IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 11, 2019, accepted April 27, 2019, date of publication May 7, 2019, date of current version May 17, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2915201

A Model-Driven Approach to Generate Schemas for Object-Document Mappers

ALBERTO HERNÁNDEZ CHILLÓN[1], DIEGO SEVILLA RUIZ[1], JESÚS GARCÍA MOLINA[1], AND SEVERINO FELICIANO MORALES[2]

[1]Faculty of Computer Science, University of Murcia, 30100 Murcia, Spain
[2]Faculty of Engineering, Autonomous University of Guerrero, Chilpancingo 39087, Mexico

Corresponding author: Alberto Hernández Chillón (alberto.hernandez1@um.es)

This work was supported in part by the Spanish Ministry of Science, Innovation and Universities, under Grant TIN2017-86853-P.

ABSTRACT Many actual NoSQL systems are schemaless, that is, the structure of the data is not defined beforehand in any schema, but it is implicit in the data itself. This characteristic is very convenient when the data structure suffers frequent changes. However, the agility and flexibility achieved is at the cost of losing some important benefits, such as 1) assuring that the data stored and retrieved fits the database schema; 2) some database utilities require to know the schema, and; 3) schema visualization helps developers to write better code. In previous work, we proposed a model-based reverse engineering approach to infer schema models from NoSQL data. Model-driven engineering (MDE) techniques can be used to take advantage of extracted models with different purposes, such as schema visualization or automatic code generation. Here, in this paper, we present an MDE solution to automate the usage of Object-NoSQL mappers when the database already exists. We will focus on mappers that are available for document systems (Object-Document mappers, ODMs), but the proposed approach is mapper-independent. These mappers are emerging to provide similar functionality to Object-Relational mappers: they are in charge of the mapping of objects into NoSQL data (documents in the case of ODMs) for object-oriented applications. We show how schemas and other artifacts (e.g. validators and indexes) for ODMs can be automatically generated from inferred schemas. The solution consists of a two-step model transformation chain, where an intermediate model is generated to ease the code generation. We have applied our approach for two popular ODMs: Mongoose and Morphia and validated it with the StackOverflow dataset.

INDEX TERMS Object-document mappers, NoSQL databases, Mongoose, Morphia, code generation, model-driven engineering, model-based solution, NoSQL data engineering.

I. INTRODUCTION

NoSQL (Not only SQL) database systems emerged to tackle the data challenges posed by moderns applications (e.g., Big Data, social media, and mobile apps). These applications evidenced the limitations of relational systems to meet their scalability, availability, and performance requirements. Over the last decade, about two hundred NoSQL systems have appeared [1]. NoSQL databases are used in well-known applications (e.g. Bigtable in Google applications, Cassandra in Facebook, and Dynamo in Amazon applications), and are part of popular software architectures (e.g. the MEAN stack includes MongoDB). Although relational systems are still predominant in the database market, interest in NoSQL is continuously growing, and some reports predict the adoption will considerably rise in next years [2], [3].

NoSQL systems are usually classified into four categories: document, wide column, key-value stores, and graph-based databases. Document databases are the most widely used NoSQL systems. In particular MongoDB appears as the fifth in the most popular database ranking [4]. The absence of an explicit database schema is one of the most attractive characteristics of NoSQL systems. Being schemaless, a great flexibility is achieved in the data management. A data schema (e.g. a relational schema) imposes restrictions on the structure of the data to be stored. Instead, in schemaless databases, data stored for an entity or type can have different structure. This facilitates managing data variation (e.g. non-uniform types or optional fields), and tackling new data requirements (e.g. migrations are easier) [5]. However, this greater

The associate editor coordinating the review of this manuscript and approving it for publication was Giambattista Gruosso.

flexibility entails losing the benefits of using data schemas in coding database applications. Schemas allow a static checking that assures that only data that fits the schema can be manipulated in application code. When the schema is only in the mind of developers, they must guarantee a correct access to data, and code is then more prone to errors. In addition, a schema is often convenient, because database applications require knowing the data organization in order to manage data efficiently.

The convenience of alleviating problems caused by the lack of schema is noted in a Dataversity report [2]. This report draws attention to the need for NoSQL tools just as they exist for relational databases. Model visualization and code generation are identified as two needed capabilities for NoSQL modeling tools, which require the extraction of the database schema. In [6], we presented an approach to extract NoSQL schemas, and here we show how the inferred schemas can be used to generate code of database applications. More specifically, software artifacts of NoSQL mappers, such as schema definitions, indexes, and data validators.

As with relational systems, mappers have become available to facilitate the storage of objects into NoSQL databases when developers create object-oriented applications. Although these mappers have been developed for the four categories of NoSQL systems [7], mappers for document stores (Object-document mappers, ODMs) are the most widely used, in particular those created for MongoDB, such as Mongoose [8] and Morphia [9]. When mappers are used, the database schema must be declared, and developers are freed from checking whether the stored data conforms to the schema.

Data schemas are models, and operations on them can be implemented using model transformations [10]. Therefore, Model-Driven Engineering (MDE) techniques [11], are appropriate in data engineering as illustrated in [12]. In this paper, we present an MDE-based solution to automate the usage of ODMs when the database already exists. We have devised a transformational approach that generates the main artifacts involved in any ODM from a NoSQL schema model extracted by applying the strategy described in [6]. For each database entity, a schema declaration, indexes, and data validators, among other artifacts, can be automatically generated from the inferred schema. These artifacts are not directly generated, but an intermediate step is introduced to tackle the complexity imposed by the existence of more than one variation for data entities. This step facilitates the generation of mapper code. The approach proposed has been validated for a database populated from a StackOverflow dataset.

The main research contributions of our work are the following. First, we present a novel approach to automate the use of NoSQL mappers for existing databases. As far as we know, our solution is the first work published with that purpose. Second, our solution shows how schemas inferred from NoSQL systems may be used to generate code. This is one of the three main objectives to be pursued by NoSQL tools, as noted in the Dataversity report [2]. Third, usage scenarios of mappers for existing databases are analyzed and some benefits are exposed. Fourth, the approach illustrates some of the advantages of using MDE techniques in database engineering, which are discussed in [12]. Metamodels have allowed us to represent schemas at a higher level of abstraction, and take advantage of model transformations to automate the solution. Finally, we have explored the use of the YAML [13] language to configure the model-to-text transformation that generates mapper code.

A preliminary version of our approach was presented in the Modelsward conference [14]. That previous work has been extended as follows:

- We have considered a Java ODM in addition to Mongoose. In particular, we have generated code for Morphia, an annotation-based Java mapper.
- We have changed the mechanism to configure the model-to-text transformation. Instead of a metamodel-based textual language, a YAML notation has been created to ease the extensibility.
- We have defined a validation process which has been applied to a MongoDB database populated from a Stackoverflow dataset.
- A union type has been defined for Mongoose in order to deal with properties of the same name (and different type) in an entity. In the case of Morphia, a strategy has been devised to manage unions.

In this paper, we will discuss all the issues mentioned above. Moreover, we added a motivation which analyzes the usage scenarios in which the approach could be useful, and a background section.

This article has been organized as follows: The next section provides the background required to understand the approach presented. Then, we motivate our work and present an overview of our solution. The three following sections describe the model-driven solution implemented: Section IV explains how NoSQL schema models are transformed into *Entity Differentiation* intermediate models; Sections V and VI describe, respectively, the generation of Mongoose and Morphia schemas; and Section VII describes the generation of additional artifacts such as indexes and validators. Then, the validation process is explained. Finally, related work is discussed, some conclusions are drawn, and further work is outlined.

II. BACKGROUND

This section introduces some basic concepts which will help to understand this paper. In particular, we will define the notion of aggregation-oriented data model, describe some essential ideas behind ODMs, and introduce some MDE basic concepts. A running example of a document database is also presented.

A. AGGREGATION-ORIENTED DATA MODEL

The NoSQL term refers to the wide variety of database systems built to manage semi-structured and unstructured data in modern applications, where high availability, scalability, and fault tolerance are demanded. The existing NoSQL

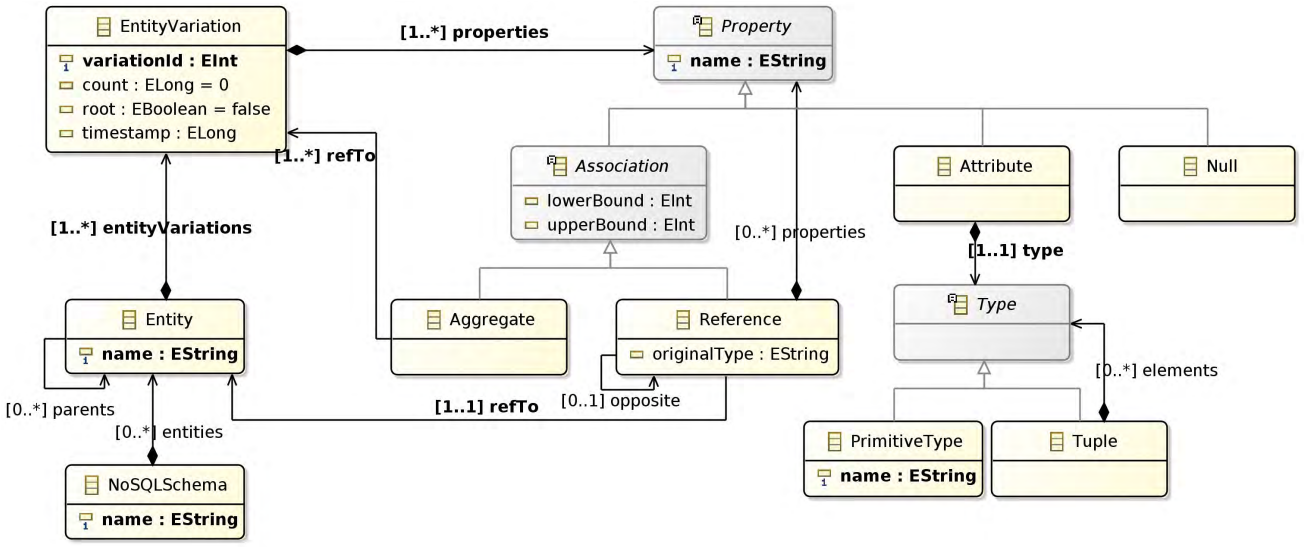


FIGURE 1. NoSQL schema metamodel.

systems have different features, although most of them share a few common properties, namely: they are schemaless, SQL language is not used, the execution on clusters is the main factor that determines its design, and they are developed as open-source initiatives.

Data are always stored following the structure that determines a schema previously devised, which can be either formally specified or kept into the mind of the developers. Therefore, not requiring the definition of the schema does not mean that there is no database schema: It is implicit in the data and in the application code. The form of a schema is determined by the database paradigm (i.e. the data model.) We will focus here on the aggregation-oriented data model.

NoSQL systems are classified into four main categories according to the underlying data model: document, wide column, key-value stores, and graph-based databases. The three former are categorized as ''aggregation-oriented databases,'' as they tend to favor aggregation over reference in the database entity representation, unlike graph databases [15]. These three aggregation-oriented data models represent information as semi-structured data: i.e. objects formed by a set of key-value pairs [16]. The form of keys and values differentiates each of the three paradigms. Most of the key-value systems do not assume any structure on the values, and treat them as blobs of information. Instead, the value takes the form of a structured *document* (normally a JSON-like document) in a document system. In addition, a document database is usually organized into a set of collections, and each collection contains the documents stored for a kind of database entity (e.g. artist or album). Then queries can be issued on collections. Finally, wide column systems are organized as a collection of rows, each of them consisting of a row key and a potentially different set of columns, each identified by its name.

In aggregation-oriented databases, complex objects are formed by embedding objects. An object structure consists of a root object that recursively embeds other objects, so that an aggregation hierarchy is formed. The use of references among objects should be very limited. Like joins in relational databases, references are explicitly managed by developers.

As previously indicated, objects of the same entity can be stored with different structure in a schemaless database. These structural variations can arise due to the need of having different variants of an entity (i.e. non-uniform types) or due to changes made during the database evolution to satisfy new requirements. The notion of NoSQL schema must take into consideration such a variation. In [6], we proposed a definition of NoSQL schema for aggregation-oriented systems, and described a schema inference strategy. Compared with other proposals [17], [18], the main novelty of our approach is discovering all the variations of the inferred entities and their relationships (i.e. aggregations and references). Moreover, we defined the *NoSQL Schema* metamodel shown in Figure 1 to represent the schemas inferred from stored data. The EMF/Ecore framework [19] was used to implement our metamodel. Representing schemas as Ecore models, we can take advantage of model-driven technology to build NoSQL database utilities.

In our metamodel, an *Entity* labels all the stored objects that refer to the same physical or conceptual thing (e.g. artist or album). An *Entity Variation* denotes each of the sets of objects that, sharing the same entity label, have a different structure. Each entity will have one or more entity variations. Each entity variation has one or more named *Properties*. Properties can be of three kinds: *Attributes*, and either *Aggregation* or *Reference* relationships (i.e. associations) among variation entities. These properties represent the key-value pairs of a entity variation, where the values can be:

```
[{"artist": [
 {
  "_id": "1",
  "name": "Pink Floyd",
  "startingYear": 1965,
  "albums": ["2","3"],
  "composedTracks": ["4","5","6","7","8","9"],
  "lyricsTracks": ["4","5","6","7","8","9"]
 }, {
  "_id": "101",
  "name": "Massive Attack",
  "startingYear": 1988,
  "albums": ["102","103"],
  "composedTracks": ["104","105","106","107","108"]
 }],
 "album": [
 {
  "_id": "2",
  "name": "Wish you were here",
  "genre": "Progressive rock",
  "releaseYear": 1975,
  "availability": ["EN","FR"],
  "formats": ["Vinyl","Album"],
  "tracks": ["4","5","6"],
  "reviews": [{
    "journalist": "Ben Edmonds",
    "media": "The Rolling Stone Album Guide",
    "rank": "Excelent",
    "stars": "5 stars"}]
 }, {
  "_id": "3",
  "name": "The Wall",
  "genres": ["Art rock"],
  "releaseYear": 1979,
  "availability": ["EN","FR","ES","JP","PT","NE"],
  "formats": ["Vinyl","Album"],
  "tracks": ["7","8","9"]
 }, {
  "_id": "102",
  "name": "Mezzanine",
  "genre": "Trip hop",
  "releaseYear": 1998,
  "availability": "ES, EN, FR",
  "formats": ["LP", "Album"],
  "tracks": ["104", "105", "106"],
  "prizes": [{
    "certification": "Platinum disk",
    "event": "RIAA",
    "name": "Australia",
    "units": 70000,
    "year": 1998
  },
  {
    "event": "SNEP",
    "names": ["France", "2x Gold disk"],
    "units": 243000,
    "year" : 1998
  }],
  "reviews": [{
    "journalist": "Barney Hoskyns",
    "media": "Rolling stone",
    "rank": "Very good",
    "stars" : "3.5/5"
  }, {
    "journalist": "Alexis Petridis",
    "media": [{
      "name": "The Guardian",
      "type": "newspaper",
      "url": "https://www.theguardian.com/us"
      }],
    "rank": "Excelent",
    "stars" : 5
  }]
 }],
 "track": [
 {
  "_id": "4",
  "artist_id": ["1"],
  "genres": ["Progressive rock"],
  "length": 13.32,
  "name": "Shine on you crazy diamond (I-V)",
  "ratings": [{
    "score": 22.7,
    "voters": 1273
  }]
 }, {
  "_id": "7",
  "artist_id": ["1"],
  "genres": ["Art rock", "Progressive rock", "Disco"],
  "length": 3.09,
  "name": "Another Brick in the Wall (Part I)"
 }, {
  "_id" : "104",
  "artist_id" : ["101"],
  "genres": ["Trip hop", "Industrial rock"],
  "length": 6.19,
  "name": "Angel",
  "ratings": [{
    "score": 18.0,
    "voters" : 180
  }]
 }]
}]
```


FIGURE 2. JSON documents in the database example.

- *Attribute*: primitive values (i.e. atomic values of types such as `Number`, `String`, `Char`, or `Boolean`), or some kind of collection (tuple).
- *Aggregation*: An object.
- *Reference*: A `String` or `Number` value that references another object.

Entities (and therefore *Entity Variations*) can be either root or nested, depending on whether they are obtained as first level objects from the database or they are embedded into other objects. Note that an entity is characterized by a set of properties that can be either *common* to all entity variations (i.e. they are part of the all objects of the entity), or *specific* to one or more entity variations.

B. DATABASE EXAMPLE

JSON [20] is a standard human-readable text format widely used to represent semi-structured data. In the majority of aggregate-oriented systems, data are stored and retrieved as JSON objects. Here, we introduce a set of JSON objects that will be used throughout this article as a running example of a document database.

Figure 2 shows the JSON documents of the *Songs* database example. The database has a collection for each type of root entity: *Artist*, *Album*, and *Track*. *Artist* has a couple of variations because *lyricsTracks* only appears in one of them. Three variations are identified for the *Album* entity, which are motivated by: (i) the existence of *prizes* and *reviews* fields, and (ii) changes applied on this entity due to database evolution: the *genre* field was replaced by the *genres* field and the type of the *availability* field was changed from string to tuple of strings. The *Track* entity two variations depending on whether the *ratings* field is present. In addition to the root entities that correspond to the three collections, four embedded entities can be identified: *Rating*, *Prize*, *Review*, and *Media*. These entities are involved in four aggregation relationships: *Album* aggregates *Review* and *Prize*, *Track*

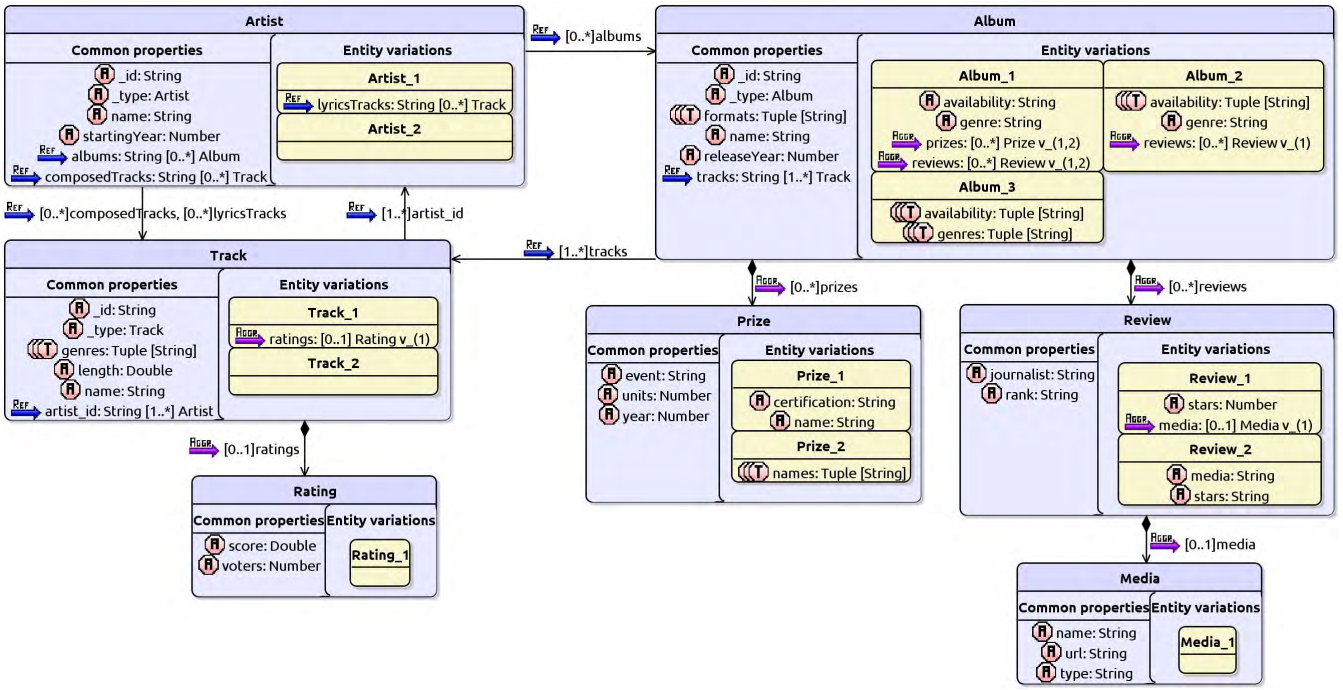


FIGURE 3. Visualization of the database schema for the running example.

aggregates *Rating*, and *Review* aggregates *Media*. Regarding to reference relationships, *Artist* refers to *Track* (*lyricsTracks* and *composedTracks* fields), *Album* refers to *Track* (*track* field), and *Track* refers to *Artist* (*artist_id* field).

Figure 3 shows the database schema diagram for the running example. This diagram has been automatically generated using the visualization tool that we presented in [21]. In that paper, we proposed several kinds of schemas for aggregation-oriented NoSQL data models, and a notation for each of them. This tool visualizes schemas inferred by applying the process defined in [6], which represents the inferred schema as an instance of the metamodel shown in Figure 1. The diagram shown in Figure 3 represents a *global schema* with all the elements (entities, attributes, and relationships) of the database. Each entity is represented as a rectangle with two compartments: the left one shows the names of the common properties, and the right one encloses the entity variations; each entity variation is represented by a rectangle that shows the non-common properties.

C. OBJECT-DOCUMENT MAPPERS

Object-relational mappers (ORMs) are essential to build object-oriented applications whose persistence is based on a relational database. They provide transparent persistence by freeing developers from converting objects into tuples. For this, they use the relational database schema to establish the mapping between persistent objects and tables. Mappers are a typical tool for relational databases that are also useful for NoSQL systems. In fact, several NoSQL mappers have appeared in the previous years, as interest in NoSQL systems has been growing [7]. Some of them have been specifically developed for some particular NoSQL system [8], [9], but some widely used ORM mappers have also been extended to support NOSQL databases [22]. Since document systems are the most widespread NoSQL systems (in particular MongoDB) most of the available NoSQL mappers are oriented towards object-document mapping (ODM) [8], [9], [23], [24]. In this paper, we focus on MongoDB ODMs, but the proposed approach is applicable to other mappers for any aggregation-oriented system.

When using NoSQL mappers, developers must define a data schema, e.g. by using JSON [8], Java classes with annotations [9], or a domain-specific language [24]. These mappers can take advantage of the defined schema to statically check that data are correctly manipulated in application code, and mistakes made by developers are spotted as early as possible. This is a significant benefit provided by NoSQL mappers, which frees developers of assuring that data are accessed in a correct way. Data validation is an error-prone task when variations are added into the mix. However, the need to define a schema is conflicting with the schemaless nature of NoSQL systems, and the use of NoSQL mappers could only be justified when changes to the schema are not frequent.

Developers have therefore two alternatives in building NoSQL database applications. They can work in a schemaless way, or use an ODM, by deciding on the trade-offs between flexibility and safety: they could prefer not having the restrictions posed by schemas or either avoid the data validation.

Next, we will briefly introduce the two ODMs considered in our work: Mongoose and Morphia. Mongoose is the most used ODM for MongoDB when writing Javascript

applications [8]. In addition to transparent persistence, it provides support for data validation, query building, and business logic code writing. With Mongoose, database schemas are defined as Javascript JSON objects, and then applications ''can interact with MongoDB data in a structured and repeatable way'' [25]. A Mongoose schema defines the structure of stored data into a MongoDB collection. In document databases, such as MongoDB, there is a collection for each root entity. A Mongoose schema is defined for each collection and also for each non-root entity. Such schemas are the key element of Mongoose, and other mechanisms are defined based on them, like validators, indexes, and other collection options. We will show some examples of schema definitions in Section V for the database example.

Morphia is an ODM for using MongoDB from Java applications [9]. It is built upon the Java driver for MongoDB. In a Morphia database schema, each database entity is declared as a POJO (*Plain old Java object*) class whose properties are annotated with different tags. POJO classes include a field for each property of the entity, with a given visibility and type. POJO objects are easily serialized by the Morphia library to be stored into the database. Morphia provides tags to define *references*, *aggregates*, or *attributes*, among other information required in a schema. Annotations will be explained in more detail in Section VI, where we will show some POJO classes for the running database example.

D. BASIS OF MODEL-DRIVEN ENGINEERING

Model-Driven Engineering (MDE) is the Software Engineering field that provides concepts, methods, and techniques for the use of models with the purpose of automating tasks related to the creation and evolution of software. Database schemas are models, and transformational approaches have traditionally been applied to automate data engineering tasks (e.g. schema conversion and schema integration), as explained in detail in [10]. However, data engineering community has paid little attention to the application of MDE, as noted in [12].

Metamodeling and model transformations are the core elements of MDE. A metamodel describes the concepts and relationships of a certain domain. Metamodels are normally expressed as object-oriented conceptual models by means of metamodeling languages such as Ecore [19]. Concepts and their properties are represented as classes with attributes, relationships between concepts as references, and generalization of concepts as inheritance. Models are instances of metamodels. The ''conforms to'' term is also used to refer the relation between a model and their metamodel. Two kinds of model transformations are normally used: model-to-model (*m2m*) and model-to-text (*m2t*). The former transforms a source model into a target model, and the latter generates text (usually code) from a source model. There are languages tailored to write each kind of transformations. M2m transformation languages facilitate to express the mapping between the source and target metamodels (e.g. ATL [26]). M2t transformation languages are template languages tailored to specify an output text with holes to be filled by elements of the input model (e.g. Acceleo [27]). We have preferred to use the Xtend language [28], which is a general-purpose language (GPL) that offers a template mechanism aimed to write m2t transformations.

An MDE solution consists of a model transformation chain. It is usually designed with the purpose of generating some kind of code (e.g., XML, SQL, and GPL code.) Given an initial model, code is generated in one or more steps. When the code generation is simple, a model-to-text transformation is usually enough. However, one or more intermediate m2m transformations can be required when the generative process is complex. These intermediate steps require the definition of intermediate metamodels that reduce the semantic gap.

Another key element of MDE are domain-specific modeling languages, or simply domain-specific languages (DSLs). These languages are created to express models. They consist of three elements: a metamodel that represents the concepts and relationships of the language (abstract syntax), a notation defined for the metamodel (concrete syntax), and a model transformation chain that generates some kind of software artifact as GPL code (semantics). The initial model that acts as input of a model transformation chain could be created with a DSL (forward engineering approach), or injected from existing artifacts (e.g. legacy code or a database). There are available tools that automate the creation of DSLs (DSL definition workbench). The most popular workbenches are Xtext [28] and MPS [29] for textual DSLs, and Metaedit [30] and Sirius [31] for graphical DSLs.

The approach presented here is based on a two-step chain as shown in Section III. We have therefore defined an intermediate model that will be described in Section IV. In our case, the initial model is obtained from a schema inference process [6]. A DSL was required to parameterize the m2t transformation, but we decided not to use MDE technology, as explained in Section VII.

III. MOTIVATION AND OVERVIEW OF THE PROPOSED APPROACH

In this Section, we will motivate our work and present an overview of the proposed approach.

Our work is aimed at software developments that make use of an existing NoSQL database. In this context, the developers could take advantage of a tool able of automatically generating schemas (and other information) required by Object-Document mappers, saving a considerable coding effort. We identified some scenarios in which such a generation could be applicable.

- *New applications*. When creating new applications, developers could make the decision to use a mapper instead of using the database API. Applying the proposed inference and automatically generating the mapper code would make the application less prone to errors, as it is now based on the data model provided by the mapper.

- *Database evolution*. When databases evolve, the structure of the stored entities in the database may change. The database then has to be migrated to the new schema. Migrating implies (i) reading all the affected objects in the database, (ii) transforming the properties of the read objects as needed, and (iii) writing the modified objects. In general, there are two options for performing this migration: offline or lazy. Offline migration implies stopping the applications accessing the database and performing the change. The generated mapper code can help in the first part of the migration (reading the data), as the documents of the database are read *semantically* using the mapper generated classes. Conversely, *lazy* migration happens while the applications are running. In this case, the generated mapper code can be modified according to the new data model, and ''pre-load'' and ''pre-save'' methods can be added to adapt the data on the database as it is processed by applications.
- *Application evolution*. When applications evolve, the new code can be written against the generated mapper code. Also, mapper code can be used to remove database-specific queries from the application code, rewriting the data logic code to use the mapper entities, shielding the code against changes in the database.

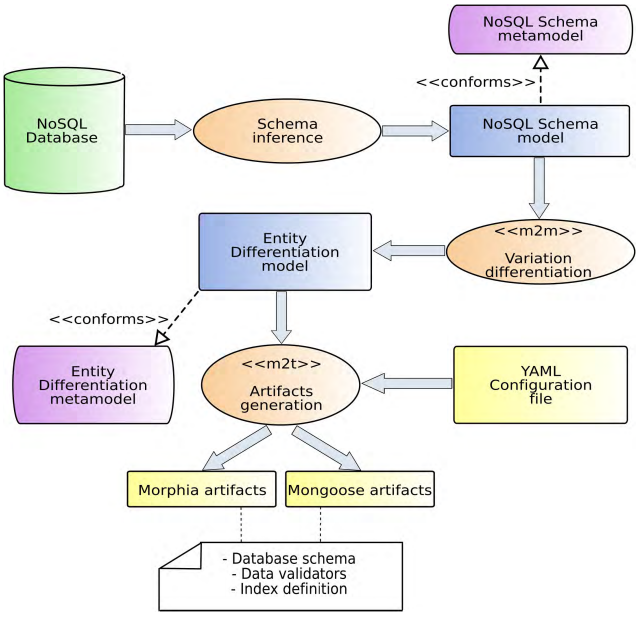


FIGURE 4. Overview of the proposed MDE solution.

We have developed a tool aimed to automate the usage of mappers for aggregation-oriented databases. This tool generates mapper code from database schemas inferred with the strategy we presented in [6]. As indicated in the previous section, the inferred schemas are represented as instances of the Ecore metamodel shown in Figure 1. We have devised a model-driven solution to implement our tool, which consists of a two-step model transformation chain, as shown in Figure 4.

In the first step, a m2m transformation converts a NoSQL schema model into an intermediate model that represents the properties of entity variations in a way that eases the code generation, e.g. common and specific properties (for each variation) are separated for each entity. The intermediate model obtained conforms to the *Entity Differentiation* metamodel which will be explained in Section IV. It should be noted that the definition of intermediate models to decompose a complex code generation process into several simpler steps is a recommended practice in MDE [32].

In the second step, a m2t transformation generates software artifacts for a target mapper. In addition to database schemas, other artifacts can be generated depending on the target mapper, e.g. data validators and indexes. Two MongoDB ODMs are currently supported, Moongoose and Morphia, but the solution can be applied to any existing ODM. A YAML file is used to parameterize m2t transformations. These configuration files provide the information required to generate code for a particular target mapper. It should also be noted that the approach is platform independent with regard to the source database as the NoSQL Schema metamodel allows us to represent schemas for any aggregation-oriented database.

Each step of our solution is described in the four following sections, and a validation is presented in Section VIII. The code that implements the solution for Mongoose and Morphia is publicly available in a GitHub repository.[1]

IV. GENERATING ENTITY DIFFERENTIATION MODELS

Entity variations must be taken into account when generating mapper artifacts. Frequently, it is necessary to differentiate between two kinds of entity properties: *common* to all the entity variations, and *specific* to a certain number of entity variations. For example, in database schema declarations, Object-Document mappers allow to annotate whether a particular property is present in all the instances of the entity or not. As NoSQL Schema models register the properties of each entity variation, to differentiate between common and specific properties may seem a trivial task. However, three properties of the inference process described in [6] made it easier to separate this differentiation from the mapper code generation process, namely:

1) The inference process is complete, that is, all the entity variations are identified and recorded, and each database object belongs exactly to one entity variation.
2) As shown in NoSQL Schema metamodel, each entity has one or more entity variations that are characterized by a set of properties (name and type).
3) For a given entity, two entity variations only differ in specific properties.

Therefore, we have introduced the *Entity Differentiation* metamodel (shown in Figure 5) in our approach. This metamodel explicitly allows to represent common and specific properties for each entity. Entity Differentiation models are derived from NoSQL Schema models through a

[1] https://github.com/catedrasaes-umu/NoSQLDataEngineering.

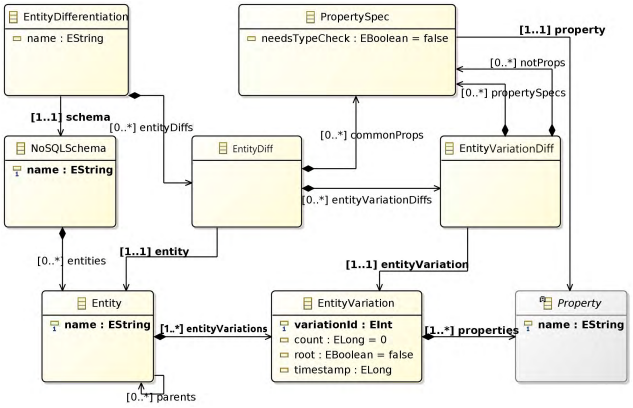


FIGURE 5. Entity differentiation metamodel.

model-to-model transformation that performs the task of identifying common and specific properties.

As seen in Figure 5, the *Entity Differentiation* metamodel is formed by four elements (i.e. metaclasses), which have references to the elements of the NoSQL schema metamodel. An entity differentiation specification (*EntityDifferentiation* metaclass) aggregates a set of entity differentiations (*EntityDiff*), and references a NoSQL schema model. An *EntityDiff* holds a set of common properties (*commonProps* relationship) that are present, with the same name and type, in all the entity variations, and a set of entity variation differentiations (*entityVariationDiffs* relationship). These specifications (*EntityVariationDiff* metaclass) hold the set of properties for each entity variation. Also, the *EntityDiff* includes another set of properties (*notProps* relationship): those present in other variations of the same entity but not in this one. Note that the union of these two sets is the same set for all the variations of a given entity.

Properties in a NoSQL schema model are linked to an Entity Differentiation model through the *PropertySpec* metaclass. Each *PropertySpec* instance references to a *Property*, and includes the *needsTypeCheck* attribute to signal when a property name is associated to more than one type. This means that given an object and entity variation having a property with that name, checking if such an object belongs to the entity variation requires to perform a type check. Therefore, the *needsTypeCheck* attribute is set for the properties that appear in any other entity variation with the same name but with different type. *EntityDiff* and *EntityVariationDiff* reference to *Entity* and *EntityVariation*, respectively, in the NoSQL schema metamododel.

As shown in Figure 4, a m2m transformation obtains an Entity Differentiation model from a NoSQL schema input model. This transformation works as follows. An *EntityDiff* element is generated for each *Entity* element in the input model. For each entity, its variations are traversed, and an *EntityVariationDiff* is generated for each. Then, the set of all properties in all variations is considered. For each property that appears in all variations, a *PropertySpec* is created referencing that *Property*, and added to the *commonProps* of

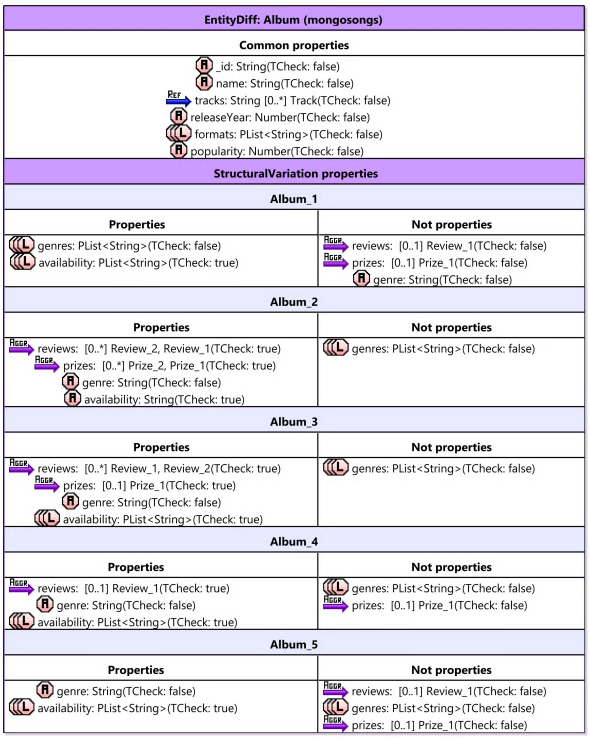


FIGURE 6. ***Entity differentiation*** **model for the** ***Album*** **entity.**

the *Entity*. Then, each *EntityVariation* is considered, and its specific properties are added to the *propertySpecs* list. In an additional step, for each *EntityVariation*, *PropertySpec*s are added to its *notProps* list. This list collects the properties present in other *EntityVariation*s, but not in the considered one. Finally, for each *PropertySpec* created, the *needsTypeCheck* attribute is set if the current property appears in some other *EntityVariation* with the same name but with different type. Figure 6 shows an excerpt of the model generated for the database example, in particular it shows the *EntityDiff* for the *Album* entity of our running example. On that diagram some common properties are shared among all variations (*commonProps* relationship) such as *name* or *releaseYear*, while some properties are specific to certain variations, such as *genres* on variation 1, or *reviews* on variations 2, 3 and 4 (*propertySpec* relationship). For each property a tooltip shows if a type checking is required, defined by the *needsTypeCheck* Boolean property in our metamodel.

V. GENERATING MONGOOSE SCHEMAS

This section describes the process of generating Mongoose schemas from *Entity Differentiation* models. As shown in Figure 4, this generation is achieved by means of a m2t transformation. The example database is used to illustrate the process. As seen in Figure 3, three root entities (one for each collection: *Album*, *Artist*, and *Track*) and four non-root entities (*Prize*, *Rating*, *Review*, and *Movie*) are included in the database example. Therefore, a Mongoose schema should be generated for each of these seven entities. Figure 7

```
// Album Schema

var mongoose = require('mongoose');
var Prize = require('./PrizeSchema');
var Review = require('./ReviewSchema');
var UnionType = require('./util/UnionType');

var Album = new mongoose.Schema({
  _id: {type: String, required: true},
  availability: UnionType(
          "U_String_[String]",
          "String", "[String]"),
  formats: {type: [String], required: true},
  genre: String,
  genres: {type: [String], default: undefined},
  name: {type: String, required: true},
  prizes: {type: [Prize],
           default: undefined},
  releaseYear: {type: Number, required: true,},
  reviews: {type: [Review],
            default: undefined},
  tracks: {type: [String], ref: "Track",
           required: true}
  }, { versionKey: false, collection: 'album'});
```

```
// Review Schema

var mongoose = require('mongoose');
var Media = require('./MediaSchema');
var UnionType = require('./util/UnionType');

var Review = new mongoose.Schema({
  journalist: {
      type: String,
      required: true},
  media: {
      type: UnionType("U_[Media]_String",
                      "[Media]",
                      "String"),
      default: undefined},
  rank: {
      type: String,
      required: true},
  stars: {
      type: UnionType("U_Number_String",
                      "Number",
                      "String")}
  }, { versionKey: false, _id : false});
```

FIGURE 7. Generated Mongoose schema for album and review entities.

shows the generated schemas for the *Album* and *Review* entities.

A Mongoose schema is declared by specifying the name and type of each entity property. In addition to the property type, several options can also be used to provide information on the property, e.g. if an index should be created for it, if it is a reference, or whether a validation must be performed. We will present here the most commonly used options. The types may be primitives or schemas previously defined. `Tuples` of these types may also be defined by enclosing the type name between brackets. The `String`, `Number`, `Date`, and `Boolean` types are included. For example, in the `Album` schema, the `genre` property is of type `String`, `releaseYear` is of type `Number`, and `formats` is a tuple of `Strings`. An aggregation may be expressed as a nested schema or either declaring a external schema (i.e. a type) for the nested entity. The latter is more convenient to improve the legibility of the schema: an aggregation is declared by merely indicating the name given to the external schema. The `ref` option is used to express references. When a property is a reference, it is required to specify the primitive type used to express references and the name of the schema of the referenced objects. In the `Album` schema, the `prizes` property is a tuple of `Prize` type, that is, an `Album` document aggregates zero or more `Prize` objects; and the `tracks` property is a tuple of strings that are references to `Track` objects. Note that a *model* is a compiled version of a schema, which is used to create, read, save, and delete documents of the corresponding schema.

When automatically generating schemas, we had to consider the existence of entity variations. For this, we have used the `required` Boolean option that acts as a validator. This option states that a value for a particular field must always be given to save documents of that kind of entity. Therefore, the declaration of a common property should set the `required` validator to `true`. When reading documents from the database, the schema has to be able to describe all of them. The schemas, then, include all the properties in all the variations, with their corresponding types.

The m2t transformation works as follows. *EntityDiff* elements in a Entity Differentiation model are traversed, and a schema is generated for each *EntityDiff*, that in turns corresponds to each entity, both root and aggregated. For each of them, all its properties are added to the generated schema, adding the `required` option only for common properties. Each aggregate and reference property is declared with the corresponding schema that it refers to, and each reference property is also declared with the `ref` option. Some additional options may also be added to generated schemas. For instance, the collection name for root entities (`collection: Album` in case of the *Album* entity), a `versionKey: false` option if document versioning should be disabled and the `_id` option is set to false for non-root entities in order to avoid the generation of the `_id` property for each object stored in the database.

As indicated in the previous Section, an entity can have properties with the same name but different type. This should be specified in Mongoose schemas by using the `Mixed` predefined type, as there is not a notion of a "union" type. When a property is declared of `Mixed` type, a value of any type can be assigned to it (equivalent to declaring the property of type `Object`.) Therefore, the property cannot be validated, limiting the schema effectiveness. To deal with properties with the same name in an entity, we have created the `Union` type: the value of a property can belong to one among a list of specified types. This type has been defined as an auxiliary Mongoose schema type contained in a separate file (`UnionType.js`), which is imported in each schema being generated. This method combines the types of the union in a single new type, which is added to the Mongoose type hierarchy. It also validates that the value assigned to a property is of one of the types specified in the union type.

```
@Entity(value = "album", noClassnameStored = true)
public class Album
{
  @Id
  @NotNull(message = "_id can't be null")
  private String _id;
  public String get_id() {return this._id;}
  public void set_id(String _id) {this._id = _id;}

  // @Union_String_List<String>
  @SuppressWarnings("unused")
  private Object availability;
  private String __availability1;
  private List<String> __availability2;
  // [...Get/Set "availability" methods...]
  // [...@PreLoad and @PreSave
  //   methods for this Union type...]

  @Property
  @NotNull(message = "formats can't be null")
  private List<String> formats;
  public List<String> getFormats()
    {return this.formats;}
  public void setFormats(List<String> formats)
    {this.formats = formats;}

  @Property
  private String genre;
  public String getGenre() {return this.genre;}
  public void setGenre(String genre)
    {this.genre = genre;}

  @Property
  private List<String> genres;
  public List<String> getGenres()
    {return this.genres;}
  public void setGenres(List<String> genres)
    {this.genres = genres;}

  @Property
  @NotNull(message = "name can't be null")
  private String name;
  public String getName() {return this.name;}
  public void setName(String name) {this.name = name;}

  @Embedded
  private List<Prize> prizes;
  public List<Prize> getPrizes() {return this.prizes;}
  public void setPrizes(List<Prize> prizes)
    {this.prizes = prizes;}

  @Property
  @NotNull(message = "releaseYear can't be null")
  private Integer releaseYear;
  public Integer getReleaseYear()
    {return this.releaseYear;}
  public void setReleaseYear(Integer releaseYear)
    {this.releaseYear = releaseYear;}

  @Embedded
  private List<Review> reviews;
  public List<Review> getReviews()
    {return this.reviews;}
  public void setReviews(List<Review> reviews)
    {this.reviews = reviews;}

  @Reference(idOnly = true, lazy = true)
  @NotNull(message = "tracks can't be null")
  private List<Track> tracks;
  public List<Track> getTracks() {return this.tracks;}
  public void setTracks(List<Track> tracks)
    {this.tracks = tracks;}
}
```

FIGURE 8. Generated Morphia schema for the *Album* entity.

Using the `Union` type requires to specify the name and the list of types, as illustrated in the properties `availability` of the `Album` schema and `media` of the `Review` schema in Figure 7. The former is a union named `U_String_[String]` of `String` and tuples of strings, and the latter is a union named `U_[Media]_String` of `Media` and `String`.

VI. GENERATING MORPHIA SCHEMAS

This section describes code generation process for Morphia. As commented in Section II, Morphia uses annotations to declare entity schemas, instead of special fields (i.e. options) as Mongoose. With annotations, the developer is able to define the semantics of each field in a class, as well as validations and certain processing to be applied when serializing/deserializing a field to/from the database.

Morphia annotations may be classified into two groups depending on whether they are used to define the database schema (native annotations) or validation mechanisms (Javax annotations). The most commonly used native annotations are the following:

- `@Entity` applied on a class to indicate that it is a root entity.
- `@Embedded` can be applied on classes and fields to indicate that a class represents a non-root entity or that a field stores an embedded object.
- `@Id`, a field is an identifier of the class, and so it will hold a unique value for each instance of the class.
- `@Property`, a field holds a primitive type value.
- `@Reference` a field holds one or more references to objects of another collection.

Native annotations to declare indexes and Javax annotations will be explained in Section VII.

Each annotation includes parameters to give additional information. For example, `@Reference(lazy = true)` declares that references will be lazily solved, and `@Entity(value = 'album')` explicitly indicates the name of the MongoDB collection in which instances of this class will be stored.

Figures 8 and 9 show how the annotations are used to declare Morphia POJO classes for the `Album` root entity and the `Review` embedded entity, respectively. This code has been generated with the m2t transformation we implemented for Morphia. The transformation works as follows.

EntityDiff elements in a Entity Differentiation model are traversed, and a POJO class is generated for each of them. These classes will be marked with the `@Entity` or `@Embedded` tag depending on whether the *EntityDiff* element is connected to a root or embedded entity, respectively. Then, a private field and its corresponding *getter* and *setter* methods are generated for each existing *PropertySpec* element in the model. As seen in Figure 8, these fields will be annotated with `@Property` (such as *genre*), `@Reference` (as in `tracks`), or `@Embedded` (`reviews`), depending on whether the *Property* referenced by the *PropertySpec* element is an *Attribute*, *Reference*, or *Aggregate*, respectively.

```
@Embedded
public class Review
{
  @Property
  @NotNull(message = "journalist can't be null")
  private String journalist;
  public String getJournalist()
    {return this.journalist;}
  public void setJournalist(String journalist)
    {this.journalist = journalist;}

  // @Union_List<Media>_String
  @SuppressWarnings("unused")
  private Object media;
  private List<Media> __media1;
  private String __media2;
  // [...Get/Set "media" methods...]
  // [...@PreLoad and @PreSave
  // methods for this Union type...]

  @Property
  @NotNull(message = "rank can't be null")
  private String rank;
  public String getRank() {return this.rank;}
  public void setRank(String rank)
    {this.rank = rank;}

  // @Union_Integer_String
  @SuppressWarnings("unused")
  private Object stars;
  private Integer __stars1;
  private String __stars2;
  // [...Get/Set "stars" methods...]
  // [...@PreLoad and @PreSave
  // methods for this Union type...]
}
```

FIGURE 9. Generated Morphia schema for the embedded *Review* entity.

Other annotations can also be generated for each field: `@NotNull` if the property is not optional (as in the `formats` property), or `@Id` if the property is the identifier field (by default, the property should be named "`_id`".) As explained in Section VII, indexes and validation tags are also generated on this step.

Types *Property*, *Reference*, and *Aggregate* are of the same type as the original type in *PropertySpec*, and thus they are created by a direct mapping when processing the *EntityDifferentiation* model.

In Morphia, properties having the same name but different type in an entity could be represented by using the `Object` Java type. This solution is similar to using the `Mixed` type in Mongoose. Validations, then, could not be applied. In addition, we encountered problems when Morphia tried to serialize/deserialize `Objects` before storing or after reading values from the database. To overcome these problems and allow validating these properties, we devised the following mechanism that simulates a union type for POJO Java classes.

- Given an entity that has n properties with the same name *prop* and m different types $t_1, t_2, \ldots, t_m$, the following private fields are generated in the corresponding POJO class: a field named *prop*, which may be annotated if necessary to provide some validation, and a field named $prop_i$, $i = 1, m$ for each different type.
- A *setter* method will be generated, which will receive an `Object` as argument value, and will store it on the suitable $prop_i$ field depending on the argument type. This method will ensure that, at any given moment, only one of the m $prop_i$ fields has a value.
- A *getter* method will be generated to return the value of the $prop_i$ field that has a valid value.
- A method annotated with the `@Preload` tag is generated, which is in charge of deserializing the *prop* property value stored into the database and assigning it to the corresponding property $prop_i$ in the POJO class. This method is named `preLoad` followed by the list of types of the union separated by the underscore symbol. It will be called when reading the property.
- Finally, a method annotated with a `@PreSave` tag is generated, which is in charge of serializing the *prop* property before storing it into the database. These methods are named like `preLoad` methods.

This approach works correctly except when the union type includes the `String` type, and references expressed as strings, e.g. the *media* property in the *Review* entity. In this case, the reference cannot be resolved automatically when loading data from a database, as it is not possible to distinguish between the `String` type and the `Reference` type when reading a database object. The user is in charge of obtaining the referenced object using a suitable query. An example of this `Union` mechanism may be seen on Figure 8, on the `availability` field, which may be a `String` or a `List<String>`.

VII. GENERATING ADDITIONAL ARTIFACTS

Additionally to the schema definition, MongoDB ODMs provide more capabilities to facilitate the development of database applications, such as index specifications, validators, and reference management. In this section, we will explain how our m2t transformations generate code related to these ODM facilities.

To generate ODM code for artifacts that are not entity schemas, it is necessary to provide additional information to that inferred in our schema extraction process, e.g. the types of indexes to be added (if any) and the type of validation to be applied. We have therefore defined a textual notation that allows developers to create configuration files with such information. This configuration file is an optional input to our m2t transformations for Mongoose and Morphia, in addition to an *EntityDifferentiation* model. In a previous version of our work, this notation was defined for Mongoose as a simple domain-specific language (DSL) [14]. This DSL was built by using Xtext, a well-know textual DSL definition workbench [33]. Generating artifacts for more ODMs entails to change this DSL, as configuration data is different for each of them. Tackling the changes for Morphia, we considered that, in this case, a YAML notation could be more appropriate to support the evolution to new ODMs.

```yaml
mapper: Mongoose
entities:
 - name: Album
   indexes:
    - attr: [name, releaseYear]
      type: [asc, desc]
      unique: true
      sparse: true
      background: false
   validators:
    - attr: releaseYear
      min: 1900
      max: 2019
    - attr: name
      maxLength: 300
      message: This album name is too long
 - name: Review
   validators:
    - attr: rank
      enumValues:
        [Excelent, Very good, Good, Poor, Terrible]
      message: A review rank may be "Excelent",
        "Very good", "Good", "Poor" or "Terrible"
    - attr: stars
      min: 0
      max: 5
```

FIGURE 10. Mongoose configuration file example for the running database.

YAML is a widely-used data serialization language [13] which has some interesting features: (i) it is easy to read, as it follows closely the patterns used to write formatted ASCII text, (ii) it is expressive, in the sense that it is equivalent to JSON, and includes data structures similar to those in popular programming languages such as Python, Ruby, or Javascript, (iii) it is programming-language agnostic, and parsing libraries are available for most programming languages. In our case, to support the customization of the code generation to more than one mapper, we decided to use YAML instead of a full-blown DSL mainly because of two reasons:

- The set of options to customize the m2m transformation is mostly a collection of key-value pairs, supported natively by YAML, so there is no need of defining a specialized grammar and parser.
- The set of options may be different for each ODM. Also, the mapper used depends on the *value* of the `mapper` property of the DSL text. This is difficult to achieve, as grammars are fixed beforehand in order to generate parsers. We may have ended up with either a DSL with all the possible configuration keywords for all supported mappers (and that we would have had to change whenever we wanted to support a new mapper), or different DSLs, one for each supported mapper. In the case of YAML, the standard interpreter can walk and recognize all the configuration options (option-value pairs).

Using YAML as an additional input to the m2t transformation is feasible because the transformation itself is written in a general-purpose language (Xtend [28]). Had we used m2t transformation languages (e.g. Acceleo [27]), the choice of YAML would have not been possible, because transformations can only take models as input.

Figure 10 shows an example of YAML configuration file for Mongoose. The file is required to have the following structure: First of all, a `mapper` property identifies the ODM being configured. Next, an `entities` block encloses the configuration for the entities the user wants to define options for. For each of these entities, an entity block with the options is defined, which is headed by the entity name. In turn, an entity block contains one or more option blocks that consist of a set of parameter/value pairs. Next, we indicate some of the parameters for the considered options.

`Indexes` are configured by providing the following information: the set of attributes to be indexed, the type of index (ascending or descending), and some Boolean optional parameters to indicate whether the index has that property: `unique`, `sparse`, `background`, or `weight`, among others. No index is created by default.

`Validators` require to indicate the name of the property to be validated, and a set of parameters that depend on the validation type, for example, `min` and `max` to establish a numeric values range, `minLength` and `maxLength` to establish `String` length, `enumValues` to declare a enumeration of possible values, `match` to define a pattern to be matched by property values, etc. Also, a `message` may be provided to be shown if the validation fails. No validation is performed on a property if the corresponding option is not defined.

In Mongoose, the validation is defined at the schema level, and some frequently used validators are already built-in. The *required* and *unique* validators can be applied to any property. As explained in section V, we have used *required* to specify what properties are common to all the entity variations. The *unique* validator is used to express that all the documents of a collection must have a different value for a given field.

The example in Figure 10 shows the configuration of two entities for the Mongoose mapper: `Album` and `Review`. Three options are also configured for *Album*: a unique sparse index is defined for the *name* and *releaseYear* properties, which is created in ascending and descending order, respectively, and two validators are defined for these two properties to check that those fields hold a correct value. Finally, enumeration and range validators are defined for the property *rank* and *starts* of the *Review* entity.

The Mongoose code generated for the options defined in Figure 10 is shown in Figure 11. This figure shows how built-in validators are introduced on the definition of certain properties, and statements to create indexes are separated from the schema definition. Note that in Mongoose there are lazy references, since all the references must explicitly be resolved by developers.

```
// Album Schema
var Album = new mongoose.Schema({
  _id: {type: String, required: true},
  availability: UnionType("U_String_[String]",
    "String", "[String]"),
  formats: {type: [String], required: true},
  genre: String,
  genres: {type: [String], default: undefined},
  name: {type: String,
    maxlength: [300, "This album's name is too long"],
    required: true},
  prizes: {type: [Prize], default: undefined},
  releaseYear: {type: Number, required: true,
    min: 1900, max: 2019},
  reviews: {type: [Review], default: undefined},
  tracks: {type: [String], ref: "Track",
    required: true}
}, { versionKey: false, collection: 'album'});

Album.index({name: 1, releaseYear: -1},
  {unique: true, sparse: true, background: false});

// Review Schema
var Review = new mongoose.Schema({
  journalist: {type: String, required: true},
  media: {type:
    UnionType("U_[Media]_String",
    "[Media]", "String"),
    default: undefined},
  rank: {type: String, required: true,
    enum: ['Excelent', 'Very good', 'Good',
      'Poor', 'Terrible']},
  stars: {type: UnionType("U_Number_String",
    "Number", "String"),
    max: 5,
    min: 0}
}, { versionKey: false, _id : false});
```

FIGURE 11. Excerpt of code generated for Mongoose using the configuration file example.

```
// Album Schema
@Entity(value = "album", noClassnameStored = true)
@Indexes({
 @Index(fields = {@Field(value = "name",
 type = IndexType.ASC), @Field(value = "releaseYear",
 type = IndexType.DESC)},
 options = @IndexOptions(unique = true, sparse = true,
 background = false))
})
public class Album
{
  @Id
  @NotNull(message = "_id can't be null")
  private String _id;
  // [...Get/Set "_id" methods...]

  // @Union_String_List<String>
  @SuppressWarnings("unused")
  private Object availability;
  private String __availability1;
  private List<String> __availability2;
  // [...Get/Set "availability" methods...]
  // [...@PreLoad and @PreSave
  // methods for this Union type...]

  @Property
  @NotNull(message = "formats can't be null")
  private List<String> formats;
  // [...Get/Set "formats" methods...]

  @Property
  private String genre;
  // [...Get/Set "genre" methods...]

  @Property
  private List<String> genres;
  // [...Get/Set "genres" methods...]

  @Property
  @NotNull(message = "name can't be null")
  @Size(max = 300,
    message = "This album's name is too long")
  private String name;
  // [...Get/Set "name" methods...]

  @Embedded
  private List<Prize> prizes;
  // [...Get/Set "prizes" methods...]

  @Property
  @NotNull(message = "releaseYear can't be null")
  @Min(value = 1900)
  @Max(value = 2019)
  private Integer releaseYear;
  // [...Get/Set "releaseYear" methods...]

  @Embedded
  private List<Review> reviews;
  // [...Get/Set "reviews" methods...]

  @Reference(idOnly = true, lazy = true)
  @NotNull(message = "tracks can't be null")
  private List<Track> tracks;
  // [...Get/Set "tracks" methods...]
}
```

FIGURE 12. Excerpt of code generated for Morphia for the *Album* entity using the configuration file example.

```
// Review Schema
@Embedded
public class Review
{
  @Property
  @NotNull(message = "journalist can't be null")
  private String journalist;
  // [...Get/Set "journalist" methods...]

  // @Union_List<Media>_String
  @SuppressWarnings("unused")
  private Object media;
  private List<Media> __media1;
  private String __media2;
  // [...Get/Set "media" methods...]
  // [...@PreLoad and @PreSave
  // methods for this Union type...]

  @Property
  @NotNull(message = "rank can't be null")
  @Pattern(regexp = "Excelent|Very good|Good|Poor|
   Terrible", flags = Pattern.Flag.CASE_INSENSITIVE,
   message = "A review rank may be \"Excelent\",
   \"Very good\", \"Good\", \"Poor\" or \"Terrible\"")
  private String rank;
  // [...Get/Set "rank" methods...]

  // @Union_Integer_String
  @Max(value = 5)
  @Min(value = 0)
  private Object stars;
  private Integer __stars1;
  private String __stars2;
  // [...Get/Set "stars" methods...]
  // [...@PreLoad and @PreSave
  // methods for this Union type...]
}
```

FIGURE 13. Excerpt of code generated for Morphia for the embedded *Review* entity using the configuration file example.

In the case of Morphia, Figures 12 and 13 show the generated code for a configuration file similar to that in Figure 10. The defined options will be translated into annotations provided by the `mongodb` and `javax` libraries. Indexes are translated to `@Indexes` annotations appearing before the class header. This annotation has several parameters: the

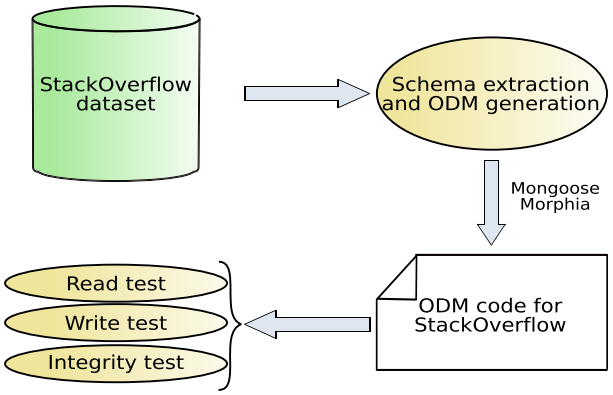


FIGURE 14. Validation process workflow.

name and type of the properties being indexed, and optional parameters providing information on the index type. In Figure 12 the index defined for the `name` and `releaseYear` properties in the `Album` entity can be observed. For each validation rule defined, some annotations are added to the header of the property definition. For example, in Figure 12, `@Size` on the `name` field or `@Min` and `@Max` on the `releaseYear` field. Finally, in Figure 13, a `@Pattern` tag may be seen to assure that the `rank` field holds allowed values.

VIII. VALIDATION PROCESS

This Section presents the validation process designed to test the approach described in this paper to generate ODM code. In particular, we have used the code generated from a NoSQL schema model to check the database consistency by executing several test suites. Figure 14 shows the workflow of our validation process.

We used a fragment of the Stackoverflow dataset[2] to populate a MongoDB database on which the validation will be performed. This fragment is composed of the following collections: `Users`, `Posts`, `Postlinks`, `Tags`, `Votes`, `Comments`, and `Badges`. For each collection, around 15 million objects were inserted. Figure 15 shows the NoSQL schema inferred by applying our schema extraction process [6].

Once the schema model for the Stackoverflow dataset is extracted, we generated the *EntityDifferentiation* model. From that model the schemas for Mongoose and Morphia were generated. Indexes and built-in validators defined in Section VII have not been considered for testing purposes. Figures 16 and 17 show the Mongoose and Morphia schema code, respectively, generated for the *Comment* entity. Finally, we have applied several test cases.

[2]https://archive.org/details/stackexchange.

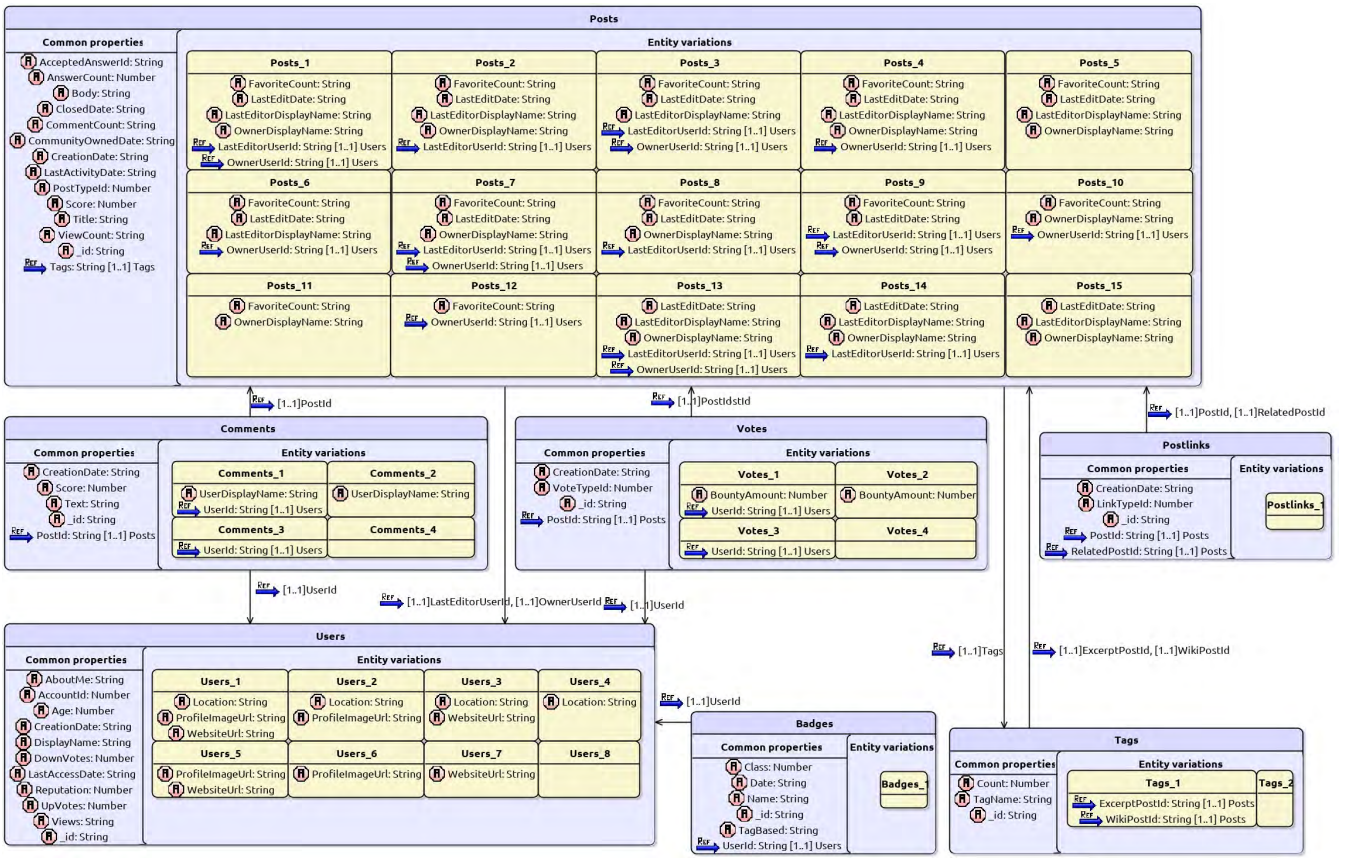


FIGURE 15. An excerpt of the StackOverflow schema.

```
//Comment schema
'use strict'

var mongoose = require('mongoose');

var Comments = new mongoose.Schema({
  CreationDate: {type: String, required: true},
  PostId: {type: Number, ref: "Posts", required: true},
  Score: {type: Number, required: true},
  Text: {type: String, required: true},
  UserDisplayName: String,
  UserId: {type: Number, ref: "Users"},
  _id: {type: String, required: true}
}, { versionKey: false, collection: 'comments'});

module.exports = Comments;
```

FIGURE 16. **StackOverflow** `Comment` **schema code generated for Mongoose.**

Three tests have been designed according to three requirements to be satisfied by the generated schemas: (i) they should allow to read each document stored into the database and create the corresponding object in memory, (ii) they should allow to write each object previously read into a database, and (iii) integrity should be preserved when the database is updated, that is, properties can only be updated with values of the types defined in the schema.

To validate the first requirement, we have performed a test that reads all the documents of the database by checking that there is no compatibility errors. The second requirement has been validated by reading all the database documents and performing a copy on another database; then the previous test is applied on the new database. The third test consists on manipulating objects mapped from the database and changing certain attributes of this objects to values of different types to those defined on the schema, such as setting `String` values on a `UserId` field, which only admits `Numbers`.

All these tests were successfully validated. The three tests are available on the GitHub public repository.[3] They are part of an Eclipse project, and code generated for several datasets is also included.

[3]https://github.com/catedrasaes-umu/NoSQLDataEngineering/blob/master/projects/es.um.nosql.s13e.entitydifferentiation.examples/test/es/um/nosql/s13e/test/morphia/StackoverflowTest.java.

IX. RELATED WORK

As indicated in the report presented in [2], the successful adoption of NoSQL database systems demands the existence of database tools with similar functionality to those offered for relational databases. Automatic code generation from schemas is one of the functionalities considered in that report. So far, the attention paid to this topic has been very limited in tools and published works.

As far as we know, the work presented here is the first published approach for automating the use of ODMs for existing databases. Recently, the Hackolade tool [34] has included functionality to generate code for the Mongoose mapper. When analyzing this tool we have found some significant differences with our approach:

- Our schema inference process extracts the entity variations which are managed in the code generation process. Instead, in Hackolade, Mongoose code is generated from a schema without variations, created by the database designer as part of a forward engineering process.
- Our approach generates a technology-independent intermediate metamodel, and we have generated code for Mongoose and Morphia mappers. In Hackolade, code is only generated for the Mongoose mapper.
- We have defined a union type to address fields with the same name but different type. In Hackolade, mixed types are not considered as union types.
- In addition to schemas and validators, other artifacts (indexes and reference management) are generated in our approach.

A few MDE-based approaches for NoSQL databases have been proposed [35]–[37], which have in common that the generation process starts from a UML class diagram. An MDE approach to generate code aimed to manipulate graph databases is presented in [35]. A two-steps model transformation chain is defined to generate code from a conceptual schema expressed as a UML class diagram and OCL statements. In the first step, the conceptual schema is transformed into a model that conforms to a metamodel that represents graph databases in a generic way. Then, a m2t transformation generates code for two technologies that

```
//Comment schema
@Entity(value = "comments", noClassnameStored = true)
public class Comments
{
  @Id
  @NotNull(message = "_id can't be null")
  private String _id;
  // [...Get/Set "_id" methods...]

  @Property
  @NotNull(message = "CreationDate can't be null")
  private String CreationDate;
  // [...Get/Set "CreationDate" methods...]

  @Reference(idOnly = true, lazy = true)
  @NotNull(message = "PostId can't be null")
  private Posts PostId;
  // [...Get/Set "PostId" methods...]

  @Property
  @NotNull(message = "Score can't be null")
  private Integer Score;
  // [...Get/Set "Score" methods...]

  @Property
  @NotNull(message = "Text can't be null")
  private String Text;
  // [...Get/Set "Text" methods...]

  @Property
  private String UserDisplayName;
  // [...Get/Set "UserDisplayName" methods...]

  @Reference(idOnly = true, lazy = true)
  private Users UserId;
  // [...Get/Set "UserId" methods...]
}
```

FIGURE 17. **An excerpt of StackOverflow** `Comment` **schema code generated for Morphia.**

provide uniform access to graph databases such as Blueprints and Gremlin. Unlike our work, this approach is based on a forward engineering strategy, where a designer creates a conceptual schema, and mapper code is not generated. In [36] and [37] a proposal to generate NoSQL schema models from UML class diagrams is presented, but no code is generated, only mappings are addressed.

X. CONCLUSIONS AND FUTURE WORK

The schemaless feature provides agility and flexibility to developers of NoSQL applications. However, modeling tools are also needed for NoSQL databases, as noted in [2], [38], [39]. Three capabilities are remarked for NoSQL data modeling tools in [2]: (i) generating code, (ii) model visualization, and (iii) metadata management. In our group, we first defined a NoSQL schema inference strategy [6], and then addressed the schema visualization [21]. Here, we have explored the code generation for Object-Document mapping tools. In our knowledge, it is the first published work for automating the use of NoSQL mappers.

We have applied MDE techniques to define a technology-independent solution, consisting of a two-step model transformation chain. As a proof of concept, we have generated code for two ODM tools: Mongoose and Morphia. Mappers for other NoSQL document systems could also be considered. In addition to schemas, we generated other artifacts such as indexes and validators. The approach shows the usefulness of a high level NoSQL schema metamodel, and how intermediate models help to simplify a m2t transformation. Generating code for other mappers would only require to implement a new m2t transformation, but the m2m transformation is completely reusable. In our approach, we have addressed the existence of entity variations and relationships (aggregation and reference) in the inferred database schemas. To represent properties with the same name and different type, we have created a *union type* in Mongoose and defined a strategy code pattern that allows to implement unions in Morphia.

A set of parameters must be provided to the m2t transformation in order to generate artifacts. In a previous version of our work, we defined a textual DSL to specify these parameters. This DSL was created with a metamodel-based workbench that automatically generates the injector that transforms text into a model that conforms to the DSL metamodel. However, in this work we have created this DSL without using MDE technology. Bearing in mind the need to extend the DSL for new mappers, we have created the notation using the YAML language. We have considered that notations defined with YAML can be extended more easily (in this case) than metamodel-based DSLs.

In our work, we created a utility that automates the use of ODM tools, and also identified usage scenarios of our solution. In all these scenarios, a document database exists, and mappers are useful to build new applications, or to migrate the existing database or code.

With regard to future work, we plan to consider some paths:

- Study adding more mappers for MongoDB and other database systems.
- Instead of using just the inferred schema, the DSL could include directives to adapt the objects of the database to the application. For instance, if the application is not interested in an attribute of an entity, ''pre-load'' methods that remove that attribute could be generated automatically. The application would be then lazily migrating the database when saving objects without that attribute.
- Similarly to the previous point, scripts or map-reduce transformations could be generated to migrate the database off-line.
- Finally, a ''strict'' mode could be implemented that prevents the application on generating new variations. Even with the *union* mechanism, the application could create combinations of attributes not previously present in the database. The strict mode would prevent it by automatically generating ''pre-save'' methods that prevent this behavior.

REFERENCES

[1] (2015). *NoSQL-Databases.Org Webpage*. Accessed: Oct. 2018. [Online]. Available: https://nosql-database.org

[2] V. Bacvanski and C. Roe, ''Insights into NoSQL modeling: A dataversity report,'' DataVersity Education, LLC, Tech. Rep., 2015.

[3] (2015). *NoSQL Market Webpage*. Accessed: Oct. 2018. [Online]. Available: https://www.alliedmarketresearch.com/NoSQL-market

[4] (2016). *NoSQL Ranking*. Accessed: Oct. 2018. [Online]. Available: http://db-engines.com/en/ranking

[5] M. Fowler. (Jan. 2013). *Schemaless Data Structures*. [Online]. Available: http://martinfowler.com/articles/schemaless/

[6] D. S. Ruiz, S. F. Morales, and J. G. Molina, ''Inferring versioned schemas from NoSQL databases and its applications,'' in *Proc. 34th Int. Conf. Conceptual Modeling (ER)*, Stockholm, Sweden, Oct. 2015, pp. 467–480.

[7] U. Störl, T. Hauf, M. Klettke, and S. Scherzinger, ''Schemaless nosql data stores—Object-NoSQL mappers to the rescue?'' in *Proc. Datenbanksyst. Bus., Technologie Web (BTW)*, Hamburg, Germany, 2015, pp. 579–599. [Online]. Available: https://dl.gi.de/20.500.12116/2432

[8] (2017). *Mongoose Web Page*. Accessed: Oct. 2018. [Online]. Available: http://mongoosejs.com

[9] (2016). *Morphia Webpage*. Accessed: Oct. 2018. [Online]. Available: https://github.com/mongodb/morphia

[10] J.-L. Hainaut, ''The transformational approach to database engineering,'' in *Generative and Transformational Techniques in Software Engineering* (Lecture Notes in Computer Science), vol. 4143. Berlin, Germany: Springer, 2006, pp. 95–143. doi: 10.1007/11877028_4.

[11] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. San Rafael, CA, USA: Morgan & Claypool, 2012.

[12] F. J. B. Ruiz, J. G. Molina, and O. D. García, ''On the application of model-driven engineering in data reengineering,'' *Inf. Syst.*, vol. 72, pp. 136–160, Dec. 2017.

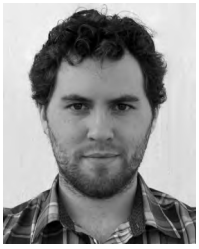
[13] (2019). *YAML: YAML Ain't Markup Language*. Accessed: Jan. 2019. [Online]. Available: https://yaml.org/

[14] D. Sevilla, S. F. Morales, and J. G. Molina, ''An MDE approach to generate schemas for object-document mappers,'' in *Proc. 5th Int. Conf. Model-Driven Eng. Softw. Develop., (MODELSWARD)*, Porto, Portugal, Feb. 2017, pp. 220–228.

[15] P. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Reading, MA, USA: Addison-Wesley, 2012.

[16] P. Buneman, ''Semistructured data,'' in *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. Princ. Database Syst.*, Phoenix, AZ, USA, 1997, pp. 117–121.

[17] L. Wang *et al.*, "Schema management for document stores," *VLDB Endowment*, vol. 8, no. 9, pp. 922–933, 2015.
[18] M. Klettke, U. Störl, and S. Scherzinger, "Schema extraction and structural outlier detection for JSON-based NoSQL data stores," in *Proc. Conf. Database Syst. Bus., Technol., Web (BTW)*, 2015, pp. 425–444.
[19] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*. Reading, MA, USA: Addison-Wesley, 2009.
[20] (2016). *JavaScript JSON Webpage*. Accessed: Oct. 2018. [Online]. Available: http://www.w3schools.com/js/js_json.asp
[21] A. H. Chillón, S. F. Morales, D. Sevilla, and J. G. Molina, "Exploring the visualization of schemas for aggregate-oriented NoSQL databases," in *Proc. ER Forum 36th Int. Conf. Conceptual Modelling (ER)*, Valencia, Spain, Nov. 2017, pp. 72–85.
[22] (2001). *Hibernate Webpage*. Accessed: Jan. 2019. [Online]. Available: http://hibernate.org/
[23] (2018). *Doctrine Project Webpage*. Accessed: Oct. 2018. [Online]. Available: http://www.doctrine-project.org/
[24] (2012). *Mandango Webpage*. Accessed: Sep. 2018. [Online]. Available: https://mandango.readthedocs.io/en/latest/
[25] S. Holmes, *Mongoose for Application Development*. Birmingham, U.K.: PACKT Publishing, 2013.
[26] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, nos. 1–2, pp. 31–39, 2008.
[27] (2018). *Acceleo Webpage*. Accessed: Oct. 2018. [Online]. Available: http://www.eclipse.org/acceleo/
[28] L. Bettini, *Implementing Domain-Specific Languages With Xtext and Xtend*, 2nd ed. Birmingham, U.K.: Packt Publishing, 2016.
[29] F. Campagne, *The MPS Language Workbench*, vol. 1, 1st ed. Scotts Valley, CA, USA: CreateSpace Independent Publishing Platform, 2014.
[30] (2018). *Metaedit Webpage*. Accessed: Oct. 2018. [Online]. Available: http://www.metacase.com/mep/
[31] (2017). *Sirius official Website*. Accessed: Oct. 2018. [Online]. Available: https://eclipse.org/sirius/
[32] M. Völter, "MD* best practices," *J. Object Technol.*, vol. 8, no. 6, pp. 79–102, 2009.
[33] M. Voelter *et al.* (2013). *DSL Engineering—Designing, Implementing and Using Domain-Specific Languages*. [Online]. Available: http://www.dslbook.org
[34] *Hackolade Webpage*. Accessed: Apr. 2019. [Online]. Available: https://hackolade.com/
[35] G. Daniel, G. Sunyé, and J. Cabot, "UMLtoGraphDB: Mapping conceptual schemas to graph databases," in *Proc. 35th Int. Conf. Conceptual Model. (ER)*, Gifu, Japan, Nov. 2016, pp. 430–444.
[36] Y. Li, P. Gu, and C. Zhang, "Transforming UML class diagrams into HBase based on meta-model," in *Proc. IEEE Int. Conf. Inf. Sci., Electron. Elect. Eng.*, vol. 2, Apr. 2014, pp. 720–724.
[37] F. Abdelhedi, A. A. Brahim, F. Atigui, and G. Zurfluh, "MDA-based approach for NoSQL databases modelling," in *Proc. Int. Conf. Big Data Anal. Knowl. Discovery*. Cham, Switzerland: Springer, 2017, pp. 88–102.
[38] P. Atzeni, "Models for NoSQL databases: A contradiction?" in *Proc. Workshops Adv. Conceptual Modeling (ER)*, Stockholm, Sweden, Oct. 2015, p. 133.
[39] A. Wang. (Feb. 2016). *Unified Data Modeling for Relational and NoSQL Databases*. [Online]. Available: https://www.infoq.com/articles/unified-data-modeling-for-relational-and-nosql-databases



ALBERTO HERNÁNDEZ CHILLÓN was born in Murcia, Spain, in 1988. He received the B.S. and M.S. degrees in computer science (specialized in software technologies) from the University of Murcia, Spain, in 2016, where he is currently pursuing the Ph.D. degree in computer science, studying database evolution and researching how to define abstract NoSQL schemas, database operations, and data definitions.

He was a member of the Cátedra SAES Team, from 2014 to 2019, where he worked on topics such as model-driven engineering, automatic code generation, and NoSQL technologies.

DIEGO SEVILLA RUIZ received the M.Sc. and Ph.D. degrees in computer science from the University of Murcia, Spain, where he is currently an Associate Professor with the Department of Computer Engineering (DITEC). His research interests include distributed systems, functional programming, model-driven engineering, big data architectures, and NoSQL data engineering.

JESÚS GARCÍA MOLINA received the Ph.D. degree in chemistry from the University of Murcia, Spain, in 1987, where he has been a Full Professor with the Faculty of Informatics, since 1991. He leads the Modelum Group, a R&D group focused on model-driven engineering with a close partnership with industry. His research interests include model-driven development, domain-specific languages, and model-driven modernization. He is the author of two textbooks, and its main research contributions are listed in DBLP.

SEVERINO FELICIANO MORALES was born in Pochahuizco, Guerrero, Mexico, in 1972. He received the B.S. degree in computer science from the National Tecnológico of Mexico (Instituto Tecnológico of Chilpancingo), Chilpancingo, Guerrero, Mexico, in 1994, the M.S. degree in computer science from the Faculty of Computer Science, Autonomous University of Guerrero, Chilpancingo, in 2008, and the Ph.D. degree from the Faculty of Computer Science, University of Murcia, Spain, in 2017. Since 1994, he has been a Professor and a Researcher with the Faculty of Computer Science, Autonomous University of Guerrero, Chilpancingo.