A Metamodeling Framework for Promoting Flexibility and Creativity Over Strict Model Conformance

Nicolas Hili[1,2]

[1] School of Computing, Queen's University, Kingston, Ontario, Canada
hili@cs.queensu.ca
[2] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
nicolas.hili@imag.fr

Abstract. This paper defines FlexiMeta, a metamodeling framework intended to promote more flexibility and creativity while not compromising validation through model conformance. It advocates less coupling between models and metamodels in order to make the creation of models and user-defined metamodels possible in an arbitrary order. It comes along with a generic process structured into several phases. For each phase, a proper balance between flexibility and validation is found in order to bridge the gap between creativity and strict model conformance.

1 Introduction and Motivation

Metamodeling techniques [1] increase validation through model conformance. However, they drastically decrease flexibility as user-defined metamodels have to be created first and each change to the metamodels questions the validity of existing models [2]. Consequently, traditional Model-Driven Engineering (MDE) approaches fail to address some issues, such as the availability, evolution, and multiplicity of metamodels as well as the prototyping of models.

Whether introducing flexibility in MDE processes is efficiently addressed or not will ultimately depend on how they are realized by modeling frameworks. Consequently, considerations have to be made about the implementation of such frameworks. Particularly, design considerations of the internal representations of a model, given a *programming language* and a *data serialization format*, and their impact at both object- and meta-levels, should be cautiously made.

This paper presents FlexiMeta, a metamodeling framework for promoting flexibility and creativity during the model creation process over a strict model conformance according to a metamodel. This framework derives from previous lessons learned during an inter-organizational project involving both industrial and academic partners from the nuclear-plant system field [3]. In opposition to traditional MDE frameworks that promote a strict coupling between models and metamodels, less coupling between models and metamodels is advocated in

FlexiMeta, in order to address the problem of conception of models and user-defined metamodels in an arbitrary order, i.e., without assuming the existence of one before the other, and without assuming any relationship (one-to-one or one-to-many) between them.

This paper is structured as follows: Section 2 details FlexiMeta; Section 3 sketches a first implementation in which FlexiMeta is used; Related work is discussed in section 4; Section 5 concludes.

2 FlexiMeta

This section introduces FlexiMeta, a metamodeling framework which promotes more flexibility and creativity over a strict model conformance. Throughout this section, FlexiMeta is illustrated over the *Families* case study, a popular example to illustrate model-to-model transformations[3].

Fig. 1 shows the architecture of FlexiMeta, highlighting the internal representations (horizontally) of a model at object- and meta-levels (vertically). Dashed nodes and edges depict components that can be used at different times, depending on the intent of the modeling. Therefore, during the first phases of a project, relying on the metamodel is not required to create models. It prevents from validating the created models but it increases creativity and flexibility. On the other side, the metamodel definition is required if one wants to ensure model conformance. In other words, the proposed framework allows for balancing flexibility and validity during time and with respect to the intent of the modeling. In the following, we will describe each part of the framework.

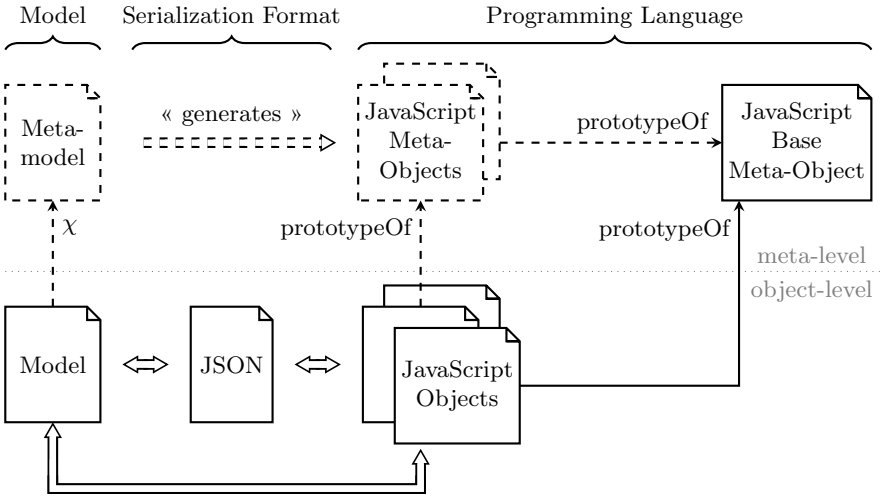

Fig. 1: Architecture of FlexiMeta

[3] `http://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf`.

Object Instanciation. JavaScript is used for instanciating model elements at run-time. JavaScript is a prototype-based, weakly-typed programming language that can be used to create interactive contents in a web-based environment or desktop stand-alone applications using NodeJS. New objects are instanciated by prototyping techniques, hence, no class or schema is required to create new objects. Consequently, even if the metamodel does not exist yet, objects can be created at run-time. JavaScript objects can have attributes and methods. An attribute can store a reference to another JavaScript object, a primitive value (i.e., Integer, Boolean, etc.), or a collection containing mixed elements (references and primitive values). A *Base* meta object is defined in JavasScript. This object defines a generic set of functions to access and edit properties of a model element. Each object of the model can then be instanciated by prototyping from the *Base* element. This minimal structure allows one to create models and to serialize them in JavaScript Object Notation (JSON) (see below).

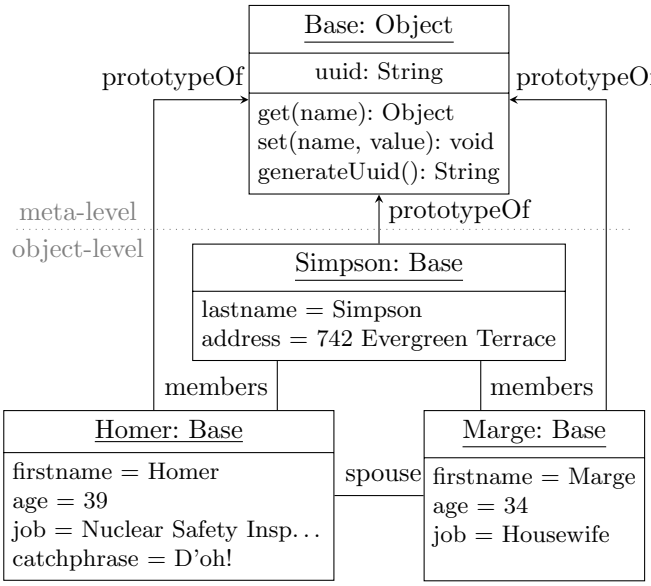

Fig. 2: Simpson Family

Fig. 2 depicts a Simpson family model. It includes a Simpson object with general attributes for the entire family (e.g., *last name* and *address*), and objects to model Homer and Marge Simpson.

Homer Simpson is defined by his first name, age, job and a catchphrase. Marge Simpson is defined by her first name, her age and her job. This example shows a basic instanciation of objects using FlexiMeta and the JavaScript *Base* meta-object.

The instanciated objects are not distinguishable by genre, as the model does not conform to a metamodel. Consequently, both the *Simpson* and the *Homer* objects are instanciated the same way. In FlexiMeta, each object is uniquely idenfitied by a Universal Unique IDentifier (UUID) which is generated during its instanciation by the *Base* meta-object. Object attributes and references are created using the `get` and `set` functions defined by the Base meta-object.

Code Generation. The JavaScript Base meta-object fosters creativity and flexibility by providing a minimal structure to extend. However, objects cannot

be validated according to a metamodel definition. FlexiMeta provides a code generation process to generate meta-objects from a metamodel. The code generation process is similar to how Eclipse Modeling Framework (EMF) generates Java classes from an Ecore metamodel. However, the main difference is that this step is optional, and one can simply inherit from the prototype of the Base meta-object, as described above.

Fig. 3 gives a glimpse of the artifact created during the code generation process. For each concept of the metamodel, a corresponding meta-object is created. The code generator allows for the generation of different artifacts, such as: (1) *getter* and *setter* functions for each attribute of the concept; (2) a *validate* function to ensure that the model object is well constructed with respect to the meta-object definition; (3) import and export functions to ensure the interoperability with Ecore models. It decreases flexibility and creativity but improves validation and model conformance.

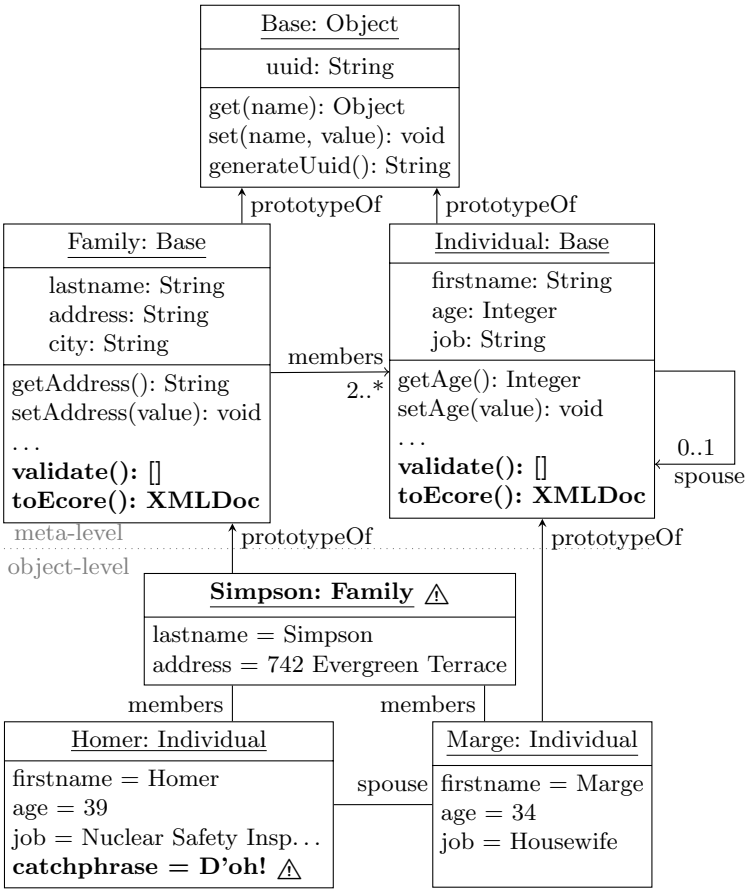

Fig. 3: FlexiMeta Code Generator

The code generator has been written using Acceleo and takes an Ecore metamodel as an input to generate JavaScript meta-objects that inherit from the Base meta-object (cf. Fig. 1). This choice was made to ensure interoperability between Ecore and FlexiMeta. The generator creates the required import and export functions to be able to import models from and export them to Eclipse.

This step is optional. When a metamodel is created, meta-objects can be generated from it. Existing objects that were previously typed with the *Base* meta-object can then be retyped using the newly generated meta-objects. Fig. 3 gives an example of the created meta-objects. Two meta-objects are created: *Family* and *Individual*. Each of them inherits from the *Base* meta-object, and getters and setters are generated from the metamodel. Two additional functions have been generated to validate the model and to export it into Ecore.

The *validate* function is recursive and detects different constraint violations. So far, it can check multiplicity violations, non-existence of required attributes, malformed value for enumerations, and existence of unexpected attributes. For example, in Fig. 3, the validator can detect that the *catchphrase* attribute of Homer Simpson is unexpected as it does not exist in the metamodel, and that the *city* attribute does not exist in the Simpson family object while it was defined as a required attribute by the metamodel. Once all these constraints are verified for one model object, the validate function hands over to the validate function of each composed element. It is worth noting that references and compositions are not distinguishable in JavaScript. Instead, this distinction exists in the metamodel. Consequently, the order of calls of the *validate* sub-function is inferred during the code generation process.

The *toEcore* function is generated from the Ecore metamodel and is specific to this metamodel definition. Listing 1.1 illustrates the export of the Simpson family model into Ecore. As the export feature is specific to the metamodel definition, compositions and references are observed to fit with the metamodel structure. In addition, unexpected object attributes are simply ignored and not exported. Finally, it is worth noting that, given the same set of attributes during the JSON export and the eXtensible Markup Language (XML) export, the size of the serialized model in JSON is reduced up to 40% compared to the size of the serialized model in XML. This result can be explained by the lightweight notation of JSON and shows how FlexiMeta addresses scalability.

Listing 1.1: Export into Ecore

```
1 <family:Family xmi:version="2.0"
2     xmlns:xmi="http://www.omg.org/XMI"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xmlns:family="http://family/0.1" xsi:schemaLocation="http://family/0.1
          family.ecore" lastname="Simpson" uuid="d229e52c-4e18-4261-9144-...">
5     <family:Individual uuid="6720a6c4-eedc-4b0c-..." age="43"></
          family:Individual>
6     <family:Individual uuid="cdb9eb06-4c13-492b-9262-..." age="49" spouse="
          6720a6c4-eedc-4b0c-..." ></family:Individual>
7 </family:Family>
```

Serialization. FlexiMeta can serialize models to and deserialize models from JSON. JSON is a standardized, platform-independent data serialization format that is natively supported by several languages without loading extra libraries. It has a very lighweight notation which makes it efficient to process. For example, it does not distinguish nested nodes from attributes[4] as XML does. Therefore, it may be less human-readable[5], however, it deters the temptation to replicate the metamodel structure to the serialized data.

For serializing models to and deserializing models from JSON, an opportunist serialization engine has been implemented. The term "opportunist" designates that the serialization engine serializes data "*as it arrives*". It brings several benefits. It is not specific to a metamodel, and is therefore generic. Consequently, it can be used to serialize and deserialize models even if the metamodel is implicit or not formalized. If the metamodel does not exist, the deserialization engine will instanciate objects using the *Base* meta-object that has been defined. If the metamodel is defined, then the deserialization process deserializes the model by using the generated meta-objects defined during the code generation process.

Listing 1.2: Serialization in JSON

```
{
  uuid: "d229e52c-4e18-4261-...",
  lastname: "Simpson",
  address: "742 Evergreen Terrace",
  members: [
   {
     uuid: "cdb9eb06-4c13-492b-...",
     firstname: "Homer",
     age: "49",
     job: "Nuclear Safety Inspector",
     catchphrase: "D'Oh!",
     spouse: {
      uuid: "6720a6c4-eedc-4b0c-...",
      firstname: "Marge",
      age: "43",
      job: "Housewife"
     }
   },
   "6720a6c4-eedc-4b0c-...",
  ]
}
```

Listing 1.2 shows how the model is serialized in JSON. An object ({}) contains several key-value pairs separated with commas. E.g., the *Simpson* object has four pairs: *uuid*, *lastname*, *address*, and *members*. The value of the *members* key is an array ([]) of objects. Listing 1.2 illustrates how data is serialized as it arrives. When the serializer has to process the *spouse* attribute of the *Homer* object, the *Marge* object has not been serialized yet. Consequently, the *Marge* object is serialized inside the *spouse* attribute. When it comes to serialize the *Marge* object in the *Simpson's members* attribute, only its UUID is serialized.

3 Implementation

A preliminary implementation of a web-based modeling environment has been developed to allow everyone to exercise FlexiMeta[6] (cf. Fig. 4). It allows for the creation of the Simpson family model. It is composed of several areas.

The main area (1) depicts a graphical editor to edit the *Simpson Family* model. The model consists of the family composed of several members. Model

[4]The concept of attributes does not exist in JSON.

[5]It is controversial though, as the learning curve to understand JSON is smoother thanks to its light notation.

[6]This environment is available at `http://fleximeta.net/demo-models2016`.

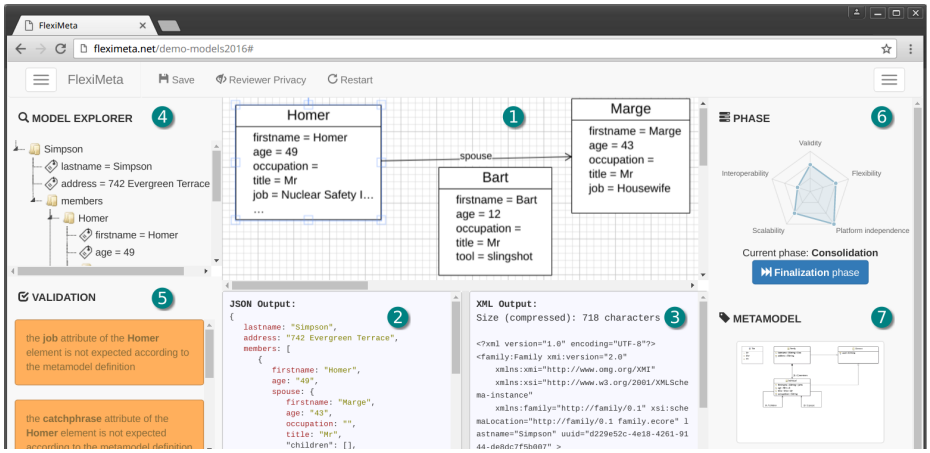

Fig. 4: Case Study: The Simpson Family

editing can be done using the contextual menu by right-clicking on the different graphical elements. Below the graphical editor, two views (2) and (3) allow for model serialization in both JSON and XML. JSON serialization is available every time while XML serialization can only be done when the metamodel is known (i.e., after the meta-objects are generated from the metamodel). On the left-hand side of the graphical editor, a model explorer (4) offers a tree view representation of the model. It is updated every time the model changes. Below the model explorer, a validation view (5) displays the list of conformance errors that occur on the model. As for the XML exporter view, validation can only be processed after the code is generated.

FlexiMeta comes along with a generic process structured into three phases – *exploration*, *consolidation*, and *finalization* –, which address specific intents. The exploration phase allows the user to create models without relying on user-defined metamodels. It promotes creativity but prevents from validating models as no metamodel has been defined yet. In the consolidation phase, the metamodel is known and this phase is used to align the created models to fit with the metamodel definition. Therefore, validation is possible, but it is still possible to create models which do not conform with the metamodel. Finally, the finalization phase is intended to create only valid models regarding the metamodel definition. This generic process has been proposed to offer a trade-off between flexibility and validation at specific times during the creation of models. Due to the limited space, the generic process is not further detailed.

To go through the generic process, a view (6) reminds the user in which phase he or she is. A radar chart displays level and comment for each challenge FlexiMeta intends to address during the current phase. In addition, a button allows the user to move forward to the next phase. Finally, a last view (7) illustrates the metamodel used during the *consolidation* and the *finalization* phases.

4 Related Work

A significant body of literature addresses the problem of flexibility in existing MDE approaches. *Model / metamodel co-evolution* techniques were proposed to withstand metamodel evolution and to automatically or semi-automatically adapt models [4, 5]. These techniques usually rely on the identification of some transformation patterns (e.g., creation of new concepts, deletion of existing concepts, addition to some attributes, etc.). Unfortunately, they cannot be fully automated as some model evolutions cannot be inferred. At best, a variation could be semi-assisted by a human intervention. At worst, a variation could require to design a specific model transformation to migrate existing models.

Bottom-up metamodeling techniques consist in inferring what the metamodel should be regarding a set of existing models [2,6,7]. It allows for the creation of a model independent of the metamodel definition. By analogy, we can compare this approach with NoSQL databases from which schemas do not have to be defined. In that case, the metamodel exists, implicitly behind the inference mechanism. As of the first technique, some automations could not be fully automated. For instance, it is not possible to infer the multiplicity of a relation.

Metamodel extension techniques leverage the use of general-purpose modeling languages, such as Unified Modeling Language (UML), instead of defining new metamodels from scratch [8–10]. Several mechanisms exist. For example, UML can be extended using the UML profile mechanism. *Metamodel extension* answers to specific challenges, such as the availability of user-defined metamodels, and model prototyping, as the existing metamodel to extend already exists. Then, even if the extension has not been defined yet, it is still possible to quickly prototype the system to define.

As for *Multi-level modeling* techniques [11] and tools (e.g., MetaDepth [12]), the same distinction is made between linguistic (i.e., the language) and ontological type. With respect to the nature of the prototype relation in JavaScript, objects in FlexiMeta can be created with no ontological type first, and then be further retyped given a user-defined metamodel. In addition, the framework precisely defines how and when (i.e., during which phase) objects are retyped, in order to align the model definition with the user-defined metamodel.

Existing techniques usually address specific and localized issues, such as metamodel evolution or model co-adaptation. Our work intends to address all the aforementioned challenges at once. An attempt was done to address them in our previous work [3], where a *metamodel-tolerant approach* and a *loose-model conformity control* were defined, but the problem space and the methodology to address it were both not formalized. In FlexiMeta, a model can be created before the metamodel (model prototyping). Once the metamodel is defined, model conformance can be checked to help designers migrate both the model and the metamodel (metamodel evolution and co-adaptation).

Several modeling frameworks were proposed over time. EMF [13] and Visualization and Modeling SDK (VMSDK) [14] rely on a code generation approach. In Eclipse, Java classes are generated for each concept of the metamodel. Models are serialized in XML using a specific serializer generated alongside the Java

implementation of the metamodel concepts. The mapping between the modeling language, data serialization format, and programming language appears at the object-level and the meta-level. Unfortunately, the combination of the class-based programming language with a schema-dependent data serialization format severely affects the flexibility during the modeling activities. Some attempts were done to bring capabilities of EMF into web-based environments. EMF-Rest [15,16] is a framework that intends to bring EMF capabilities through a REST API. Another interesting framework is Ecore.js[7], which is developed in JavaScript and available through NodeJS.

Moddle[8] is a utility library for creating user-defined metamodels using JavaScript and JSON. A metamodel can be defined at run-time and models can be created using it. However, models cannot be created without defining the metamodel in JSON first. Besides, models cannot be validated and Moddle provides a limited model coverage. For example, it is not possible to define enumerations.

MoDiGen [17] is an interesting work to address the scalability challenge. Models and metamodels are defined using JSON. However, it seems that there is no implementation to use it as no programming language is mentioned. Moreover, there is no mention about how models are concretely "instanciated" from the metamodel definition in JSON.

Compared to other frameworks, FlexiMeta relies on a code generation process to generate JavaScript implementation of concepts from external metamodels (Ecore metamodels, so far). This part is not mandatory though, and one can use the minimal implementation using the *Base* JavaScript meta-object (cf. Section 2). Flexibility and validation challenges are not addressed simultaneously and can be balanced over time, which allows one to benefit from both.

5 Conclusion

This paper introduces a new metamodeling framework for promoting more flexibility when creating models and metamodels. Unlike existing approaches, it balances flexibility and strict model conformance throughout the development life-cycle. To do so, less coupling between a model and a metamodel is advocated to give more freedom during the modeling activities. A preliminary tool has been sketched to exercise the new metamodeling framework.

The prototype-based programming style of JavaScript, combined to the use of a schema-free data serialization format opens up some interesting horizons for the development of new modeling frameworks. For instance, JavaScript supports the dynamic creation of meta-objects at run-time and at different levels of modeling (deep instanciation). FlexiMeta could take advantage of it to support the binding of models to several metamodels or several versions of the same metamodel at the same time, in order to adress issues such as the multiplicity of metamodels and metamodel evolution.

[7] Available here: `https://github.com/emfjson/ecore.js`.
[8] Available here: https://github.com/bpmn-io/moddle.

References

1. F. Fondement and R. Silaghi, "Defining model driven engineering processes," in *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*, 2004.
2. P. Gomez, M. E. Sánchez, H. Florez, and J. Villalobos, "An approach to the co-creation of models and metamodels in Enterprise Architecture Projects," *Journal of Object Technology*, vol. 13, no. 3, pp. 2–1, 2014.
3. N. Hili, Y. Laurillau, S. Dupuy-Chessa, and G. Calvary, "Innovative Key Features for Mastering Model Complexity: Flexilab, a Multimodel Editor Illustrated on Task Modeling," in *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '15, 2015.
4. G. Wachsmuth, "Metamodel Adaptation and Model Co-adaptation," in *European Conference on Object-Oriented Programming*, ser. ECOOP'07, E. Ernst, Ed.
5. A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio, "Automating Co-evolution in Model-Driven Engineering," in *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE*, 2008, pp. 222–231.
6. J. Sánchez-Cuadrado, J. Lara, and E. Guerra, "Bottom-up meta-modelling: An interactive approach," in *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012*. Springer Berlin Heidelberg, 2012.
7. H. Cho, J. Gray, and E. Syriani, "Creating Visual Domain-specific Modeling Languages from End-user Demonstration," in *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, ser. MiSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 22–28.
8. I. Weisemöller and A. Schürr, "A Comparison of Standard Compliant Ways to Define Domain Specific Languages," in *Models in Software Engineering*.
9. B. Selic, "A systematic approach to domain-specific language design using uml," in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, 2007, pp. 2–9.
10. P. Langer, K. Wieland, M. Wimmer, J. Cabot *et al.*, "Emf profiles: A lightweight extension approach for emf models." *Journal of Object Technology*, vol. 11, no. 1, pp. 1–29, 2012.
11. C. Atkinson and T. Kühne, "The Essence of Multilevel Metamodeling," in *International Conference on the Unified Modeling Language*. Springer, 2001, pp. 19–33.
12. J. de Lara and E. Guerra, "Deep meta-modelling with metadepth," in *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, ser. TOOLS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–20.
13. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF Eclipse Modeling Framework*, ser. The Eclipse Series. Addison Wesley, 2009.
14. S. Cook, G. Jones, S. Kent, and A. Wills, *Domain-specific Development with Visual Studio Dsl Tools*, 1st ed. Addison-Wesley Professional, 2007.
15. B. Costa, P. F. Pires, F. C. Delicato, and F. Oquendo, "Towards a View-Based Process for Designing and Documenting RESTful Service Architectures," in *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ACM, 2015, p. 50.
16. H. Ed-Douibi, J. L. C. Izquierdo, A. Gómez, M. Tisi, and J. Cabot, "Emf-rest: Generation of restful apis from models," *arXiv preprint arXiv:1504.03498*, 2015.
17. M. Gerhart, J. Bayer, J. M. Höfner, and M. Boger, "Approach to define highly scalable metamodels based on json," *BigMDE 2015*, p. 11, 2015.