such a process, it could be beneficial for a developer to have a language with concise set of concepts which are specific to the domain of REST microservice architecture development. Such a language should allow developers to have a single specification of a microservice without writing any boilerplate or redundant code. In Model-Driven Software Engineering (MDSE) such a language that is tailored to a specific application domain is called a Domain-Specific Language (DSL). The main goal of the research presented in this paper is to provide such a language and use it to solve real-world problems.

In this paper, we present a model-driven software tool for the REST microservice architecture specification and program code generation, named MicroBuilder. MicroBuilder comprises the following modules: (i) MicroDSL, the module that provides a DSL for the specification of REST microservice software architecture, (ii) MicroGenerator, the module which comprises set of code generators used to generate executable program code based on MicroDSL specifications. To develop the MicroBuilder tool we have used the Eclipse Modeling Framework (EMF) [11]. MicroDSL concrete syntax is developed using the Xtext language, while individual code generators within the MicroGenerator module are developed using the Xtend language [12,13].

Apart from the Introduction and Conclusion, the paper has four sections. In Section 2 we present the architecture of MicroBuilder. The abstract syntax of MicroDSL is introduced in Section 3. In the fourth section we present the concrete textual syntax of the language, alongside examples of its usage. In Section 5 we give an overview of the related work.

II. THE ARCHITECTURE OF MICROBUILDER

In this section we present the architecture of the MicroBuilder tool. The overview of main modules and their inputs and outputs are given in Figure 1.

The MicroBuilder tool comprises two main modules: the MicroDSL module and the MicroGenerator module. The MicroDSL module provides a domain-specific language used for microservice software architecture specification, named MicroDSL. The MicroDSL language provides a set of concepts which are related to the domain of REST microservice architecture development [7]. These concepts are presented in detail in the third section of this paper where we describe the language meta-model. For the specification of the MicroDSL concepts, the textual concrete syntax has been developed. Using the MicroDSL concrete syntax, user is able to specify the whole microservice architecture at one place, using just concepts related to the REST microservice architecture development domain. A microroservice architecture specification is then used as an input to the MicroGenerator module. The MicroGenerator module provides a set of code generators which are used to generate executable program code for the target execution platform. Currently, we have developed a set of code generators for the Java programming language. Generated program code uses Spring, Spring Cloud and NetflixOSS frameworks [14, 15, 16]. These are all open-source frameworks developed by Netflix, Amazon and Pivotal companies. These companies have become early adopters of microservice architecture in large-scale software systems implemented in cloud architecture [17]. Aforementioned frameworks provide a set of tools such as: (i) Zuul, for user request routing and filtering (ii) Eureka, for microservice auto-discovery and registry, (iii) Ribbon, for resilient and intelligent inter-process communication and load balancing, and (iv) Hystrics, for isolated latency and fault tolerance among microservices. These tools are utilized in order to resolve some of the aforementioned challenges caused by a large number of microservices per application (c.f. Intoduction). The remaining challenges concerning redundant coding and not so trivial configuration of microservice architectures, using these tools, can be resolved by using our MicroBuilder tool. This would allow developers to specify microservice architecture and to generate executable program code. Generated program code consists of: (i) Netflix Zuul microservice, (ii) Netflix Eureka microservice, and (iii) user-defined microservices with basic REST interfaces for insert, update and delete operations over the MongoDB database [18]. This way, a user is able to specify each microservice at one place, and necessary Java code will be generated in all application layers where it is required.

Furthermore, if a user decides to use other frameworks involved in REST microservice architecture development, there is no need to change a MicroDSL specification. Only a new code generator for a chosen framework needs to be developed and added to the MicroBuilder tool as a plugin.

III. MICRODSL ABSTRACT SYNTAX

In this section we present the abstract syntax of the MicroDSL language. The abstract syntax is specified in a form of a meta-model that conforms to the Ecore meta-meta-model [19]. The developed meta-model is presented in Figure 2. In the rest of this section, we present each of the MicroDSL concepts with the names of corresponding meta-model classes and attributes written in italics inside parentheses.

The main language concept is microservice architecture specification (*McServiceArchitecture*), which comprises all other language concepts. Such a specification is identified by its name (*name*) and may also have a description (*mcsDescription*). Each microservice architecture specification comprises zero or more microservices (*McService*). Each microservice is described by the name (*name*) and its working group (*mcsGorupId*). Working group is used as microservice unique identifier within microservice architecture specification. For each microservice, an HTTP port (*mcsPort*) is specified which will be used by the microservice to receive user requests. Also, at this level, it can be specified whether the microservice is to be registered within microservice auto-discovery registry or not (*mcAutoDiscovery*). Each microservice comprises a set of microservice resources (*McsResources*) which may be one of the following: (i) proxy filters (*McsProxyFilter*), used for modeling custom user request filters or (ii) microservice entities (*McsEntity*), used for modeling microservice business entities. For a proxy filter user may specify its type (*mcFilterType*) which values are predefined in the filter type enumeration (*FilterType*). Proxy filter type is used to specify different types of proxy filters: (i) *pre*, filter is executed before the user request has been routed, (ii) *routing*, filter is executed in the moment when user request is accepted, (iii) *post*, filter is executed after the user request was routed, and (iv) *error*, filter is executed if an error occurred in the moment of user request handling. Further, it is possible to define a logical expression that will determine should a proxy filter be applied on user request

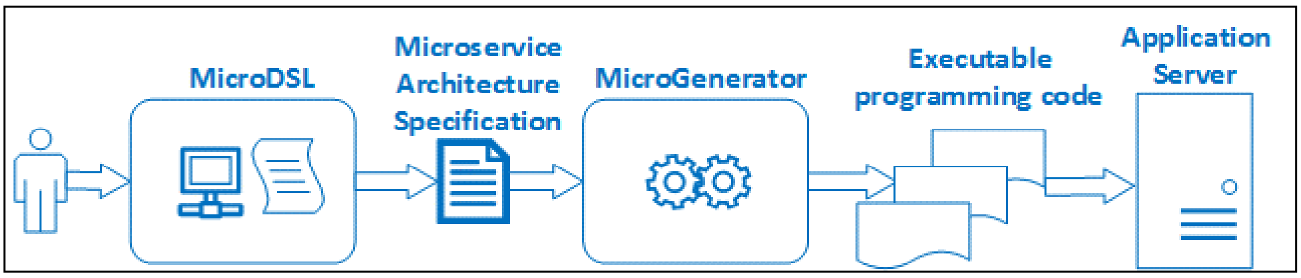

MicroDSL
Microservice Architecture Specification
MicroGenerator
Executable programming code
Application Server

Figure1. Overview of the MicroBuilder architecture

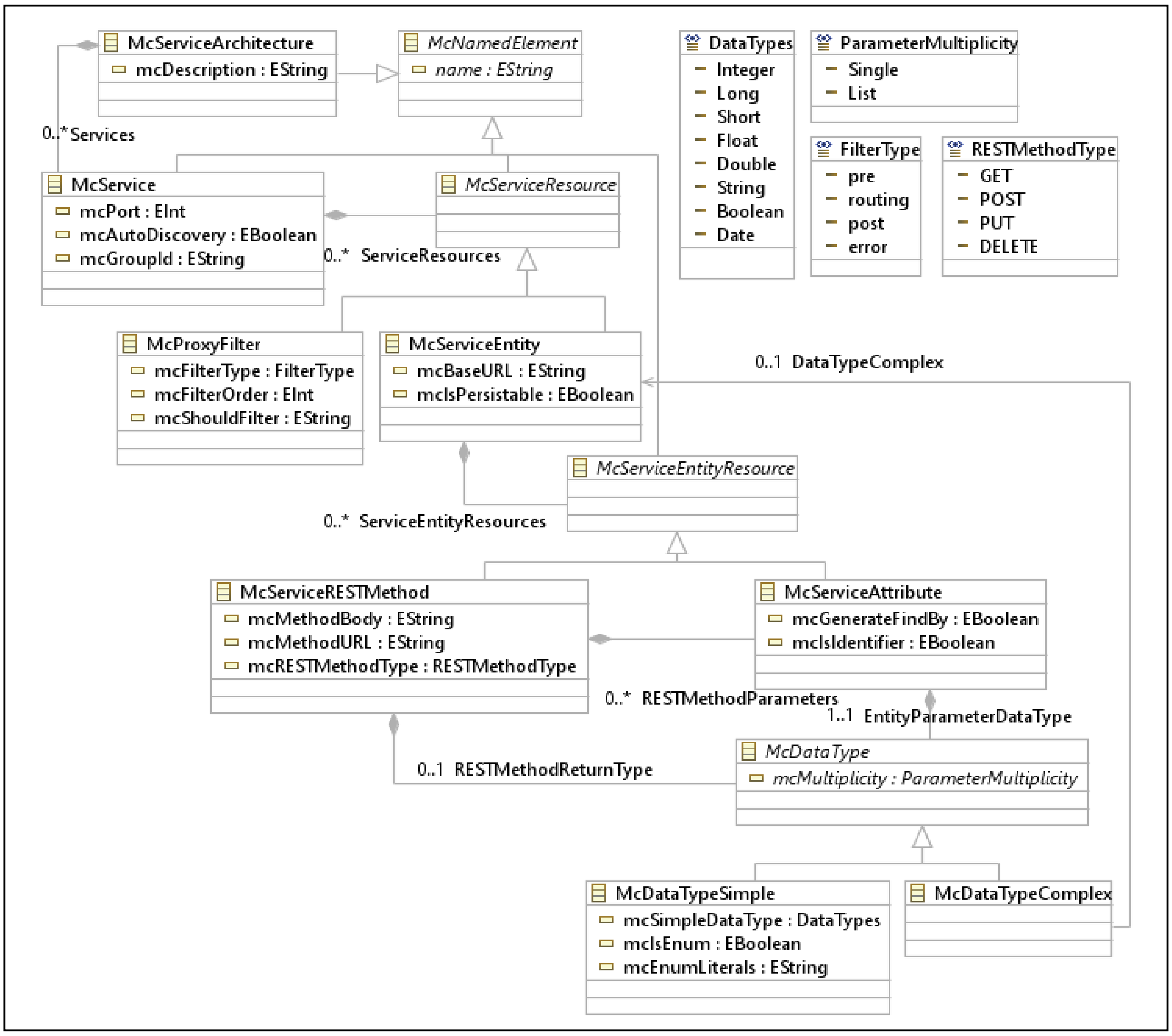
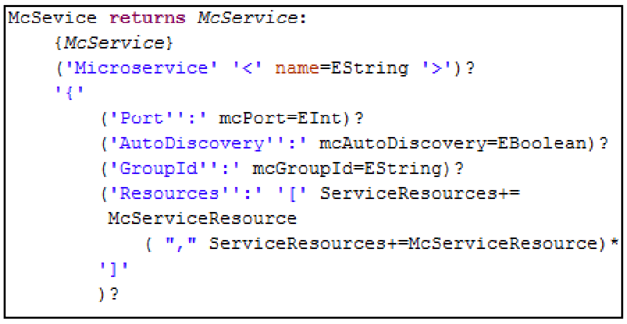


Figure 2. The MicroDSL meta-model

or not (*mcShouldFilter*). Also, the priority of a proxy filter application can be specified (*mcFilterOrder*). The microservice entity is defined as an aggregation of different resources (*McEntityResource*) that can be reached from the base URI (*mcBaseURI*). It can be specified should a microservice entity be persisted within database storage or not (*mcIsPersistable*). Entity resources can be classified as REST methods (*McRESTMethod*) or microservice attributes (*McServiceAttribute*). REST methods are described by the base URL which represents a unique identifier for REST API (*mcMethodURL*). URL description is used by web clients to access REST method resources. Different types of REST methods are predefined in the method type enumeration (*RESTMethodType*), while the implementation of method resources is specified within method body (*mcMethodBody*). For each REST method both, method return type and method arguments can be defined. These are modeled using *RESTMethodReturnType* and *RESTMethodParameters* aggregation relationships, respectively. Microservice attributes (*McServiceAttribute*) can be used to describe microservice business entity fields or REST method parameters. Accordingly, when they are used to describe microservice business entity fields, the following may be specified: (i) should this field be used as unique database identifier (*mcIsUnique*) or (ii) should the REST API search operation for entity field be added to the generated code (*mcGenerateFindBy*). Further, when the microservices attributes are used to describe REST method parameter, the name and the attribute data type are specified. The microservice attribute data type (*McDataType*) is described by data type multiplicity (*mcMultiplicity*). User is able to specify both, simple (*McDataTypeSimple*) data types and complex (*McDataTypeComplex*) data types. Simple data types are used for the specification of predefined data types such as: numbers, date, boolean and string. All predefined data types are specified in the data type enumeration (*DataTypes*). Within simple data types, user is also able to specify enumerations (*mcIsEnum*), alongside enumeration literals (*mcEnumLiterals*). Complex data types are used to describe user-defined data types, that are modeled using *DataTypeComplex* association relationship.

IV. A Web Shop Specification Using MicroDSL Concrete Syntax

In this section we present a textual concrete syntax of MicroDSL, alongside the example of its usage. In the example, we used MicroDSL to model the web shop microservice architecture. Also we present a sample of Java program code generated using the MicroGenerator module. Here, we discuss the structure of generated Java code in order to explain all benefits of using the MicroDSL language. Also, we compare the number of lines of code needed to specify the web shop microservice architecture using MicroDSL to the number of manual written lines of Java code needed to specify the same microservice architecture.



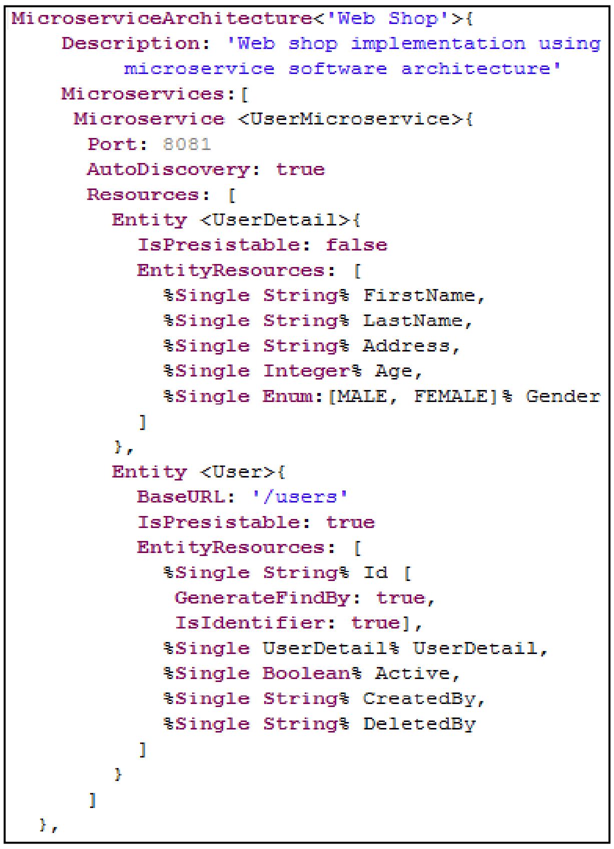
```
McSevice returns McService:
    {McService}
    ('Microservice' '<' name=EString '>')?
    '{'
        ('Port'':' mcPort=EInt)?
        ('AutoDiscovery'':' mcAutoDiscovery=EBoolean)?
        ('GroupId'':' mcGroupId=EString)?
        ('Resources'':' '[' ServiceResources+=
         McServiceResource
            ( "," ServiceResources+=McServiceResource)*
        ']'
        )?
```

Figure 3. The *McService* EBNF rule

We have created the concrete syntax of the MicroDSL language using Eclipse plug-in named Xtext [11, 12]. In Figure 3, we present the Extended Backus-Naur Format (EBNF) rule for definition of individual microservices. First, a user needs to specify the “Microservice” reserved word. After that, the name of the microservice within “<” and “>” characters can be specified. The special characters “{”, “}”, “[” and “]” are used to open and close the detailed specification of a concept. The attribute values for the microservice concept are specified as follows: a reserved word is followed by the “:” character after which the value of the attribute can be specified. The HTTP port of the microservice can be specified using the “Port” reserved word, while microservice unique identifier is specified by using the “GroupId” reserved word. The attribute value that determines if microservice be a part of the auto-discovery configuration is specified using the “AutoDiscovery” reserved word. Microservice proxy filters and microservice business entities are specified between “[” and “]” characters, after the “Resources” reserved word. Different microservice resources are separated using the “,” character.

The web shop specification comprises the following microservices: (i) the *User microservice*, responsible for user registration, authentication and authorization, (ii) the *Product microservice*, used for tracking information about products, (iii) the *ShoppingCart microservice*, responsible for storing and tracking of the user shopping cart information, and (iv) the *Payment microservice*, used for tracking information about user payments.

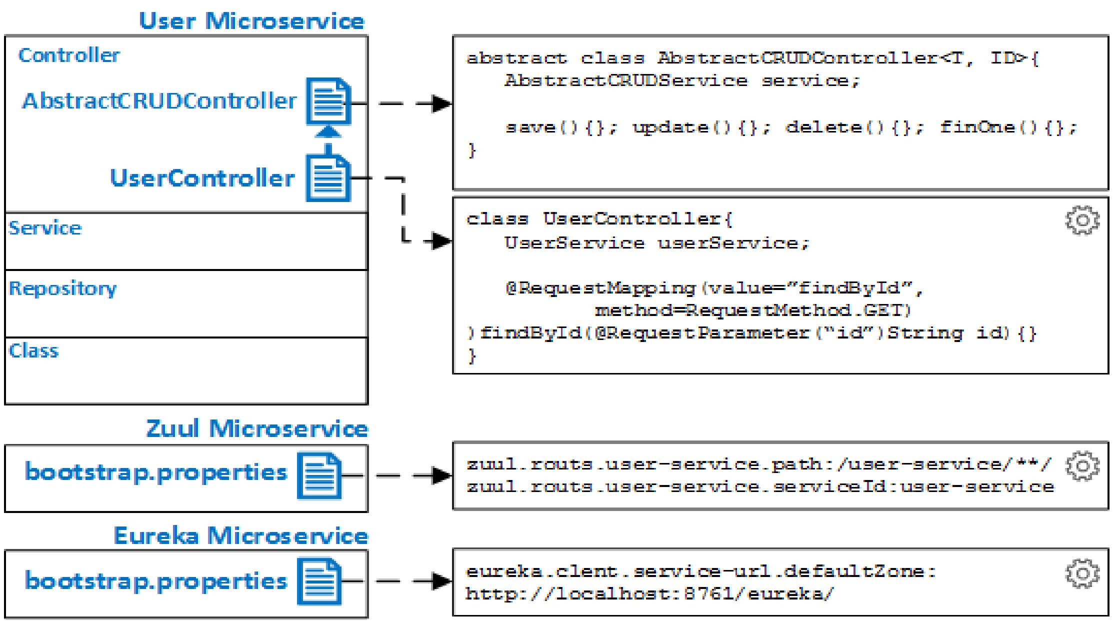
In Figure 4 we present the specification of the User microservice. The User microservice will be listening for the user requests on 8081 HTTP port, and will be included into microrservice auto-discovery configuration. This microservice comprises a set of two microservice entities: the UserDetail microservice entity and the User microservice entity. The UserDetail microservice entity values should not be persisted within database storage, and they are only used to describe basic user information. The User microservice entity is defined by its base URL which represents the API specification for database entity values access. Within the User microservice entity attributes, the attribute named Id is specified as a unique database identifier. Also, the REST search method for the User entity will be generated based on the Id attribute. The UserDetail microservice entity is used to specify the complex data type attribute within the User microservice entity attributes.



```
MicroserviceArchitecture<'Web Shop'>{
    Description: 'Web shop implementation using
         microservice software architecture'
    Microservices:[
     Microservice <UserMicroservice>{
      Port: 8081
      AutoDiscovery: true
      Resources: [
        Entity <UserDetail>{
          IsPresistable: false
          EntityResources: [
            %Single String% FirstName,
            %Single String% LastName,
            %Single String% Address,
            %Single Integer% Age,
            %Single Enum:[MALE, FEMALE]% Gender
          ]
        },
        Entity <User>{
          BaseURL: '/users'
          IsPresistable: true
          EntityResources: [
            %Single String% Id [
             GenerateFindBy: true,
             IsIdentifier: true],
            %Single UserDetail% UserDetail,
            %Single Boolean% Active,
            %Single String% CreatedBy,
            %Single String% DeletedBy
          ]
        }
      ]
  },
```

Figure 4. The *User microservice* specification

In Figure 5 we present a structure and specification of Java program code which is generated using the User microservice specification as an input of the MicroGenerator module. The generated code is divided in four distinct functional layers: (i) the *Controller*, the code that specifies the REST APIs, (ii) the *Service*, the code that specifies the application business logic (iii), the *Repository*, the code that specifies data management methods for the target database, and (iv) the *Class*, the code that specifies microservice business entities. In top right corner of Figure 5 we present a sample of the Java code specification for the User microservice Controller level. The code specifications containing a “gear” icon represent the code generated by MicroGenerator. The other code specifications represent the hand-written code which is the same for each microservice. The *AbstractCRUDController* is an abstract class that specifies REST APIs for basic create, update, delete and



```
abstract class AbstractCRUDController<T, ID>{
   AbstractCRUDService service;

   save(){}; update(){}; delete(){}; finOne(){};
}
```

```
class UserController{
   UserService userService;

   @RequestMapping(value="findById",
         method=RequestMethod.GET)
)findById(@RequestParameter("id")String id){}
}
```

```
zuul.routs.user-service.path:/user-service/**/
zuul.routs.user-service.serviceId:user-service
```

```
eureka.clent.service-url.defaultZone:
http://localhost:8761/eureka/
```

Figure 5. The *User microservice* generated programming code structure

search operations over all User microservice business entities. The *UserController* is class which inherits the *AbstractCRUDController*, and it is used to specify REST APIs for methods which describe the User microservice business logic. The same structure of program code is present on Service and Repository levels as well. We also present the configuration code for User microservice registry, auto-discovery and user request routing, within Zuul and Eureka microservices. We can conclude that we have specified the whole User microservice in one place and program code was generated in many different application layers and configuration files. In this way the risk for introducing unintentional errors by developer is minimized as well as the invested effort.

In Table 1, the number of MicroDSL lines of code is presented alongside the number of manually written lines of code for each microservice separately. We present also the sum of lines of code for all microservices within web shop specification as well as the sum of manually written lines of code. We conclude that the number of manually written lines is by an order of magnitude greater compering to number of lines written using the MicroDSL language. The number of manually written lines of code is obtained as a result of our effort in applying the best programming practices and minimizing the number of blank lines. In practice the number of lines of code needed to implement the same software solution would be probably much higher. Based on these results, we can conclude that the MicroBuilder tool can be suitable for fast prototyping in situations when we need quick solution which requires minimal coding interventions on the generated program code.

V. Related Work

While surveying the state-of-the-art literature in this area, we have found several papers that deal with the specification of REST-based web service architectures, using the MDSE approach. However, to the best of our knowledge, there are no approaches which utilize MDSE in resolving problems regarding specification of REST microservice software architectures. In [20], the authors propose a model-driven approach to REST web application modeling and code generation based on web application meta-data. In [21], the authors present an approach that uses Eclipse Modeling Framework data models as input and generates web applications following the REST principles, called EMF-REST. EMF-REST also integrates model and web-specific features to provide model validation and security capabilities, respectively, to the generated application programming interface (API). The authors of the paper [22] argue that the process of describing REST web services can be described as a series of model transformations, starting from service functionality and gradually refining phase, until a REST service application is reached. In [23], the authors introduce a model-driven approach for the

TABLE I.
The Comparison of Number of Lines for MicroDSL and Manually Written Java Code

Microservice name	The number of lines of code in the MicroDSL specification	The number of manually written lines of code
User microservice	28	406
Payment microservice	17	270
Product microservice	18	335
ShoppingCart microservice	25	385
Total	98	1802

integration of Web 2.0 functionalities, implemented as REST services. The authors present a REST meta-model for specifying these REST services at a conceptual level. In [24], the authors present a multi layered meta-model for REST applications, discuss the connection to REST compliance and show an implementation of their approach based on the proposed meta-model and method. In [25], the authors discuss the challenges of composing REST web services and propose a formal model for describing individual web services and automating the composition.

CONCLUSION

In this paper we have presented a model-driven tool for a specification of REST microservice architectures, named MicroBuilder. Our goal was to automate and ease the process of microservice software architecture specification and configuration. In order to achieve this goal, we have developed the MicroDSL domain-specific language used for microservice software architecture modeling. First, we have developed the MicroDSL meta-model, specified using Ecore. The MicroDSL meta-model represents the abstract syntax of the language. Then, we have developed textual syntax which is used as a visual representation of the MicroDSL modeling concepts. Using the MicroDSL concrete syntax, users are able to specify the whole microservice architecture at one place. We have also developed a set of code generators that are part of the MicroBuilder tool. These code generators are used to generate executable program code, based on MicroDSL specification.

In our future research we plan to extend the MicroDSL meta-model with concepts that will enable the specification of microservice architecture load balancing and inter-service communication patterns. We also plan to develop a graphical concrete syntax, that will be used for the specification of inter-service communication patterns. The graphical concrete syntax will not be used as an alternative to existing textual syntax. We plan to use them both in parallel in order to determine which approach is more suitable for our users. The detail analyses of our approach will be investigated and presented in detail in a case study.

ACKNOWLEDGMENT

The research presented in this paper was supported by Ministry of Education, Science and Technological Development of Republic of Serbia, Grant III-44010.

REFERENCES

[1] C. Pautasso: RESTful web services: principles, patterns, emerging technologies. In *Web Services Foundations*. pp. 31-51, Springer New York, 2014.

[2] S. Mumbaikar, P. Padiya: Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5), May 2013.

[3] S. Vinoski: Restful web services development checklist. *IEEE Internet Computing*, 12(6), Novembr 2008.

[4] C. Pautasso, O. Zimmermann, F. Leymann Restful web services vs. "big" web services: making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web* ACM, pp. 805-814, April 2008.

[5] M. Villamizar, O. Garcés, L. Ochoa, H. Castro., L. Salamanca, M. Verano, M. Lang. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. pp. 179-182, IEEE, May 2016.

[6] H. Knoche. Sustaining runtime performance while incrementally modernizing transactional monolithic software towards microservices. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. pp. 121-124, ACM, March 2016.

[7] D. Namiot, M. Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*. 2(9), pp. 24-27, 2014.

[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016.

[9] J. I. Fernández Villamor, C. A. Iglesias Fernandez, M. Garijo Ayestaran. Microservices: Lightweight service descriptions for REST architectural style. 2nd International Conference on Agents and Artificial Intelligence, Valencia, España, 2010.

[10] A. Balalaie, A. Heydarnoori, P. Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*. pp. 201-215, Springer International Publishing, September, 2015.

[11] "Eclipse" [Online], Available: http://projects.eclipse.org/projects/modeling. [Accessed: 22-March-2017.].

[12] M. Eysholdt, H. Behrens, Xtext: "Implement your language faster than the quick and dirty way", in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, OOPSLA '10, ACM, New York, NY, USA, pp. 307-309, 2010.

[13] "Xtend" [Online], Available: http://www.eclipse.org/xtend/ [Accessed: 22-May-2017.].

[14] "Spring" [Online], Available: https://spring.io/ [Accessed: 22-March-2017.].

[15] "Spring Cloud" [Online], Available: http://projects.spring.io/spring-cloud/ [Accessed: 22-March-2017.].

[16] "NetflixOSS" [Online], Available: https://cloud.spring.io/spring-cloud-netflix/ [Accessed: 22-May-2017.].

[17] F. Montesi, J. Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. *arXiv preprint arXiv:1609.05830*, 2016.

[18] "MongoDB" [Online], Available: https://www.mongodb.com/ [Accessed:22-03-2017.].

[19] "Ecore" [Online], Available https://github.com/occiware/ecore [Accessed:22-03-2017.].

[20] S. Pérez, F. Durao, S. Meliá, P. Dolog, O. Díaz. RESTful, resource-oriented architectures: a model-driven approach. In *International Conference on Web Information Systems Engineering*. pp. 282-294, Springer Berlin Heidelberg, December 2010.

[21] H. Ed-Douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, J. Cabot. EMF-REST: generation of restful apis from models. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. pp. 1446-1453, ACM, April 2016.

[22] M. Laitkorpi, P. Selonen, T. Systa: Towards a model-driven process for designing restful web services. In *Web Services, 2009. ICWS 2009. IEEE International Conference on* IEEE. pp. 173-180, July 2009.

[23] F. Valverde, O. Pastor. Dealing with REST services in model-driven web engineering methods. *V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB*, pp. 243-250, 2009.

[24] F. Haupt, D. Karastoyanova, F. Leymann, B. Schroth. A model-driven approach for REST compliant services. In *Web Services (ICWS), 2014 IEEE International Conference on* (pp. 129-136). IEEE, June 2014.

[25] H. Zhao, P. Doshi. Towards automated restful web service composition. In *Web Services, 2009. ICWS 2009. IEEE International Conference on* IEEE, pp. 189-196, July 2009.