An Approach and DSL in support of Migration from Relational to NoSQL Databases

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

***Abstract*—In this paper we present a domain specific language, named *MongooseDSL*, used for modeling Mongoose validation schemas and documents. *MongooseDSL* language concepts are specified using Ecore, a widely used language for the specification of meta-models. Besides the meta-model we present the concrete syntax of the language alongside the examples of its usage. This *MongooseDSL* language is a part of the developed *NoSQLMigrator* tool. The tool can be used for migrating data from relational to NoSQL database systems.**

I. Introduction

Relational database management systems are preferred way of storing and managing data in the last few decades. Nowadays, due to the development of technologies, primarily the Internet, there was an increase in the number of different data sources. A lot of data is being generated every second and usually it is unstructured or semi-structured. Requirements for storing and processing such data are beyond the capabilities of traditional relational database management systems. In order to alleviate this problem, NoSQL database systems were introduced comprising a new approach to storing and processing large amounts of data [1]. These systems have built-in mechanisms for processing and analyzing large amounts of data, as well as the ability to save data in various formats, such is JSON. The absence of formally specified database schema in majority of NoSQL systems allows easier handling of variations of input data. This leads to the increase in number of users who are using NoSQL database systems in their applications. Accordingly, the need for reengineering legacy databases and migrating existing data to NoSQL systems is considered as an unavoidable step in such a process.

From the other side, model-driven approaches to software development increase the importance and power of models. Model is no longer just a bare image of a system, taken at the end of design process and used mostly for the communication and documentation purposes. Model-driven software engineering (MDSE) promotes the idea of abstracting implementation details by focusing on: models as first class entities and automated generation of models or code from other models. Each model is expressed by the concepts of a modeling language, which is in turn defined by a meta-model.

A reengineering process, and thus the whole migration processes, can benefit of using meta-models in almost every step. In this paper, we present a part of our research efforts focused on the database reengineering and the data migration process. We have developed a model-driven software tool named *NoSQLMigrator* that aims to fully automatize the migration process. *NoSQLMigrator* provides means to extract data from relational databases and then to validate and insert extracted data into a NoSQL database. Currently, *NoSQLMigrator* supports extracting data from most of the modern relational databases and inserting data into MongoDB database [2]. MongoDB has been chosen as it is one of the most used NoSQL databases. It is a document-oriented database, which stores data as a collection of documents serialized in JavaScript Object Notation (JSON). Migration process in *NoSQLMigrator* is implemented by means of a series of model-to-model (M2M) and model-to-text (M2T) transformations, so as to generate fully functional transaction programs and applications that are executed over a legacy relational database and new NoSQL database. One of the main reasons for the development of such a tool was to make developers' job easier, and particularly to free them from manual coding and testing. M2M transformations are based on meta-models to which source and target database models conform to. We denote such meta-models as database meta-models. We have developed Relational database schema meta-model in [3]. For the needs of developing the *NoSQLMigrator* tool, we have developed a domain specific language (DSL), named *MongooseDSL*. In this paper, we present both abstract (meta-model) and concrete syntaxes of this language.

MongooseDSL is a modeling language that can be used for modeling of Mongoose validation schemas and documents [4]. Mongoose is an object modeling tool that provides validation and data insert functionalities for MongoDB [5]. Mongoose validation schemas can be used for specifying constraints on data before it is inserted into MongoDB. For inserting the documents into MongoDB we use functions provided by the Mongoose tool. Meta-model of the *MongooseDSL* is also used as a database meta-model in the migration process.

Apart from the Introduction and Conclusion, the paper has four sections. In Section 2 we present the architecture of *NoSQLMigrator*. Abstract syntax of *MongooseDSL* is briefly described in Section 3, while in the fourth section we present the concrete textual syntax of the language. In the Section 5 we give an overview of the related work.

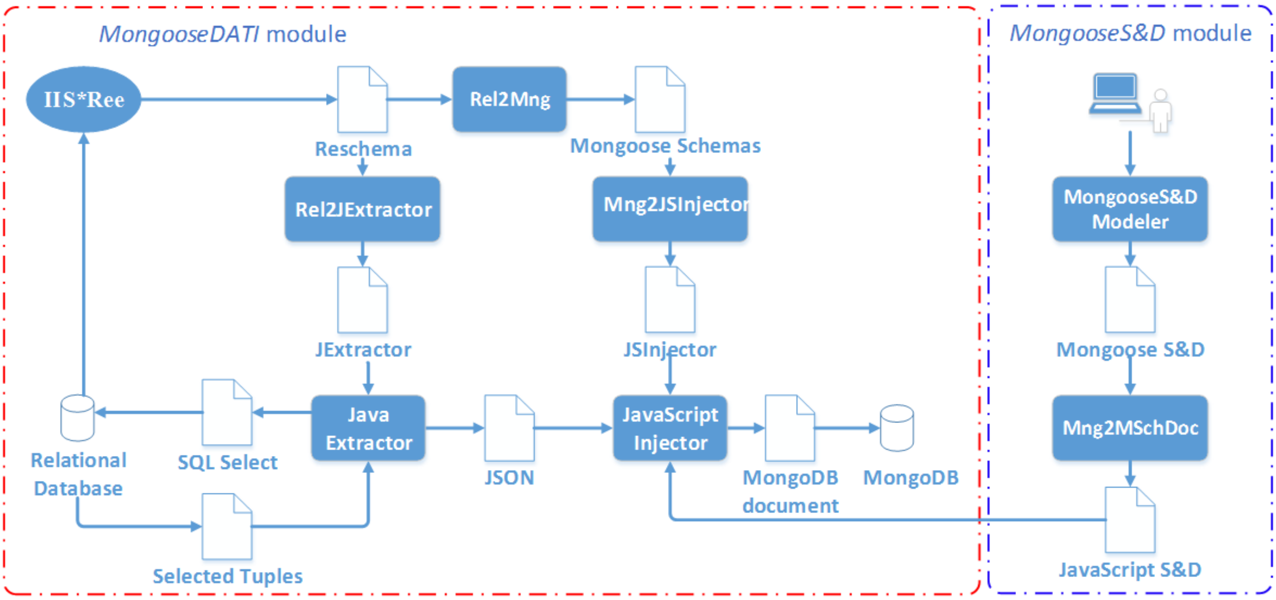

Figure 1. *NoSQLMigrator* architecture

II. The Architecture of NoSQLMigrator

In this section we present the architecture of the *NoSQLMigrator* tool. Its global picture is depicted in Figure 1. *NoSQLMigrator* comprises the following modules: *MongooseDATI* module and *MongooseS&D* module.

The *MongooseDATI* (Mongoose Data Aqusition, Transofrmation, and Injection) module allows user to perform the main part of migration process. *MongooseDATI* module comprises following components: *Rel2Mng, Rel2JExtractor, Mng2JSInjector, Java Extractor, and JavaScript Injector*.

Migration process is divided in four phases. During first phase reengineering of the relational database is done by using the IIS*Ree tool [6]. This tool provides a relational database schema specification according to relational database dictionary data. The specification conforms to meta-model based on standards, typical for the most relational database management systems (SQL:1999, SQL:2003, SQL:2011). In Figure 1. we present this specification as *Reschema* and the entire meta-model can be found in paper [3]. In the second phase of migration process, *Rel2Mng* component performs transformation from *Reschema* to Mongoose validation schemas specification. This specification is presented as *Mongoose Schemas*, and conforms to meta-model of developed *MongooseDSL* language.

The third phase of the migration process involves code generation using *Rel2JExtractor* and *Mng2JSInjector* components. *Rel2JExtractor* provides executable Java code based on *Reschema* specification. *Mng2JSInjector* provides executable JavaScript code based on *Mongoose Schemas* specification. In Figure 1. generated executable Java code is presented as *JExtractor* and generated executable JavaScript code as *JSInjector*. Code in *JExtractor* is used for data extraction from realational database schema. Code in *JSInjector* is used for validation of extracted data. Validation process is performed before data insertion into MongoDB database.

The last phase of data migration is generated code execution. The execution of generated Java code using *Java Extractor* component performs extraction of data from relational database and tranformation of extracted data to JSON documents. After transformation, JSON documents are sent to *JavaScript Injector* component. The execution of generated JavaScript code using *JavaScript Injector* compont results with acceptance of sent data, data validation acording to appropriate Mongoose validation schema and insertion of valid data to MongoDB database.

The *MongooseS&D* (Mongose Schema and Document) module enables user to specify Mongoose validation schemas and documents. *MongooseS&D* comprises *MongooseS&D Modeler* and *Mng2MSchDoc* components. This module provides generation of executable JavaScript code according to user specification. The user specifies Mongoose validation schemas and documents, using *MongooseDSL* concrete syntax within *MongooseS&D Modeler* component. Using *Mng2MSchDoc* component executable JavaScript code is generated based on user specification. Generated code comprises implementation of specified Mongoose validation schemas, documents and functions for insertion of valid data to MongoDB database.

III. *MongooseDSL* Abstract Syntax

In this section, we present the abstract syntax of the *MongooseDSL* language. The abstract syntax is implemented in a form of a meta-model that conforms to the Ecore meta-meta-model [7]. The meta-model is presented in Figure 2. In the rest of this section, we describe each of the *MongooseDSL* concepts with the corresponding meta-model class written in italics inside the parentheses.

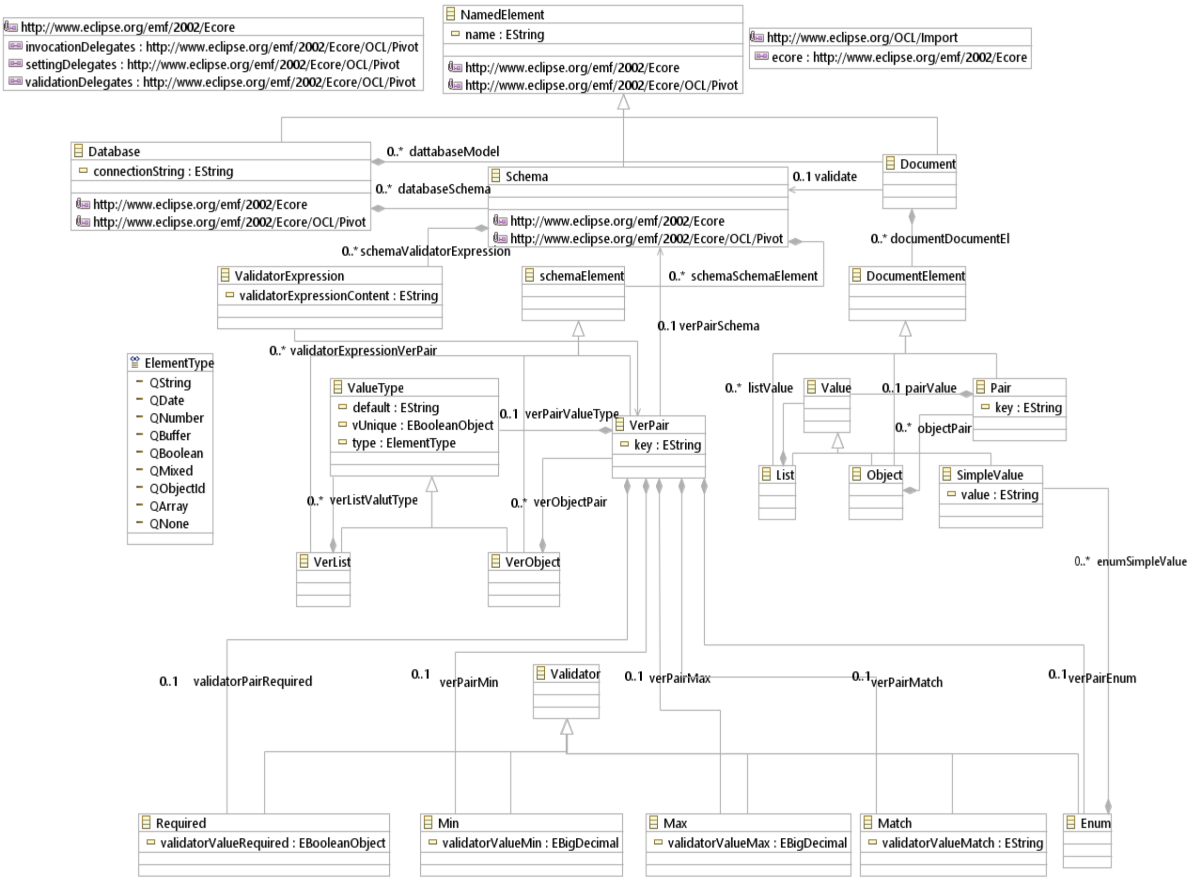

Figure 2. *MongooseDSL* meta-model.

The main language concept is the Mongoose-compliant database (*Database*) that comprises zero or more Mongoose validation schemas and zero or more Mongoose documents that are validated according to a defined schema.

The Mongoose document (*Document*) is a set of key-value pairs (*Pair*) called fields. For each field a key is defined in the *key* attribute of the *Pair* class, while values (*Value*) can be categorized as:

- Simple values (*SimpleValue*). Basic or atomic values that cannot be decomposed to more basic values such as integers, real numbers, etc.
- Complex values (*Object*). These are the values that represent structures comprising of other key-value pairs.
- List of values (*List*) that can contain any of the mentioned values.

Each Mongoose document is validated according to the associated validation schema. This is modeled with the *validate* reference.

The Mongoose validation schema (*Schema*) allows specification of validators for validating Mongoose documents before they are inserted into a database. By specifying such a schema, a user may define the structure, datatypes, and constraints on data. Each schema comprises zero or more schema fields (*VerPair*) that are in fact again key-value pairs. The key is modeled with the *key* attribute of *VerPair*, while the value (*VerType*) represents type constraints on values inserted into the document that is under validation. In *MongooseDSL*, we support creation of simple and complex type constraints. For simple type constraints, a user may choose one of the predefined types (*type* attribute of *VerType*) and a default value to be used if no other value is inserted (*default* attribute of *VerType*). Types are predefined in a form of the enumeration (*ElementType*) and cover the common datatypes found in modern database systems and programming languages. Further, it is possible to set the modeled value as unique (*vUnique* attribute of *VerType*). Complext type constraints may be comprised of other key-value pairs (*VerObject*), or they may represent lists of other simple or complex constraints (*VerList*). A schema field can be also a reference to other Mongoose validation schema allowing a user to decompose complex validation schemas into smaller ones. This is modeled with the *verPairSchema* reference.

Besides defining the type constraint, more detailed constraints on document values may be specified. These constraints are specified in a form of schema validators (*Validator*). *MongooseDSL* supports two types of validators: predefined and user-defined validators. The following predefined validators are supported:

- The validator for specifying if a Mongoose document field should be present in each document validated according the specified schema (*Required*). The attribute *validatorValueRequired* should be set to *true* if the field is required.
- The validator for specifying the minimum value of a document field (*Min*). The minimal

numerical value is specified in the attribute *validatorValueMin.*

- The validator for specifying the maximum value of a document field (*Max*). The maximal numerical value is specified in the attribute *validatorValueMax.*
- The validator for specifying a set of allowed values a Mongoose document field can have (*Enum*). Values are defined as simple values (*SimpleValue*) in the *enumSimpleValue* reference.
- The validator for specifying a regular expression that must be matched by a value of a Mongoose document field being validated (*Match*). The regular expression is defined in the *validatorValueMatch* attribute.

User-defined validators (*ValidatorExpression*) are specified using the validation functions whose body is defined in the *validatorExpressionContent* attribute. These functions implement the whole custom validation process.

IV. MongooseDSL concrete syntax

In this section we present *MongooseDSL* textual concrete syntax. The *MongooseDSL* concrete syntax represents the visual representation of the meta-model concepts. The instances of the *MongooseDSL* concepts and their attribute values are modeled by the production rules specified by concrete syntax.

```
VerPair returns VerPair:
    {VerPair}
    name=EString ":"
        ('[' verPairSchema=[Schema|EString]']')?
        (('(')?verPairValueType=ValueType)?
        ('{ ' subVerPair+=VerPair ( "," subVerPair+=VerPair)* '}' )?
        ( validatorPairRequired=Required(')')?)?
        ( verPairMin=Min(')')?)?
        ( verPairMax=Max(')')?)?
        ( verPairEnum=Enum(')')?)?
        ( verPairMatch=Match(')')?)?
    ;
```

Figure 3. *VerPair* production rule

By means of Eclipse plug-in named Xtext [8, 9], we have generated the concrete syntax of *MongooseDSL*. In Figure 3. we present the production rule for defining Mongoose fields in the validation schema. First the user needs to specify the field name (*name*). After the special character ":", the user can specify the name (*verPairSchema*) of another validation schema used for the field validation. The user may also specify the validation type (*verPairValueType*), or specify the value of Mongoose predefined validator (*validatorPairRequired, validatorPairMin, validatorParirMax, validatorPairEnum* and *validatorPairMatch).*

MongooseDSL does not require knowledge of programming language JavaScript. It is necessary for the user to be familiar with the *MongooseDSL* concepts and the language production rules. The number of *MongooseDSL* concepts is much smaller than number of JavaScript concepts. Executable JavaScript code is generated on the user specification defined by *MongooseDSL*. Generated code comprises complete specification of modeled Mongoose validation schemas and documents. It also contains functions for the operations of validation and insertion in MongoDB database.

In the presented *MongooseDSL* concrete syntax each meta-model concept is presented by its name. The special characters "{", "}", "(" and ")" are used for the representation of edges between the modeling concepts. First the user needs to specify the main concept *Database*, while the other concepts are defined within the main concept. The character "," is used as the delimiter between the concepts within the main concept. The references between the linked concepts are specified by the name of the connected concept within the specified concept. Each of the concepts are represented by the other color. This approach is used because of better overview of the model structure. The main concept *Database* is represented by red color. Schema and Document concepts are presented by green and blue color. Each of the concepts modeled by the Mongoose predefined validator are presented by pink color.

```
Schema<'userSchema'>{
    email:(
        type:QString
        unique:true
        required:true
    ),
    password:(
        type:QString
        unique:true
        required:true
        match:'[/a-zA-z0-9]/'
    ),
    gender:(
        type:QString
        required:true
        enum:{'male', 'female'}
    ),
    name:{
        first:(
            type:QString
            required:true
        ),
        last:(
            type:QString
            required:true
        )
    },
    addres:{
        country:(
            type:QString
            required:true
        ),
        city:(
            type:QString
            required:true
        ),
        zip:(
            type:QNumber
            required:true
            min:1.0
        )
    },
    phone:[
            {
                no:(
                    type:QNumber
                    required:false
                    match:'^(\\+\\d{1,2}\\s)?\\(?\\d{3}\\
                          ?[\\s.-]\\d{3}[\\s.-]\\d{4}$'
                )
            }
    ]
}
```

Figure 4. Mongoose validaiton schema model

In the following part of this section we present a fragment of a model specified using a textual syntax of *MongooseDSL.*

In Figure 4. we present an example of modeled Mongoose validation schema, used for validation of document comprising an internet portal users. This Mongoose validation schema is modeled by the Schema concept. The validation schema name is specified by the attribute *name* within the characters "<" and ">". The schema comprises a filed set, specified within special characters "{" and "}". First we model the filed *email.* The field name is modeled by specifying the value of *VerPair* concept attribute *key*. The field value is presented by usage *ValueType* concept. Within this concept the user specifies the field type setting the value of *type* attribute, choosing one of the predefined values of *ElementType* concept. The attribute *unique* is used to specify the field uniqueness at the level of collection in MongoDB database. In this field the user can model *required* Mongoose embedded validator, using the *Required* concept. Specifying the value *true* of *validatorValueRequired* attribute, the user defines the *email* field mandatory accorrding to the validation schema. *Password* field is modeled in the the similar way as the *email* field. The value of *Password* field is modeled by the *Match* concept that represents *match* Mongoose predefined validator. The Mongoose document field is validated by the value of *validatorValueMatch* attribute that stores the regular expression. The field is unique at the level of the collection that comprises Mongoose document. The field is also mandatory within this Mongoose document. The field *name* is modeled by the concepts *VerPair* and *VerObject*. The name of the field is specified by the value of the attribute *key*, within the instance of the *VerPair* concept. The value of this field is defined as complex value in the instance of *VerObject*

```
Document<'User_1423412',userSchema>{
    email:'user_1423412@mail.com',
    password:'user_1423412_pass',
    name:{
        first:'Jonh',
        last:'Sattler'
    },
    gender:'male',
    address:{
        country:'Louisiana, USA',
        city:'New Orleans',
        zip:'70129'
    },
    phone:[
        {
            no:'(504) 842-3000'
        },
        {
            no:'(504) 842-3012'
        }
    ]
}
```

Figure 5. Mongoose document model

concept. It comprises two fields *first* and *last*. Both of the fields are modeled as the instances of the *VerPair* and *ValueType* concepts.

The value of attribute *require* specifies that both of theese fields are mandatory. In the field *gender* we modeled field type, *required* Mongoose validator and *enum* Mongoose validator. *Enum* Mongoose validator is modeled by *Enum* concept. The instance of *Enum* concept comprise list of allowed values for an appropriate field of Mongoose document. The value of the field *phone* is modeled as the instance of *VerList* concept. The filed *phone* comprises the list of *VerObject* instances. Each *VerObject* instance contains a field modeled by the *VerPair* and *ValueType* concepts. Each of the fields comprises two Mongoose validators *required and match* are modeled, using *Required* i *Match* concepts.

Mongoose document presented in Figure 5. is modeled by the *Document* concept. The name of the document and and the name of Mongoose validaiton shema used for document validation, are presented within special characters “<” and “>”. The name of the document is specified by the value of attribute *name*. The edges of the document specification are presented by special characters “{” and “}”.

In the document *email* field is modeled as the instance of *Pair* and *Value* concepts. The field name is specified by the value of *Pair* concept attribute *key*. The value of the field is modeled by the attribute *value* in the *SimpleValue* concept instance. The edges of the modeled field are presented by special characters “(” and “)”. The field *password* is modeled in the same way as the field *email*. The field *name* represents complex field comprising two sub-fields. The field name is specified by the value of *Pair* concept attribute *key*. The value of the *name* field comprises two sub-fields *first* and *last*. *First* and *last* sub-fields are specified by the instance of *Object* concept. The name of the field is defined by the attribute *key* of the *Pair* concept. The value of the field is specified by the *value* attribute of the *SimpleValue* concept. The *first* and *last* sub-fields store information about first and last name of registered user. The field *gender* specifies information about gender of registered user. The field name is specified by the value of *Pair* concept attribute *key*. The value of the field is modeled by the attribute *value* in the *SimpleValue* concept instance. The field *phone* contains the list of telephone numbers of the registered user. The value of the field is modeled by the *List* concept. The instance of the *List* concept contains instances of the *Object* concept. Each instance of the *Object* concept represents a telephone number of registered user. Each field in the *Object* instance is modeled by the *Pair* and *SimpleValue* concepts.

V. Related Work

There are many papers describing migration data and services, but to the best of our konowledge there are no approaches to this problem by using *MDSD* (Model Driven Software Development) paradigm. Rocha et al. [10] prsent *NoSQLayer*, a framework capable to support conveniently migrating from relational (i.e., MySQL) to NoSQL DBMS (i.e., MongoDB). Lee et al. [11] describe how to migrate content management system (CMS) data from relational to NoSQL database to provide horizontal scaling and improve access performance. Zhao et al. [12] present a schema conversion model for transforming SQL database to NoSQL, providing high performance of join query with nesting relevant tables, and a graph transforming algorithm for containing all required content of join query in a table by offering correctly nested sequence. Zhao et al. [13] describe approach to migration of data from relational database do HBase NoSQL database and algorithm to find column name corresponding to attribute in relational database.

Many NoSQL database vendors, like MongoDB and Couchbase provide their own mechanisms and tools for data migration from relational to their own databases [14, 15].

VI. CONCLUSION

In this paper we presented a DSL for Mongoose schema and document specification, named *MongooseDSL*. Through our research we developed the *NoSQLMigrator* tool. It provides a data migration approach based upon the usage of *MongooseDSL*. Our intention was to provide automated mechanism for data migration from most of the relational databases to MongoDB. First of all we needed to create the *MongooseDSL* meta-model specified by Ecore that actually represents the abstract syntax of the language. Then, we created textual notations for *MongooseDSL*. Using textual notation user is able to specify Mongoose validation schemas and documents. *MongooseDSL* does not require knowledge of programming language JavaScript. The number of *MongooseDSL* concepts is much smaller than number of JavaScript concepts.

In our further research, we plan to extend *MongooseDSL* and the *NoSQLMigrator* tool to provide data migration to document-oriented databases of different vendors. We also plan to develop and embed into *MongooseDSL* some graphical notation. Also, another research direction would be to extend *MongooseDSL* with new concepts allowing more detailed specifications of data models. These new concepts should provide new constraint specifications. For example, formal specification of database referential integrity constraint is not implemented in our solution.

VII. REFERENCES

[1] Pramod J. Sadalage, Martin Fowler, “NoSQL Distilled: a brief guide to the emerging world polyglot persistence”, Crawfordsville, Indiana, pp 35-45, January 2013.

[2] “MongoDB” [Online], Available: https://www.mongodb.com/ [Accessed:01-02-2016]

[3] V. Dimitrieski, M. Čeliković S. Aleksić, S. Ristić, I. Luković, “Extended entity-relationship approach in a multi-paradigm information system modeling tool”, in: Proceedings of the 2014 Federated Conference on Computer Science and Infor Systems, Warsaw, Poland, pp.1611-1620, November 2014.

[4] Mernik, M., Heering, J., Sloane, M. A.: “When and how to develop domain-specific languages.” ACM Computing Surveys. pp. 316-344, December 2005.

[5] “Mongoose“.[Online], Available: http://mongoosejs.com/ [Accessed:01-Jan-2016]

[6] S. Aleksić, “Methods of database schema transformations in support of the information system reengineering process”, Ph.D. thesis, University of Novi Sad (2013).

[7] “Meta-Object Facility” [Online] Available: http://www.omg.org/mof/ [Accessed:01-Jan-2016]

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

[9] “Eclipse” [Online], Available: http://projects.eclipse.org/projects/modeling. [Accessed: 02-Jan-2014].

[10] L. Rocha, F. vale, E. Cirilo, D. Barbosa F. Murao, “A Framework for Migration Relational Datasets to NoSQL”, International Conference On Computational Science, Reykjavik, Iceland, pp. 2593–2602, June 2015.

[11] Lee Chao-Hsien, Zheng Yu-Lin, “SQL-to-NoSQL Schema Demoralization and Migration: A Study on Content Management Systems”, Systems, Man, and Cybernetics (SMC), IEEE International Conference, Konwloon Tong, Hong Kong, pp. 2022-2026, October 2015.

[12] G. Zhao, Q. Lin, L. Li, Z. Li, “Schema Conversation Model of SQL Database to NoSQL”, P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference , Guangdong, pp. 355-362, November 2014.

[13] G. Zhao, L. Li, Z. Li, Q. Lin, “Multiple Nested Schema of HBase for Migration from SQL”, P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference, Guangdong, pp 338-343, November 2014.

[14] “RDBMS to MongoDB Migration Guide” [Online], Available: https://s3.amazonaws.com/info-mongodb-com/RDBMStoMongoDBMigration.pdf. [Accessed: 02-Jan-2014].

[15] “Moving from Relational to NoSQL: How to Get Started” [Online], Available: http://info.couchbase.com/rs/302-GJY-034/images/Moving_from_Relational_to_NoSQL_Couchbase_2016.pdf. [Accessed: 02-Jan-2014].