Exploring the Scala Macro System for Compile Time Model-Based Generation of Statically Type-Safe REST Services

Filipe R. R. Oliveira, Hugo Sereno Ferreira, and Tiago Boldt Sousa

Department of Informatics Engineering
Faculty of Engineering, University of Porto, Portugal
Email: {filipe.rroliveira, hugo.sereno, tiago.boldt}@fe.up.pt

***Abstract*—Representational State Transfer (REST) is a prolific architectural style among modern Web services, mainly due to its better performance, scalability and simplicity. A common usage of the style includes services that implement CRUD (Create, Read, Update, and Delete) operations for entities of a model. Most frameworks that do this automatically, apply the models logic in run-time usually using reflection or in-memory data structures, and are written in dynamically typed programming languages. Such technical choices usually hinder two software quality attributes: performance (due to run-time adaptation) and maintainability (due to absence of compile-time guarantees). This paper studies the impact of interpreting the models at compile-time with statically typed programming languages, using Scala as a representative. Based on a generic architecture, we implemented a proof of concept, called the Metamorphic framework, which uses a Domain Specific Language (DSL), supported by a macro system, to generate entire applications. Evaluation was executed by performing both quantitative benchmarks and qualitative analysis of Metamorphic against other frameworks.**

Keywords–Model-Driven Engineering; REST; Internal DSL; Scala Macros.

I. INTRODUCTION

The number of Internet users has tripled in the last decade [1] mainly due to the appearance and growth of mobile devices. These users and devices stay connected and explore their potentialities through the consumption of Web services, such as, static or dynamic Web pages, mobile applications content, and real-time services. Two common architectures for implementing these services were the Remote Procedure Call (RPC) and the Service-Oriented Architecture (SOA) [2], mainly explored through the Simple Object Access Protocol (SOAP).

In the same time-frame of SOAP's specification, Roy Fielding defined the Representational State Transfer (REST) architectural style [3] to be applied in distributed hypermedia systems. The style defines a set of six constraints: client-server, stateless, cache, uniform interface, layered system, and code-on-demand. The uniform interface constraint uses the concept of resource, around which communication is built. A resource is an abstract instance of any concept that can be uniquely identified. All these constraints enable scalability, portability, visibility, and simplicity in exchange for some degraded efficiency and reliability. It is normally preferred to the SOAP approach as it achieves better performance most of the time [4]. In practice, REST is usually implemented using URI (Uniform Resource Identifier) for resource identification, and HTTP for stateless client-server cacheable communication.

The need to implement more complex and robust Web services led to the development of frameworks, that provide solutions for recurrent problems and enable better structured implementations. Some of these use model-driven engineering [5], i.e., they can deliver CRUD operations for a set of model entities, reducing repeated code when compared with most traditional frameworks. This approach has the following advantages: short-time-to-market, fewer bugs, increased reuse, and easier-to-understand up-to-date documentation [6].

In general, current model-driven REST frameworks do in fact reduce repetition of code but due to implementation decisions there are two main problems.

Firstly, they are mostly implemented in dynamically typed languages [7], such as Python and JavaScript. These kind of languages don't require the use of explicit types, in which case type-related errors are more susceptible to happen. This fact combined with dynamic typechecking delays resolution of these errors to run-time, suggesting longer debugging sessions. Strongly and statically typed languages reduce substantially this problem and consequently may enable better performed services, due to compiler optimizations based on types.

Secondly, these frameworks implement model-based generation through the inspection of variables or introspection [8] for collecting the schema, and through parameterized functions or intercession [8] for responding to requests. All this work is done at run-time, increasing the program's setup time or even the response time to requests.

Following such logic and considering a statically type-safe programming language that enables generation of REST services in compile time, a question can be raised:

> *Can a model-driven REST framework written in that language improve the development process and performance when compared to current ones?*

In this research, Scala [9] was used as proof of concept to answer this question. This language, that was built with scalability in mind, offers a strong static type system, and an easy capacity for compile-time generation, through macros [10]. These characteristics promise that developers may be able to implement their model-driven REST services even faster and with more robustness, as type errors may be identified sooner.

The paper is organized as follows. Firstly, in Section II the most relevant model-driven REST frameworks are briefly presented and compared. Secondly, in Section III, we describe a generic architecture for model-driven REST applications and how that was translated into the Metamorphic framework. Thirdly, in Section IV the validation criteria is defined as well

 ISBN: 978-1-61208-458-9

as the steps executed to verify that, which include framework benchmarks and a synthetic environment experiment. At last, the conclusion of the work is presented followed by future work that can be explored.

II. Model-Driven REST Frameworks

Model-driven REST frameworks usually consider two types of resources: collections of entities which may have create and read operations; and instances of entities which may have update, replace, and delete operations.

A. Django REST Framework

Django REST framework [11] is an open source framework in Python to build Web APIs (Application Program Interfaces), and is built on top of the Django framework [12], a tool that enables fast development of Web applications, including model-driven development. The framework is funded in: *views* that given requests perform necessary actions and prepare responses; *serializers* that define the structure of object's data; and *urls* that connect URLs (Uniform Resource Identifiers) with views.

```
class Category(models.Model):
    name = models.CharField(max_length = 50)
    description = models.CharField(max_length = 100)

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category

class CategoryViewSet(viewsets.ModelViewSet):
    queryset = Category.objects.all()
    serializer_class = CategorySerializer

router = routers.DefaultRouter()
router.register(r'categories', CategoryViewSet)
urlpatterns = router.urls
```

Listing 1. Example of a simple API with the Django REST framework

The support for model-driven development is delivered by subclassing a base model, a base model serializer, and a base generic view, as shown in Listing 1. These subclasses override the interface methods and use reflection in order to implement the intended functionality. This adds an overhead on responses when compared with manual implementations that directly access variables without having to inspect their name in the beginning.

B. Eve

Eve [13] is also an open source framework in Python, and is built on top of the Flask microframework [14] that supports HTTP (Hypertext Transfer Protocol) I/O (Input/Output) operations and routing. In contrast to Django REST that supports four types of SQL (Structured Query Language) databases, Eve only supports non relational MongoDB databases.

```
DOMAIN = {
'categories': {
  'schema': {
    'name': {
      'type': 'string', 'maxlength': 50, 'required': True
    },
    'description': {
      'type': 'string', 'maxlength': 100, 'required': True
    }
  }
}}
```

Listing 2. Example of a simple API with the Eve framework

It is more based in specification rather than writing code requiring only the initialization of the *DOMAIN* variable, as shown in Listing 2.

C. LoopBack

LoopBack [15] is an open source Node.js framework [16], which means that is written in JavaScript. It is built on top of the Express framework [17] that provides a thin layer of Web application features. It considers relations between entities as resources, besides instances and collections of entities.

```
{
  "name": "Category",
  "plural": "categories",
  "base": "PersistedModel",
  "idInjection": true,
  "properties": {
    "name": { "type": "string", "required": true },
    "description": { "type": "string", "required": true }
  },
  "validations": [],
  "relations": {},
  "acls": [],
  "methods": []
}
```

Listing 3. Example of a simple API with the LoopBack framework

The framework tries to hide its inner workings by providing a command-line tool through which model entities are specified. In fact, this tool generates JSON (JavaScript Object Notation) files with the provided specification which may be edited, as shown in Listing 3. When the server application is started the model is interpreted and the correct dispatch functions are dynamically generated, similar to Eve and contrary to Django REST.

D. Sails

Sails [18] is an open source Node.js framework [16], and is also built on top of the Express framework [17] providing a Model-View-Controller (MVC) development architecture. It enables model-driven development by providing entity scaffolding (generation of code templates) using *sails generate api <entity_name>*.

```
module.exports = {
  attributes: {
    name: {
      type: "string", maxLength: 50, required: true },
    description: {
      type: "string", maxLength: 100, required: true }}};
```

Listing 4. Example of a simple API with the Sails framework

The developer must then complete the generated files with a specification of the entities, just like in Listing 4. Just like the previous examples the model is only known by the framework in run-time by importing the modules.

E. Conclusion

The identified frameworks are implemented in a dynamically typechecked language, either Python or JavaScript. Each of them implements model-based services with different approaches: class specialization in the case of Django REST; variable initialization in the case of Eve and Sails; and command-line interaction in the case of LoopBack.

 ISBN: 978-1-61208-458-9

III. Proposed Framework

Building high-quality frameworks is usually the result of many design iterations [19] using: either a *bottom-up* approach that starts with concrete applications and iteratively abstracts concepts into the framework; or *top-down* which relies on domain knowledge. The development of Metamorphic used a bottom-up approach. Considering architectures proposed by the online community, we built a non model-driven base application that followed a supposedly ideal architecture. In order to generate such kind of applications, we designed an internal DSL that relied on the use of macros. Macro architecture and tests were designed and implemented iteratively.

A. Application Architecture

Generated applications can either be *synchronous* or *asynchronous* but they have only one architecture (Figure 1). The architecture is not model centered, allowing generation of generic Web applications. This fact assures better software quality as components have to be less decoupled from their real use. Influenced by this decision, the architecture is composed of a mandatory layer, the *application logic*, and an optional layer, *data storage*, which may be used by the first layer.

The application logic is implemented by an *App* object that initiates services. These services may require access to data storage through repositories. A repository must have an entity associated with it and the whole application has access to a set of developer settings.

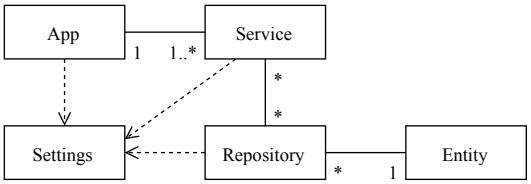

Figure 1. Architecture of a generated application.

To enable greater flexibility of generation, the application logic is defined by a model, as shown in Figure 2. The model allows the specification of services, which may have dependencies and a set of operations. Each operation implements an HTTP method for a path, expects the request body to be serializable for a specified class, and contains a body. It is at the operations level that one of the possible programming styles is applied, by using the *isAsync* flag.

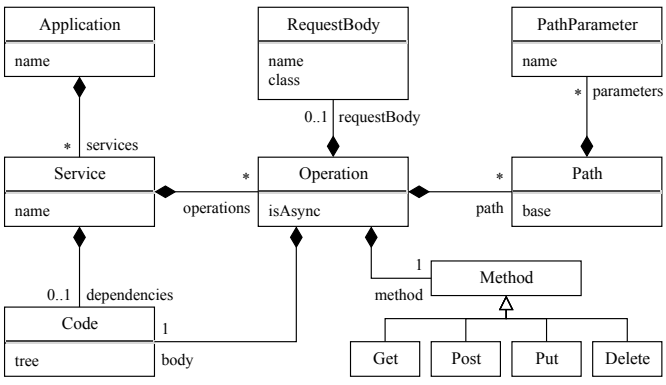

Figure 2. Application logic model.

To enable configurations, the *Config* [20] library was used, which enables the use of one file for setting configurations of all dependencies of a project. This means that besides Metamorphic's configurations, developers can still configure underlying libraries.

```
metamorphic {
  host = "111.111.111.111"
  port = 9000
  databases.default.name = "file.db"
}
```

Listing 5. Example of configuration file with a SQLite database

The configurations (Listing 5) are defined inside the *metamorphic* scope and shall be either host ("localhost" as default), port (8080 as default) or databases. Scopes inside *databases* may specify a name, an user, a password, an host, a port, a number of threads (numThreads) or a maximum queue size (queueSize).

B. Internal DSL

Scala macros enables generation of classes, traits and objects either through type providers or macro annotations [21]. The first discourages reuse of types in the scope calling the macros, while the second despite some limitations allows reuse. Through an internal DSL the framework makes use of these annotations.

Applications are identified by *@app* annotations in objects (Listing 6), which may have a set of entity definitions, a list of default operations, and a set of service definitions.

```
import metamorphic.dsl._
@app object PersonApp {

  @entity class Person {
    def fullname = StringField()
    def birthdate = DateField()
  }

  class PersonService extends EntityService[Person] {
    val operations = List(GetAll)

    def create(person: Person) = {
      if (person.fullname.length < 5)
        Response("Name is too short.", BadRequest)
      else
        super.create(person)
    }
  }
}
```

Listing 6. Example of simple API with the Metamorphic framework

The model specification follows the metametamodel in Figure 3 which is independent of any specification source such as this DSL. In this case entities can be defined using the *@entity* annotation in a class. The macro annotation expansion adds a companion object with replicated content, which helps to identify types errors as the compiler will typecheck the result after expansion. Entities are composed by fields which are case classes that accept a variable number of values to enable configuration and may be of type: *IntegerField*, *DoubleField*, *StringField*, *BooleanField*, *DateField*, *DateTimeField*, *ObjectField*, *ListField*, *ReverseField*.

All fields accept an *Option* argument, meaning that the property is not required. The first argument of a *ListField* or an *ObjectField* is the companion object of an entity definition. These two types of fields are by default mapped to many-to-many and one-to-many relations, respectively. The use of

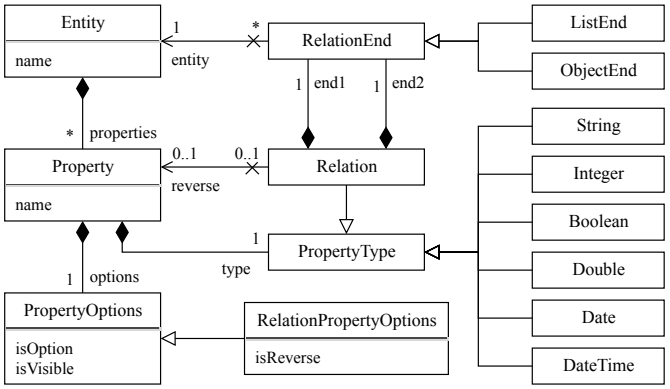

Figure 3. Framework's metametamodel.

R.Object as argument changes this behavior to many-to-one and one-to-one relations, respectively.

The generated entity operations can be the ones identified in Table I, which are implemented using entities plural as the base path. By default, each entity has all operations automatically implemented. Declaring the *operations* variable in the *@app* scope changes the set of default operations to be implemented. Operations are identified using the following objects: *Create*, *GetAll*, *Get*, *Replace*, and *Delete*.

TABLE I. ENTITY OPERATIONS SPECIFICATION

Operation	HTTP method	Path	Success code	Error codes
Create	POST	basePath/	201 (Created)	400 (Bad Request)
GetAll	GET	basePath/	200 (Ok)	-
Get	GET	basePath/:id	200 (Ok)	404 (Not Found)
Replace	PUT	basePath/:id	200 (Ok)	400 (Bad Request), 404 (Not Found)
Delete	DELETE	basePath/:id	204 (No Content)	404 (Not Found)

Changes to the default implemented operations are done via services, which can override the set of default operations for a particular entity. Operations implementations return a *Response* if synchronous or *Future[Response]* if asynchronous and can also be overridden. Customized operations can use a *repository* variable for accessing storage and use the keyword *super* for applying the default implementation.

C. Implementation

The classes and traits required by the DSL were created in the package *metamorphic.dsl* (Figure 4), including the *@app* annotation. The annotation implementation depends on the package *matcher* for translating the code tree into a metametamodel instance and an application logic model instance. There is also a dependency of package *generator* for mapping those models into the final application tree.

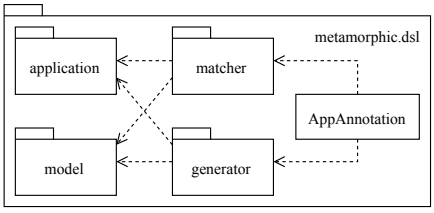

Figure 4. Diagram of packages for the Metamorphic framework.

The framework was designed to have a flexible and loosely decoupled architecture that allows long-term maintainability. With that in mind the *generator* package doesn't provide any concrete generation of applications, building them instead using dependency injection. The framework requires for a project to provide two dependencies/generators: a *repository generator* and a *service generator*.

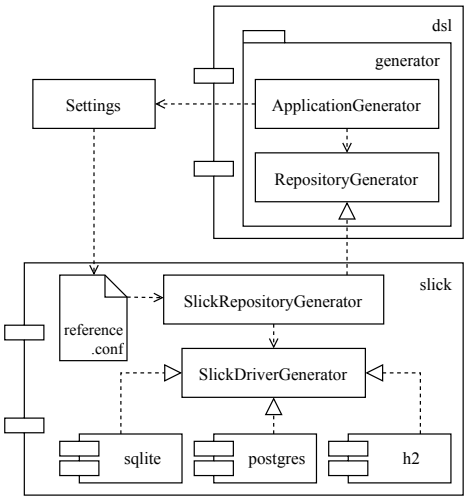

Figure 5. Dependency injection of a RepositoryGenerator that uses Slick.

To test this proof of concept, two repository generators were implemented based in the Slick library [22], one for synchronous applications and the other for asynchronous applications. Figure 5 illustrates how dependency injection is performed for a RepositoryGenerator that uses Slick, which also uses dependency injection to implement different database systems. We also implemented a service generator that uses components from the Spray toolkit [23].

IV. VALIDATION

It was expected that the developed framework would have the following characteristics:

- *Quick and easy to use*. Developers that look for these kind of frameworks want to have a Minimum Viable Product (MVP) as soon as possible without having to explore all the framework's documentation.
- *Error preventive*. Statically typechecked programs reassure developers about their code quality and reduce frustration when debugging.
- *Better response times*. Due to its programming language origin, improvements in performance should be noted when compared with existing model-driven frameworks.

The first and second characteristics were validated using *synthetic environment experiments* [24], in the form of an academic quasi-experiment. The last characteristic was validated through *dynamic analysis* [24], in the form of benchmarks.

A. Benchmarks

Comparing response times of different REST frameworks shall not be generic, i.e., comparison must be executed between services with the same characteristics. The categorization of compared services followed two properties: the type of entities and the type of operations. Entities may be: (i) *simple entities* which don't have navigable relations with other entities; (ii)

 ISBN: 978-1-61208-458-9

entities with objects which have a navigable relation with multiplicity one; (iii) or *entities with lists* which have a navigable relation with an infinite upper bound.

The same scenario was compared using seven different implementations: (i) synchronous version of Metamorphic; (ii) asynchronous version of Metamorphic; (iii) LoopBack; (iv) Sails; (v) Django REST in Python 2.7; (vi) Django REST in Python 3.4; (vii) Eve. All of these used a PostgreSQL 9.3.8 database (except Eve that used MongoDB) and were installed in production environments to guarantee maximum performance.

The experiment was executed in a portable computer Lenovo Thinkpad T430 with the following specifications: Ubuntu 14.04 LTS 32-bit; Intel Core i5-3210M @ 2.5GHzx4; 4GB Sodimm DDR3 Memory (1600 MHz); and 500GB 7200 RPM 32 MB Cache SATA Hard Drive. The tests were locally executed in an environment without Internet connection, all non-essential programs closed, and two terminals in foreground: server application and benchmark application.

All batches of operations were performed with a maximum of 10 concurrent requests. For each tuple, (Framework, Operation, Entity Type) 5000 requests were executed and, to discard any possible setup effects, 1000 requests were executed before testing each framework.

TABLE II. FRAMEWORK'S RANK BY ENTITY TYPE AND RANK SUM

Framework	Create	GetAll	Get	Replace	Delete	Sum
LoopBack	1 | 1 | 1	3 | 2 | 1	1 | 1 | 1	2 | 2 | 2	1 | 1 | 1	21
Metamorphic Async	2 | 2 | 2	1 | 1 | 2	3 | 2 | 2	1 | 1 | 1	2 | 2 | 2	26
Sails	3 | 3 | 6	2 | 6 | 4	2 | 3 | 3	3 | 3 | 6	3 | 3 | 5	55
Django REST 3.4	5 | 5 | 4	5 | 4 | 5	6 | 5 | 5	5 | 4 | 3	4 | 4 | 4	68
Django REST	4 | 4 | 3	6 | 5 | 6	5 | 6 | 6	4 | 5 | 4	5 | 5 | 3	71
Metamorphic	6 | 6 | 5	4 | 3 | 3	4 | 4 | 4	6 | 6 | 5	6 | 6 | 6	74
Eve*	1 | 1 | 1	4 | 3 | 2	4 | 3 | 3	1 | 1 | 2	3 | 3 | 3	35

* pseudo-rank; not comparable.

The frameworks were compared using rank sums. Table II presents the ranking of the frameworks by entity type in the following order: simple entities; entities with objects, and entities with lists. Considering the sum of these rank sums it is possible to conclude that, in spite of not being the best performant framework, the asynchronous version of Metamorphic already achieves performances better than most model-driven frameworks. In fact, without almost no effort to optimize framework's implementation, its results are close to the most performant framework, LoopBack, and it is the best solution to implement GetAll and Replace operations.

B. Academic Quasi-Experiment

8 MSc students in their 5^{th} year of the Master in Informatics and Computing Engineering, from the Faculty of Engineering of the University of Porto, were asked to participate. The experiment tested only one of the current model-driven frameworks against the synchronous version of Metamorphic.

All subjects started by answering a questionnaire and reading a problem guide. The subjects were split in two groups with two different treatments and had to perform the same set of tasks (Round 1). Then, each subject performed the same set of tasks as before with another treatment (Round 2). The test finished by answering another questionnaire.

The treatments were: *baseline treatment* - a default ready-to-use Django REST project; and *experimental treatment* - a default ready-to-use Metamorphic synchronous project. In both treatments, a guide about the framework and the language syntax were handed to the subjects. The experiment consisted in three tasks: (i) modeling using a UML diagram and the entities schema; (ii) creation of services operations for the entities; and (iii) customization of the defined operations.

Each subject executed the test in an isolated area of a low noise laboratory, with a single portable computer with Internet access mimicking a real programming situation. The subjects could only use a text editor of their choice and clarify any doubts they had. Application running and testing had to be done using the terminal. A *screencast* program was used to correctly measure time and development metrics in a non-intrusive way.

The questionnaires were designed with a five-point Likert scale [25]. Their responses were compared using the non-parametric, two-sample, rank-sum Wilcoxon-Mann-Whitney [26] test, with $n1 = n2 = 4$ and significance level of 5%. The results revealed statistical validity of the experiment and that implementing model-based operations is easier and more intuitive to do using Metamorphic ($\rho = 0.014$ in both rounds). Also, there is an high chance that implementations of models and customizations are easier and more intuitive ($\rho_1 = 0.043, \rho_2 = 0.200$ and $\rho_1 = 0.014, \rho_2 = 0.100$ respectively).

The framework may be considered quicker to use as time measurements indicate that development time may decrease 35%, and lines of code measurements indicate that the quantity of code may decrease 28%. As can be seen in Table III, modeling (Task 1) and operations definition (Task 2) may be implemented faster even after having knowledge about the problem (Round 2). The results of Task 3 were unexpected and may be explained by the reduced amount of requested customizations, that was 2.

TABLE III. STATISTICS OF TIME MEASUREMENTS (MINUTES)

Measurement	Round	$\bar{x}_E$	σ_E	$\bar{x}_B$	σ_B	$\bar{x}_B - \bar{x}_E$	$(\bar{x}_B - \bar{x}_E)/\bar{x}_B$
Task 1	1	11.44	03.11	20.13	09.18	08.69	43.2%
	2	07.94	01.48	08.81	03.06	00.87	09.9%
Task 2	1	08.31	06.46	22.94	13.73	14.63	63.8%
	2	10.69	05.76	17.25	01.49	06.56	38.0%
Task 3	1	05.06	02.12	03.92	02.67	-1.14	-29.1%
	2	05.31	01.91	03.44	00.97	-1.87	-54.4%
Total	1	45.81	08.19	70.56	12.14	24.75	35.1%
	2	31.50	01.86	51.38	08.29	19.88	38.7%

As illustrated in Table IV, applications built with the Metamorphic framework barely have runtime errors with measurements indicating the contrary for the baseline treatment. It should still be noted that the amount of non-runtime errors using the experimental framework are slightly lower than runtime errors using the baseline. This corroborates the error preventive nature as there are less errors that when occur are detected faster.

At last, the number of test executions measurement indicates that applications built with Metamorphic require less iterations to validate all the tests.

The results of this experiment should be carefully considered, as the number of subjects may not be representative of the developer community. In order to diminish this threat

 ISBN: 978-1-61208-458-9

TABLE IV. STATISTICS OF ERROR MEASUREMENTS

Measurement	Round	$\bar{x}_E$	σ_E	$\bar{x}_B$	σ_B
# Non-runtime errors	1	02.8	2.06	01.5	1.29
	2	04.3	1.71	00.3	0.50
# Runtime errors	1	00.0	0.00	05.8	4.99
	2	00.3	0.50	05.5	1.73

each subject executed the tasks for each of the frameworks increasing the number of data points.

V. CONCLUSION AND FUTURE WORK

This work required a full and extensive review on REST frameworks, specially model-driven REST frameworks, as very little is documented scientifically. With that knowledge a meta-architecture was developed, through modeling, that is independent of any programming language. A framework that follows such meta-architecture was also developed. This was designed to be as modular as possible by enabling the use of components through cross-project dependency injection.

Validation of the proof of concept revealed that the response to the research problem may be positive. This means that it is possible to improve the development process and execution performance of model-based REST services, through a framework that is written in a statically type-safe programming language that enables code generation in compile time. Improvements in the development process are backed by a reduced number of lines of code, a reduced number of runtime errors, and a reduced development time when using the proof of concept. Despite unexpected execution performances in some cases the authors believe the premise may hold for a more mature framework.

The only identified disadvantage of use of the framework is the additional development time required for compiling applications before testing. For big applications this may be critical as for any small change the entire code will be regenerated.

The source code of the framework can be found in https://github.com/frroliveira/metamorphic and more details about this research, such as the experiments can be found in http://paginas.fe.up.pt/ ei10038/dissert. Further research of the identified problem would be connected with Metamorphic as it is not yet a full featured framework. Future work could be:

1) *Test models flexibility.* Knowledge on Metamorphic's flexibility to other generators is empirical. To ensure such information another repository and service generators could be implemented based in other libraries.
2) *Extension of models.* For a framework to be useful, it may contain most features developers will need. Metamorphic has the basic features, so others would be welcome such as entity inheritance, authentication, database migrations, filtering, and pagination.
3) *Profiling.* Understanding any possible bottleneck in generated applications could be done trough profiling. This would aim to fully validate the performance goal initially established.
4) *Experiments.* Having a new and more mature version of the framework, its validity should be tested in industrial environments with samples that are more size significant. This time both versions, synchronous and asynchronous, should be tested.
5) *Swagger definition.* Swagger [27] defines a "standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code". This would allow faster testing of the generated services.

REFERENCES

[1] I. Society, "Global Internet Report 2014," 2014, URL: http://internetsociety.org/sites/default/files/Global_Internet_Report_2014_0.pdf [accessed: 01, 2016].

[2] J. Kopecký, P. Fremantle, and R. Boakes, "A history and future of web apis," it - Information Technology, vol. 56, no. 3, 2014, pp. 90–97.

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] P. K. Potti, S. Ahuja, K. Umapathy, and Z. Prodanoff, "Comparing performance of web service interaction styles: Soap vs. rest," in Proceedings of the Conference on Information Systems Applied Research ISSN, vol. 2167, 2012, p. 1508.

[5] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," Computer, vol. 39, no. 2, 2006, pp. 25–31.

[6] D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe, "The architecture of a uml virtual machine," SIGPLAN Not., vol. 36, no. 11, Oct. 2001, pp. 327–341.

[7] L. Cardelli, "Type systems," ACM Computing Surveys, vol. 28, no. 1, 1996, pp. 263–264.

[8] D. G. Bobrow, R. P. Gabriel, and J. L. White, "Clos in context-the shape of the design space," Object Oriented Programming: The CLOS Perspective, 1993, pp. 29–61.

[9] Ecole Polytechnique Fdrale de Lausanne - EPFL, "The Scala Programming Language," URL: http://scala-lang.org/ [accessed: 01, 2016].

[10] E. Burmako, "Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming," in Proceedings of the 4th Workshop on Scala, ser. SCALA '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:10.

[11] T. Christie, "Django REST framework," URL: http://django-rest-framework.org/ [accessed: 01, 2016].

[12] D. S. Foundation, "Django," URL: https://djangoproject.com/ [accessed: 01, 2016].

[13] N. Iarocci, "Python REST API Framework — Eve 0.5 documentation," URL: http://python-eve.org/ [accessed: 01, 2016].

[14] A. Ronacher, "Flask," URL: http://flask.pocoo.org/ [accessed: 01, 2016].

[15] StrongLoop, "LoopBack," URL: http://loopback.io/ [accessed: 01, 2016].

[16] N. Foundation, "About — Node.js," URL: https://nodejs.org/about/ [accessed: 01, 2016].

[17] Express, "Express - Node.js web application framework," URL: http://expressjs.com/ [accessed: 01, 2016].

[18] M. McNeil, "Sails.js — Realtime MVC Framework for Node.js," URL: http://sailsjs.org/ [accessed: 01, 2016].

[19] R. J. Wirfs-Brock and R. E. Johnson, "Surveying current research in object-oriented design," Communications of the ACM, vol. 33, no. 9, 1990, pp. 104–124.

[20] Typesafe Inc., "typesafehub/config," URL: https://github.com/typesafehub/config [accessed: 01, 2016].

[21] E. Burmako, M. Odersky, C. Vogt, S. Zeiger, and A. Moors, "Scala macros," November 2013, URL: http://scalamacros.org/paperstalks/2013-11-25-ScalaMacrosPoster.pdf [accessed: 01, 2016].

[22] T. Inc, "Slick," URL: http://slick.typesafe.com/ [accessed: 2016-01-05].

[23] Typesafe Inc, "spray — REST/HTTP for your Akka/Scala Actors," URL: http://spray.io/ [accessed: 01, 2016].

[24] M. V. Zelkowitz and D. R. Wallace, "Experimental models for validating technology," Computer, vol. 31, no. 5, 1998, pp. 23–31.

 ISBN: 978-1-61208-458-9

[25] R. Likert, “A technique for the measurement of attitudes.” Archives of psychology, 1932.

[26] M. Hollander, D. A. Wolfe, and E. Chicken, Nonparametric statistical methods. John Wiley & Sons, 2013.

[27] SmartBear, “Swagger — The World’s Most Popular Framework for APIs.” URL: http://swagger.io/ [accessed: 01, 2016].

 ISBN: 978-1-61208-458-9