An MDE Approach to Generate Schemas for Object-document Mappers

Diego Sevilla Ruiz, Severino Feliciano Morales and Jesús García-Molina
Faculty of Computer Science, University of Murcia, Campus Espinardo, Murcia, Spain

Keywords: NoSQL Databases, NOSQL Data Engineering, Code Generation, MDE Solution, Object-document Mappers.

Abstract: Most NoSQL systems are schemaless. This lack of schema offers a greater flexibility than relational systems. However, this comes at the cost of losing benefits such as the static checking that assure that stored data conforms to the database schema. Instead, developers must be in charge of this task that is time-consuming and error-prone. Object-NoSQL mappers are emerging to alleviate this task and facilitate the development of NoSQL applications. These mappers allow the definition of schemas, which are used to assure that data are correctly manipulated. In this article, we present an MDE approach that automatically generates schemas and other artefacts for mappers. As proof of concept, we have considered Mongoose, which is a widely used mapper for MongoDB, but the solution is mapper-independent. Firstly, the schemas are inferred from stored data by using an approach defined in a previous work. Then, NoSQL schema models are input to a two-step model transformation chain that generates schemas, validators and updating procedures, among other Mongoose artefacts. An intermediate metamodel has been defined to ease the implementation of the transformations. This work shows how MDE techniques can be applied in the emerging "NoSQL Data Engineering" field.

1 INTRODUCTION

Interest in NoSQL (Not only SQL) databases has steadily grown over the last decade. Modern applications (e.g., social media, Internet of Things, mobile apps, and Big Data) have evidenced the limitations of traditional relational systems to meet the scalability and performance demands of these applications. A large number of companies have already embraced NoSQL databases, and the adoption will rise considerably in next years, as reported in (Dataversity, 2015; NoSQL-Market, 2016). The "nosql-database.org" website shows a list of 225 existing NoSQL systems. Actually, the NOSQL term refers to a varied set of data modelling paradigms aimed to manage semi-structured and unstructured data. The major NoSQL categories are: document, wide column and key-value stores, and graph-based databases. Except for graph databases, the paradigms aim to represent semi-structured data using reference and aggregation (Sadalage and Fowler, 2012). MongoDB (MongoDB, 2016) is the most widely used NoSQL system (NoSQL Ranking, 2016).

Most NoSQL systems are schemaless. This is a significant difference with respect to relational systems which need the definition of a schema to store data. Schemaless provides flexibility to manage data, for example, different versions of a data entity can be stored, and migrations are easier (Fowler, 2013). This characteristic is probably considered the most interesting and attractive of NoSQL systems. However, it should be noted that a data schema is often convenient, because database applications require knowing the data organization in order to manage data efficiently. In schemaless databases, the schema is in the minds of developers, and implicitly represented in the code and stored data. Therefore, two alternatives are emerging for NoSQL database applications: (i) combining the schemaless approach with mechanisms that guarantee a correct access to data (e.g. data validators) (Fowler, 2013), and (ii) using mappers which converts NoSQL data into objects of a programming language. Most current mappers are for document stores (Object-document mappers, ODMs), and Mongoose (Mongoose, 2016) is the most widely used ODM, created for MongoDB and Javascript. However, ODMs for other languages, such as Java and PHP, are also available.

Some recent works have drawn attention to the need for NoSQL tools just as they exist for relational databases. A Dataversity report (Dataversity, 2015) has remarked that data modeling will be a crucial activity for NoSQL databases. The authors identified three main capabilities to be offered by NoSQL modelling tools: model visualization, metadata management, and code generation. The emergence of a

Sevilla D., Feliciano S. and GarcÃ■a-Molina J.
An MDE Approach to Generate Schemas for Object-document Mappers.
DOI: 10.5220/0006279102200228
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 220-228
ISBN: 978-989-758-210-3

NoSQL data engineering, which would be a R&D area similar to relational data engineering, was suggested in (Sevilla Ruiz et al., 2015b).

Models and model transformations are key elements in data engineering. Data schemas are models, and operations on them can be implemented using model transformations. Therefore MDE techniques can be very useful to develop NoSQL tools which assist developers. In this article, we will present a MDE-based solution to automate the usage of ODMs when the database already exists. We have defined a chain of model transformations that generate some of the main artefacts involved in any ODM: schema definition, and validators, among others. This chain has as input schema models inferred by means of the strategy presented in (Sevilla Ruiz et al., 2015a). These schema models are transformed into models that conform to a *EntityDifferentiation* metamodel, defined in this work to facilitate the generation of code. Finally, the *EntityDifferentiation* models are used to generate mapper code. Our solution deals with the existence of more than one version for data entities, which imposes an additional complexity; conversely, it captures all the variability in the data base.

Therefore the contribution of our work is twofold. We present an application of the inference process defined in (Sevilla Ruiz et al., 2015a), and we show how MDE techniques are useful in the "NoSQL data engineering" area, more specifically in automating the usage of ODM mappers.

This article has been organized as follows. The following Section presents an overview of the proposal and introduces some basic concepts. Section 3 explains how schema models are transformed into *EntityDifferentiation* models. Section 4 describes the generation of Mongoose schemas in detail. Section 5 shows the DSL defined to create parameter models and describes the generation of more Mongoose artefacts. Finally, related work, conclusions, and further work are discussed.

2 OVERVIEW OF THE APPROACH

NoSQL Schemas for Aggregation-oriented Data Models. As indicated above, most NoSQL database systems do not require the definition of an explicit schema, but it is implicit in the data. In the case of NoSQL databases that are based on an aggregation-oriented data model, the schema is basically formed by a set of entities connected through two types of relationships: aggregation (a.k.a. part_of) and reference. Each entity has one or more attributes that are specified by their names and data types. These data types can be either a primitive data type (number, boolean, char, string) or some kind of collection (e.g. tuples in MongoDB) that stores primitive type values. A remarkable feature of these schemas is the possible existence of several versions of an entity, as the absence of an explicit schema allows the non-uniformity of the stored data. These versions can arise due to the need of having different variants of an entity or due to changes made during the evolution of data. Since this article focuses on MongoDB, we will use the term *document* to refer to the instances of an entity, and *field* to refer to properties (attributes, aggregates, and references).

Entities (and therefore entity versions) can be of three kinds: (i) *root* of a aggregation hierarchy; (ii) *aggregated* to a root or other aggregated entity, and (iii) *leaf* that are does not aggregate any entity.

An entity version is characterized by a set of fields that can be of three kinds: (i) *Common* to all entity versions (i.e. they are part of the all documents of the entity); (ii) *Shared* with other entity versions; and (iii) *specific* to particular entity version.

The Proposed Solution. In (Sevilla Ruiz et al., 2015a) we presented an MDE approach that implements a reverse engineering strategy to infer the implicit schema in NoSQL databases, and we outlined how the inferred model could be used to develop database utilities. Figure 1 shows the metamodel defined to represent NoSQL schemas. Compared with the related work (Wang et al., 2015; Klettke et al., 2015), the main novelty of our approach is discovering all the versions of the inferred entities and their relationships (i.e. composition and reference). Here, we shall show how the inferred schemas are useful to automate the usage of object-document mappers. We have considered Mongoose for MongoDB, but the solution presented is applicable to other object-document mappers.

We have defined a two-step model transformation chain which has as input an inferred NoSQL schema model, and generates Javascript code of Mongoose artefacts. The generated artefacts are mainly the database schema and validators. Figure 2 shows this generation process, which will be explained in detail in the next Sections. The first step of the chain is a model-to-model (m2m) transformation that represents the properties of entity versions in a way that facilitates the code generation. The model obtained conforms to the *EntityDifferentiation* metamodel which will be explained in Section 3. The second step is a model-to-text (m2t) transformation that generates Mongoose code from the model obtained

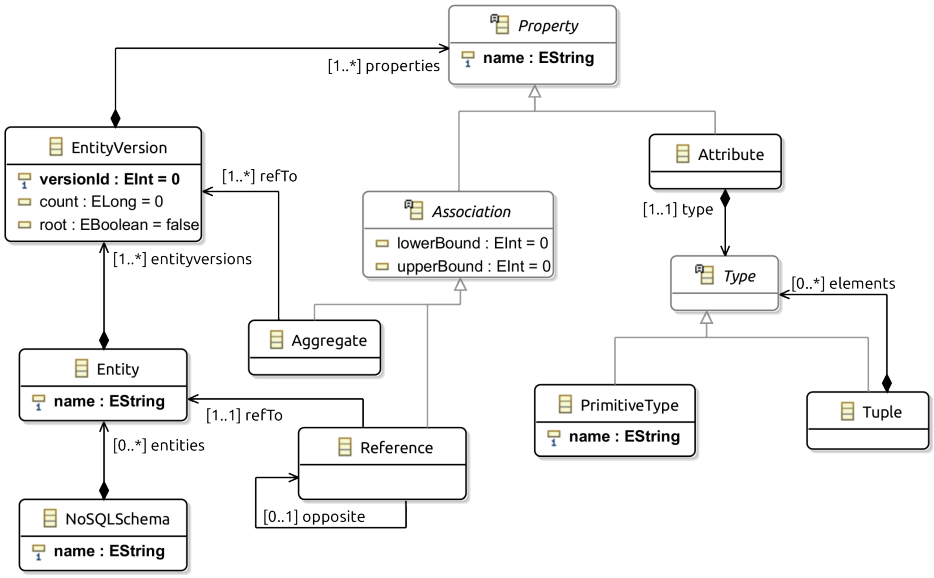

Figure 1: NoSQL Schema Metamodel.

in the previous step. The first transformation has been implemented in Java, while the model-to-text has been written in Xtend (Xtend, 2016). They can be downloaded from https://github.com/catedrasaes-umu/NoSQLDataEngineering.

Database Example. JSON is the format commonly used to store/read data into/from document databases. Figure 3 shows the JSON documents of the *Movie* database example used in this article to illustrate our process. This database includes two collections: *Movie* and *Director*. We have supposed that initially a *Movie* document had four mandatory fields: *title*, *year*, *director* and *genre*, and an optional field *prizes*. The field *criticisms* was added later. Each *Director* document has the mandatory fields *name* and *director-movies*, and the *actor-movies* optional field. The *Criticism* documents have the mandatory fields *content*, *journalist*, *media*, and the *url* optional field. The *Prize* documents have the fields *year*, *event*, and *name*.

According to the terminology introduced above, the schema is formed by two root entities (*Movie* and *Director*), and two aggregated entities which are leaf entities (*Prize* and *Criticism*). Three entity versions are identified for movies: *Movie1* (has prizes and criticisms), *Movie2* (has criticisms but not prizes) and *Movie4* (neither prizes nor criticisms). Note that *title*, *year*, *director*, and *genre* are common fields for *Movie* documents. We can observe two entity versions for *Director* and *Criticism* and a single version for *Prize*.

Figure 4 shows the schema inferred for the *Movie1* root version entity by using a notation similar to UML class diagrams. It is worth noting that we have implemented a m2t transformation to generate these diagrams that show the properties (attributes, aggregations, and references) of a root entity. This transformation traverses an input NoSQL schema model and generates the corresponding PlantUML code for all the entities involved. PlantUML (PlantUML, 2016) is a generator of UML diagrams built on GraphViz (GraphViz, 2016); it provides a simple and intuitive textual language to express UML diagrams. In the case of relationships, the possibility of a recursive composition have been considered by our m2t transformation.

Object-document Mappers: Mongoose. When database systems (e.g. relational systems) require the definition of a schema that specifies the structure of the stored data, a static checking assures that only data that fits the schema can be manipulated in application code, and mistakes made by developers in accessing data are statically spotted. However, schemaless databases entail developers to guarantee the correct access to data. This is an error-prone task, more so when the existence of several versions of each entity is possible. Therefore, some database utilities are emerging in order to alleviate this task. Object-NoSQL mappers are probably the more useful of these new tools. Like object-relational mappers, these mappers provide transparent persistence and perform a mapping between stored data and application objects. This requires that developers define a data schema, e.g. by using JSON (Mongoose,

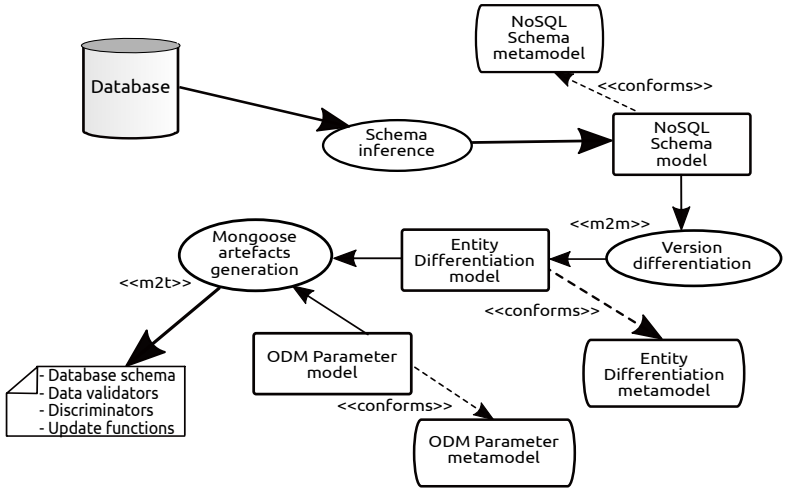

Figure 2: Overview of the Proposed MDE Solution.


```
[{
    "movie": [
        {
            "type": "movie",
            "title": "Citizen Kane",
            "year": 1941,
            "director_id": "123451",
            "genre": "Drama",
            "_id": "1",
            "prizes": [
                {
                    "year": 1941,
                    "event": "Oscar",
                    "names": [
                        "Best screenplay",
                        "Best Writing"
                    ]
                },
                {
                    "year": 1941,
                    "event": "NY Film Critics",
                    "names": ["Best Screenplay"]
                },
            ],
            "criticisms": [
                { "journalist": "R. Brody",
                  "media": "The New Yorker",
                  "color": "green"
                }
            ]
        },
        {
            "type": "movie",
            "title": "The Man Who Would Be King",
            "year": 1975,
            "director_id": "928672",
            "genre": "Adventures",
            "_id": "2"
        },
        {
            "_id": "4",
            "type": "movie",
            "title": "Truth",
            "year": 2014,
            "director_id": "345679",
            "genre": "Drama",
            "criticisms": [
                {
                    "journalist": "Jordi Costa",
                    "media": "El pais",
                    "url": "http://elpais.com/",
                    "color": "red"
                }
            ]
        }],
    "director": [
        {
            "name": "Orson Welles",
            "directed_movies": ["1", "5"],
            "actor_movies": ["1","5"],
            "type": "director",
            "_id": "123451"
        },
        {
            "type": "director",
            "directed_movies": ["4"],
            "name": "James Vanderbilt",
            "_id": "345679"
        },
        {
            "type": "director",
            "directed_movies": ["2"],
            "name": "John Huston",
            "_id": "928672"
        }
    ]
}]
```


Figure 3: JSON Documents in the Database Example.

2016), annotations (Doctrine, 2016), or a domain specific language (Mandango, 2016). Most of these mappers are Object-document mappers (ODM) because document-based databases (mainly MongoDB[1]) are the most widespread NoSQL systems. It is worth noting that developers have two alternatives in building NoSQL database applications. They can work in a schemaless way or use an ODM mapper, 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.

Mongoose is the *de facto* standard for defining schemas for MongoDB when writting Javascript applications. With Mongoose, database schemas can be defined as Javascript JSON objects, and then applications "can interact with MongoDB data in a structured

[1]Actually MongoDB uses BSON (Binary JSON), a variation of JSON with optimized binary storage and some added data types.

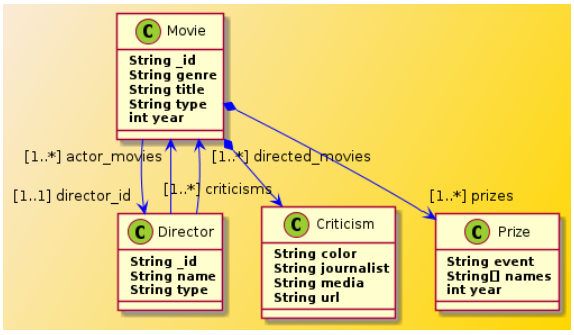

Figure 4: Version Schema for the *Movie1* Entity Version.

and repeatable way" (Holmes, 2013). Since JSON is a subset of the object literal notation of JavaScript, the Mongoose schemas are really Javascript code. We will show some examples of schema definitions in Section 4.

3 GENERATING ENTITY DIFFERENTIATION MODELS

Generating artefacts to manage the different entities and entity versions often have to differentiate between the properties common to all entity versions and other properties specific of a given entity version, as described in Section 2.

This may seem a trivial task, but some subtleties that will be addressed in this Section made it easier to take this process as separate of the artefact generation process, and also made the generation process itself easier. Thus, the *Entity Differentiation Metamodel* (shown in Figure 5) was created. Instances of this model are obtained via a model-to-model transformation from the *NoSQLSchema* metamodel. Note that the Entity Differentiation metamodel has references to the elements of the NoSQLSchema metamodel.

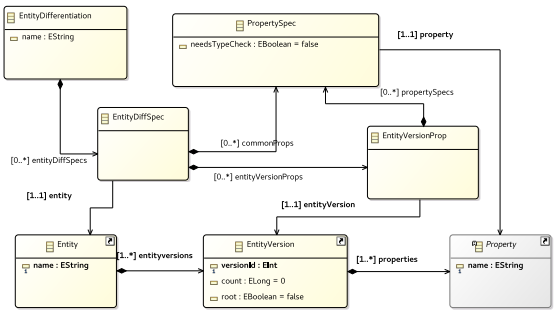

Figure 5: Entity Differentiation Metamodel.

The rationale behind this metamodel lies in two facts of the inference process:

1. The inference process is *complete*, that is, all documents in the database are considered, and their different entity versions recorded. Each document of the database belongs *exactly to one* entity version.
2. In order to differentiate between versions of a given entity, only properties specific of the given entity version need to be considered.

As seen in the *NoSQLSchema* metamodel, each entity has a set of entity versions. Each entity version has a set of properties, that in turn have a name and a type.

The transformation generates several set of interesting properties: For each Entity, an *EntityDiffSpec* model element is generated. This specification holds a set of common properties across all the versions, *commonProps*, and a set of differentiation set of properties for each version, *entityVersionProps*.

Common properties are those properties that are present (with the same name and type) in all the entity versions of a given entity. Conversely, the set of specific properties (*entityVersionProps*) for a given entity version is composed of the properties that are present in this entity version, but are not present in all other entity versions.

Properties in the Entity Differentiation metamodel are linked through the *PropertySpec* class. This class includes an attribute, *needsTypeCheck*, to signal when a discrimination cannot be made just using the *name* of the property. For instance, if two entity versions share a property with the same name but with different type, the fact that a document has a property with that name cannot be used to discriminate between these two entity versions: a type check must be performed. So, the *needsTypeCheck* attribute is set for the properties that appear in any other entity version with the same name *but with different type*.

An excerpt of the generated model for the database example can be seen in the Figure 6.

4 GENERATING MONGOOSE SCHEMAS

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. In our database example there would be two collections: *Movie* and *Director*. Therefore, a Mongoose schema should be defined for each of the two collections. Such schemas are the key element of Mongoose, and other mechanisms are defined based on them, such as validators, discriminators, or index

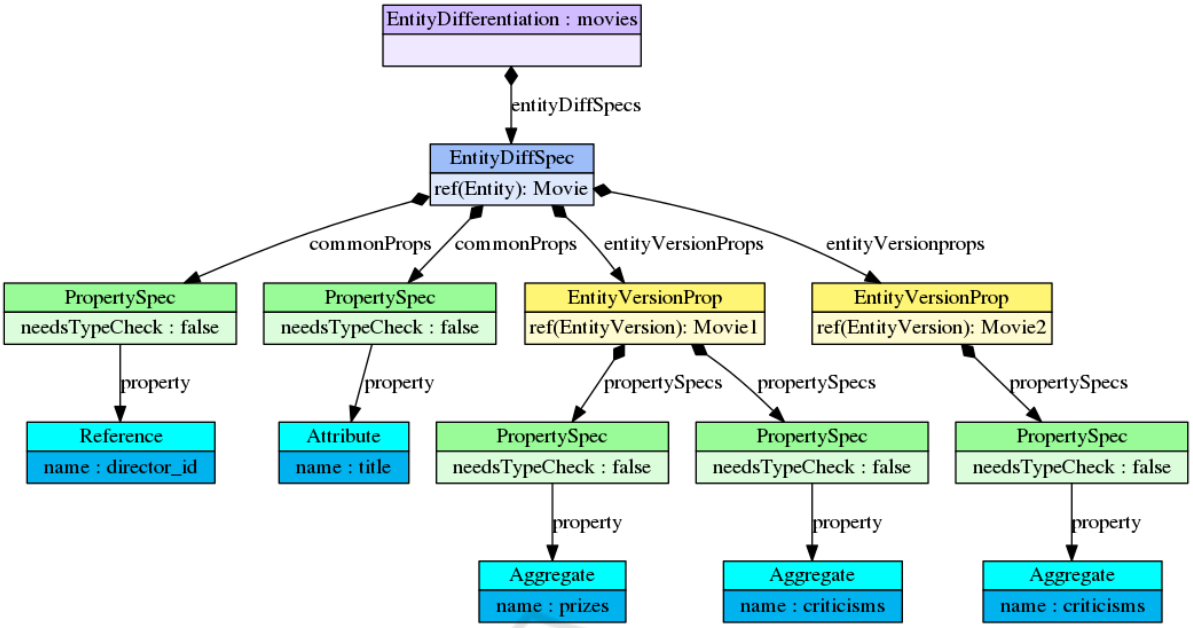

Figure 6: Excerpt of the EntityDifferentiation Model for the Example.

building. In this Section, we shall explain the process of generating schemas by means of the m2t transformation indicated in Section 2. Figure 7 shows the generated schemas for the example database.

Aggregations and references can be specified in Mongoose schemas. An aggregation is expressed as an nested document which defines the schema of the aggregated entity. A reference is expressed by means of the *ref* option in the definition of the type of an attribute. In addition to the type (i.e. a primitive type as `ObjectID`, `Number` or `String`), the *ref* option is used to indicate the name of a model of the referenced schema. In Figure 7, the *Movie* schema aggregates schemas for *Prize* (*prizes* field) and *Criticism* (*criticisms* field), and includes a reference to *Director* documents stored in the *Director* collection (*director_id* field). The *ref* option is used by Mongoose to ease the use of references in queries.

Note that we are assuming a scenario in which a company wants to use Mongoose for an existing database in order to take advantage of its facilities to check that data are correctly managed. Then, our tool would infer the database schema and generate code facilitating the use of the mapper. In generating schemas, we had to consider the existence of entity versions. For this, we have used the *require* validator that Mongoose provides to specify that a value for a particular field must always be given to save documents of a schema. Specifications of fields that are common to all the versions of an entity includes the validator *requires:true* to guarantee that any document stored of the entity will include these attributes. To work with a particular entity version, developers should add to the schema *require* restrictions for each of the specific fields of the version. In the schema of Figure 7, the *Movie* schema includes the *require* for the four common fields. Once the schema is defined, two *require* restrictions are added for the *prizes* and *criticisms* fields, which are specific to the *Movie1* entity version.

The transformation works as follows to generate database schemas: A schema is generated for each *EntityDiffSpec* connected to a root entity. For each of them, its common and entity version properties are added to the generated schema, but the *require* option is added only for common properties. For each aggregate property, an external declaration of type is added to improve the legibility of the schema. A model is created for schemas referenced from other schemas, which is needed to add the *ref* option in the declaration of the reference property. Note that this strategy will recursively operate because the existence of aggregate and reference properties.

5 GENERATING OTHER MONGOOSE ARTEFACTS

In addition to the schema definitions and the management of references between documents, Mongoose provides functionality to facilitate the development of MongoDB applications, such as validators, discriminators, and index specification. To automatically gen-

```
// Movie Schema

var criticismsSchema = {
   color: {type: String,
           enum:['green', 'yellow', 'red'],
           required: true},
   journalist:{type: String, unique:true,
               required: true},
   media:{type: String, required: true},
   url: String
}
var prizesSchema = {
   event:{type: String, required: true},
   names:{type: [String], required: true},
   year: {type: Number, required: true}
}
var movieSchema = new mongoose.Schema({
   title:{type:String, maxlength:40,
          unique:true, required:true},
   _id:  {type:String, index:true,
          required:true},
   year: {type:Number, index:true,
          required:true},
   type: {type:String, required:true},
   director_id: {type: String, required: true,
                 ref:'Director'},
   genre: {type:String,
        enum:['drama','comedy','children'],
        required:true},
   criticisms: {type:criticismsSchema},
   prizes: {type:prizesSchema}
},{collection:'Movie'});
```

```
// add required for Movie1 entity version

movieSchema.path('criticisms').required();
movieSchema.path('prizes').required();

var Movie = mongoose.model('Movie',movieSchema);


// add Director1 schema referenced by Movie1

var directorSchema = new mongoose.Schema({
    _id:        {type:String, index:true,
                 required:true},
    name:       {type:String, unique: true,
                 required:true},
    type:       {type:String, required:true},
    actor_movies:       {type:String,
                         ref:'Movie'},
    directed_movies: {type:String,
                      required:true,
                      ref:'Movie'}
},{collection:'Director'});


// add for Director 1 entity Version
directorSchema.path('actor_movies').required();

var Director = mongoose.model('Director',
      directorSchema);
```

Figure 7: Generated Mongoose Schema.

erate Mongoose code involved in all these mechanisms, we have created a domain-specific language (DSL) aimed to specify the information needed for such generation. This DSL is named *ODM Parameter Language* and it is independent of a concrete mapper technology. Figure 8 shows an example of specification for the entities of our database example. This DSL has been created with the Xtext (Xtext, 2016), and models are obtained by means of the parser generated by this tool. These DSL models are input to the m2t transformation that generates Mongoose artefacts from *EntityDifferentiation* models, as shown in Figure 2.

In Mongoose, the validation is defined at the schema level. Some frequently used validators are already built-in. The *require* and *unique* validators can be applied to any property. As explained in previous Section, we have used *require* to specify what properties are common to all the versions. The *unique* validator is used to express that all the documents of a collection must have a different value for a field of primitive type. Other examples of validators are *min* and *max* for Number fields, and *enum*, *minlength* and *maxlength* for String fields. Indexes are also defined at schema level, for instance an index can be specified with the *index* option or the *unique* validator (which also implies the creation of an index). Examples of use of these validators are shown in the schema in Figure 7. For instance an enumeration is defined for the *color* field of *Criticism* and the *title* field of *Movie* is *unique*. These validators have been generated from the information provided by the DSL specification shown in Figure 8.

Our schema inference mechanism cannot discover the decisions behind a version entity. As indicated above, version variation can be caused by different reasons, such as requirement changes, non-uniform data, or custom fields in entities. We have used the *required* validator to specify which fields are part of a particular entity version. However, Mongoose provides the discriminator mechanism to have collections of non-uniform data types. This mechanism would be more appropriate than the *require* option for non-uniform data. For instance, a *MovieTheaters* could register two kinds of movie theaters in our *Movie* database: single screen or multiplexed theaters. The *name*, *city*, *country* fields would be common, but the *roomNumber* field would be only part of multiplexed theaters. Figure 8 shows how to declare a discriminator for an entity, *MovieTheater* in the example. The generated code is shown in Figure 9.

Mongoose provides *update()* helper methods, but they do not apply validators, so the code to perform updating must be written following three steps (find-update-save). We also automate the generation of this code. For instance, in Figure 10 we show the code generated for updating the *genre* field of the *Movie* schema.

```
movies.odm
ODMParameters {
    mapper: Mongoose
    Entity Movie{
        validators{(genre : 'enum (drama,comedy, children)'),
                   (title : 'length < 40')
        }
        uniques {title}
        updates {genre, title}
        indexes {_id -> kind:Hash, year -> kind:Sorted}
    }

    Entity Director{
        uniques{name}
        updates{name}
        indexes {_id -> kind:Hash}
    }
    Entity Criticism{
        validators{(color : 'enum (green,red, yellow)')}
        uniques {journalist}
    }
    discriminator Entity MovieTheater{}
}
```

Figure 8: Specification Example with the ODM Parameter Language.

```
var options = {discriminatorKey: 'kind'};

var movieTheaterSchema = new mongoose.Schema(
    {name: String, city: String, country: String},
    options);
var MovieTheater1 = mongoose.model('MovieTheater1',
                        theaterSchema);

var MovieTheater2 = MovieTheater1.discriminator(
    'MovieTheater2',
    new mongoose.Schema({roomNumber: Number}, options));
```

Figure 9: *MovieTheater* schema using discriminator.

```
function update_genre(query, aGenre) {
   Movie.findOne (
      query,
      function (err, movie) {
         if (!err) {
            movie.genre = aGenre;
            movie.save(function (err, user) {
               console.log('Movie saved:', movie);
            });
         }
      }
   );
}
```

Figure 10: Code generated for updating the *genre* field of the *Movie* schema.

6 CONCLUSIONS AND RELATED WORK

The Dataversity report (Dataversity, 2015) evidenced the necessity of building tools to support the development of NoSQL applications. This demands a great effort of industry and academia in the emerging *NoSQL Data engineering* field. Tools with a functionality similar to those offered for relational databases are required, specially tools for (i) generating code, (ii) model visualization, and (iii) metadata management.

Research effort in this field has been very limited, and it has been mainly focused on the inference of schemas from stored data (Sevilla Ruiz et al., 2015a; Wang et al., 2015; Klettke et al., 2015). In fact, some existing tools for relational design are being extended to provide NoSQL inferred schema visualization (ER-Studio, 2016; DbSchema, 2016; ERWin, 2016). Moreover, NoSQL systems are offering tools for viewing, analyzing, and querying stored data, as Compass for MongoDB (Compass, 2016). Some kind of data analysis is also performed in (Klettke et al., 2015) where outliers are identified.

An MDE approach to generate code to manipulate GraphDB graph databases from conceptual schemas expressed in UML/OCL (Daniel et al., 2016). The authors have defined a GraphDB metamodel for graph databases and a mapping from UML class diagrams to this metamodel.

It is worth noting that the existence of version entities (and therefore versioned schemas) and relationships among entities has been only considered in (Sevilla Ruiz et al., 2015a) and (Wang et al., 2015). We can take advantage of this fact in building utilities from inferred schemas, as we have shown in (Hernández et al., 2016) and in the proposal presented in this article. We have recently presented a tool of visualization for document databases, which offers several diagrams and perspectives to show NoSQL versioned schemas (Hernández et al., 2016).

In our knowledge, the work presented here is the first approach for automating the use of ODM

mappers for existing databases. It is a technology-independent solution, and as a proof of concept it has been applied to Mongoose. We have been able to generate schemas and artefacts for the different functionality provided by Mongoose, such as validators, discriminators, and reference management. The solution presented has shown the usefulness of defining intermediate metamodels.

With regard to future work, we plan to consider more mappers for MongoDB and for other databases, and to build a generative architecture to automate the building of Mongoose-based MEAN applications for existing databases. Moreover, we are working on the definition of a DSL aimed to specify the necessary steps to convert one version of a database object to another version. This could be used in at least two ways: (i) A new application may require that all the recovered objects comply with a new version; (ii) In the case of a batch database migration, Map-Reduce jobs could be generated to transform old version objects into new versions.

ACKNOWLEDGEMENTS

This work has been partially supported by the Cátedra SAES of the University of Murcia (http://www.catedrasaes.org), a research lab sponsored by the SAES company (http://www.electronica-submarina.com/).

REFERENCES

Compass (2016). Mongodb Compass Web Page. https://www.mongodb.com/products/compass. Accessed: November 2016.

Daniel, G., Sunyé, G., and Cabot, J. (2016). *UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases*, pages 430–444. Springer International Publishing, Cham.

Dataversity (2015). Insights into NoSQL Modeling Report.

DbSchema (2016). DbSchema Web Page. http://www.dbschema.com. Accessed: November 2016.

Doctrine (2016). Doctrine Web Page. http://docs.doctrine-project.org/projects/doctrine-mongodb-odm/en/latest/. Accessed: November 2016.

ER-Studio (2016). ER-Studio Web Page. https://www.idera.com/er-studio-enterprise-data-modeling-and-architecture-tools. Accessed: November 2016.

ERWin (2016). CA ERwin Web Page. http://erwin.com/products/data-modeler. Accessed: November 2016.

Fowler, M. (2013). Schemaless Data Structures. http://martinfowler.com/articles/schemaless/.

GraphViz (2016). GraphViz Web Page. http://www.graphviz.org/. Accessed: November 2016.

Hernández, A., Sevilla Ruiz, D., and García-Molina, J. (2016). Visualization of Inferred Versioned Schemas from NoSQL Databases. SiriusCon 2016. http://www.slideshare.net/Obeo_corp/siriuscon2016-visualization-of-inferred-versioned-schemas-from-nosql-database

Holmes, S. (2013). *Mongoose for Application Development*. PACKT Publishing.

Klettke, M., Scherzinger, S., and Störl, U. (2015). Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In *BTW*.

Mandango (2016). Mandango Web Page. https://mandango.org/. Accessed: November 2016.

MongoDB (2016). MongoDB Web Page. https://www.mongodb.com/. Accessed: November 2016.

Mongoose (2016). Mongoose Web Page. http://mongoosejs.com. Accessed: November 2016.

NoSQL-Market (2016). NoSQL Market. https://www.alliedmarketresearch.com/NoSQL-market. Accessed: November 2016.

NoSQL Ranking (2016). NoSQL Ranking. http://db-engines.com/en/ranking. Accessed: November 2016.

PlantUML (2016). PlantUML Web Page. http://plantuml.com. Accessed: November 2016.

Sadalage, P. and Fowler, M. (2012). *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.

Sevilla Ruiz, D., Feliciano Morales, S., and García Molina, J. (2015a). Inferring Versioned Schemas from NoSQL Databases and its Applications. In *ER*, pages 467–480.

Sevilla Ruiz, D., Feliciano Morales, S., and García Molina, J. (2015b). Model Driven NoSQL Data Engineering. In *JISBD*, Santander.

Wang, L., Hassanzadeh, O., Zhang, S., Shi, J., Jiao, L., Zou, J., and Wang, C. (2015). Schema Management for Document Stores. In *VLDB Endowment*, volume 8.

Xtend (2016). Xtend Main Web Page. http://www.eclipse.org/xtend/. Accessed: May 2016.

Xtext (2016). Xtext Main Web Page. http://www.eclipse.org/Xtext/. Accessed: November 2016.